NuMicro™ NUC100 系列
驱动参考指南
V1.01.002

发布日期: 8. 2010

Support Chips:

NuMicro™ NUC100 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.

目录

1. 概述

1.1. 文档结构

本文档是 NUC100 系列驱动参考手册，系统级软件开发人员可以使用 NUC100 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 NUC100 的 BSP（板级支持包）里。

本文档分为若干个章节，第一章是概述。第二章到第十七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，RTC 驱动，CAN 驱动，PWM 驱动，PS2 驱动，FMC 驱动，USB 驱动，PDMA 驱动，I2S 驱动和 EBI 驱动。

附录是 NUC100 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：

- NuMicroTM NUC100 series Technical Reference Manual(TRM)
- NuMicroTM NUC100 series Application Notes
  - AN1001EN How to create an USB HID V1.00.pdf
  - AN1002EN How to use ADC V1.01.pdf
  - AN1003EN How to use TIMER V1.01.pdf
  - AN1004EN How to print information into PC via UART V1.01.pdf
  - AN1005EN How to use WDT V1.01.pdf
  - AN1006EN How to use AD7793 via SPI V1.01.pdf

- AN1007EN Power Management V1.00.pdf
- AN1008EN How to use CAN transmit or receive messages V1.00.pdf
- AN1009EN How to access 24C64 via IIC V1.00.pdf
- AN1010EN How to use PWM V1.00.pdf
- AN1012EN Configurat ion V1.00.pdf
- AN1015EN How to use I2S with CODEC V1.00.pdf
- AN1016EN_How to use PDMA to transfer data V1.00.pdf
- AN1020EN_USB MassStorage V1.00.pdf
- AN1022EN How to use SPI 2-Bit Mode V1.00.pdf

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
BUF	缓冲区
CAN	局域控制网络总线
CFG	配置
DSQ	数据序列
EBI	外部总线接口
EP	端点
FIFO	先进先出
FLD	悬空检测
FMC	Flash存储控制器
GPIO	通用输入/输出
I2C	内部集成电路
I2S	音频集成IC
LIN	本地互联网络
LVR	低压复位
PDID	产品设备ID
PDMA	外围设备直接存储器访问
PHY	物理层

PLL	锁相环
POR	上电复位
PWM	脉宽调制
PS/2	IBM个人系统/2
SPI	串行外围设备接口
TOG	切换
TRIG	触发
UART	通用异步收/发

1.4. 数据类型定义

NUC100 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	signed 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

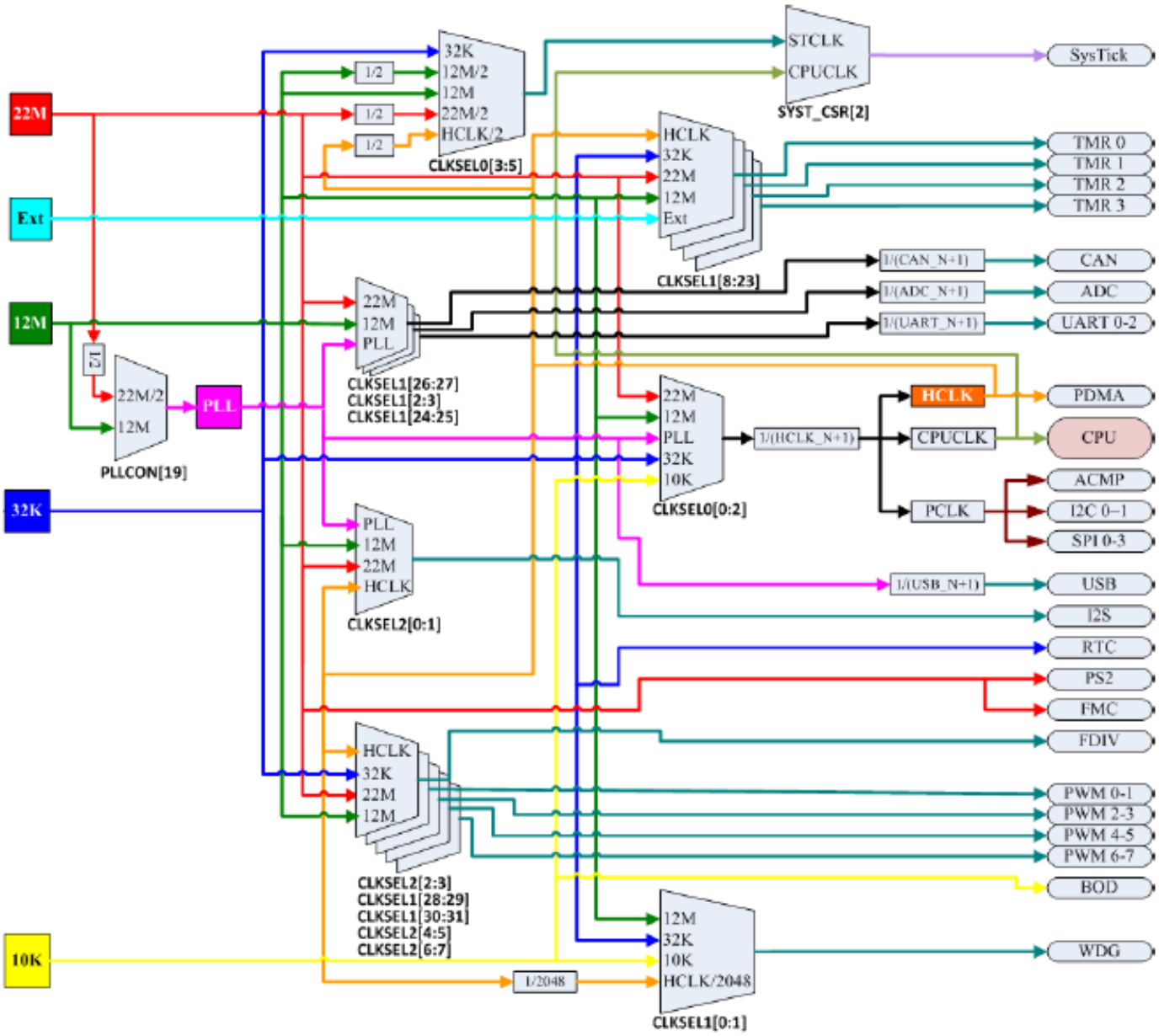
2.1. 介绍

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初始化.
- 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	定时器 0 复位
E_SYS_TMR1_RST	3	定时器 1 复位
E_SYS_TMR2_RST	4	定时器 2 复位
E_SYS_TMR3_RST	5	定时器 3 复位
E_SYS_I2C0_RST	8	I2C0 复位
E_SYS_I2C1_RST	9	I2C1 复位
E_SYS_SPI0_RST	12	SPI0 复位
E_SYS_SPI1_RST	13	SPI1 复位
E_SYS_SPI2_RST	14	SPI2 复位
E_SYS_SPI3_RST	15	SPI3 复位
E_SYS_UART0_RST	16	UART0 复位
E_SYS_UART1_RST	17	UART1 复位
E_SYS_UART2_RST	18	UART2 复位
E_SYS_PWM03_RST	20	PWM0~3 复位
E_SYS_PWM47_RST	21	PWM4~7 复位
E_SYS_ACMP_RST	22	模拟比较器复位
E_SYS_PS2_RST	23	PS2 复位
E_SYS_CAN0_RST	24	CAN0 复位
E_SYS_USBD_RST	27	USB 设备复位
E_SYS_ADC_RST	28	ADC 复位
E_SYS_I2S_RST	29	I2S 复位
E_SYS_PDMA_RST	32	PDMA 复位
E_SYS_EBI_RST	33	EBI 复位

E_SYS_IP_CLK

枚举标识符	值	描述
E_SYS_WDT_CLK	0	Watch Dog 时钟使能
E_SYS_RTC_CLK	1	RTC 时钟使能
E_SYS_TMR0_CLK	2	定时器 0 时钟使能
E_SYS_TMR1_CLK	3	定时器 1 时钟使能
E_SYS_TMR2_CLK	4	定时器 2 时钟使能
E_SYS_TMR3_CLK	5	定时器 3 时钟使能
E_SYS_FDIV_CLK	6	频率分频器时钟使能
E_SYS_I2C0_CLK	8	I2C0 时钟使能
E_SYS_I2C1_CLK	9	I2C1 时钟使能
E_SYS_SPI0_CLK	12	SPI0 时钟使能
E_SYS_SPI1_CLK	13	SPI1 时钟使能

枚举标识符	值	描述
E_SYS_SPI2_CLK	14	SPI2 时钟使能
E_SYS_SPI3_CLK	15	SPI3 时钟使能
E_SYS_UART0_CLK	16	UART0 时钟使能
E_SYS_UART1_CLK	17	UART1 时钟使能
E_SYS_UART2_CLK	18	UART2 时钟使能
E_SYS_PWM01_CLK	20	PWM01 时钟使能
E_SYS_PWM23_CLK	21	PWM23 时钟使能
E_SYS_PWM45_CLK	22	PWM45 时钟使能
E_SYS_PWM67_CLK	23	PWM67 时钟使能
E_SYS_CAN0_CLK	24	CAN0 时钟使能
E_SYS_USBD_CLK	27	USB 设备时钟使能
E_SYS_ADC_CLK	28	ADC 时钟使能
E_SYS_I2S_CLK	29	I2S 时钟使能
E_SYS_ACMP_CLK	30	模拟比较器时钟使能
E_SYS_PS2_CLK	31	PS2 时钟使能
E_SYS_PDMA_CLK	33	PDMA 时钟使能
E_SYS_ISP_CLK	34	Flash ISP 控制器时钟使能
E_SYS_EBI_CLK	35	EBI 时钟使能

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
E_SYS_INTERNAL_22M	1	PLL 源时钟来自内部 22M

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_CAN_DIV	1	CAN 源时钟分频器设定
E_SYS_UART_DIV	2	UART 源时钟分频器设定
E_SYS_USB_DIV	3	USB 源时钟分频器设定
E_SYS_HCLK_DIV	4	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	Watch Dog Timer 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

枚举标识符	值	描述
E_SYS_CAN_CLKSRC	7	CAN 时钟源设定
E_SYS_PWM01_CLKSRC	8	PWM01 时钟源设定
E_SYS_PWM23_CLKSRC	9	PWM23 时钟源设定
E_SYS_I2S_CLKSRC	10	I2S 时钟源设定
E_SYS_FRQDIV_CLKSRC	11	频率分频器输出时钟源设定
E_SYS_PWM45_CLKSRC	12	PWM45 时钟源设定
E_SYS_PWM67_CLKSRC	13	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 晶体振荡器
E_SYS_XTL32K	1	选择外部 32K 晶振振荡器
E_SYS_OSC22M	2	选择内部 22M 振荡器
E_SYS_OSC10K	3	选择内部 10K 振荡器
E_SYS_PLL	4	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中 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。

位编号	描述
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 RSTSRC 寄存器相应指示标志。

参数

u32Src [in]

要清除的比特

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, I2C0, I2C1, SPI0, SPI1, SPI2, SPI3, UART0, UART1, UART2, PWM03, PWM47, ACMP, PS2, CAN0, USBD, ADC, I2S，PDMA.和 EBI。

参数

eIpRst [in]

要复位的 IP，参考 2.3 节的 E_SYS_IP_RST 的定义。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_ResetIP(E_SYS_I2C0_RST);     /* Reset I2C0 */
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 BOD 极限电压

参数

u8Volt [in]

可能的值：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, 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 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 Brow-out 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_EnableLowVoltReset(void);       /* 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_DisableLowVoltReset(void);       /* Disable low voltage reset function */
```

DrvSYS_GetBODState

原型

```
uint32_t DrvSYS_GetBODState(void);
```

描述

取得 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 u32Flag；
/* Get Brown-Out state if Brown-Out detector function is enabled */
U32Flag = DrvSYS_GetBODState();
```

DrvSYS_EnableTemperatureSensor

原型

```
void   DrvSYS_EnableTemperatureSensor(void);
```

描述

使能温度传感器功能。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_EnableTemperatureSensor();     /* Enable temperature sensor function */
```

DrvSYS_DisableTemperatureSensor

原型

```
void   DrvSYS_DisableTemperatureSensor(void);
```

描述

禁止温度传感器功能。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
DrvSYS_DisableTemperatureSensor();     /* Disable temperature sensor function */
```

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，为了安全考虑，一些寄存器被加锁，要写这些寄存器需要先开锁。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。建议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 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_SetRCAdjValue

原型

void DrvSYS_SetRCAdjValue(uint32_t u32Adj);

描述

设定电阻电容振荡器(RC oscillator)的调整值来修正振荡器的频率。RC 的调整值越大，振荡器的输出频率越小，调整的单步值大约是 1%，中间数字是 0x20，即 0%。下表给出了调整步数与频率偏移量之间的关系。

调整值	偏移量
0x0	~ -32%
…	
0x19	~ -1%
0x20	~ 0%
0x21	~ 1%
…	
0x3F	~ 31%

参数

u32Adj [in]

RC 调整值，范围是 0x00~0x3F。

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Set RC adjustment value 0x20 */
DrvSYS_SetRCAdjValue(0x20);
```

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, 实时时钟, 定时器 0, 定时器 1, 定时器 2, 定时器 3, I2C0, I2C1, SPI0, SPI1, SPI2, SPI3, UART0, UART1, UART2, PWM01, PWM23, PWM45, PWM67, CAN0, USBD, ADC, I2S, ACMP, PS2, PDMA, EBI, Flash ISP 控制器和频率分频器输出。

参数

eIpClk [in]

要设定时钟的 IP, 参考本文档 2.3 节中 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, 外部 32K 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

1：外部 32K 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 32K crystal 时钟/2, 外部 12M crystal 时钟/2, HCLK/2, 或者内部 22M oscillator 时钟/2。SysTick 定时器是属于 Cortex-M0 的一个标准的定时器。

参数

u8ClkSrcSel [in]

0：外部 12M 时钟

1：外部 32K 时钟

2：外部 12M 时钟/ 2

3：HCLK / 2

4~7：内部 22M 时钟/ 2

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, CAN, PWM01, PWM23, PWM45, PWM67, I2S 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_CAN_CLKSRC / E_SYS_PWM01_CLKSRC
E_SYS_PWM23_CLKSRC / E_SYS_PWM45_CLKSRC /
E_SYS_PWM67_CLKSRC/E_SYS_FRQDIV_CLKSRC/E_SYS_I2S_CLKSRC

u8ClkSrcSel　[in]

相应 IP 的时钟源。

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);
/* Select I2S clock source from HCLK */
DrvSYS_SelectIPClockSource(E_SYS_I2S_CLKSRC,0x02);
```

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_CAN_DIV / E_SYS_UART_DIV

E_SYS_USB_DIV /E_SYS_HCLK_DIV

i32value [in]

除频值。范围：

HCLK, USB, UART：0~15

CAN：0~15 中密度系列

0~1023 低密度系列

请参考附录中 NuMicro™ NUC100 系列产品选型表。

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 clock source / (1+1) */
DrvSYS_SetClockDivider(E_SYS_ADC_DIV, 0x01);
/* Set UART clock divide number 0x02; UART clock = UART clock source / (2+1) */
DrvSYS_SetClockDivider(E_SYS_UART_DIV, 0x02);
/* Set HCLK clock divide number 0x03; HCLK clock = HCLK clock source / (3+1) */
DrvSYS_SetClockDivider(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 32K 和 12M crystal.

参数

eClkSrc [in]

E_SYS_XTL12M / E_SYS_XTL32K / 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/关闭 power down 唤醒中断，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 时钟状态稳定。当 GPIO, USB, UART, WDT, CAN, ACMP, BOD 或者 RTC 被唤醒的时候，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 ePDTpye);

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

ePDTpye [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 down mode immediately */
DrvSYS_EnterPowerDown(E_SYS_IMMEDIATE);
/* Wait for CPU enters sleep mode, then Chip enter power down mode immediately */
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 指定的目标 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

设置 PLL 控制寄存器。用户可以使用 DrvSYS_GetPLLContent()获取合适的 PLL 设定值，使用 DrvSYS_GetPLLClockFreq()获取实际 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_SYS_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

取得 PLL 时钟输出频率。

参数

无

头文件

Driver/DrvSYS.h

返回值

PLL 时钟频率，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

取得 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_GetHCLKClockFreq();      /* Get actual 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()可以用来获得实际的 PLL 时钟频率。

DrvSYS_GetHCLKFreq()可以用来获得实际的 HCLK 时钟频率。

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

NUC100 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，用户可以使用该函数使能时钟输出功能。然而，用户还需要把 CLKO 设定为输出引脚，这可以通过配置相应的多功能 GPIO 控制寄存器的相应为实现。

参数

i32Flag [in]

1：使能，0：禁止

U8Divider [in]

满足输出频率的除频值，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_EnableHighPerformanceMode

原型

void DrvSYS_EnableHighPerformanceMode(void);

描述

使能芯片高性能模式。当该功能使能，对于内部 RAM 和 GPIO 的访问是零等待的。

Note

仅低密度版本和 NuMicroTM NUC100 系列的 NUC101 支持该功能。详细信息请参考附录 NuMicroTM NUC100 系列产品选型指导。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Enable high performance mode */
DrvSYS_EnableHighPerformanceMode();
```

DrvSYS_DisableHighPerformanceMode

原型

```
void  DrvSYS_DisableHighPerformanceMode(void);
```

描述

禁止芯片高性能模式。

Note

仅低密度版本和 NuMicroTM NUC100 系列的 NUC101 支持该功能。详细信息请参考附录 NuMicroTM NUC100 系列产品选型指导。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
/* Disable high performance mode */
DrvSYS_DisableHighPerformanceMode();
```

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

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 32K, 12M, 或 PLL 时钟。

Note

仅低密度版本和 NuMicroTM NUC100 系列的 NUC101 支持该功能。详细信息请参考附录 NuMicroTM NUC100 系列产品选型指导。

参数

eClkSrc [in]

E_SYS_XTL12M/E_SYS_XTL32K/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。

Note

仅低密度版本和 NuMicroTM NUC100 系列的 NUC101 支持该功能。详细信息请参考附录 NuMicroTM NUC100 系列产品选型指导。

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。

Note

仅低密度版本和 NuMicroTM NUC100 系列的 NUC101 支持该功能。详细信息请参考附录 NuMicroTM NUC100 系列产品选型指导。

参数

无

头文件

Driver/DrvSYS.h

返回值

无

示例

```
uint32_t u32flag;
DrvSYS_SelectHCLKSource (2);      /* Change HCLK clock source to be PLL */
u32flag = DrvSYS_GetClockSwitchStatus ();   /* Get clock switch flag */
if (u32flag)
  DrvSYS_ClearClockSwitchStatus ();    /* Clear clock switch fail flag */
```

DrvSYS_GetVersion

原型

uint32_t DrvSYS_GetVersion(void);

描述

获取 DrvSYS 驱动的版本号。

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. 串口介绍

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.

细节请参考芯片说明书 UART 章节.

3.2. 串口特性

串口包含下面的特性:

- 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 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	0x1040000	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：1.5 比特
DRVUART_STOPBITS_2	0x1	停止位长度：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

从串口发送 1 字节数据。

头文件

Driver/DrvUART.h

返回值

无

示例

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

_DRVUART_RECEIVEBYTE

原型

uint_8 _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

int_8 _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

初始化串口。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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

sParam [in]

指定串口的特性。包括

u32BaudRate: 波特率(Hz)

u8cParity: 无/奇/偶校验

可取值为：

DRVUART_PARITY_NONE(None parity).

DRVUART_PARITY_EVEN(Even parity).

DRVUART_PARITY_ODD(Odd parity).

u8cDataBits: 数据位设定

可取值为：

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

可取值为：

DRVUART_STOPBITS_1 (1 比特).

