NuMicro™ M051 系列驱动参考指南
V1.00.001

发布日期: 8. 2010

Support Chips:

M051 系列

Support Platforms:

Nuvoton

The information in this document is subject to change without notice.

The Nuvoton Technology Corp. shall not be liable for technical or editorial errors or omissions contained herein; nor for incidental or consequential damages resulting from the furnishing, performance, or use of this material.

Table of Contents

1. 概述

1.1. 文档结构

本文档是 NuMicro™ M051 系列驱动参考手册。系统级软件开发人员可以使用 NuMicro™ M051 系列驱动来代替直接使用寄存器的编程方式进行快速的应用软件开发，这可以大大减少总的开发时间。在本文档中，对于每一个驱动应用接口，会提供一个关于该驱动应用接口的描述，使用和示例代码。完整的驱动例程和驱动源码在 NuMicro™ M051 系列的 BSP（板级支持包）里。

本文档分为若干个章节，第一章是概述。第二章到第十一章是详细的驱动描述，包括：System 驱动，UART 驱动，Timer 驱动，GPIO 驱动，ADC 驱动，SPI 驱动，I2C 驱动，PWM 驱动，FMC 驱动，和 EBI 驱动。

附录是 NuMicro™ M051 系列选型指导和产品特性表。

1.2. 相关文档

其他一些相关的信息，用户可以在我们的网站上找到如下文档：

- NuMicro™ M051 series Technical Reference Manual (TRM)

1.3. 缩略语和术语

ADC	模数转换器
AHB	增强型高性能总线
AMBA	增强型微控制器总线架构
APB	增强型外围设备总线
BOD	欠压检测
EBI	外部总线接口
FIFO	先进先出
FMC	Flash 存储控制器
GPIO	通用输入/输出
I2C	内部集成电路
LVR	低压复位

PDID	产品设备 ID
PLL	锁相环
POR	上电复位
PWM	脉宽调制
SPI	串行外围设备接口
UART	通用异步收/发器

1.4. 数据类型定义

在我们的驱动中所有的基本数据类型定义遵循 ANSI C 的定义并且和 ARM CMSIS（Cortex-M 软件接口标准）兼容。功能相关的枚举数据类型在各个相应的章节中定义。基本数据类型定义如下表。

类型	定义	描述
int8_t	singed char	8 位有符号整数
int16_t	signed short	16 位有符号整数
int32_t	signed int	32 位有符号整数
uint8_t	unsigned char	8 位无符号整数
uint16_t	unsigned short	16 位无符号整数
uint32_t	unsigned int	32 位无符号整数

2. SYS 驱动

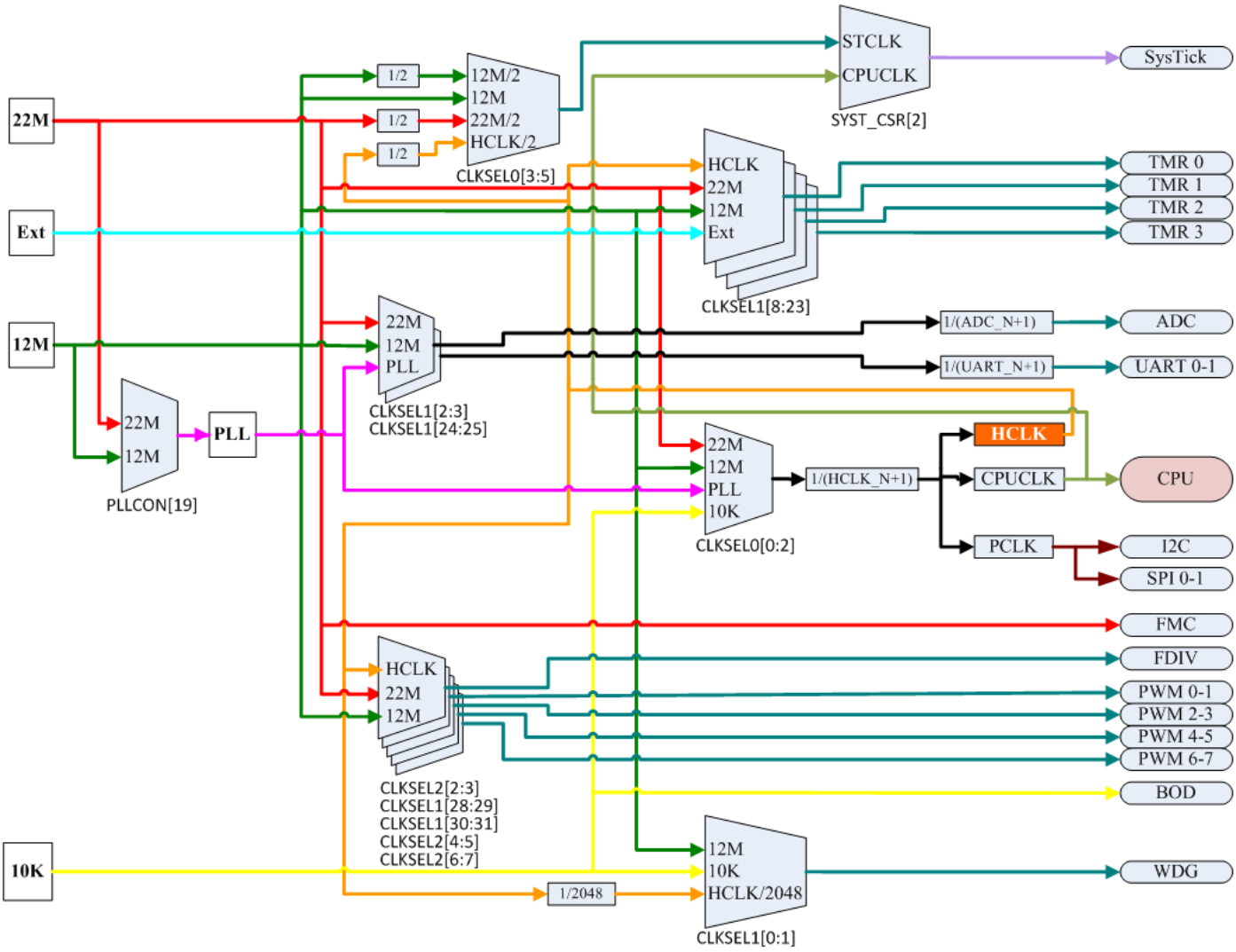
2.1. 介绍

系统管理和时钟控制模块包含下面的功能：

- 产品设备 ID
- 系统管理寄存器，可用于芯片和各个功能模块复位
- Brown-Out 和芯片各种其它的控制
- 时钟发生器
- 系统时钟和外设时钟
- Power down 模式

2. 2. 时钟模块图

时钟模块图给出了整个芯片所有相关的时钟，包括系统时钟（CPU 时钟，HCLK 和 PCLK）和所有的外设时钟。在这里，12M 表示外部晶振时钟源，与一个 12M 的晶体振荡器相连。22M 表示内部 22MHz 的 RC 时钟源，其频率为 22.1184M，偏差为 1%。32K 表示外部 32768Hz 的晶体振荡器，用于实时时钟（RTC）。10K 表示内部 10KHz 的 RC 时钟源，其偏差为 30%。



2.3. 类型定义

E_SYS_IP_RST

枚举标识符	值	描述
E_SYS_GPIO_RST	1	GPIO 复位
E_SYS_TMR0_RST	2	Timer0 复位
E_SYS_TMR1_RST	3	Timer1 复位
E_SYS_TMR2_RST	4	Timer2 复位
E_SYS_TMR3_RST	5	Timer3 复位
E_SYS_I2C_RST	8	I2C 复位
E_SYS_SPI0_RST	12	SPI0 复位
E_SYS_SPI1_RST	13	SPI1 复位
E_SYS_UART0_RST	16	UART0 复位
E_SYS_UART1_RST	17	UART1 复位
E_SYS_PWM03_RST	20	PWM0~3 复位
E_SYS_PWM47_RST	21	PWM4~7 复位
E_SYS_ADC_RST	28	ADC 复位
E_SYS_EBI_RST	32	EBI 复位

E_SYS_IP_CLK

枚举标识符	值	描述
E_SYS_WDT_CLK	0	看门狗定时器时钟使能控制
E_SYS_TMR0_CLK	2	定时器 0 时钟使能控制
E_SYS_TMR1_CLK	3	定时器 1 时钟使能控制
E_SYS_TMR2_CLK	4	定时器 2 时钟使能控制
E_SYS_TMR3_CLK	5	定时器 3 时钟使能控制
E_SYS_FDIV_CLK	6	时钟分频器时钟使能控制
E_SYS_I2C_CLK	8	I2C 时钟使能控制
E_SYS_SPI0_CLK	12	SPI0 时钟使能控制
E_SYS_SPI1_CLK	13	SPI1 时钟使能控制
E_SYS_UART0_CLK	16	UART0 时钟使能控制
E_SYS_UART1_CLK	17	UART1 时钟使能控制
E_SYS_PWM01_CLK	20	PWM01 时钟使能控制
E_SYS_PWM23_CLK	21	PWM23 时钟使能控制
E_SYS_PWM45_CLK	22	PWM45 时钟使能控制
E_SYS_PWM67_CLK	23	PWM67 时钟使能控制
E_SYS_ADC_CLK	28	ADC 时钟使能控制
E_SYS_ISP_CLK	34	Flash ISP 控制器时钟使能控制
E_SYS_EBI_CLK	35	EBI 时钟使能控制

E_SYS_PLL_CLKSRC

枚举标识符	值	描述
E_SYS_EXTERNAL_12M	0	PLL 源时钟来自外部 12M Hz
E_SYS_INTERNAL_22M	1	PLL 源时钟来自内部 22MHz

E_SYS_IP_DIV

枚举标识符	值	描述
E_SYS_ADC_DIV	0	ADC 源时钟分频器设定
E_SYS_UART_DIV	1	UART 源时钟分频器设定
E_SYS_HCLK_DIV	2	HCLK 源时钟分频器设定

E_SYS_IP_CLKSRC

枚举标识符	值	描述
E_SYS_WDT_CLKSRC	0	看门狗定时器时钟源设定
E_SYS_ADC_CLKSRC	1	ADC 时钟源设定
E_SYS_TMR0_CLKSRC	2	定时器 0 时钟源设定
E_SYS_TMR1_CLKSRC	3	定时器 1 时钟源设定
E_SYS_TMR2_CLKSRC	4	定时器 2 时钟源设定
E_SYS_TMR3_CLKSRC	5	定时器 3 时钟源设定
E_SYS_UART_CLKSRC	6	UART 时钟源设定
E_SYS_PWM01_CLKSRC	7	PWM01 时钟源设定
E_SYS_PWM23_CLKSRC	8	PWM23 时钟源设定
E_SYS_FRQDIV_CLKSRC	10	频率分频器输出时钟源设定
E_SYS_PWM45_CLKSRC	11	PWM45 时钟源设定
E_SYS_PWM67_CLKSRC	12	PWM67 时钟源设定

E_SYS_CHIP_CLKSRC

枚举标识符	值	描述
E_SYS_XTL12M	0	选择外部 12M Crystal
E_SYS_OSC22M	1	选择内部 22M Oscillator
E_SYS_OSC10K	2	选择内部 10K Oscillator
E_SYS_PLL	3	选择 PLL 时钟

E_SYS_PD_TYPE

枚举标识符	值	描述
E_SYS_IMMEDIATE	0	立即进入 Power Down 模式
E_SYS_WAIT_FOR_CPU	1	等待 CPU Sleep 命令进入 Power Down 模式

2.4. 函数

DrvSYS_ReadProductID

原型

uint32_t DrvSYS_ReadProductID (void);

描述

读取产品设备 ID。产品设备 ID 取决于芯片的 part number。详细信息请参考附录 PDID 表。

参数

无

头文件

Driver/DrvSYS.h

返回值

产品设备 ID

示例

```
uint32_t u32data;
u32data = DrvSYS_ReadProductID ( );          /* Read Product Device ID */
```

DrvSYS_GetResetSource

原型

uint32_t DrvSYS_GetResetSource (void);

描述

辨别最后一次”复位信号”的出处。详细的对应于各个复位源的位在 TRM 的‘RSTSRC’寄存器中给出。

Bit 0	上电复位
Bit 1	RESET引脚
Bit 2	看门狗定时器
Bit 3	低电压复位
Bit 4	欠压检测复位
Bit 5	Cortex-M0内核复位
Bit 6	保留

Bit 7	CPU复位

参数

无

头文件

Driver/DrvSYS.h

返回值

RSTSRC 寄存器的值

示例

```
uint32_t u32data;
u32data = DrvSYS_GetResetSource ( );              /* Get reset source from last operation */
```

DrvSYS_ClearResetSource

原型

uint32_t DrvSYS_ClearResetSource (uint32_t u32Src);

描述

写 1 清除复位源。

参数

u32Src [in]

复位源的相应位

头文件

Driver/DrvSYS.h

返回值

0　　成功

示例

```
DrvSYS_ClearResetSource (1 << 3);   /* Clear Bit 3 (Low Voltage Reset) */
```

DrvSYS_ResetIP

原型

void DrvSYS_ResetIP(E_SYS_IP_RST eIpRst);

描述

复位 IP, 包括 GPIO, Timer0, Timer1, Timer2, Timer3, I2C, SPI0, SPI1, UART0, UART1, PWM03, PWM47, ADC 和 EBI。

参数

eIpRst [in]

要复位的 IP 的枚举值，参考 2.3 节 E_SYS_IP_RST 的定义。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_ResetIP (E_SYS_I2C_RST);     /* Reset I2C */
DrvSYS_ResetIP (E_SYS_SPI0_RST);    /* Reset SPI0 */
DrvSYS_ResetIP (E_SYS_UART0_RST);   /* Reset UART0 */
```

DrvSYS_ResetCPU

原型

```
void   DrvSYS_ResetCPU (void);
```

描述

复位 CPU。软件会置位 CPU_RST(IPRSTC1[1])来复位 Cortex-M0 CPU 内核和 Flash 存储器控制其(FMC)。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_ResetCPU ( );    /* Reset CPU and FMC */
```

DrvSYS_ResetChip

原型

void DrvSYS_ResetChip (void);

描述

复位整个芯片，包括 Cortex-M0 CPU 内核和所有外围设备。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_ResetChip();      /* Reset Whole Chip */
```

DrvSYS_SelectBODVolt

原型

void DrvSYS_SelectBODVolt (uint8_t u8Volt);

描述

选择 Brown-Out 门限电压。

参数

u8Volt [in]

3: 4.5V, 2: 3.8V, 1: 2.7V, 0: 2.2V

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_SelectBODVolt (0);   /* Set Brown-Out Detector voltage 2.2V */
DrvSYS_SelectBODVolt (1);   /* Set Brown-Out Detector voltage 2.7V */
DrvSYS_SelectBODVolt (2);   /* Set Brown-Out Detector voltage 3.8V */
```

DrvSYS_SetBODFunction

原型

void DrvSYS_SetBODFunction (int32_t i32Enable, int32_t i32Flag, BOD_CALLBACK bodcallbackFn);

描述

使能欠压检测并选择 Brown-Out 复位功能或 Brown-Out 中断功能。如果选择 Brown-Out 中断功能，该函数会安装 BOD 中断处理函数的回调函数。当 AVDD 引脚的电压值低于所选择的 Brown-Out 的门限电压时，Brown-Out 检测器会复位芯片或者触发一个中断。用户可以使用 DrvSYS_SelectBODVolt ( ) 函数取选择 Brown-Out 的门限电压。

参数

i32Enable [in]

1：使能，0：禁止

i32Flag [in]

1：使能 Brown-Out 复位功能， 0：使能 Brown-Out 中断功能

bodcallbackFn [in]

当中断功能使能时，安装 Brown-Out 的回调函数。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Enable Brown-Out Detector , select Brown-Out Interrupt function and install callback function 'BOD_CallbackFn' */
DrvSYS_SetBODFunction (1, 0, BOD_CallbackFn);
/* Enable Brown-Out Detector and select Brown-Out reset function */
DrvSYS_SetBODFunction (1, 1, NULL);
/* Disable Brown-Out Detector */
DrvSYS_SetBODFunction (0, 0, NULL);
```

DrvSYS_EnableBODLowPowerMode

原型

void DrvSYS_EnableBODLowPowerMode (void);

描述

使能 Brow-out 检测器 low power 模式。在 normal 模式，Brown-Out 检测器消耗大约 100uA 电能，在 low power 模式，电能消耗能减少到大约是 normal 模式下的 10%，但是会增大 Brown-Out 检测器的反应时间。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_EnableBODLowPowerMode ( );   /* Enable Brown-Out low power mode */
```

DrvSYS_DisableBODLowPowerMode

原型

```
void   DrvSYS_DisableBODLowPowerMode (void);
```

描述

禁止 Brown-out 检测器 low power 模式。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_DisableBODLowPowerMode ( );   /* Disable Brown-Out low power mode */
```

DrvSYS_EnableLowVoltReset

原型

```
void   DrvSYS_EnableLowVoltReset (void);
```

描述

当输入电压低于 LVR 电路电压时，使能低压复位功能复位芯片。典型的低压门限值是 2.0V，LVR 的门限电压特性在 TRM 的电气特性章节中给出。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

DrvSYS_EnableLowVoltRst ( ); /* Enable low voltage reset function */

DrvSYS_DisableLowVoltReset

原型

void DrvSYS_DisableLowVoltReset (void);

描述

禁止低压复位功能。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

DrvSYS_DisableLowVoltRst ( ); /* Disable low voltage reset function */

DrvSYS_GetBODState

原型

uint32_t DrvSYS_GetBODState (void);

描述

获取 Brown-out 检测器状态。

参数

无

头文件

Driver/DrvSYS.h

返回值

1: 检测到的电压低于 BOD 门限电压.

0: 检测到的电压高于 BOD 门限电压

示例

```
uint32_t u32flag;
/* Get Brown-out state if Brown-out detector function is enabled */
u32flag = DrvSYS_GetBODState ( );
```

DrvSYS_UnlockProtectedReg

原型

```
int32_t   DrvSYS_UnlockProtectedReg (void);
```

描述

解锁被保护的寄存器。为了避免因为一些系统控制寄存器被无意的写入而影响到芯片的正常操作，这些系统控制寄存器需要被保护起来。这些系统控制寄存器在上电复位的时候即被锁住。如果用户想要修改这些寄存器，就必须先为它们解锁。详细的被保护寄存器的相关信息在 TRM 的系统管理章节的“REGWRPROT”寄存器中给出。

参数

无

头文件

Driver/DrvSYS.h

返回值

0　成功

<0　失败

示例

```
int32_t i32ret;
/* Unlock protected registers */
i32ret = DrvSYS_UnlockProtectedReg ( );
```

DrvSYS_LockProtectedReg

原型

```
int32_t   DrvSYS_LockProtectedReg (void);
```

描述

重新锁上被保护的寄存器。推荐用户在修改完被保护的寄存器之后重新锁上它们。

参数

无

头文件

Driver/DrvSYS.h

返回值

0　成功

<0　失败

示例

```
int32_t i32ret;
/* Lock protected registers */
i32ret = DrvSYS_LockProtectedReg ( );
```

DrvSYS_IsProtectedRegLocked

原型

int32_t　DrvSYS_IsProtectedRegLocked (void);

描述

检验被保护的寄存器是否被锁上。

参数

无

头文件

Driver/DrvSYS.h

返回值

1：被保护的寄存器没有被锁上

0：被保护的寄存器已经被锁上

示例

```
int32_t i32flag;
/* Check the protected registers are unlocked or not */
i32flag = DrvSYS_IsProtectedRegLocked ( );
If (i32flag)
   /* do something for unlock */
```

```
else
  /* do something for lock */
```

DrvSYS_EnablePOR

原型

void DrvSYS_EnablePOR (void);

描述

重新使能上电复位控制。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_EnablePOR ( );  /* Enable power-on-reset control */
```

DrvSYS_DisablePOR

原型

void DrvSYS_DisablePOR (void);

描述

禁止上电复位控制。芯片上电时，POR 电路产生一个复位信号复位整个芯片，但是电源上的一些噪声信号也可能导致 POR 电路误产生复位信号，为了防止这种情况发生，用户可以禁止 POR 控制电路。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_DisablePOR ( );   /* Disable power-on-reset control */
```

DrvSYS_SetIPClock

原型

```
void   DrvSYS_SetIPClock (E_SYS_IP_CLK eIpClk, int32_t i32Enable);
```

描述

使能/关闭 IP 时钟，包括看门狗定时器，定时器 0, 定时器 1, 定时器 2, 定时器 3, I2C, SPI0, SPI1, UART0, UART1, PWM01, PWM23, PWM45, PWM67, ADC, EBI, Flash ISP 控制器和频率分频器输出。

参数

eIpClk [in]

IP 时钟枚举值，参考 2.3 节 E_SYS_IP_CLK 的定义

i32Enable [in]

1：使能，0：禁止

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_SetIPClock (E_SYS_I2C0_CLK, 1);    /* Enable I2C0 engine clock */
DrvSYS_SetIPClock (E_SYS_I2C0_CLK, 0);    /* Disable I2C0 engine clock */
DrvSYS_SetIPClock (E_SYS_SPI0_CLK, 1);    /* Enable SPI0 engine clock */
DrvSYS_SetIPClock (E_SYS_SPI0_CLK, 0);    /* Disable SPI0 engine clock */
DrvSYS_SetIPClock (E_SYS_TMR0_CLK, 1);   /* Enable TIMER0 engine clock */
DrvSYS_SetIPClock (E_SYS_TMR0_CLK, 0);   /* Disable TIMER0 engine clock */
```

DrvSYS_SelectHCLKSource

原型

```
int32_t  DrvSYS_SelectHCLKSource (uint8_t u8ClkSrcSel);
```

描述

选择 HCLK 时钟源，时钟源可以是外部 12M crystal 时钟, PLL 时钟, 内部 10K oscillator 时钟, 或者内部 22M oscillator 时钟。HCLK 的详细用法请参考本文档 2.2 节的时钟模块图。

参数

u8ClkSrcSel [in]

0：外部 12M 时钟

2：PLL 时钟

3：内部 10K 时钟

7：内部 22M 时钟

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 参数错误

示例

```
DrvSYS_SelectHCLKSource (0);   /* Change HCLK clock source to be external 12M */
DrvSYS_SelectHCLKSource (2);   /* Change HCLK clock source to be PLL */
```

DrvSYS_SelectSysTickSource

原型

int32_t DrvSYS_SelectSysTickSource (uint8_t u8ClkSrcSel);

描述

选择 Cortex-M0 SysTick 时钟源，可以是外部 12M crystal 时钟, 外部 12M crystal 时钟/2, HCLK/2, 或者内部 22M oscillator 时钟/2。

参数

u8ClkSrcSel [in]

0：外部 12M 时钟

2：PLL 时钟

3：内部 10K 时钟

4~7：内部 22M 时钟

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 参数错误

示例

```
DrvSYS_SelectSysTickSource (0);   /* Change SysTick clock source to be external 12M */
DrvSYS_SelectSysTickSource (3);   /* Change SysTick clock source to be HCLK / 2 */
```

DrvSYS_SelectIPClockSource

原型

int32_t DrvSYS_SelectIPClockSource (E_SYS_IP_CLKSRC eIpClkSrc, uint8_t u8ClkSrcSel);

描述

选择 IP 时钟源，包括看门狗定时器, 模数转换器, 定时器 0~3, UART, PWM01, PWM23, PWM45, PWM67 和频率分频器输出，时钟源的相关信息请参考本文档 2.2 节的时钟模块图。详细的 IP 时钟源的设定在 TRM 的“CLKSEL1”和“CLKSEL2”中给出。

参数

eIpClkSrc [in]

E_SYS_WDT_CLKSRC / E_SYS_ADC_CLKSRC / E_SYS_TMR0_CLKSRC
E_SYS_TMR1_CLKSRC / E_SYS_TMR2_CLKSRC / E_SYS_TMR3_CLKSRC
E_SYS_UART_CLKSRC / E_SYS_PWM01_CLKSRC / E_SYS_PWM23_CLKSRC
E_SYS_PWM45_CLKSRC / E_SYS_PWM67_CLKSRC / E_SYS_FRQDIV_CLKSRC

u8ClkSrcSel [in]

IP 相应的时钟源

u8ClkSrcSel	0	1	2	3	7
看门狗定时器	外部 12M	保留	HCLK/2048	内部 10K	X
ADC	外部 12M	PLL	保留	内部 22M	X
Timer	外部 12M	保留	HCLK	外部触发	内部 22M
UART	外部 12M	PLL	保留	内部 22M	X
PWM	外部 12M	保留	HCLK	内部 22M	X
频率分频器输出	外部 12M	保留	HCLK	内部 22M	X

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 参数错误

示例

```
/* Select ADC clock source from 12M */
DrvSYS_SelectIPClockSource (E_SYS_ADC_CLKSRC, 0x00);
/* Select TIMER0 clock source from external trigger */
DrvSYS_SelectIPClockSource (E_SYS_TMR0_CLKSRC, 0x03);
```

DrvSYS_SetClockDivider

原型

int32_t DrvSYS_SetClockDivider (E_SYS_IP_DIV eIpDiv, int32_t i32value);

描述

设定 IP 时钟源的除频值。

IP 时钟源频率计算公式是：

IP 时钟源频率 / (i32value+1)。

参数

eIpDiv [in]

