K210
FreeRTOS SDK 編程指南

KENDRYTE
勘智

關於本手冊

本文檔為用戶提供基於 FreeRTOS SDK 開發時的編程指南.

對應 SDK 版本

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

發佈說明

日期	版本	發佈說明
2018-10-12	V0.1.0	初始版本

免責聲明

本文中的資訊，包括參考的 URL 地址，如有變更，恕不另行通知。文檔“按現狀”提供，不負任何擔保責任，包括對適銷性、適用於特定用途或非侵權性的任何擔保，和任何提案、規格或樣品在他處提到的任何擔保。本文檔不負任何責任，包括使用本文檔內資訊產生的侵犯任何專利權行為的責任。本文檔在此未以禁止反言或其他方式授予任何知識產權使用許可，不管是明示許可還是暗示許可。文中提到的所有商標名稱、商標和註冊商標均屬其各自所有者的財產，特此聲明。

版權公告

目錄

第 1 章

FreeRTOS 擴展

1.1 概述

FreeRTOS 是一個輕量級的實時操作系統。本 SDK 在其基礎上新增了一些適用於 K210 的功能。

1.2 功能描述

FreeRTOS 擴展模組具有以下功能:

- 獲取當前任務所在的邏輯處理器 Id
- 在指定邏輯處理器創建任務

K210 包含 2 個邏輯處理器，Id 分別為 0 和 1。

1.3 API 參考

對應的頭文件 task.h
為用戶提供以下介面:

- uxTaskGetProcessorId
- xTaskCreateAtProcessor

1.3.1 uxTaskGetProcessorId

1.3.1.1 描述

獲取當前邏輯處理器 Id。

1.3.1.2　函數原型

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

1.3.1.3　返回值

當前邏輯處理器 Id。

1.3.2　xTaskCreateAtProcessor

1.3.2.1　描述

在指定邏輯處理器創建任務。

1.3.2.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
uxProcessor	邏輯處理器 Id	輸入
pxTaskCode	任務入口點	輸入
pcName	任務名稱	輸入
usStackDepth	棧空間	輸入
pvParameters	參數	輸入
uxPriority	優先順序	輸入
pxCreatedTask	創建的任務句柄	輸入

1.3.2.4　返回值

返回值	描述
pdPASS	成功
其他	失敗

第 2 章

裝置列表

路徑	類型	備註
/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	不可與 /dev/pwm0 同時使用
/dev/timer1	TIMER	不可與 /dev/pwm0 同時使用
/dev/timer2	TIMER	不可與 /dev/pwm0 同時使用

路徑	類型	備註
/dev/timer3	TIMER	不可與 /dev/pwm0 同時使用
/dev/timer4	TIMER	不可與 /dev/pwm1 同時使用
/dev/timer5	TIMER	不可與 /dev/pwm1 同時使用
/dev/timer6	TIMER	不可與 /dev/pwm1 同時使用
/dev/timer7	TIMER	不可與 /dev/pwm1 同時使用
/dev/timer8	TIMER	不可與 /dev/pwm2 同時使用
/dev/timer9	TIMER	不可與 /dev/pwm2 同時使用
/dev/timer10	TIMER	不可與 /dev/pwm2 同時使用
/dev/timer11	TIMER	不可與 /dev/pwm2 同時使用
/dev/pwm0	PWM	不可與 /dev/timer[0-3] 同時使用
/dev/pwm1	PWM	不可與 /dev/timer[4-7] 同時使用
/dev/pwm2	PWM	不可與 /dev/timer[8-11] 同時使用
/dev/wdt0	WDT	
/dev/wdt1	WDT	
/dev/rtc0	RTC	

第 3 章

腳位配置

3.1 概述

腳位配置包含 FPIOA 和電源域配置等。

3.2 功能描述

- 支持 IO 的可編程功能選擇
- 配置電源域

3.3 資料類型

對應的頭文件 pin_cfg.h
相關資料類型、資料結構定義如下：

- fpioa_function_t: 腳位的功能編號。
- fpioa_cfg_item_t: FPIOA 腳位配置。
- fpioa_cfg_t: FPIOA 配置。
- sysctl_power_bank_t: 電源域編號。
- sysctl_io_power_mode_t: IO 輸出電壓值。
- power_bank_item_t: 單個電源域配置。
- power_bank_cfg_t: 電源域配置。
- pin_cfg_t: 腳位配置。