STOPBITS_1_5 或者(1.5 比特).

DRVUART_STOPBITS_2 (2 比特).

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。

可取值为：

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_DATA_BITS_8;
sParam.u8cStopBits        =DRVUART_STOP_BITS_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

关闭串口时钟和中断，在检验 Tx 为空时清除回调函数指针。

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

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

使能指定串口中断，安装中断回调函数，并使能 NVIC 串口中断。

参数

u32Port [in]

指定串口： UART_PORT0/UART_PORT1/UART_PORT2

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断

DRVUART_BUFERRINT : Buffer Error 中断

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 Interrupt and Time-out 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

关闭 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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断

DRVUART_BUFERRINT : Buffer Error 中断

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 Interrupt and Time-out 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

用来清除串口指定中断标志

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断

DRVUART_BUFERRINT : Buffer Error 中断

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 clear UART channel 0 Tx empty Interrupt Flag */
DrvUART_ClearIntFlag(UART_PORT0, DRVUART_THREINT);
```

DrvUART_GetIntStatus

原型

```
int8_t
DrvUART_GetIntStatus (
    E_UART_PORT u32Port
    uint32_t    u32InterruptFlag
);
```

描述

这个函数用来取得指定串口中断状态

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

u32InterruptFlag [in]

DRVUART_LININT : LIN RX Break Field Detected 中断

DRVUART_BUFERRINT : Buffer Error 中断

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。

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_GetCTS (
```

```
    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]

指定串口：UART_PORT0/UART_PORT1(UART_PORT2 is not supported.)

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 信息

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1(UART_PORT2 is not supported.)

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 to 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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

pu8RxBuf [out]

指定存放接收到的数据的缓存地址。

u32ReadBytes [in]

设定要接收的字节数。

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，然后由串口发送出去

参数

u32Port [in]

说明串口：UART_PORT0/UART_PORT1/UART_PORT2

pu8TxBuf [in]

指定用于发送数据到串口 FIFO 的缓存地址。

u32WriteBytes [in]

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 to buffer to TX FIFO. */
Uint8_t bInChar[1] = 0x55;
DrvUART_Write(UART_PORT0, bInChar, 1);
```

DrvUART_EnablePDMA

原型

```
void
DrvUART_EnablePDMA (
    E_UART_PORT u32Port
);
```

描述

该函数用于控制使能 PDMA 发送/接收通道

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1(UART_PORT2 is not supported.)

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Enable TX and RX PDMA in UART1 */
DrvUART_EnablePDMA(UART_PORT1);
```

DrvUART_DisablePDMA

原型

```
void
DrvUART_EnablePDMA (
    E_UART_PORT u32Port
);
```

描述

该函数用于控制禁止 PDMA 发送/接收通道

参数

u32Port [in]

指定串口：UART_PORT0/UART_PORT1(UART_PORT2 is not supported.)

头文件

Driver/DrvUART.h

返回值

无

示例

```
/* Disable TX and RX PDMA in UART1 */
DrvUART_DisablePDMA(UART_PORT1);
```

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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

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

None

示例

```
/* Condition: Change UART1 to RS485 function. Set relat ive 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]

指定串口：UART_PORT0/UART_PORT1/UART_PORT2

u16Mode [in]

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 informat ion. */
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定时。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。定时器可产生中断信号并根据中断发生点记录当前数据。

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

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_CONTINUOUS_MODE	0x3	设定定时器为 Continuous Counting 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.

uTickPerSecond [in]

定时器每秒中断 tick 数。

op_mode [in]

E_TIMER_OPMODE, 可以是 E_ONESHOT_MODE/

E_PERIODIC_MODE /E_TOGGLE_MODE/E_CONTINUOUS_MODE.

头文件

Driver/DrvTimer.h

返回值

E_SUCCESS：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。

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：定时器通道无效。

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 uTimeEventNo
);

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.

uTimeEventNo [in]

定时器事件编号。

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_DisableInt (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 说明 “没有中断”, 1 说明 “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

Example

```
/* 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

uin32_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 ticks

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_OpenCounter

原型

```
int32_t DrvTIMER_OpenCounter (
    E_TIMER_CHANNEL ch,
    uint32_t uCounterBoundary,
    E_TIMER_OPMODE op_mode
);
```

描述

该函数用于打开定时器通道，并指定其操作模式。定时器计数源为外部事件/计数器。

Note

只有 NuMicro™ NUC100 系列的低密度系列支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

参数

ch [in]

E_TIMER_CHANNEL, 可以是 E_TMR0 / E_TMR1 / E_TMR2 / E_TMR3

uCounterBoundary [in]

该参数决定了计数值达到多少次会引发一次定时器中断。

op_mode [in]

E_TIMER_OPMODE, 包括 E_ONESHOT_MODE / E_PERIODIC_MODE / E_CONTINUOUS_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_EIO：定时器未初始化

示例

```
/* Set Timer-0 run in One-Shot mode by external counter.
And when the counter counting to 123, Time r-0 interrupt will occurred */
DrvTIMER_OpenCounter (E_TMR0, 123, E_ONESHOT_MODE);
```

DrvTIMER_StartCounter

原型

int32_t DrvTIMER_StartCounter (E_TIMER_CHANNEL ch);

描述

指定定时器通道开始计数。

Note

只有 NuMicro™ NUC100 系列的低密度系列支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

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 by external counter */

DrvTIMER_StartCounter (E_TMR0);

DrvTIMER_GetCounters

原型

int32_t DrvTIMER_GetCounters (E_TIMER_CHANNEL ch);

描述

该函数用于获取指定定时器通道的当前计数值。只有 NuMicro™ NUC100 系列的低密度系列支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

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

u32Counters：返回当前计数值

E_DRVTIMER_CHANNEL：无效定时器通道

示例

/* Get the current counts of Timer -0 */

u32TMR0ExtTicks = DrvTIMER_GetCounters (E_TMR0);

DrvTIMER_GetVersion

原型

uint32_t

DrvTimer_GetVersion (void);

描述

返回 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

Example

/* 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_DRVTIMER_CHANNEL：无效定时器通道

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

NUC100 中密度系列 MCU 有多达 80 个通用 I/O 引脚，可以与其他功能引脚复用，这取决于芯片配置。80 个引脚分配在 GPIOA，GPIOB，GPIOC，GPIOD 与 GPIOE 五个口上，每个口最多 16 个引脚。

NUC100 中密度系列 MCU 有多达 65 个通用 I/O 引脚，可以与其他功能引脚复用，这取决于芯片配置和封装。80 个引脚分配在五个口上，其中，GPIOA，GPIOB，GPIOC 和 GPIOD 每个口最多有 16 个引脚，而 GPIOE 只有一个引脚 GPE[5]。

5.2. GPIO 特性

- 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_GPA	0	定义 GPIO 端口 A
E_GPB	1	定义 GPIO 端口 B
E_GPC	2	定义 GPIO 端口 C
E_GPD	3	定义 GPIO 端口 D
E_GPE	4	定义 GPIO 端口 E

E_DRVGPIO_IO

枚举标识符	值	描述
E_IO_INPUT	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	引脚分配	描述
E_FUNC_GPIO	所有 GPIO 引脚	设定所有的 GPIO 引脚为 GPIO 功能
E_FUNC_CLKO	GPB.12	使能频率输出功能
E_FUNC_I2C0/ E_FUNC_I2C1	GPA.8~9/GPA.10~11	使能 I2C0/I2C1 功能
E_FUNC_I2S	GPA.15, GPC.0~3	使能 I2S 功能
E_FUNC_CAN0	GPD.6, GPD.7	使能 CAN 功能
E_FUNC_ACMP0/ E_FUNC_ACMP1	GPC.6~7/GPC.14~15	使能 ACMP0/ACMP1 功能
E_FUNC_SPI0/ E_FUNC_SPI1/ E_FUNC_SPI2/ E_FUNC_SPI3	GPB.10, GPC.0~3 GPB.9, GPC.8~11 GPA.9, GPD.0~13 GPB.14, GPD.8~11	使能 SPI0/SPI1/SPI2/SPI3 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	GPA.0/ GPA.1/ GPA.2/ GPA.3/ GPA.4/ GPA.5/ GPA.6/ GPA.7	使能 ADC0/ADC1/ADC2/ADC3/ ADC4/ADC5/ADC6/ADC7 功能
E_FUNC_EXINT0/ E_FUNC_EXINT1	GPB.14/GPB.15	使能外部 INT0/INT1 功能
E_FUNC_TMR0/ E_FUNC_TMR1/ E_FUNC_TMR2/ E_FUNC_TMR3	GPB.8/GPB.9/ GPB.10/GPB.11	使能 TIMER0/TIMER1/TIMER2/ TIMER3 为 Toggle/Counter 功能
E_FUNC_UART0/E_FUNC_UART1 / E_FUNC_UART2	GPB.0~3/GPB.4~7/ GPD.14~15	使能 UART0/UART1/UART2 功能
E_FUNC_PWM01/ E_FUNC_PWM23/ E_FUNC_PWM45/ E_FUNC_PWM67	GPA.12~13/ GPA.14~15/ GPB.11, GPE.5/ GPE.0~1	使能 PWM01/PWM23/PWM45/ PWM67 功能
E_FUNC_EBI_8B	GPA.6~7, GPA.10~11 GPB.6~7, GPB.12~13 GPC.6~8, GPC.14~15	使能 EBI，数据宽度为 8 位

E_FUNC_EBI_16B	GPA.1~7, GPA.10~11 GPA.13~15, GPB.2~3 GPB.6~7, GPB.12~13 GPC.6~8, GPC.14~15	使能 EBI，数据宽度为 16 位
E_FUNC_SPI0_QFN36PIN	GPC.0~3, GPD.1	使能 QFN36PIN 封装芯片 SPI0 功能

5.4. 宏定义

_DRVGPIO_DOUT

原型

_DRVGPIO_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。用户可以通过调用 _DRVGPIO_DOUT 宏来设定指定引脚的输出数据，若该引脚被配置为输出模式。或者当 GPIO 引脚配置为输入模式时，通过直接调用_DRVGPIO_DOUT 来获取输入数据。只有 NUC1xx 低密度系列支持该功能。

Note

只有 NuMicro™ NUC100 系列的低密度系列支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

参数

PortNum [in]

指定 GPIO 口。可以为 0~3，对应于 GPIO-A/B/C/D。

PinNum [in]

指定 GPIO 口的引脚。可以为 0~15。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Configure GPA-1 to output mode */
DrvGPIO_Open (E_GPA, 1, E_IO_OUTPUT);
/* Set GPA-1 to high */
_DRVGPIO_DOUT (E_GPA, 1) = 1;
/* ..... */
```

```
/* Configure GPB-3 to input mode */
uint8_t u8PinValue;
DrvGPIO_Open (E_GPB, 3, E_IO_INPUT);
/* Get GPB-3 pin value */
u8PinValue = _DRVGPIO_DOUT (E_GPB, 3);
```

GPA_[n] / GPB_[n] / GPC_[n] / GPD_[n]

原型

GPA_0~GPA_15 / GPB_0~GPB_15 / GPC_0~GPC_15 / GPD_0~GPD_15

描述

这些宏定义跟_DRVGPIO_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 GPA_0 往指定的引脚输出数据，或者从指定的引脚获取该引脚值。只有 NUC1xx 低密度系列支持该功能。

Note

只有 NuMicro™ NUC100 系列的低密度系列支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

参数

无

头文件

Driver/DrvGPIO.h

示例

```
/* Configure GPA-1 to output mode */
DrvGPIO_Open (E_GPA, 1, E_IO_OUTPUT);
/* Set GPA-1 to high */
GPA_1 = 1;
/* ..... */
/* Configure GPB-3 to input mode */
uint8_t u8PinValue;
DrvGPIO_Open (E_GPB, 3, E_IO_INPUT);
/* Get GPB-3 pin value */
u8PinValue = GPB_3;
```

5.5. 函数

DrvGPIO_Open

原型

```
int32_t DrvGPIO_Open(
    E_DRVGPIO_PORT    port,
    int32_t           i32Bit,
    E_DRVGPIO_IO      mode,
);
```

描述

配置指定的 GPIO 端口为指定的操作模式。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

mode [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

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

```
/* Configure GPA-0 to GPIO output mode and GPA-1 to GPIO input mode*/
DrvGPIO_Open (E_GPA, 0, E_IO_OUTPUT);
DrvGPIO_Open (E_GPA, 1, E_IO_INPUT);
```

DrvGPIO_Close

原型

int32_t DrvGPIO_Close(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

```
/* Close GPA-0 function and set to default quasi-bidirectional mode */
DrvGPIO_Close (E_GPA, 0);
```

DrvGPIO_SetBit

原型

int32_t DrvGPIO_SetBit(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

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

DrvGPIO_GetBit

原型

int32_t DrvGPIO_GetBit(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

指定的输入引脚的值：0/1

E_DRVGPIO_ARGUMENT：参数错误

示例

```
int32_t i32BitValue;
/* Configure GPA-1 as GPIO input mode*/
DrvGPIO_Open (E_GPA, 1, E_IO_INPUT);
i32BitValue = DrvGPIO_GetBit (E_GPA, 1);
```

```
if (u32BitValue == 1)
{
   printf("GPA-1 pin status is high.\n");
}
else
{
   printf("GPA-1 pin status is low.\n");
}
```

DrvGPIO_ClrBit

原型

int32_t DrvGPIO_ClrBit(E_DRVGPIO_PORT port,int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

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

DrvGPIO_SetPortBits

原型

int32_t DrvGPIO_SetPortBits(E_DRVGPIO_PORT port, int32_t i32Data);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Data [in]

输出数据值，可以是 0~0xFFFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

```
/* Set the output value of GPA port to 0x1234 */
DrvGPIO_SetPortBits (E_GPA, 0x1234);
```

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

头文件

Driver/DrvGPIO.h

返回值

指定的输入端口的值：0 ~ 0xFFFF

E_DRVGPIO_ARGUMENT：参数错误

示例

```
/* Get the GPA port input data value */
int32_t i32PortValue;
i32PortValue = DrvGPIO_GetPortBits (E_GPA);
```

DrvGPIO_GetDoutBit

原型

int32_t DrvGPIO_GetDoutBit(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器某一位的值：0/1

E_DRVGPIO_ARGUMENT：参数错误

示例

```
/* Get the GPA-1 data output value */
int32_t i32BitValue;
i32BitValue = DrvGPIO_GetDoutBit (E_GPA, 1);
```

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0 ~ 0xFFFF

E_DRVGPIO_ARGUMENT：参数错误

示例

```
/* Get the GPA port data output value */
int32_t i32PortValue;
i32PortValue = DrvGPIO_GetPortDoutBits (E_GPA);
```

DrvGPIO_SetBitMask

原型

int32_t DrvGPIO_SetBitMask(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Protect GPA-0 write data function */
DrvGPIO_SetBitMask (E_GPA, 0);
```

DrvGPIO_GetBitMask

原型

int32_t DrvGPIO_GetBitMask(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

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 GPA Data Output Write Mask Resister */
int32_t i32MaskValue;
i32MaskValue = DrvGPIO_GetBittMask (E_GPA, 0);
/* If (i32MaskValue = 1), its meaning GPA-0 is write protected */
```

DrvGPIO_ClrBitMask

原型

int32_t DrvGPIO_ClrBitMask(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0~15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Remove the GPA-0 write protect function */
DrvGPIO_ClrBitMask (E_GPA, 0);
```

DrvGPIO_SetPortMask

原型

int32_t DrvGPIO_SetPortMask(E_DRVGPIO_PORT port, int32_t i32MaskData);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32MaskData [in]

指定 GPIO 端口的引脚。可以是 0~0xFFFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS	操作成功
E_DRVGPIO_ARGUMENT	参数错误

示例

```
/* Protect GPA-0/4 write data function */
DrvGPIO_SetPortMask (E_GPA, 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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0~0xFFFF。

示例

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

DrvGPIO_ClrPortMask

原型

int32_t DrvGPIO_ClrPortMask(E_DRVGPIO_PORT port, int32_t i32MaskData);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0 ~ 0xFFFF。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Remove the GPA-0/4 write protect function */
DrvGPIO_ClrPortMask (E_GPA, 0x11);
```

DrvGPIO_EnableDebounce

原型

int32_t DrvGPIO_EnableDebounce(E_DRVGPIO_PORT port, int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0 ~ 15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Enable GPA-0 interrupt de-bounce function */
DrvGPIO_EnableDebounce (E_GPA, 0);
```

DrvGPIO_DisableDebounce

原型

int32_t DrvGPIO_DisableDebounce(E_DRVGPIO_PORT port,int32_t i32Bit);

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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0 ~ 15。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Disable GPA-0 interrupt de-bounce function */
DrvGPIO_DisableDebounce (E_GPA, 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

i32CycleSelection [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：参数错误

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 i32CycleSelect ion;
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,
    int32_t i32Bit,
    E_DRVGPIO_INT_TYPE TriggerType,
    E_DRVGPIO_INT_MODE Mode
);
```

描述

使能指定 GPIO 引脚的中断功能。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0 ~ 15。但是 GPB.14/15 只用于外部中断 0/1。

TriggerType [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：参数错误

示例

```
/* Enable GPB-13 interrupt function and its rising and edge trigger. */
DrvGPIO_EnableInt (E_GPB, 13, E_IO_RISING, E_MODE_EDGE);
```

DrvGPIO_DisableInt

原型

int32_t DrvGPIO_DisableInt(E_DRVGPIO_PORT port, int32_t i32Bit);

描述

禁止 GPIO 引脚的中断功能。GPB.14 和 GPB.15 除外。

参数

port [in]

E_DRVGPIO_PORT，指定 GPIO 口。可以是 E_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

i32Bit [in]

指定 GPIO 端口的引脚。可以是 0 ~ 15。但是 GPB.14/15 只用于外部中断 0/1。

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功

示例

```
/* Disable GPB-13 interrupt function. */
DrvGPIO_DisableInt (E_GPB, 13);
```

DrvGPIO_SetIntCallback

原型

```
void DrvGPIO_SetIntCallback (
  GPIO_GPAB_CALLBACK pfGPABCallback,
  GPIO_GPCDE_CALLBACK pfGPCDECallback
)
```

描述

为 GPA/GPB 端口和 GPC/GPD/GPE 端口安装中断回调函数，除了 GPB.14 和 GPB.15 引脚。

参数

pfGPABCallback [in]

GPA/GPB 回调函数的函数指针。

pfGPCDECallback [in]

GPC/GPD/GPE 回调函数的函数指针。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Set GPA/B and GPC/D/E interrupt callback functions */
DrvGPIO_SetIntCallback (GPABCallback, GPCDECallback);
```

DrvGPIO_EnableEINT0

原型

```
void DrvGPIO_EnableEINT0 (
  E_DRVGPIO_INT_TYPE TriggerType,
  E_DRVGPIO_INT_MODE Mode,
  GPIO_EINT0_CALLBACK pfEINT0Callback
)
```

描述

使能/INT0(GPB.14)引脚外部 GPIO 中断功能。

参数

TriggerType [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。

pfEINT0Callback [in]

外部 INT0 回调函数的函数指针。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Enable external INT0 interrupt as falling and both-edge trigger. */
DrvGPIO_EnableEINT0 (E_IO_BOTH_EDGE, E_MODE_EDGE, EINT1Callback);
```

DrvGPIO_DisableEINT0

原型

```
void DrvGPIO_DisableEINT0 (void)
```

描述

禁止/INT0(GPB.14)引脚外部 GPIO 中断功能。

参数

无

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Disable external INT0 interrupt function. */
DrvGPIO_DisableEINT0 ();
```

DrvGPIO_EnableEINT1

原型

```
void DrvGPIO_EnableEINT1 (
    E_DRVGPIO_INT_TYPE TriggerType,
    E_DRVGPIO_INT_MODE Mode,
    GPIO_EINT1_CALLBACK pfEINT1Callback
)
```