E_SYS_ADC_DIV / E_SYS_UART_DIV / E_SYS_HCLK_DIV

i32value [in]

除频值。

HCLK, UART：0~15

ADC：0~255

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 参数错误

示例

```
/* Set ADC clock divide number 0x01; ADC clock = ADC source clock / (1+1) */
DrvSYS_SetClockDivider (E_SYS_ADC_DIV, 0x01);
/* Set UART clock divide number 0x02; UART clock = UART source clock / (2+1) */
DrvSYS_SetClockDivider (E_SYS_UART_DIV, 0x02);
/* Set HCLK clock divide number 0x03; HCLK clock = HCLK source clock / (3+1) */
DrvSYS_SetIPClockSource (E_SYS_HCLK_DIV, 0x03);
```

DrvSYS_SetOscCtrl

原型

int32_t DrvSYS_SetOscCtrl (E_SYS_CHIP_CLKSRC eClkSrc, int32_t i32Enable);

描述

使能/禁止内部 oscillator 和外部 crystal ， 包括内部 10K 和 22M oscillator, 外部 12M crystal。

参数

eOscCtrl [in]

E_SYS_XTL12M / E_SYS_OSC22M / E_SYS_OSC10K.

i32Enable [in]

1：使能，0：禁止

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 参数错误

示例

```
DrvSYS_SetOscCtrl (E_SYS_XTL12M, 1);        /* Enable external 12M */
DrvSYS_SetOscCtrl (E_SYS_XTL12M, 0);        /* Disable external 12M */
```

DrvSYS_SetPowerDownWakeUpInt

原型

void DrvSYS_SetPowerDownWakeUpInt (int32_t i32Enable, PWRWU_CALLBACK pdwucallbackFn, int32_t i32enWUDelay);

描述

使能/禁止 power down 唤醒中断功能，如果 power down 唤醒中断使能的话，将安装回调函数，并且可以使能 4096 个时钟延迟来等待 12M crystal 或者 22M oscillator 时钟稳定。当 GPIO, UART, WDT 或者 BOD 唤醒的时候，power down 唤醒中断将发生。

参数

i32Enable [in]

1：使能，0：禁止

pdwucallbackFn [in]

如果唤醒中断使能的话，安装唤醒中断回调函数

i32enWUDelay [in]

1：使能 4096 个时钟延迟，0：关闭 4096 个时钟延迟

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Enable Power down Wake up Interrupt function, install callback function
'PWRWU_CallbackFn', and enable the 4096 clock cycle delay */
DrvSYS_SetPowerDownWakeUpInt (1, PWRWU_CallbackFn, 1);
/* Disable Power down Wake up Interrupt function, and uninstall callback function */
DrvSYS_SetPowerDownWakeUpInt (0, NULL, 0);
```

DrvSYS_EnterPowerDown

原型

void DrvSYS_EnterPowerDown (E_SYS_PD_TYPE ePDType);

描述

立即进入系统 power down 模式或等待 CPU 进入 sleep 模式后进入 power down 模式。当系统进入 power down 模式后，LDO，12M crystal 和 22M oscillator 将被禁止，应用请参考 Application Note，*AN_1007_EN_Power_Management*。

参数

ePDType [in]

E_SYS_IMMEDIATE：芯片立即进入 power down 模式。

E_SYS_WAIT_FOR_CPU：等到 CPU 进入 sleep 模式后，芯片才进入 power down 模式。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Chip enter power mode immediately */
DrvSYS_EnterPowerDown (E_SYS_ IMMEDIATE);
/* Wait for CPU enters sleep mode, then Chip enter power mode */
```

DrvSYS_EnterPowerDown (E_SYS_WAIT_FOR_CPU);

DrvSYS_SelectPLLSource

原型

void DrvSYS_SelectPLLSource (E_SYS_PLL_CLKSRC ePllSrc);

描述

选择 PLL 时钟源，可以是内部 22M oscillator 和外部 12M crystal。

参数

ePllSrc [in]

E_SYS_EXTERNAL_12M / E_SYS_INTERNAL_22M

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Select PLL clock source from 12M */
DrvSYS_ SelectPLLSource (E_SYS_EXTERNAL_12M);
/* Select PLL clock source from 22M */
DrvSYS_ SelectPLLSource (E_SYS_INTERNAL_22M);
```

DrvSYS_SetPLLMode

原型

void DrvSYS_SetPLLMode (int32_t i32Flag);

描述

设置 PLL 模式为 power down 模式或 normal 模式。

参数

i32Flag [in]

1：PLL 处于 power down 模式.

0：PLL 处于 normal 模式.

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Enable PLL power down mode, PLL operates in power down mode */
DrvSYS_SetPLLMode (1);
/* Disable PLL power down mode, PLL operates in normal mode */
DrvSYS_SetPLLMode (0);
```

DrvSYS_GetExtClockFreq

原型

uint32_t DrvSYS_GetExtClockFreq (void);

描述

取得外部 crystal 时钟频率。单位是 Hz。

参数

无

头文件

Driver/DrvSYS.h

返回值

外部 crystal 时钟频率

示例

```
uint32_t u32clock;
u32clock = DrvSYS_GetExtClockFreq ( ); /* Get external crystal clock frequency */
```

DrvSYS_GetPLLContent

原型

uint32_t DrvSYS_GetPLLContent(E_SYS_PLL_CLKSRC ePllSrc, uint32_t u32PllClk);

描述

根据 u32PllClk 定义的目标 PLL 频率，计算出最接近于该频率的 PLL 频率值。

参数

ePllSrc [in]

E_SYS_EXTERNAL_12M / E_SYS_INTERNAL_22M

u32PllClk [in]

目标 PLL 时钟频率，单位是 Hz。u32PllClk 的取值范围是 25MHz~200MHz。

头文件

Driver/DrvSYS.h

返回值

PLL 控制寄存器的设定值。

示例

```
uint32_t u32PllCr;
/* Get PLL control register setting for target PLL clock 50MHz */
u32PllCr = DrvSYS_GetPLLContent (E_SYS_EXTERNAL_12M, 50000000);
```

DrvSYS_SetPLLContent

原型

```
void  DrvSYS_SetPLLContent (uint32_t u32PllContent);
```

描述

设定 PLL 的设定值。用户可以使用 DrvSYS_GetPLLContent ( )来获取合适的 PLL 设定值和使用 DrvSYS_GetPLLClockFreq ( ) 来获取实际的 PLL 时钟频率。

参数

u32PllContent [in]

PLL 寄存器需要的设定值，为了获得目标 PLL 时钟频率。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
uint32_t u32PllCr;
/* Get PLL control register setting for target PLL clock 50MHz */
u32PllCr = DrvSYS_GetPLLContent (E_DRVSYS_EXTERNAL_12M, 50000000);
/* Set PLL control register setting to get nearest PLL clock */
DrvSYS_SetPLLContent (u32PllCr);
```

DrvSYS_GetPLLClockFreq

原型

uint32_t DrvSYS_GetPLLClockFreq (void);

描述

获取 PLL 时钟输出频率。

参数

无

头文件

Driver/DrvSYS.h

返回值

PLL 时钟输出频率，单位 Hz

示例

```
uint32_t u32clock;
u32clock = DrvSYS_GetPLLClockFreq ( );   /* Get actual PLL clock */
```

DrvSYS_GetHCLKFreq

原型

uint32_t DrvSYS_GetHCLKFreq (void);

描述

获取 HCLK 时钟频率。

参数

无

头文件

Driver/DrvSYS.h

返回值

HCLK 时钟频率，单位 Hz

示例

```
uint32_t u32clock;
u32clock = DrvSYS_GetHCLKFreq ( );   /* Get current HCLK clock */
```

DrvSYS_Open

原型

int32_t DrvSYS_Open (uint32_t u32Hclk);

描述

根据 PLL 源时钟和目标 HCLK 时钟频率，配置 PLL 设定值。由于硬件的限制，实际的 HCLK 时钟可能跟目标 HCLK 时钟略有不同。

DrvSYS_GetPLLClockFreq ( ) 可以用于获取实际的 PLL 时钟

DrvSYS_GetHCLKFreq ( ) 可以用于获取实际的 HCLK 时钟

参数

u32Hclk [in]

目标 HCLK 时钟频率，单位是 Hz。u32Hclk 的取值范围是 25MHz~200MHz。

头文件

Driver/DrvSYS.h

返回值

0 成功

< 0 时钟设定值超出了其取值范围

示例

```
/* Set PLL clock 50MHz, and switch HCLK source clock to PLL */
DrvSYS_Open (50000000);
```

DrvSYS_SetFreqDividerOutput

原型

int32_t DrvSYS_SetFreqDividerOutput (int32_t i32Flag, uint8_t u8Divider);

描述

M051 系列支持通过 CLKO 输出引脚来监视时钟源频率。该函数用来使能或禁止频率时钟输出，并设定其除频值。输出频率大小计算公式为 $F_{out} = F_{in} / 2^{N+1}$，$F_{in}$ 是输出时钟频率，F_{out} 是除频器输出时钟频率，N 是一个 4-bit 的数值。

为了监视时钟源频率，我们可以使用该函数使能时钟输出功能。然而，我们还需要把 CLKO 设定为输出引脚，这可通过把 M051 系列 GPIO 多功能选择选择为输出时钟到时钟输出引脚来实现。

参数

i32Flag [in]

1：使能，0：禁止

U8Divider [in]

输出频率的除频值，范围是 0~15。

头文件

Driver/DrvSYS.h

返回值

0 成功

<0 参数错误

示例

```
/* Enable frequency clock output and set its divide number 2,
The output frequency = input clock / 2^(2+1) */
DrvSYS_SetFreqDividerOutput (1, 2);
/* Disable frequency clock output */
DrvSYS_SetFreqDividerOutput (0, 0);
```

DrvSYS_Delay

原型

```
void   DrvSYS_Delay (uint32_t us);
```

描述

使用 Cortex-M0 的 SysTick 定时器产生延时时间，单位是 us。SysTick 的时钟源默认是来自 HCLK 时钟。如果 SysTick 的时钟源被用户修改了，延时时间可能不正确。

参数

us [in]

延时时间，最大延时时间是 335000us。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_Delay (5000);    /* Delay 5000us */
```

DrvSYS_GetChipClockSourceStatus

原型

int32_t DrvSYS_GetChipClockSourceStatus (E_SYS_CHIP_CLKSRC eClkSrc);

描述

监视芯片时钟源是否稳定，包括内部 10K, 22M Oscillator, 外部 12M crystal, 或 PLL 时钟。

参数

eClkSrc [in]

E_SYS_XTL12M / E_SYS_OSC22M / E_SYS_OSC10K / E_SYS_PLL

头文件

Driver/DrvSYS.h

返回值

0 时钟源不稳定或者时钟未被使能

1 时钟源稳定

<0 参数错误

示例

```
/* Enable external 12M */
DrvSYS_SetOscCtrl (E_SYS_XTL12M, 1);
/* Waiting for 12M Crystal stable */
while (DrvSYS_GetChipClockSourceStatus (E_SYS_XTL12M) != 1);
/* Disable PLL power down mode */
DrvSYS_SetPLLMode (0);
/* Waiting for PLL clock stable */
while (DrvSYS_GetChipClockSourceStatus (E_SYS_PLL) != 1);
```

DrvSYS_GetClockSwitchStatus

原型

uint32_t DrvSYS_GetClockSwitchStatus (void);

描述

当软件切换系统时钟源时，检验切换目标时钟成功与否。

参数

无

头文件

Driver/DrvSYS.h

返回值

0：时钟切换成功

1：时钟切换失败

示例

```
uint32_t u32flag;
DrvSYS_SelectHCLKSource (2);   /* Change HCLK clock source to be PLL */
u32flag = DrvSYS_GetClockSwitchStatus ( );   /* Get clock switch flag */
If (u32flag)
  /* do something for clock switch fail */
```

DrvSYS_ClearClockSwitchStatus

原型

void DrvSYS_ClearClockSwitchStatus (void);

描述

清除时钟切换失败标志。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
uint32_t u32flag;
DrvSYS_SelectHCLKSource (0);   /* Change HCLK clock source to be external 12M */
u32flag = DrvSYS_GetClockSwitchStatus ( );   /* Get clock switch fail flag */
if (u32flag)
  DrvSYS_ClearClockSwitchStatus ( );       /* Clear clock switch fail flag */
```

DrvSYS_GetVersion

原型

uint32_t DrvSYS_GetVersion (void);

描述

获取该 DrvSYS 驱动版本。

参数

无

头文件

Driver/DrvSYS.h

返回值

版本号：

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

3. UART 驱动

3.1. UART 介绍

通用异步收发器(UART)从外设，比如调制解调器接收到数据的时候实现串到并转换，从 CPU 收到数据的时候实现并到串的转换。

细节请参考芯片规格书 UART 章节。

3.2. UART Feature

UART 包含如下特性：

- 用作收/发数据缓冲的 64 字节(UART0)/16 字节(UART1) 缓冲区
- 支持自动流控/流控功能(CTS, RTS).
- 完全可编程的串口特性:
  -- 5-, 6-, 7-, 或者 8 比特字符
  -- 奇、偶或者无校验比特产生和探测
  -- 1-, 1&1/2, 或者 2 比特停止位
  -- 波特率发生器
  -- 错误的起始位探测.
- 全优先中断系统控制
- 用于内部测试的回送模式
- 支持 IrDA SIR 功能
- 支持 LIN(本地互联网络)主模式
- 可编程波特率发生器，允许时钟除以可编程的分频

3.3. 常量定义

常量名	值	描述
MODE_TX	1	IRDA 或 LIN 发送模式
MODE_RX	2	IRDA 或 LIN 接收模式

3.4. 类型定义

E_UART_PORT

枚举标识符	值	描述
UART_PORT0	0x000	UART 端口 0
UART_PORT1	0x100000	UART 端口 1
UART_PORT2	0x104000	UART 端口 2

E_INT_SOURCE

枚举标识符	值	描述
DRVUART_RDAINT	0x1	接收数据有效中断和超时中断
DRVUART_THREINT	0x2	发送保存寄存器空中断
DRVUART_WAKEUPINT	0x40	唤醒中断使能
DRVUART_RLSINT	0x4	接收线上中断
DRVUART_MOSINT	0x8	调制解调器中断
DRVUART_TOUTINT	0x10	超时中断
DRVUART_BUFERRINT	0x20	缓冲区错误中断使能
DRVUART_LININT	0x100	LIN RX Break Field Detected 中断使能

E_DATABITS_SETTINGS

枚举标识符	值	描述
DRVUART_DATABITS_5	0x0	字长选择：字符长度是 5 比特
DRVUART_DATABITS_6	0x1	字长选择：字符长度是 6 比特
DRVUART_DATABITS_7	0x2	字长选择：字符长度是 7 比特
DRVUART_DATABITS_8	0x3	字长选择：字符长度是 8 比特

E_PARITY_SETTINGS

枚举标识符	值	描述
DRVUART_PARITY_NONE	0x0	无校验
DRVUART_PARITY_ODD	0x1	使能奇校验

DRVUART_PARITY_EVEN	0x3	使能偶校验
DRVUART_PARITY_MARK	0x5	Parity mask
DRVUART_PARITY_SPACE	0x7	Parity space

E_STOPBITS_SETTINGS

枚举标识符	值	描述
DRVUART_STOPBITS_1	0x0	停止位长度：1 比特
DRVUART_STOPBITS_1_5	0x1	停止位长度：当字长为 5 比特时 1.5 比特
DRVUART_STOPBITS_2	0x1	停止位长度：当字长为 6，7 或 8 比特时 2 比特

E_FIFO_SETTINGS

枚举标识符	值	描述
DRVUART_FIFO_1BYTES	0x0	接收缓冲区中断触发级别是 1 个字节
DRVUART_FIFO_4BYTES	0x1	接收缓冲区中断触发级别是 4 个字节
DRVUART_FIFO_8BYTES	0x2	接收缓冲区中断触发级别是 8 个字节
DRVUART_FIFO_14BYTES	0x3	接收缓冲区中断触发级别是 14 个字节
DRVUART_FIFO_30BYTES	0x4	接收缓冲区中断触发级别是 30 个字节
DRVUART_FIFO_46BYTES	0x5	接收缓冲区中断触发级别是 46 个字节
DRVUART_FIFO_62BYTES	0x6	接收缓冲区中断触发级别是 62 个字节

E_UART_FUNC

枚举标识符	值	描述
FUN_UART	0	选择 UART 功能
FUN_LIN	1	选择 LIN 功能
FUN_IRDA	2	选择 IrDA 功能
FUN_RS485	3	选择 RS485 功能

E_MODE_RS485

枚举标识符	值	描述
MODE_RS485_NMM	1	RS-485 正常多点操作模式
MODE_RS485_AAD	2	RS-485 自动地址检测操作模式
MODE_RS485_AUD	4	RS-485 自动方向模式

3.5. 宏

_DRVUART_SENDBYTE

原型

void _DRVUART_SENDBYTE (u32Port, byData);

描述

从 UART 发送 1 字节数据。

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Using UART port0 to send one byte 0x55 */
_DRVUART_SENDBYTE (UART_PORT0, 0x55);
```

_DRVUART_RECEIVEBYTE

原型

uint8_t _DRVUART_RECEIVEBYTE (u32Port);

描述

从指定串口的 FIFO 接收 1 字节数据。

头文件

Driver/DrvUART.h

返回值

1 字节数据。

示例

```
/* Using UART port0 to receive one byte */
uint8_t u8data;
u8data = _DRVUART_RECEIVEBYTE (UART_PORT0);
```

_DRVUART_SET_DIVIDER

原型

void _DRVUART_SET_DIVIDER (u32Port, u16Divider);

描述

设定 UART 除频值来控制 UART 波特率。

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Set the divider of UART is 6 */
_DRVUART_SET_DIVIDER (UART_PORT0, 6);
```

_DRVUART_RECEIVEAVAILABLE

原型

```
int8_t  _DRVUART_RECEIVEAVAILABLE (u32Port);
```

描述

获取当前接收 FIFO 指针。

头文件

Driver/DrvUART.h

返回值

接收 FIFO 指针值。

示例

```
/* To get UART channel 0 current Rx FIFO pointer */
_DRVUART_RECEIVEAVAILABLE (UART_PORT0);
```

_DRVUART_WAIT_TX_EMPTY

原型

```
void  _DRVUART_WAIT_TX_EMPTY (u32Port);
```

描述

查询 Tx FIFO 空标志来检验 Tx FIFO 为空。

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Send 0x55 from UART0 and check TX FIFO is empty */
_DRVUART_SENDBYTE (UART_PORT0, 0x55);
```

```
_DRVUART_WAIT_TX_EMPTY (UART_PORT0);
```

3.6. 函数

DrvUART_Open

原型

```
int32_t
DrvUART_Open (
    E_UART_PORT u32Port,
    UART_T *sParam
);
```

描述

该函数用于初始化 UART。包括波特率、奇偶校验、数据位、停止位、接收触发级别和超时时间等的设定。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

sParam [in]

指定 UART 的特性。包括

u32BaudRate: 波特率(Hz)

u8cParity: 无/奇/偶校验

可以是

DRVUART_PARITY_NONE 无校验).

DRVUART_PARITY_EVEN (偶校验)

DRVUART_PARITY_ODD(奇校验).

u8cDataBits: 数据位设定

可以是

DRVUART_DATA_BITS_5 (5 比特).

DRVUART_DATA_BITS_6 (6 比特)

DRVUART_DATA_BITS_7 (7 比特).

DRVUART_DATA_BITS_8 (8 比特).

u8cStopBits: 停止位设定

可以是

DRVUART_STOPBITS_1 (1 停止位).

DRVUART_STOPBITS_1_5 (1.5 停止位)

DRVUART_STOPBITS_2 (2 停止位).

u8cRxTriggerLevel: 接收 FIFO 中断触发级别

LEVEL_X_BYTE 说明串口中断触发级别是 X 字节。

可以是

DRVUART_FIFO_1BYTE, DRVUART_FIFO_4BYTES

DRVUART_FIFO_8BYTES, DRVUART_FIFO_14BYTES

DRVUART_FIFO_30BYTES, DRVUART_FIFO_46BYTES

DRVUART_FIFO_62BYTES

对于 UART0，可取值为从 LEVEL_1_BYTE 到 LEVEL_62_BYTES.

其他的，取值范围为从 LEVEL_1_BYTE 到 LEVEL_14_BYTES.

u8TimeOut: 超时时间“N”，表示 N 个时钟周期，计数时钟是波特率。

头文件

Driver/DrvUART.h

返回值

E_SUCCESS: 成功.

E_DRVUART_ERR_PORT_INVALID: 端口错误

E_DRVUART_ERR_PARITY_INVALID: 校验设定错误

E_DRVUART_ERR_DATA_BITS_INVALID: 数据比特错误

E_DRVUART_ERR_STOP_BITS_INVALID: 停止位设定错误

E_DRVUART_ERR_TRIGGERLEVEL_INVALID: 缓冲区触发级别错误

示例

```
/* Set UART0 under 115200bps, 8 data bits ,1 stop bit and none parity and 1 byte Rx trigger level settings. */
STR_UART_T sParam;
sParam.u32BaudRate      = 115200;
sParam.u8cDataBits      = DRVUART_DATABITS_8;
sParam.u8cStopBits      = DRVUART_STOPBITS_1;
sParam.u8cParity        = DRVUART_PARITY_NONE;
sParam.u8cRxTriggerLevel    = DRVUART_FIFO_1BYTES;
DrvUART_Open (UART_PORT0, &sParam);
```

DrvUART_Close

原型

```
void DrvUART_Close (
    E_UART_PORT u32Port
);
```

描述

该函数用于关闭 UART 时钟和中断，在检验 Tx 为空时清除回调函数指针。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Close UART channel 0 */
DrvUART_Close (UART_PORT0);
```

DrvUART_EnableInt

原型

```
void      DrvUART_EnableInt (
    E_UART_PORT u32Port,
    uint32_t            u32InterruptFlag,
    PFN_DRVUART_CALLBACK pfncallback
);
```

描述

该函数用于使能指定 UART 中断，安装中断回调函数，并使能 NVIC 串口中断。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断使能

DRVUART_BUFERRINT : Buffer Error 中断使能

DRVUART_WAKEINT : Wakeup 中断.

DRVUART_MOSINT : MODEM Status 中断

DRVUART_RLSNT : Receive Line Status 中断

DRVUART_THREINT : Transmit Holding Register Empty 中断

DRVUART_RDAINT : Receive Data Available 中断

DRVUART_TOUTINT : Time-out 中断

pfncallback [in]

回调函数指针

头文件

Driver/DrvUART.h

返回值

无

Note

使用 "|" 运算符来连接中断标志可以同时使能多个中断。

如果你在同一个工程中调用该函数 2 次，设定结果取决于第二次的设定值。

示例

```
/* Enable UART channel 0 RDA and THRE interrupt. Finally, install UART_INT_HANDLE function to be callback function. */

DrvUART_EnableInt(UART_PORT0, (DRVUART_RDAINT | DRVUART_THREINT ),UART_INT_HANDLE);
```

DrvUART_DisableInt

原型

```
void      DrvUART_DisableInt (
    E_UART_PORT u32Port
    uint32_t            u32InterruptFlag
);
```

描述

该函数用于禁止 UART 指定的中断，卸载相应的中断回调函数，并禁止 NVIC 串口中断。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断使能

DRVUART_BUFERRINT : Buffer Error 中断使能

DRVUART_WAKEINT : Wakeup 中断

DRVUART_MOSINT : MODEM Status 中断

DRVUART_RLSNT : Receive Line Status 中断

DRVUART_THREINT : Transmit Holding Register Empty 中断

DRVUART_RDAINT : Receive Data Available 中断

DRVUART_TOUTINT : Time-out 中断

头文件

Driver/DrvUART.h

返回值

无

Note

使用 "|" 运算符可以同时关闭多个中断

示例

```
/* To disable the THRE interrupt enable flag. */
DrvUART_DisableInt (UART_PORT0, DRVUART_THREINT);
```

DrvUART_ClearIntFlag

原型

```
uint32_t
DrvUART_ClearIntFlag (
    E_UART_PORT u32Port
    uint32_t        u32InterruptFlag
);
```

描述

该函数用于清除 UART 指定中断标志。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断使能

DRVUART_BUFERRINT : Buffer Error 中断使能

DRVUART_WAKEINT : Wakeup 中断

DRVUART_MOSINT : MODEM Status 中断

DRVUART_RLSNT : Receive Line Status 中断

DRVUART_THREINT : Transmit Holding Register Empty 中断

DRVUART_RDAINT : Receive Data Available 中断

DRVUART_TOUTINT : Time-out 中断

头文件

Driver/DrvUART.h

返回值

E_SUCESS 成功

示例

```
/* To disable the THRE interrupt enable flag. */
DrvUART_DisableInt (UART_PORT0, DRVUART_THREINT);
```

DrvUART_GetIntStatus

原型

```
int32_t
DrvUART_GetIntStatus (
    E_UART_PORT u32Port
    uint32_t    u32InterruptFlag
);
```

描述

这个函数用来取得指定 UART 中断状态

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断使能

DRVUART_BUFERRINT : Buffer Error 中断使能

DRVUART_WAKEINT : Wakeup 中断

DRVUART_MOSINT : MODEM Status 中断

DRVUART_RLSNT : Receive Line Status 中断

DRVUART_THREINT : Transmit Holding Register Empty 中断

DRVUART_RDAINT : Receive Data Available 中断

DRVUART_TOUTINT : Time-out 中断

头文件

Driver/DrvUART.h

返回值

0：指定中断没有发生。

1：指定中断发生。

E_DRVUART_ARGUMENT：参数错误

Note

一次只能查询一个中断。

示例

```
/* To get the THRE interrupt enable flag. */
If(DrvUART_GetIntStatus (UART_PORT0, DRVUART_THREINT))
   printf("THRE INT is happened!\n");
else
   printf("THRE INT is not happened or error parameter\n");
```

DrvUART_GetCTSInfo

原型

```
void
DrvUART_GetCTSInfo(
   E_UART_PORT  u32Port,
   uint8_t              *pu8CTSValue,
   uint8_t          *pu8CTSChangeState
}
```

描述

这个函数可以用来取得 CTS 引脚值并且检测 CTS 引脚状态是否改变。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

pu8CTSValue [out]

指定存放 CTS 值的缓存地址。返回当前 CTS 引脚状态。

pu8CTSChangeState [out]

指定存放 CTS 改变状态的缓存地址，返回 CTS 引脚状态是否改变。1 表示 CTS 引脚状态发生了改变，0 表示状态没有改变。

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* To get CTS pin status and save to u8CTS_value. To get detect CTS change flag and save to u8CTS_state. */
uint8_t u8CTS_value, u8CTS_state;
DrvUART_GetCTSInfo(UART_PORT1,& u8CTS_value,& u8CTS_state);
```

DrvUART_SetRTS

原型

```
void
DrvUART_SetRTS (
    E_UART_PORT u32Port,
    uint8_t             u8Value,
    uint16_t      u16TriggerLevel
}
```

描述

这个函数可以用来设定 RTS 设定值

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u8Value [in]

设置为 0：把 RTS 引脚拉为逻辑“1”(如果 LEV_RTS 设置为低电平触发)。

把 RTS 引脚拉为逻辑“0”(如果 LEV_RTS 设置为高电平触发)。

设置为 1：把 RTS 引脚拉为逻辑“0”(如果 LEV_RTS 设置为低电平触发)。

把 RTS 引脚拉为逻辑“1”(如果 LEV_RTS 设置为高电平触发)。

NOTE：LEV_RTS 是 RTS 触发电平。“0”是低电平，“1”是高电平。

u16TriggerLevel [in]

RTS 触发级别：DRVUART_FIFO_1BYTES 到 DRVUART_FIFO_62BYTES.

