K210
FreeRTOS SDK 编程指南

KENDRYTE
勘智

关于本手册

本文档为用户提供基于 FreeRTOS SDK 开发时的编程指南.

对应 SDK 版本

Kendryte FreeRTOS SDK v0.4.0 (*9c7b0e0d23e46e87a2bfd4dd86d1a1f0d3c899e9*)

发布说明

日期	版本	发布说明
2018-10-12	V0.1.0	初始版本

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

FreeRTOS 扩展

1.1 概述

FreeRTOS 是一个轻量级的实时操作系统。本 SDK 在其基础上新增了一些适用于 K210 的功能。

1.2 功能描述

FreeRTOS 扩展模块具有以下功能：

- 获取当前任务所在的逻辑处理器 Id
- 在指定逻辑处理器创建任务

K210 包含 2 个逻辑处理器，Id 分别为 0 和 1。

1.3 API 参考

对应的头文件 task.h
为用户提供以下接口：

- uxTaskGetProcessorId
- xTaskCreateAtProcessor

1.3.1 uxTaskGetProcessorId

1.3.1.1 描述

获取当前逻辑处理器 Id。

1.3.1.2 函数原型

```
UBaseType_t uxTaskGetProcessorId(void);
```

1.3.1.3 返回值

当前逻辑处理器 Id。

1.3.2 xTaskCreateAtProcessor

1.3.2.1 描述

在指定逻辑处理器创建任务。

1.3.2.2 函数原型

```
BaseType_t xTaskCreateAtProcessor(UBaseType_t uxProcessor, TaskFunction_t pxTaskCode,
    const char * const pcName, const configSTACK_DEPTH_TYPE usStackDepth, void * const
     pvParameters, UBaseType_t uxPriority, TaskHandle_t * const pxCreatedTask);
```

1.3.2.3 参数

参数名称	描述	输入输出
uxProcessor	逻辑处理器 Id	输入
pxTaskCode	任务入口点	输入
pcName	任务名称	输入
usStackDepth	栈空间	输入
pvParameters	参数	输入
uxPriority	优先级	输入
pxCreatedTask	创建的任务句柄	输入

1.3.2.4 返回值

返回值	描述
pdPASS	成功
其他	失败

第 2 章

设备列表

路径	类型	备注
/dev/uart1	UART	
/dev/uart2	UART	
/dev/uart3	UART	
/dev/gpio0	GPIO	高速 GPIO
/dev/gpio1	GPIO	
/dev/i2c0	I²C	
/dev/i2c1	I²C	
/dev/i2c2	I²C	
/dev/i2s0	I2S	
/dev/i2s1	I2S	
/dev/i2s2	I2S	
/dev/spi0	SPI	
/dev/spi1	SPI	
/dev/spi3	SPI	
/dev/sccb0	SCCB	
/dev/dvp0	DVP	
/dev/fft0	FFT	
/dev/aes0	AES	
/dev/sha256	SHA256	
/dev/timer0	TIMER	不可与 /dev/pwm0 同时使用
/dev/timer1	TIMER	不可与 /dev/pwm0 同时使用
/dev/timer2	TIMER	不可与 /dev/pwm0 同时使用

路径	类型	备注
/dev/timer3	TIMER	不可与 /dev/pwm0 同时使用
/dev/timer4	TIMER	不可与 /dev/pwm1 同时使用
/dev/timer5	TIMER	不可与 /dev/pwm1 同时使用
/dev/timer6	TIMER	不可与 /dev/pwm1 同时使用
/dev/timer7	TIMER	不可与 /dev/pwm1 同时使用
/dev/timer8	TIMER	不可与 /dev/pwm2 同时使用
/dev/timer9	TIMER	不可与 /dev/pwm2 同时使用
/dev/timer10	TIMER	不可与 /dev/pwm2 同时使用
/dev/timer11	TIMER	不可与 /dev/pwm2 同时使用
/dev/pwm0	PWM	不可与 /dev/timer[0-3] 同时使用
/dev/pwm1	PWM	不可与 /dev/timer[4-7] 同时使用
/dev/pwm2	PWM	不可与 /dev/timer[8-11] 同时使用
/dev/wdt0	WDT	
/dev/wdt1	WDT	
/dev/rtc0	RTC	

第 3 章

管脚配置

3.1 概述

管脚配置包含 FPIOA 和电源域配置等。

3.2 功能描述

- 支持 IO 的可编程功能选择
- 配置电源域

3.3 数据类型

对应的头文件 pin_cfg.h

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_cfg_item_t：FPIOA 管脚配置。
- fpioa_cfg_t：FPIOA 配置。
- sysctl_power_bank_t：电源域编号。
- sysctl_io_power_mode_t：IO 输出电压值。
- power_bank_item_t：单个电源域配置。
- power_bank_cfg_t：电源域配置。
- pin_cfg_t：管脚配置。

3.3.1　fpioa_function_t

3.3.1.1　描述

管脚的功能编号。

3.3.1.2　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 Receiver */
    FUNC_UARTHS_TX        = 19,     /*!< UART High speed Transmitter */
    FUNC_CLK_IN1          = 20,     /*!< Clock Input 1 */
    FUNC_CLK_IN2          = 21,     /*!< Clock Input 2 */
    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 Receiver */
    FUNC_UART1_TX         = 65,     /*!< UART1 Transmitter */
    FUNC_UART2_RX         = 66,     /*!< UART2 Receiver */
    FUNC_UART2_TX         = 67,     /*!< UART2 Transmitter */
    FUNC_UART3_RX         = 68,     /*!< UART3 Receiver */
    FUNC_UART3_TX         = 69,     /*!< UART3 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_PWND        = 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 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 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 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;
```

3.3.1.3　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
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

成员名称	描述
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
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

成员名称	描述
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
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

成员名称	描述
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）
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

成员名称	描述
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
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

成员名称	描述
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
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

成员名称	描述
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
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

成员名称	描述
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
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

3.3.2 fpioa_cfg_item_t

3.3.2.1 描述

FPIOA 管脚配置。

3.3.2.2 定义

```
typedef struct _fpioa_cfg_item
{
    int number;
    fpioa_function_t function;
} fpioa_cfg_item_t;
```

3.3.2.3 成员

成员名称	描述
number	管脚编号

成员名称	描述
function	功能编号

3.3.3　fpioa_cfg_t

3.3.3.1　描述

FPIOA 配置。

3.3.3.2　定义

```
typedef struct _fpioa_cfg
{
    uint32_t version;
    uint32_t functions_count;
    fpioa_cfg_item_t functions[];
} fpioa_cfg_t;
```

3.3.3.3　成员

成员名称	描述
version	配置版本，必须设为 FPIOA_CFG_VERSION
functions_count	功能配置数量
functions	功能配置列表

3.3.4　sysctl_power_bank_t

3.3.4.1　描述

电源域编号。

3.3.4.2　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;
```

3.3.4.3　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 0，控制 IO6-IO11
SYSCTL_POWER_BANK2	电源域 0，控制 IO12-IO17
SYSCTL_POWER_BANK3	电源域 0，控制 IO18-IO23
SYSCTL_POWER_BANK4	电源域 0，控制 IO24-IO29
SYSCTL_POWER_BANK5	电源域 0，控制 IO30-IO35
SYSCTL_POWER_BANK6	电源域 0，控制 IO36-IO41
SYSCTL_POWER_BANK7	电源域 0，控制 IO42-IO47

3.3.5　sysctl_io_power_mode_t

3.3.5.1　描述

IO 输出电压值。

3.3.5.2　定义

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

3.3.5.3　成员

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

3.3.6　power_bank_item_t

3.3.6.1　描述

单个电源域配置。

3.3.6.2　定义

```
typedef struct _power_bank_item
{
    sysctl_power_bank_t power_bank;
    sysctl_io_power_mode_t io_power_mode;
} power_bank_item_t;
```

3.3.6.3　成员

成员名称	描述
power_bank	电源域编号
io*power*mode	IO 输出电压值

3.3.7　power_bank_cfg_t

3.3.7.1　描述

电源域配置。

3.3.7.2　定义

```
typedef struct _power_bank_cfg
{
    uint32_t version;
    uint32_t power_banks_count;
    power_bank_item_t power_banks[];
} power_bank_cfg_t;
```