描述

使能/INT1(GPB.15)引脚外部 GPIO 中断功能。

参数

TriggerType [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。

pfEINT1Callback [in]

外部 INT1 回调函数的函数指针。

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Enable external INT1 interrupt as low level trigger. */
DrvGPIO_EnableEINT1 (E_IO_FALLING, E_MODE_LEVEL, EINT1Callback);
```

DrvGPIO_DisableEINT1

原型

void DrvGPIO_DisableEINT01(void)

描述

禁止/INT1(GPB.15)引脚外部 GPIO 中断功能。

参数

无

头文件

Driver/DrvGPIO.h

返回值

无

示例

```
/* Disable external INT1 interrupt function. */
DrvGPIO_DisableEINT1 ();
```

DrvGPIO_GetIntStatus

原型

uint32_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_GPA，E_GPB，E_GPC，E_GPD 和 E_GPE。

头文件

Driver/DrvGPIO.h

返回值

指定寄存器的端口值：0 ~ 0xFFFF。

示例

```
/* Get GPA interrupt status. */
int32_t i32INTStatus;
i32INTStatus = DrvGPIO_GetIntStatus (E_GPA);
```

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。

Note

并不是所有的芯片都支持这些功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指南。

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_I2C0 / E_FUNC_I2C1，

E_FUNC_I2S，

E_FUNC_CAN0，

E_FUNC_ACMP0 / E_FUNC_ACMP1，

E_FUNC_SPI0 / E_FUNC_SPI1 / E_FUNC_SPI2 / E_FUNC_SPI3，

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_UART2，

E_FUNC_PWM01 / E_FUNC_PWM23 / E_FUNC_PWM45 / E_FUNC_PWM67，

E_FUNC_EBI_8B / E_FUNC_EBI_16B，

E_FUNC_SPI0_QFN36PIN

头文件

Driver/DrvGPIO.h

返回值

E_SUCCESS：操作成功.

E_DRVGPIO_ARGUMENT：参数错误

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

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

NuMicro™ NUC100 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 3 种操作模式：单次，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/PB.8 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
- 三种操作模式
  - 单次模式
  - 单周期扫描模式
  - 连续扫描模式
- A/D 转换可以由下列动作启动
  - 软件写 1 到 ADST 位
  - 外部引脚 STADC
- 转换结果可以跟指定的值比较，如果转换结果与比较寄存器中的值匹配，用户可以选择是否产生一个中断
- 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 3 个输入源:：外部模拟电压, 内部固定带隙电压和内部温度传感输出
- 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_SINGLE_CYCLE_OP	1	单周期扫描模式
ADC_CONTINUOUS_OP	2	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 I
INTERNAL_BANDGAP	1	内部 bandgap 电压
INTERNAL_TEMPERATURE_SENSOR	2	内部温度传感器

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

E_ADC_DIFF_MODE_OUTPUT_FORMAT

枚举标识符	值	描述

UNSIGNED_OUTPUT	0	无符号格式
TWOS_COMPLEMENT	1	二进制补码格式

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 比较起 0 中断标志状态。

头文件

Driver/DrvADC.h

返回值

0：没有 ADC 比较器 0 中断发生。

1：发生了 ADC 比较器 0 中断。

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 比较起 1 中断标志状态。

头文件

Driver/DrvADC.h

返回值

0：没有 ADC 比较器 1 中断发生。

1：发生了 ADC 比较器 1 中断。

示例

```
/* Get the status of ADC comparator 0 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，单周期扫描或者连续扫描模式。

ADC_SINGLE_OP：单次模式

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。

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,
  E_ADC_INPUT_MODE InputMode
);
```

描述

选择 ADC 输入通道。

参数

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。

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。

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

头文件

Driver/DrvADC.h

返回值

无

示例

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

DrvADC_ConfigADCChannel7

原型

void DrvADC_ConfigADCChannel7 (E_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

INTERNAL_TEMPERATURE_SENSOR：内部温度传感器

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_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_SetADCOperat ionMode(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。

- 单次模式下，指定通道 A/D 转换完成。
- 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 中断的回调函数指针。

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 中断回调函数指针。

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_DisableADCCmp0Int (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_ADCMP0_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 中断回调函数指针。

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(PB8)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：转换完成

FALSE：正在转换

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

uint32_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 */
printf("Conversion result of channel 3: %d\n", DrvADC_GetConversionData(3));
```

DrvADC_EnablePDMA

原型

void DrvADC_EnablePDMA (void);

描述

使能 PDMA 传输。用户可以通过 PDMA 传输 A/D 转换结果到用户指定的内存空间，而不需要 CPU 的干预。单次模式下，只有被选择通道的转换结果会被传输；单周期扫描模式或连续扫描模式下，所有使能的通道的转换结果都会被通过 PDMA 传输。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Enable PDMA transfer */
DrvADC_EnablePDMA();
```

DrvADC_DisablePDMA

原型

void DrvADC_DisablePDMA (void);

描述

禁止 PDMA 传输。

参数

无

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* Disable PDMA transfer */
DrvADC_DisablePDMA();
```

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 A/D 通道号。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 A/D 通道号。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

```
uint32_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 0~0xFFF。

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

无

示例

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

DrvADC_EnableADCCmp1

原型

```
uint32_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 0~0xFFF。

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 1。

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_EnableSelfCalibrat ion (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_EnableSelfCalibrat ion();
```

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_IsCalibrat ionDone() )
   printf("Self calibrat ion 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 calibrat ion function */
DrvADC_DisableSelfCalibration();
```

DrvADC_DiffModeOutputFormat

原型

```
void DrvADC_DiffModeOutputFormat (
    E_ADC_DIFF_MODE_OUTPUT_FORMAT OutputFormat
);
```

描述

选择差分输入模式下的输出格式。只有 NUC101 和 NuMicro™ NUC100 系列产品的低密度版本支持该功能。请参考附录 NuMicro™ NUC100 系列产品选型指导。

参数

OutputFormat [in]

指定输出格式。可以是无符号格式(UNSIGNED_OUTPUT)或者二进制补码格式(TWOS_COMPLEMENT)。

头文件

Driver/DrvADC.h

返回值

无

示例

```
/* 2'"s complement format */
DrvADC_DiffModeOutputFormat(TWOS_COMPLEMENT);
```

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 NuMicro™ NUC100 系列有 4 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。每个 SPI 控制器可以驱动 2 个外设。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，写 1 可以清除中断标志。从设备的选中信号激活电平可以是低/高（SSR[SS_LVL]位），具体设定依靠连接的外设。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。

SPI 控制器也支持 2 比特数据模式，在寄存器 SPI_CNTL[22]中定义。当 TWOB 位使能，SPI 控制器可以通过发送/接收缓冲区来发送/接收 2 比特串行数据。第一个比特从寄存器 SPI_TX0 发送，同时接收第一个比特到寄存器 SPI_RX0 中；第二个比特从寄存器 SPI_TX1 发送，并且接收第二个比特到寄存器 SPI_RX1 中。

本文档介绍 SPI 驱动的用法。

7.2. SPI 特性

- 四组 SPI 控制器
- 支持主/从模式
- 支持 1, 2 比特串行数据 IN/OUT
- 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
- 支持大端/小端优先数据传输
- 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. 类型定义

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
eDRVSPI_PORT2	2	SPI 端口 2
eDRVSPI_PORT3	3	SPI 端口 3

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

E_DRVSPI_SLAVE_SEL

枚举标识符	值	描述
eDRVSPI_NONE	0	没有从设备被选择
eDRVSPI_SS0	1	选择第一个从设备选择引脚
eDRVSPI_SS1	2	选择第二个从设备选择引脚
eDRVSPI_SS0_SS1	3	两个选择引脚都被选择

E_DRVSPI_DMA_MODE

枚举标识符	值	描述
eDRVSPI_TX_DMA	0	使能 TX DMA
eDRVSPI_RX_DMA	1	使能 RX DMA

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,
uint8_t bQFN36PinPackage
);

描述

这个函数用来打开 SPI 功能。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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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。

bQFN36PinPackage [in]

在不同的封装类型中，SPI 复用不同的 I/O 引脚。该参数指定封装类型。

TRUE：QFN 36 脚封装

FALSE：其他封装类型

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：SPI 端口正忙

示例

```
/* Configure SPI0 as a master, 32-bit transaction, not QFN 36-pin package */
DrvSPI_Open(eDRVSPI_PORT0, eDRVSPI_MASTER, eDRVSPI_TYPE1, 32, FALSE);
```

DrvSPI_Close

原型

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

描述

关闭指定的 SPI 功能并且禁止 SPI 中断。

参数

eSpiPort [in]

指定 SPI 端口。

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

头文件

Driver/DrvSPI.h

返回值

无

示例

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

DrvSPI_Set2BitTransferMode

原型

```
void DrvSPI_Set2BitTransferMode (
    E_DRVSPI_PORT eSpiPort,
    uint8_t Enable
);
```

描述

设置 2 比特传输模式。

参数

eSpiPort [in]

指定 SPI 端口。

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

Enable [in]

使能(TRUE)/禁止(FALSE)

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable 2-bit transfer mode of SPI0 */
DrvSPI_Set2BitTransferMode(eDRVSPI_PORT0, TRUE);
```

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

i32BitLength [in]

指定位宽(1 ~ 32 比特)

头文件

Driver/DrvSPI.h

返回值

E_SUCCESS：成功

E_DRVSPI_ERR_BIT_LENGTH：位宽超过范围

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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 t rigger */
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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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接传输的位宽与设定不符

示例

```
/* Level t rigger */
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,
   E_DRVSPI_SLAVE_SEL eSlaveSel
);
```

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

eSlaveSel [in]

选择将要使用的从选择引脚。

eDRVSPI_NONE：没有从引脚被选择

eDRVSPI_SS0：选择 SS0

eDRVSPI_SS1：选择 SS1

eDRVSPI_SS0_SS1：同时选择 SS0 和 SS1

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable the automatic slave select function of SS0. */
DrvSPI_EnableAutoSS(eDRVSPI_PORT0, eDRVSPI_SS0);
```

DrvSPI_DisableAutoCS

原型

```
void DrvSPI_DisableAutoCS (
    E_DRVSPI_PORT eSpiPort
);
```

描述

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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,
    E_DRVSPI_SLAVE_SEL eSlaveSel
);
```

描述

配置从选择引脚。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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

eSlaveSel [in]

自动从选择模式下，使用这个参数来选择将被使用的从选择引脚。

手动从选择模式下，指定的从选择引脚将被激活。可以是 eDRVSPI_NONE，eDRVSPI_SS0，eDRVSPI_SS1 或者 eDRVSPI_SS0_SS1。

eDRVSPI_NONE：没有从引脚被选择

eDRVSPI_SS0：选择 SS0

eDRVSPI_SS1：选择 SS1

eDRVSPI_SS0_SS1：同时选择 SS0 和 SS1

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 SS0 pin to active state */
DrvSPI_SetSS(eDRVSPI_PORT0, eDRVSPI_SS0);
```

DrvSPI_ClrSS

原型

```
void DrvSPI_ClrSS(
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_SLAVE_SEL eSlaveSel
);
```

描述

设置指定的从选择引脚为非激活状态。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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

eSlaveSel [in]

指定从选择引脚。

eDRVSPI_SS0，eDRVSPI_SS1 或者 eDRVSPI_SS0_SS1。

eDRVSPI_NONE：没有从引脚被选择

eDRVSPI_SS0：选择 SS0

eDRVSPI_SS1：选择 SS1

eDRVSPI_SS0_SS1：同时选择 SS0 和 SS1

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 SS0 pin to inactive state */
DrvSPI_ClrSS(eDRVSPI_PORT0, eDRVSPI_SS0);
```

DrvSPI_IsBusy

原型

```
uint8_t DrvSPI_IsBusy(
    E_DRVSPI_PORT eSpiPort
);
```

描述

检查 SPI 端口是否正忙。

参数

eSpiPort [in]

指定 SPI 端口。

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

```
uint32_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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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]

说明 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_SetClock1Freq(
    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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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_SetClock2Freq(
    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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

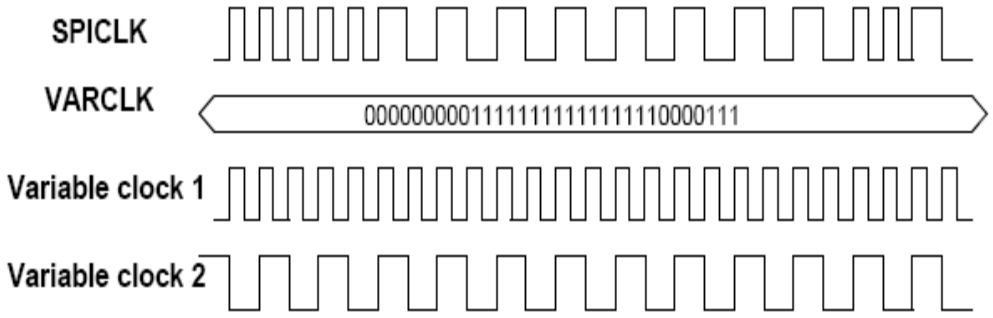
```
void
DrvSPI_SetVariableClockFunction (
   E_DRVSPI_PORT eSpiPort,
   uint8_t bEnable,
   uint32_t u32Pattern
);
```

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

bEnable [in]

使能(TRUE)/禁止(FALSE)

u32Pattern [in]

指定时钟模式。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

eSpiPort [in]

指定 SPI 端口。

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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 t ransfer. */
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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

写数据到发送寄存器。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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

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

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

头文件

Driver/DrvSPI.h

返回值

无

示例

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

DrvSPI_SetPMDA

原型

```
void DrvSPI_SetPDMA (
    E_DRVSPI_PORT eSpiPort,
    E_DRVSPI_DMA_MODE eDmaMode,
    uint8_t bEnable
);
```

描述

且配置 DMA.。

参数

eSpiPort [in]

指定 SPI 端口。

eDRVSPI_PORT0：SPI0

eDRVSPI_PORT1：SPI1

eDRVSPI_PORT2：SPI2

eDRVSPI_PORT3：SPI3

eDmaMode [in]

指定 DMA 模式。

eDRVSPI_TX_DMA：DMA 发送

eDRVSPI_RX_DMA：DMA 接收

eEnable [in]

True：使能 DMA.

False：禁止 DMA.

头文件

Driver/DrvSPI.h

返回值

无

示例

```
/* Enable the SPI0 DMA-Receiving function */
DrvSPI_SetPDMA(eDRVSPI_PORT0, eDRVSPI_RX_DMA, TRUE);
```

DrvSPI_GetVersion

原型

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

描述

获取 SPI 驱动版本号。

参数

无

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 NuMicroTMNUC100 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_PORT

枚举标识符	值	描述
I2C_PORT0	0	I2C 端口 0
I2C_PORT1	1	I2C 端口 1

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(E_I2C_PORT port, 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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 I2C0 and set I2C0 bus clock 100 KHz */
DrvI2C_Open (I2C_PORT0, 100000);
```

DrvI2C_Close

原型

int32_t DrvI2C_Close(E_I2C_PORT port);

描述

关闭 I2C 硬件。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

0 成功

示例

DrvI2C_Close (I2C_PORT0); /* Disable I2C0 */

DrvI2C_SetClockFreq

原型

int32_t DrvI2C_SetClockFreq (E_I2C_PORT port, 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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 I2C0 bus clock 200 KHz */

DrvI2C_SetClockFreq (I2C_PORT0, 200000);

DrvI2C_GetClockFreq

原型

uint32_t DrvI2C_GetClockFreq (E_I2C_PORT port,);

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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

I2C 总线时钟频率

示例

```
uint32_t u32clock;
u32clock = DrvI2C_GetClockFreq (I2C_PORT0); /* Get I2C0 bus clock */
```

DrvI2C_SetAddress

原型

```
int32_t  DrvI2C_SetAddress (
   E_I2C_PORT port,
   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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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(I2C_PORT0, 0, 0x15, 0); /* Set I2C0 1st slave address 0x15 */
DrvI2C_SetAddress(I2C_PORT0, 1, 0x35, 0); /* Set I2C0 2nd slave address 0x35 */
DrvI2C_SetAddress(I2C_PORT0, 2, 0x55, 0); /* Set I2C0 3rd slave address 0x55 */
DrvI2C_SetAddress(I2C_PORT0, 3, 0x75, 0); /* Set I2C0 4th slave address 0x75 */
```

DrvI2C_SetAddressMask

原型

```
int32_t DrvI2C_SetAddressMask  (
    E_I2C_PORT port,
    uint8_t slaveNo,
    uint8_t slaveAddrMask
);
```

描述

向指定的 I2C 从地址设定 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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

slaveNo [in]

选择从地址掩码。取值是 0 ~ 3

slaveAddrMask [in]

为选择的从地址设定 7 比特物理从地址，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 (I2C_PORT0, 0, 0x15, 0); /* Set I2C0 1st slave address 0x15 */
DrvI2C_SetAddress (I2C_PORT0, 1, 0x35, 0); /* Set I2C0 2nd slave address 0x35 */
/* Set I2C0 1st slave address mask 0x01, slave address 0x15 and 0x14 would be addressed */
DrvI2C_SetAddressMask (I2C_PORT0, 0, 0x01);
/* Set I2C0 2nd slave address mask 0x04, slave address 0x35 and 0x31 would be addressed
*/
DrvI2C_SetAddressMask (I2C_PORT0, 1, 0x04);
```

DrvI2C_GetStatus

原型

uint32_t DrvI2C_GetStatus (E_I2C_PORT port);

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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

I2C 状态码

示例

```
uint32_t u32status;
u32status = DrvI2C_GetStatus (I2C_PORT0); /* Get I2C0 current status code */
```

DrvI2C_WriteData

原型

void DrvI2C_WriteData(E_I2C_PORT port, uint8_t u8data);

描述

设定将要发送的 1 字节数据。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

u8data [in]

数据字节

头文件

Driver/DrvI2C.h

返回值

无

示例

DrvI2C_WriteData (I2C_PORT0, 0x55); /* Set byte data 0x55 into I2C0 data register */

DrvI2C_ReadData

原型

uint8_t DrvI2C_ReadData(E_I2C_PORT port);

描述

从 I2C 总线读上一个数据。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

上一个字节数据

示例

```
uint8_t u8data;
u8data = DrvI2C_ReadData (I2C_PORT0); /* Read out byte data from I2C0 data register */
```

DrvI2C_Ctrl

原型

void DrvI2C_Ctrl(E_I2C_PORT port, 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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 (I2C_PORT0, 0, 0, 1, 0); /* Set I2C0 SI bit to clear SI flag */
DrvI2C_Ctrl (I2C_PORT0, 1, 0, 0, 0); /* Set I2C0 STA bit to send START signal */
```

DrvI2C_GetIntFlag

原型

uint8_t DrvI2C_GetIntFlag(E_I2C_PORT port);

描述

获取 I2C 中断标志状态。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

中断状态 (1 或 0)

示例

```
uint8_t u8flagStatus;
u8flagStatus = DrvI2C_GetIntFlag (I2C_PORT0); /* Get the status of I2C0 interrupt flag */
```

DrvI2C_ClearIntFlag

原型

```
void  DrvI2C_ClearIntFlag(E_I2C_PORT port);
```

描述

如果 I2C 中断标志被置为 1，清除该标志。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

无

示例

```
DrvI2C_ClearIntFlag (I2C_PORT0); /* Clear I2C0 interrupt flag (SI) */
```

DrvI2C_EnableInt

原型