3.3.1　fpioa_function_t

3.3.1.1　描述

腳位的功能編號。

3.3.1.2　定義

```
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　成員

成員名稱	描述
FUNC_JTAG_TCLK	JTAG 時脈介面
FUNC_JTAG_TDI	JTAG 資料輸入介面
FUNC_JTAG_TMS	JTAG 控制 TAP 狀態機的轉換
FUNC_JTAG_TDO	JTAG 資料輸出介面
FUNC_SPI0_D0	SPI0 資料線 0
FUNC_SPI0_D1	SPI0 資料線 1
FUNC_SPI0_D2	SPI0 資料線 2
FUNC_SPI0_D3	SPI0 資料線 3
FUNC_SPI0_D4	SPI0 資料線 4
FUNC_SPI0_D5	SPI0 資料線 5
FUNC_SPI0_D6	SPI0 資料線 6
FUNC_SPI0_D7	SPI0 資料線 7
FUNC_SPI0_SS0	SPI0 片選信號 0
FUNC_SPI0_SS1	SPI0 片選信號 1
FUNC_SPI0_SS2	SPI0 片選信號 2
FUNC_SPI0_SS3	SPI0 片選信號 3
FUNC_SPI0_ARB	SPI0 仲裁信號
FUNC_SPI0_SCLK	SPI0 時脈
FUNC_UARTHS_RX	UART 高速接收資料介面
FUNC_UARTHS_TX	UART 高速發送資料介面
FUNC_RESV6	保留功能
FUNC_RESV7	保留功能
FUNC_CLK_SPI1	SPI1 時脈
FUNC_CLK_I2C1	I2C1 時脈
FUNC_GPIOHS0	高速 GPIO0
FUNC_GPIOHS1	高速 GPIO1
FUNC_GPIOHS2	高速 GPIO2
FUNC_GPIOHS3	高速 GPIO3

成員名稱	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

成員名稱	描述
FUNC_GPIO7	GPIO7
FUNC_UART1_RX	UART1 接收資料介面
FUNC_UART1_TX	UART1 發送資料介面
FUNC_UART2_RX	UART2 接收資料介面
FUNC_UART2_TX	UART2 發送資料介面
FUNC_UART3_RX	UART3 接收資料介面
FUNC_UART3_TX	UART3 發送資料介面
FUNC_SPI1_D0	SPI1 資料線 0
FUNC_SPI1_D1	SPI1 資料線 1
FUNC_SPI1_D2	SPI1 資料線 2
FUNC_SPI1_D3	SPI1 資料線 3
FUNC_SPI1_D4	SPI1 資料線 4
FUNC_SPI1_D5	SPI1 資料線 5
FUNC_SPI1_D6	SPI1 資料線 6
FUNC_SPI1_D7	SPI1 資料線 7
FUNC_SPI1_SS0	SPI1 片選信號 0
FUNC_SPI1_SS1	SPI1 片選信號 1
FUNC_SPI1_SS2	SPI1 片選信號 2
FUNC_SPI1_SS3	SPI1 片選信號 3
FUNC_SPI1_ARB	SPI1 仲裁信號
FUNC_SPI1_SCLK	SPI1 時脈
FUNC_SPI_SLAVE_D0	SPI 從模式資料線 0
FUNC_SPI_SLAVE_SS	SPI 從模式片選信號
FUNC_SPI_SLAVE_SCLK	SPI 從模式時脈
FUNC_I2S0_MCLK	I2S0 主時脈（系統時脈）
FUNC_I2S0_SCLK	I2S0 串列時脈（位時脈）
FUNC_I2S0_WS	I2S0 幀時脈
FUNC_I2S0_IN_D0	I2S0 串列輸入資料介面 0
FUNC_I2S0_IN_D1	I2S0 串列輸入資料介面 1
FUNC_I2S0_IN_D2	I2S0 串列輸入資料介面 2
FUNC_I2S0_IN_D3	I2S0 串列輸入資料介面 3
FUNC_I2S0_OUT_D0	I2S0 串列輸出資料介面 0
FUNC_I2S0_OUT_D1	I2S0 串列輸出資料介面 1
FUNC_I2S0_OUT_D2	I2S0 串列輸出資料介面 2
FUNC_I2S0_OUT_D3	I2S0 串列輸出資料介面 3

成員名稱	描述
FUNC_I2S1_MCLK	I2S1 主時脈（系統時脈）
FUNC_I2S1_SCLK	I2S1 串列時脈（位時脈）
FUNC_I2S1_WS	I2S1 幀時脈
FUNC_I2S1_IN_D0	I2S1 串列輸入資料介面 0
FUNC_I2S1_IN_D1	I2S1 串列輸入資料介面 1
FUNC_I2S1_IN_D2	I2S1 串列輸入資料介面 2
FUNC_I2S1_IN_D3	I2S1 串列輸入資料介面 3
FUNC_I2S1_OUT_D0	I2S1 串列輸出資料介面 0
FUNC_I2S1_OUT_D1	I2S1 串列輸出資料介面 1
FUNC_I2S1_OUT_D2	I2S1 串列輸出資料介面 2
FUNC_I2S1_OUT_D3	I2S1 串列輸出資料介面 3
FUNC_I2S2_MCLK	I2S2 主時脈（系統時脈）
FUNC_I2S2_SCLK	I2S2 串列時脈（位時脈）
FUNC_I2S2_WS	I2S2 幀時脈
FUNC_I2S2_IN_D0	I2S2 串列輸入資料介面 0
FUNC_I2S2_IN_D1	I2S2 串列輸入資料介面 1
FUNC_I2S2_IN_D2	I2S2 串列輸入資料介面 2
FUNC_I2S2_IN_D3	I2S2 串列輸入資料介面 3
FUNC_I2S2_OUT_D0	I2S2 串列輸出資料介面 0
FUNC_I2S2_OUT_D1	I2S2 串列輸出資料介面 1
FUNC_I2S2_OUT_D2	I2S2 串列輸出資料介面 2
FUNC_I2S2_OUT_D3	I2S2 串列輸出資料介面 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 串列時脈
FUNC_I2C0_SDA	I2C0 串列資料介面
FUNC_I2C1_SCLK	I2C1 串列時脈
FUNC_I2C1_SDA	I2C1 串列資料介面
FUNC_I2C2_SCLK	I2C2 串列時脈
FUNC_I2C2_SDA	I2C2 串列資料介面
FUNC_CMOS_XCLK	DVP 系統時脈

成員名稱	描述
FUNC_CMOS_RST	DVP 系統複位信號
FUNC_CMOS_PWDN	DVP 啟動信號
FUNC_CMOS_VSYNC	DVP 場同步
FUNC_CMOS_HREF	DVP 行參考信號
FUNC_CMOS_PCLK	像素時脈
FUNC_CMOS_D0	像素資料 0
FUNC_CMOS_D1	像素資料 1
FUNC_CMOS_D2	像素資料 2
FUNC_CMOS_D3	像素資料 3
FUNC_CMOS_D4	像素資料 4
FUNC_CMOS_D5	像素資料 5
FUNC_CMOS_D6	像素資料 6
FUNC_CMOS_D7	像素資料 7
FUNC_SCCB_SCLK	SCCB 時脈
FUNC_SCCB_SDA	SCCB 串列資料信號
FUNC_UART1_CTS	UART1 清除發送信號
FUNC_UART1_DSR	UART1 資料裝置準備信號
FUNC_UART1_DCD	UART1 資料載波檢測
FUNC_UART1_RI	UART1 振鈴指示
FUNC_UART1_SIR_IN	UART1 串列紅外輸入信號
FUNC_UART1_DTR	UART1 資料終端準備信號
FUNC_UART1_RTS	UART1 發送請求信號
FUNC_UART1_OUT2	UART1 用戶指定輸出信號 2
FUNC_UART1_OUT1	UART1 用戶指定輸出信號 1
FUNC_UART1_SIR_OUT	UART1 串列紅外輸出信號
FUNC_UART1_BAUD	UART1 時脈
FUNC_UART1_RE	UART1 接收啟動
FUNC_UART1_DE	UART1 發送啟動
FUNC_UART1_RS485_EN	UART1 啟動 RS485
FUNC_UART2_CTS	UART2 清除發送信號
FUNC_UART2_DSR	UART2 資料裝置準備信號
FUNC_UART2_DCD	UART2 資料載波檢測
FUNC_UART2_RI	UART2 振鈴指示
FUNC_UART2_SIR_IN	UART2 串列紅外輸入信號
FUNC_UART2_DTR	UART2 資料終端準備信號

成員名稱	描述
FUNC_UART2_RTS	UART2 發送請求信號
FUNC_UART2_OUT2	UART2 用戶指定輸出信號 2
FUNC_UART2_OUT1	UART2 用戶指定輸出信號 1
FUNC_UART2_SIR_OUT	UART2 串列紅外輸出信號
FUNC_UART2_BAUD	UART2 時脈
FUNC_UART2_RE	UART2 接收啟動
FUNC_UART2_DE	UART2 發送啟動
FUNC_UART2_RS485_EN	UART2 啟動 RS485
FUNC_UART3_CTS	清除發送信號
FUNC_UART3_DSR	資料裝置準備信號
FUNC_UART3_DCD	UART3 資料載波檢測
FUNC_UART3_RI	UART3 振鈴指示
FUNC_UART3_SIR_IN	UART3 串列紅外輸入信號
FUNC_UART3_DTR	UART3 資料終端準備信號
FUNC_UART3_RTS	UART3 發送請求信號
FUNC_UART3_OUT2	UART3 用戶指定輸出信號 2
FUNC_UART3_OUT1	UART3 用戶指定輸出信號 1
FUNC_UART3_SIR_OUT	UART3 串列紅外輸出信號
FUNC_UART3_BAUD	UART3 時脈
FUNC_UART3_RE	UART3 接收啟動
FUNC_UART3_DE	UART3 發送啟動
FUNC_UART3_RS485_EN	UART3 啟動 RS485
FUNC_TIMER0_TOGGLE1	TIMER0 輸出信號 1
FUNC_TIMER0_TOGGLE2	TIMER0 輸出信號 2
FUNC_TIMER0_TOGGLE3	TIMER0 輸出信號 3
FUNC_TIMER0_TOGGLE4	TIMER0 輸出信號 4
FUNC_TIMER1_TOGGLE1	TIMER1 輸出信號 1
FUNC_TIMER1_TOGGLE2	TIMER1 輸出信號 2
FUNC_TIMER1_TOGGLE3	TIMER1 輸出信號 3
FUNC_TIMER1_TOGGLE4	TIMER1 輸出信號 4
FUNC_TIMER2_TOGGLE1	TIMER2 輸出信號 1
FUNC_TIMER2_TOGGLE2	TIMER2 輸出信號 2
FUNC_TIMER2_TOGGLE3	TIMER2 輸出信號 3
FUNC_TIMER2_TOGGLE4	TIMER2 輸出信號 4
FUNC_CLK_SPI2	SPI2 時脈

成員名稱	描述
FUNC_CLK_I2C2	I2C2 時脈
FUNC_INTERNAL0	內部功能 0
FUNC_INTERNAL1	內部功能 1
FUNC_INTERNAL2	內部功能 2
FUNC_INTERNAL3	內部功能 3
FUNC_INTERNAL4	內部功能 4
FUNC_INTERNAL5	內部功能 5
FUNC_INTERNAL6	內部功能 6
FUNC_INTERNAL7	內部功能 7
FUNC_INTERNAL8	內部功能 8
FUNC_INTERNAL9	內部功能 9
FUNC_INTERNAL10	內部功能 10
FUNC_INTERNAL11	內部功能 11
FUNC_INTERNAL12	內部功能 12
FUNC_INTERNAL13	內部功能 13
FUNC_INTERNAL14	內部功能 14
FUNC_INTERNAL15	內部功能 15
FUNC_INTERNAL16	內部功能 16
FUNC_INTERNAL17	內部功能 17
FUNC_CONSTANT	常量
FUNC_INTERNAL18	內部功能 18
FUNC_DEBUG0	調試功能 0
FUNC_DEBUG1	調試功能 1
FUNC_DEBUG2	調試功能 2
FUNC_DEBUG3	調試功能 3
FUNC_DEBUG4	調試功能 4
FUNC_DEBUG5	調試功能 5
FUNC_DEBUG6	調試功能 6
FUNC_DEBUG7	調試功能 7
FUNC_DEBUG8	調試功能 8
FUNC_DEBUG9	調試功能 9
FUNC_DEBUG10	調試功能 10
FUNC_DEBUG11	調試功能 11
FUNC_DEBUG12	調試功能 12
FUNC_DEBUG13	調試功能 13

成員名稱	描述
FUNC_DEBUG14	調試功能 14
FUNC_DEBUG15	調試功能 15
FUNC_DEBUG16	調試功能 16
FUNC_DEBUG17	調試功能 17
FUNC_DEBUG18	調試功能 18
FUNC_DEBUG19	調試功能 19
FUNC_DEBUG20	調試功能 20
FUNC_DEBUG21	調試功能 21
FUNC_DEBUG22	調試功能 22
FUNC_DEBUG23	調試功能 23
FUNC_DEBUG24	調試功能 24
FUNC_DEBUG25	調試功能 25
FUNC_DEBUG26	調試功能 26
FUNC_DEBUG27	調試功能 27
FUNC_DEBUG28	調試功能 28
FUNC_DEBUG29	調試功能 29
FUNC_DEBUG30	調試功能 30
FUNC_DEBUG31	調試功能 31

3.3.2 fpioa_cfg_item_t

3.3.2.1 描述

FPIOA 腳位配置。

3.3.2.2 定義

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

3.3.2.3 成員

成員名稱	描述
number	腳位編號

成員名稱	描述
function	功能編號

3.3.3 fpioa_cfg_t

3.3.3.1 描述

FPIOA 配置。

3.3.3.2 定義

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

3.3.3.3 成員

成員名稱	描述
version	配置版本，必須設為 FPIOA_CFG_VERSION
functions_count	功能配置數量
functions	功能配置列表

3.3.4 sysctl_power_bank_t

3.3.4.1 描述

電源域編號。

3.3.4.2 定義