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Condition: Drive RTS to logic 1 in UART channel 1 and Set RTS trigger level is 1 bytes*/
DrvUART_SetRTS (UART_PORT1,1, DRVUART_FIFO_1BYTES);
```

DrvUART_Read

原型

```
int32_t
DrvUART_Read (
    E_UART_PORT         u32Port
    uint8_t             *pu8RxBuf,
    uint32_t            u32ReadBytes
);
```

描述

这个函数用来从接收缓存中读取数据，并把这些数据存储在 pu8RxBuf 指定的地址中。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

pu8RxBuf [out]

指定存放接收到的数据的缓存地址。

u32ReadBytes [in]

设定要读取的字节数。

头文件

Driver/DrvUART.h

返回值

E_SUCCESS：成功

E_DRVUART_TIMEOUT：FIFO 查询超时

示例

```
/* Condition: Read RX FIFO 1 byte and store in bInChar buffer. */
uint8_t bInChar[1];
DrvUART_Read(UART_PORT0,bInChar,1);
```

DrvUART_Write

原型

```
int32_t
DrvUART_Write(
    E_UART_PORT u32Port
    uint8_t             *pu8TxBuf,
    uint32_t            u32WriteBytes
);
```

描述

这个函数可用来写数据到发送缓存，然后通过 UART 发送出去

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

pu8TxBuf [in]

指定用于发送数据到 UART FIFO 的缓存地址。

u32WriteBytes [in]

指定要发送的字节数。

头文件

Driver/DrvUART.h

返回值

E_SUCCESS：成功

E_DRVUART_TIMEOUT：FIFO 查询超时

示例

```
/* Condition: Send 1 byte from bInChar buffer to TX FIFO. */
```

```
uint8_t bInChar[1] = 0x55;
DrvUART_Write(UART_PORT0,bInChar,1);
```

DrvUART_SetFnIRDA

原型

```
void
DrvUART_SetFnIRDA (
   E_UART_PORT u32Port
   STR_IRCR_T   str_IRCR
);
```

描述

这个函数用来配置 IRDA 相关的设定值。包括 TX 或 RX 模式和反转 TX 或 RX 信号。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

str_IRCR [in]

IrDA 结构体

包括

u8cTXSelect ： 1：使能 IrDA 发送功能。TX 模式。

0：禁止 IrDA 发送功能。RX 模式。

u8cInvTX ： 反转 Tx 信号功能 TRUE 或 FALSE。

u8cInvRX ： 反转 Rx 信号功能(默认值是 TRUE) TRUE 或 FALSE。

头文件

Driver/DrvUART.h

返回值

无

Note

在使用该 API 之前，需要首先配置 UART 的设定值，并确认在配置波特率时使用 Mode 0(UART divider is 16)设定。

示例

```
/* Change UART1 to IRDA function and Inverse the RX signals. */
STR_IRCR_T sIrda;
```

```
sIrda.u8cTXSelect  = ENABLE;
sIrda.u8cInvTX    = FALSE;
sIrda.u8cInvRX    = TRUE;
DrvUART_SetFnIRDA(UART_PORT1,&sIrda);
```

DrvUART_SetFnRS485

原型

```
void
DrvUART_OpenRS485 (
   E_UART_PORT u32Port,
   STR_RS485_T *str_RS485
);
```

描述

该函数用于设定与 RS485 相关的设置。

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

str_RS485 [in]

RS485 结构体

包括

u8cModeSelect：选择操作模式

MODE_RS485_NMM：RS-485 Normal Multi-drop 模式

MODE_RS485_AAD：RS-485 Auto Address Detection 模式

MODE_RS485_AUD：RS-485 Auto Direction 模式

u8cAddrEnable：使能或禁止 RS-485 地址检测

u8cAddrValue：设定地址匹配值

u8cDelayTime：设定发送延时时间

u8cRxDisable：使能或禁止接收器功能。

头文件

Driver/DrvUART.h

返回值

无

Note

无

示例

```
/* Condition: Change UART1 to RS485 function. Set relative setting as below.*/
STR_RS485_T sParam_RS485;
sParam_RS485.u8cAddrEnable  = ENABLE;
sParam_RS485.u8cAddrValue   = 0xC0;             /* Address */
sParam_RS485.u8cModeSelect  = MODE_RS485_AAD|MODE_RS485_AUD;
sParam_RS485.u8cDelayTime   = 0;
sParam_RS485.u8cRxDisable           = TRUE;
DrvUART_SetFnRS485(UART_PORT1,&sParam_RS485);
```

DrvUART_SetFnLIN

原型

```
void
DrvUART_SetFnLIN (
    E_UART_PORT u32Port
    uint16_t u16Mode,
    uint16_t u16BreakLength
);
```

描述

该函数用来设定与 LIN 相关的设定

参数

u32Port [in]

指定 UART_PORT0/UART_PORT1

u16Mode [in]

Specify LIN direction：指定 LIN 方向：MODE_TX 和/或 MODE_RX

u16BreakLength [in]

指定 Break Field 的长度。根据 LIN 的协议，该值应当大于 13 比特。

头文件

Driver/DrvUART.h

返回值

无

示例

/* Change UART1 to LIN function and set to transmit the header information. */

DrvUART_SetFnLIN(uart_ch,MODE_TX | MODE_RX,13);

DrvUART_GetVersion

原型

int32_t

DrvUART_GetVersion (void);

描述

返回当前驱动版本号。

头文件

Driver/DrvUART.h

返回值

版本号

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

4. TIMER/WDT 驱动

4.1. TIMER/WDT 介绍

定时器模块包含 4 个通道：TIMER0~TIMER3，用户利用它们可以便捷的实现计数方案。定时器模块可用于实现频率测量、事件计数、间隔时间测量、时钟发生器、延时等功能。定时器可产生超时中断信号，或者在操作期间提供当前计数值。

看门狗定时器(WDT)用在当软件运行出现问题的时候执行系统复位，可以防止系统无限期挂起。

4.2. TIMER/WDT 特性

- 每个通道都有独立的时钟源

  TMR0_CLK, TMR1_CLK, TMR2_CLK 和 TMR3_CLK
- 内部 8 位预分频计数器
- 内部 24 位向上计数器，通过 TDR(定时器数据寄存器)可读。
- 时间溢出周期 = (Period of timer clock input) * (8-bit pre-scale counter + 1) * (24-bit TCMP)
- 最大计数周期 = (1 / 25 MHz) * (2^8) * (2^24 - 1) 当 TCLK = 25 MHz 时
- 18 位自动运行计数器，以避免 CPU 在延时时间到期之前从看门狗复位。
- 可选的超时间隔(2^4～2^18)，当 WDT_CLK = 12MHz 时，超时间隔是 86.67us～21.93ms
- 当 WDT_CLK = 12MHz 时, 复位时间 = (1/12M)*63
- Reset period = (1/12MHz) * 63, if WDT_CLK = 12MHz.

4.3. 类型定义

E_TIMER_CHANNEL

枚举标识符	值	描述
E_TMR0	0x0	指定定时器通道 0
E_TMR1	0x1	指定定时器通道 1

E_TMR2	0x2	指定定时器通道 2
E_TMR3	0x3	指定定时器通道 3

E_TIMER_OPMODE

枚举标识符	值	描述
E_ONESHOT_MODE	0x0	设定定时器为 One-Shot 模式
E_PERIODIC_MODE	0x1	设定定时器为 Periodic 模式
E_TOGGLE_MODE	0x2	设定定时器为 Toggle 模式

E_WDT_CMD

枚举标识符	值	描述
E_WDT_IOC_START_TIMER	0x0	开始 WDT 计数
E_WDT_IOC_STOP_TIMER	0x1	停止 WDT 计数
E_WDT_IOC_ENABLE_INT	0x2	使能 WDT 中断
E_WDT_IOC_DISABLE_INT	0x3	禁止 WDT 中断
E_WDT_IOC_ENABLE_WAKEUP	0x4	使能 WDT 超时唤醒功能
E_WDT_IOC_DISABLE_WAKEUP	0x5	禁止 WDT 超时唤醒功能
E_WDT_IOC_RESET_TIMER	0x6	复位 WDT 计数器
E_WDT_IOC_ENABLE_RESET_FUNC	0x7	使能 WDT 复位功能
E_WDT_IOC_DISABLE_RESET_FUNC	0x8	禁止 WDT 复位功能
E_WDT_IOC_SET_INTERVAL	0x9	设定 WDT 超时间隔

E_WDT_INTERVAL

枚举标识符	值	描述
E_LEVEL0	0x0	设定 WDT 超时间隔为 2^4WDT_CLK
E_LEVEL1	0x1	设定 WDT 超时间隔为 2^6WDT_CLK
E_LEVEL2	0x2	设定 WDT 超时间隔为 2^8WDT_CLK
E_LEVEL3	0x3	设定 WDT 超时间隔为 2^10WDT_CLK
E_LEVEL4	0x4	设定 WDT 超时间隔为 2^12WDT_CLK
E_LEVEL5	0x5	设定 WDT 超时间隔为 2^14WDT_CLK
E_LEVEL6	0x6	设定 WDT 超时间隔为 2^16WDT_CLK
E_LEVEL7	0x7	设定 WDT 超时间隔为 2^18WDT_CLK

4.4. 函数

DrvTIMER_Init

原型

void DrvTIMER_Init (void)

描述

系统启动后，用户在对定时器做任何操作之前，一定要先调用该函数。

参数

无

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Info the system can accept Timer APIs after calling DrvTIMER_Init() */
DrvTIMER_Init ();
```

DrvTIMER_Open

原型

```
int32_t DrvTIMER_Open (
    E_TIMER_CHANNEL ch,
    uint32_t                uTicksPerSecond,
    E_TIMER_OPMODE   op_mode
)
```

描述

打开指定的定时器，并指定该定时器操作模式。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

uTicksPerSecond [in]

定时器每秒中断 tick 数。

op_moode [in]

E_TIMER_OPMODE, E_ONESHOT_MODE / E_PERIODIC_MODE / E_TOGGLE_MODE

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道
E_DRVTIMER_CLOCK_RATE：	计算初始值失败

示例

```
/* Using TIMER0 at PERIODIC_MODE, 2 ticks / sec */
DrvTIMER_Open (E_TMR0, 2, E_PERIODIC_MODE);
```

DrvTIMER_Close

原型

int32_t DrvTIMER_Close (E_TIMER_CHANNEL ch)

描述

这个函数用来关闭定时器通道。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Close the specified timer channel */
DrvTIMER_Close (E_TMR0);
```

DrvTIMER_SetTimerEvent

原型

```
int32_t DrvTIMER_SetTimerEvent (
    E_TIMER_CHANNEL ch,
    uint32_t            uInterruptTicks,
    TIMER_CALLBACK  pTimerCallback ,
    uint32_t            parameter
)
```

描述

安装指定定时器通道的中断回调函数，当中断次数达到 *uInterruptTicks* 次时触发定时器回调函数。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

uInterruptTicks [in]

定时器中断发生次数。

pTimerCallback [in]

中断回调函数函数指针。

parameter [in]

传给回调函数的参数。

头文件

Driver/DrvTIMER.h

返回值

uTimerEventNo：　　定时器事件数

E_DRVTIMER_EVENT_FULL：　　定时器事件满

示例

```
/* Install callback "TMR_Callback" and trigger callback
when timer interrupt happen twice */
uTimerEventNo = DrvTIMER_SetTimerEvent (E_TMR0, 2,
    (TIMER_CALLBACK)TMR_Callback, 0);
```

DrvTIMER_ClearTimerEvent

原型

```
void DrvTIMER_ClearTimerEvent (
    E_TIMER_CHANNEL ch,
    uint32_t            uTimerEventNo
```

)

描述

清除指定定时器通道的定时器事件。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

uTimerEventNo [in]

定时器事件数

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Close the specified timer event */
DrvTIMER_ClearTimerEvent (E_TMR0, uTimerEventNo);
```

DrvTIMER_EnableInt

原型

int32_t DrvTIMER_EnableInt (E_TIMER_CHANNEL ch)

描述

该函数用于使能指定定时器中断。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Enable Timer-0 interrupt function */
```

DrvTIMER_EnableInt (E_TMR0);

DrvTIMER_DisableInt

原型

int32_t DrvTIMER_DisableInt (E_TIMER_CHANNEL ch)

描述

该函数用于禁止指定定时器中断。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Disable Timer-0 interrupt function */
DrvTIMER_DisaleInt (E_TMR0);
```

DrvTIMER_GetIntFlag

原型

int32_t DrvTIMER_GetIntFlag (E_TIMER_CHANNEL ch)

描述

获取指定定时器通道的中断标志状态。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

iIntStatus：	0：没有中断

1：发生了中断

E_DRVTIMER_CHANNEL： 无效定时器通道

示例

```
/* Get the interrupt flag status from Timer-0 */
u32TMR0IntFlag = DrvTIMER_GetIntFlag (E_TMR0);
```

DrvTIMER_ClearIntFlag

原型

int32_t DrvTIMER_ClearIntFlag (E_TIMER_CHANNEL ch)

描述

清除指定定时器通道的中断标志。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS： 操作成功

E_DRVTIMER_CHANNEL： 无效定时器通道

示例

```
/* Clear Timer-0 interrupt flag */
DrvTIMER_ClearIntFlag (E_TMR0);
```

DrvTIMER_Start

原型

int32_t DrvTIMER_Start (E_TIMER_CHANNEL ch)

描述

指定定时器通道开始计数。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Start to count the Timer-0 */
DrvTIMER_Start (E_TMR0);
```

DrvTIMER_GetIntTicks

原型

uin32_t DrvTIMER_GetIntTicks (E_TIMER_CHANNEL ch)

描述

该函数用于获取定时器中断功能使能后发生的定时器中断次数。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

uTimerTick：	返回中断 tick 数
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Get the current interrupt ticks from Timer-1 */
u32TMR1Ticks = DrvTIMER_GetIntTicks (E_TMR1);
```

DrvTIMER_ResetIntTicks

原型

int32_t DrvTIMER_ResetIntTicks (E_TIMER_CHANNEL ch)

描述

该函数用于清零中断 tick 数。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_DRVTIMER_CHANNEL：	无效定时器通道

示例

```
/* Reset the interrupt ticks of Timer-1 to 0 */
DrvTIMER_ResetIntTicks (E_TMR1);
```

DrvTIMER_Delay

原型

void DrvTIMER_Delay (E_TIMER_CHANNEL ch, uint32_t uIntTicks)

描述

该函数通过指定定时器通道中断 tick 数来增加一个延时循环。

参数

ch [in]

E_TIMER_CHANNEL，可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

uIntTicks [in]

延迟 tick 数

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Delay Timer-0 3000 ticks */
DrvTIMER_Delay (E_TMR0, 3000);
```

DrvTIMER_SetEXTClockFreq

原型

void DrvTIMER_SetEXTClockFreq (uint32_t u32ClockValue)

描述

设定外部时钟频率，用于定时器时钟源。用户可以通过调用 DrvSYS_SelectIPClockSource (…) 来改变定时器时钟源为外部时钟源。

参数

u32ClockFreq [in]

设定外部时钟源频率(Hz)

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Set external clock value is 32 KHz */
DrvTIMER_SetEXTClockFreq (32000);
```

DrvTIMER_GetVersion

原型

uint32_t DrvTIMER_GetVersion (void)

描述

获取 Timer/WDT 驱动版本号。

头文件

Driver/DrvTIMER.h

返回值

版本号

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
/* Get the current version of Timer Driver */
u32Version = DrvTIMER_GetVersion ();
```

DrvWDT_Open

原型

void DrvWDT_Open (E_WDT_INTERVAL WDTlevel)

描述

使能 WDT 时钟并设定 WDT 超时时间。

参数

WDTlevel [in]

E_WDT_INTERVAL，枚举 WDT 超时间隔，详细的超时值请参考 WDT_INTERVAL 枚举。

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Set the WDT time-out interval is (2^16)*WDT_CLK */
DrvWDT_Open (E_WDT_LEVEL6);
```

DrvWDT_Close

原型

void DrvWDT_Close (void)

描述

该函数用于停止/禁止 WDT 相关功能。

参数

无

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Close Watch Dog Timer */
DrvWDT_Close ();
```

DrvWDT_InstallISR

原型

void DrvWDT_InstallISR (WDT_CALLBACK pvWDTISR)

描述

该函数用于安装 WDT 中断服务程序。

参数

pvWDTISR [in]

中断服务程序的函数指针。

头文件

Driver/DrvTIMER.h

返回值

无

示例

```
/* Install the WDT callback function */
DrvWDT_InstallISR ((WDT_CALLBACK)WDT_Callback);
```

DrvWDT_Ioctl

原型

int32_T DrvWDT_Ioctl (E_WDT_CMD uWDTCmd, uint32_t uArgument)

描述

该函数用于操作更多的 WDT 应用，可以是开始/停止 WDT，使能/禁止 WDT 中断功能，使能/禁止 WDT 超时唤醒功能，使能/禁止 WDT 超时时系统复位功能和设定 WDT 超时间隔。

参数

uWDTCmd [in]

E_WDT_CMD 命令，可以是下列命令之一：

E_WDT_IOC_START_TIMER，

E_WDT_IOC_STOP_TIMER，

E_WDT_IOC_ENABLE_INT，

E_WDT_IOC_DISABLE_INT，

E_WDT_IOC_ENABLE _WAKEUP，

E_WDT_IOC_DISABLE_WAKEUP，

E_WDT_IOC_RESET_TIMER，

E_WDT_IOC_ENABLE_RESET_FUNC，

E_WDT_IOC_DISABLE_RESET_FUNC，

E_WDT_IOC_SET_INTERVAL

uArgument [in]

为指定的 WDT 命令设定参数。

头文件

Driver/DrvTIMER.h

返回值

E_SUCCESS：	操作成功
E_ DRVWDT_CMD：	无效 WDT 命令

示例

```
/* Start to count WDT by calling WDT_IOC_START_TIMER command */
DrvWDT_Ioctl (E_WDT_IOC_START_TIMER, 0);
```

5. GPIO 驱动

5.1. GPIO 介绍

M051 系列 MCU 有多达 40 个通用 I/O 引脚，可以与特殊功能引脚复用。40 个引脚分配在五个口上，分别命名为 P0，P1，P2，P3 和 P4。每个口最多有 8 个引脚。

5.2. GPIO Feature

- 每个 GPIO 引脚都是独立的，有相应的寄存器位控制该引脚模式，功能与数据。
- 每个 I/O 引脚的输入/输出类型可以由软件独立的配置为输入，输出，开漏和准双向模式。

5.3. 类型定义

E_DRVGPIO_PORT

枚举标识符	值	描述
E_PORT0	0	定义 GPIO 端口 0
E_PORT1	1	定义 GPIO 端口 1
E_PORT2	2	定义 GPIO 端口 2
E_PORT3	3	定义 GPIO 端口 3
E_PORT4	4	定义 GPIO 端口 4

E_DRVGPIO_PIN

枚举标识符	值	描述
E_PIN0	0	定义 GPIO 引脚 0
E_PIN1	1	定义 GPIO 引脚 1
E_PIN2	2	定义 GPIO 引脚 2
E_PIN3	3	定义 GPIO 引脚 3
E_PIN4	4	定义 GPIO 引脚 4
E_PIN5	5	定义 GPIO 引脚 5
E_PIN6	6	定义 GPIO 引脚 6
E_PIN7	7	定义 GPIO 引脚 7

E_DRVGPIO_EXT_INT_PIN

枚举标识符	值	描述
E_EINT0_PIN	2	定义 GPIO 端口 3 的 2 引脚为外部中断引脚
E_EINT1_PIN	3	定义 GPIO 端口 3 的 3 引脚为外部中断引脚

E_DRVGPIO_IO

枚举标识符	值	描述
E_IO_INPIT	0	设定 GPIO 为输入模式
E_IO_OUTPUT	1	设定 GPIO 为输出模式
E_IO_OPENDRAIN	2	设定 GPIO 为开漏模式
E_IO_QUASI	3	设定 GPIO 为准双向模式

E_DRVGPIO_INT_TYPE

枚举标识符	值	描述
E_IO_RISING	0	设定中断触发信号类型为上升沿或高电平
E_IO_FALLING	1	设定中断触发信号类型为下降沿或低电平
E_IO_BOTH_EDGE	2	设定中断触发信号类型为上双边沿（上升沿和下降沿）

E_DRVGPIO_INT_MODE

枚举标识符	值	描述
E_MODE_EDGE	0	设定中断触发模式边沿触发
E_MODE_LEVEL	1	设定中断触发模式电平触发

E_DRVGPIO_DBCLKSRC

枚举标识符	值	描述
E_DBCLKSRC_HCLK	0	去抖计数器时钟源为 HCLK
E_DBCLKSRC_10K	1	去抖计数器时钟源为内部 10K RC 振荡器

E_DRVGPIO_FUNC

枚举标识符	值	描述
E_FUNC_GPIO	All GPIO pins	设定所有的 GPIO 引脚为 GPIO 功能
E_FUNC_CLKO	P3.6	使能频率输出功能
E_FUNC_I2C	P3.4 & P3.5	使能 I2C 功能
E_FUNC_SPI0 / E_FUNC_SPI1	P1.4~P1.7 / P0.4~P0.7	使能 SPI0/SPI1 功能
E_FUNC_ADC0 / E_FUNC_ADC1 / E_FUNC_ADC2 / E_FUNC_ADC3 / E_FUNC_ADC4 / E_FUNC_ADC5 / E_FUNC_ADC6 / E_FUNC_ADC7	P1.0 / P1.1 / P1.2 / P1.3 / P1.4 / P1.5 / P1.6 / P1.7	使能 ADC0/ADC1/ADC2/ADC3/ ADC4/ADC5/ADC6/ADC7 功能
E_FUNC_EXTINT0 / E_FUNC_EXTINT1	P3.2 / P3.3	使能外部 INT0/INT1 功能
E_FUNC_TMR0 / E_FUNC_TMR1 / E_FUNC_TMR2 / E_FUNC_TMR3	P3.4 / P3.5 / P1.0 / P1.1	使能 TIMER0/TIMER1/TIMER2/ TIMER3 为 Toggle/Counter 功能
E_FUNC_UART0 / E_FUNC_UART1	P3.0&P3.1&P0.2&P0.3 /	使能 UART0/UART1 功能

	P1.2&P1.3&P0.0&P0.1	
E_FUNC_PWM01 / E_FUNC_PWM23 / E_FUNC_PWM45 / E_FUNC_PWM67	P2.0&P2.1&P4.0&P4.1 / P2.2&P2.3&P4.2&P4.3 / P2.4&P2.5 / P2.6&P2.7 /	使能 PWM01/PWM23/PWM45/ PWM67 功能
E_FUNC_EBI_8B	P0.0 ~ P0.7 / P3.3 & P3.6 & P3.7 / P4.4 & P4.5	使能 EBI，数据宽度为 8 位
E_FUNC_EBI_16B	P2.0~P2.7 / P0.0 ~ P0.7 / P3.3 & P3.6 & P3.7 / P4.4 & P4.5	使能 EBI，数据宽度为 16 位
E_FUNC_ICE	P4.6 & P4.7	为 ICE CLK 和 DAT 配置引脚

5.4. 宏

_PORT_DOUT

原型

_PORT_DOUT (PortNum, PinNum)

描述

该宏定义用于控制指定引脚的 I/O 位输出控制寄存器。若该引脚被配置为输出模式，用户可以通过调用_PORT_DOUT 宏来设定指定引脚的输出数据。或者当 GPIO 引脚配置为输入模式时，通过直接调用_PORT_DOUT 来获取输入数据。

参数

PortNum [in]

指定 GPIO 端口。可以为 0~4，对应于 Port0/1/2/3/4。

PinNum [in]

指定 GPIO 端口的引脚编号。可以为 0~7。

头文件

Driver/DrvGPIO.h

示例