```
int32_t  DrvI2C_EnableInt(E_I2C_PORT port);
```

描述

使能 I2C 中断功能。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

0　　成功

示例

DrvI2C_EnableInt (I2C_PORT0); /* Enable I2C0 interrupt */

DrvI2C_DisableInt

原型

int32_t DrvI2C_DisableInt (E_I2C_PORT port);

描述

关闭 I2C 中断和相应的 NVIC 比特.

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

0 成功

示例

DrvI2C_DisableInt (I2C_PORT0); /* Disable I2C0 interrupt */

DrvI2C_InstallCallBack

原型

```
int32_t DrvI2C_InstallCallBack (
  E_I2C_PORT port,
  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

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 I2C0 ca ll back function „I2C0_ Ca llbac k_Normal‟ for I2C normal condition */
DrvI2C_InstallCallback (I2C_PORT0, I2CFUNC, I2C0_Callback_Normal);
/* Install I2C0 ca ll back function „I2C0_ Ca llbac k_BusErr‟ for Bus Error condition */
DrvI2C_InstallCallback (I2C_PORT0, BUSERROR, I2C0_Callback_BusErr);
```

DrvI2C_UninstallCallBack

原型

int32_t DrvI2C_UninstallCallBack(E_I2C_PORT port, E_I2C_CALLBACK_TYPE Type);

描述

卸载 I2C 中断处理函数的回调函数。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 I2C0 call back function for I2C normal condition */
DrvI2C_UninstallCallBack (I2C_PORT0, I2CFUNC);
/* Uninstall I2C0 call back function for Bus Error condition */
DrvI2C_UninstallCallBack (I2C_PORT0, BUSERROR);
```

DrvI2C_SetTimeoutCounter

原型

```
int32_t  DrvI2C_SetTimeoutCounter (
    E_I2C_PORT port,
    int32_t i32enable,
    uint8_t u8div4
);
```

描述

配置 14 比特超时计数器。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

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 I2C0 14-bit t imeout counter and disable its DIV4 function */
DrvI2C_EnableTimeoutCount (I2C_PORT0, 1, 0);
```

DrvI2C_ClearTimeoutFlag

原型

void DrvI2C_ClearTimeoutFlag(E_I2C_PORT port);

描述

如果 I2C 超时中断标志 TIF 被置为 1，清除该标志。

参数

port [in]

指定 I2C 端口。(I2C_PORT0 / I2C_PORT1)

头文件

Driver/DrvI2C.h

返回值

无

示例

DrvI2C_ClearTimeoutFlag (I2C_PORT0); /* Clear I2C0 TIF flag */

DrvI2C_GetVersion

原型

uint32_t DrvI2C_GetVersion (void);

描述

获取该模块版本号。

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. RTC 驱动

9.1. RTC 介绍

实时时钟(RTC)单元提供给用户实时时间和日历信息。RTC 使用频率为 32.768KHz 的外部晶振。内嵌 RTC 被设计用于产生周期性的中断信号，中断周期可以是 0.25/ 0.5/ 1/ 2/ 4/ 8 秒。有一个 RTC 溢出计数器，可以由软件调节。

9.2. RTC 特性

- 有一个时钟计数器，用户可以用来查看时间
- 支持周期性时间 tick 中断，有 8 个中断周期可供选择，1/128, 1/64, 1/32, 1/16, 1/8, 1/4, 1/2 和 1 秒
- 支持 RTC 时间 Tick 和警报匹配中断
- 支持唤醒功能，可以把 CPU 从 sleep 或 power-down 模式唤醒

9.3. 常量定义

常量名	值	描述
DRVRTC_INIT_KEY	0xa5eb1357	用于复位所有 RTC 逻辑的关键数值
DRVRTC_WRITE_KEY	0xA965	用于解锁被保护寄存器的关键数值
DRVRTC_CLOCK_12	0	12-Hour 模式
DRVRTC_CLOCK_24	1	24-Hour 模式
DRVRTC_AM	1	a.m.
DRVRTC_PM	2	p.m.
DRVRTC_YEAR2000	2000	设置年为 2000
DRVRTC_FCR_REFERENCE	32761	用于补偿 32MHz 的参考值

9.4. 类型定义

E_DRVRTC_INT_SOURCE

枚举标识符	值	描述
DRVRTC_ALARM_INT	1	设置 alarm 中断
DRVRTC_TICK_INT	2	设置 tick 中断
DRVRTC_ALL_INT	3	设置 alarm 和 tick 中断

E_DRVRTC_TICK

枚举标识符	值	描述
DRVRTC_TICK_1_SEC	0	设定 tick 周期为 1 tick 每秒
DRVRTC_TICK_1_2_SEC	1	设定 tick 周期为 2 tick 每秒
DRVRTC_TICK_1_4_SEC	2	设定 tick 周期为 4 tick 每秒
DRVRTC_TICK_1_8_SEC	3	设定 tick 周期为 8 tick 每秒
DRVRTC_TICK_1_16_SEC	4	设定 tick 周期为 16 tick 每秒
DRVRTC_TICK_1_32_SEC	5	设定 tick 周期为 32 tick 每秒
DRVRTC_TICK_1_64_SEC	6	设定 tick 周期为 64 tick 每秒
DRVRTC_TICK_1_128_SEC	7	设定 tick 周期为 128 tick 每秒

E_DRVRTC_TIME_SELECT

枚举标识符	值	描述
DRVRTC_CURRENT_TIME	0	选择当前时间选项
DRVRTC_ALARM_TIME	1	选择报警时间选项

E_DRVRTC_DWR_PARAMETER

枚举标识符	值	描述

DRVRTC_SUNDAY	0	周日
DRVRTC_MONDAY	1	周一
DRVRTC_TUESDAY	2	周二
DRVRTC_WEDNESDAY	3	周三
DRVRTC_THURSDAY	4	周四
DRVRTC_FRIDAY	5	周五
DRVRTC_SATURDAY	6	周六

9.5. 函数

DrvRTC_SetFrequencyCompensation

原型

int32_t

DrvRTC_SetFrequencyCompensation (

int32_t i32FrequencyX100;

);

描述

设定频率补偿数据。

参数

i32FrequencyX100 [in]

指定 RTC 时钟 X100，例如 3277365 表示 32773.65

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_FCR_VALUE：补偿值错误

示例

```
/* If the measured RTC crystal frequency is 32773.65Hz.*/
DrvRTC_SetFrequencyCompensation (3277365) ;
```

DrvRTC_IsLeapYear

原型

int32_t

DrvRTC_IsLeapYear (void);

描述

根据当前时间，返回今年是不是闰年。

参数

无

头文件

Driver/DrvRTC.h

返回值

1：今年是闰年

0：今年不是闰年

示例

```
If(DrvRTC_IsLeapYear(void))
   printf("This is Leap year!");
else
   printf("This is not Leap year!");
```

DrvRTC_GetIntTick

原型

int32_t DrvRTC_GetIntTick (void);

描述

用户可以使用 DrvRTC_SetTickMode 来设置 tick 周期，该函数用于获取使能 tick 中断后当前中断 tick 数量。

参数

无

头文件

Driver/DrvRTC.h

返回值

中断 tick 数量

示例

```
/* Polling the tick count to wait 3 sec.*/
DrvRTC_SetTickMode(DRVRTC_TICK_1_2_SEC) ; /* 1 tick is 0.5 sec.*/
```

```
DrvRTC_EnableInt(DRVRTC_TICK_INT,NULL);
while(DrvRTC_GetTick(void)<6);
   printf("Pass though 3 sec\n");
```

DrvRTC_ResetIntTick

原型

void DrvRTC_ResetTick (void);

描述

该函数用于复位中断中正在计数的 tick 计数值。

参数

无

头文件

Driver/DrvRTC.h

返回值

无

示例

DrvRTC_ResetTick (void) ;

DrvRTC_WriteEnable

原型

int32_t

DrvRTC_WriteEnable (void);

描述

写密码到 AER 寄存器来允许对其他寄存器的访问。

参数

无

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_FAILED：失败

Note

写密码到 AER 寄存器之后，可以写 TLR / CLR /DWR / TAR /CAR 寄存器。

示例

```
/* Before you want to set the value in TLR / CLR /DWR / TAR /CAR register, using the
function to open access account. */
DrvRTC_WriteEnable (void) ;
```

DrvRTC_Init

原型

int32_t DrvRTC_Init (void);

描述

初始化 RTC。包括清除回调函数指针，使能 32K 时钟和 RTC 时钟，并写初始化关键字使 RTC 开始计数。

参数

无

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_EIO：初始化 RTC 失败

示例

```
/*In the beginning, call the function to init ial RTC */
DrvRTC_Init(void) ;
```

DrvRTC_SetTickMode

原型

int32_t DrvRTC_SetTickMode(uint8_t ucMode);

描述

该函数用于为周期性时间 tick 中断设定 tick 周期。

参数

ucMode [in]

DRVRTC_TICK 结构体，用于为周期性时间 tick 中断请求设定 RTC tick 周期。包括

DRVRTC_TICK_1_SEC：每个 tick 是 1 秒

DRVRTC_TICK_1_2_SEC：每个 tick 是 1/2 秒

DRVRTC_TICK_1_ 4_SEC：每个 tick 是 1/4 秒

DRVRTC_TICK_1_ 8_SEC：每个 tick 是 1/8 秒

DRVRTC_TICK_1_ 16_SEC：每个 tick 是 1/16 秒

DRVRTC_TICK_1_ 32_SEC：每个 tick 是 1/32 秒

DRVRTC_TICK_1_ 64_SEC：每个 tick 是 1/64 秒

DRVRTC_TICK_1_ 128_SEC：每个 tick 是 1/128 秒

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_EIO：初始化 RTC 失败

E_DRVRTC_ERR_ENOTTY：参数错误

示例

```
/* Set Tick interrupt is 128 tick/sec */
DrvRTC_SetTickMode (DRVRTC_TICK_1_128_SEC) ;
```

DrvRTC_EnableInt

原型

```
int32_t DrvRTC_EnableInt (
   DRVRTC_INT_SOURCE str_IntSrc,
   PFN_DRVRTC_CALLBACK pfncallback
);
```

描述

该函数用于使能指定中断，并安装回调函数。

参数

`str_IntSrc [in]`

中断源结构体。包括：

DRVRTC_ALARM_INT：Alarm 中断

DRVRTC_TICK_INT：Tick 中断

DRVRTC_ALL_INT：Alarm 和 tick 中断

pfncallback [in]

回调函数指针

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_ENOTTY：参数错误

示例

```
/* Enable tick interrupt and install callback function “RTC_TickCallBackfn”.*/
DrvRTC_EnableInt(DRVRTC_TICK_INT,RTC_TickCallBackfn);
```

DrvRTC_DisableInt

原型

```
int32_t DrvRTC_DisableInt (
  DRVRTC_INT_SOURCE str_IntSrc,
);
```

描述

该函数用于禁止指定中断，并清除回调函数指针。

参数

str_IntSrc [in]

中断源结构体。包括：

DRVRTC_ALARM_INT：Alarm 中断

DRVRTC_TICK_INT：Tick 中断

DRVRTC_ALL_INT：Alarm 和 tick 中断

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_ENOTTY：参数错误

示例

```
/* Disable tick and alarm interrupt*/
DrvRTC_DisableInt(DRVRTC_ALL_INT);
```

DrvRTC_Open

原型

```
int32_t
DrvRTC_Open (
   S_DRVRTC_TIME_DATA_T *sPt
);
```

描述

设定当前时间(年/月/日, 时/分/秒和星期)。

参数

***sPt [in]**

指定时间属性和当前时间。包括：

u8cClockDisplay： DRVRTC_CLOCK_12 / DRVRTC_CLOCK_24

u8cAmPm： DRVRTC_AM / DRVRTC_PM

u32cSecond ： 秒

u32cMinute： 分

u32cHour： 小时

u32cDayOfWeek： 星期

u32cDay： 日

u32cMonth： 月

u32Year： 年

u8IsEnableWakeUp： 是否使能时间报警发生时的唤醒功能

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_EIO：初始化 RTC 失败

示例

```
/* Start RTC count from 2009.Jan.19, 13:20:00 . */
S_DRVRTC_TIME_DATA_T sInitTime;
```

```
sInitTime.u32Year = 2009;
sInitTime.u32cMonth = 1;
sInitTime.u32cDay = 19;
sInitTime.u32cHour = 13;
sInitTime.u32cMinute = 20;
sInitTime.u32cSecond = 0;
sInitTime.u32cDayOfWeek = DRVRTC_MONDAY;
sInitTime.u8cClockDisplay = DRVRTC_CLOCK_24;
if(DrvRTC_Open(&sInitTime) !=E_SUCCESS){
   printf("RTC Open Fail!!\n");}
```

DrvRTC_Read

原型

```
int32_t
DrvRTC_Read (
   E_DRVRTC_TIME_SELECT eTime,
   S_DRVRTC_TIME_DATA_T *sPt
);
```

描述

从 RTC 读取当前日期/时间或者报警日期/时间。

参数

eTime [in]

指定读取当前时间还是警报时间。

DRVRTC_CURRENT_TIME：当前时间

DRVRTC_ALARM_TIME：警报时间

***sPt [in]**

指定缓存来存储从 RTC 读取的数据。包括：

u8cClockDisplay：	DRVRTC_CLOCK_12 / DRVRTC_CLOCK_24
u8cAmPm：	DRVRTC_AM / DRVRTC_PM
u32cSecond ：	秒
u32cMinute：	分
u32cHour：	小时
u32cDayOfWeek：	星期
u32cDay：	日

u32cMonth： 月

u32Year： 年

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_EIO：初始化 RTC 失败

示例

```
/* Condition: You want to get current RTC calendar and time */
S_DRVRTC_TIME_DATA_T sCurTime;
DrvRTC_Read(DRVRTC_CURRENT_TIME, &sCurTime);
printf("Current Time:%d/%02d/%02d %02d:%02d:%02d\n",
  sCurTime.u32Year,sCurTime.u32cMonth,sCurTime.u32cDay,sCurTime.u32cHour,sCurTi
  me.u32cMinute,sCurTime.u32cSecond);
```

DrvRTC_Write

原型

```
int32_t
DrvRTC_Write (
  E_DRVRTC_TIME_SELECT eTime,
  S_DRVRTC_TIME_DATA_T *sPt
);
```

描述

给 RTC 设置当前日期/时间或者报警日期/时间。

参数

eTime [in]

指定写当前时间还是警报时间。

DRVRTC_CURRENT_TIME：当前时间

DRVRTC_ALARM_TIME：警报时间

***sPt [in]**

指定往 RTC 写的数据。包括：

u8cClockDisplay： DRVRTC_CLOCK_12 / DRVRTC_CLOCK_24

u8cAmPm：	DRVRTC_AM / DRVRTC_PM
u32cSecond ：	秒
u32cMinute：	分
u32cHour：	小时
u32cDayOfWeek：	星期
u32cDay：	日
u32cMonth：	月
u32Year：	年

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

E_DRVRTC_ERR_EIO：初始化 RTC 失败

示例

```
/* Condition: Update current the second of time to zero */
S_DRVRTC_TIME_DATA_T sCurTime;
DrvRTC_Read(DRVRTC_ALARM_TIME,&sCurTime);
sCurTime.u32cSecond = 0;
DrvRTC_Write(DRVRTC_ALARM_TIME,&sCurTime);
```

DrvRTC_Close

原型

int32_t

DrvRTC_Close (void);

描述

禁止 RTC NVIC 通道和 tick，alarm 中断。

头文件

Driver/DrvRTC.h

返回值

E_SUCCESS：成功

示例

```
DrvRTC_Close(void);
```

DrvRTC_GetVersion

原型

int32_t

DrvRTC_GetVersion (void);

描述

获取当前驱动版本号。

头文件

Driver/DrvRTC.h

返回值

版本号：

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

10. CAN 驱动

10.1. CAN 介绍

控制器局域网(CAN)是一个串行通讯协议，支持多主设备并且可以有效的支持分布式实时控制，并且有很高的保密性。在 CAN 系统中，一个节点(Node)不使用系统配置的任何信息(station addresses)。不用请求其它节点软件/硬件任何改变，节点就可以加入 CAN 网络。只有 NuMicro™ 130/140 系列支持 CAN 功能。

10.2. CAN 特性

CAN 处理器包含以下特性：

- 与 CAN 2.0B 协议兼容
- 与 AMBA APB 总线接口兼容
- 多主设备节点
- 支持 11 比特标识符和 29 比特标识符
- 最高比特率可达 1Mbit/s
- NRZ 比特编码
- 错误侦测：比特错误, 填充错误, 格式错误, 15 比特 CRC 校验错误, 和应答错误
- 只侦听模式(没有应答, 不激活错误标志)
- 报文验收滤波扩展(4 字节标识符, 4 字节掩码)
- 每个 CAN 总线错误都有错误中断
- 扩展接收缓存(8 字节缓冲区)
- 唤醒功能

10.3. 常量定义

常量名	值	描述
STANDARD_FORMAT	0	设定帧格式为标准格式(11 比特标识符)
EXTENDED_FORMAT	1	设定帧格式为扩展格式(29 比特标识符)
DATA_TYPE	0	设定帧类型为 Data
REMOTE_TYPE	1	设定帧类型为 Remote
AMR_ALLPASS	0	验收屏蔽寄存器没有影响
AMR_ALLMASK	0xFFFFFFFF	使能验收屏蔽寄存器所有位

10.4. 类型定义

E_DRVCAN_INT_SOURCE

枚举标识符	值	描述
DRVCAN_INT_RI	0x1	接收中断
DRVCAN_INT_TI	0x2	发送中断
DRVCAN_INT_WUI	0x10	唤醒中断
DRVCAN_INT_ALI	0x40	仲裁丢失中断
DRVCAN_INT_BEI	0x80	总线错误中断

E_DRVCAN_ERRFLAG

枚举标识符	值	描述
DRVCAN_ERRSTUF	0	填充错误
DRVCAN_ERRFORM	1	格式错误
DRVCAN_ERRCRC	2	CRC 错误
DRVCAN_ERRACK	3	应答错误
DRVCAN_ERRBIT	4	比特错误

E_DRVCAN_CALLBACK_TYPE

枚举标识符	值	描述
TYPE_RI	0	接收中断回调函数
TYPE_TI	1	发送中断回调函数
TYPE_WUI	2	唤醒中断回调函数
TYPE_ALI	3	仲裁丢失中断回调函数
TYPE_BEI	4	总线错误中断回调函数

10.5. 函数

DrvCAN_Init

原型

void DrvCAN_Init (void);

描述

该函书用于初始化 CAN。包括复位 IP 和使能 CAN 时钟。

参数

无

头文件

Driver/DrvCAN.h

返回值

无

示例

```
/* In the beginning, user want to use CAN */
DrvCAN_Init ();
```

DrvCAN_Open

原型

int32_t DrvCAN_Open(int32_t i32CANKiloBitRate, int16_t i16SamplePointPos);

描述

该函数用于根据当前系统设定值来设定 CAN 比特率，它用于改变 CAN 分频值，并使用当前系统时钟来计算最接近的设定值。

参数

i32CANKiloBitRate [in]

需要的比特率。取值范围是 1 ~ 1000，单位是 kbps。

DRVCAN_PORT0 / DRVCAN_PORT1

i16SamplePointPos [in]

采样点百分比，取值应当在 0 ~ 100

头文件

Driver/DrvCAN.h

返回值

E_DRVCAN_ERR_BITRATE：错误的比特率设置

E_SUCEESS：成功

示例

```
/* Set CAN BUS bitrate: 1000kbit/s and sample point position is at 87.5% */
DrvCAN_Open(1000,875);
```

DrvCAN_InstallCallback

原型