```
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　成員

成員名稱	描述
SYSCTL_POWER_BANK0	電源域 0，控制 IO0-IO5
SYSCTL_POWER_BANK1	電源域 0，控制 IO6-IO11
SYSCTL_POWER_BANK2	電源域 0，控制 IO12-IO17
SYSCTL_POWER_BANK3	電源域 0，控制 IO18-IO23
SYSCTL_POWER_BANK4	電源域 0，控制 IO24-IO29
SYSCTL_POWER_BANK5	電源域 0，控制 IO30-IO35
SYSCTL_POWER_BANK6	電源域 0，控制 IO36-IO41
SYSCTL_POWER_BANK7	電源域 0，控制 IO42-IO47

3.3.5　sysctl_io_power_mode_t

3.3.5.1　描述

IO 輸出電壓值。

3.3.5.2　定義

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

3.3.5.3　成員

成員名稱	描述
SYSCTL_POWER_V33	設置為 3.3V
SYSCTL_POWER_V18	設置為 1.8V

3.3.6　power_bank_item_t

3.3.6.1　描述

單個電源域配置。

3.3.6.2　定義

```
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　成員

成員名稱	描述
power_bank	電源域編號
io*power*mode	IO 輸出電壓值

3.3.7　power_bank_cfg_t

3.3.7.1　描述

電源域配置。

3.3.7.2　定義

```
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　成員

成員名稱	描述
version	配置版本，必須設為 FPIOA_CFG_VERSION
power*banks*count	電源域配置數量
power_banks	電源域配置列表

3.3.8 pin_cfg_t

3.3.8.1 描述

腳位配置。

3.3.8.2 定義

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

3.3.8.3 成員

成員名稱	描述
version	配置版本，必須設為 FPIOA_CFG_VERSION
set*spi0*dvp_data	是否設置 SPI*0D0-D7 與 DVP*D0-D7 為 SPI0 資料輸出與 DVP 資料輸入

3.3.9 舉例

```
/* 配置 IO6、IO7 的功能分別為 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

系統控制

4.1 概述

系統控制模組提供對操作系統的配置功能。

4.2 功能描述

系統控制模組具有以下功能：

- 設置 CPU 頻率
- 安裝自定義驅動

4.3 API 參考

對應的頭文件 `hal.h`
為用戶提供以下介面：

- `system_set_cpu_frequency`
- `system_install_custom_driver`

4.3.1 system_set_cpu_frequency

4.3.1.1 描述

設置 CPU 頻率。

4.3.1.2　函數原型

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

4.3.1.3　參數

參數名稱	描述	輸入輸出
frequency	要設置的頻率（Hz）	輸入

4.3.1.4　返回值

設置後的實際頻率（Hz）。

4.3.2　system_install_custom_driver

4.3.2.1　描述

安裝自定義驅動。

4.3.2.2　函數原型

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

4.3.2.3　參數

參數名稱	描述	輸入輸出
name	指定訪問該裝置的路徑	輸入
driver	自定義驅動實現	輸入

4.3.2.4　返回值

無。

4.3.3　舉例

```
/* 設置 CPU 頻率為 400MHz */
system_set_cpu_frequency(400000000);
```

4.4　資料類型

相關資料類型、資料結構定義如下：

- driver_base_t：驅動實現基類。
- custom_driver_t：自定義驅動實現。

4.4.1　driver_base_t

4.4.1.1　描述

驅動實現基類。

4.4.1.2　定義

```
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　成員

成員名稱	描述
userdata	用戶資料
install	安裝時被調用
open	打開時被調用
close	關閉時被調用

4.4.2　custom_driver_t

4.4.2.1　描述

自定義驅動實現。

4.4.2.2　定義

```
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　成員

成員名稱	描述
base	驅動實現基類
io_control	收到控制資訊時被調用

第 5 章

可編程中斷控制器（PIC）

5.1 概述

可以將任一外部中斷源單獨分配到每個 CPU 的外部中斷上。這提供了強大的靈活性，能適應不同的應用需求。

5.2 功能描述

PIC 模組具有以下功能：

- 啟用或禁用中斷
- 設置中斷處理程序
- 配置中斷優先順序

5.3 API 參考

對應的頭文件 `hal.h`
為用戶提供以下介面：

- `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

設置 IRQ 是否啟用。

5.3.1.2 函數原型

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

5.3.1.3 參數

參數名稱	描述	輸入輸出
irq	IRQ 編號	輸入
enable	是否啟用	輸入

5.3.1.4 返回值

無。

5.3.2 pic_set_irq_handler

5.3.2.1 描述

設置 IRQ 處理程序。

5.3.2.2 函數原型

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

5.3.2.3 參數

參數名稱	描述	輸入輸出
irq	IRQ 編號	輸入
handler	處理程序	輸入
userdata	處理程序用戶資料	輸入

5.3.2.4　返回值

無。

5.3.3　pic_set_irq_priority

5.3.3.1　描述

設置 IRQ 優先順序。

5.3.3.2　函數原型

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

5.3.3.3　參數

參數名稱	描述	輸入輸出
irq	IRQ 編號	輸入
priority	優先順序	輸入

5.3.3.4　返回值

無。

5.4　資料類型

相關資料類型、資料結構定義如下：

- pic_irq_handler_t: IRQ 處理程序。

5.4.1　pic_irq_handler_t

5.4.1.1　描述

IRQ 處理程序。

5.4.1.2　定義

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

5.4.1.3　參數

參數名稱	描述	輸入輸出
userdata	用戶資料	輸入

第 6 章

直接存儲訪問（DMA）

6.1 概述

直接存儲訪問（Direct Memory Access，DMA）用於在外部裝置與記憶體之間以及記憶體與記憶體之間提供高速資料傳輸。可以在無需任何 CPU 操作的情況下通過 DMA 快速移動資料，從而提高了 CPU 的效率。

6.2 功能描述

DMA 模組具有以下功能：

- 自動選擇一路空閑的 DMA 通道用於傳輸
- 根據源地址和目標地址自動選擇軟體或硬體握手協議
- 支持 1、2、4、8 位元組的元素大小，源和目標大小不必一致
- 非同步或同步傳輸功能
- 循環傳輸功能，常用於刷新屏幕或音頻錄放等場景

6.3 API 參考

對應的頭文件 `hal.h`
為用戶提供以下介面：

- `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

打開一個可用的 DMA 裝置。

6.3.1.2　函數原型

```
handle_t dma_open_free();
```

6.3.1.3　返回值

DMA 裝置句柄。

6.3.2　dma_close

6.3.2.1　描述

關閉 DMA 裝置。

6.3.2.2　函數原型

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

6.3.2.3　參數

參數名稱	描述	輸入輸出
file	DMA 裝置句柄	輸入

6.3.2.4　返回值

無。

6.3.3　dma_set_request_source

6.3.3.1　描述

設置 DMA 請求源。

6.3.3.2　函數原型

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

6.3.3.3　參數

參數名稱	描述	輸入輸出
file	DMA 裝置句柄	輸入
request	請求源編號	輸入

6.3.3.4　返回值

無。

6.3.4　dma_transmit_async

6.3.4.1　描述

進行 DMA 非同步傳輸。

6.3.4.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	DMA 裝置句柄	輸入
src	源地址	輸入
dest	目標地址	輸出
src_inc	源地址是否自增	輸入
dest_inc	目標地址是否自增	輸入
element_size	元素大小（位元組）	輸入
count	元素數量	輸入
burst_size	突發傳輸數量	輸入

參數名稱	描述	輸入輸出
completion_event	傳輸完成事件	輸入

6.3.4.4　返回值

無。

6.3.5　dma_transmit

6.3.5.1　描述

進行 DMA 同步傳輸。

6.3.5.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	DMA 裝置句柄	輸入
src	源地址	輸入
dest	目標地址	輸出
src_inc	源地址是否自增	輸入
dest_inc	目標地址是否自增	輸入
element_size	元素大小（位元組）	輸入
count	元素數量	輸入
burst_size	突發傳輸數量	輸入

6.3.5.4　返回值

無。

6.3.6　dma_loop_async

6.3.6.1　描述

進行 DMA 非同步循環傳輸。

6.3.6.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	DMA 裝置句柄	輸入
srcs	源地址列表	輸入
src_num	源地址數量	輸入
dests	目標地址列表	輸出
dest_num	目標地址數量	輸入
src_inc	源地址是否自增	輸入
dest_inc	目標地址是否自增	輸入
element_size	元素大小（位元組）	輸入
count	元素數量	輸入
burst_size	突發傳輸數量	輸入
stage_completion_handler	階段完成處理程序	輸入
stage_completion_handler_data	階段完成處理程序用戶資料	輸入
completion_event	傳輸完成事件	輸入
stop_signal	停止信號	輸入

註： 階段完成是指單次源到目標 count 個元素的傳輸完成。

6.3.6.4　返回值

無。

6.3.7　舉例