3.3.7.3　成员

成员名称	描述
version	配置版本，必须设为 FPIOA_CFG_VERSION
power*banks*count	电源域配置数量
power_banks	电源域配置列表

3.3.8　pin_cfg_t

3.3.8.1　描述

管脚配置。

3.3.8.2　定义

```
typedef struct _pin_cfg
{
    uint32_t version;
    bool set_spi0_dvp_data;
} pin_cfg_t;
```

3.3.8.3　成员

成员名称	描述
version	配置版本，必须设为 FPIOA_CFG_VERSION
set*spi0*dvp_data	是否设置 SPI*0 D0-D7 与 DVP* D0-D7 为 SPI0 数据输出与 DVP 数据输入

3.3.9　举例

```
/* 配置 IO6、IO7 的功能分别为 GPIOHS0 和 GPIOHS1 */
const fpioa_cfg_t g_fpioa_cfg =
{
    .version = FPIOA_CFG_VERSION,
    .functions_count = 2,
    .functions =
    {
        { .number = 6, .function = FUNC_GPIOHS0 },
        { .number = 7, .function = FUNC_GPIOHS1 }
    }
};
```

第 4 章

系统控制

4.1 概述

系统控制模块提供对操作系统的配置功能。

4.2 功能描述

系统控制模块具有以下功能：

- 设置 CPU 频率
- 安装自定义驱动

4.3 API 参考

对应的头文件 `hal.h`
为用户提供以下接口：

- `system_set_cpu_frequency`
- `system_install_custom_driver`

4.3.1 system_set_cpu_frequency

4.3.1.1 描述

设置 CPU 频率。

4.3.1.2　函数原型

```
uint32_t system_set_cpu_frequency(uint32_t frequency);
```

4.3.1.3　参数

参数名称	描述	输入输出
frequency	要设置的频率（Hz）	输入

4.3.1.4　返回值

设置后的实际频率（Hz）。

4.3.2　system_install_custom_driver

4.3.2.1　描述

安装自定义驱动。

4.3.2.2　函数原型

```
void system_install_custom_driver(const char *name, const custom_driver_t *driver);
```

4.3.2.3　参数

参数名称	描述	输入输出
name	指定访问该设备的路径	输入
driver	自定义驱动实现	输入

4.3.2.4　返回值

无。

4.3.3　举例

```
/* 设置 CPU 频率为 400MHz */
system_set_cpu_frequency(400000000);
```

4.4　数据类型

相关数据类型、数据结构定义如下：

- driver_base_t：驱动实现基类。
- custom_driver_t：自定义驱动实现。

4.4.1　driver_base_t

4.4.1.1　描述

驱动实现基类。

4.4.1.2　定义

```
typedef struct _driver_base
{
    void *userdata;
    void (*install)(void *userdata);
    int (*open)(void *userdata);
    void (*close)(void *userdata);
} driver_base_t;
```

4.4.1.3　成员

成员名称	描述
userdata	用户数据
install	安装时被调用
open	打开时被调用
close	关闭时被调用

4.4.2　custom_driver_t

4.4.2.1　描述

自定义驱动实现。

4.4.2.2　定义

```
typedef struct _custom_driver
```

```
{
    driver_base_t base;
    int (*io_control)(uint32_t control_code, const uint8_t *write_buffer, size_t
        write_len, uint8_t *read_buffer, size_t read_len, void *userdata);
} custom_driver_t;
```

4.4.2.3　成员

成员名称	描述
base	驱动实现基类
io_control	收到控制信息时被调用

第 5 章

可编程中断控制器（PIC）

5.1 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。

5.2 功能描述

PIC 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

5.3 API 参考

对应的头文件 `hal.h`
为用户提供以下接口：

- `pic_set_irq_enable`
- `pic_set_irq_handler`
- `pic_set_irq_priority`

5.3.1　pic_set_irq_enable

5.3.1.1　描述

设置 IRQ 是否启用。

5.3.1.2　函数原型

```
void pic_set_irq_enable(uint32_t irq, bool enable);
```

5.3.1.3　参数

参数名称	描述	输入输出
irq	IRQ 编号	输入
enable	是否启用	输入

5.3.1.4　返回值

无。

5.3.2　pic_set_irq_handler

5.3.2.1　描述

设置 IRQ 处理程序。

5.3.2.2　函数原型

```
void pic_set_irq_handler(uint32_t irq, pic_irq_handler_t handler, void *userdata);
```

5.3.2.3　参数

参数名称	描述	输入输出
irq	IRQ 编号	输入
handler	处理程序	输入
userdata	处理程序用户数据	输入

5.3.2.4　返回值

无。

5.3.3　pic_set_irq_priority

5.3.3.1　描述

设置 IRQ 优先级。

5.3.3.2　函数原型

```
void pic_set_irq_priority(uint32_t irq, uint32_t priority);
```

5.3.3.3　参数

参数名称	描述	输入输出
irq	IRQ 编号	输入
priority	优先级	输入

5.3.3.4　返回值

无。

5.4　数据类型

相关数据类型、数据结构定义如下：

- pic_irq_handler_t：IRQ 处理程序。

5.4.1　pic_irq_handler_t

5.4.1.1　描述

IRQ 处理程序。

5.4.1.2　定义

```
typedef void (*pic_irq_handler_t)(void *userdata);
```

5.4.1.3　参数

参数名称	描述	输入输出
userdata	用户数据	输入

第 6 章

直接存储访问（DMA）

6.1 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。

6.2 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

6.3 API 参考

对应的头文件 `hal.h`
为用户提供以下接口：

- `dma_open_free`
- `dma_close`
- `dma_set_request_source`
- `dma_transmit_async`

- dma_transmit
- dma_loop_async

6.3.1　dma_open_free

6.3.1.1　描述

打开一个可用的 DMA 设备。

6.3.1.2　函数原型

```
handle_t dma_open_free();
```

6.3.1.3　返回值

DMA 设备句柄。

6.3.2　dma_close

6.3.2.1　描述

关闭 DMA 设备。

6.3.2.2　函数原型

```
void dma_close(handle_t file);
```

6.3.2.3　参数

参数名称	描述	输入输出
file	DMA 设备句柄	输入

6.3.2.4　返回值

无。

6.3.3　dma_set_request_source

6.3.3.1　描述

设置 DMA 请求源。

6.3.3.2　函数原型

```
void dma_set_request_source(handle_t file, uint32_t request);
```

6.3.3.3　参数

参数名称	描述	输入输出
file	DMA 设备句柄	输入
request	请求源编号	输入

6.3.3.4　返回值

无。

6.3.4　dma_transmit_async

6.3.4.1　描述

进行 DMA 异步传输。

6.3.4.2　函数原型

```
void dma_transmit_async(handle_t file, const volatile void *src, volatile void *dest,
    int src_inc, int dest_inc, size_t element_size, size_t count, size_t burst_size,
    SemaphoreHandle_t completion_event);
```

6.3.4.3　参数

参数名称	描述	输入输出
file	DMA 设备句柄	输入
src	源地址	输入
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
element_size	元素大小（字节）	输入
count	元素数量	输入
burst_size	突发传输数量	输入

参数名称	描述	输入输出
completion_event	传输完成事件	输入

6.3.4.4　返回值

无。

6.3.5　dma_transmit

6.3.5.1　描述

进行 DMA 同步传输。

6.3.5.2　函数原型

```
void dma_transmit(handle_t file, const volatile void *src, volatile void *dest, int
    src_inc, int dest_inc, size_t element_size, size_t count, size_t burst_size);
```

6.3.5.3　参数

参数名称	描述	输入输出
file	DMA 设备句柄	输入
src	源地址	输入
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
element_size	元素大小（字节）	输入
count	元素数量	输入
burst_size	突发传输数量	输入