```
void DrvCAN_InstallCallback (
  E_DRVCAN_CALLBACK_TYPE Type,
  PFN_DRVCAN_CALLBACK callbackfn
);
```

描述

该函书用于安装指定的回调函数。该回调函数将会在指定的中断服务程序中执行。

参数

Type [in]

回调中断类型

可以是 TYPE_RI /TYPE_TI/ TYPE_WUI/ TYPE_ALI/ TYPE_BEI

callbackfn [in]

回调函数指针

头文件

Driver/DrvCAN.h

返回值

无

示例

```
/* Install transmit interrupt callback function named “callbackfn” */
DrvCAN_InstallCallback(TYPE_TI, callbackfn);
```

DrvCAN_UnInstallCallback

原型

void DrvCAN_UnInstallCallback (E_DRVCAN_CALLBACK_TYPE Type);

描述

该函书用于卸载指定的回调函数。使用该函数移除安装的回调函数。

参数

Type [in]

回调中断类型

可以是 TYPE_RI /TYPE_TI/ TYPE_WUI/ TYPE_ALI/ TYPE_BEI

头文件

Driver/DrvCAN.h

返回值

无

示例

```
/* uninstall transmit interrupt callback function */
DrvCAN_UninstallCallback(TYPE_TI);
```

DrvCAN_EnableInt

原型

int32_t DrvCAN_EnableInt (uint32_t u32InterruptSrc);

描述

该函数用于使能 CAN 指定的中断。

参数

u32InterruptSrc [in]

中断源：可以是 DRVCAN_INT_BEI , DRVCAN_INT_ALI, DRVCAN_INT_WUI , DRVCAN_INT_TI 和 DRVCAN_INT_RI。用户可以通过“或”这些中断源来同时使能多个中断。如果用户在一个工程中调用该函数两次，设定值以第二次的设定为准。

头文件

Driver/DrvCAN.h

返回值

TRUE

示例

```
/* Enable specified CAN transmit done and receive done interrupt */
```

DrvCAN_EnbaleInt(INT_TI | INT_RI);

DrvCAN_DisableInt

原型

int32_t DrvCAN_DisableInt (uint32_t u32InterruptSrc);

描述

该函数用于禁止 CAN 中断并卸载中断回调函数。

参数

u32InterruptSrc [in]

中断源：可以是 DRVCAN_INT_BEI , DRVCAN_INT_ALI, DRVCAN_INT_WUI , DRVCAN_INT_TI 和 DRVCAN_INT_RI。用户可以通过“或”这些中断源来同时使能多个中断。如果用户在一个工程中调用该函数两次，设定值以第二次的设定为准。

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS

示例

```
/* Disable CAN transmit done and receive done interrupt */
DrvCAN_DisableInt(INT_TI | INT_RI);
```

DrvCAN_GetErrorStatus

原型

```
int32_t DrvCAN_GetErrorStatus (
    DRVCAN_ERRFLAG u32ErrorFlag
)
```

描述

该函数用于获取指定的 CAN 错误状态，看是否发生了错误。

参数

u32ErrorFlag [in]

错误标志：DRVCAN_ERRSTUFF/ DRVCAN_ERRFORM/DRVCAN_ERRCRC /DRVCAN_ERRACK/ DRVCAN_ERRBIT

头文件

Driver/DrvCAN.h

返回值

TRUE：指定的错误状态发生了

E_SUCCESS：指定的错误状态没有发生

示例

```
/* Get the bit error is happened or not */
DrvCAN_GetErrorStatus(DRVCAN_ERRBIT);
```

DrvCAN_Write

原型

int32_t DrvCAN_Write (STR_CAN_T *Msg);

描述

该函数设定 CAN 信息并发送到 CAN 总线。

参数

Msg [in]

指定 CAN 属性。包括：

***id*：**	11比特或29比特标识符，取决于格式设定
***u32cData[2]*：**	传输数据域
***u8cLen*：**	数据域字节数
***u8cFormat*：**	标准或扩展标识符
***u8cType*：**	DATA或REMOTE帧

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS：成功

示例 1.

```
/* Send a data frame which is extended format (29-bits).*/
/* The frame identifier is 0x10000000, 8 data bytes(78h,56h,534,12h,AAh,00h,55h,FFh) */
STR_CAN_T CAN_TxMsg;
CAN_TxMsg.u8cType = DATA_TYPE;
CAN_TxMsg.u8cFormat = EXTENDED_FORMAT;
CAN_TxMsg.id = 0x10000000 ;
CAN_TxMsg.u8cLen = 8;
```

```
CAN_TxMsg.u32cData[0] = 0x12345678;
CAN_TxMsg.u32cData[1] = 0xFF5500AA;
DrvCAN_Write(&CAN_TxMsg);
```

示例 2.

```
/* Send a data frame which is standard format (11-bits).*/
/* The frame identifier is 0x7FF, 1 data bytes(12h) */
STR_CAN_T CAN_TxMsg;
CAN_TxMsg.u8cType = DATA_TYPE;
CAN_TxMsg.u8cFormat = STANDARD_FORMAT;
CAN_TxMsg.id = 0x7FF ;
CAN_TxMsg.u8cLen = 1;
CAN_TxMsg.u32cData[0] = 0x12;
DrvCAN_Write(&CAN_TxMsg);
```

DrvCAN_Read

原型

STR_CAN_T DrvCAN_Read (void);

描述

该函数用于获取 CAN 接收消息信息。

参数

无

头文件

Driver/DrvCAN.h

返回值

CAN 结构体

示例

```
/* Receive a data frame, stored in the structure “CAN_RxMsg”.*/
STR_CAN_T CAN_RxMsg;
CAN_RxMsg = DrvCAN_Read();
printf("[RX] ID=%d ,Len=%d ,Data=0x%x \n",
CAN_RxMsg.id,CAN_RxMsg.u8cLen,CAN_RxMsg.u32cData[0]);
```

DrvCAN_SetAcceptanceFilter

原型

int32_t DrvCAN_SetAcceptanceFilter (
    int32_t id_Filter
);

描述

这个函数可以用来设定接收标识符过滤。

参数

id_Filter [in]

写到特定标识符过滤器中的数据

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS：成功

示例 1.

```
/* The node will recevie CAN message informat ion which ID is 0x7FE ~ 0x7FF. */
DrvCAN_SetMaskFilter(0x7FE);
DrvCAN_SetAcceptanceFilter(0x7FF);
```

示例 2.

```
/* The node will recevie CAN message informat ion which ID is 0x100. */
#define AMR_ALLMASK 0x1FFFFFF
DrvCAN_SetMaskFilter(AMR_ALLMASK); /* The node will compare all bits ID value */
DrvCAN_SetAcceptanceFilter(0x100);
```

DrvCAN_SetMaskFilter

原型

uin32_t DrvCAN_SetMaskFilter (int32_t id_Filter);

描述

这个函数可以用来设定标识符过滤掩码。

参数

id_Filter [in]

写到特定标识符过滤掩码中的数据

相应的接收掩码寄存器允许定义某些比特位置被屏蔽或者没有影响。

0 = 相应位总是通过

1 = 相应位被屏蔽，取决于 ACR 位

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS：成功

示例

```
/* The node can receive all ID (all passed) */
#define AMR_ALLPASS 0
DrvCAN_SetMaskFilter(AMR_ALLPASS); /*It means the node will compare all bits*/
```

DrvCAN_SetBusTiming

原型

```
int32_t DrvCAN_SetBusTiming(
  int8_t i8SynJumpWidth,
  int16_t i16TimeSeg1,
  int8_t i8TimeSeg2,
  int8_t i8SampPtNo
)
```

描述

该函数用于配置 CAN 时序。

参数

i8SynJumpWidth [in]

同步跳转宽度值。可以是 0 ~ 3

i16TimeSeg1 [in]

时间区域 1 的值。可以是 0 ~ 31

i8TimeSeg2 [in]

时间区域 2 的值。可以是 0 ~ 15

i8SampPtNo [in]

采样点数量。 0：单个 1：三个

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS：成功

E_DRVCAN_ERR_ARGUMENT：无效参数

示例

```
/* Set the bit timing parameter: SJW = 0; Time segment = 1 */
/* Time segment =2 and Single Sample Point ; */
DrvCAN_SetBusTiming( 0 , 1, 10 ,0);
```

DrvCAN_GetTxErrorCount

原型

int32_t DrvCAN_GetTxErrorCount(void);

描述

该函数用于获取发送错误次数。

参数

无

头文件

Driver/DrvCAN.h

返回值

发送错误次数(TEC)

示例

```
/* Get current transmit error count and stored in “i32count” parameter. */
int32_t i32count;
i32count = DrvCAN_GetTxErrorCount() ;
```

DrvCAN_GetRxErrorCount

原型

int32_t DrvCAN_GetRxErrorCount(void);

描述

该函数用于获取当前接收错误次数。

参数

无

头文件

Driver/DrvCAN.h

返回值

接收错误次数(TEC)

示例

```
/* Get current receive error count and stored in “i32count” parameter. */
int32_t i32count;
i32count = DrvCAN_GetRxErrorCount() ;
```

DrvCAN_ReTransmission

原型

void DrvCAN_ReTransmission(int32_t bIsEnable);

描述

该函数用于使能或禁止自动重传。

参数

bIsEnable [in]

使能或禁止自动重传功能

头文件

Driver/DrvCAN.h

返回值

无

示例

```
/* Disable auto-retransmission function */
DrvCAN_ReTransmission(DISABLE) ;
```

DrvCAN_Close

原型

int32_t DrvCAN_Close(void);

描述

禁止 CAN APB 时钟和总线使能功能。

参数

无

头文件

Driver/DrvCAN.h

返回值

E_SUCCESS：成功

示例

DrvCAN_Close();

DrvCAN_GetClockFreq

原型

int32_t DrvCAN_GetClockFreq(void);

描述

该函数用于获取当前 CAN 时钟频率。

参数

无

头文件

Driver/DrvCAN.h

返回值

CAN 时钟频率，单位是 Hz

如果返回值是 12000000，表示 CAN 时钟是 12MHz

示例

```
/* Get CAN clock value */
int32_t i32Value = DrvCAN_GetClock Freq();
printf("CAN Clock = %dKHz".i32Value);
```

DrvCAN_GetVersion

原型

iint32_t

DrvCAN_GetVersion (void);

描述

返回驱动当前版本号。

头文件

Driver/DrvCAN.h

返回值

版本号：

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

11. PWM 驱动

11.1. 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。有四种时钟源：12 MHz crystal 时钟，32 KHz crystal 时钟, HCLK，和内部 22MHZ 时钟。一个时钟分频器可以提供 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和极限值寄存器中的值来控制 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模式。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轮循环。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中断。

11.2. 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™ NUC100 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。

11.3. 常量定义

常量名	值	描述
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

11.4. 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 t imer 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

这个函数可以用来设定 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)。如果计数值被设置为 0，定时器将停止。

头文件

Driver/DrvPWM.h

返回值

无

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 t imer 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

这个函数可以用来取得 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

这个函数可以用来使能 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

这个函数可以用来禁止 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

这个函数可以用来清除 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 t imer 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

这个函数可以用来取得 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 t imer 6 interrupt occurs!\n);
else if(DrvPWM_GetIntFlag(DRVPWM_TIMER6)==0)
printf(”PWM t imer 6 interrupt dosen‟t occur!\n);
```

DrvPWM_GetRisingCounter

原型

uint16_t DrvPWM_GetRisingCounter(uint8_t u8Capture);

描述

这个函数可以用来取得当有上升转变时，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

这个函数可以用来取得当有下降转变时，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]

指定捕获器锁存的指示符。

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]

指定捕获器锁存的指示符。

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 timer 0~7 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]

指定定时器。

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

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

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

DRVPWM_TIMER6 或 DRVPWM_TIMER7：PWM 定时器 6 或 PWM 定时器 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 t imer 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);
```

样例代码

```
/* 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.u8HighPulseRat io = 30; /* High Pulse period : Total Pulse period = 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

这个函数可以用来使能 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

这个函数可以用来配置频率/脉冲/模式/反转功能。当用户通过 *u32Frequency* 设置一个非零的频率值时，该函数将会自动设置频率属性。当频率设定值(*u32Frequency*)没有被指定，即设置为 0 时，用户需要提供时钟选择器，预分频器和占空比等的设定值来产生需要的频率。

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]**

包含下面的参数

参数	描述
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2 DRVPWM_CLOCK_DIV_4：PWM输入时钟除4 DRVPWM_CLOCK_DIV_8：PWM输入时钟除8 DRVPWM_CLOCK_DIV_16：PWM输入时钟除16 (当u32Frequency = 0时该参数才起作用)
u8PreScale	预分频值(1~255)。如果 u8PreScale 被设置为 0，定时器将停止。 PWM 输入时钟 = PWM 源时钟/(u8PreScale + 1)。 (当 u32Frequency = 0 时该参数才起作用)
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

制定 PWM 的实际频率(Hz)

示例

```
/* PWM timer 0 output 1KHz 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.u8HighPulseRat io = 20; /* High Pulse peroid : Total Pulse peroid = 20 : 100 */
    sPt.i32Inverter = 0;
    sPt.u32Frequency = 1000; // Set 1KHz 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.u8HighPulseRat io = 20; /* High Pulse peroid : Total Pulse peroid = 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

这个函数可以用来选择 PWM0 与 PWM1，PWM2 与 PWM3，PWM4 与 PWM5，PWM6 与 PWM7 的时钟源。表示 PWM0/1 使用一个时钟源。PWM 可以使用其它的时钟源等等。换句话说，如果用户把 PWM 定时器 0 的时钟源从外部 12MHz 改变为内部 22MHz，PWM 定时器 1 的时钟源也被从外部 12MHz 改变为内部 22MHz。此外，用户也可以设置 PWM1 的时钟源为外部 12MHz，并且设置 PWM2 的时钟源为外部 32.768KHz

参数

u8Timer [in]

指定定时器

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

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

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

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

u8ClockSourceSelector [in]

设置指定 PWM 定时器的时钟源。可以是 DRVPWM_EXT_12M/DRVPWM_EXT_32K/DRVPWM_HCLK/DRVPWM_INTERNAL_22M。其中 DRVPWM_EXT_12M 为外部 12MHz crystal 时钟，DRVPWM_EXT_32K 为外部 32.768KHz crystal 时钟，DRVPWM_HCLK 为 HCLK，DRVPWM_INTERNAL_22M 为内部 22.1184MHz crystal 时钟

头文件

Driver/DrvPWM.h

返回值

无

示例

Select PWM t imer 0 and PWM timer 1 engine clock source from HCLK.

DrvPWM_SelectClockSource(DRVPWM_TIMER0, DRVPWM_HCLK);

Select PWM t imer 6 and PWM timer 7 engine clock source from external 12MHz.

DrvPWM_SelectClockSource(DRVPWM_TIMER7, DRVPWM_EXT_12M);

DrvPWM_SelectClearLatchFlagOption

原型

int32_t DrvPWM_SelectClearLatchFlagOption (int32_t i32option);

描述

该函书用于选择如何去清除捕获器上升与下降锁存指示符。

参数

i32option [in]

0：选择通过写'0'来清除捕获器锁存指示符。

1：选择通过写'1'来清除捕获器锁存指示符。

头文件

Driver/DrvPWM.h

返回值

0：成功

<0：不支持该选项

Note

只有 NuMicro™ NUC100 系列中低密度版本和 NUC101 支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指导。

DrvPWM_GetVersion

原型

uint32_t DrvPWM_GetVersion (void);

描述

获取该模块的版本号

参数

无

头文件

Driver/DrvCAN.h

返回值

PWM 驱动当前版本号：

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();
```

12. PS2 驱动

12.1. PS2 介绍

PS/2 设备控制器为 PS/2 通讯提供基本的时序控制。设备和主机之间的所有通讯都通过 CLK 和 DATA 引脚管理。接收到一个发送请求之后，设备控制器产生 CLK 信号，但是主机有最高控制权。从主机发送到设备的数据在上升沿读取，从设备发送到主机的数据在上升沿以后改变。一个 16 字节的发送 FIFO 用来减少 CPU 的干预，但是没有接收 FIFO。连续发送的时候，软件可以选择 1-16 个字节的发送 FIFO 深度。

因为 PS2 设备控制器非常简单，为了速度考虑，我们推荐尽量使用宏定义。因为没有接收 FIFO，所以 DrvPS2_Read 只读一个字节；但是 DrvPS2_Write 可以写任意长度的字节到主机

缺省 PS2 中断处理函数已经实现，就是 PS2_IRQHandler。用户可以通过函数 DrvPS2_EnableInt 安装中断回调函数，通过函数 DrvPS2_DisableInt 卸中断回调函数

12.2. PS2 特性

PS/2 设备控制器包含下面的特性：

- APB 接口兼容
- 主机通讯抑制并请求发送检测
- 接收帧错误检测
- 可编程 1 到 16 字节发送 FIFO，以降低 CPU 干涉，但是没有接收 FIFO
- 接收支持双缓冲
- 支持软件重置总线

12.3. 常量定义

常量名	值	描述
DRVPS2_RXINT	0x00000001	PS2 接收中断
DRVPS2_TXINT	0x00000002	PS2 发送中断
DRVPS2_TXFIFODEPTH	16	发送 FIFO 深度

12.4. 宏

_DRVPS2_OVERRIDE

原型

void _DRVPS2_OVERRIDE(bool state);

描述

这个宏可以用来使能/禁止软件控制 DATA/CLK 线的能力。

参数

state [in]

说明软件重置与否。1 表示使能软件控制 PS2 CLK/DATA 引脚状态；0 表示禁止软件重置功能

头文件

Driver/DrvPS2.h

返回值

无

示例

```
/* Enable Software to control DATA/CLK pin */
_DRVPS2_OVERRIDE(1)
/* Disable Software to control DATA/CLK pin */
_DRVPS2_OVERRIDE(0)
```

_DRVPS2_PS2CLK

原型

void _DRVPS2_PS2CLK(bool state);

描述

如果_DRVPS2_OVERRIDE 被调用，这个宏可以用来迫使 PS2CLK 高/低，而不考虑设备控制器内部的状态。1 表示高，0 表示低。

参数

state [in]

指定 PS2CLK 线高/低

头文件

Driver/ DrvPS2.h

返回值

无

Note

该宏仅当 DRVPS2_OVERRIDE 被调用之后才有效。

示例

```
/* Force PS2CLK pin high. */
_DRVPS2_PS2CLK(1);
/* Force PS2CLK pin low. */
_DRVPS2_PS2CLK(0);
```

_DRVPS2_PS2DATA

原型

void _DRVPS2_PS2DATA(bool state);

描述

如果_DRVPS2_OVERRIDE 被调用，这个宏可以用来迫使 PS2DATA 高/低，而不考虑设备控制器内部的状态。1 表示高，0 表示低。

参数

state [in]

指定 PS2DATA 线高/低

头文件

Driver/ DrvPS2.h

返回值

无

Note

该宏仅当 DRVPS2_OVERRIDE 被调用之后才有效。

示例

```
/* Force PS2DATApin high. */
_DRVPS2_PS2DATA (1);
/* Force PS2DATA pin low. */
_DRVPS2_PS2DATA (0);
```

_DRVPS2_CLRFIFO

原型

void _DRVPS2_CLRFIFO();

描述

这个宏可以用来清除发送 FIFO。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Clear TX FIFO. */
_DRVPS2_CLRFIFO();
```

_DRVPS2_ACKNOTALWAYS

原型

void _DRVPS2_ACKNOTALWAYS();

描述

这个宏可以用来使能不总是应答功能。如果校验错误或者停止位没有正确收到，在第 12 个时钟的时候，应答比特将不会被发送给主机。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Enable ackknowlwde NOT always. */
_DRVPS2_ACKNOTALWAYS()
```

_DRVPS2_ACKALWAYS

原型

void _DRVPS2_ACKALWAYS();