```
/* Configure Port0-1 to output mode */
DrvGPIO_Open (E_PORT0, E_PIN1, E_IO_OUTPUT);
/* Set Port0-1 to high */
_PORT_DOUT (0, 1) = 1;
/* ..... */
/* Configure Port1-3 to input mode */
uint8_t u8PinValue;
DrvGPIO_Open (E_ PORT1, E_PIN3, E_IO_INPUT);
/* Get Port1-3 pin value */
```

```
u8PinValue = _PORT_DOUT (1, 3);
```

P0[n]_DOUT / P1[n]_DOUT / P2[n]_DOUT / P3[n]_DOUT / P4[n]_DOUT

原型

P00_DOUT~P07_DOUT / P10_DOUT~P17_DOUT / P20_DOUT~P27_DOUT / P30_DOUT~P37_DOUT / P40_DOUT~P47_DOUT

描述

这些宏定义跟_PORT_DOUT 宏一样，只是没有任何参数。用户可以直接使用这些宏定义，比如 P00_DOUT，往指定的引脚输出数据，或者从指定的引脚获取该引脚值。

参数

无

头文件

Driver/DrvGPIO.h

示例

```
/* Configure Port0-1 to output mode */
DrvGPIO_Open (E_PORT0, E_PIN1, E_IO_OUTPUT);
/* Set Port0-1 to high */
_PORT_DOUT (0, 1) = 1;
/* ..... */
/* Configure Port1-3 to input mode */
uint8_t u8PinValue;
DrvGPIO_Open (E_ PORT1, E_PIN3, E_IO_INPUT);
/* Get Port1-3 pin value */
u8PinValue = _PORT_DOUT (1, 3);
```

5.5. 函数

DrvGPIO_Open

原型

```
void DrvGPIO_Open (
    E_DRVGPIO_PORT    port,
    E_DRVGPIO_PIN     pin,
    E_DRVGPIO_IO      IOMode
)
```

描述

配置指定的 GPIO 引脚为指定的操作模式。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

IOMode [in]

E_DRVGPIO_MODE，设定指定的 GPIO 引脚为 E_IO_INPUT，E_IO_OUTPUT，E_IO_OPENDRAIN 或者 E_IO_QUASI 模式。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Configure Port0-0 to GPIO output mode and Port0-1 to GPIO input mode*/
DrvGPIO_Open (E_PORT0, E_PIN0, E_IO_OUTPUT);
DrvGPIO_Open (E_PORT0, E_PIN1, E_IO_INPUT);
```

DrvGPIO_Close

原型

void DrvGPIO_Close (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

关闭指定的 GPIO 引脚功能，并设定该引脚为准双向模式。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Close Port0-0 function and set to default quasi-bidirectional mode */
DrvGPIO_Close (E_PORT0, E_PIN0);
```

DrvGPIO_SetBit

原型

int32_t DrvGPIO_SetBit (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

设定指定的 GPIO 引脚为 1。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：	操作成功

示例

```
/* Configure Port0-0 as GPIO output mode*/
DrvGPIO_Open (E_PORT0, E_PIN0, E_IO_OUTPUT);
/* Set Port0-0 to 1(high) */
DrvGPIO_SetBit (E_PORT0, E_PIN0);
```

DrvGPIO_GetBit

原型

int32_t DrvGPIO_GetBit (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

获取指定输入 GPIO 引脚的值。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

指定的输入引脚的值：0/1

示例

```
int32_t i32BitValue;
/* Configure Port0-1 as GPIO input mode*/
DrvGPIO_Open (E_PORT0, E_PIN1, E_IO_INPUT);
i32BitValue = DrvGPIO_GetBit (E_PORT0, E_PIN1);
if  (u32BitValue == 1)
{
   printf("Port0-1 pin status is high.\n");
}else
{
   printf("Port0-1 pin status is low.\n");
}
```

DrvGPIO_ClrBit

原型

int32_t DrvGPIO_ClrBit (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

设定指定的 GPIO 引脚为 0。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：　　操作成功

示例

```
/* Configure Port0-0 as GPIO output mode*/
DrvGPIO_Open (E_PORT0, E_PIN0, E_IO_OUTPUT);
/* Set Port0-0 to 0(low) */
DrvGPIO_ClrBit (E_PORT0, E_PIN0);
```

DrvGPIO_SetPortBitsts

原型

int32_tDrvGPIO_SetPortBits (E_DRVGPIO_PORT port, int32_t i32PortValue)

描述

设定输出端口值到指定的 GPIO 端口。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

i32PortValue [in]

输出数据值，可以是 0~0xFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Set the output value of GPIO Port0 to 0x12 */
DrvGPIO_SetPortBits (E_PORT0, 0x12);
```

DrvGPIO_GetPortBits

原型

int32_t DrvGPIO_GetPortBits (E_DRVGPIO_PORT port)

描述

从指定的 GPIO 端口获取输入端口值。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

头文件

Driver/DrvGPIO.h

返回值

指定的输入端口的值：0 ~ 0xFF

示例

```
/* Get the GPIO Port0 input data value */
int32_t i32PortValue;
i32PortValue = DrvGPIO_GetPortBits (E_PORT0);
```

DrvGPIO_GetDoutBit

原型

int32_t DrvGPIO_GetDoutBit (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

获取指定数据输出寄存器某一位的值。如果该位为 1，说明相应引脚输出高电平数据，否则输出低电平数据。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器某一位的值：0/1

示例

```
/* Get the Port0-1 data output value */
int32_t i32BitValue;
i32BitValue = DrvGPIO_GetDoutBit (E_PORT0, E_PIN1);
```

DrvGPIO_GetPortDoutBits

原型

int32_t DrvGPIO_GetPortDoutBits (E_DRVGPIO_PORT port)

描述

获取指定数据输出寄存器的值。如果返回的端口值中相应位为 1，说明相应引脚输出高电平数据，否则输出低电平数据。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0 ~ 0xFF

示例

```
/* Get the GPIO Port0 data output value */
int32_t i32PortValue;
i32PortValue = DrvGPIO_GetPortDoutBits (E_PORT0);
```

DrvGPIO_SetBitMask

原型

int32_t DrvGPIO_SetBitMask (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

该函数用于保护相应 GPIO 引脚的写功能。当设置了某一位屏蔽，写信号被屏蔽，对被保护位的写操作将被忽略。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：	操作成功

示例

```
/* Protect Port0-0 write data function */
DrvGPIO_SetBitMask (E_PORT0, E_PIN0);
```

DrvGPIO_GetBitMask

原型

int32_t DrvGPIO_GetBitMask (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

获取指定的数据输出写屏蔽寄存器某一位的值。如果该位值是 1，表示相应位被保护。写数据到该位将被忽略。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

指定的寄存器某一位的值：0/1

示例

```
/* Get the bit value from GPIO Port0 Data Output Write Mask Resister */
int32_t i32MaskValue;
i32MaskValue = DrvGPIO_GetBittMask (E_PORT0, E_PIN0);
/* If (i32MaskValue = 1), its meaning Port0-0 is write protected */
```

DrvGPIO_ClrBitMask

原型

int32_t DrvGPIO_ClrBitMask (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

该函数用于清除相应 GPIO 引脚的写保护功能。某一屏蔽位被清除后，写数据到该位是有效的。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2,

E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Remove the Port0-0 write protect function */
DrvGPIO_ClrBitMask (E_PORT0, E_PORT0);
```

DrvGPIO_SetPortMask

原型

int32_t DrvGPIO_SetPortMask (E_DRVGPIO_PORT port, int32_t i32PortValue)

描述

该函数用于保护相应 GPIO 某些引脚的写功能。当设置了某些位被屏蔽，对被保护位的写操作将被忽略。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

i32PortValue [in]

指定 GPIO 端口的引脚。可以是 0~0xFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Protect Port0-0 and Port0-4 write data function */
DrvGPIO_SetPortMask (E_PORT0, 0x11);
```

DrvGPIO_GetPortMask

原型

int32_t DrvGPIO_GetPortMask (E_DRVGPIO_PORT port)

描述

获取指定的数据输出写屏蔽寄存器的值。如果返回的端口值中相应位为 1，表示这些位被保护。写数据到这些位将被忽略。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0~0xFF

示例

```
/* Get the port value from GPIO Port0 Data Output Write Mask Resister */
int32_t i32MaskValue;
i32MaskValue = DrvGPIO_GetPortMask (E_PORT0);
/* If (i32MaskValue = 0x11), its meaning Port0-0 and Port0-4 are protected */
```

DrvGPIO_ClrPortMask

原型

int32_t DrvGPIO_ClrPortMask (E_DRVGPIO_PORT port, int32_t i32PortValue)

描述

该函数用于清除相应 GPIO 引脚的写保护功能。那些屏蔽位被清除后，写数据到相应位是有效的。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

i32PortValue [in]

指定 GPIO 端口的引脚。可以是 0~0xFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Remove the Port0-0 and Port0-4 write protect function */
DrvGPIO_ClrPortMask (E_PORT0, 0x11);
```

DrvGPIO_EnableDebounce

原型

int32_t DrvGPIO_EnableDebounce (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

使能指定 GPIO 输入引脚的去抖动功能。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Enable Port0-0 interrupt de-bounce function */
DrvGPIO_EnableDebounce (E_PORT0, 0);
```

DrvGPIO_DisableDebounce

原型

int32_t DrvGPIO_DisableDebounce (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

禁止指定 GPIO 输入引脚的去抖动功能。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Disable Port0-0 interrupt de-bounce function */
DrvGPIO_DisableDebounce (E_PORT0, 0);
```

DrvGPIO_SetDebounceTime

原型

```
int32_t DrvGPIO_SetDebounceTime (
    uint32_t                u32CycleSelection,
    E_DRVGPIO_DBCLKSRC      ClockSource
)
```

描述

基于去抖动计数器时钟源设定中断去抖动采样时间。假如去抖动时钟源为内部 10KHz RC 振荡器，且采样周期选择 4，则目标去抖动时间为(2^4)*(1/(10*1000)) s = 16*0.0001 s = 1600 us，系统每隔 1600us 会去采样一次中断输入。

参数

u32CycleSelection [in]

采样周期数选择，取值范围是 0 ~ 15，目标去抖时间是

(2^(u32CycleSelection))*(ClockSource)秒。

ClockSource [in]

E_DRVGPIO_DBCLKSRC，可以是 DBCLKSRC_HCLK or DBCLKSRC_10K。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

E_DRVGPIO_ARGUMENT： 无效参数

示例

```
/* Set de-bounce sampling time to 1600 us. (2^4)*(10 KHz) */
DrvGPIO_SetDebounceTime (4, E_DBCLKSRC_10K);
```

DrvGPIO_GetDebounceSampleCycle

原型

int32_t DrvGPIO_GetDebounceSampleCycle (void)

描述

该函数用于获取选择的去抖采样周期数。

参数

无

头文件

Driver/DrvGPIO.h

返回值

选择的采样周期数：0 ~ 15。

示例

```
int32_t i32CycleSelection;
i32CycleSelection = DrvGPIO_GetDebounceSampleCycle ();
/* If i32CycleSelection is 4 and clock source from 10 KHz.          */
/* It's meaning to sample interrupt input once per 16*100us.    */
```

DrvGPIO_EnableInt

原型

```
int32_t DrvGPIO_EnableInt (
    E_DRVGPIO_PORT          port,
    E_DRVGPIO_PIN           pin,
    E_DRVGPIO_INT_TYPE      Type,
    E_DRVGPIO_INT_MODE  Mode
)
```

描述

使能指定 GPIO 引脚的中断功能。用作外部中断的 Port3-2 和 Port3-3 除外。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。

Type [in]

E_DRVGPIO_INT_TYPE，指定中断触发类型。可以是 E_IO_RISING，E_IO_FALLING 或者 E_IO_BOTH_EDGE，表示中断触发类型为：上升沿/高电平，下降沿/低电平或双边沿(上升沿和下降沿)。如果中断模式是 E_MODE_LEVEL，且中断类型是 E_BOTH_EDGE，对该 API 的调用将被忽略。

Mode [in]

E_DRVGPIO_INT_MODE，指定中断模式。可以是 E_MODE_EDGE 或 E_MODE_LEVEL，控制中断是边沿触发或电平触发。如果中断模式是 E_MODE_LEVEL，且中断类型是 E_BOTH_EDGE，对该 API 的调用将被忽略。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：	操作成功
E_DRVGPIO_ARGUMENT：	无效参数

示例

```
/* Enable Port1-3 interrupt function and its rising and edge trigger. */
DrvGPIO_EnableInt (E_PORT1, E_PIN3, E_IO_RISING, E_MODE_EDGE);
```

DrvGPIO_DisableInt

原型

int32_t DrvGPIO_DisableInt (E_DRVGPIO_PORT port, E_DRVGPIO_PIN pin)

描述

禁止指定 GPIO 引脚的中断功能。用作外部中断的 Port3-2 和 Port3-3 除外。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

pin [in]

指定 GPIO 端口的引脚。可以是 E_PIN0, E_PIN2 ... ~ E_PIN7。但是 Port3-2 和 Port3-3 仅用于外部中断 0/1。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Disable Port1-3 interrupt function. */
DrvGPIO_DisableInt (E_PORT1, E_PIN3);
```

DrvGPIO_SetIntCallback

原型

```
void DrvGPIO_SetIntCallback (
    P0P1_CALLBACK       pfP0P1Callback,
    P2P3P4_CALLBACK     pfP2P3P4Callback
)
```

描述

为 Port0/1 和 Port2/3/4 安装中断回调函数。

参数

pfP0P1Callback [in]

GPIO Port0/1 回调函数的函数指针。

pfP2P3P4Callback [in]

GPIO Port2/3/4 回调函数的函数指针。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Set GPIO Port0/1 and Port2/3/4 interrupt callback functions */
DrvGPIO_SetIntCallback (P0P1Callback, P2P3P4Callback);
```

DrvGPIO_EnableEINT

原型

Int32_t DrvGPIO_EnableEINT (

E_DRVGPIO_EXT_INT_PIN	pin,
E_DRVGPIO_INT_TYPE	Type,
E_DRVGPIO_INT_MODE	Mode,
EINT_CALLBACK	pfEINTCallback

)

描述

使能外部 GPIO 中断 Port3-2 或 Port3-3 的中断功能。

参数

pin [in]

指定 GPIO 端口 3 的外部中断引脚。可以是 E_EINT0_PIN (Port3-2) 或 E_EINT1_PIN (Port3-3)。

Type [in]

E_DRVGPIO_INT_TYPE，指定中断触发类型。可以是 E_IO_RISING，E_IO_FALLING 或者 E_IO_BOTH_EDGE，表示中断触发类型为：上升沿/高电平，下降沿/低电平或双边沿(上升沿和下降沿)。如果中断模式是 E_MODE_LEVEL，且中断类型是 E_BOTH_EDGE，对该 API 的调用将被忽略。

Mode [in]

E_DRVGPIO_INT_MODE，指定中断模式。可以是 E_MODE_EDGE 或 E_MODE_LEVEL，控制中断是边沿触发或电平触发。如果中断模式是 E_MODE_LEVEL，且中断类型是 E_BOTH_EDGE，对该 API 的调用将被忽略。

pfEINTCallback [in]

外部中断回调函数的函数指针

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：	操作成功

示例

```
/* Enable external INT0 interrupt as both-edge trigger. */
DrvGPIO_EnableEINT (E_EINT0_PIN, E_IO_BOTH_EDGE, E_MODE_EDGE,
EINT0Callback);
```

DrvGPIO_DisableEINT

原型

Int32_t DrvGPIO_DisableEINT (E_DRVGPIO_EXT_INT_PIN pin)

描述

禁止外部 GPIO 中断 Port3-2 或 Port3-3 的中断功能。

参数

pin [in]

指定 GPIO 端口 3 的外部中断引脚。可以是 E_EINT0_PIN (Port3-2) 或 E_EINT1_PIN (Port3-3)。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS： 操作成功

示例

```
/* Disable external INT0 interrupt function. */
DrvGPIO_DisableEINT (E_EINT0_PIN);
```

DrvGPIO_GetIntStatus

原型

int32_t DrvGPIO_GetIntStatus (E_DRVGPIO_PORT port)

描述

从指定的中断触发源指示寄存器获取端口值。如果返回的端口值的某一位是 1，表示对应于该位的 GPIO 引脚发生了中断，否则没有中断在对应于该位的 GPIO 引脚发生。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_PORT0, E_PORT1, E_PORT2, E_PORT3 和 E_PORT4。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0 ~ 0xFF

示例

```
/* Get GPIO Port0 interrupt status. */
int32_t i32INTStatus;
i32INTStatus = DrvGPIO_GetIntStatus (E_PORT0);
```

DrvGPIO_InitFunction

原型

int32_t DrvGPIO_InitFunction (E_DRVGPIO_FUNC function)

描述

初始化指定的功能，并为该功能配置相关引脚。

参数

function [in]

DRVGPIO_FUNC，指定相关的 GPIO 引脚为特殊功能引脚。

可以是：

E_FUNC_GPIO,

E_FUNC_CLKO,

E_FUNC_I2C,

E_FUNC_SPI0 / E_FUNC_SPI1,

E_FUNC_ADC0 / E_FUNC_ADC1 / E_FUNC_ADC2 / E_FUNC_ADC3 /

E_FUNC_ADC4 / E_FUNC_ADC5 / E_FUNC_ADC6 / FUNC_ADC7,

E_FUNC_EXTINT0 / E_FUNC_EXTINT1,

E_FUNC_TMR0 / E_FUNC_TMR1 / E_FUNC_TMR2 / E_FUNC_TMR3,

E_FUNC_UART0 / E_FUNC_UART1,

E_FUNC_PWM01 / E_FUNC_PWM23 / E_FUNC_PWM45 / E_FUNC_PWM67,

E_FUNC_EBI_8B / E_FUNC_EBI_16B,

E_FUNC_ICE

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：	操作成功
E_DRVGPIO_ARGUMENT：	无效参数

示例

```
/* Init UART0 function */
DrvGPIO_InitFunction (E_FUNC_UART0);
```

DrvGPIO_GetVersion

原型

uint32_t DrvGPIO_GetVersion (void)

描述

该函数用于返回 GPIO 驱动版本号。

头文件

Driver/DrvGPIO.h

返回值

GPIO 驱动版本号

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
/* Get the current version of GPIO Driver */
int32_t i32GPIOVer;
i32GPIOVer = DrvGPIO_GetVersion ();
```

6. ADC 驱动

6.1. ADC 介绍

M051 系列 MCU 包含一个 12 比特 8 通道的逐次逼近型模数转换器(SAR A/D converter)。转换一个采样数据需要 27 个 ADC 时钟，ADC 最大输入时钟是 16M(5V)。A/D 转换支持 4 种操作模式：单次，突发，单周期扫描和连续扫描模式。A/D 转换可以通过软件或外部 STADC/P3.2 引脚启动。本文档介绍怎样使用 ADC 驱动。.

6.2. ADC 特性

模数转换器包含下面的特性：

- 模拟输入电压范围： 0~Vref (Max to 5.0V)
- 12 比特分辨率
- 8 个模拟输入通道
- 最大 ADC 时钟频率是 16MHz
- 四种操作模式
  1. 单次模式
  2. 突发模式
  3. 单周期扫描模式
  4. 连续扫描模式
- A/D 转换可以由下列动作启动
  1. 软件写 1 到 ADST 位
  2. 外部引脚 STADC
- 转换结果可以跟指定的值比较，并且当转换结果与比较寄存器中的值匹配时提供中断功能
- 应用程序接口包括 ADC 转换条件的设定和转换结果的获取
- 通道 7 支持 2 个输入源：外部模拟电压和内部固定带隙电压
- 支持自校正以减小转换误差
- 支持单端和差分输入信号

6.3. 类型定义

E_ADC_INPUT_MODE

枚举标识符	值	描述
ADC_SINGLE_END	0	ADC 单端输入
ADC_DIFFERENTIAL	1	ADC 差分输入

E_ADC_OPERATION_MODE

枚举标识符	值	描述
ADC_SINGLE_OP	0	单次操作模式
ADC_BURST_OP	1	突发操作模式
ADC_SINGLE_CYCLE_OP	2	单周期扫描模式
ADC_CONTINUOUS_OP	3	连续扫描模式

E_ADC_CLK_SRC

枚举标识符	值	描述
EXTERNAL_12MHZ	0	外部 12MHz 时钟
INTERNAL_PLL	1	内部 PLL 时钟
INTERNAL_RC22MHZ	2	内部 22MHz 时钟

E_ADC_EXT_TRI_COND

枚举标识符	值	描述
LOW_LEVEL	0	低电平触发
HIGH_LEVEL	1	高电平触发
FALLING_EDGE	2	下降沿触发
RISING_EDGE	3	上升沿触发

E_ADC_CH7_SRC

枚举标识符	值	描述
EXTERNAL_INPUT_SIGNAL	0	外部输入信号
INTERNAL_BANDGAP	1	内部 bandgap 电压

E_ADC_CMP_CONDITION

枚举标识符	值	描述
LESS_THAN	0	小于比较数据
GREATER_OR_EQUAL	1	大于或等于比较数据

6.4. 宏

_DRVADC_CONV

原型

void _DRVADC_CONV (void);

描述

通知 ADC 开始 A/D 转换。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Start an A/D conversion */
_DRVADC_CONV();
```

_DRVADC_GET_ADC_INT_FLAG

原型

uint32_t _DRVADC_GET_ADC_INT_FLAG (void);

描述

获取 ADC 中断标志状态。

头文件

Driver/DrvADC.h

返回值

0：没有 ADC 中断发生。

1：发生了 ADC 中断。

示例

```
/* Get the status of ADC interrupt flag */
if(_DRVADC_GET_ADC_INT_FLAG())
   printf("ADC interrupt occurs.\n");
```

_DRVADC_GET_CMP0_INT_FLAG

原型

uint32_t _DRVADC_GET_CMP0_INT_FLAG (void);

描述

获取 ADC 比较器 0 中断标志状态。

头文件

Driver/DrvADC.h

返回值

0：ADC 比较器 0 中断没有发生。

1：ADC 比较器 0 中断发生。

示例

```
/* Get the status of ADC comparator 0 interrupt flag */
if(_DRVADC_GET_CMP0_INT_FLAG())
    printf("ADC comparator 0 interrupt occurs.\n");
```

_DRVADC_GET_CMP1_INT_FLAG

原型

uint32_t _DRVADC_GET_CMP1_INT_FLAG (void);

描述

获取 ADC 比较器 1 中断标志状态。

头文件

Driver/DrvADC.h

返回值

0：ADC 比较器 1 中断没有发生。

1：ADC 比较器 1 中断发生。

示例

```
/* Get the status of ADC comparator 1 interrupt flag */
if(_DRVADC_GET_CMP1_INT_FLAG())
    printf("ADC comparator 1 interrupt occurs.\n");
```

_DRVADC_CLEAR_ADC_INT_FLAG

原型

void _DRVADC_CLEAR_ADC_INT_FLAG (void);

描述

清除 ADC 中断标志。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Clear the ADC interrupt flag */
_DRVADC_CLEAR_ADC_INT_FLAG();
```

_DRVADC_CLEAR_CMP0_INT_FLAG

原型

void _DRVADC_CLEAR_CMP0_INT_FLAG (void);

描述

清除 ADC 比较器 0 中断标志。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Clear the ADC comparator 0 interrupt flag */
_DRVADC_CLEAR_CMP0_INT_FLAG();
```

_DRVADC_CLEAR_CMP1_INT_FLAG

原型

void _DRVADC_CLEAR_CMP1_INT_FLAG (void);

描述

清除 ADC 比较器 1 中断标志。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Clear the ADC comparator 1 interrupt flag */
_DRVADC_CLEAR_CMP1_INT_FLAG();
```

6.5. 函数

DrvADC_Open

原型

```
void DrvADC_Open (
    E_ADC_INPUT_MODE InputMode,
    E_ADC_OPERATION_MODE OpMode,
    uint8_t u8ChannelSelBitwise,
    E_ADC_CLK_SRC ClockSrc,
    uint8_t u8AdcDivisor
);
```

描述

使能 ADC 功能并且完成相关设定。

参数

InputMode [in]

指定模拟信号输入类型。可以是单端或者差分输入。

ADC_SINGLE_END：单端输入模式

ADC_DIFFERENTIAL：差分输入模式

OpMode [in]

指定操作模式。可以是单次，突发，单周期扫描或者连续扫描模式。

ADC_SINGLE_OP：单次模式

ADC_BURST_OP：突发模式

ADC_SINGLE_CYCLE_OP：单周期扫描模式

ADC_CONTINUOUS_OP：连续扫描模式

u8ChannelSelBitwise [in]