6.3.5.4　返回值

无。

6.3.6　dma_loop_async

6.3.6.1　描述

进行 DMA 异步循环传输。

6.3.6.2　函数原型

```
void dma_loop_async(handle_t file, const volatile void **srcs, size_t src_num, volatile
     void **dests, size_t dest_num, int src_inc, int dest_inc, size_t element_size,
    size_t count, size_t burst_size, dma_stage_completion_handler_t
    stage_completion_handler, void *stage_completion_handler_data, SemaphoreHandle_t
    completion_event, int *stop_signal);
```

6.3.6.3　参数

参数名称	描述	输入输出
file	DMA 设备句柄	输入
srcs	源地址列表	输入
src_num	源地址数量	输入
dests	目标地址列表	输出
dest_num	目标地址数量	输入
src_inc	源地址是否自增	输入
dest_inc	目标地址是否自增	输入
element_size	元素大小（字节）	输入
count	元素数量	输入
burst_size	突发传输数量	输入
stage_completion_handler	阶段完成处理程序	输入
stage_completion_handler_data	阶段完成处理程序用户数据	输入
completion_event	传输完成事件	输入
stop_signal	停止信号	输入

注： 阶段完成是指单次源到目标 count 个元素的传输完成。

6.3.6.4　返回值

无。

6.3.7　举例

```
int src[256] = { [0 ... 255] = 1 };
int dest[256];
handle_t dma = dma_open_free();
```

```
dma_transmit(dma, src, dest, true, true, sizeof(int), 256, 4);
assert(dest[0] == src[0]);
dma_close(dma);
```

6.4　数据类型

相关数据类型、数据结构定义如下：

- dma_stage_completion_handler_t：DMA 阶段完成处理程序。

6.4.1　dma_stage_completion_handler_t

6.4.1.1　描述

DMA 阶段完成处理程序。

6.4.1.2　定义

```
typedef void (*dma_stage_completion_handler_t)(void *userdata);
```

6.4.1.3　参数

参数名称	描述	输入输出
userdata	用户数据	输入

第 7 章

标准 IO

7.1 概述

标准 IO 模块是访问外设的基本接口。

7.2 功能描述

标准 IO 模块具有以下功能：

- 根据路径寻找外设
- 统一的读写和控制接口

7.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- io_open
- io_close
- io_read
- io_write
- io_control

7.3.1　io_open

7.3.1.1　描述

打开一个设备。

7.3.1.2　函数原型

```
handle_t io_open(const char *name);
```

7.3.1.3　参数

参数名称	描述	输入输出
name	设备路径	输入

7.3.1.4　返回值

返回值	描述
0	失败
其他	设备句柄

7.3.2　io_close

7.3.2.1　描述

关闭一个设备。

7.3.2.2　函数原型

```
int io_close(handle_t file);
```

7.3.2.3　参数

参数名称	描述	输入输出
file	设备句柄	输入

7.3.2.4　返回值

返回值	描述
0	成功
其他	失败

7.3.3　io_read

7.3.3.1　描述

从设备读取。

7.3.3.2　函数原型

```
int io_read(handle_t file, uint8_t *buffer, size_t len);
```

7.3.3.3　参数

参数名称	描述	输入输出
file	设备句柄	输入
buffer	目标缓冲区	输出
len	最多读取的字节数	输入

7.3.3.4　返回值

实际读取的字节数。

7.3.4　io_write

7.3.4.1　描述

向设备写入。

7.3.4.2　函数原型

```
int io_write(handle_t file, const uint8_t *buffer, size_t len);
```

7.3.4.3　参数

参数名称	描述	输入输出
file	设备句柄	输入
buffer	源缓冲区	输入
len	要写入的字节数	输入

7.3.4.4　返回值

返回值	描述
len	成功
其他	失败

7.3.5　io_control

7.3.5.1　描述

向设备发送控制信息。

7.3.5.2　函数原型

```
int io_control(handle_t file, uint32_t control_code, const uint8_t *write_buffer,
    size_t write_len, uint8_t *read_buffer, size_t read_len);
```

7.3.5.3　参数

参数名称	描述	输入输出
file	设备句柄	输入
control_code	控制码	输入
write_buffer	源缓冲区	输入
write_len	要写入的字节数	输入
read_buffer	目标缓冲区	输出
read_len	最多读取的字节数	输入

7.3.5.4　返回值

实际读取的字节数。

7.3.6　举例

```
handle_t uart = io_open("/dev/uart1");
io_write(uart, "hello\n", 6);
io_close(uart);
```

第 8 章

通用异步收发传输器（UART）

8.1 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。

8.2 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

8.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- uart_config

8.3.1 uart_config

8.3.1.1 描述

配置 UART 设备。

8.3.1.2　函数原型

```
void uart_config(handle_t file, uint32_t baud_rate, uint32_t databits, uart_stopbits_t
    stopbits, uart_parity_t parity);
```

8.3.1.3　参数

参数名称	描述	输入输出
file	UART 设备句柄	输入
baud_rate	波特率	输入
databits	数据位（5-8）	输入
stopbits	停止位	输入
parity	校验位	输入

8.3.1.4　返回值

无。

8.3.2　举例

```
handle_t uart = io_open("/dev/uart1");

uint8_t b = 1;
/* 写入 1 个字节 */
io_write(uart, &b, 1);
/* 读取 1 个字节 */
while (io_read(uart, &b, 1) != 1);
```

8.4　数据类型

相关数据类型、数据结构定义如下：

- uart_stopbits_t：UART 停止位。
- uart_parity_t：UART 校验位。

8.4.1　uart_stopbits_t

8.4.1.1　描述

UART 停止位。

8.4.1.2　定义

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

8.4.1.3　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

8.4.2　uart_parity_t

8.4.2.1　描述

UART 校验位。

8.4.2.2　定义

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

8.4.2.3　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

第 9 章

通用输入/输出（GPIO）

9.1 概述

芯片有 32 个高速 GPIO 和 8 个通用 GPIO。

9.2 功能描述

GPIO 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

9.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- gpio_get_pin_count
- gpio_set_drive_mode
- gpio_set_pin_edge
- gpio_set_on_\changed
- gpio_get_pin_value
- gpio_set_pin_value

9.3.1　gpio_get_pin_count

9.3.1.1　描述

获取 GPIO 管脚数量。

9.3.1.2　函数原型

```
uint32_t gpio_get_pin_count(handle_t file);
```

9.3.1.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入

9.3.1.4　返回值

管脚数量。

9.3.2　gpio_set_drive_mode

9.3.2.1　描述

设置 GPIO 管脚驱动模式。

9.3.2.2　函数原型

```
void gpio_set_drive_mode(handle_t file, uint32_t pin, gpio_drive_mode_t mode);
```

9.3.2.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入
pin	管脚编号	输入
mode	驱动模式	输入

9.3.2.4　返回值

无。

9.3.3　gpio_set_pin_edge

9.3.3.1　描述

设置 GPIO 管脚边沿触发模式。

注： /dev/gpio1 暂不支持。

9.3.3.2　函数原型

```
void gpio_set_pin_edge(handle_t file, uint32_t pin, gpio_pin_edge_t edge);
```

9.3.3.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入
pin	管脚编号	输入
edge	边沿触发模式	输入

9.3.3.4　返回值

无。

9.3.4　gpio_set_on_changed

9.3.4.1　描述

设置 GPIO 管脚边沿触发处理程序。

注： /dev/gpio1 暂不支持。

9.3.4.2　函数原型

```
void gpio_set_on_changed(handle_t file, uint32_t pin, gpio_on_changed_t callback, void
    *userdata);
```

9.3.4.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入

参数名称	描述	输入输出
pin	管脚编号	输入
callback	处理程序	输入
userdata	处理程序用户数据	输入

9.3.4.4　返回值

无。

9.3.5　gpio_get_pin_value

9.3.5.1　描述

获取 GPIO 管脚的值。

9.3.5.2　函数原型

```
gpio_pin_value_t gpio_get_pin_value(handle_t file, uint32_t pin);
```

9.3.5.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入
pin	管脚编号	输入

9.3.5.4　返回值

GPIO 管脚的值。

9.3.6　gpio_set_pin_value

9.3.6.1　描述

设置 GPIO 管脚的值。

9.3.6.2　函数原型

```
void gpio_set_pin_value(handle_t file, uint32_t pin, gpio_pin_value_t value);
```