描述

这个宏可以用来使能总是应答功能。如果校验错误或者停止位没有正确收到，在第 12 个时钟的时候，主机到从机通讯时，应答比特总是会发送应答给主机。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Enable ackknowlwde always. */
_DRVPS2_ACKALWAYS()
```

_DRVPS2_RXINTENABLE

原型

void _DRVPS2_RXINTENABLE();

描述

这个宏可以用来使能接收中断。当应答比特被从主机发送给设备之后，接收中断将发生

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Enable RX interrupt. */
_DRVPS2_RXINTENABLE();
```

_DRVPS2_RXINTDISABLE

原型

void _DRVPS2_RXINTDISABLE();

描述

这个宏可以用来禁止接收中断。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Disable RX interrupt. */
_DRVPS2_RXINTDISABLE ();
```

_DRVPS2_TXINTENABLE

原型

void _DRVPS2_TXINTENABLE();

描述

这个宏可以用来使能发送中断。当 STOP 比特被发送时，发送中断将发生。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Enable TX interrupt. */
_DRVPS2_TXINTENABLE();
```

_DRVPS2_TXINTDISABLE

原型

void _DRVPS2_TXINTDISABLE ();

描述

这个宏可以用来禁止发送中断。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Disable TX interrupt. */
_DRVPS2_TXINTDISABLE();
```

_DRVPS2_PS2ENABLE

原型

void _RVPS2_PS2ENABLE ();

描述

这个宏可以用来使能 PS/2 设备控制器。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Enable PS/2 device controller. */
_DRVPS2_PS2ENABLE ();
```

_DRVPS2_PS2DISABLE

原型

void _RVPS2_PS2DISABLE ();

描述

这个宏可以用来禁止 PS/2 设备控制器。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Disable PS/2 device controller. */
_DRVPS2_PS2DISABLE ();
```

_DRVPS2_TXFIFO

原型

void _DRVPS2_TXFIFO(depth);

描述

这个宏可以用来设定发送 FIFO 深度。范围[1,16]

参数

depth [in]

指定 PS2DATA 线高/低

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Set TX FIFO depth to 16 bytes. */
_DRVPS2_TXFIFO(16);
```

```
/* Set TX FIFO depth to 1 bytes. */
_DRVPS2_TXFIFO(1);
```

_DRVPS2_SWOVERRIDE

原型

void _DRVPS2_SWOVERRIDE(bool data, bool clk);

描述

这个宏可以通过软件重置用来设定 PS2DATA 和 PS2CLK 线的状态。它等于下面的宏：

DRVPS2_PS2DATA(data);

DRVPS2_PS2CLK(clk);

DRVPS2_OVERRIDE(1);

参数

data [in]

指定 PS2DATA 线高/低

clk [in]

指定 PS2CLK 线高/低

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Set PS2DATA to high and set PS2CLK to low. */
_DRVPS2_SWOVERRIDE(1, 0);
/* Set PS2DATA to low and set PS2CLK to high. */
_DRVPS2_SWOVERRIDE(0, 1);
```

_DRVPS2_INTCLR

原型

void _DRVPS2_INTCLR(uint8_t intclr) ;

描述

这个宏可以用来清除中断状态。

参数

intclr [in]

指定清除接收/发送中断。Intclr=0x1：清除接收中断；Intclr=0x2 清除发送中断；Intclr=0x3 清除接收和发送中断

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Clear RX interrupt. */
_DRVPS2_INTCLR(1);
/* Clear TX interrupt. */
_DRVPS2_INTCLR(2);
/* Clear TX and RX interrupt. */
_DRVPS2_INTCLR(3);
```

_DRVPS2_RXDATA

原型

uint8_t _DRVPS2_RXDATA();

描述

从接收寄存器读一个字节。

参数

无

头文件

Driver/ DrvPS2.h

返回值

从主机收到的一个字节

示例

```
/* Read one byte from PS/ 2 receive data register. */
uint8_t u8ReceiveData;
u8ReceiveData = _DRVPS2_RXDATA();
```

_DRVPS2_TXDATAWAIT

原型

void _DRVPS2_TXDATAWAIT(uint32_t data, uint32_t len);

描述

这个宏可用来等待发送 FIFO 空，设定发送 FIFO 深度(length-1)和填充发送 FIFO0-3(寄存器 PS2TXDATA0)。如果总线空闲，数据将马上被发送。长度取值范围为 1 到 16 字节。如果传输大小超过 4 字节，用户应该在调用完_DRVPS2_TXDATAWAIT()之后调用 DRVPS2_TXDATA1~3()来传输余下的数据。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

要发送的数据

len [in]

要发送的数据长度。单位是字节。范围[1, 16]

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Wait TX FIFO empty and then write 16 bytes to TX FIFO. The sixteen bytes consist of 0x01 to
0x16. */
_DRVPS2_TXDATAWAIT(0x04030201, 16);
_DRVPS2_TXDATA1(0x08070605);
_DRVPS2_TXDATA2(0x0C0B0A09);
_DRVPS2_TXDATA3(0x100F0E0D);
/* Wait TX FIFO empty and then write 5 bytes to TX FIFO. The six bytes consist of 0x01 to
0x05. */
_DRVPS2_TXDATAWAIT(0x04030201, 5);
_DRVPS2_TXDATA1(0x05);
/* Wait TX FIFO empty and then write 3 bytes to TX FIFO. The three bytes consist of 0x01 to
0x03. */
_DRVPS2_TXDATAWAIT(0x030201, 3);
```

_DRVPS2_TXDATA

原型

void _DRVPS2_TXDATA(uint32_t data, uint32_t len);

描述

这个宏可用来设定发送 FIFO 深度和填充发送 FIFO0-3，而不用等待发送 FIFO 空，如果总线空闲，数据将马上被发送。len 取值范围为[1, 16]。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

要发送的数据

len [in]

要发送的数据长度，单位字节，范围 [1, 16]

头文件

Driver/ DrvPS2.h

返回值

无

Note

如果传输大小超过 4 字节，用户应该在调用完_DRVPS2_TXDATAWAIT()之后调用 DRVPS2_TXDATA1~3()来传输余下的数据。

示例

```
/*Write 16 bytes to TX FIFO. The sixteen bytes consist of 0x01 to 0x16. */
_DRVPS2_TXDATA(0x04030201, 16);
_DRVPS2_TXDATA1(0x08070605);
_DRVPS2_TXDATA2(0x0C0B0A09);
_DRVPS2_TXDATA3(0x100F0E0D);
/* Write 5 bytes to TX FIFO. The six bytes consist of 0x01 to 0x05. */
_DRVPS2_TXDATA(0x04030201, 5);
_DRVPS2_TXDATA1(0x05);
/* Write 3 bytes to TX FIFO. The three bytes consist of 0x01 to 0x03. */
_DRVPS2_TXDATA(0x030201, 3);
```

_DRVPS2_TXDATA0

原型

void _DRVPS2_TXDATA0(uint32_t data);

描述

这个宏可用来填充发送 FIFO0-3，而不用等待发送 FIFO 空和设定发送 FIFO 深度，如果总线空闲，数据将马上被发送。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

要发送的数据

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Write 16 bytes to TX FIFO. The sixteen bytes consist of 0x01 to 0x16. */
while(_DRVPS2_ISTXEMPTY()==0);
_DRVPS2_TXFIFO(16);
_DRVPS2_TXDATA0(0x04030201);
_DRVPS2_TXDATA1(0x08070605);
_DRVPS2_TXDATA2(0x0C0B0A09);
_DRVPS2_TXDATA3(0x100F0E0D);
```

_DRVPS2_TXDATA1

原型

void _DRVPS2_TXDATA1(uint32_t data);

描述

这个宏可用来填充发送 FIFO4-7，而不用等待发送 FIFO 空和设定发送 FIFO 深度，如果总线空闲，数据将马上被发送。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

要发送的数据

头文件

Driver/ DrvPS2.h

返回值

无

示例

请参考_DRVPS2_TXDATA0()中的示例

_DRVPS2_TXDATA2

原型

void _DRVPS2_TXDATA2(uint32_t data);

描述

这个宏可用来填充发送 FIFO8-11，而不用等待发送 FIFO 空和设定发送 FIFO 深度，如果总线空闲，数据将马上被发送。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

说明要发送的数据

头文件

Driver/ DrvPS2.h

返回值

无

示例

请参考_DRVPS2_TXDATA0()中的示例

_DRVPS2_TXDATA3

原型

void _DRVPS2_TXDATA3(uint32_t data);

描述

这个宏可用来填充发送 FIFO12-15，而不用等待发送 FIFO 空和设定发送 FIFO 深度，如果总线空闲，数据将马上被发送。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

data [in]

要发送的数据.

头文件

Driver/ DrvPS2.h

返回值

无

示例

请参考_DRVPS2_TXDATA0()中的示例

_DRVPS2_ISTXEMPTY

原型

uint8_t _DRVPS2_ISTXEMPTY();

描述

这个宏可以用来检查发送 FIFO 是否为空。

当发送字节数等于 FIFODEPTH 时，TXEMPTY 位将被设成 1。

参数

无

头文件

Driver/ DrvPS2.h

返回值

发送 FIFO 空状态

0：发送 FIFO 空

1：发送 FIFO 非空

示例

请参考_DRVPS2_TXDATA0()中的示例

_DRVPS2_ISFRAMEERR

原型

uint8_t _DRVPS2_ISFRAMEERR();

描述

这个宏可以用来检查是否发生帧错误。主机发送数据到设备的时候，如果

STOP 比特没有收到，帧错误发生。如果帧错误发生，第 12 个时钟之后，DATA 线将保持在低电平状态。这时软件重置 PS2CLK 来发送时钟信号，直到 PS2DATA 变成高电平这之后，设备发送一个 “Resend”命令到主机。

参数

无

头文件

Driver/ DrvPS2.h

返回值

帧错误状态

0：没有帧错误

1：帧错误

示例

```
/* Check Frame error and print the result. */
if(_DRVPS2_ISFRAMEERR()==1)
   printf(“Frame error happen!!\n”);
else
   printf(“Frame error not happen!!\n”);
```

_DRVPS2_ISRXBUSY

原型

uint8_t _DRVPS2_ISRXBUSY();

描述

这个宏可以用来检查接收是否正忙。如果正忙表示 PS/2 设备正在接收数据。

参数

无

头文件

Driver/ DrvPS2.h

返回值

接收忙标志。

0：接收不忙

1：接收正忙

示例

```
/* Check RX is busy or not. */
if(_DRVPS2_ISRXBUSY()==1)
   printf("RX is busy!\n");
else
   printf("RX is not busy!\n");
```

12.5. 函数

DrvPS2_Open

原型

int32_t DrvPS2_Open();

描述

这个函数可以用来初始化 PS/2。包括使能 PS2 时钟，使能 PS/2 控制器，清除发送 FIFO， 设定发送 FIFO 深度为默认值 0。

参数

无

头文件

Driver/DrvPS2.h

返回值

E_SUCCESS

示例

```
/* Initialize PS/2 IP. */
DrvPS2_Open();
```

DrvPS2_Close

原型

void DrvPS2_Close();

描述

这个函数可以用来禁止 PS/2 控制器，禁止 PS/2 时钟，设定发送 FIFO 深度为默认值 0。

参数

无

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Close PS2 IP. */
DrvPS2_Close ();
```

DrvPS2_EnableInt

原型

```
int32_t DrvPS2_EnableInt (
    uint32_t  u32InterruptFlag,
    PFN_DRVPS2_CALLBACK pfncallback
);
```

描述

这个函数可以用来使能接收/发送中断，并且安装中断回调函数。

参数

u32InterruptFlag [in]

指定要使能的接收/发送中断标志。可以是 DRVPS2_TXINT 或者 DRVPS2_RXINT 或者 DRVPS2_TXINT| DRVPS2_RXINT

pfncallback [in]

指定中断回调函数指针。当 PS2 中断发生时，这个函数将被调用

头文件

Driver/ DrvPS2.h

返回值

E_SUCCESS

示例

```
/* Enable TX/RX interrupt, install TX/RX call back function: PS2Mouse_IRQHandler(); */
DrvPS2_EnableInt(DRVPS2_TXINT| DRVPS2_RXINT, PS2Mouse_IRQHandler);
```

DrvPS2_DisableInt

原型

void DrvPS2_DisableInt (uint32_t u32InterruptFlag);

描述

这个函数可以用来禁止接收/发送中断并且卸载中断回调函数。

参数

u32InterruptFlag [in]

指定要被禁止的接收/发送中断标志。可以是 DRVPS2_TXINT 或者 DRVPS2_RXINT 或者 DRVPS2_TXINT| DRVPS2_RXINT

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Disable TX/RX interrupt and uninstall TX and RX call back function. */
DrvPS2_DisableInt(DRVPS2_TXINT| DRVPS2_RXINT);
```

DrvPS2_IsIntEnabled

原型

uint32_t DrvPS2_IsIntEnabled (uint32_t u32InterruptFlag);

描述

这个函数可以用来检查中断是否被使能。

参数

u32InterruptFlag [in]

指定要被检查的接收/发送中断标志。可以是 DRVPS2_TXINT 或者 DRVPS2_RXINT 或者 DRVPS2_TXINT| DRVPS2_RXINT

头文件

Driver/ DrvPS2.h

返回值

0：没有中断被使能

2：发送中断被使能

4：接收中断被使能

6：发送和接收中断均被使能

示例

```
/* Check TX and RX interrupt enable or not enable. */
uint32_u32TXRXIntEnable
u32TXRXIntEnable = DrvPS2_IsIntEnabled(DRVPS2_TXINT| DRVPS2_RXINT)
if(u32TXRXIntEnable ==0)
   printf("No interrupt be enable!!\n");
else if(u32TXRXIntEnable ==2)
   printf("TX interrupt be enable!!\n");
else if(u32TXRXIntEnable ==4)
   printf("RX interrupt be enable!!\n");
else if(u32TXRXIntEnable ==6)
   printf("TX and RX interrupt be enable!!\n");
```

DrvPS2_ClearInt

原型

uint32_t DrvPS2_ClearInt (uint32_t u32InterruptFlag);

描述

这个函数可以用来清除中断标志。

参数

U32InterruptFlag [in]

指定要被清除的接收/发送中断标志。可以是 DRVPS2_TXINT 或者 DRVPS2_RXINT 或者 DRVPS2_TXINT| DRVPS2_RXINT

头文件

Driver/DrvPS2.h

返回值

E_SUCCESS：成功

示例

```
/* Clear TX interrupt. */
DrvPS2_ClearInt(DRVPS2_TXINT);
/* Clear RX interrupt. */
DrvPS2_ClearInt(DRVPS2_RXINT);
/* Clear TX and RX interrupt. */
DrvPS2_ClearInt(DRVPS2_TXINT| DRVPS2_RXINT);
```

DrvPS2_GetIntStatus

原型

int8_t DrvPS2_GetIntStatus(uint32_t u32InterruptFlag);

描述

这个函数可以用来检查中断标志。如果相应中断发生将返回 TRUE。

参数

U32InterruptFlag [in]

指定要被检查的接收/发送中断标志。可以是 DRVPS2_TXINT 或者 DRVPS2_RXINT

头文件

Driver/ DrvPS2.h

返回值

TRUE：相应中断发生

FALSE：检查的中断没有发生

示例

```
/* Check TX interrupt status */
int8_t i8InterruptStatus;
i8InterruptStatus = DrvPS2_GetIntStatus(DRVPS2_TXINT);
if(i8InterruptStatus==TRUE)
   printf(“TX interrupt that be checked happens”\n);
else
   printf(“TX interrupt doesn‟t happen”\n);
```

DrvPS2_SetTxFIFODepth

原型

void DrvPS2_SetTxFIFODepth (uint16_t u16TxFIFODepth);

描述

这个函数可以用来设定发送 FIFO 深度。这个函数将调用宏 DRVPS2_TXFIFO 来设定发送 FIFO 深度。

参数

u16TxFIFODepth [in]

指定发送 FIFO 深度。范围[1, 16]

头文件

Driver/ DrvPS2.h

返回值

无

示例

```
/* Set TX FIFO depth to 16 bytes. */
DrvPS2_SetTxFIFODepth(16);
/* Set TX FIFO depth to 1 byte. */
DrvPS2_SetTxFIFODepth(1);
```

DrvPS2_Read

原型

int32_t DrvPS2_Read (uint8_t *pu8RxBuf);

描述

这个函数可以用来读一个字节到缓存 pu8RxBuf 中。这个函数将调用宏 DRVPS2_RXDATA 来接收数据。

参数

pu8RxBuf [out]

存放接收数据的缓存地址。缓存只要一个字节就可以

头文件

Driver/ DrvPS2.h

返回值

E_SUCCESS：成功

示例

```
/* Read RX data and print it. */
uint8_t u8RXData;
DrvPS2_Read(&u8RXData);
printf("RX data is %x\n", u8RXData);
```

DrvPS2_Write

原型

int32_t

```
DrvPS2_Write(
    uint32_t     *pu32TxBuf,
    uint32_t     u32WriteBytes
);
```

描述

这个函数可以用来写缓存 pu32TxBuf 和长度 u32WriteBytes 到主机。如果要发送的数据长度小于 16，为了速度考虑，请使用系列宏定义 DRVPS2_TXDATAxxx

参数

pu32TxBuf [in]

要发送到主机的数据

u32WriteBytes [in]

要发送到主机的数据长度

头文件

Driver/ DrvPS2.h

返回值

E_SUCCESS：成功

示例

```
/* Write 64 bytes to TX buffer and TX buffer will send the 64 bytes out. */
uint32_t au32TXData[64];
DrvPS2_Write(au32TXData, 64);
```

DrvPS2_GetVersion

原型

int32_t DrvPS2_GetVersion(void);

描述

返回驱动当前版本号。

头文件

Driver/ DrvPS2.h

返回值

PS2 驱动当前版本号：

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

示例

```
/* Get PS/2 driver current version number */
int32_t i32Ps2VersionNum;
i32Ps2VersionNum = DrvPS2_GetVersion ();
```

13. FMC 驱动

13.1. FMC 介绍

NuMicro™ NUC100 系列配置了 128/64/32k 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 与 Config1)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™ NUC100 系列还额外提供 4K 字节数据闪存，用于存放应用程序相关的数据。对于 128K 字节的设备，数据闪存和 128K 应用程序闪存共享 128K 字节空间，数据区基地址通过 Config1 来定义。数据闪存大小可由用户根据应用程序需要定义。