指定输入通道。如果该值为 0，通道 0 会自动使能。在单次模式下，如果软件使能超过 1 个通道，只有通道值最低的通道会进行转换，而其他使能的通道将会被忽略，

例如，如果用户在单次模式下使能通道 2，3 和 4，只有通道 2 会进行转换。在差分输入模式下，对于两个相应的通道，只有其中一个需要被选择，转换结果会存放在

被选择通道的数据寄存器中。例如，在单端输入模式下，0x8 表示通道 3 被选择；而在差分输入模式下，它表示差分通道 1 的一对通道被选择。

ClockSrc [in]

指定 ADC 时钟源。

EXTERNAL_12MHZ：外部 12MHz 晶振

INTERNAL_PLL：内部 PLL 输出

INTERNAL_RC22MHZ：内部 22MHz RC 振荡器

u8AdcDivisor [in]

决定 ADC 时钟频率。取值范围是 0~0xFF。

ADC 时钟频率 = ADC 时钟源频率 / (u8AdcDivisor + 1)

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* single end input, single operation mode, channel 5 is selected, ADC clock frequency = 12MHz/(5+1) */
DrvADC_Open(ADC_SINGLE_END, ADC_SINGLE_OP, 0x20, EXTERNAL_12MHZ, 5);
```

DrvADC_Close

原型

void DrvAdc_Close (void);

描述

关闭 ADC 功能。禁止 ADC 时钟和 ADC 中断，ADC 比较功能和它的中断。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Close the ADC function */
DrvAdc_Close();
```

DrvADC_SetADCChannel

原型

```
void DrvADC_SetADCChannel (
    uint8_t u8ChannelSelBitwise
);
```

描述

选择 ADC 输入通道。

参数

u8ChannelSelBitwise [in]

指定模拟输入通道。如果该值为 0，通道 0 会自动使能。在单次模式下，如果软件使能超过 1 个通道，只有通道值最低的通道会进行转换，而其他使能的通道将会被
忽略。在差分输入模式下，对于两个相应的通道，只有其中一个需要被选择，转换结果会存放在被选择通道的数据寄存器中。例如，在单端输入模式下，0x8 表示通道 3 被选择；而在差分输入模式下，它表示差分通道 1 的一对通道被选择。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* In single-end input mode, this function selects channel 0 and channel 2; In differential
input mode, it selects channel pair 0 and channel pair 1. */
DrvADC_SetADCChannel (0x5);
```

DrvADC_ConfigADCChannel7

原型

```
void DrvADC_ConfigADCChannel7 (ADC_CH7_SRC Ch7Src);
```

描述

选择通道 7 的输入信号源。

参数

Ch7Src [in]

指定模拟信号输入源。

EXTERNAL_INPUT_SIGNAL：外部模拟输入

INTERNAL_BANDGAP：内部带隙电压

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Select the external analog input as the source of channel 7 */
DrvADC_ConfigADCChannel7(EXTERNAL_INPUT_SIGNAL);
```

DrvADC_SetADCInputMode

原型

void DrvADC_SetADCInputMode (E_ADC_INPUT_MODE InputMode);

描述

设定 ADC 输入模式。

参数

InputMode [in]

指定模拟信号输入类型。

ADC_SINGLE_END：单端输入模式

ADC_DIFFERENTIAL：差分输入模式

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* The following statement indicates that the external analog input is a single-end input */
```

```
DrvADC_SetADCInputMode(ADC_SINGLE_END);
```

DrvADC_SetADCOperationMode

原型

void DrvADC_SetADCOperationMode (E_ADC_OPERATION_MODE OpMode);

描述

设定 ADC 操作模式。

参数

OpMode [in]

指定操作模式。

ADC_SINGLE_OP：单次模式

ADC_BURST_OP：突发模式

ADC_SINGLE_CYCLE_OP：单周期扫描模式

ADC_CONTINUOUS_OP：连续扫描模式

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* The following statement configures the single mode as the operation mode */
DrvADC_SetADCOperationMode(ADC_SINGLE_OP);
```

DrvADC_SetADCClkSrc

原型

void DrvADC_SetADCClkSrc (E_ADC_CLK_SRC ClockSrc);

描述

选择 ADC 时钟源。

参数

ClockSrc [in]

指定 ADC 时钟源。

EXTERNAL_12MHZ：外部 12MHz 晶振

INTERNAL_PLL：内部 PLL 输出

INTERNAL_RC22MHZ：内部 22MHz RC 振荡器

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Select the external 12MHz crystal as the clock source of ADC */
DrvADC_SetADCClkSrc (EXTERNAL_12MHZ);
```

DrvADC_SetADCDivisor

原型

```
void DrvADC_SetADCDivisor (uint8_t u8AdcDivisor);
```

描述

设定 ADC 时钟的除频值来决定 ADC 时钟频率。

ADC 时钟频率 = ADC 时钟源频率 / (u8AdcDivisor + 1)

参数

u8AdcDivisor [in]

指定除频值。取值范围是 0~0xFF。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* The clock source of ADC is from external 12MHz crystal. The ADC clock frequency is
2MHz. */
DrvADC_SetADCClkSrc (EXTERNAL_12MHZ);
DrvADC_SetADCDivisor (5);
```

DrvADC_EnableADCInt

原型

```
void DrvADC_EnableADCInt (
```

```
        DRVADC_ADC_CALLBACK Callback,
        uint32_t u32UserData
);
```

描述

使能 ADC 中断，安装中断回调函数。当 ADC 中断发生，中断回调函数会被执行。当 ADC 中断使能，如果有以下情况之一发生，ADC 中断会被触发。

1. 单次模式下，指定通道 A/D 转换完成。
2. 单周期扫描或连续扫描模式下，所有选择的通道 A/D 转换完成。

参数

Callback [in]

ADC 中断的回调函数。

u32UserData [in]

传给回调函数的参数。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* ADC interrupt callback function */
void AdcIntCallback(uint32_t u32UserData)
{
   gu8AdcIntFlag = 1;
}
/* Enable the ADC interrupt and setup the callback function. The parameter 0 will be passed to the callback function. */
DrvADC_EnableADCInt(AdcIntCallback, 0);
```

DrvADC_DisableADCInt

原型

```
void DrvADC_DisableADCInt (void);
```

描述

禁止 ADC 中断。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the ADC interrupt */
DrvADC_DisableADCInt();
```

DrvADC_EnableADCCmp0Int

原型

```
void DrvADC_EnableADCCmp0Int (
    DRVADC_ADCMP0_CALLBACK Callback,
    uint32_t u32UserData
);
```

描述

使能 ADC 比较器 0 的中断，并且安装中断回调函数。如果转换结果满足 DrvADC_EnableADCCmp0()设定的比较条件，比较器 0 的中断会被触发，中断回调函数会被执行。

参数

Callback [in]

ADC 比较器 0 中断回调函数。

u32UserData [in]

传给回调函数的参数。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* ADC comparator 0 interrupt callback function */
void Cmp0IntCallback(uint32_t u32UserData)
{
```

```
    gu8AdcCmp0IntFlag = 1;
}
int32_t main()
{
    …
    /* Enable the ADC comparator 0 interrupt and setup the callback function. The parameter
    0 will be passed to the callback function. */
    DrvADC_EnableADCCmp0Int(Cmp0IntCallback, 0);
}
```

DrvADC_DisableADCCmp0Int

原型

void DrvADC_DisableAdcmp0Int (void);

描述

禁止 ADC 比较器 0 中断。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the ADC comparator 0 interrupt */
DrvADC_DisableADCCmp0Int();
```

DrvADC_EnableADCCmp1Int

原型

```
void DrvADC_EnableADCCmp1Int (
    DRVADC_ADCMP1_CALLBACK Callback,
    uint32_t u32UserData
);
```

描述

使能 ADC 比较器 1 的中断，并且安装中断回调函数。如果转换结果满足

DrvADC_EnableADCCmp1()设定的比较条件，比较器 1 的中断会被触发，中断回调函数会被执行。

参数

Callback [in]

ADC 比较器 1 中断回调函数。

u32UserData [in]

传给回调函数的参数。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* ADC comparator 1 interrupt callback function */
void Cmp1IntCallback(uint32_t u32UserData)
{
  gu8AdcCmp1IntFlag = 1;
}
int32_t main()
{
  …
  /* Enable the ADC comparator 1 interrupt and setup the callback function. The parameter
  0 will be passed to the callback function. */
  DrvADC_EnableADCCmp1Int(Cmp1IntCallback, 0);
}
```

DrvADC_DisableADCCmp1Int

原型

void DrvADC_DisableADCCmp1Int (void);

描述

禁止 ADC 比较器 1 中断。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the ADC comparator 1 interrupt */
DrvADC_DisableADCCmp1Int();
```

DrvADC_GetConversionRate

原型

uint32_t DrvADC_GetConversionRate (void);

描述

获取 A/D 转换的速率。完成一次 A/D 转换大约需要 27 个 ADC 时钟周期。

参数

无

头文件

Driver/DrvADC.h

返回值

返回转换频速率。单位是 sample/second。

示例

```
/* The clock source of ADC is from external 12MHz crystal. The ADC clock frequency is
2MHz. The conversion rate is about 74K sample/second */
DrvADC_SetADCClkSrc (EXTERNAL_12MHZ);
DrvADC_SetADCDivisor (5);
/* Get the conversion rate */
printf("Conversion rate: %d samples/second\n", DrvADC_GetConversionRate());
```

DrvADC_EnableExtTrigger

原型

void DrvADC_EnableExtTrigger (E_ADC_EXT_TRI_COND TriggerCondition);

描述

允许外部触发引脚(P3.2)做 ADC 的触发源。

参数

TriggerCondition [in]

指定触发条件。触发条件可以是低电平/高电平/下降沿/上升沿。

LOW_LEVEL：低电平

HIGH_LEVEL：高电平

FALLING_EDGE：下降沿

RISING_EDGE：上升沿

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Use PB8 pin as the external trigger pin. The trigger condition is low level trigger. */
DrvADC_EnableExtTrigger(LOW_LEVEL);
```

DrvADC_DisableExtTrigger

原型

void DrvADC_DisableExtTrigger (void);

描述

禁止外部 ADC 触发。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the ADC external trigger source */
DrvADC_DisableExtTrigger ();
```

DrvADC_StartConvert

原型

void DrvADC_StartConvert(void);

描述

清除 ADC 中断标志(ADF)，开始 A/D 转换。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Clear ADF bit and start converting */
DrvADC_StartConvert();
```

DrvADC_StopConvert

原型

void DrvADC_StopConvert(void);

描述

停止 A/D 转换。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Stop converting */
DrvADC_StopConvert();
```

DrvADC_IsConversionDone

原型

uint32_t DrvADC_IsConversionDone (void);

描述

检查转换是否完成。

参数

无

头文件

Driver/DrvADC.h

返回值

TURE	转换结束
FALSE	转换正在进行

示例

```
/* If the ADC interrupt is not enabled, user can call this function to check the state of conversion action */
/* Start A/D conversion */
DrvADC_StartConvert();
/* Wait conversion done */
while(!DrvADC_IsConversionDone());
```

DrvADC_GetConversionData

原型

int32_t DrvADC_GetConversionData (uint8_t u8ChannelNum);

描述

获取指定 ADC 通道的转换结果数据。

参数

u8ChannelNum [in]

指定 ADC 通道。取值范围是 0~7。

头文件

Driver/DrvADC.h

返回值

32 比特转换结果。通过扩展原始的 12 比特转换结果得到。

示例

```
/* Get the conversion result of ADC channel 3 */
u32AdcData = DrvADC_GetConversionData(3);
```

DrvADC_IsDataValid

原型

uint32_t DrvADC_IsDataValid (uint8_t u8ChannelNum);

描述

检查 A/D 转换的数据是否有效。

参数

u8ChannelNum [in]

指定 ADC 通道。取值范围是 0~7。

头文件

Driver/DrvADC.h

返回值

TURE：数据有效

FALSE：数据无效

示例

```
/* Check if the data of channel 3 is valid. */
If( DrvADC_IsDataValid(3) )
  u32ConversionData = DrvADC_GetConversionData(u8ChannelNum); /* Get the data */
```

DrvADC_IsDataOverrun

原型

uint32_t DrvADC_IsDataOverrun (uint8_t u8ChannelNum);

描述

检查转换的数据是否溢出。

参数

u8ChannelNum [in]

指定 ADC 通道。取值范围是 0~7。

头文件

Driver/DrvADC.h

返回值

TURE：溢出

FALSE：没有溢出

示例

```
/* Check if the data of channel 3 is overrun. */
If(DrvADC_IsDataOverrun(3) )
   printf("The data has been overwritten.\n");
```

DrvADC_EnableADCCmp0

原型

```
int32_t DrvADC_EnableADCCmp0 (
   uint8_t u8CmpChannelNum,
   E_ADC_CMP_CONDITION CmpCondition,
   uint16_t u16CmpData,
   uint8_t u8CmpMatchCount
);
```

描述

使能 ADC 比较器 0，并且配置必要的设定。

参数

u8CmpChannelNum [in]

指定想要比较的通道号。取值范围是 0 ~ 7。

CmpCondition [in]

指定比较条件。

LESS_THAN：小于比较数据

GREATER_OR_EQUAL：大于或等于比较数据

u16CmpData [in]

指定比较数据。取值范围是 0~0xFF。

u8CmpMatchCount [in]

指定比较匹配总数。取值范围是 0 ~ 15。当指定的 A/D 通道的模拟转换结果与比较条件匹配，内部匹配计数器加 1。当内部计数器的值增加到(u8CmpMatchCount +1), 比较器 0 中断标志会被置位。

头文件

Driver/DrvADC.h

返回值

E_SUCCESS：成功。比较功能使能

E_DRVADC_ARGUMENT：某一个输入参数溢出

示例

```
u8CmpChannelNum = 0;
u8CmpMatchCount = 5;
/* Enable ADC comparator0. Compare condition: conversion result < 0x800. */
DrvADC_EnableADCCmp0(u8CmpChannelNum, LESS_THAN, 0x800,
u8CmpMatchCount);
```

DrvADC_DisableADCCmp0

原型

void DrvADC_DisableADCCmp0 (void);

描述

禁止 ADC 比较器 0。

参数

无

头文件

Driver/DrvADC.h

返回值

None.

示例

```
/* Disable the ADC comparator 0 */
DrvADC_DisableADCCmp0();
```

DrvADC_EnableADCCmp1

原型

```
int32_t DrvADC_EnableADCCmp1 (
    uint8_t u8CmpChannelNum,
    E_ADC_CMP_CONDITION CmpCondition,
```

uint16_t u16CmpData,

uint8_t u8CmpMatchCount

);

描述

使能 ADC 比较器 1，并且配置必要的设定。

参数

u8CmpChannelNum [in]

指定想要比较的通道号。取值范围是 0 ~ 7。

CmpCondition [in]

指定比较条件。

LESS_THAN：小于比较数据

GREATER_OR_EQUAL：大于或等于比较数据

u16CmpData [in]

指定比较数据。取值范围是 0~0xFF。

u8CmpMatchCount [in]

指定比较匹配总数。取值范围是 0 ~ 15。当指定的 A/D 通道的模拟转换结果与比较条件匹配，内部匹配计数器加 1。当内部计数器的值增加到(u8CmpMatchCount +1), 比较器 1 中断标志会被置位。

头文件

Driver/DrvADC.h

返回值

E_SUCCESS：成功。比较功能使能

E_DRVADC_ARGUMENT：某一个输入参数溢出

示例

```
u8CmpChannelNum = 0;
u8CmpMatchCount = 5;
/* Enable ADC comparator1. Compare condition: conversion result < 0x800. */
DrvADC_EnableADCCmp1(u8CmpChannelNum, LESS_THAN, 0x800,
u8CmpMatchCount);
```

DrvADC_DisableADCCmp1

原型

void DrvADC_DisableADCCmp1 (void);

描述

禁止 ADC 比较器 0。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the ADC comparator 1 */
DrvADC_DisableADCCmp1();
```

DrvADC_EnableSelfCalibration

原型

void DrvADC_EnableSelfCalibration (void);

描述

使能自校正功能来减少 A/D 转换误差。当芯片上电或者软件在 ADC 单端输入和差分输之模式间切换时，用户需要调用这个函数来使能自校正功能。这个函数被调用后，用户在进行 A/D 转换前可以调用 DrvADC_IsCalibrationDone()函数来检查自校正是否完成。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Enable the self calibration function */
DrvADC_EnableSelfCalibration();
```

DrvADC_IsCalibrationDone

原型

uint32_t DrvADC_IsCalibrationDone (void);

描述

检查自校正操作是否已经完成。

参数

无

头文件

Driver/DrvADC.h

返回值

TURE：自校正操作已经完成

FALSE：自校正操作正在进行中

示例

```
if( DrvADC_IsCalibrationDone() )
  printf("Self calibration done.\n");
```

DrvADC_DisableSelfCalibration

原型

void DrvADC_DisableSelfCalibration (void);

描述

禁止自校正功能。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable the self calibration function */
DrvADC_DisableSelfCalibration();
```

DrvADC_GetVersion

原型

uint32_t DrvADC_GetVersion (void);

描述

返回当前 ADC 驱动的版本号。

参数

无

头文件

Driver/DrvADC.h

返回值

版本号：

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
printf("Driver version: %x\n", DrvADC_GetVersion());
```

7. SPI 驱动

7.1. SPI 介绍

串行外设接口(SPI)是一个全双工同步串行数据通讯协议。设备通讯使用主/从模式，4 线，双向接口 M051 系列有 2 组 SPI 控制器，当从外设收到数据的时候实现串到并的转换，当发送数据到外设的时候实现并到串的转换。SLAVE 位(CNTRL[18])被设定之后，MCU 也能作为从设备工作。

当数据传输完成的时候，每个控制器可以产生一个独立的中断信号，向各自的中断标志位写 1 可以清除相应的中断标志。从设备的选中信号激活电平可以是低/高（SSR[SS_LVL]位），具体设定决定于连接的外设。作为主设备的时候，可以写一个除数到 DIVIDER 寄存器来编程 SPI 时钟频率。如果 SPI_CNTRL[23]中的 VARCLK_EN 位被使能，串行时钟可以被设成两个可编程的频率，除数定义在 DIVIDER 和 DIVIDER2 中。可变频率的格式定义在寄存器 VARCLK 中。

每个 SPI 控制器包含两个 32 比特的发送缓冲区(TX0 和 TX1)和两个接收缓冲区(RX0 和 RX1)，支持突发模式，可变长度传输。

本文档介绍 SPI 驱动的用法。

7. 2. 通用特性

- 2 组 SPI 控制器
- 支持主/从模式
- 数据传输长度可配，最大 32 比特
- 主模式时，输出串行时钟频率可变
- 支持突发模式，一次传输最多可以执行两次收/发
- 支持 MSB/LSB 优先数据传输
- 作为主设备的时候，支持 2 个从设备选择线
- 字节重新排序模式
- 与 Motorola SPI 和 National Semiconductor Microwire 总线兼容

7. 3. 常量定义

E_DRVSPI_PORT

枚举标识符	值	描述
eDRVSPI_PORT0	0	SPI 端口 0
eDRVSPI_PORT1	1	SPI 端口 1

E_DRVSPI_MODE

枚举标识符	值	描述
eDRVSPI_MASTER	0	主模式
eDRVSPI_SLAVE	1	从模式

E_DRVSPI_TRANS_TYPE

枚举标识符	值	描述
eDRVSPI_TYPE0	0	SPI 传输类型 0
eDRVSPI_TYPE1	1	SPI 传输类型 1
eDRVSPI_TYPE2	2	SPI 传输类型 2
eDRVSPI_TYPE3	3	SPI 传输类型 3
eDRVSPI_TYPE4	4	SPI 传输类型 4
eDRVSPI_TYPE5	5	SPI 传输类型 5
eDRVSPI_TYPE6	6	SPI 传输类型 6
eDRVSPI_TYPE7	7	SPI 传输类型 7

E_DRVSPI_ENDIAN

枚举标识符	值	描述
eDRVSPI_LSB_FIRST	0	先发送 LSB
eDRVSPI_MSB_FIRST	1	先发送 MSB

E_DRVSPI_BYTE_REORDER

枚举标识符	值	描述
eDRVSPI_BYTE_REORDER_SUSPEND_DISABLE	0	禁止字节排序和字节挂起功能
eDRVSPI_BYTE_REORDER_SUSPEND	1	使能字节排序和字节挂起功能
eDRVSPI_BYTE_REORDER	2	使能字节排序功能
eDRVSPI_BYTE_SUSPEND	3	使能字节挂起功能

E_DRVSPI_SSLTRIG

枚举标识符	值	描述
eDRVSPI_EDGE_TRIGGER	0	边沿触发
eDRVSPI_LEVEL_TRIGGER	1	电平触发

E_DRVSPI_SS_ACT_TYPE

枚举标识符	值	描述
eDRVSPI_ACTIVE_LOW_FALLING	0	低电平/下降沿激活
eDRVSPI_ACTIVE_HIGH_RISING	1	高电平/上升沿激活

7.4. 函数

DrvSPI_Open

原型

```
int32_t DrvSPI_Open(
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_MODE eMode,
    E_DRVSPI_TRANS_TYPE eType,
    int32_t i32BitLength,
);
```

描述

这个函数用来打开 SPI 模块。配置 SPI 工作在主或从模式、SPI 总线时序和每笔传输的长度。自动从设备选择功能被使能。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eMode [in]

工作在主(eDRVSPI_MASTER) 或从(eDRVSPI_SLAVE)模式。

eType [in]

传输类型，也就是总线时序。可以是 eDRVSPI_TYPE0~eDRVSPI_TYPE7。

eDRVSPI_TYPE0：时钟空闲状态为低电平，在串行时钟上升沿传输数据，上升沿锁存数据。

eDRVSPI_TYPE1：时钟空闲状态为低电平，在串行时钟下降沿传输数据，上升沿锁存数据。

eDRVSPI_TYPE2：时钟空闲状态为低电平，在串行时钟上升沿传输数据，下降沿锁存数据。

eDRVSPI_TYPE3：时钟空闲状态为低电平，在串行时钟下降沿传输数据，下降沿锁存数据。

eDRVSPI_TYPE4：时钟空闲状态为高电平，在串行时钟上升沿传输数据，上升沿锁存数据。

eDRVSPI_TYPE5：时钟空闲状态为高电平，在串行时钟下降沿传输数据，上升沿锁存数据。

eDRVSPI_TYPE6：时钟空闲状态为高电平，在串行时钟上升沿传输数据，下降沿

锁存数据。

eDRVSPI_TYPE7：时钟空闲状态为高电平，在串行时钟下降沿传输数据，下降沿锁存数据。

i32BitLength [in]

每笔传输的比特长度。取值范围是 1 ~ 32。

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPI_ERR_INIT：指定的 SPI 端口已经被打开了

E_DRVSPIMS_ERR_BIT_LENGTH：比特长度超过范围

E_DRVSPIMS_ERR_BUSY：指定的 SPI 端口正忙

示例

```
/* Configure SPI0 as a master, 32-bit transaction */
DrvSPI_Open(eDRVSPI_PORT0, eDRVSPI_MASTER, eDRVSPI_TYPE1, 32);
```

DrvSPI_Close

原型

```
void DrvSPI_Close (
    E_DRVSPI_PORT eSpiPort
);
```

描述

关闭指定的 SPI 模块并且禁止 SPI 中断。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Close SPI0 */
DrvSPI_Close(eDRVSPI_PORT0);
```

DrvSPI_SetEndian

原型

```
void DrvSPI_SetEndian (
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_ENDIAN eEndian
);
```

描述

该函数用于配置每笔传输的比特顺序。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eEndian [in]

指定 LSB 优先(eDRVSPI_LSB_FIRST)或 MSB 优先(eDRVSPI_MSB_FIRST)

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* The transfer order of SPI0 is LSB first */
DrvSPI_SetEndian(eDRVSPI_PORT0, eDRVSPI_LSB_FIRST);
```

DrvSPI_SetBitLength

原型

```
int32_t DrvSPI_SetBitLength(
    E_DRVSPI_PORT eSpiPort,
    int32_t i32BitLength
);
```

描述

该函数用于配置 SPI 传输位宽。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

i32BitLength [in]

指定位宽，范围是 1 ~ 32 比特

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPIMS_ERR_BIT_LENGTH：比特长度超过范围

示例

```
/* The transfer bit length of SPI0 is 8-bit */
DrvSPI_SetBitLength(eDRVSPI_PORT0, 8);
```

DrvSPI_SetByteReorder

原型

```
int32_t DrvSPI_SetByteReorder (
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_BYTE_REORDER eOption
);
```

描述

这个函数用于使能/禁止字节重新排序功能。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eOption [in]

字节重排功能和字节挂起功能选项。字节挂起功能只有在 32 比特传输模式下有效。

eDRVSPI_BYTE_REORDER_SUSPEND_DISABLE：禁止字节重排和字节挂起功能

eDRVSPI_BYTE_REORDER_SUSPEND：使能字节重排和字节挂起功能

eDRVSPI_BYTE_REORDER：使能字节重排功能

eDRVSPI_BYTE_SUSPEND：使能字节挂起功能

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPIMS_ERR_BIT_LENGTH：位宽必须是 8/16/24/32 比特