```
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　資料類型

相關資料類型、資料結構定義如下：

- dma_stage_completion_handler_t: DMA 階段完成處理程序。

6.4.1　dma_stage_completion_handler_t

6.4.1.1　描述

DMA 階段完成處理程序。

6.4.1.2　定義

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

6.4.1.3　參數

參數名稱	描述	輸入輸出
userdata	用戶資料	輸入

第 7 章

標準 IO

7.1 概述

標準 IO 模組是訪問外部裝置的基本介面。

7.2 功能描述

標準 IO 模組具有以下功能:

- 根據路徑尋找外部裝置
- 統一的讀寫和控制介面

7.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面:

- io_open
- io_close
- io_read
- io_write
- io_control

7.3.1　io_open

7.3.1.1　描述

打開一個裝置。

7.3.1.2　函數原型

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

7.3.1.3　參數

參數名稱	描述	輸入輸出
name	裝置路徑	輸入

7.3.1.4　返回值

返回值	描述
0	失敗
其他	裝置句柄

7.3.2　io_close

7.3.2.1　描述

關閉一個裝置。

7.3.2.2　函數原型

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

7.3.2.3　參數

參數名稱	描述	輸入輸出
file	裝置句柄	輸入

7.3.2.4　返回值

返回值	描述
0	成功
其他	失敗

7.3.3　io_read

7.3.3.1　描述

從裝置讀取。

7.3.3.2　函數原型

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

7.3.3.3　參數

參數名稱	描述	輸入輸出
file	裝置句柄	輸入
buffer	目標緩衝區	輸出
len	最多讀取的位元組數	輸入

7.3.3.4　返回值

實際讀取的位元組數。

7.3.4　io_write

7.3.4.1　描述

向裝置寫入。

7.3.4.2　函數原型

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

7.3.4.3　參數

參數名稱	描述	輸入輸出
file	裝置句柄	輸入
buffer	源緩衝區	輸入
len	要寫入的位元組數	輸入

7.3.4.4　返回值

返回值	描述
len	成功
其他	失敗

7.3.5　io_control

7.3.5.1　描述

向裝置發送控制資訊。

7.3.5.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	裝置句柄	輸入
control_code	控制碼	輸入
write_buffer	源緩衝區	輸入
write_len	要寫入的位元組數	輸入
read_buffer	目標緩衝區	輸出
read_len	最多讀取的位元組數	輸入

7.3.5.4　返回值

實際讀取的位元組數。

7.3.6　舉例

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

第 8 章

通用非同步收發傳輸器（UART）

8.1 概述

嵌入式應用通常要求一個簡單的並且占用系統資源少的方法來傳輸資料。通用非同步收發傳輸器（UART）即可以滿足這些要求，它能夠靈活地與外部裝置進行全雙工資料交換。

8.2 功能描述

UART 模組具有以下功能：

- 配置 UART 參數
- 自動收取資料到緩衝區

8.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- uart_config

8.3.1 uart_config

8.3.1.1 描述

配置 UART 裝置。

8.3.1.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	UART 裝置句柄	輸入
baud_rate	波特率	輸入
databits	資料位（5-8）	輸入
stopbits	停止位	輸入
parity	校驗位	輸入

8.3.1.4　返回值

無。

8.3.2　舉例

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

uint8_t b = 1;
/* 寫入 1 個位元組 */
io_write(uart, &b, 1);
/* 讀取 1 個位元組 */
while (io_read(uart, &b, 1) != 1);
```

8.4　資料類型

相關資料類型、資料結構定義如下：

- uart_stopbits_t: UART 停止位。
- uart_parity_t: UART 校驗位。

8.4.1　uart_stopbits_t

8.4.1.1　描述

UART 停止位。

8.4.1.2　定義

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

8.4.1.3　成員

成員名稱	描述
UART_STOP_1	1 個停止位
UART_STOP_1_5	1.5 個停止位
UART_STOP_2	2 個停止位

8.4.2　uart_parity_t

8.4.2.1　描述

UART 校驗位。

8.4.2.2　定義

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

8.4.2.3　成員

成員名稱	描述
UART_PARITY_NONE	無校驗位
UART_PARITY_ODD	奇校驗
UART_PARITY_EVEN	偶校驗

第 9 章

通用輸入/輸出（GPIO）

9.1 概述

晶片有 32 個高速 GPIO 和 8 個通用 GPIO。

9.2 功能描述

GPIO 模組具有以下功能：

- 可配置上下拉驅動模式
- 支持正緣、負緣和雙緣觸發

9.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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

獲取 GPIO 腳位數量。

9.3.1.2　函數原型

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

9.3.1.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入

9.3.1.4　返回值

腳位數量。

9.3.2　gpio_set_drive_mode

9.3.2.1　描述

設置 GPIO 腳位驅動模式。

9.3.2.2　函數原型

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

9.3.2.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入
pin	腳位編號	輸入
mode	驅動模式	輸入

9.3.2.4　返回值

無。

9.3.3　gpio_set_pin_edge

9.3.3.1　描述

設置 GPIO 腳位邊緣觸發模式。

註： /dev/gpio1 暫不支持。

9.3.3.2　函數原型

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

9.3.3.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入
pin	腳位編號	輸入
edge	邊緣觸發模式	輸入

9.3.3.4　返回值

無。

9.3.4　gpio_set_on_changed

9.3.4.1　描述

設置 GPIO 腳位邊緣觸發處理程序。

註： /dev/gpio1 暫不支持。

9.3.4.2　函數原型

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

9.3.4.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入

參數名稱	描述	輸入輸出
pin	腳位編號	輸入
callback	處理程序	輸入
userdata	處理程序用戶資料	輸入

9.3.4.4　返回值

無。

9.3.5　gpio_get_pin_value

9.3.5.1　描述

獲取 GPIO 腳位的值。

9.3.5.2　函數原型

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

9.3.5.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入
pin	腳位編號	輸入

9.3.5.4　返回值

GPIO 腳位的值。

9.3.6　gpio_set_pin_value

9.3.6.1　描述

設置 GPIO 腳位的值。

9.3.6.2　函數原型

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

9.3.6.3　參數

參數名稱	描述	輸入輸出
file	GPIO 控制器句柄	輸入
pin	腳位編號	輸入
value	要設置的值	輸入

9.3.6.4　返回值

無。

9.3.7　舉例

```
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　資料類型

相關資料類型、資料結構定義如下：

- gpio_drive_mode_t: GPIO 驅動模式。
- gpio_pin_edge_t: GPIO 邊緣觸發模式。
- gpio_pin_value_t: GPIO 值。
- gpio_on_changed_t: GPIO 邊緣觸發處理程序。

9.4.1　gpio_drive_mode_t

9.4.1.1　描述

GPIO 驅動模式。

9.4.1.2　定義

```
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　成員

成員名稱	描述
GPIO_DM_INPUT	輸入
GPIO_DM_INPUT_PULL_DOWN	輸入下拉
GPIO_DM_INPUT_PULL_UP	輸入上拉
GPIO_DM_OUTPUT	輸出

9.4.2　gpio_pin_edge_t

9.4.2.1　描述

GPIO 邊緣觸發模式。

9.4.2.2　定義

```
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　成員

成員名稱	描述
GPIO_PE_NONE	不觸發
GPIO_PE_FALLING	負緣觸發
GPIO_PE_RISING	正緣觸發
GPIO_PE_BOTH	雙緣觸發

9.4.3　gpio_pin_value_t

9.4.3.1　描述

GPIO 值。

9.4.3.2　定義

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

9.4.3.3　成員

成員名稱	描述
GPIO_PV_LOW	低
GPIO_PV_HIGH	高

9.4.4　gpio_on_changed_t

9.4.4.1　描述

GPIO 邊緣觸發處理程序。

9.4.4.2　定義

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

9.4.4.3　參數

參數名稱	描述	輸入輸出
pin	腳位編號	輸入
userdata	用戶資料	輸入

第 10 章

集成電路內置匯流排（I²C）

10.1 概述

I²C 匯流排用於和多個外部裝置進行通信。多個外部裝置可以共用一個 I²C 匯流排。

10.2 功能描述

I²C 模組具有以下功能：

- 獨立的 I²C 裝置封裝外部裝置相關參數
- 自動處理多裝置匯流排爭用
- 支持從模式

10.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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