9.3.6.3　参数

参数名称	描述	输入输出
file	GPIO 控制器句柄	输入
pin	管脚编号	输入
value	要设置的值	输入

9.3.6.4　返回值

无。

9.3.7　举例

```
handle_t gpio = io_open("/dev/gpio0");

gpio_set_drive_mode(gpio, 0, GPIO_DM_OUTPUT);
gpio_set_pin_value(gpio, 0, GPIO_PV_LOW);
```

9.4　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_edge_t：GPIO 边沿触发模式。
- gpio_pin_value_t：GPIO 值。
- gpio_on_changed_t：GPIO 边沿触发处理程序。

9.4.1　gpio_drive_mode_t

9.4.1.1　描述

GPIO 驱动模式。

9.4.1.2　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;
```

9.4.1.3　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

9.4.2　gpio_pin_edge_t

9.4.2.1　描述

GPIO 边沿触发模式。

9.4.2.2　定义

```
typedef enum _gpio_pin_edge
{
    GPIO_PE_NONE,
    GPIO_PE_FALLING,
    GPIO_PE_RISING,
    GPIO_PE_BOTH
} gpio_pin_edge_t;
```

9.4.2.3　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

9.4.3　gpio_pin_value_t

9.4.3.1　描述

GPIO 值。

9.4.3.2　定义

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

9.4.3.3　成员

成员名称	描述
GPIO_PV_LOW	低
GPIO_PV_HIGH	高

9.4.4　gpio_on_changed_t

9.4.4.1　描述

GPIO 边沿触发处理程序。

9.4.4.2　定义

```
typedef void (*gpio_on_changed_t)(uint32_t pin, void *userdata);
```

9.4.4.3　参数

参数名称	描述	输入输出
pin	管脚编号	输入
userdata	用户数据	输入

第 10 章

集成电路内置总线（I²C）

10.1 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。

10.2 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
- 支持从模式

10.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- i2c_get_device
- i2c_dev_set_clock_rate
- i2c_dev_transfer_sequential
- i2c_config_as_slave
- i2c_slave_set_clock_rate

10.3.1　i2c_get_device

10.3.1.1　描述

注册并打开一个 I²C 设备。

10.3.1.2　函数原型

```
handle_t i2c_get_device(handle_t file, const char *name, uint32_t slave_address,
    uint32_t address_width);
```

10.3.1.3　参数

参数名称	描述	输入输出
file	I²C 控制器句柄	输入
name	指定访问该设备的路径	输入
slave_address	从设备地址	输入
address_width	从设备地址宽度	输入

10.3.1.4　返回值

I²C 设备句柄。

10.3.2　i2c_dev_set_clock_rate

10.3.2.1　描述

配置 I²C 设备的时钟速率。

10.3.2.2　函数原型

```
double i2c_dev_set_clock_rate(handle_t file, double clock_rate);
```

10.3.2.3　参数

参数名称	描述	输入输出
file	I²C 设备句柄	输入
clock_rate	期望的时钟速率	输入

10.3.2.4　返回值

设置后的实际速率。

10.3.3　i2c_dev_transfer_sequential

10.3.3.1　描述

对 I²C 设备先读后写。

10.3.3.2　函数原型

```
int i2c_dev_transfer_sequential(handle_t file, const uint8_t *write_buffer, size_t
    write_len, uint8_t *read_buffer, size_t read_len);
```

10.3.3.3　参数

参数名称	描述	输入输出
file	I²C 设备句柄	输入
write_buffer	源缓冲区	输入
write_len	要写入的字节数	输入
read_buffer	目标缓冲区	输出
read_len	最多读取的字节数	输入

10.3.3.4　返回值

实际读取的字节数。

10.3.4　i2c_config_as_slave

10.3.4.1　描述

配置 I²C 控制器为从模式。

10.3.4.2　函数原型

```
void i2c_config_as_slave(handle_t file, uint32_t slave_address, uint32_t address_width,
     i2c_slave_handler_t *handler);
```

10.3.4.3　参数

参数名称	描述	输入输出
file	I²C 控制器句柄	输入
slave_address	从设备地址	输入
address_width	从设备地址宽度	输入
handler	从设备处理程序	输入

10.3.4.4　返回值

无。

10.3.5　spi_dev_set_clock_rate

10.3.5.1　描述

配置 I²C 从模式的时钟速率。

10.3.5.2　函数原型

```
double i2c_slave_set_clock_rate(handle_t file, double clock_rate);
```

10.3.5.3　参数

参数名称	描述	输入输出
file	I²C 控制器句柄	输入
clock_rate	期望的时钟速率	输入

10.3.5.4　返回值

设置后的实际速率。

10.3.6　举例

```
handle_t i2c = io_open("/dev/i2c0");
/* i2c外设地址是0x32, 7位地址，速率200K */
handle_t dev0 = i2c_get_device(i2c, "/dev/i2c0/dev0", 0x32, 7);
i2c_dev_set_clock_rate(dev0, 200000);

uint8_t reg = 0;
uint8_t data_buf[2] = { 0x00,0x01 };
data_buf[0] = reg;
```

```
/* 向0寄存器写0x01 */
io_write(dev0, data_buf, 2);
/* 从0寄存器读取1字节数据 */
i2c_dev_transfer_sequential(dev0, &reg, 1, data_buf, 1);
```

10.4　数据类型

相关数据类型、数据结构定义如下：

- i2c_event_t: I²C 事件。
- i2c_slave_handler_t: I²C 从设备处理程序。

10.4.1　i2c_event_t

10.4.1.1　描述

I²C 事件。

10.4.1.2　定义

```
typedef enum _i2c_event
{
    I2C_EV_START,
    I2C_EV_RESTART,
    I2C_EV_STOP
} i2c_event_t;
```

10.4.1.3　成员

成员名称	描述
I2C_EV_START	收到 Start 信号
I2C_EV_RESTART	收到 Restart 信号
I2C_EV_STOP	收到 Stop 信号

10.4.2　i2c_slave_handler_t

10.4.2.1　描述

I²C 从设备处理程序。

10.4.2.2　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;
```

10.4.2.3　成员

成员名称	描述
on_receive	收到数据时被调用
on_transmit	需要发送数据时被调用
on_event	发生事件时被调用

第 11 章

集成电路内置音频总线（I2S）

11.1 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。

11.2 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

11.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- i2s_config_as_render
- i2s_config_as_capture
- i2s_get_buffer
- i2s_release_buffer
- i2s_start
- i2s_stop

11.3.1　i2s_config_as_render

11.3.1.1　描述

配置 I2S 控制器为输出模式。

11.3.1.2　函数原型

```
void i2s_config_as_render(handle_t file, const audio_format_t *format, size_t delay_ms,
     i2s_align_mode_t align_mode, size_t channels_mask);
```

11.3.1.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入
format	音频格式	输入
delay_ms	缓冲区长度	输入
align_mode	对齐模式	输入
channels_mask	通道掩码	输入

11.3.1.4　返回值

无。

11.3.2　i2s_config_as_capture

11.3.2.1　描述

配置 I2S 控制器为捕获模式。

11.3.2.2　函数原型

```
void i2s_config_as_capture(handle_t file, const audio_format_t *format, size_t delay_ms
    , i2s_align_mode_t align_mode, size_t channels_mask);
```

11.3.2.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入

参数名称	描述	输入输出
format	音频格式	输入
delay_ms	缓冲区长度	输入
align_mode	对齐模式	输入
channels_mask	通道掩码	输入

11.3.2.4　返回值

无。

11.3.3　i2s_get_buffer

11.3.3.1　描述

获取音频缓冲区。

11.3.3.2　函数原型

```
void i2s_get_buffer(handle_t file, uint8_t **buffer, size_t *frames);
```

11.3.3.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入
buffer	缓冲区	输出
frames	缓冲区帧数	输出

11.3.3.4　返回值

无。

11.3.4　i2s_release_buffer

11.3.4.1　描述

释放音频缓冲区。

11.3.4.2　函数原型

```
void i2s_release_buffer(handle_t file, size_t frames);
```

11.3.4.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入
frames	确认已读取或写入的帧数	输入