示例

```
/* The transfer bit length of SPI0 is 32-bit */
DrvSPI_SetBitLength(eDRVSPI_PORT0, 32);
/* Enable the Byte Reorder function of SPI0 */
DrvSPI_SetByteReorder(eDRVSPI_PORT0, eDRVSPI_BYTE_REORDER);
```

DrvSPI_SetSuspendCycle

原型

```
int32_t DrvSPI_SetSuspendCycle (
    E_DRVSPI_PORT eSpiPort,
    int32_t i32Interval
);
```

描述

设置挂起间隔的时钟周期数。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

i32Interval [in]

在突发传输模式下，该值指定连续两次传输之间的延迟时钟数。如果字节挂起功能被使能，该值指定每个字节之间的延迟时钟数。可以是 2 ~ 17。

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPI_ERR_SUSPEND_INTERVAL：挂起间隔值超出范围

示例

```
/* The suspend interval is 10 SPI clock cycles */
DrvSPI_SetSuspendCycle (eDRVSPI_PORT0, 10);
```

DrvSPI_SetTriggerMode

原型

```
void DrvSPI_SetTriggerMode (
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_SSLTRIG eSSTriggerMode
);
```

描述

设定从选择引脚触发模式。在主模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eSSTriggerMode [in]

指定触发模式。

eDRVSPI_EDGE_TRIGGER：边沿触发

eDRVSPI_LEVEL_TRIGGER：电平触发

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Level trigger */
```

DrvSPI_SetTriggerMode(eDRVSPI_PORT0, eDRVSPI_LEVEL_TRIGGER);

DrvSPI_SetSlaveSelectActiveLevel

原型

void DrvSPI_SetSlaveSelectActiveLevel (
E_DRVSPI_PORT eSpiPort,
E_DRVSPI_SS_ACT_TYPE eSSActType
);

描述

设定从选择线的激活电平。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eSSActType [in]

选择从选择线激活类型。

eDRVSPI_ACTIVE_LOW_FALLING：

电平触发模式下，从选择引脚低电平激活；边缘触发模式下，从选择引脚下降沿激活。

eDRVSPI_ACTIVE_HIGH_RISING：

电平触发模式下，从选择引脚高电平激活；边缘触发模式下，从选择引脚上升沿激活。

头文件

Driver/DrvSPI.h

返回值

无

示例

/* Configure the active level of SPI0 slave select pin */

DrvSPI_SetSlaveSelectActiveLevel(eDRVSPI_PORT0,
eDRVSPI_ACTIVE_LOW_FALLING);

DrvSPI_GetLevelTriggerStatus

原型

uint8_t DrvSPI_GetLevelTriggerStatus (

E_DRVSPI_PORT eSpiPort

);

描述

这个函数用于获取从设备电平触发传送的状态。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

TRUE：传输笔数和传输的位宽与设定相符

FALSE：某次传输的笔数或者传输的位宽与设定不符

示例

```
/* Level trigger */
DrvSPI_SetTriggerMode(eDRVSPI_PORT0, eDRVSPI_LEVEL_TRIGGER);
…
/* Check the level-trigger transmission status */
If( DrvSPI_GetLevelTriggerStatus(eDRVSPI_PORT0) )
        DrvSPI_DumpRxRegister(eDRVSPI_PORT0,
        &au32DestinationData[u32DataCount], 1); /* Read Rx buffer */
```

DrvSPI_EnableAutoSS

原型

void DrvSPI_EnableAutoSS (

E_DRVSPI_PORT eSpiPort,

);

描述

该函数用于使能自动从选择功能，并且选择从选择引脚。自动从选择意味着当 SPI 传输数据的时候，将自动激活从选择引脚，传输完成的时候将自动设置为非激活状态。对一些设备来说，在多次传输中，从选择引脚需要保持在激活状态，对于这样的设备，用户应该关闭自动从选择功能，改为手动控制。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable the automatic slave select function */
DrvSPI_EnableAutoSS(eDRVSPI_PORT0);
```

DrvSPI_DisableAutoSS

原型

```
void DrvSPI_DisableAutoSS (
  E_DRVSPI_PORT eSpiPort
);
```

Description

该函数用于禁止自动从选择功能。如果用户想要在多次传输过程中保持从选择信号为激活状态，用户可以禁止自动片选功能，手动控制从选择信号。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Disable the automatic slave select function of SPI0 */
DrvSPI_DisableAutoSS(eDRVSPI_PORT0);
```

DrvSPI_SetSS

原型

```
void DrvSPI_SetSS(
    E_DRVSPI_PORT eSpiPort,
);
```

描述

配置从选择引脚。当自动从选择功能被使能，调用该函数来选择从选择引脚。从选择引脚的状态将会被硬件控制。当自动从选择功能被禁止时，从选择引脚将会被设置为激活状态。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Disable the automatic slave select function of SPI0 */
DrvSPI_DisableAutoSS(eDRVSPI_PORT0);
/* Set the slave select pin to active state */
DrvSPI_SetSS(eDRVSPI_PORT0);
```

DrvSPI_ClrSS

原型

```
void DrvSPI_ClrSS(
```

E_DRVSPI_PORT eSpiPort,

);

描述

当自动从选择功能被使能，调用该函数来释放从选择引脚。从选择引脚将会被设置为非激活状态。当自动从选择功能被禁止时，从选择引脚将会被设置为非激活状态。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Disable the automatic slave select function of SPI0 */
DrvSPI_DisableAutoSS(eDRVSPI_PORT0);
/* Set the slave select pin to inactive state */
DrvSPI_ClrSS(eDRVSPI_PORT0);
```

DrvSPI_IsBusy

原型

uint8_t DrvSPI_IsBusy(

E_DRVSPI_PORT eSpiPort

);

描述

检查指定的 SPI 端口是否正忙。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

TURE：SPI 端口正忙

FALSE：SPI 端口空闲

示例

```
/* set the GO_BUSY bit of SPI0 */
DrvSPI_SetGo(eDRVSPI_PORT0);
/* Check the busy status of SPI0 */
while( DrvSPI_IsBusy(eDRVSPI_PORT0) );
```

DrvSPI_BurstTransfer

原型

```
int32_t DrvSPI_BurstTransfer(
    E_DRVSPI_PORT eSpiPort,
    int32_t i32BurstCnt,
    int32_t i32Interval
);
```

描述

配置突发传输模式的相关参数。如果 i32BurstCnt 被设置为 2，则执行突发传输，SPI 控制器会进行两笔连续的数据传输。在两笔连续传输之间的挂起间隔决定于 i32Interval 的值。在从模式下，i32Interval 的设定值无用。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

i32BurstCnt [in]

指定一次突发传输中的传输笔数。可以是 1 或者 2。

i32Interval [in]

挂起间隔长度。指定两次连续的传输之间的 SPI 时钟周期数。可以是 2 ~ 17。

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPIMS_ERR_BURST_CNT：突发传输笔数超过范围.

E_DRVSPIMS_ERR_TRANSMIT_INTERVAL：间隔超过范围.

示例

```
/* Configure the SPI0 burst transfer mode; two transactions in one transfer; 10 delay clocks between the transactions. */
DrvSPI_BurstTransfer(eDRVSPI_PORT0, 2, 10);
```

DrvSPI_SetClockFreq

原型

```
uint32_t
DrvSPI_SetClockFreq(
    E_DRVSPI_PORT eSpiPort,
    uint32_t u32Clock1,
    uint32_t u32Clock2
);
```

描述

配置 SPI 时钟频率。在主模式下，串行时钟输出频率是可编程的。如果可变时钟功能使能，串行时钟的输出模式在 **VARCLK** 中定义。如果 **VARCLK** 位是"0"，则 SPICLK 的输出频率等于可变时钟 1 的频率；反之，SPICLK 的输出频率等于可变时钟 2 的频率。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

u32Clock1 [in]

指定 SPI 时钟频率，单位 Hz。它是 SPI 时钟和可变时钟 1 的频率。

u32Clock2 [in]

指定 SPI 时钟频率，单位 Hz。它是可变时钟 2 的频率。

头文件

Driver/DrvSPI.h

Driver/DrvSYS.h

返回值

SPI 时钟的实际频率。由于硬件的限制，实际的时钟频率可能与目标有差异

示例

```
/* SPI0 clock rate of clock 1 is 2MHz; the clock rate of clock 2 is 1MHz */
DrvSPI_SetClockFreq(eDRVSPI_PORT0, 2000000, 1000000);
```

DrvSPI_GetClock1Freq

原型

```
uint32_t
DrvSPI_GetClock1Freq(
    E_DRVSPI_PORT eSpiPort
);
```

描述

获取 SPI 时钟频率，单位 Hz。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

Driver/DrvSYS.h

返回值

SPI 总线时钟频率，单位 Hz。

示例

```
/* Get the engine clock rate of SPI0 */
printf("SPI clock rate: %d Hz\n", DrvSPI_GetClock1Freq(eDRVSPI_PORT0));
```

DrvSPI_GetClock2Freq

原型

```
uint32_t
```

DrvSPI_GetClock2Freq(

E_DRVSPI_PORT eSpiPort

);

描述

获取 SPI 可变时钟 2 的频率，单位 Hz。在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

Driver/DrvSYS.h

返回值

SPI 可变时钟 2 的频率，单位 Hz

示例

```
/* Get the clock rate of SPI0 variable clock 2 */
printf("SPI clock rate of variable clock 2: %d Hz\n",
DrvSPI_GetClock2Freq(eDRVSPI_PORT0));
```

DrvSPI_SetVariableClockFunction

原型

void

DrvSPI_SetVariableClockFunction (

E_DRVSPI_PORT eSpiPort,

uint8_t bEnable,

uint32_t u32Pattern

);

描述

设置可变时钟功能。串行时钟输出模式在 **VARCLK** 寄存器中定义。**VARCLK** 寄存器的两位联合定义一个串行时钟模式。位域 **VARCLK** [31:30]定义 SPICLK 的第一个时钟周期，位域 **VARCLK** [29:28]定义 SPICLK 的第二个时钟周期，等等。下图是串行时钟(SPICLK)，**VARCLK** 寄存器和可变时钟源三者之间时序关系图。

如果时钟模式 **VARCLK** 是 ‘0’, SPICLK 的输出频率等于可变时钟 1 的频率。

如果时钟模式 **VARCLK** 是 ‘1’, SPICLK 的输出频率等于可变时钟 2 的频率。

注意当可变时钟功能使能时，传输位宽的设定值只能编程为 0x10(16 比特模式)。

在从模式下，执行该函数是无用的。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

bEnable [in]

使能(TRUE) / 禁止(FALSE)

u32Pattern [in]

指定可变时钟模式。如果 **bEnable** 设置为 0，该设定值无用。

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable the SPI0 variable clock function and set the variable clock pattern */
DrvSPI_SetVariableClockFunction(eDRVSPI_PORT0, TRUE, 0x007FFF87);
```

DrvSPI_EnableInt

原型

```
void DrvSPI_EnableInt(
    E_DRVSPI_PORT eSpiPort,
    PFN_DRVSPI_CALLBACK pfnCallback,
    uint32_t u32UserData
```

);

描述

使能指定 SPI 端口的 SPI 中断，并安装中断回调函数。

参数

u16Port [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pfnCallback [in]

相应 SPI 中断回调函数指针。

u32UserData [in]

传给回调函数的参数。

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable the SPI0 interrupt and install the callback function. The parameter 0 will be passed to the callback function. */
DrvSPI_EnableInt(eDRVSPI_PORT0, SPI0_Callback, 0);
```

DrvSPI_DisableInt

原型

```
void DrvSPI_DisableInt(
    E_DRVSPI_PORT eSpiPort
);
```

描述

禁止指定的 SPI 端口的 SPI 中断。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Disable the SPI0 interrupt */
DrvSPI_DisableInt(eDRVSPI_PORT0);
```

DrvSPI_GetIntFlag

原型

```
uint32_t DrvSPI_GetIntFlag (
    E_DRVSPI_PORT eSpiPort
);
```

描述

获取 SPI 中断标志。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

0：SPI 中断没有发生

1：发生了 SPI 中断

示例

```
/* Get the SPI0 interrupt flag */
DrvSPI_GetIntFlag(eDRVSPI_PORT0);
```

DrvSPI_ClrIntFlag

原型

void DrvSPI_ClrIntFlag (
E_DRVSPI_PORT eSpiPort
);

描述

清除 SPI 中断标志。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Clear the SPI0 interrupt flag */
DrvSPI_ClrIntFlag(eDRVSPI_PORT0);
```

DrvSPI_SingleRead

原型

uint8_t DrvSPI_SingleRead(
E_DRVSPI_PORT eSpiPort,
uint32_t *pu32Data
);

描述

从 SPI 接收寄存器读数据，并触发下一次 SPI 传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Data [out]

缓存指针。该缓存用来存储从 SPI 总线获取的数据

头文件

Driver/DrvSPI.h

返回值

TRUE：存在 pu32Data 中的数据有效

FALSE：存在 pu32Data 中的数据无效

示例

```
/* Read the previous retrieved data and trigger next transfer. */
uint32_t u32DestinationData;
DrvSPI_SingleRead(eDRVSPI_PORT0, &u32DestinationData);
```

DrvSPI_SingleWrite

原型

```
uint8_t DrvSPI_SingleWrite (
    E_DRVSPI_PORT eSpiPort,
    uint32_t *pu32Data
);
```

描述

写数据到 SPI TX0 寄存器，并触发 SPI 开始传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Data [in]

缓存指针。存储在该缓存的数据会通过 SPI 总线发出

头文件

Driver/DrvSPI.h

返回值

TRUE：存在 pu32Data 中的数据已被发送

FALSE：SPI 正忙。存在 pu32Data 中的数据未被发送

示例

/* Write the data stored in u32SourceData to TX buffer of SPI0 and trigger SPI to start transfer. */

uint32_t u32SourceData;

DrvSPI_SingleWrite(eDRVSPI_PORT0, &u32SourceData);

DrvSPI_BurstRead

原型

```
uint8_t DrvSPI_BurstRead (
    E_DRVSPI_PORT eSpiPort,
    uint32_t *pu32Buf
);
```

描述

从 SPI 接收寄存器读取两个字的数据，并触发下一次 SPI 传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Buf [out]

缓存指针，该缓存用来存储从 SPI 总线获取的数据

头文件

Driver/DrvSPI.h

返回值

TRUE：存在 pu32Buf 中的数据有效

FALSE：存在 pu32Buf 中的数据无效

示例

/* Read two words of data from SPI0 RX registers to au32DestinationData[u32DataCount] and au32DestinationData[u32DataCount+1]. And then trigger SPI for next transfer. */

DrvSPI_BurstRead(eDRVSPI_PORT0, &au32DestinationData[u32DataCount]);

DrvSPI_BurstWrite

原型

```
uint8_t DrvSPI_BurstWrite (
    E_DRVSPI_PORT eSpiPort,
    uint32_t *pu32Buf
);
```

描述

写两个字的数据到 SPI TX 寄存器，并触发 SPI 开始传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Buf [in]

缓存指针。存储在该缓存的数据会通过 SPI 总线发出

头文件

Driver/DrvSPI.h

返回值

TRUE：存在 pu32Buf 中的数据已经被发送

FALSE：SPI 正忙。存在 pu32Buf 中的数据未被发送

示例

```
/* Write two words of data stored in au32SourceData[u32DataCount] and
au32SourceData[u32DataCount+1] to SPI0 TX registers. And then trigger SPI for next
transfer. */

DrvSPI_BurstWrite(eDRVSPI_PORT0, &au32SourceData[u32DataCount]);
```

DrvSPI_DumpRxRegister

原型

```
uint32_t
DrvSPI_DumpRxRegister (
    E_DRVSPI_PORT eSpiPort,
    uint32_t *pu32Buf,
    uint32_t u32DataCount
);
```

描述

从接收寄存器读数据。该函数不会触发一次 SPI 数据传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Buf [out]

缓存指针，该缓存用来存放从接收寄存器获取的数据

u32DataCount [in]

从接收寄存器读取数据的个数，最大值是 2

头文件

Driver/DrvSPI.h

返回值

从接收寄存器实际读取数据的个数。

示例

```
/* Read one word of data from SPI0 RX buffer and store to
au32DestinationData[u32DataCount] */

DrvSPI_DumpRxRegister(eDRVSPI_PORT0, &au32DestinationData[u32DataCount], 1);
```

DrvSPI_SetTxRegister

原型

```
uint32_t
DrvSPI_SetTxRegister (
    E_DRVSPI_PORT eSpiPort,
    uint32_t *pu32Buf,
    uint32_t u32DataCount
);
```

描述

写数据到 Tx 寄存器。该函数不会触发 SPI 数据传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

pu32Buf [in]

存储将要写到 TX 寄存器的数据的缓存。

u32DataCount [in]

写到 TX 寄存器的数据个数

头文件

Driver/DrvSPI.h

返回值

实际写到 TX 寄存器的数据个数

示例

```
/* Write one word of data stored in u32Buffer to SPI0 TX register. */
DrvSPI_SetTxRegister(eDRVSPI_PORT0, &u32Buffer, 1);
```

DrvSPI_SetGo

原型

```
void DrvSPI_SetGo (
    E_DRVSPI_PORT eSpiPort
);
```

描述

主模式下，调用该函数可以开始依次 SPI 数据传输。从模式下，执行该函数表示从设备已经准备好和主机进行通信。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Trigger a SPI data transfer */
DrvSPI_SetGo(eDRVSPI_PORT0);
```

DrvSPI_ClrGo

原型

```
void DrvSPI_ClrGo (
    E_DRVSPI_PORT eSpiPort
);
```

描述

停止 SPI 数据传输。

参数

eSpiPort [in]

指定 SPI 端口

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Stop a SPI data transfer */
DrvSPI_ClrGo(eDRVSPI_PORT0);
```

DrvSPI_GetVersion

原型

```
uint32_t
DrvSPI_GetVersion (void);
```

描述

获取 M051 系列 SPI 驱动版本号。

头文件

Driver/DrvSPI.h

返回值

版本号

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
printf("Driver version: %x\n", DrvSPI_GetVersion());
```

8. I2C 驱动

8.1. I2C 介绍

I2C 是两线双向串行总线，提供了两设备间简单有效的数据交换方式。I2C 标准是一个真正的多主总线，包含冲突检测和总线仲裁，可以防止当两个或者多个主机同时试图获取总线控制权时引发数据混乱。串行，8 比特双向数据传输速度可达 1.0Mbps。

对于 NuMicro™ M051 系列，I2C 设备可以作为主机或者从机，I2C 驱动可以帮助用户很容易的使用 I2C 功能。

8.2. I2C 特性

I2C 包含以下特性：

- 支持主和从模式，最高速度可以达到 1Mbps。
- 内嵌一个 14 比特的超时计数器，如果 I2C 总线被挂起并且超时发生，I2C 将发出中断。
- 支持 7 比特寻址模式。
- 支持多地址识别功能 (四个从属地址，支持掩码)。

8.3. 类型定义

E_I2C_CALLBACK_TYPE

枚举标识符	值	描述
I2CFUNC	0	I2C 正常状态
ARBITLOSS	1	I2C 作为主机时的仲裁丢失状态
BUSERROR	2	I2C 总线错误状态
TIMEOUT	3	I2C 14 比特超时计数器溢出

8.4. 函数

DrvI2C_Open

原型

int32_t DrvI2C_Open (uint32_t u32BusClock);

描述

打开 I2C 硬件，并配置 I2C 总线时钟。I2C 总线时钟最大为 1MHz。

参数

u32BusClock [in]

配置 I2C 总线时钟。单位 Hz

头文件

Driver/DrvI2C.h

返回值

0 成功

示例

```
/* Enable I2C and set I2C bus clock 100KHz */
DrvI2C_Open (100000);
```

DrvI2C_Close

原型

int32_t DrvI2C_Close (void);

描述

关闭 I2C 硬件。

参数

无

头文件

Driver/DrvI2C.h

返回值

0 成功

示例

```
DrvI2C_Close ( );              /* Disable I2C */
```

DrvI2C_SetClockFreq

原型

int32_t DrvI2C_SetClockFreq (uint32_t u32BusClock);

描述

配置 I2C 总线时钟。I2C 总线时钟 = I2C 时钟源频率 / (4 x (I2CCLK_DIV+1))。

I2C 总线时钟最大为 1MHz。

参数

u32BusClock [in]

配置 I2C 总线时钟。单位 Hz

头文件

Driver/DrvI2C.h

返回值

0 成功

示例

```
/* Set I2C bus clock 200 KHz */
DrvI2C_SetClockFreq (200000);
```

DrvI2C_GetClockFreq

原型

uint32_t DrvI2C_GetClockFreq (void);

描述

获取 I2C 总线时钟频率。I2C 总线时钟 = I2C 时钟源频率 / (4 x (I2CCLK_DIV+1))。

参数

无

头文件

Driver/DrvI2C.h

返回值

I2C 总线时钟频率

示例

```
uint32_t  u32clock;
u32clock = DrvI2C_GetClockFreq ( );   /* Get I2C bus clock */
```

DrvI2C_SetAddress

原型

int32_t DrvI2C_SetAddress (uint8_t slaveNo, uint8_t slave_addr, uint8_t GC_Flag);

描述

向指定的 I2C 从地址设定 7 比特 I2C 物理从地址，总共可以设定 4 个从地址。设定只在 I2C 工作在从机模式时有效。

参数

slaveNo [in]

选择从地址，可以是 0 ~ 3

slave_addr [in]

为选择的从地址设定 7 比特物理从地址

GC_Flag [in]

使能/关闭全呼叫功能(general call)。(1：使能，0：关闭)

头文件

Driver/DrvI2C.h

返回值

0	成功
<0	失败

示例

```
DrvI2C_SetAddress(0, 0x15, 0);  /* Set I2C 1st slave address 0x15 */ DrvI2C_SetAddress(1, 0x35, 0);  /* Set I2C 2nd slave address 0x35 */ DrvI2C_SetAddress(2, 0x55, 0);  /* Set I2C 3rd slave address 0x55 */ DrvI2C_SetAddress(3, 0x75, 0);  /* Set I2C 4th slave address 0x75 */
```

DrvI2C_SetAddressMask

原型

int32_t DrvI2C_SetAddressMask (uint8_t slaveNo, uint8_t slaveAddrMask);

描述

向指定的 I2C 从地址掩码设定 7 比特 I2C 物理从地址掩码，总共可以设定 4 个从地址掩码。设定只在 I2C 工作在从机模式时有效。

参数

slaveNo [in]

选择从地址掩码。取值是 0 ~ 3

slaveAddrMask [in]

为选择的从地址掩码设定 7 比特物理从地址掩码，相应的地址位被忽略

头文件

Driver/DrvI2C.h

返回值

0 成功

<0 失败

示例

```
DrvI2C_SetAddress (0, 0x15, 0);   /* Set I2C 1st slave address 0x15 */
DrvI2C_SetAddress (1, 0x35, 0);   /* Set I2C 2nd slave address 0x35 */
/* Set I2C 1st slave address mask 0x01, slave address 0x15 and 0x14 would be addressed */
DrvI2C_SetAddressMask (0, 0x01);
/* Set I2C 2nd slave address mask 0x04, slave address 0x35 and 0x31 would be addressed */
DrvI2C_SetAddressMask (1, 0x04);
```

DrvI2C_GetStatus

原型

uint32_t DrvI2C_GetStatus (void);

描述

获取 I2C 状态码。共有 26 个状态码。详细请参考 TRM I2C 章节数据传输流程图。

参数

无

头文件

Driver/DrvI2C.h

返回值

I2C 状态码

示例

```
uint32_t  u32status;
u32status = DrvI2C_GetStatus ( );   /* Get I2C current status code */
```

DrvI2C_WriteData

原型

void DrvI2C_WriteData (uint8_t u8data);

描述

设定将要发送的 1 字节数据。

参数

u8data [in]

字节数据

头文件

Driver/DrvI2C.h

返回值

无

示例

```
DrvI2C_WriteData (0x55); /* Set byte data 0x55 into I2C data register */
```

DrvI2C_ReadData

原型

uint8_t DrvI2C_ReadData (void);

描述

从 I2C 总线读上一个数据。

参数

无

头文件

Driver/DrvI2C.h

返回值

上一个字节数据

示例

uint8_t u8data;

u8data = DrvI2C_ReadData ( ); /* Read out byte data from I2C data register */

DrvI2C_Ctrl

原型

void DrvI2C_Ctrl (uint8_t start, uint8_t stop, uint8_t intFlag, uint8_t ack);

描述

设定 I2C 控制位，包括控制寄存器中的 STA, STO, AA, SI。

参数

start [in]

是否置位 STA 位。(1：置位，0：不置位)。如果 STA 置位，在 I2C 总线空闲时，会产生一个开始或者重复开始信号。

stop [in]

是否置位 STO 位。(1 置位，0 不置位)。如果 STO 置位，将会产生一个停止信号。当停止条件被检测到，硬件自动清除该位。

intFlag [in]

清除 SI 标志(I2C 中断标志)。(1：清除，0：不起作用)

ack [in]

使能 AA 位(声明应答控制位)。(1：使能，0：禁止)

头文件

Driver/DrvI2C.h

返回值

无

示例

DrvI2C_Ctrl (0, 0, 1, 0); /* Set I2C SI bit to clear SI flag */

DrvI2C_Ctrl (1, 0, 0, 0); /* Set I2C STA bit to send START signal */

DrvI2C_GetIntFlag

原型

uint8_t DrvI2C_GetIntFlag (void);