註冊並打開一個 I²C 裝置。

10.3.1.2　函數原型

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

10.3.1.3　參數

參數名稱	描述	輸入輸出
file	I²C 控制器句柄	輸入
name	指定訪問該裝置的路徑	輸入
slave_address	從裝置地址	輸入
address_width	從裝置地址寬度	輸入

10.3.1.4　返回值

I²C 裝置句柄。

10.3.2　i2c_dev_set_clock_rate

10.3.2.1　描述

配置 I²C 裝置的時脈速率。

10.3.2.2　函數原型

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

10.3.2.3　參數

參數名稱	描述	輸入輸出
file	I²C 裝置句柄	輸入
clock_rate	期望的時脈速率	輸入

10.3.2.4　返回值

設置後的實際速率。

10.3.3　i2c_dev_transfer_sequential

10.3.3.1　描述

對 I²C 裝置先讀後寫。

10.3.3.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	I²C 裝置句柄	輸入
write_buffer	源緩衝區	輸入
write_len	要寫入的位元組數	輸入
read_buffer	目標緩衝區	輸出
read_len	最多讀取的位元組數	輸入

10.3.3.4　返回值

實際讀取的位元組數。

10.3.4　i2c_config_as_slave

10.3.4.1　描述

配置 I²C 控制器為從模式。

10.3.4.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	I²C 控制器句柄	輸入
slave_address	從裝置地址	輸入
address_width	從裝置地址寬度	輸入
handler	從裝置處理程序	輸入

10.3.4.4　返回值

無。

10.3.5　spi_dev_set_clock_rate

10.3.5.1　描述

配置 I²C 從模式的時脈速率。

10.3.5.2　函數原型

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

10.3.5.3　參數

參數名稱	描述	輸入輸出
file	I²C 控制器句柄	輸入
clock_rate	期望的時脈速率	輸入

10.3.5.4　返回值

設置後的實際速率。

10.3.6　舉例

```
handle_t i2c = io_open("/dev/i2c0");
/* i2c外部裝置地址是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寄存器寫0x01 */
io_write(dev0, data_buf, 2);
/* 從0寄存器讀取1位元組資料 */
i2c_dev_transfer_sequential(dev0, &reg, 1, data_buf, 1);
```

10.4　資料類型

相關資料類型、資料結構定義如下：

- i2c_event_t: I²C 事件。
- i2c_slave_handler_t: I²C 從裝置處理程序。

10.4.1　i2c_event_t

10.4.1.1　描述

I²C 事件。

10.4.1.2　定義

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

10.4.1.3　成員

成員名稱	描述
I2C_EV_START	收到 Start 信號
I2C_EV_RESTART	收到 Restart 信號
I2C_EV_STOP	收到 Stop 信號

10.4.2　i2c_slave_handler_t

10.4.2.1　描述

I²C 從裝置處理程序。

10.4.2.2　定義

```
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　成員

成員名稱	描述
on_receive	收到資料時被調用
on_transmit	需要發送資料時被調用
on_event	發生事件時被調用

第 11 章

集成電路內置音頻匯流排（I2S）

11.1 概述

I2S 標準匯流排定義了三種信號：時脈信號 BCK、聲道選擇信號 WS 和串列資料信號 SD。一個基本的 I2S 資料匯流排有一個主機和一個從機。主機和從機的角色在通信過程中保持不變。I2S 模組包含獨立的發送和接收聲道，能夠保證優良的通信性能。

11.2 功能描述

I2S 模組具有以下功能：

- 根據音頻格式自動配置裝置（支持 16、24、32 位深，44100 採樣率，1 - 4 聲道）
- 可配置為播放或錄音模式
- 自動管理音頻緩衝區

11.3 API 參考

對應的頭文件 `devices.h`
為用戶提供以下介面：

- `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 控制器為輸出模式。

11.3.1.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入
format	音頻格式	輸入
delay_ms	緩衝區長度	輸入
align_mode	對齊模式	輸入
channels_mask	通道掩碼	輸入

11.3.1.4　返回值

無。

11.3.2　i2s_config_as_capture

11.3.2.1　描述

配置 I2S 控制器為捕獲模式。

11.3.2.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入

參數名稱	描述	輸入輸出
format	音頻格式	輸入
delay_ms	緩衝區長度	輸入
align_mode	對齊模式	輸入
channels_mask	通道掩碼	輸入

11.3.2.4　返回值

無。

11.3.3　i2s_get_buffer

11.3.3.1　描述

獲取音頻緩衝區。

11.3.3.2　函數原型

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

11.3.3.3　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入
buffer	緩衝區	輸出
frames	緩衝區幀數	輸出

11.3.3.4　返回值

無。

11.3.4　i2s_release_buffer

11.3.4.1　描述

釋放音頻緩衝區。

11.3.4.2　函數原型

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

11.3.4.3　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入
frames	確認已讀取或寫入的幀數	輸入

11.3.4.4　返回值

無。

11.3.5　i2s_start

11.3.5.1　描述

開始播放或錄音。

11.3.5.2　函數原型

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

11.3.5.3　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入

11.3.5.4　返回值

無。

11.3.6　i2s_stop

11.3.6.1　描述

停止播放或錄音。

11.3.6.2　函數原型

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

11.3.6.3　參數

參數名稱	描述	輸入輸出
file	I2S 控制器句柄	輸入

11.3.6.4　返回值

無。

11.3.7　舉例

```
/* 循環播放 PCM 音頻 */
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　資料類型

相關資料類型、資料結構定義如下：

- audio_format_type_t：音頻格式類型。
- audio_format_t：音頻格式。
- i2s_align_mode_t：I2S 對齊模式。

11.4.1　audio_format_type_t

11.4.1.1　描述

音頻格式類型。

11.4.1.2　定義

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

11.4.1.3　成員

成員名稱	描述
AUDIO_FMT_PCM	PCM

11.4.2　audio_format_t

11.4.2.1　描述

音頻格式。

11.4.2.2　定義

```
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　成員

成員名稱	描述
type	音頻格式類型
bits_per_sample	採樣深度
sample_rate	採樣率

成員名稱	描述
channels	聲道數

11.4.3　i2s_align_mode_t

11.4.3.1　描述

I2S 對齊模式。

11.4.3.2　定義

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

11.4.3.3　成員

成員名稱	描述
I2S_AM_STANDARD	標準模式
I2S_AM_RIGHT	右對齊
I2S_AM_LEFT	左對齊

第 12 章

串列外部裝置介面（SPI）

12.1 概述

SPI 是一種高速的，全雙工，同步的通信匯流排。

12.2 功能描述

SPI 模組具有以下功能：

- 獨立的 SPI 裝置封裝外部裝置相關參數
- 自動處理多裝置匯流排爭用
- 支持標準、雙線、四線、八線模式
- 支持先寫後讀和全雙工讀寫
- 支持發送一串相同的資料幀，常用於清屏、填充存儲扇區等場景

12.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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

註冊並打開一個 SPI 裝置。

12.3.1.2 函數原型

```
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 參數

參數名稱	描述	輸入輸出
file	SPI 控制器句柄	輸入
name	指定訪問該裝置的路徑	輸入
mode	SPI 模式	輸入
frame_format	幀格式	輸入
chip_select_mask	片選掩碼	輸入
data_bit_length	資料位長度	輸入

12.3.1.4 返回值

SPI 裝置句柄。

12.3.2 spi_dev_config_non_standard

12.3.2.1 描述

配置 SPI 裝置的非標準幀格式參數。

12.3.2.2 函數原型

```
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 參數

參數名稱	描述	輸入輸出
file	SPI 裝置句柄	輸入
instruction_length	指令長度	輸入
address_length	地址長度	輸入
wait_cycles	等待周期數	輸入
trans_mode	指令和地址的傳輸模式	輸入

12.3.2.4　返回值

無。

12.3.3　spi_dev_set_clock_rate

12.3.3.1　描述

配置 SPI 裝置的時脈速率。

12.3.3.2　函數原型

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

12.3.3.3　參數

參數名稱	描述	輸入輸出
file	SPI 裝置句柄	輸入
clock_rate	期望的時脈速率	輸入

12.3.3.4　返回值

設置後的實際速率。

12.3.4　spi_dev_transfer_full_duplex

12.3.4.1　描述

對 SPI 裝置進行全雙工傳輸。

註： 僅支持標準幀格式。

12.3.4.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	SPI 裝置句柄	輸入
write_buffer	源緩衝區	輸入
write_len	要寫入的位元組數	輸入
read_buffer	目標緩衝區	輸出
read_len	最多讀取的位元組數	輸入

12.3.4.4　返回值

實際讀取的位元組數。

12.3.5　spi_dev_transfer_sequential

12.3.5.1　描述

對 SPI 裝置進行先寫後讀。

註： 僅支持標準幀格式。

12.3.5.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
file	SPI 裝置句柄	輸入
write_buffer	源緩衝區	輸入
write_len	要寫入的位元組數	輸入
read_buffer	目標緩衝區	輸出
read_len	最多讀取的位元組數	輸入

12.3.5.4　返回值

實際讀取的位元組數。

12.3.6　spi_dev_fill

12.3.6.1　描述

對 SPI 裝置填充一串相同的幀。

註： 僅支持標準幀格式。

12.3.6.2　函數原型

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

12.3.6.3　參數

參數名稱	描述	輸入輸出
file	SPI 裝置句柄	輸入
instruction	指令（標準幀格式下忽略）	輸入
address	地址（標準幀格式下忽略）	輸入
value	幀資料	輸出
count	幀數	輸入

12.3.6.4　返回值

無。

12.3.7　舉例