11.3.4.4　返回值

无。

11.3.5　i2s_start

11.3.5.1　描述

开始播放或录音。

11.3.5.2　函数原型

```
void i2s_start(handle_t file);
```

11.3.5.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入

11.3.5.4　返回值

无。

11.3.6　i2s_stop

11.3.6.1　描述

停止播放或录音。

11.3.6.2　函数原型

```
void i2s_stop(handle_t file);
```

11.3.6.3　参数

参数名称	描述	输入输出
file	I2S 控制器句柄	输入

11.3.6.4　返回值

无。

11.3.7　举例

```
/* 循环播放 PCM 音频 */
handle_t i2s = io_open("/dev/i2s0");
audio_format_t audio_fmt = { .type = AUDIO_FMT_PCM, .bits_per_sample = 16, .sample_rate
     = 44100, .channels = 2 };
i2s_config_as_render(i2s, &audio_fmt, 100, I2S_AM_RIGHT, 0b11);
i2s_start(i2s);

while (1)
{
    uint8_t *buffer;
    size_t frames;
    i2s_get_buffer(i2s, &buffer, &frames);
    memcpy(buffer, pcm, 4 * frames);
    i2s_release_buffer(i2s, frames);
    pcm += frames;
    if (pcm >= pcm_end)
        pcm = pcm_start;
}
```

11.4　数据类型

相关数据类型、数据结构定义如下：

- audio_format_type_t：音频格式类型。
- audio_format_t：音频格式。
- i2s_align_mode_t：I2S 对齐模式。

11.4.1　audio_format_type_t

11.4.1.1　描述

音频格式类型。

11.4.1.2　定义

```
typedef enum _audio_format_type
{
    AUDIO_FMT_PCM
} audio_format_type_t;
```

11.4.1.3　成员

成员名称	描述
AUDIO_FMT_PCM	PCM

11.4.2　audio_format_t

11.4.2.1　描述

音频格式。

11.4.2.2　定义

```
typedef struct _audio_format
{
    audio_format_type_t type;
    uint32_t bits_per_sample;
    uint32_t sample_rate;
    uint32_t channels;
} audio_format_t;
```

11.4.2.3　成员

成员名称	描述
type	音频格式类型
bits_per_sample	采样深度
sample_rate	采样率

成员名称	描述
channels	声道数

11.4.3　i2s_align_mode_t

11.4.3.1　描述

I2S 对齐模式。

11.4.3.2　定义

```
typedef enum _i2s_align_mode
{
    I2S_AM_STANDARD,
    I2S_AM_RIGHT,
    I2S_AM_LEFT
} i2s_align_mode_t;
```

11.4.3.3　成员

成员名称	描述
I2S_AM_STANDARD	标准模式
I2S_AM_RIGHT	右对齐
I2S_AM_LEFT	左对齐

第 12 章

串行外设接口（SPI）

12.1 概述

SPI 是一种高速的，全双工，同步的通信总线。

12.2 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

12.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- spi_get_device
- spi_dev_config_non_standard
- spi_dev_set_clock_rate
- spi_dev_transfer_full_duplex
- spi_dev_transfer_sequential
- spi_dev_fill

12.3.1　spi_get_device

12.3.1.1　描述

注册并打开一个 SPI 设备。

12.3.1.2　函数原型

```
handle_t spi_get_device(handle_t file, const char *name, spi_mode mode,
    spi_frame_format frame_format, uint32_t chip_select_mask, uint32_t data_bit_length
    );
```

12.3.1.3　参数

参数名称	描述	输入输出
file	SPI 控制器句柄	输入
name	指定访问该设备的路径	输入
mode	SPI 模式	输入
frame_format	帧格式	输入
chip_select_mask	片选掩码	输入
data_bit_length	数据位长度	输入

12.3.1.4　返回值

SPI 设备句柄。

12.3.2　spi_dev_config_non_standard

12.3.2.1　描述

配置 SPI 设备的非标准帧格式参数。

12.3.2.2　函数原型

```
void spi_dev_config_non_standard(handle_t file, uint32_t instruction_length, uint32_t
    address_length, uint32_t wait_cycles, spi_inst_addr_trans_mode_t trans_mode);
```

12.3.2.3　参数

参数名称	描述	输入输出
file	SPI 设备句柄	输入
instruction_length	指令长度	输入
address_length	地址长度	输入
wait_cycles	等待周期数	输入
trans_mode	指令和地址的传输模式	输入

12.3.2.4　返回值

无。

12.3.3　spi_dev_set_clock_rate

12.3.3.1　描述

配置 SPI 设备的时钟速率。

12.3.3.2　函数原型

```
double spi_dev_set_clock_rate(handle_t file, double clock_rate);
```

12.3.3.3　参数

参数名称	描述	输入输出
file	SPI 设备句柄	输入
clock_rate	期望的时钟速率	输入

12.3.3.4　返回值

设置后的实际速率。

12.3.4　spi_dev_transfer_full_duplex

12.3.4.1　描述

对 SPI 设备进行全双工传输。

注： 仅支持标准帧格式。

12.3.4.2　函数原型

```
int spi_dev_transfer_full_duplex(handle_t file, const uint8_t *write_buffer, size_t
    write_len, uint8_t *read_buffer, size_t read_len);
```

12.3.4.3　参数

参数名称	描述	输入输出
file	SPI 设备句柄	输入
write_buffer	源缓冲区	输入
write_len	要写入的字节数	输入
read_buffer	目标缓冲区	输出
read_len	最多读取的字节数	输入

12.3.4.4　返回值

实际读取的字节数。

12.3.5　spi_dev_transfer_sequential

12.3.5.1　描述

对 SPI 设备进行先写后读。

注： 仅支持标准帧格式。

12.3.5.2　函数原型

```
int spi_dev_transfer_sequential(handle_t file, const uint8_t *write_buffer, size_t
    write_len, uint8_t *read_buffer, size_t read_len);
```

12.3.5.3　参数

参数名称	描述	输入输出
file	SPI 设备句柄	输入
write_buffer	源缓冲区	输入
write_len	要写入的字节数	输入
read_buffer	目标缓冲区	输出
read_len	最多读取的字节数	输入

12.3.5.4　返回值

实际读取的字节数。

12.3.6　spi_dev_fill

12.3.6.1　描述

对 SPI 设备填充一串相同的帧。

注： 仅支持标准帧格式。

12.3.6.2　函数原型

```
void spi_dev_fill(handle_t file, uint32_t instruction, uint32_t address, uint32_t value
    , size_t count);
```

12.3.6.3　参数

参数名称	描述	输入输出
file	SPI 设备句柄	输入
instruction	指令（标准帧格式下忽略）	输入
address	地址（标准帧格式下忽略）	输入
value	帧数据	输出
count	帧数	输入

12.3.6.4　返回值

无。

12.3.7　举例

```
handle_t spi = io_open("/dev/spi0");
/* dev0 工作在 MODE0 模式 标准 SPI 模式 单次发送 8 位数据 使用片选 0 */
handle_t dev0 = spi_get_device(spi, "/dev/spi0/dev0", SPI_MODE_0, SPI_FF_STANDARD, 0b1,
     8);
uint8_t data_buf[] = { 0x06, 0x01, 0x02, 0x04, 0, 1, 2, 3 };
/* 发送指令 0x06 向地址 0x010204 发送 0，1，2，3 四个字节数据 */
io_write(dev0, data_buf, sizeof(data_buf));
/* 发送指令 0x06 地址 0x010204 接收四个字节的数据 */
spi_dev_transfer_sequential(dev0, data_buf, 4, data_buf, 4);
```

12.4　数据类型

相关数据类型、数据结构定义如下：

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

12.4.1　spi_mode_t

12.4.1.1　描述

SPI 模式。

12.4.1.2　定义

```
typedef enum _spi_mode
{
    SPI_MODE_0,
    SPI_MODE_1,
    SPI_MODE_2,
    SPI_MODE_3,
} spi_mode_t;
```

12.4.1.3　成员

成员名称	描述
SPI_MODE_0	SPI 模式 0
SPI_MODE_1	SPI 模式 1
SPI_MODE_2	SPI 模式 2
SPI_MODE_3	SPI 模式 3