13.2. FMC 特性

FMC 包含下面的特性：

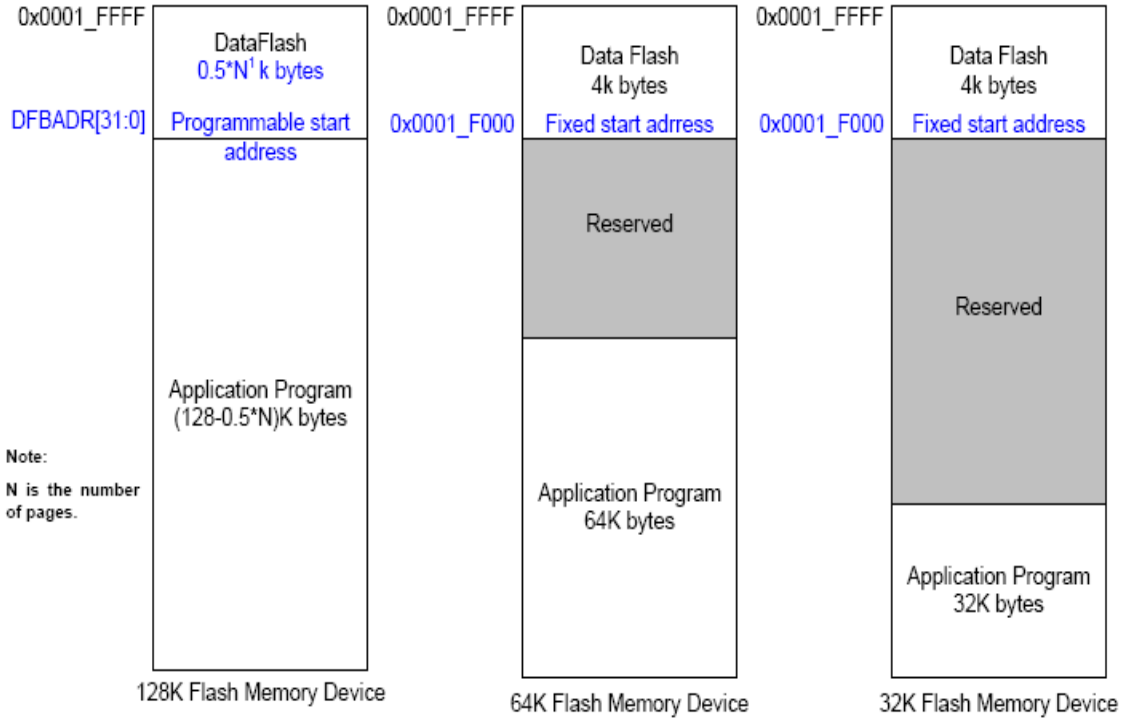
- 128/64/32kB 应用程序闪存 (APROM)
- 4kB 在系统编程加载器闪存(LDROM)
- 4kB 数据闪存，擦除单位 512 字节
- 对于 128K 字节应用程序闪存，数据闪存开始地址和存储器大小可编程
- 提供用户配置来控制系统逻辑
- 当 MCU 运行在 APROM 时，APROM 不能更新；当 MCU 运行在 LDROM 时，LDROM 不能更新

Memory Address Map

块名称	大小	起始地址	结束地址
AP ROM	32KB 64KB 128KB (128-0.5*N)KB	0x00000000	0x00007FFF 0x0000FFFF 128KB, DFEN = 0, 0x0001FFFF 128KB, DFEN = 1, (DFBADR - 1)
Data Flash	4KB 4KB 0KB (0.5*N)KB	0x0001F000 0x0001F000 无 DFBADR	0x0001FFFF 0x0001FFFF 128KB, DFEN = 0, 无 128KB, DFEN = 1, 0x0001FFFF
LD ROM	4KB	0x00100000	0x00100FFF

User Configuration	2Words	0x00300000	0x00300004

Flash Memory Structure



13.3. 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

13.4. 函数

DrvFMC_EnableISP

原型

void DrvFMC_EnableISP(void);

描述

使能 ISP 功能。该函数将会检查内部 22M 振荡器是否使能，如果没有使能，该函数将会自动使能 22M 振荡器。在 ISP 结束后，如果需要，如果用户可以使用 DrvSYS_SetOscCtrl()来禁止 22M 振荡器。

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

选择下次从 APROM 还是 LDROM 启动。

参数

boot [in]

指定 E_FMC_APROM 还是 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

取得当前启动设定值。

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在 APROM

E_FMC_LDROM 当前启动选择设定值是在 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 更新功能使能，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

使能闪存访问节电功能。如果 CPU 时钟低于 24MHz，用户可以使能闪存节电功能。

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

禁止闪存访问节电功能。

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 或者 Config 区域。APROM 和数据闪存的存储映射取决于 NuMicro™ NUC100 系列的产品类型。闪存大小请参考附录 NuMicro™ NUC100 系列产品选型指导。详细的 Config0 和 Config1 中对应的功能在 TRM 的 FMC 章节描述。

参数

u32addr [in]

APROM, LDROM, Data Flash 或者 Config 区域的字地址

u32data [in]

要写到 APROM, LDROM, Data Flash 或者 Config 区域中的字数据

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 或者 Config 区域中读 4 个字节的数据。APROM 和数据闪存的存储映射取决于 NuMicro™ NUC100 系列的产品类型。闪存大小请参考附录 NuMicro™ NUC100 系列产品选型指导。

参数

u32addr [in]

APROM, LDROM, Data Flash 或者 Config 区域的字地址

u32data [in]

缓存地址，用于存放从 APROM, LDROM, Data Flash 或者 Config 区域中读取的数据。

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

以页为单位擦除 APROM, LDROM, Data Flash 或者 Config 区域，每页 512 个字节。APROM 和数据闪存的存储映射取决于 NuMicro™ NUC100 系列的产品类型。闪存大小请参考附录 NuMicro™ NUC100 系列产品选型指导。

参数

u32addr [in]

APROM, LDROM, Data Flash 的页基地址，或者 Config0 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, uint32_t u32data1);

描述

擦除 Config 区域并且写数据到 Config0 和 Config1。详细的 Config0 和 Config1 中对应的功能在 TRM 的 FMC 章节描述。

参数

u32data0 [in]

写到 Config0 的字数据

u32data1 [in]

写到 Config1 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 and word data 0x1E000 into Config1 */
DrvFMC_Config (0xFFFFFFFE, 0x1E000);
```

DrvFMC_ReadDataFlashBaseAddr

原型

uint32_t DrvFMC_ReadDataFlashBaseAddr(void);

描述

读取数据闪存基地址。对于 128k 字节闪存的 MCU，数据闪存的基地址由用户在 Config1 中定义。对于闪存小于 128k 字节的 MCU，数据闪存基地址固定在 0x1F000。

参数

无

头文件

Driver/DrvFMC.h

返回值

数据闪存基地址

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 访问的低速模式。当 CPU 运行在低速时，可以提高 Flash 访问性能。

Note

仅 NuMicro™ NUC100 系列的低密度版本和 NUC101 支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指导。而且，该位只能在 HCLK ≤ 25MHz 时置位，若 HCLK > 25MHz，CPU 会获得错误代码，并导致失败的结果。

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。

Note

仅 NuMicro™ NUC100 系列的低密度版本和 NUC101 支持该功能。详细请参考附录 NuMicro™ NUC100 系列产品选型指导。

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

int32_t DrvFMC_GetVersion(void);

描述

获取该模块版本号。

头文件

Driver/ DrvPS2.h

返回值

版本号：

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

14. USB 驱动

14.1. 介绍

该章节提供给那些正在用 USB 设备控制器来完成 USB 应用的制造商。假设用户对 USB1.1/USB2.0 熟悉。

14.2. 特性

- 与 USB2.0 全速兼容, 12Mbps.
- 提供 1 个中断源，4 个中断事件.
- 支持控制, 批量, 中断, 和等时传输.
- 当没有总线信号超过 3ms 时，挂起.
- 提供 6 个端点，可配置.
- 包含 512 个字节的内部 SRAM 用作 USB 缓存.
- 提供远程唤醒能力.

14.3. USB 框架

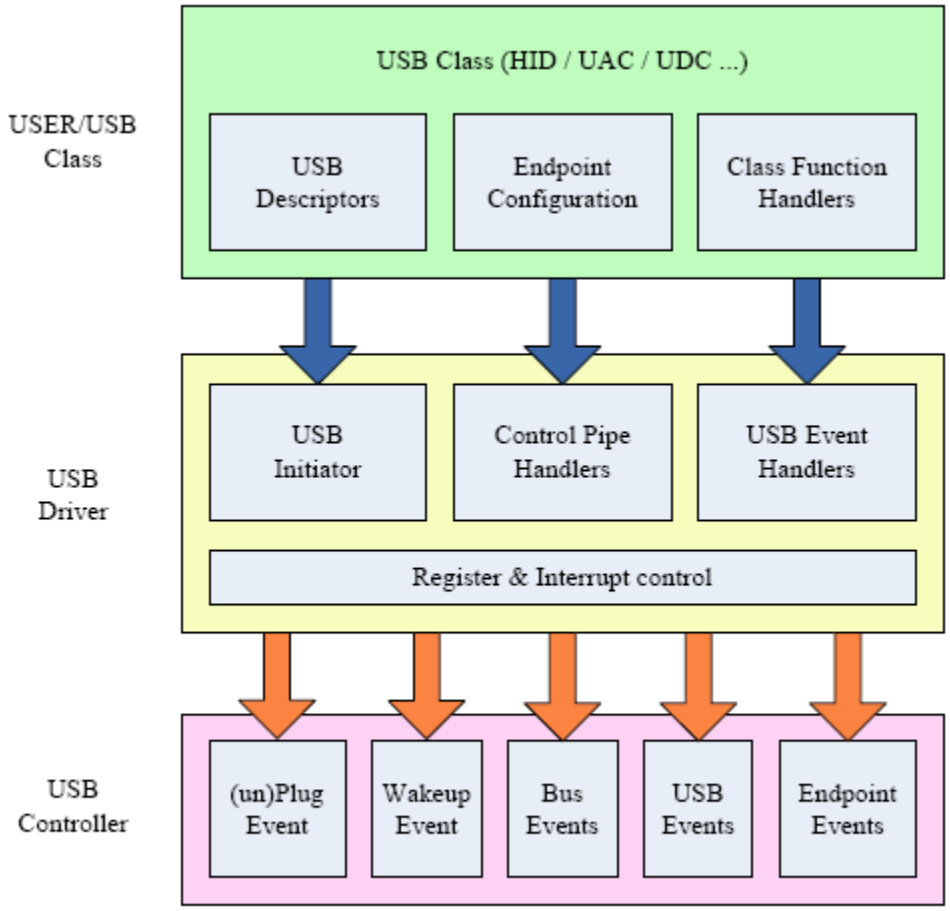


Figure 14-1: USB Framework

上图展示了 USB 设备库的框架。最底层是 USB 控制器。USB 控制器会根据 USB，BUS 和悬空检测等的状态来产生不同的中断事件。所有事件都是通过 USB 驱动相关的事件处理函数被处理。USB 驱动还会处理 USB 协议中控制管道的基本处理函数。大部分具有功能依赖性的处理函数和 USB 描述符必须由用户应用程序或者 USB 类来定义。

14.4. 调用流程

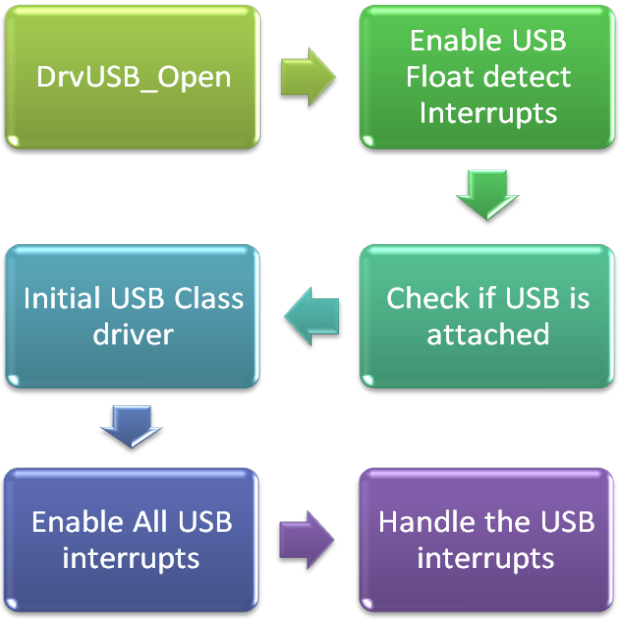


Figure 14-2: USB Driver Call Flow

上图展示了 USB 驱动的调用流程。DrvUSB_Open 用于初始化 USB 设备控制器。然后 USB 悬空检测被使能，来检测 USB 插入/拔出事件。如果 USB 已经连接，需要调用 USB 类驱动来初始化 USB 类指定的描述，事件处理函数。最后，所有相关的 USB 中断被使能，用于处理 USB 事件。

14.5. 常量定义

USB Register Address

常量名	值	描述
USBD_INTEN	0x40060000	USB 中断使能寄存器地址
USBD_INTSTS	0x40060004	USB 中断事件状态寄存器地址
USBD_FADDR	0x40060008	USB 设备功能地址寄存器地址
USBD_EPSTS	0x4006000C	USB 端点状态寄存器地址
USBD_ATTR	0x40060010	USB 总线状态和属性寄存器地址
USBD_FLDETB	0x40060014	USB 悬空检测寄存器地址
USBD_BUFSEG	0x40060018	Setup 令牌缓冲分割寄存器地址
USBD_BUFSEG0	0x40060020	端点 0 缓冲分割寄存器地址
USBD_MXPLD0	0x40060024	端点 0 最大有效载荷寄存器地址
USBD_CFG0	0x40060028	端点 0 配置寄存器地址
USBD_CFGP0	0x4006002C	端点 0 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_BUFSEG1	0x40060030	端点 1 缓冲分割寄存器地址
USBD_MXPLD1	0x40060034	端点 1 最大有效载荷寄存器地址

USBD_CFG1	0x40060038	端点 1 配置寄存器地址
USBD_CFGP1	0x4006003C	端点 1 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_BUFSEG2	0x40060040	端点 2 缓冲分割寄存器地址
USBD_MXPLD2	0x40060044	端点 2 最大有效载荷寄存器地址
USBD_CFG2	0x40060048	端点 2 配置寄存器地址
USBD_CFGP2	0x4006004C	端点 2 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_BUFSEG3	0x40060050	端点 3 缓冲分割寄存器地址
USBD_MXPLD3	0x40060054	端点 3 最大有效载荷寄存器地址
USBD_CFG3	0x40060058	端点 3 配置寄存器地址
USBD_CFGP3	0x4006005C	端点 3 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_BUFSEG4	0x40060060	端点 4 缓冲分割寄存器地址
USBD_MXPLD4	0x40060064	端点 4 最大有效载荷寄存器地址
USBD_CFG4	0x40060068	端点 4 配置寄存器地址
USBD_CFGP4	0x4006006C	端点 4 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_BUFSEG5	0x40060070	端点 5 缓冲分割寄存器地址
USBD_MXPLD5	0x40060074	端点 5 最大有效载荷寄存器地址
USBD_CFG5	0x40060078	端点 5 配置寄存器地址
USBD_CFGP5	0x4006007C	端点 5 设定 Stall 和清除 In/Out 准备好控制寄存器地址
USBD_DRVSE0	0x40060090	USB 驱动 SE0 控制寄存器地址
USBD_SRAM_BASE	0x40060100	USB PDMA 控制寄存器地址

INTEN Register Bit Definition

常量名	值	描述
INTEN_INNAK	0x00008000	激活 IN 令牌 NAK 中断功能和它的状态标志
INTEN_WAKEUP_EN	0x00000100	唤醒功能使能
INTEN_WAKEUP_IE	0x00000008	USB 唤醒中断使能
INTEN_FLDET_IE	0x00000004	悬空检测中断使能
INTEN_USB_IE	0x00000002	USB 事件中断使能
INTEN_BUS_IE	0x00000001	BUS 事件中断使能

INTSTS Register Bit Definition

常量名	值	描述
INTSTS_SETUP	0x80000000	Setup 事件状态
INTSTS_EPEVT5	0x00200000	端点 5 的 USB 事件状态
INTSTS_EPEVT4	0x00100000	端点 4 的 USB 事件状态
INTSTS_EPEVT3	0x00080000	端点 3 的 USB 事件状态
INTSTS_EPEVT2	0x00040000	端点 2 的 USB 事件状态
INTSTS_EPEVT1	0x00020000	端点 1 的 USB 事件状态
INTSTS_EPEVT0	0x00010000	端点 0 的 USB 事件状态
INTSTS_WAKEUP_STS	0x00000008	唤醒中断状态
INTSTS_FLDET_STS	0x00000004	悬空检测中断状态
INTSTS_USB_STS	0x00000002	USB 事件中断状态
INTSTS_BUS_STS	0x00000001	总线中断状态

ATTR Register Bit Definition

常量名	值	描述
ATTR_BYTEM	0x00000400	CPU 访问 USB RAM 大小模式选择
ATTR_PWMDN	0x00000200	Power down PHY，低有效
ATTR_DPPU_EN	0x00000100	接在 D+上的上拉电阻使能
ATTR_USB_EN	0x00000080	USB 控制器使能
ATTR_RWAKEUP	0x00000020	远程唤醒
ATTR_PHY_EN	0x00000010	PHY 功能使能
ATTR_TIMEOUT	0x00000008	超时状态
ATTR_RESUME	0x00000004	恢复状态
ATTR_SUSPEND	0x00000002	挂起状态
ATTR_USBRST	0x00000001	USB 复位状态

Configration Register Bit Definition

常量名	值	描述
CFG_CSTALL	0x00000200	清除 STALL 响应
CFG_DSQ_SYNC	0x00000080	数据序列同步
CFG_STATE	0x00000060	端点状态
CFG_EPT_IN	0x00000040	IN 端点
CFG_EPT_OUT	0x00000020	OUT 端点
CFG_ISOCH	0x00000010	等时端点
CFG_EP_NUM	0x0000000F	端点号

Extra-Configration Register Bit Definition

常量名	值	描述
CFGP_SSTALL	0x00000002	设置设备响应 STALL
CFGP_CLRRDY	0x00000001	清除 Ready

14.6. 宏

_DRVUSB_ENABLE_MISC_INT

原型

```
void _DRVUSB_ENABLE_MISC_INT (
    uint32_t      u32Flags
);
```

描述

使能/关闭各种 USB 中断，包括 USB 事件，唤醒事件，悬空检测事件和总线事件。

参数

u32Flags [in]

USB 中断事件，可以是下列的标志

IEF_WAKEUP：唤醒中断标志

IEF_FLD：悬空检测中断标志

IEF_USB：USB 事件中断标志

IEF_BUS：总线事件中断标志

u32Flag = 0 将禁止所有的 USB 中断

头文件

Driver/DrvUsb.h

返回值

无

示例

```
_DRVUSB_ENABLE_MISC_INT(0); /* Disable All USB-related interrupts. */
_DRVUSB_ENABLE_MISC_INT(IEF_WAKEUP | IEF_WAKEUPEN | IEF_FLD |
IEF_USB | IEF_BUS); /* Enable wakeup, float-detection, USB and bus interrupts */
```

_DRVUSB_ENABLE_WAKEUP

原型

void _DRVUSB_ENABLE_WAKEUP (void);

描述

使能 USB 唤醒功能。如果 USB 唤醒功能被使能，USB 总线上的任何活动都可以用于从 power down 唤醒 CPU。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

```
_DRVUSB_ENABLE_WAKEUP(); /* To enable the USB wakeup function */
```

_DRVUSB_DISABLE_WAKEUP

原型

void _DRVUSB_DISABLE_WAKEUP (void);

描述

禁止 USB 唤醒功能。如果 USB 唤醒功能被禁止，USB 不能用于从 power down 唤醒 CPU。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_DISABLE_WAKEUP(); /* To avoid wakeup CPU by USB */

_DRVUSB_ENABLE_WAKEUP_INT

原型

void _DRVUSB_ENABLE_WAKEUP_INT (void);

描述

使能唤醒中断。当唤醒中断被使能，USB 会产生一个唤醒事件中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_ENABLE_WAKEUP_INT() /* To enable wakeup event interrupt */

_DRVUSB_DISABLE_WAKEUP_INT

原型

void _DRVUSB_DISABLE_WAKEUP_INT (void);

描述

禁止唤醒中断，避免当 CPU 从 power down 唤醒的时候产生一个中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_DISABLE_WAKEUP_INT () /* To disable wakeup event interrupt */

_DRVUSB_ENABLE_FLDET_INT

原型

void _DRVUSB_ENABLE_FLDET_INT (void);

描述

使能悬空检测中断。当 USB 插入/拔出时将发生中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_ENABLE_FLDET_INT() /* To enable float-detection interrupt */

_DRVUSB_DISABLE_FLDET_INT

原型

void _DRVUSB_DISABLE_FLDET_INT (void);

描述

禁止悬空检测中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_DISABLE_FLDET_INT() /* To disable float-detection interrupt */

_DRVUSB_ENABLE_USB_INT

原型

void _DRVUSB_ENABLE_USB_INT (void);

描述

使能 USB 中断。仅可以用于控制 USB 中断，而_DRVUSB_ENABLE_MISC_INT()可以用于同时控制所有 USB 相关的中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_ENABLE_USB_INT () /* To enable USB interrupt */

_DRVUSB_DISABLE_USB_INT

原型

void _DRVUSB_DISABLE_USB_INT (void);

描述

禁止 USB 中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

DRVUSB DISABLE _USB_INT () /* To disable USB interrupt */

_DRVUSB_ENABLE_BUS_INT

原型

void _DRVUSB_ENABLE_BUS_INT (void);

描述

使能 USB 总线中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_ENABLE_BUS_INT () /* To enable USB bus interrupt */

_DRVUSB_DISABLE_BUS_INT

原型

void _DRVUSB_DISABLE_BUS_INT (void);

描述

禁止 USB 总线中断。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_DISABLE_BUS_INT () /* To disable USB bus interrupt */

_DRVUSB_CLEAR_EP_READY_AND_TRIG_STALL

原型