```
handle_t spi = io_open("/dev/spi0");
/* dev0 工作在 MODE0 模式 標準 SPI 模式 單次發送 8 位資料 使用片選 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 };
/* 發送指令 0x06 向地址 0x010204 發送 0，1，2，3 四個位元組資料 */
io_write(dev0, data_buf, sizeof(data_buf));
/* 發送指令 0x06 地址 0x010204 接收四個位元組的資料 */
spi_dev_transfer_sequential(dev0, data_buf, 4, data_buf, 4);
```

12.4　資料類型

相關資料類型、資料結構定義如下：

- spi_mode_t：SPI 模式。
- spi_frame_format_t：SPI 幀格式。
- spi_inst_addr_trans_mode_t：SPI 指令和地址的傳輸模式。

12.4.1　spi_mode_t

12.4.1.1　描述

SPI 模式。

12.4.1.2　定義

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

12.4.1.3　成員

成員名稱	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 幀格式。

12.4.2.2　定義

```
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　成員

成員名稱	描述
SPI_FF_STANDARD	標準
SPI_FF_DUAL	雙線
SPI_FF_QUAD	四線
SPI_FF_OCTAL	八線（/dev/spi3 不支持）

12.4.3　spi_inst_addr_trans_mode_t

12.4.3.1　描述

SPI 指令和地址的傳輸模式。

12.4.3.2　定義

```
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　成員

成員名稱	描述
SPI_AITM_STANDARD	均使用標準幀格式
SPI_AITM_ADDR_STANDARD	指令使用配置的值，地址使用標準幀格式
SPI_AITM_AS_FRAME_FORMAT	均使用配置的值

第 13 章

數位攝像頭介面（DVP）

13.1 概述

DVP 是攝像頭介面模組，支持把攝像頭輸入圖像資料轉發給 AI 模組或者內部儲存。

13.2 功能描述

DVP 模組具有以下功能：

- 支持 RGB565、RGB422 與單通道 Y 灰度輸入模式
- 支持設置幀中斷
- 支持設置傳輸地址
- 支持同時向兩個地址寫資料（輸出格式分別是 RGB888 與 RGB565）
- 支持丟棄不需要處理的幀

13.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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 的頻率。

13.3.1.2 函數原型

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

13.3.1.3 參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
clock_rate	配置 XCLK 的頻率，如 OV5640 配置為 20MHz	輸入

13.3.1.4 返回值

設置後的實際頻率。

13.3.2 dvp_config

13.3.2.1 描述

配置 DVP 裝置。

13.3.2.2 函數原型

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

13.3.2.3 參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
width	幀寬度	輸入

參數名稱	描述	輸入輸出
height	幀高度	輸入
auto_enable	自動啟用幀處理	輸入

13.3.2.4　返回值

無。

13.3.3　dvp_enable_frame

13.3.3.1　描述

啟用對當前幀的處理。

13.3.3.2　函數原型

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

13.3.3.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入

13.3.3.4　返回值

無。

13.3.4　dvp_get_output_num

13.3.4.1　描述

獲取 DVP 裝置的輸出數目。

13.3.4.2　函數原型

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

13.3.4.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入

13.3.4.4　返回值

輸出數目。

13.3.5　dvp_set_signal

13.3.5.1　描述

設置 DVP 信號狀態。

13.3.5.2　函數原型

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

13.3.5.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
type	信號類型	輸入
value	狀態值	輸入

13.3.5.4　返回值

無。

13.3.6　dvp_set_output_enable

13.3.6.1　描述

設置 DVP 輸出是否啟用。

13.3.6.2　函數原型

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

13.3.6.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
index	輸出索引	輸入
enable	是否啟用	輸入

13.3.6.4　返回值

無。

13.3.7　dvp_set_output_attributes

13.3.7.1　描述

設置 DVP 輸出特性。

13.3.7.2　函數原型

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

13.3.7.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
index	輸出索引	輸入
format	視訊格式	輸入
output_buffer	輸出緩衝	輸出

13.3.7.4　返回值

無。

13.3.8　dvp_set_frame_event_enable

13.3.8.1　描述

設置 DVP 幀事件是否啟用。

13.3.8.2　函數原型

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

13.3.8.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
event	幀事件	輸入
enable	是否啟用	輸入

13.3.8.4　返回值

無。

13.3.9　dvp_set_on_frame_event

13.3.9.1　描述

設置 DVP 幀事件處理程序。

13.3.9.2　函數原型

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

13.3.9.3　參數

參數名稱	描述	輸入輸出
file	DVP 裝置句柄	輸入
handler	處理程序	輸入
userdata	處理程序用戶資料	輸入

13.3.9.4　返回值

無。

13.3.10　舉例

```
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　資料類型

相關資料類型、資料結構定義如下：

- video_format_t：視訊格式。
- dvp_frame_event_t：DVP 幀事件。
- dvp_signal_type_t：DVP 信號類型。
- dvp_on_frame_event_t：DVP 幀事件處理程序。

13.4.1　video_format_t

13.4.1.1　描述

視訊格式。

13.4.1.2　定義

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

13.4.1.3　成員

成員名稱	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 幀事件。

13.4.2.2　定義

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

13.4.2.3　成員

成員名稱	描述
VIDEO_FE_BEGIN	幀開始
VIDEO_FE_END	幀結束

13.4.3　dvp_signal_type_t

13.4.3.1　描述

DVP 信號類型。

13.4.3.2　定義

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

13.4.3.3　成員

成員名稱	描述
DVP_SIG_POWER_DOWN	掉電
DVP_SIG_RESET	複位

13.4.4　dvp_on_frame_event_t

13.4.4.1　描述

TIMER 觸發時的處理程序。

13.4.4.2　定義

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

13.4.4.3　參數

參數名稱	描述	輸入輸出
userdata	用戶資料	輸入

第 14 章

串列攝像機控制匯流排（SCCB）

14.1 概述

SCCB 是一種串列攝像機控制匯流排。

14.2 功能描述

SCCB 模組具有以下功能：

- 獨立的 SCCB 裝置封裝外部裝置相關參數
- 自動處理多裝置匯流排爭用

14.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- sccb_get_device
- sccb_dev_read_byte
- sccb_dev_write_byte

14.3.1 sccb_get_device

14.3.1.1 描述

註冊並打開一個 SCCB 裝置。

14.3.1.2　函數原型

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

14.3.1.3　參數

參數名稱	描述	輸入輸出
file	SCCB 控制器句柄	輸入
name	指定訪問該裝置的路徑	輸入
slave_address	從裝置地址	輸入
reg_address_width	寄存器地址寬度	輸入

14.3.1.4　返回值

SCCB 裝置句柄。

14.3.2　sccb_dev_read_byte

14.3.2.1　描述

從 SCCB 裝置讀取一個位元組。

14.3.2.2　函數原型

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

14.3.2.3　參數

參數名稱	描述	輸入輸出
file	SCCB 裝置句柄	輸入
reg_address	寄存器地址	輸入

14.3.2.4　返回值

讀取的位元組。

14.3.3　sccb_dev_write_byte

14.3.3.1　描述

向 SCCB 裝置寫入一個位元組。

14.3.3.2　函數原型

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

14.3.3.3　參數

參數名稱	描述	輸入輸出
file	SCCB 裝置句柄	輸入
reg_address	寄存器地址	輸入
value	要寫入的位元組	輸入

14.3.3.4　返回值

無。

14.3.4　舉例

```
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

定時器（TIMER）

15.1 概述

TIMER 提供高精度定時功能。

15.2 功能描述

TIMER 模組具有以下功能：

- 啟用或禁用定時器
- 配置定時器觸發間隔
- 配置定時器觸發處理程序

15.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- timer_set_interval
- timer_set_on_tick
- timer_set_enable

15.3.1 timer_set_interval

15.3.1.1 描述

設置 TIMER 觸發間隔。

15.3.1.2　函數原型

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

15.3.1.3　參數

參數名稱	描述	輸入輸出
file	TIMER 裝置句柄	輸入
nanoseconds	間隔（納秒）	輸入

15.3.1.4　返回值

實際的觸發間隔（納秒）。

15.3.2　timer_set_on_tick

15.3.2.1　描述

設置 TIMER 觸發時的處理程序。

15.3.2.2　函數原型

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

15.3.2.3　參數

參數名稱	描述	輸入輸出
file	TIMER 裝置句柄	輸入
on_tick	處理程序	輸入
userdata	處理程序用戶資料	輸入

15.3.2.4　返回值

無。

15.3.3　timer_set_enable

15.3.3.1　描述

設置 TIMER 是否啟用。

15.3.3.2　函數原型

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

15.3.3.3　參數

參數名稱	描述	輸入輸出
file	TIMER 裝置句柄	輸入
enable	是否啟用	輸入

15.3.3.4　返回值

無。

15.3.4　舉例