12.4.2　spi_frame_format_t

12.4.2.1　描述

SPI 帧格式。

12.4.2.2　定义

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

12.4.2.3　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（/dev/spi3 不支持）

12.4.3　spi_inst_addr_trans_mode_t

12.4.3.1　描述

SPI 指令和地址的传输模式。

12.4.3.2　定义

```
typedef enum _spi_inst_addr_trans_mode
{
    SPI_AITM_STANDARD,
    SPI_AITM_ADDR_STANDARD,
    SPI_AITM_AS_FRAME_FORMAT
} spi_inst_addr_trans_mode_t;
```

12.4.3.3　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 13 章

数字摄像头接口（DVP）

13.1 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。

13.2 功能描述

DVP 模块具有以下功能：

- 支持 RGB565、RGB422 与单通道 Y 灰度输入模式
- 支持设置帧中断
- 支持设置传输地址
- 支持同时向两个地址写数据（输出格式分别是 RGB888 与 RGB565）
- 支持丢弃不需要处理的帧

13.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- dvp_xclk_set_clock_rate
- dvp_config
- dvp_enable_frame
- dvp_get_output_num
- dvp_set_signal
- dvp_set_output_enable

- dvp_set_output_attributes
- dvp_set_frame_event_enable
- dvp_set_on_frame_event

13.3.1　dvp_xclk_set_clock_rate

13.3.1.1　描述

配置 DVP XCLK 的频率。

13.3.1.2　函数原型

```
double dvp_xclk_set_clock_rate(handle_t file, double clock_rate);
```

13.3.1.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
clock_rate	配置 XCLK 的频率，如 OV5640 配置为 20MHz	输入

13.3.1.4　返回值

设置后的实际频率。

13.3.2　dvp_config

13.3.2.1　描述

配置 DVP 设备。

13.3.2.2　函数原型

```
void dvp_config(handle_t file, uint32_t width, uint32_t height, bool auto_enable);
```

13.3.2.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
width	帧宽度	输入

参数名称	描述	输入输出
height	帧高度	输入
auto_enable	自动启用帧处理	输入

13.3.2.4　返回值

无。

13.3.3　dvp_enable_frame

13.3.3.1　描述

启用对当前帧的处理。

13.3.3.2　函数原型

```
void dvp_enable_frame(handle_t file);
```

13.3.3.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入

13.3.3.4　返回值

无。

13.3.4　dvp_get_output_num

13.3.4.1　描述

获取 DVP 设备的输出数目。

13.3.4.2　函数原型

```
uint32_t dvp_get_output_num(handle_t file);
```

13.3.4.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入

13.3.4.4　返回值

输出数目。

13.3.5　dvp_set_signal

13.3.5.1　描述

设置 DVP 信号状态。

13.3.5.2　函数原型

```
void dvp_set_signal(handle_t file, dvp_signal_type_t type, bool value);
```

13.3.5.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
type	信号类型	输入
value	状态值	输入

13.3.5.4　返回值

无。

13.3.6　dvp_set_output_enable

13.3.6.1　描述

设置 DVP 输出是否启用。

13.3.6.2　函数原型

```
void dvp_set_output_enable(handle_t file, uint32_t index, bool enable);
```

13.3.6.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
index	输出索引	输入
enable	是否启用	输入

13.3.6.4　返回值

无。

13.3.7　dvp_set_output_attributes

13.3.7.1　描述

设置 DVP 输出特性。

13.3.7.2　函数原型

```
void dvp_set_output_attributes(handle_t file, uint32_t index, video_format_t format,
    void *output_buffer);
```

13.3.7.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
index	输出索引	输入
format	视频格式	输入
output_buffer	输出缓冲	输出

13.3.7.4　返回值

无。

13.3.8　dvp_set_frame_event_enable

13.3.8.1　描述

设置 DVP 帧事件是否启用。

13.3.8.2　函数原型

```
void dvp_set_frame_event_enable(handle_t file, dvp_frame_event_t event, bool enable);
```

13.3.8.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
event	帧事件	输入
enable	是否启用	输入

13.3.8.4　返回值

无。

13.3.9　dvp_set_on_frame_event

13.3.9.1　描述

设置 DVP 帧事件处理程序。

13.3.9.2　函数原型

```
void dvp_set_on_frame_event(handle_t file, dvp_on_frame_event_t handler, void *userdata
    );
```

13.3.9.3　参数

参数名称	描述	输入输出
file	DVP 设备句柄	输入
handler	处理程序	输入
userdata	处理程序用户数据	输入

13.3.9.4　返回值

无。

13.3.10　举例

```
handle_t dvp = io_open("/dev/dvp0");
```

```
dvp_config(dvp, 320, 240, false);
dvp_set_on_frame_event(dvp, on_frame_isr, NULL);
dvp_set_frame_event_enable(dvp, VIDEO_FE_BEGIN, true);
dvp_set_output_attributes(dvp, 0, VIDEO_FMT_RGB565, lcd_gram0);
dvp_set_output_enable(dvp, 0, true);
```

13.4　数据类型

相关数据类型、数据结构定义如下：

- video_format_t：视频格式。
- dvp_frame_event_t：DVP 帧事件。
- dvp_signal_type_t：DVP 信号类型。
- dvp_on_frame_event_t：DVP 帧事件处理程序。

13.4.1　video_format_t

13.4.1.1　描述

视频格式。

13.4.1.2　定义

```
typedef enum _video_format
{
    VIDEO_FMT_RGB565,
    VIDEO_FMT_RGB24_PLANAR
} video_format_t;
```

13.4.1.3　成员

成员名称	描述
VIDEO_FMT_RGB565	RGB565
VIDEO_FMT_RGB24_PLANAR	RGB24 Planar

13.4.2　dvp_frame_event_t

13.4.2.1　描述

DVP 帧事件。

13.4.2.2　定义

```
typedef enum _video_frame_event
{
    VIDEO_FE_BEGIN,
    VIDEO_FE_END
} dvp_frame_event_t;
```

13.4.2.3　成员

成员名称	描述
VIDEO_FE_BEGIN	帧开始
VIDEO_FE_END	帧结束

13.4.3　dvp_signal_type_t

13.4.3.1　描述

DVP 信号类型。

13.4.3.2　定义

```
typedef enum _dvp_signal_type
{
    DVP_SIG_POWER_DOWN,
    DVP_SIG_RESET
} dvp_signal_type_t;
```

13.4.3.3　成员

成员名称	描述
DVP_SIG_POWER_DOWN	掉电
DVP_SIG_RESET	复位

13.4.4　dvp_on_frame_event_t

13.4.4.1　描述

TIMER 触发时的处理程序。

13.4.4.2　定义

```
typedef void (*dvp_on_frame_event_t)(dvp_frame_event_t event, void *userdata);
```

13.4.4.3　参数

参数名称	描述	输入输出
userdata	用户数据	输入

第 14 章

串行摄像机控制总线（SCCB）

14.1 概述

SCCB 是一种串行摄像机控制总线。

14.2 功能描述

SCCB 模块具有以下功能：

- 独立的 SCCB 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 devices.h
为用户提供以下接口：

- sccb_get_device
- sccb_dev_read_byte
- sccb_dev_write_byte

14.3.1 sccb_get_device

14.3.1.1 描述

注册并打开一个 SCCB 设备。

14.3.1.2　函数原型

```
handle_t sccb_get_device(handle_t file, const char *name, size_t slave_address, size_t
    reg_address_width);
```

14.3.1.3　参数

参数名称	描述	输入输出
file	SCCB 控制器句柄	输入
name	指定访问该设备的路径	输入
slave_address	从设备地址	输入
reg_address_width	寄存器地址宽度	输入

14.3.1.4　返回值

SCCB 设备句柄。

14.3.2　sccb_dev_read_byte

14.3.2.1　描述

从 SCCB 设备读取一个字节。

14.3.2.2　函数原型

```
uint8_t sccb_dev_read_byte(handle_t file, uint16_t reg_address);
```