描述

获取 I2C 中断标志状态。

参数

无

头文件

Driver/DrvI2C.h

返回值

中断状态 (1 或 0)

示例

```
uint8_t  u8flagStatus;
u8flagStatus = DrvI2C_GetIntFlag ( ); /* Get the status of I2C interrupt flag */
```

DrvI2C_ClearIntFlag

原型

```
void   DrvI2C_ClearIntFlag (void);
```

描述

如果 I2C 中断标志被置为 1，清除该标志。

参数

无

头文件

Driver/DrvI2C.h

返回值

无

示例

```
DrvI2C_ClearIntFlag ( ); /* Clear I2C interrupt flag (SI) */
```

DrvI2C_EnableInt

原型

```
int32_t  DrvI2C_EnableInt (void);
```

描述

使能 I2C 中断功能。

参数

无

头文件

Driver/DrvI2C.h

返回值

0　　成功

示例

DrvI2C_EnableInt ( ); /* Enable I2C interrupt */

DrvI2C_DisableInt

原型

int32_t DrvI2C_DisableInt (void);

描述

禁止 I2C 中断功能

参数

无

头文件

Driver/DrvI2C.h

返回值

0　　成功

示例

DrvI2C_DisableInt ( ); /* Disable I2C interrupt */

DrvI2C_InstallCallBack

原型

int32_t DrvI2C_InstallCallBack (E_I2C_CALLBACK_TYPE Type, I2C_CALLBACK callbackfn);

描述

安装 I2C 中断处理函数的回调函数。

参数

Type [in]

回调函数有四种类型。(I2CFUNC / ARBITLOSS / BUSERROR / TIMEOUT)

I2CFUNC：正常的 I2C 状态

ARBITLOSS：主模式下仲裁丢失。状态码 0x38

BUSERROR：总线错误。状态码 0x00

TIMEOUT：14 比特超时计数器溢出

callbackfn [in]

指定中断事件的回调函数名

头文件

Driver/DrvI2C.h

返回值

0 成功

<0 失败

示例

```
/* Install I2C call back function 'I2C_Callback_Normal' for I2C normal condition */
DrvI2C_InstallCallback (I2CFUNC, I2C_Callback_Normal);
/* Install I2C call back function 'I2C_Callback_BusErr' for Bus Error condition */
DrvI2C_InstallCallback (BUSERROR, I2C_Callback_BusErr);
```

DrvI2C_UninstallCallBack

原型

int32_t DrvI2C_UninstallCallBack (E_I2C_CALLBACK_TYPE Type);

描述

卸载 I2C 中断处理函数的回调函数。

参数

Type [in]

回调函数有四种类型。(I2CFUNC / ARBITLOSS / BUSERROR / TIMEOUT)

I2CFUNC：正常的 I2C 状态

ARBITLOSS：主模式下仲裁丢失。状态码 0x38

BUSERROR：总线错误。状态码 0x00

TIMEOUT：14 比特超时计数器溢出

头文件

Driver/DrvI2C.h

返回值

0　成功

<0　失败

示例

```
/* Uninstall I2C call back function for I2C normal condition */
DrvI2C_UninstallCallBack (I2CFUNC);
/* Uninstall I2C call back function for Bus Error condition */
DrvI2C_UninstallCallBack (BUSERROR);
```

DrvI2C_SetTimeoutCounter

原型

int32_t　DrvI2C_SetTimeoutCounter (int32_t i32enable, uint8_t u8div4);

描述

配置 14 比特超时计数器。

参数

i32enable [in]

使能或禁止 14 比特超时计数器。 (1：使能，0：禁止)

u8div4 [in]

1：使能 DIV4 功能。当超时计数器被使能，超时计数器时钟源等于 HCLK/4。

0：禁止 DIV4 功能。当超时计数器被使能，超时计数器时钟源来自于 HCLK。

头文件

Driver/DrvI2C.h

返回值

0　成功

示例

```
/* Enable I2C 14-bit timeout counter and disable its DIV4 function */
DrvI2C_EnableTimeoutCount (1, 0);
```

DrvI2C_ClearTimeoutFlag

原型

void　DrvI2C_ClearTimeoutFlag (void);

描述

如果 I2C 超时中断标志 TIF 被置为 1，清除该标志。

参数

无

头文件

Driver/DrvI2C.h

返回值

无

示例

DrvI2C_ClearTimeoutFlag ( ); /* Clear I2C TIF flag */

DrvI2C_GetVersion

原型

uint32_t DrvI2C_GetVersion (void);

描述

获取该模块版本。

参数

无

头文件

Driver/DrvI2C.h

返回值

版本号

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

9. PWM 驱动

9.1. PWM 介绍

在一组 PWM 中，基本组成部分是预分频器，时钟分频器，16 比特计数器，16 比特比较器，反转器，死区发生器。它们都由 PWM 时钟源驱动。有三种时钟源：12 MHz crystal 时钟，HCLK 和内部 22MHZ 时钟。时钟分频器提供给通道 5 种时钟源(1, 1/2, 1/4, 1/8, 1/16)。每个 PWM 定时器从时钟分频器接收自己的时钟信号，每个时钟分频器的时钟源来自 8 比特预分频器。每个通道的 16 比特计数器接收来自时钟选择器的时钟信号作为一个时钟周期。16 比特比较器比较计数器中的值和事先装载到寄存器中的极限值来产生 PWM 的占空比。

为了避免 PWM 在不稳定的状态下驱动输出引脚，16 比特的计数器和 16 比特的比较器使用双缓存特性。用户可以随意写数据到计数器缓存寄存器和比较器缓存寄存器，不用考虑短时脉冲波型干扰。

当 16 比特递减计数器减到 0 时，中断产生通知 CPU 时间到。当计数器减到 0 时，如果计数器被设成自动重加载(auto-reload)模式，它的值会被重新自动加载并且自动开始下一个周期。用户也可以将计数器设成单次(one-shot)模式，这样减到 0 的时候，计数器将停止计数并且产生一个中断。

9.2. PWM 特性

PWM 控制器包含如下特性：

- 两个 PWM 组(PWMA/PWMB)。PWM 组编号请参考附录 NuMicro™ M051 系列产品选型指导。
- 每个 PWM 组包含两个 PWM 发生器。每个发生器支持 8 比特预分频器，一个时钟分频器，两个 PWM 计数器(向下计数)，一个死区发生器和两路 PWM 输出。
- One-shot 或 Auto-reload PWM 模式。
- 八个捕获输入通道。
- 每个捕获输入通道支持上升沿/下降沿锁存寄存器和捕获中断标志。

9. 3. 常量定义

Name	Value	Description
DRVPWM_TIMER0	0x00	PWM 定时器 0
DRVPWM_TIMER1	0x01	PWM 定时器 1
DRVPWM_TIMER2	0x02	PWM 定时器 2
DRVPWM_TIMER3	0x03	PWM 定时器 3
DRVPWM_TIMER4	0x04	PWM 定时器 4
DRVPWM_TIMER5	0x05	PWM 定时器 5
DRVPWM_TIMER6	0x06	PWM 定时器 6
DRVPWM_TIMER7	0x07	PWM 定时器 7
DRVPWM_CAP0	0x10	PWM 捕获器 0
DRVPWM_CAP1	0x11	PWM 捕获器 1
DRVPWM_CAP2	0x12	PWM 捕获器 2
DRVPWM_CAP3	0x13	PWM 捕获器 3
DRVPWM_CAP4	0x14	PWM 捕获器 4
DRVPWM_CAP5	0x15	PWM 捕获器 5
DRVPWM_CAP6	0x16	PWM 捕获器 6
DRVPWM_CAP7	0x17	PWM 捕获器 7
DRVPWM_CAP_ALL_INT	3	PWM 捕获器上升沿和下降沿中断
DRVPWM_CAP_RISING_INT	1	PWM 捕获器上升沿中断
DRVPWM_CAP_FALLING_INT	2	PWM 捕获器下降沿中断
DRVPWM_CAP_RISING_FLAG	6	PWM 捕获器上升沿中断标志
DRVPWM_CAP_FALLING_FLAG	7	PWM 捕获器下降沿中断标志
DRVPWM_CLOCK_DIV_1	4	输入时钟除 1
DRVPWM_CLOCK_DIV_2	0	输入时钟除 2
DRVPWM_CLOCK_DIV_4	1	输入时钟除 4
DRVPWM_CLOCK_DIV_8	2	输入时钟除 8
DRVPWM_CLOCK_DIV_16	3	输入时钟除 16
DRVPWM_AUTO_RELOAD_MODE	1	PWM 定时器 auto-reload 模式
DRVPWM_ONE_SHOT_MODE	0	PWM 定时器 one-shot 模式

9. 4. 函数

DrvPWM_IsTimerEnabled

原型

int32_t DrvPWM_IsTimerEnabled (uint8_t u8Timer);

描述

该函数用于获取 PWM 指定定时器使能/禁止状态。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

头文件

Driver/DrvPWM.h

返回值

1：指定的定时器是使能的

0：指定的定时器没有使能

示例

```
int32_t  i32state ;
/* Check if PWM timer 3 is enabled or not */
if(DrvPWM_IsTimerEnabled (DRVPWM_TIMER3)==1)
   printf("PWM timer 3 is enabled!\n");
else if(DrvPWM_IsTimerEnabled (DRVPWM_TIMER3)==0)
  printf("PWM timer 3 is disabled!\n");
```

DrvPWM_SetTimerCounter

原型

void DrvPWM_SetTimerCounter (uint8_t u8Timer, uint16_t u16Counter);

描述

这个函数用于设定 PWM 指定定时器的计数值。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

u16Counter [in]

指定定时器的计数值(0~65535)。

头文件

Driver/DrvPWM.h

返回值

无

Note

如果计数值被设置为 0，定时器将停止。

示例

```
/* Set 10000 to PWM timer 3 counter register. When the PWM timer 3 start to count down, PWM timer 3 will count down from 10000 to 0. If PWM timer 3 is set to auto-reload mode, the PWM timer 3 will reload 10000 to PWM timer 3 counter register after PWM timer 3 count down to 0 and PWM timer 3 will continue to count down from 10000 to 0 again. */
DrvPWM_SetTimerCounter (DRVPWM_TIMER3, 10000);
```

DrvPWM_GetTimerCounter

原型

uint32_t DrvPWM_GetTimerCounter (uint8_t u8Timer);

描述

这个函数用于获取 PWM 指定定时器的计数值。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

头文件

Driver/DrvPWM.h

返回值

指定定时器的计数值

示例

```
/* Get PWM timer 5 counter value. */
uint32_t u32RetValTimer5CounterValue;
u32RetValTimer5CounterValue = DrvPWM_GetTimerCounter (DRVPWM_TIMER5);
```

DrvPWM_EnableInt

原型

void DrvPWM_EnableInt(uint8_t u8Timer, uint8_t u8Int, PFN_DRVPWM_CALLBACK pfncallback);

描述

这个函数用于使能 PWM 定时器/捕获器的中断并且安装中断回调函数。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

u8Int [in]

指定捕获器中断类型(只在 PWM 运行在捕获功能时这个参数才有效)

DRVPWM_CAP_RISING_INT： 捕获上升沿时中断

DRVPWM_CAP_FALLING_INT： 捕获下降沿时中断

DRVPWM_CAP_ALL_INT： 上升/下降沿都发生中断

pfncallback [in]

指定定时器/捕获器的中断回调函数指针

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Enable PWM capture 5 falling edge interrupt and install DRVPWM_CapIRQHandler() as it's interrupt callback function.*/
DrvPWM_EnableInt (DRVPWM_CAP5, DRVPWM_CAP_FALLING_INT,
                  DRVPWM_CapIRQHandler);
```

DrvPWM_DisableInt

原型

void DrvPWM_DisableInt (uint8_t u8Timer);

描述

这个函数用于禁止 PWM 定时器/捕获器中断

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Disable PWM capture 5 interrupts including rising and falling interrupt source and also
uninstall PWM capture 5 rising and falling interrupt callback functions. */

DrvPWM_DisableInt (DRVPWM_CAP5);

/* Disable PWM timer 5 interrupt and uninstall PWM timer 5 callback function.*/

DrvPWM_DisableInt (DRVPWM_TIMER5);
```

DrvPWM_ClearInt

原型

void DrvPWM_ClearInt (uint8_t u8Timer);

描述

这个函数用于清除 PWM 定时器/捕获器中断标志。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Clear PWM timer 1 interrupt flag.*/
DrvPWM_ClearInt (DRVPWM_TIMER1);
/* Clear PWM capture 0 interrupt flag. */
DrvPWM_ClearInt (DRVPWM_CAP0);
```

DrvPWM_GetIntFlag

原型

int32_t DrvPWM_GetIntFlag (uint8_t u8Timer);

描述

这个函数用于获取 PWM 定时器/捕获器中断标志。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

头文件

Driver/DrvPWM.h

返回值

1：指定的中断已经发生

0：指定的中断没有发生

示例

```
/* Get PWM timer 6 interrupt flag.*/
if(DrvPWM_GetIntFlag(DRVPWM_TIMER6)==1)
   printf(”PWM timer 6 interrupt occurs!\n);
else if(DrvPWM_GetIntFlag(DRVPWM_TIMER6)==0)
   printf(”PWM timer 6 interrupt dosen’t occur!\n);
```

DrvPWM_GetRisingCounter

原型

uint16_t DrvPWM_GetRisingCounter(uint8_t u8Capture);

描述

这个函数用于获取当有上升转变时，锁存的计数值。

参数

u8Capture [in]

指定捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

头文件

Driver/DrvPWM.h

返回值

当有上升动作时，从 PWM 捕获器当前计数器锁存的计数值。

示例

```
/* Get PWM capture 7 rising latch register value. */
uint16_t u16RetValTimer7RisingLatchValue;
u16RetValTimer7RisingLatchValue = DrvPWM_GetRisingCounter (DRVPWM_CAP7);
```

DrvPWM_GetFallingCounter

原型

uint16_t DrvPWM_GetFallingCounter (uint8_t u8Capture);

描述

这个函数用于获取当有下降转变时，锁存的计数值。

参数

u8Capture [in]

指定捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

头文件

Driver/DrvPWM.h

返回值

当有下降动作时，从 PWM 捕获器当前计数器锁存的计数值。

示例

```
/* Get PWM capture 7 falling latch register value.*/
uint16_t u16RetValTimer7FallingLatchValue;
u16RetValTimer7FallingLatchValue = DrvPWM_GetFallingCounter (DRVPWM_CAP7);
```

DrvPWM_GetCaptureIntStatus

原型

int32_t DrvPWM_GetCaptureIntStatus (uint8_t u8Capture, uint8_t u8IntType);

描述

检查是否有发生上升/下降变换。

参数

u8Capture [in]

指定捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

u8IntType [in]

指定捕获器锁存指示符。

DRVPWM_CAP_RISING_FLAG：捕获器上升沿指示符标志

DRVPWM_CAP_FALLING_FLAG：捕获器下降沿指示符标志

头文件

Driver/DrvPWM.h

返回值

TRUE：指定的变换发生

FALSE：指定的变换没有发生

示例

```
/* Get PWM capture 5 rising transition flag.*/
if(DrvPWM_GetCaptureIntStatus(DRVPWM_CAP5,
DRVPWM_CAP_RISING_FLAG)==TRUE)
   printf("PWM capture 5 rising transition occurs!\n")
else if(DrvPWM_GetCaptureIntStatus(DRVPWM_CAP5,
DRVPWM_CAP_RISING_FLAG)==FALSE)
   printf("PWM capture 5 rising transition doesn't occur!\n")
```

DrvPWM_ClearCaptureIntStatus

原型

void DrvPWM_ClearCaptureIntStatus (uint8_t u8Capture, uint8_t u8IntType);

描述

清除上升/下降变换指示符标志。

参数

u8Capture [in]

指定捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

u8IntType [in]

指定捕获器锁存指示符。

DRVPWM_CAP_RISING_FLAG：捕获器上升沿指示符标志

DRVPWM_CAP_FALLING_FLAG：捕获器下降沿指示符标志

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Clear PWM capture 5 falling transition flag.*/
DrvPWM_ClearCaptureIntStatus (DRVPWM_CAP5, DRVPWM_CAP_FALLING_FLAG);
```

DrvPWM_Open

原型

void DrvPWM_Open (void);

描述

使能 PWM 时钟并且复位 PWM。

头文件

Driver/DrvPWM.h

返回值

无

示例

/* Enable PWM engine clock and reset PWM engine. */

DrvPWM_Open ( );

DrvPWM_Close

原型

void DrvPWM_Close (void);

描述

禁止 PWM 时钟和捕获输入/PWM 输出使能功能。

头文件

Driver/DrvPWM.h

返回值

无

示例

/* Disable PWM timer0~7 output, PWM capture 0~7 output and disable PWM engine clock.*/

DrvPWM_Close ( );

DrvPWM_EnableDeadZone

原型

void DrvPWM_EnableDeadZone (uint8_t u8Timer, uint8_t u8Length, int32_t i32EnableDeadZone);

描述

这个函数可以用来配置死区长度并且使能/禁止死区功能。

参数

u8Timer [in]

Specify the timer

DRVPWM_TIMER0 or DRVPWM_TIMER1: PWM timer 0 & PWM timer 1.

DRVPWM_TIMER2 or DRVPWM_TIMER3: PWM timer 2 & PWM timer 3.

DRVPWM_TIMER4 or DRVPWM_TIMER5: PWM timer 4 & PWM timer 5.

DRVPWM_TIMER6 or DRVPWM_TIMER7: PWM timer 6 & PWM timer 7.

u8Length [in]

指定死区长度：0 ~ 255。单位是一个 PWM 时钟周期

i32EnableDeadZone [in]

使能 DeadZone (1) / 禁止 DeadZone (0)

头文件

Driver/DrvPWM.h

返回值

无

示例

/* Enable PWM timer 0 and time 1 Dead-Zone function. PWM timer 0 and PWM timer 1 became a complementary pair. Set Dead-Zone time length to 100 and the unit time of Dead-Zone length which is the same as the unit of received PWM timer clock.*/

uint8_t u8DeadZoneLength = 100;

DrvPWM_EnableDeadZone (DRVPWM_TIMER0, u8DeadZoneLength, 1);

示例代码

/* Enable Timer0 and Timer1 Dead-Zone function and set Dead-Zone interval to 5us. */

Dead zone interval = [1 / (PWM0 engine clock source / sPt.u8PreScale /

sPt.u8ClockSelector)]* u8DeadZoneLength

= unit time * u8DeadZoneLength

= [1 / (12000000 / 6 / 1)] * 10 = 5us

uint8_t u8DeadZoneLength = 10; // Set dead zone length to 10 unit time

/* PWM Timer property */

sPt.u8Mode = DRVPWM_AUTO_RELOAD_MODE;

sPt.u8HighPulseRatio = 30; /* High Pulse peroid: Total Pulse peroid = 30 : 100 */

sPt.i32Inverter = 0;

sPt.u32Duty = 1000;

sPt.u8ClockSelector = DRVPWM_CLOCK_DIV_1;

sPt.u8PreScale = 6;

u8Timer = DRVPWM_TIMER0;

/* Select PWM engine clock source */

DrvPWM_SelectClockSource (u8Timer, DRVPWM_EXT_12M);

/* Set PWM Timer0 Configuration */

DrvPWM_SetTimerClk(u8Timer, &sPt);

/* Enable Output for PWM Timer0 */

DrvPWM_SetTimerIO(u8Timer, 1);

/* Enable Output for PWM Timer1 */

DrvPWM_SetTimerIO(DRVPWM_TIMER1, 1);

```
/* Enable Timer0 and Time1 dead zone function and Set dead zone length to 10 */
DrvPWM_EnableDeadZone (u8Timer, u8DeadZoneLength, 1);
/* Enable the PWM Timer 0 */
DrvPWM_Enable (u8Timer, 1);
```

DrvPWM_Enable

原型

void DrvPWM_Enable (uint8_t u8Timer, int32_t i32Enable);

描述

这个函数用于使能 PWM 定时器/捕获器功能。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

i32Enable [in]

使能(1) / 禁止(0)

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Enable PWM timer 0 function. */
DrvPWM_Enable (DRVPWM_TIMER0, 1);
/* Enable PWM capture 1 function.*/
DrvPWM_Enable (DRVPWM_CAP1, 1);
```

DrvPWM_SetTimerClk

原型

uint32_t DrvPWM_SetTimerClk (uint8_t u8Timer, S_DRVPWM_TIME_DATA_T *sPt);

描述

这个函数用于配置频率/脉冲/模式/反转功能。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

***sPt [in]**

包含以下参数

u32Frequency：定时器/捕获器频率(Hz)

u8HighPulseRatio：高脉冲比率(1~100)

u8Mode：DRVPWM_ONE_SHOT_MODE / DRVPWM_AUTO_RELOAD_MODE

bInverter：反转使能 (1) /反转禁止 (0)

u8ClockSelector：时钟选择器

DRVPWM_CLOCK_DIV_1：PWM 输入时钟除 1

DRVPWM_CLOCK_DIV_2：PWM 输入时钟除 1

DRVPWM_CLOCK_DIV_4：PWM 输入时钟除 1

DRVPWM_CLOCK_DIV_8：PWM 输入时钟除 1

DRVPWM_CLOCK_DIV_16：PWM 输入时钟除 1

(当 u32Frequency = 0 时该参数才起作用)

u8PreScale：预分频值(1~255)。

PWM 输入时钟 = PWM 源时钟/(u8PreScale + 1)。

(当 u32Frequency = 0 时该参数才起作用)

Note: 如果 *u8PreScale* 被设置为 0, 定时器会停止。

u32Duty：脉冲占空比(0x1~0x10000)

(当 u32Frequency = 0 或者 u8Timer = DRVPWM_CAP0/ DRVPWM_CAP1/ DRVPWM_CAP2/ DRVPWM_CAP3/ DRVPWM_CAP4/ DRVPWM_CAP5/ DRVPWM_CAP6/ DRVPWM_CAP7 时该参数才起作用)

头文件

Driver/DrvPWM.h

返回值

指定 PWM 的实际频率(Hz)

Note

1. 当用户设定一个非 0 的频率值时，该函数将会自动设定频率属性。
2. 当设定频率值为 0 时，用户也自己可以设定频率属性(时钟选择器/分频器/占空比)。
3. 该函数可以为 PWM 定时器/捕获器功能设定适当的频率属性(时钟选择器/分频器/占空比)，而且，用户需要自己设定合适的占空比。

示例代码

/* PWM timer 0 output 1 KHz waveform and duty cycle of waveform is 20% */

Method 1:

Fill sPt.u32Frequency = 1000 to determine the waveform frequency and

DrvPWM_SetTimerClk ( ) will set the frequency property automatically.