```
/* 定時器0  定時 1 秒列印 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　資料類型

相關資料類型、資料結構定義如下：

- timer_on_tick_t: TIMER 觸發時的處理程序。

15.4.1　timer_on_tick_t

15.4.1.1　描述

TIMER 觸發時的處理程序。

15.4.1.2　定義

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

15.4.1.3　參數

參數名稱	描述	輸入輸出
userdata	用戶資料	輸入

第 16 章

脈衝寬度調製器（PWM）

16.1 概述

PWM 用於控制脈衝輸出的占空比。

16.2 功能描述

PWM 模組具有以下功能：

- 配置 PWM 輸出頻率
- 配置 PWM 每個腳位的輸出占空比

16.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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

獲取 PWM 腳位數量。

16.3.1.2　函數原型

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

16.3.1.3　參數

參數名稱	描述	輸入輸出
file	PWM 裝置句柄	輸入

16.3.1.4　返回值

PWM 腳位數量。

16.3.2　pwm_set_frequency

16.3.2.1　描述

設置 PWM 頻率。

16.3.2.2　函數原型

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

16.3.2.3　參數

參數名稱	描述	輸入輸出
file	PWM 裝置句柄	輸入
frequency	期望的頻率（Hz）	輸入

16.3.2.4　返回值

設置後實際的頻率（Hz）。

16.3.3　pwm_set_active_duty_cycle_percentage

16.3.3.1　描述

設置 PWM 腳位占空比。

16.3.3.2　函數原型

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

16.3.3.3　參數

參數名稱	描述	輸入輸出
file	PWM 裝置句柄	輸入
pin	腳位編號	輸入
duty_cycle_percentage	期望的占空比	輸入

16.3.3.4　返回值

設置後實際的占空比。

16.3.4　pwm_set_enable

16.3.4.1　描述

設置 PWM 腳位是否啟用。

16.3.4.2　函數原型

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

16.3.4.3　參數

參數名稱	描述	輸入輸出
file	PWM 裝置句柄	輸入
pin	腳位編號	輸入
enable	是否啟用	輸入

16.3.4.4　返回值

設置後實際的占空比。

16.3.5　舉例

```
/* pwm0 pin0 輸出 200KHZ 占空比為 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

看門狗定時器（WDT）

17.1 概述

WDT 提供系統出錯或無響應時的恢復功能。

17.2 功能描述

WDT 模組具有以下功能：

- 配置超時時間
- 手動重啟計時
- 配置為超時後複位或進入中斷
- 進入中斷後清除中斷可取消複位，否則等待第二次超時後複位

17.3 API 參考

對應的頭文件 devices.h
為用戶提供以下介面：

- 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

設置 WDT 響應模式。

17.3.1.2　函數原型

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

17.3.1.3　參數

參數名稱	描述	輸入輸出
file	WDT 裝置句柄	輸入
mode	響應模式	輸入

17.3.1.4　返回值

無。

17.3.2　wdt_set_timeout

17.3.2.1　描述

設置 WDT 超時時間。

17.3.2.2　函數原型

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

17.3.2.3　參數

參數名稱	描述	輸入輸出
file	WDT 裝置句柄	輸入
nanoseconds	期望的超時時間（納秒）	輸入

17.3.2.4　返回值

設置後實際的超時時間（納秒）。

17.3.3　wdt_set_on_timeout

17.3.3.1　描述

設置 WDT 超時處理程序。

17.3.3.2　函數原型

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

17.3.3.3　參數

參數名稱	描述	輸入輸出
file	WDT 裝置句柄	輸入
handler	處理程序	輸入
userdata	處理程序用戶資料	輸入

17.3.3.4　返回值

無。

17.3.4　wdt_restart_counter

17.3.4.1　描述

使 WDT 重新開始計數。

17.3.4.2　函數原型

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

17.3.4.3　參數

參數名稱	描述	輸入輸出
file	WDT 裝置句柄	輸入

17.3.4.4　返回值

無。

17.3.5　wdt_set_enable

17.3.5.1　描述

設置 WDT 是否啟用。

17.3.5.2　函數原型

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

17.3.5.3　參數

參數名稱	描述	輸入輸出
file	WDT 裝置句柄	輸入
enable	是否啟用	輸入

17.3.5.4　返回值

無。

17.3.6　舉例

```
/* 2 秒後進入看門狗中斷函數列印 Timeout，再過 2 秒複位 */
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　資料類型

相關資料類型、資料結構定義如下：

- wdt_response_mode_t：WDT 響應模式。

- wdt_on_timeout_t: WDT 超時處理程序。

17.4.1　wdt_response_mode_t

17.4.1.1　描述

WDT 響應模式。

17.4.1.2　定義

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

17.4.1.3　成員

成員名稱	描述
WDT_RESP_RESET	超時後複位系統
WDT_RESP_INTERRUPT	超時後進入中斷，再次超時複位系統

17.4.2　wdt_on_timeout_t

17.4.2.1　描述

WDT 超時處理程序。

17.4.2.2　定義

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

17.4.2.3　參數

參數名稱	描述	輸入輸出
userdata	用戶資料	輸入

17.4.2.4　返回值

返回值	描述
0	不清除中斷，系統將複位
1	清除中斷，系統不複位

第 18 章

快速傅立葉變換加速器（FFT）

18.1 概述

FFT 模組是用硬體的方式來實現 FFT 的基 2 時分運算加速。

18.2 功能描述

目前該模組可以支持 64 點、128 點、256 點以及 512 點的 FFT 以及 IFFT。在 FFT 內部有兩塊大小為 512 * 32 bit 的 SRAM，在配置完成後 FFT 會向 DMA 發送 TX 請求，將 DMA 送來的送據放到其中的一塊 SRAM 中去，直到滿足當前 FFT 運算所需要的資料量並開始 FFT 運算，蝶形運算單元從包含有有效資料的 SRAM 中讀出資料，運算結束後將資料寫到另外一塊 SRAM 中去，下次蝶形運算再從剛寫入的 SRAM 中讀出資料，運算結束後並寫入另外一塊 SRAM，如此反覆交替進行直到完成整個 FFT 運算。

18.3 API 參考

對應的頭文件 `fft.h`
為用戶提供以下介面:

- `fft_complex_uint16`

18.3.1 fft_complex_uint16

18.3.1.1 描述

FFT 運算。

18.3.1.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
shift	FFT 模組 16 位寄存器導致資料溢出 (-32768~32767)，FFT 變換有 9 層，shift 決定哪一層需要移位操作 (如 0x1ff 表示 9 層都做移位操作；0x03 表示第第一層與第二層做移位操作)，防止溢出。如果移位了，則變換後的幅值不是正常 FFT 變換的幅值，對應關係可以參考 fft_test 測試 demo 程序。包含了求解頻率點、相位、幅值的示例	輸入
direction	FFT 正變換或是逆變換	輸入
input	輸入的資料序列，格式為 RIRI..，實部與虛部的精度都為 16 bit	輸入
point_num	待運算的資料點數，只能為 512/256/128/64	輸入
output	運算後結果。格式為 RIRI.. ，實部與虛部的精度都為 16 bit	輸出