14.3.2.3　参数

参数名称	描述	输入输出
file	SCCB 设备句柄	输入
reg_address	寄存器地址	输入

14.3.2.4　返回值

读取的字节。

14.3.3　sccb_dev_write_byte

14.3.3.1　描述

向 SCCB 设备写入一个字节。

14.3.3.2　函数原型

```
void sccb_dev_write_byte(handle_t file, uint16_t reg_address, uint8_t value);
```

14.3.3.3　参数

参数名称	描述	输入输出
file	SCCB 设备句柄	输入
reg_address	寄存器地址	输入
value	要写入的字节	输入

14.3.3.4　返回值

无。

14.3.4　举例

```
handle_t sccb = io_open("/dev/sccb0");
handle_t dev0 = sccb_get_device(sccb, "/dev/sccb0/dev0", 0x60, 8);

sccb_dev_write_byte(dev0, 0xFF, 0);
uint8_t value = sccb_dev_read_byte(dev0, 0xFF);
```

第 15 章

定时器（TIMER）

15.1 概述

TIMER 提供高精度定时功能。

15.2 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

15.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- timer_set_interval
- timer_set_on_tick
- timer_set_enable

15.3.1 timer_set_interval

15.3.1.1 描述

设置 TIMER 触发间隔。

15.3.1.2　函数原型

```
size_t timer_set_interval(handle_t file, size_t nanoseconds);
```

15.3.1.3　参数

参数名称	描述	输入输出
file	TIMER 设备句柄	输入
nanoseconds	间隔（纳秒）	输入

15.3.1.4　返回值

实际的触发间隔（纳秒）。

15.3.2　timer_set_on_tick

15.3.2.1　描述

设置 TIMER 触发时的处理程序。

15.3.2.2　函数原型

```
void timer_set_on_tick(handle_t file, timer_on_tick_t on_tick, void *userdata);
```

15.3.2.3　参数

参数名称	描述	输入输出
file	TIMER 设备句柄	输入
on_tick	处理程序	输入
userdata	处理程序用户数据	输入

15.3.2.4　返回值

无。

15.3.3　timer_set_enable

15.3.3.1　描述

设置 TIMER 是否启用。

15.3.3.2　函数原型

```
void timer_set_enable(handle_t file, bool enable);
```

15.3.3.3　参数

参数名称	描述	输入输出
file	TIMER 设备句柄	输入
enable	是否启用	输入

15.3.3.4　返回值

无。

15.3.4　举例

```
/* 定时器0  定时 1 秒打印 Time OK! */
void on_tick(void *unused)
{
    printf("Time OK!\n");
}

handle_t timer = io_open("/dev/timer0");

timer_set_interval(timer, 1e9);
timer_set_on_tick(timer, on_tick, NULL);
timer_set_enable(timer, true);
```

15.4　数据类型

相关数据类型、数据结构定义如下：

- timer_on_tick_t：TIMER 触发时的处理程序。

15.4.1　timer_on_tick_t

15.4.1.1　描述

TIMER 触发时的处理程序。

15.4.1.2　定义

```
typedef void (*timer_on_tick_t)(void *userdata);
```

15.4.1.3　参数

参数名称	描述	输入输出
userdata	用户数据	输入

第 16 章

脉冲宽度调制器（PWM）

16.1 概述

PWM 用于控制脉冲输出的占空比。

16.2 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

16.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- pwm_get_pin_count
- pwm_set_frequency
- pwm_set_active_duty_cycle_percentage
- pwm_set_enable

16.3.1 pwm_get_pin_count

16.3.1.1 描述

获取 PWM 管脚数量。

16.3.1.2　函数原型

```
uint32_t pwm_get_pin_count(handle_t file);
```

16.3.1.3　参数

参数名称	描述	输入输出
file	PWM 设备句柄	输入

16.3.1.4　返回值

PWM 管脚数量。

16.3.2　pwm_set_frequency

16.3.2.1　描述

设置 PWM 频率。

16.3.2.2　函数原型

```
double pwm_set_frequency(handle_t file, double frequency);
```

16.3.2.3　参数

参数名称	描述	输入输出
file	PWM 设备句柄	输入
frequency	期望的频率（Hz）	输入

16.3.2.4　返回值

设置后实际的频率（Hz）。

16.3.3　pwm_set_active_duty_cycle_percentage

16.3.3.1　描述

设置 PWM 管脚占空比。

16.3.3.2　函数原型

```
double pwm_set_active_duty_cycle_percentage(handle_t file, uint32_t pin, double
    duty_cycle_percentage);
```

16.3.3.3　参数

参数名称	描述	输入输出
file	PWM 设备句柄	输入
pin	管脚编号	输入
duty_cycle_percentage	期望的占空比	输入

16.3.3.4　返回值

设置后实际的占空比。

16.3.4　pwm_set_enable

16.3.4.1　描述

设置 PWM 管脚是否启用。

16.3.4.2　函数原型

```
void pwm_set_enable(handle_t file, uint32_t pin, bool enable);
```

16.3.4.3　参数

参数名称	描述	输入输出
file	PWM 设备句柄	输入
pin	管脚编号	输入
enable	是否启用	输入

16.3.4.4　返回值

设置后实际的占空比。

16.3.5　举例

```
/* pwm0 pin0 输出 200KHZ 占空比为 0.5 的方波 */
handle_t pwm = io_open("/dev/pwm0");
pwm_set_frequency(pwm, 200000);
pwm_set_active_duty_cycle_percentage(pwm, 0, 0.5);
pwm_set_enable(pwm, 0, true);
```

第 17 章

看门狗定时器（WDT）

17.1 概述

WDT 提供系统出错或无响应时的恢复功能。

17.2 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
- 配置为超时后复位或进入中断
- 进入中断后清除中断可取消复位，否则等待第二次超时后复位

17.3 API 参考

对应的头文件 devices.h
为用户提供以下接口：

- wdt_set_response_mode
- wdt_set_timeout
- wdt_set_on_timeout
- wdt_restart_counter
- wdt_set_enable

17.3.1　wdt_set_response_mode

17.3.1.1　描述

设置 WDT 响应模式。

17.3.1.2　函数原型

```
void wdt_set_response_mode(handle_t file, wdt_response_mode_t mode);
```

17.3.1.3　参数

参数名称	描述	输入输出
file	WDT 设备句柄	输入
mode	响应模式	输入

17.3.1.4　返回值

无。

17.3.2　wdt_set_timeout

17.3.2.1　描述

设置 WDT 超时时间。

17.3.2.2　函数原型

```
size_t wdt_set_timeout(handle_t file, size_t nanoseconds);
```

17.3.2.3　参数

参数名称	描述	输入输出
file	WDT 设备句柄	输入
nanoseconds	期望的超时时间（纳秒）	输入

17.3.2.4　返回值

设置后实际的超时时间（纳秒）。

17.3.3　wdt_set_on_timeout

17.3.3.1　描述

设置 WDT 超时处理程序。

17.3.3.2　函数原型

```
void wdt_set_on_timeout(handle_t file, wdt_on_timeout_t handler, void *userdata);
```

17.3.3.3　参数

参数名称	描述	输入输出
file	WDT 设备句柄	输入
handler	处理程序	输入
userdata	处理程序用户数据	输入

17.3.3.4　返回值

无。

17.3.4　wdt_restart_counter

17.3.4.1　描述

使 WDT 重新开始计数。

17.3.4.2　函数原型

```
void wdt_restart_counter(handle_t file);
```

17.3.4.3　参数

参数名称	描述	输入输出
file	WDT 设备句柄	输入

17.3.4.4　返回值

无。

17.3.5　wdt_set_enable

17.3.5.1　描述

设置 WDT 是否启用。

17.3.5.2　函数原型

```
void wdt_set_enable(handle_t file, bool enable);
```

17.3.5.3　参数

参数名称	描述	输入输出
file	WDT 设备句柄	输入
enable	是否启用	输入

17.3.5.4　返回值

无。

17.3.6　举例

```
/* 2 秒后进入看门狗中断函数打印 Timeout,再过 2 秒复位 */
void on_timeout(void *unused)
{
    printf("Timeout\n");
}

handle_t wdt = io_open("/dev/wdt0");

wdt_set_response_mode(wdt, WDT_RESP_INTERRUPT);
wdt_set_timeout(wdt, 2e9);
wdt_set_on_timeout(wdt, on_timeout, NULL);
wdt_set_enable(wdt, true);
```