```
/* PWM Timer property */
sPt.u8Mode = DRVPWM_AUTO_RELOAD_MODE;
sPt.u8HighPulseRatio = 20; /* High Pulse peroid : Total Pulse peroid = 20 : 100 */
sPt.i32Inverter = 0;
sPt.u32Frequency = 1000; // Set 1 KHz to PWM timer output frequency
u8Timer = DRVPWM_TIMER0;
/* Select PWM engine clock */
DrvPWM_SelectClockSource (u8Timer, DRVPWM_HCLK);
/* Set PWM Timer0 Configuration */
DrvPWM_SetTimerClk(u8Timer, &sPt);
/* Enable Output for PWM Timer0 */
DrvPWM_SetTimerIO (u8Timer, 1);
/* Enable Interrupt Sources of PWM Timer 0 and install call back function */
DrvPWM_EnableInt (u8Timer, 0, DRVPWM_PwmIRQHandler);
/* Enable the PWM Timer 0 */
DrvPWM_Enable (u8Timer, 1);
```

Method 2:

Fill sPt.u8ClockSelector, sPt.u8PreScale and sPt.u32Duty to determine the output waveform frequency.

Assume HCLK frequency is 22MHz.

Output frequency = HCLK freq / sPt.u8ClockSelector / sPt.u8PreScale / sPt.u32Duty

= 22MHz / 1 / 22 / 1000 = 1KHz

```
/* PWM Timer property */
sPt.u8Mode = DRVPWM_AUTO_RELOAD_MODE;
sPt.u8HighPulseRatio = 20; /* High Pulse period : Total Pulse period = 20 : 100 */
sPt.i32Inverter = 0;
sPt.u8ClockSelector = DRVPWM_CLOCK_DIV_1;
sPt.u8PreScale = 22;
```

```
sPt.u32Duty = 1000;
u8Timer = DRVPWM_TIMER0;

/* Select PWM engine clock and user must know the HCLK frequency*/
DrvPWM_SelectClockSource (u8Timer, DRVPWM_HCLK);
/* Set PWM Timer0 Configuration */
DrvPWM_SetTimerClk(u8Timer, &sPt);
/* Enable Output for PWM Timer0 */
DrvPWM_SetTimerIO (u8Timer, 1);
/* Enable Interrupt Sources of PWM Timer0 and install call back function */
DrvPWM_EnableInt (u8Timer, 0, DRVPWM_PwmIRQHandler);
/* Enable the PWM Timer 0 */
DrvPWM_Enable (u8Timer, 1);
```

DrvPWM_SetTimerIO

原型

void DrvPWM_SetTimerIO (uint8_t u8Timer, int32_t i32Enable);

描述

这个函数可以用来使能/禁止 PWM 定时器/捕获器输入/输出功能。

参数

u8Timer [in]

指定定时器。

DRVPWM_TIMER0：PWM 定时器 0

DRVPWM_TIMER1：PWM 定时器 1

DRVPWM_TIMER2：PWM 定时器 2

DRVPWM_TIMER3：PWM 定时器 3

DRVPWM_TIMER4：PWM 定时器 4

DRVPWM_TIMER5：PWM 定时器 5

DRVPWM_TIMER6：PWM 定时器 6

DRVPWM_TIMER7：PWM 定时器 7

或者捕获器。

DRVPWM_CAP0：PWM 捕获器 0

DRVPWM_CAP1：PWM 捕获器 1

DRVPWM_CAP2：PWM 捕获器 2

DRVPWM_CAP3：PWM 捕获器 3

DRVPWM_CAP4：PWM 捕获器 4

DRVPWM_CAP5：PWM 捕获器 5

DRVPWM_CAP6：PWM 捕获器 6

DRVPWM_CAP7：PWM 捕获器 7

i32Enable [in]

使能(1) / 禁止(0)

头文件

Driver/DrvPWM.h

返回值

无

示例

```
/* Enable PWM timer 0 output.*/
DrvPWM_SetTimerIO (DRVPWM_TIMER0, 1);
/* Disable PWM timer 0 output. */
DrvPWM_SetTimerIO (DRVPWM_TIMER0, 0);
/* Enable PWM capture 3 input. */
DrvPWM_SetTimerIO (DRVPWM_CAP3, 1);
/* Disable PWM capture timer 3 input
DrvPWM_SetTimerIO (DRVPWM_CAP3, 0);
```

DrvPWM_SelectClockSource

原型

void DrvPWM_SelectClockSource(uint8_t u8Timer, uint8_t u8ClockSourceSelector);

描述

这个函数用于选择 PWM0 与 PWM1，PWM2 与 PWM3，PWM4 与 PWM5，PWM6 与 PWM7 的时钟源。

参数

u8Timer [in]

指定定时器

DRVPWM_TIMER0 或 DRVPWM_TIMER1：PWM 定时器 0 与 PWM 定时器 1.

DRVPWM_TIMER2 或 DRVPWM_TIMER3：PWM 定时器 2 与 PWM 定时器 3

DRVPWM_TIMER4 或 DRVPWM_TIMER5：PWM 定时器 4 与 PWM 定时器 5

DRVPWM_TIMER6 或 DRVPWM_TIMER7：PWM 定时器 6 与 PWM 定时器 7

u8ClockSourceSelector [in]

DRVPWM_EXT_12M / DRVPWM_HCLK / DRVPWM_INTERNAL_22M

DRVPWM_EXT_12M：外部 12 MHz crystal 时钟

DRVPWM_HCLK：HCLK

DRVPWM_INTERNAL_22M：内部 22 MHz crystal 时钟

头文件

Driver/DrvPWM.h

返回值

无

Note

1.PWM 定时器 0 和 PWM 定时器 1 使用同一个时钟源。如果用户把 PWM0 的时钟源从外部 12 MHz 改变为内部 22MHz，PWM1 的时钟源也会从外部 12MHz 改变为内部 22MHz。

2. PWM 定时器 2 和 PWM 定时器 3 使用同一个时钟源。

3. PWM 定时器 4 和 PWM 定时器 3 使用同一个时钟源。

4. PWM 定时器 2 和 PWM 定时器 3 使用同一个时钟源。

示例

```
Select PWM timer 0 and PWM timer 1 engine clock source from HCLK.
DrvPWM_SelectClockSource (DRVPWM_TIMER0, DRVPWM_HCLK);
Select PWM timer 6 and PWM timer 7 engine clock source from external 12MHz.
DrvPWM_SelectClockSource (DRVPWM_TIMER7, DRVPWM_EXT_12M);
```

DrvPWM_GetVersion

原型

uint32_t DrvPWM_GetVersion (void);

描述

获取该模块版本。

参数

无

头文件

Driver/DrvPWM.h

返回值

版本号：

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
/* Get PWM driver current version number */
int32_t i32PWMVersionNum ;
i32PWMVersionNum = DrvPWM_GetVersion();
```

10. FMC 驱动

10.1. FMC 介绍

NuMicro™ M051 系列配置了 64/32/16/8k 字节的片上嵌入式闪存，用于存储应用程序(APROM)，4k 字节用于存储 ISP 加载器(LDROM)的片上闪存，和用户配置(Config0)区域。用户配置区域提供若干字节用于控制系统逻辑状态，比如闪存安全锁，启动选择，Brown-Out 电压，数据闪存基地址，等等。NuMicro™ M051 系列还额外提供 4K 字节数据闪存，用户可以在芯片断电前在其中存放应用程序相关的数据。

10.2. FMC 特性

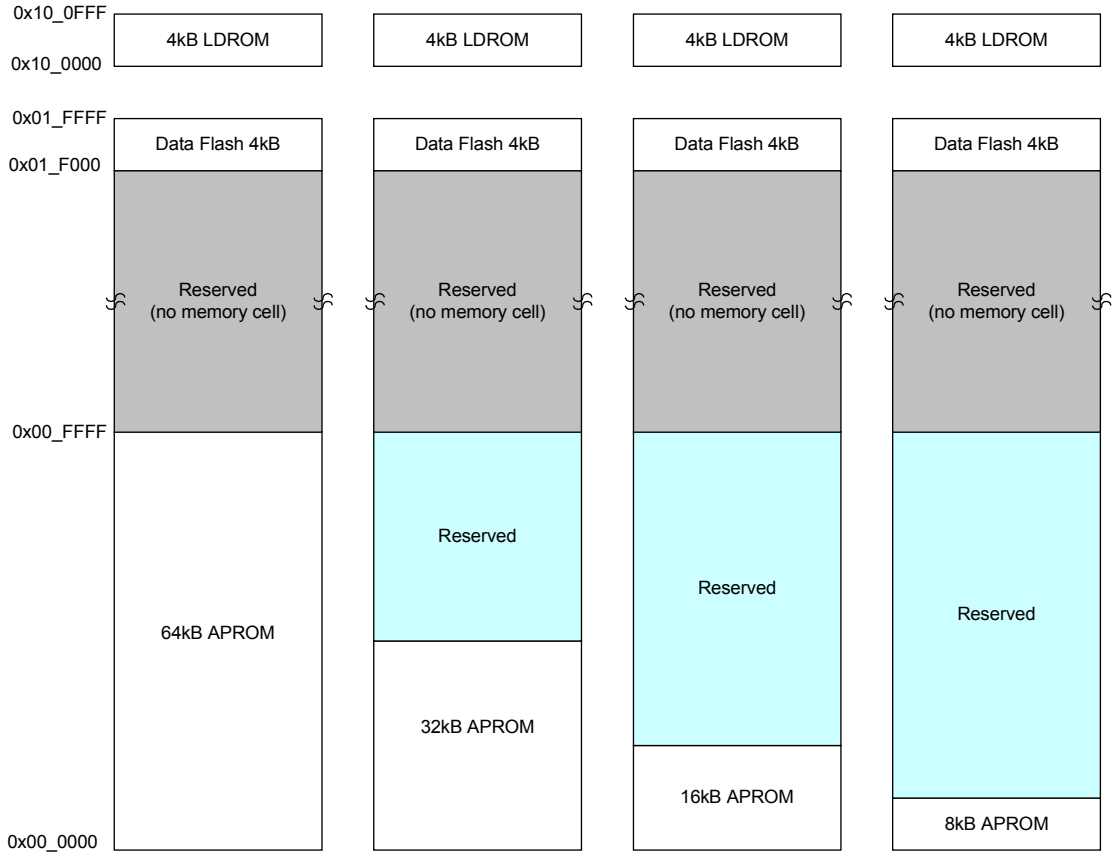
FMC 包含如下特性：

- 64/32/16/8kB 应用程序闪存(APROM)
- 4kB 在系统编程加载器闪存(LDROM)
- 4kB 数据闪存，擦除单位 512 字节
- 提供用户配置来控制系统逻辑
- 当 MCU 运行在 APROM 时，APROM 不能更新；当 MCU 运行在 LDROM 时，LDROM 不能更新

Memory Address Map

Block Name	Size	Start Address	End Address
AP ROM	64 KB 32 KB 16 KB 8 KB	0x00000000	0x0000FFFF 0x00007FFF 0x00003FFF 0x00001FFF
Data Flash	4 KB	0x0001F000	0x0001FFFF
LD ROM	4KB	0x00100000	0x00100FFF
User Configuration	1 words	0x00300000	0x00300000

Flash Memory Structure



8/16/32/64kB Flash Memory Structure

10.3. 类型定义

E_FMC_BOOTSELECT

枚举标识符	值	描述
E_FMC_APROM	0	从 APROM 启动
E_FMC_LDROM	1	从 LDROM 启动

10.4. 函数

DrvFMC_EnableISP

原型

void DrvFMC_EnableISP (void);

描述

使能 ISP 功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_EnableISP ( );   /* Enable ISP function */
```

DrvFMC_DisableISP

原型

```
void   DrvFMC_DisableISP (void);
```

描述

禁止 ISP 功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_DisableISP ( );   /* Disable ISP function */
```

DrvFMC_BootSelect

原型

```
void  DrvFMC_BootSelect(E_FMC_BOOTSELECT boot);
```

描述

选择下一次是从 APROM 还是 LDROM 启动。

参数

boot [in]

指定 E_FMC_APROM 或 E_FMC_LDROM

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_BootSelect (E_FMC_LDROM);    /* Next booting from LDROM */
DrvFMC_BootSelect (E_FMC_APROM);    /* Next booting from APROM */
```

DrvFMC_GetBootSelect

原型

E_FMC_BOOTSELECT DrvFMC_GetBootSelect(void);

描述

获取当前启动选择设定值。

参数

无.

头文件

Driver/DrvFMC.h

返回值

E_FMC_APROM 当前启动选择设定值是 APROM

E_FMC_LDROM 当前启动选择设定值是 LDROM

示例

```
E_FMC_BOOTSELECT e_bootSelect
/* Check this booting is from APROM or LDROM */
e_bootSelect = DrvFMC_GetBootSelect ( );
```

DrvFMC_EnableLDUpdate

原型

void DrvFMC_EnableLDUpdate (void);

描述

使能 LDROM 更新功能。当 MCU 运行在 APROM 时，如果 LDROM 更新功能被使能，则 LDROM 可以被更新。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_EnableLDUpdate ( );          /* Enable LDROM update function */
```

DrvFMC_DisableLDUpdate

原型

```
void   DrvFMC_DisableLDUpdate (void);
```

描述

禁止 LDROM 更新功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_DisableLDUpdate ( );         /* Disable LDROM update function */
```

DrvFMC_EnableConfigUpdate

原型

```
void   DrvFMC_EnableConfigUpdate (void);
```

描述

使能用户配置更新功能。如果用户配置更新功能使能，不管 MCU 是运行在 APROM 还是 LDROM，用户配置区域都可以被更新。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_EnableConfigUpdate ( );          /* Enable Config update function */
```

DrvFMC_DisableConfigUpdate

原型

```
void   DrvFMC_DisableConfigUpdate (void);
```

描述

禁止用户配置更新功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_DisableConfigUpdate ( );          /* Disable Config update function */
```

DrvFMC_EnablePowerSaving

原型

```
void   DrvFMC_EnablePowerSaving (void);
```

描述

使能 flash 访问省电功能。如果 CPU 时钟低于 24MHz，用户可以使能 flash 省电功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_EnablePowerSaving ( );          /* Enable flash power saving function */
```

DrvFMC_DisablePowerSaving

原型

```
void   DrvFMC_DisablePowerSaving (void);
```

描述

禁止 flash 访问省电功能。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_DisablePowerSaving ( );          /* Disable flash power saving function */
```

DrvFMC_Write

原型

```
int32_t  DrvFMC_Write (uint32_t u32addr, uint32_t u32data);
```

描述

写字数据到 APROM，LDROM，Data Flash 或 Config 区域。APROM 的存储器映射取决于 NuMicro™ M051 系列产品。APROM 大小请参考附录 NuMicro™ M051 系列选型指导 Config0 的相关功能在 TRM 中 FMC 部分有详细描述。

参数

u32addr [in]

APROM，LDROM，Data Flash 或 Config0 的字地址。

u32data [in]

要写入到 APROM，LDROM，Data Flash 或者 Config0 的字数据。

头文件

Driver/DrvFMC.h

返回值

0 成功

<0 失败

示例

```
/* Program word data 0x12345678 into address 0x1F000 */
DrvFMC_Write (0x1F000, 0x12345678);
```

DrvFMC_Read

原型

int32_t DrvFMC_Read (uint32_t u32addr, uint32_t * u32data);

描述

从 APROM，LDROM，Data Flash 或 Config 区域读取数据。APROM 的存储器映射取决于 NuMicro™ M051 系列产品。APROM 大小请参考附录 NuMicro™ M051 系列选型指导

参数

u32addr [in]

APROM，LDROM，Data Flash 或 Config0 的字地址。

u32data [in]

要存储从 APROM，LDROM，Data Flash 或者 Config0 读取的数据的字数据。

头文件

Driver/DrvFMC.h

返回值

0 成功

<0 失败

示例

```
uint32_t u32Data;
/* Read word data from address 0x1F000, and read data is stored to u32Data */
```

DrvFMC_Read (0x1F000, &u32Data);

DrvFMC_Erase

原型

int32_t DrvFMC_Erase (uint32_t u32addr);

描述

页擦除 APROM，LDROM，Data Flash 或者 Config0 区域。Flash 页擦除单位是 512 字节 APROM 的存储器映射取决于 NuMicro™ M051 系列产品。APROM 大小请参考附录 NuMicro™ M051 系列选型指导。

参数

u32addr [in]

APROM，LDROM 和 Data Flash 的页擦除基地址或者 Config0 区域地址。

头文件

Driver/DrvFMC.h

返回值

0　成功

<0　失败

示例

```
/* Page Erase from 0x1F000 to 0x1F1FF */
DrvFMC_Erase (0x1F000);
```

DrvFMC_WriteConfig

原型

int32_t DrvFMC_WriteConfig (uint32_t u32data0);

描述

擦除并写数据到 Config0 区域。Config0 区域中相应的功能的详细描述在 TRM 的 FMC 章节。

参数

u32data0 [in]

要写到 Config0 区域的字数据。

头文件

Driver/DrvFMC.h

返回值

0　成功

<0　失败

示例

```
/* Program word data 0xFFFFFFFE into Config0 */
DrvFMC_Config (0xFFFFFFFE);
```

DrvFMC_ReadDataFlashBaseAddr

原型

```
uint32_t   DrvFMC_ReadDataFlashBaseAddr (void);
```

描述

读取 data flash 基地址。对于 M051 系列，data flash 的基地址固定在地址 0x1F000。

参数

无

头文件

Driver/DrvFMC.h

返回值

Data Flash 基地址。

示例

```
uint32_t u32Data;
/* Read Data Flash base address */
u32Data = DrvFMC_ReadDataFlashBaseAddr ( );
```

DrvFMC_EnableLowSpeedMode

原型

```
void   DrvFMC_EnableLowSpeedMode (void);
```

描述

使能 flash 访问低速模式。当 CPU 运行在低速时可以提高 flash 访问性能。

Note

仅当 HCLK ≤ 25MHz 时置位该位。如果 HCLK > 25MHz，CPU 会获取到错误的代码，从而导致访问失败。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_EnableLowSpeedMode ( );          /* Enable flash access low speed mode */
```

DrvFMC_DisableLowSpeedMode

原型

```
void   DrvFMC_DisableLowSpeedMode (void);
```

描述

禁止 flash 访问的低速模式。

参数

无

头文件

Driver/DrvFMC.h

返回值

无

示例

```
DrvFMC_DisableLowSpeedMode ( );          /* Disable flash access low speed mode */
```

DrvFMC_GetVersion

原型

```
uint32_t DrvFMC_GetVersion (void);
```

描述

获取该模块的版本。

参数

无

头文件

Driver/DrvFMC.h

返回值

版本号：

31:24	23:16	15:8	7:0
00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

11. EBI 驱动

11.1. EBI 介绍

NuMicro™ M051 系列带有一个外部总线接口(EBI)供外部设备使用。为了节省外部设备和 MCU 芯片之间的引脚连接数量，EBI 支持地址总线和数据总线复用。而且，地址锁存使能(ALE)信号支持区分地址和数据周期。

11.2. EBI 特性

- 支持外部设备最大 64K 字节(8 比特数据宽度) / 128K 字节(16 比特数据宽度)
- 外部总线基时钟(MCLK)是可变的
- 支持 8 比特或者 16 比特数据宽度
- 支持可变的数据访问时间(tACC)，地址锁存使能时间(tALE)和地址保持时间(tAHD)
- 支持地址总线和数据总线复用模式来节省地址引脚
- 支持可配置的 idle 周期，用于不同的访问条件：写命令结束(W2X)，Read-to-Read (R2R)，Read-to-Write (R2W)
- 相应的时序控制波形图如下，

11.3. 类型定义

E_DRVEBI_BUS_WIDTH

枚举标识符	值	描述
E_DRVEBI_BUS_8BIT	0x0	EBI 数据总线宽度是 8 比特
E_DRVEBI_BUS_16BIT	0x1	EBI 数据总线宽度是 16 比特

E_DRVEBI_MCLKDIV

枚举标识符	值	描述
E_DRVEBI_MCLKDIV_1	0x0	EBI 输出时钟是 HCLK/1
E_DRVEBI_MCLKDIV_2	0x1	EBI 输出时钟是 HCLK/2
E_DRVEBI_MCLKDIV_4	0x2	EBI 输出时钟是 HCLK/4
E_DRVEBI_MCLKDIV_8	0x3	EBI 输出时钟是 HCLK/8
E_DRVEBI_MCLKDIV_16	0x4	EBI 输出时钟是 HCLK/16
E_DRVEBI_MCLKDIV_32	0x5	EBI 输出时钟是 HCLK/32
E_DRVEBI_MCLKDIV_DEFAULT	0x6	EBI 输出时钟是 HCLK/1

11.4. API 函数

DrvEBI_Open

原型

int32_t DrvEBI_Open (DRVEBI_CONFIG_T sEBIConfig)

描述

使能 EBI 功能，配置相关的 EBI 控制寄存器。

参数

sEBIConfig [in]

输入 EBI 控制寄存器设定值，DRVEBI_CONFIG_T 数据结构。

DRVEBI_CONFIG_T

eBusWidth：

E_DRVEBI_BUS_WIDTH，可以是E_DRVEBI_BUS_8BIT或 E_DRVEBI_BUS_16BIT

u32BaseAddress：

如果eBusWidth是8比特：0x60000000 <= u32BaseAddress <0x60010000

如果eBusWidth是16比特：0x60000000 <= u32BaseAddress <0x60020000

u32Size：

如果eBusWidth是8比特：0x0 < u32Size <= 0x10000

如果eBusWidth是16比特：0x0 < u32Size <= 0x20000

头文件

Driver/DrvEBI.h

返回值

E_SUCCESS：操作成功

E _DRVEBI_ERR_ARGUMENT：无效参数

示例

```
/* Open the EBI device with 16bit bus width. The start address of the device is at
0x60000000
and the storage size is 128KB */
DRVEBI_CONFIG_T sEBIConfig;
sEBIConfig.eBusWidth = eDRVEBI_BUS_16BIT;
sEBIConfig.u32BaseAddress = 0x60000000;
sEBIConfig.u32Size = 0x20000;
DrvEBI_Open (sEBIConfig);
```

DrvEBI_Close

原型

void DrvEBI_Close (void)

描述

禁止 EBI 功能，释放相应的引脚，这些被释放的引脚可用作 GPIO。

参数

无

头文件

Driver/DrvEBI.h

返回值

无

示例

```
/* Close the EBI device */
DrvEBI_Close ();
```

DrvEBI_SetBusTiming

原型

void DrvEBI_SetBusTiming (DRVEBI_TIMING_T sEBITiming)

描述

配置相关的 EBI 总线时序。

参数

sEBITiming [in]

DRVEBI_TIMING_T 结构体，包括
eMCLKDIV，u8ExttALE，u8ExtIR2R，u8ExtIR2W, u8ExtIW2X, u8ExttAHD 和 u8ExttACC

DRVEBI_TIMING_T

eMCLKDIV：

E_DRVEBI_MCLKDIV，可以是E_DRVEBI_MCLKDIV_1, E_DRVEBI_MCLKDIV_ 2, E_DRVEBI_MCLKDIV_ 4, E_DRVEBI_MCLKDIV_ 8, E_DRVEBI_MCLKDIV_16, E_DRVEBI_MCLKDIV_ 32或者E_DRVEBI_MCLKDIV_DEFAULT

u8ExttALE：ALE的扩展时序0~7，tALE = (u8ExttALE+1)*MCLK.

u8ExtIR2R：Read-Read之间的Idle周期0~15，idle周期 = u8Ext IR2R*MCLK

u8ExtIR2W：Read-Write之间的Idle周期0~15，idle周期 = u8ExtIR2W*MCLK

u8ExtIW2X：Write之后的Idle周期0~15，idle周期 = u8ExtIW2X*MCLK

u8ExttAHD：EBI总线保持时间0~7，tAHD = (u8ExttAHD+1)*MCLK

u8ExttACC：EBI数据访问时间0~31，tAHD = (u8ExttACC+1)*MCLK

头文件

Driver/DrvEBI.h

返回值

无

示例

```
/* Set the relative EBI bus timing */
DRVEBI_TIMING_T sEBITiming;
sEBITiming.eMCLKDIV = eDRVEBI_MCLKDIV_1;
sEBITiming.u8ExttALE = 0;
sEBITiming.u8Ext IR2R = 0;
sEBITiming.u8Ext IR2W = 0;
sEBITiming.u8Ext IW2X = 0;
sEBITiming.u8ExttAHD = 0;
sEBITiming.u8ExttACC = 0;
DrvEBI_SetBusTiming (sEBITiming);
```

DrvEBI_GetBusTiming

原型

void DrvEBI_GetBusTiming (DRVEBI_TIMING_T *psEBITiming)

描述

获取 EBI 当前总线时序。

参数

psEBITiming [out]

DRVEBI_TIMING_T 结构体，包括 eMCLKDIV，u8ExttALE，u8ExtIR2R，u8ExtIR2W, u8ExtIW2X, u8ExttAHD 和 u8ExttACC

头文件

Driver/DrvEBI.h

返回值

存储 EBI 总线时序设定值的数据缓存指针

示例

```
/* Get the current EBI bus timing */
DRVEBI_TIMING_T sEBITiming;
DrvEBI_GetBusTiming (&sEBITiming);
```

DrvEBI_GetVersion

原型

uint32_t DrvEBI_GetVersion (void);

描述

获取 EBI 驱动的版本号。

参数

无

头文件

Driver/ DrvEBI.h

返回值

版本号：

31:24	23:16	15:8	7:0

00000000	MAJOR_NUM	MINOR_NUM	BUILD_NUM

示例

```
/* Get the current version of EBI Driver */
u32Version = DrvEBI_GetVersion ();
```

12. 附录

12.1. NuMicro™ M051 系列选型指导

Part number	Flash	SRAM	Data Flash	Connectivity			PWM	ADC	Timer	EBI	ISP ICP	I/O	Package
				UART	SPI	I2C							
M052LAN	8 KB	4 KB	4 KB	2	2	1	8	8x12-bit	4	v	v	up to 38	LQFP48
M052ZAN	8 KB	4 KB	4 KB	2	1	1	5	5x12-bit	4	-	v	up to 22	QFN32
M054LAN	16 KB	4 KB	4 KB	2	2	1	8	8x12-bit	4	v	v	up to 38	LQFP48
M054ZAN	16 KB	4 KB	4 KB	2	1	1	5	5x12-bit	4	-	v	up to 22	QFN32
M058LAN	32 KB	4 KB	4 KB	2	2	1	8	8x12-bit	4	v	v	up to 38	LQFP48
M058ZAN	32 KB	4 KB	4 KB	2	1	1	5	5x12-bit	4	-	v	up to 22	QFN32
M0516LAN	64 KB	4 KB	4 KB	2	2	1	8	8x12-bit	4	v	v	up to 38	LQFP48
M0516ZAN	64 KB	4 KB	4 KB	2	1	1	5	5x12-bit	4	-	v	up to 22	QFN32

12.2. 产品 ID (PDID) Table

Part number	PDID
M052LAN	0x00005200
M052ZAN	0x00005203
M054LAN	0x00005400
M054ZAN	0x00005403
M058LAN	0x00005800
M058ZAN	0x00005803
M0516LAN	0x00005A00
M0516ZAN	0x00005A03

13. Revision History

Version	Date	Description
V1.00.001	Jan. 8, 2009	• Created
V1.00.002	July. 30, 2010	• Fix errors • Add example of API

Important Notice

Nuvoton products are not designed, intended, authorized or warranted for use as components in equipment or systems intended for surgical implantation, atomic energy control instruments, aircraft or spacecraft instruments, transportation instruments, traffic signal instruments, combustion control instruments, or for any other applications intended to support or sustain life. Furthermore, Nuvoton products are not intended for applications whereby failure could result or lead to personal injury, death or severe property or environmental damage.

Nuvoton customers using or selling these products for such applications do so at their own risk and agree to fully indemnify Nuvoton for any damages resulting from their improper use or sales.