18.3.2　舉例

```
#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　資料類型

相關資料類型、資料結構定義如下：

- fft_data_t：FFT 運算傳入的資料格式。
- fft_direction_t：FFT 運算模式。

18.4.1　fft_data_t

18.4.1.1　描述

FFT 運算傳入的資料格式。

18.4.1.2　定義

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

18.4.1.3　成員

成員名稱	描述
I1	第一個資料的虛部
R1	第一個資料的實部
I2	第二個資料的虛部
R2	第二個資料的實部

18.4.2　fft_direction_t

18.4.2.1　描述

FFT 運算模式

18.4.2.2　定義

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

18.4.2.3　成員

成員名稱	描述
FFT_DIR_BACKWARD	FFT 逆變換
FFT_DIR_FORWARD	FFT 正變換

第 19 章

安全散列演算法加速器（SHA256）

19.1 概述

SHA256 模組是用硬體的方式來實現 SHA256 的時分運算加速。

19.2 功能描述

- 支持 SHA-256 的計算

19.3 API 參考

對應的頭文件 sha256.h

為用戶提供以下介面：

- sha256_hard_calculate

19.3.1 sha256_hard_calculate

19.3.1.1 描述

對資料進行 SHA256 計算

19.3.1.2 函數原型

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

19.3.1.3　參數

參數名稱	描述	輸入輸出
input	待 SHA256 計算的資料	輸入
input_len	待 SHA256 計算資料的長度	輸入
output	存放 SHA256 計算的結果，需保證傳入這個 buffer 的大小為 32 byte	輸出

19.3.2　舉例

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

第 20 章

高級加密加速器（AES）

20.1 概述

AES 模組是用硬體的方式來實現 AES 的時分運算加速。

20.2 功能描述

K210 內置 AES(高級加密加速器)，相對於軟體可以極大的提高 AES 運算速度。AES 加速器支持多種加密/解密模式 (ECB,CBC,GCM)，多種長度的 KEY(128,192,256) 的運算。

20.3 API 參考

對應的頭文件 aes.h
為用戶提供以下介面:

- 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 加密運算，當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.1.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-128 加密的密鑰	輸入
input_data	AES-ECB-128 待加密的明文資料	輸入
input_len	AES-ECB-128 待加密明文資料的長度	輸入
output_data	AES-ECB-128 加密運算後的結果存放在這個 buffer。這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.1.4　返回值

無。

20.3.2　aes_ecb128_hard_decrypt

20.3.2.1　描述

AES-ECB-128 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.2.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-128 解密的密鑰	輸入
input_data	AES-ECB-128 待解密的密文資料	輸入
input_len	AES-ECB-128 待解密密文資料的長度	輸入
output_data	AES-ECB-128 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.2.4　返回值

無。

20.3.3　aes_ecb192_hard_encrypt

20.3.3.1　描述

AES-ECB-192 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.3.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-192 加密的密鑰	輸入
input_data	AES-ECB-192 待加密的明文資料	輸入
input_len	AES-ECB-192 待加密明文資料的長度	輸入
output_data	AES-ECB-192 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.3.4　返回值

無。

20.3.4　aes_ecb192_hard_decrypt

20.3.4.1　描述

AES-ECB-192 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.4.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-192 解密的密鑰	輸入
input_data	AES-ECB-192 待解密的密文資料	輸入
input_len	AES-ECB-192 待解密密文資料的長度	輸入

參數名稱	描述	輸入輸出
output_data	AES-ECB-192 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.4.4　返回值

無。

20.3.5　aes_ecb256_hard_encrypt

20.3.5.1　描述

AES-ECB-256 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.5.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-256 加密的密鑰	輸入
input_data	AES-ECB-256 待加密的明文資料	輸入
input_len	AES-ECB-256 待加密明文資料的長度	輸入
output_data	AES-ECB-256 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.5.4　返回值

無。

20.3.6　aes_ecb256_hard_decrypt

20.3.6.1　描述

AES-ECB-256 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.6.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
input_key	AES-ECB-256 解密的密鑰	輸入
input_data	AES-ECB-256 待解密的密文資料	輸入
input_len	AES-ECB-256 待解密密文資料的長度	輸入
output_data	AES-ECB-256 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.6.4　返回值

無。

20.3.7　aes_cbc128_hard_encrypt

20.3.7.1　描述

AES-CBC-128 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.7.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-128 加密計算的結構體，包含加密密鑰與偏移向量	輸入
input_data	AES-CBC-128 待加密的明文資料	輸入
input_len	AES-CBC-128 待加密明文資料的長度	輸入
output_data	AES-CBC-128 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.7.4　返回值

無。

20.3.8　aes_cbc128_hard_decrypt

20.3.8.1　描述

AES-CBC-128 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.8.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-128 解密計算的結構體，包含解密密鑰與偏移向量	輸入

參數名稱	描述	輸入輸出
input_data	AES-CBC-128 待解密的密文資料	輸入
input_len	AES-CBC-128 待解密密文資料的長度	輸入
output_data	AES-CBC-128 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.8.4　返回值

無。

20.3.9　aes_cbc192_hard_encrypt

20.3.9.1　描述

AES-CBC-192 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.9.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-192 加密計算的結構體，包含加密密鑰與偏移向量	輸入
input_data	AES-CBC-192 待加密的明文資料	輸入
input_len	AES-CBC-192 待加密明文資料的長度	輸入

參數名稱	描述	輸入輸出
output_data	AES-CBC-192 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.9.4　返回值

無。

20.3.10　aes_cbc192_hard_decrypt

20.3.10.1　描述

AES-CBC-192 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.10.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-192 解密計算的結構體，包含解密密鑰與偏移向量	輸入
input_data	AES-CBC-192 待解密的密文資料	輸入
input_len	AES-CBC-192 待解密密文資料的長度	輸入
output_data	AES-CBC-192 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.10.4　返回值

無。

20.3.11　aes_cbc256_hard_encrypt

20.3.11.1　描述

AES-CBC-256 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.11.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-256 加密計算的結構體，包含加密密鑰與偏移向量	輸入
input_data	AES-CBC-256 待加密的明文資料	輸入
input_len	AES-CBC-256 待加密明文資料的長度	輸入
output_data	AES-CBC-256 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.11.4　返回值

無。

20.3.12　aes_cbc256_hard_decrypt

20.3.12.1　描述

AES-CBC-256 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.12.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-CBC-256 解密計算的結構體，包含解密密鑰與偏移向量	輸入
input_data	AES-CBC-256 待解密的密文資料	輸入
input_len	AES-CBC-256 待解密密文資料的長度	輸入
output_data	AES-CBC-256 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 16bytes 的整數倍	輸出

20.3.12.4　返回值

無。

20.3.13　aes_gcm128_hard_encrypt

20.3.13.1　描述

AES-GCM-128 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.13.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-128 加密計算的結構體，包含加密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-128 待加密的明文資料	輸入
input_len	AES-GCM-128 待加密明文資料的長度	輸入
output_data	AES-GCM-128 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-128 加密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.13.4　返回值

無。

20.3.14　aes_gcm128_hard_decrypt

20.3.14.1　描述

AES-GCM-128 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.14.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-128 解密計算的結構體，包含解密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-128 待解密的密文資料	輸入
input_len	AES-GCM-128 待解密密文資料的長度。	輸入
output_data	AES-GCM-128 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-128 解密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.14.4　返回值

無。

20.3.15　aes_gcm192_hard_encrypt

20.3.15.1　描述

AES-GCM-192 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.15.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-192 加密計算的結構體，包含加密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-192 待加密的明文資料	輸入
input_len	AES-GCM-192 待加密明文資料的長度。	輸入
output_data	AES-GCM-192 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-192 加密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.15.4　返回值

無。

20.3.16　aes_gcm192_hard_decrypt

20.3.16.1　描述

AES-GCM-192 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.16.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-192 解密計算的結構體，包含解密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-192 待解密的密文資料	輸入
input_len	AES-GCM-192 待解密密文資料的長度。	輸入
output_data	AES-GCM-192 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-192 解密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.16.4　返回值

無。

20.3.17　aes_gcm256_hard_encrypt

20.3.17.1　描述

AES-GCM-256 加密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.17.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-256 加密計算的結構體，包含加密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-256 待加密的明文資料	輸入
input_len	AES-GCM-256 待加密明文資料的長度。	輸入
output_data	AES-GCM-256 加密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-256 加密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.17.4　返回值

無。

20.3.18　aes_gcm256_hard_decrypt

20.3.18.1　描述

AES-GCM-256 解密運算。當加密資料量小於等於 896bytes 時會使用 cpu 來傳輸資料，大於 896bytes 時，會使用 dma 來傳輸資料，從而提高計算的效率。

20.3.18.2　函數原型

```
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　參數

參數名稱	描述	輸入輸出
context	AES-GCM-256 解密計算的結構體，包含解密密鑰/偏移向量/aad/aad 長度	輸入
input_data	AES-GCM-256 待解密的密文資料	輸入
input_len	AES-GCM-256 待解密密文資料的長度。	輸入
output_data	AES-GCM-256 解密運算後的結果存放在這個 buffer。 這個 buffer 大小需要至少為 4bytes 的整數倍， 因為 DMA 的傳輸資料的最小粒度為 4bytes。	輸出
gcm_tag	AES-GCM-256 解密運算後的 tag 存放在這個 buffer。 這個 buffer 的大小需要保證為 16bytes	輸出

20.3.18.4　返回值

無。

20.3.19　舉例

```
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　資料類型

相關資料類型、資料結構定義如下：

- 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　定義

```
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　成員

成員名稱	描述
AES_ECB	ECB 加密/解密
AES_CBC	CBC 加密/解密
AES_GCM	GCM 加密/解密