17.4　数据类型

相关数据类型、数据结构定义如下：

- wdt_response_mode_t：WDT 响应模式。

- wdt_on_timeout_t: WDT 超时处理程序。

17.4.1　wdt_response_mode_t

17.4.1.1　描述

WDT 响应模式。

17.4.1.2　定义

```
typedef enum _wdt_response_mode
{
    WDT_RESP_RESET,
    WDT_RESP_INTERRUPT
} wdt_response_mode_t;
```

17.4.1.3　成员

成员名称	描述
WDT_RESP_RESET	超时后复位系统
WDT_RESP_INTERRUPT	超时后进入中断，再次超时复位系统

17.4.2　wdt_on_timeout_t

17.4.2.1　描述

WDT 超时处理程序。

17.4.2.2　定义

```
typedef int (*wdt_on_timeout_t)(void *userdata);
```

17.4.2.3　参数

参数名称	描述	输入输出
userdata	用户数据	输入

17.4.2.4　返回值

返回值	描述
0	不清除中断，系统将复位
1	清除中断，系统不复位

第 18 章

快速傅立叶变换加速器（FFT）

18.1 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。

18.2 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 * 32 bit 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。

18.3 API 参考

对应的头文件 `fft.h`
为用户提供以下接口：

- `fft_complex_uint16`

18.3.1 fft_complex_uint16

18.3.1.1 描述

FFT 运算。

18.3.1.2　函数原型

```
void fft_complex_uint16(uint16_t shift, fft_direction_t direction, const uint64_t *
    input, size_t point_num, uint64_t *output);
```

18.3.1.3　参数

参数名称	描述	输入输出
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 16 bit	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 16 bit	输出

18.3.2　举例

```
#define FFT_N                512U
#define FFT_FORWARD_SHIFT    0x0U
#define FFT_BACKWARD_SHIFT   0x1ffU
#define PI                   3.14159265358979323846
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(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(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 ;
}
```

18.4　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 运算模式。

18.4.1　fft_data_t

18.4.1.1　描述

FFT 运算传入的数据格式。

18.4.1.2　定义

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

18.4.1.3　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

18.4.2　fft_direction_t

18.4.2.1　描述

FFT 运算模式

18.4.2.2　定义

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

18.4.2.3　成员

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

第 19 章

安全散列算法加速器（SHA256）

19.1 概述

SHA256 模块是用硬件的方式来实现 SHA256 的时分运算加速。

19.2 功能描述

- 支持 SHA-256 的计算

19.3 API 参考

对应的头文件 sha256.h

为用户提供以下接口：

- sha256_hard_calculate

19.3.1 sha256_hard_calculate

19.3.1.1 描述

对数据进行 SHA256 计算

19.3.1.2 函数原型

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

19.3.1.3　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 32 byte	输出

19.3.2　举例

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

第 20 章

高级加密加速器（AES）

20.1 概述

AES 模块是用硬件的方式来实现 AES 的时分运算加速。

20.2 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。

20.3 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

20.3.1　aes_ecb128_hard_encrypt

20.3.1.1　描述

AES-ECB-128 加密运算，当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.1.2　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)
```

20.3.1.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.1.4　返回值

无。

20.3.2　aes_ecb128_hard_decrypt

20.3.2.1　描述

AES-ECB-128 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.2.2　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)
```

20.3.2.3　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
output_data	AES-ECB-128 解密运算后的结果存放在这个 buffer。 这个 buffer 大小需要至少为 16bytes 的整数倍	输出

20.3.2.4　返回值

无。

20.3.3　aes_ecb192_hard_encrypt

20.3.3.1　描述

AES-ECB-192 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.3.2　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)
```

20.3.3.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.3.4　返回值

无。

20.3.4　aes_ecb192_hard_decrypt

20.3.4.1　描述

AES-ECB-192 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.4.2　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)
```

20.3.4.3　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

参数名称	描述	输入输出
output_data	AES-ECB-192 解密运算后的结果存放在这个 buffer。 这个 buffer 大小需要至少为 16bytes 的整数倍	输出

20.3.4.4　返回值

无。

20.3.5　aes_ecb256_hard_encrypt

20.3.5.1　描述

AES-ECB-256 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.5.2　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)
```

20.3.5.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.5.4　返回值

无。

20.3.6　aes_ecb256_hard_decrypt

20.3.6.1　描述

AES-ECB-256 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.6.2　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)
```

20.3.6.3　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
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 大小需要至少为 16bytes 的整数倍	输出

20.3.6.4　返回值

无。

20.3.7　aes_cbc128_hard_encrypt

20.3.7.1　描述

AES-CBC-128 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.7.2　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)
```

20.3.7.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.7.4　返回值

无。

20.3.8　aes_cbc128_hard_decrypt

20.3.8.1　描述

AES-CBC-128 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.8.2　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)
```

20.3.8.3　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

参数名称	描述	输入输出
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 大小需要至少为 16bytes 的整数倍	输出

20.3.8.4　返回值

无。

20.3.9　aes_cbc192_hard_encrypt

20.3.9.1　描述

AES-CBC-192 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.9.2　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)
```

20.3.9.3　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

参数名称	描述	输入输出
output_data	AES-CBC-192 加密运算后的结果存放在这个 buffer。 这个 buffer 大小需要至少为 16bytes 的整数倍	输出

20.3.9.4　返回值

无。

20.3.10　aes_cbc192_hard_decrypt

20.3.10.1　描述

AES-CBC-192 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.10.2　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)
```

20.3.10.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.10.4　返回值

无。

20.3.11　aes_cbc256_hard_encrypt

20.3.11.1　描述

AES-CBC-256 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.11.2　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)
```

20.3.11.3　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
output_data	AES-CBC-256 加密运算后的结果存放在这个 buffer。 这个 buffer 大小需要至少为 16bytes 的整数倍	输出

20.3.11.4　返回值

无。

20.3.12　aes_cbc256_hard_decrypt

20.3.12.1　描述

AES-CBC-256 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.12.2　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)
```

20.3.12.3　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 大小需要至少为 16bytes 的整数倍	输出

20.3.12.4　返回值

无。

20.3.13　aes_gcm128_hard_encrypt

20.3.13.1　描述

AES-GCM-128 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.13.2　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)
```

20.3.13.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-128 加密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.13.4　返回值

无。

20.3.14　aes_gcm128_hard_decrypt

20.3.14.1　描述

AES-GCM-128 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.14.2　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)
```

20.3.14.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-128 解密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.14.4　返回值

无。

20.3.15　aes_gcm192_hard_encrypt

20.3.15.1　描述

AES-GCM-192 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.15.2　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)
```

20.3.15.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-192 加密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.15.4　返回值

无。

20.3.16　aes_gcm192_hard_decrypt

20.3.16.1　描述

AES-GCM-192 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.16.2　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)
```

20.3.16.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-192 解密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.16.4　返回值

无。

20.3.17　aes_gcm256_hard_encrypt

20.3.17.1　描述

AES-GCM-256 加密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.17.2　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)
```

20.3.17.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-256 加密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.17.4　返回值

无。

20.3.18　aes_gcm256_hard_decrypt

20.3.18.1　描述

AES-GCM-256 解密运算。当加密数据量小于等于 896bytes 时会使用 cpu 来传输数据，大于 896bytes 时，会使用 dma 来传输数据，从而提高计算的效率。

20.3.18.2　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)
```

20.3.18.3　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。 这个 buffer 大小需要至少为 4bytes 的整数倍， 因为 DMA 的传输数据的最小粒度为 4bytes。	输出
gcm_tag	AES-GCM-256 解密运算后的 tag 存放在这个 buffer。 这个 buffer 的大小需要保证为 16bytes	输出

20.3.18.4　返回值

无。

20.3.19　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);
```

20.4　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。

20.4.1　aes_cipher_mode_t

20.4.1.1　描述

AES 加密/解密的方式。

20.4.1.2　定义

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

20.4.1.3　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