```
void     _DRVUSB_CLEAR_EP_READY_AND_TRIG_STALL (
   uint32_t     u32EPId
);
```

描述

清除 USB 端点 In/Out 就绪标志并且响应 STALL。

参数

u32EPId[in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_CLEAR_EP_READY_AND_TRIG_STALL(3) /* To clear ready flag of USB endpoint identity 3 and let it to response STALL. */

Notes

这里, 端点号表示 USB 设备硬件中的端点号，而不是 USB 标准中定义的端点号。

_DRVUSB_CLEAR_EP_READY

原型

```
void  _DRVUSB_CLEAR_EP_READY (
   uint32_t     u32EPId
);
```

描述

清除端点 In/Out 就绪标志。

参数

u32EPId[in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

无

示例

```
_DRVUSB_CLEAR_EP_READY(1) /* To clear ready flag of USB endpoint identity 1. */
```

_DRVUSB_SET_SETUP_BUF

原型

```
void _DRVUSB_SET_SETUP_BUF (
   uint32_t     u32BufAddr
);
```

描述

指定 Setup 传输的缓冲区地址。该缓冲区用于存储 setup 令牌数据，数据大小根据 USB 标准固定为 8 个字节。所以，该缓冲区地址必须是 8 字节对齐的。

参数

u32BufAddr [in]

setup 令牌的缓冲区地址。必须是 USB_BA+0x100 ~ USB_BA+0x1FF，USB_BA 为 0x40060000

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_SET_SETUP_BUF(0x400602F8) /* Set the setup packet address to 0x400602F8 */

_DRVUSB_SET_EP_BUF

原型

void _DRVUSB_SET_EP_BUF (
  uint32_t u32EPId,
  uint32_t u32BufAddr
);

描述

为指定的硬件端点号指定缓冲区地址，缓冲区地址必须是 8 字节对齐。该缓冲区将会用于缓冲 USB 传输中 IN/OUT 的数据。缓冲区大小取决于相关端点的最大载荷。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

u32BufAddr [in]

用于设定缓冲区地址，有效的地址是 0x40060100 ~ 0x400602F8，而且缓冲区地址+最大载荷必须要小于 0x400602FF

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_SET_ EP_BUF(1, 0x40060100) /* Set the buffer address of endpoint identity 1 to 0x40060100 */

_DRVUSB_TRIG_EP

原型

void _DRVUSB_TRIG_EP (
  uint32_t u32EPId,
  uint32_t u32TrigSize
);

描述

触发指定端点的下一次传输，同时传输大小被定义。

参数

u32EPId [in]

触发数据 In 或 Out 传输的端点号(有效值：0 ~ 5)

u32TrigSize [in]

对数据 Out 传输来说, 这个值表示从主机接收的最大字节数。对数据 In 传输来说，这个值表示发送到主机的字节数

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Trigger the transaction of endpoint identity 1 and the transaction payload size is 64 bytes */
_DRVUSB_TRIG_EP (1, 64)
```

_DRVUSB_GET_EP_DATA_SIZE

原型

```
uint32_t
_DRVUSB_GET_EP_DATA_SIZE (
    uint32_t      u32EPId
);
```

描述

指定端点发送到主机或者从主机收到的数据长度。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

对于 IN 端点：发送到主机的数据长度，单位是字节

对于 OUT 端点：从主机接收到的实际数据长度，单位是字节

示例

```
/* To get the size of received data of endpoint identity 1. */
size = _DRVUSB_GET_EP_DATA_SIZE(1);
```

_DRVUSB_SET_EP_TOG_BIT

原型

```
void     _DRVUSB_SET_EP_TOG_BIT (
   uint32_t     u32EPId,
   int32_t      bData0
)
```

描述

为指定的端点指定 Data0 或者 Data1。在 Host 应答 IN 令牌之后，该位会自动切换。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

bData0 [in]

为 IN 传输指定 DATA0 或 DATA1。TRUE：DATA0；FALSE：DATA1

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To set the toggle bit as DATA0 for endpoint identity 1 */
_DRVUSB_SET_EP_TOG_BIT(1, TRUE);
```

_DRVUSB_SET_EVENT_FLAG

原型

```
void _DRVUSB_SET_EVENT_FLAG (
   uint32_t     u32Data
);
```

描述

写中断事件标志寄存器，以清除中断标志。中断事件标志写 1 清零。

参数

u32Data [in]

指定要清除的事件标志。可以是

事件	值	描述
EVF_SETUP	0x80000000	Setup 令牌事件
EVF_EPTF5	0x00200000	端点 5 的 USB 事件
EVF_EPTF4	0x00100000	端点 4 的 USB 事件
EVF_EPTF3	0x00080000	端点 3 的 USB 事件
EVF_EPTF2	0x00040000	端点 2 的 USB 事件
EVF_EPTF1	0x00020000	端点 1 的 USB 事件
EVF_EPTF0	0x00010000	端点 0 的 USB 事件
EVF_WAKEUP	0x00000008	唤醒事件
EVF_FLDET	0x00000004	悬空检测事件
EVF_USB	0x00000002	USB 事件，包括端点事件和 Setup 事件
EVF_BUS	0x00000001	USB 总线事件

头文件

Driver/DrvUsb.h

返回值

无

示例

```
_DRVUSB_SET_EVENT_FLAG(EVF_BUS); /* Clear USB bus event */
_DRVUSB_SET_EVENT_FLAG(EVF_BUS | EVF_FLD); /* Clear USB bus event and
float-detection event */
```

_DRVUSB_GET_EVENT_FLAG

原型

```
uint32_t
_DRVUSB_GET_EVENT_FLAG (void);
```

描述

获取中断事件标志。

参数

无

头文件

Driver/DrvUsb.h

返回值

返回 EVF 寄存器的值。详细的事件信息请参考_DRVUSB_GET_EVENT_FLAG ()

示例

```
u32Events = _DRVUSB_GET_EVF(); /* Get events */
```

_DRVUSB_CLEAR_EP_STALL

原型

```
void _DRVUSB_CLEAR_EP_STALL (
    uint32_t      u32EPId
);
```

描述

停止强制端点响应 STALL。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

无

示例

```
_DRVUSB_CLEAR_EP_STALL(1);/* Clear the STALL of endpoint identity 1 */
```

_DRVUSB_TRIG_EP_STALL

原型

```
void _DRVUSB_TRIG_EP_STALL (
    uint32_t      u32EPId
);
```

描述

强制端点 EPx(x = 0 ~ 5)响应 STALL。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

无

示例

_DRVUSB_TRIG_EP_STALL (1); /* Force to STALL endpoint identity 1 */

_DRVUSB_CLEAR_EP_DSQ_SYNC

原型

```
void _DRVUSB_CLEAR_EP_DSQ_SYNC (
    uint32_t      u32EPId
);
```

描述

清除端点切换位为 DATA0，即强制切换位为 DATA0。从主机接收到 IN 令牌应答之后，该位自动切换。

参数

u32EPId [in]

端点号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Force the toggle bit of endpoint identity 2 to be DATA0 */
_DRVUSB_CLEAR_EP_DSQ_SYNC (2);
```

_DRVUSB_SET_CFG

原型

```
void _DRVUSB_SET_CFG (
        uint32_t      u32CFGNum,
        uint32_t      u32Data
```

);

描述

配置 USB CFG 寄存器。

参数

u32CFGNum [in]

CFG 寄存器编号(有效值：0 ~ 5)

u32Data [in]

为 CFG 寄存器指定设定值

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Set USB CFG2 control register as 0x3 */
_DRVUSB_SET_CFG (2, 0x3);
```

_DRVUSB_GET_CFG

原型

```
uint32_t
_DRVUSB_GET_CFG (
    uint32_t      u32CFGNum
);
```

描述

获取 USB CFG 寄存器当前值。

参数

u32CFGNum [in]

CFG 寄存器编号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

返回指定的 CFG 寄存器的值

示例

```
/* Get the setting of USB CFG2 control register */
u32Cfg = _DRVUSB_GET_CFG (3);
```

_DRVUSB_SET_FADDR

原型

```
void _DRVUSB_SET_FADDR (
    uint32_t      u32Addr
)
```

描述

设定 USB 设备地址，有效地址是 0~127。

参数

u32Addr [in]

USB 设备地址，可以是 0~127

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Set the USB devcie address as 3 */
_DRVUSB_SET_FADDR (3);
```

_DRVUSB_GET_FADDR

原型

```
uint32_t
_DRVUSB_GET_FADDR (void)
```

描述

取得 USB 设备地址

参数

无

头文件

Driver/DrvUsb.h

返回值

USB 设备地址

示例

```
/* Get USB devcie address */
u32Addr = _DRVUSB_GET_FADDR ();
```

_DRVUSB_GET_EPSTS

原型

```
uint32_t
_DRVUSB_GET_EPSTS (void)
```

描述

获取端点状态寄存器(EPSTS)的值。该状态寄存器可以用于标示 USB 事件的详细信息。关于 EPSTS 的详细信息，请参考 NuMicro Technical Reference Manual。

参数

无

头文件

Driver/DrvUsb.h

返回值

STS 寄存器的值

示例

```
/* Get USB STS register value */
u32Reg = _DRVUSB_GET_EPSTS();
```

_DRVUSB_SET_CFGP

原型

```
void _DRVUSB_SET_CFGP(
    uint8_t     u8CFGPNum,
    uint32_t    u32Data
);
```

描述

设置额外配置寄存器(CFGP)。CFGP 寄存器可以用于 STALL 端点和清除端点就绪标志。

CFGP[1]：STALL 控制位。设为 1 来强制端点响应 STALL

CFGP[0]：就绪标志，写 1 清除

参数

u8CFGPNum[in]

CFGP 寄存器编号(有效值：0 ~ 5)

u32Data [in]

指定 CFGP 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL the endpoint identity 1. */
_DRVUSB_SET_CFGP(1, 0x2);
```

_DRVUSB_GET_CFGP

原型

```
uint32_t
_DRVUSB_GET_CFGP (
    uint32_t      u32CFGPNum
);
```

描述

取得 CFGP 寄存器的值。

参数

u8CFGPNum[in]

CFGP 寄存器编号(有效值：0 ~ 5)

头文件

Driver/DrvUsb.h

返回值

CFGP 寄存器的值

示例

```
/* Get the register value of CFG1 */
_DRVUSB_GET_CFGP(1);
```

_DRVUSB_ENABLE_USB

原型

void _DRVUSB_ENABLE_USB (void)

描述

使能 USB, PHY，使用远程唤醒

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Enable USB, PHY and remote wakeup. */
_DRVUSB_ENABLE_USB();
```

_DRVUSB_DISABLE_USB

原型

void _DRVUSB_DISABLE_USB (void)

描述

关闭 USB, PHY，但是仍然使能远程唤醒。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Disable USB, PHY but still enable remote wakeup. */
```

_DRVUSB_DISABLE_USB();

_DRVUSB_DISABLE_PHY

原型

void _DRVUSB_DISABLE_PHY (void)

描述

禁止 PHY 和远程唤醒。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

/* Disable PHY and remote wakeup. */

_DRVUSB_DISABLE_ PHY();

_DRVUSB_ENABLE_SE0

原型

void _DRVUSB_ENABLE_SE0 (void)

描述

强制 USB PHY 驱动 SE0。可以用于模拟拔出事件，以使 Host 重新连接设备。更多关于 SE0 的信息请参考 USB 标准。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

/* Force bus to be SE0 state */

```
_DRVUSB_ENABLE_SE0();
```

_DRVUSB_DISABLE_SE0

原型

void _DRVUSB_DISABLE_SE0 (void)

描述

停止驱动 SE0。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* Stop to drive SE0 state to USB bus */
_DRVUSB_DISABLE_SE0();
```

_DRVUSB_SET_CFG_EP0

原型

```
void _DRVUSB_SET_CFG_EP0 (
    uint32_t      u32Data
)
```

描述

端点 0 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP0 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 0 */
_DRVUSB_SET_CFG_EP0(0x2);
```

_DRVUSB_SET_CFG_EP1

原型

```
void _DRVUSB_SET_CFG_EP1 (
    uint32_t      u32Data
)
```

描述

端点 1 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP1 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 1 */
_DRVUSB_SET_CFG_EP1(0x2);
```

_DRVUSB_SET_CFG_EP2

原型

```
void _DRVUSB_SET_CFG_EP2 (
    uint32_t      u32Data
)
```

描述

端点 2 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP2 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 2 */
_DRVUSB_SET_CFG_EP2(0x2);
```

_DRVUSB_SET_CFG_EP3

原型

```
void _DRVUSB_SET_CFG_EP3 (
    uint32_t      u32Data
)
```

描述

端点 3 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP3 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 3 */
_DRVUSB_SET_CFG_EP3(0x2);
```

_DRVUSB_SET_CFG_EP4

原型

```
void _DRVUSB_SET_CFG_EP4 (
    uint32_t      u32Data
)
```

描述

端点 4 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP4 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 4 */
_DRVUSB_SET_CFG_EP4(0x2);
```

_DRVUSB_SET_CFG_EP5

原型

```
void _DRVUSB_SET_CFG_EP5 (
    uint32_t      u32Data
)
```

描述

端点 5 的 Stall 控制和清除 In/Out 就绪标志。CFGP 寄存器的位定义请参考 _DRVUSB_SET_CFGP()。

参数

u32Data [in]

指定 CFGP5 寄存器的值来 STALL 端点或清除就绪标志

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To STALL endpoint identity 5 */
_DRVUSB_SET_CFG_EP5(0x2);
```

14.7. 函数

DrvUSB_GetVersion

原型

```
uint32_t
DrvUSB_GetVersion (void);
```

描述

获取该模块版本号.

参数

无

头文件

Driver/DrvUsb.h

返回值

版本号：

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

示例

```
/* To get module"s version */
u32Version = DrvUSB_GetVersion();
```

DrvUSB_Open

原型

```
int32_t
DrvUSB_Open (
    void *        pVoid
)
```

描述

这个函数可以用来复位 USB 控制器，初始化 USB 端点、中断和 USB 驱动结构。如果在

USB 驱动打开之前，USB 已经连接，函数 DrvUSB_Open 会调用相关的处理函数。用户在调用 DrvUSB_Open 之前必须填好结构 sEpDescription 和 g_sBusOps。

sEpDescription：

sEpDescription 结构定义如下:

```
typedef struct
{
    //比特 7 指示方向, 1: input; 0: output
    uint32_t  u32EPAddr;
    uint32_t  u32MaxPacketSize;
    uint8_t * u8SramBuffer;
}S_DRVUSB_EP_CTRL;
```

这个结构用来设定端点号，最大包尺寸和指定硬件端点传输缓存地址。NUC100 系列 USB 控制器有 6 个硬件端点可用.

g_sBusOps:

g_sBusOps 结构定义如下:

```
typedef struct
{
    PFN_DRVUSB_CALLBACK        apfnCallback;
    void *                     apCallbackArgu;
}S_DRVUSB_EVENT_PROCESS
```

可以用来安装 USB 总线事件处理函数，例如：

```
/* 总线事件回调 */
S_DRVUSB_EVENT_PROCESS g_sBusOps[6] =
{
    {NULL, NULL},                   /* 连接事件回调函数 */
    {NULL, NULL},                   /* 脱离事件回调函数 */
    {DrvUSB_BusResetCallback, &g_HID_sDevice}, /* 总线复位事件回调函数*/
    {NULL, NULL},                   /* 总线暂停事件回调函数*/
    {NULL, NULL},                   /* 总线重新开始事件回调函数*/
    {DrvUSB_CtrlSetupAck, &g_HID_sDevice},    /* setup 事件回调函数*/
};
```

参数

pVoid

NULL	无
Callback function	如果 pVoid 非空，那么它是 USB 中断回调函数指针，由 USB 中断处理函数在调用 DrvUSB_PreDispatchEvent 函数之后调用

头文件

Driver/DrvUsb.h

返回值

E_SUCCESS：成功

示例

```
/* To open USB device */
i32Ret = DrvUSB_Open(0);
if(i32Ret != E_SUCCESS)
   return i32Ret;
```

DrvUSB_Close

原型

```
void     DrvUSB_Close (void);
```

描述

关闭 USB 控制器并且禁止 USB 中断。

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To close USB device */
DrvUSB_Close();
```

DrvUSB_PreDispatchEvent

原型

```
void DrvUSB_PreDispatchEvent(void);
```

描述

基于 EVF 寄存器的值，预分派事件。

参数

无

头文件

Driver/DrvUsb.h

返回值

无

示例

```
/* To pre dispatch USB device events at IRQ handler */
USBD_IRQHandler()
{
   DrvUSB_PreDispatchEvent();
}
```

DrvUSB_DispatchEvent

原型

void DrvUSB_DispatchEvent(void)

描述

转发杂项和端点事件。杂项事件包括连接/脱离/总线复位/总线暂停/总线重新开始和 setup ACK, 杂项事件处理函数由结构 g_sBusOps[]定义。在使用 USB 驱动之前，用户必须提供 g_sBusOps[]。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* To dispatch USB events to handle them by related callback funcitons. */
DrvUSB_DispatchEvent();
```

DrvUSB_IsData0

原型

int32_t DrvUSB_IsData0(uint32_t u32EpId)

描述

检查当前 DATA 是否 DATA0。如果返回 FALSE，当前 DATA 是 DATA1。

参数

u32EpId

硬件端点索引。可以是 0~5

头文件

Driver/DrvUSB.h

返回值

TRUE　　当前数据包使用 DATA0

FALSE　　当前数据包使用 DATA1

示例

```
/* Get toggle bit of endpoint identity 2 */
if(DrvUSB_IsData0(2) )
{
  /* The toggle bit of endpoint identity 2 is DATA0 */
}
```

DrvUSB_GetUsbState

原型

E_DRVUSB_STATE DrvUSB_GetUsbState(void)

描述

取得当前 USB 状态 E_DRVUSB_STATE。状态列表如下：

USB 状态	描述
eDRVUSB_DETACHED	USB 设备已经脱离主机.
eDRVUSB_ATTACHED	USB 设备已经连接到主机.
eDRVUSB_POWERED	USB 已经上电
eDRVUSB_DEFAULT	USB 处于正常状态.
eDRVUSB_ADDRESS	USB 处于 ADDRESS 状态
eDRVUSB_CONFIGURED	USB 处于 CONFIGURATION 状态
eDRVUSB_SUSPENDED	USB 暂停

参数

无

头文件

Driver/DrvUSB.h

返回值

当前 USB 状态.

示例

```
/* Get current USB state */
eUsbState = DrvUSB_GetUsbState();
if (eUsbState == eDRVUSB_DETACHED)
{
  /* USB unplug */
}
```

DrvUSB_SetUsbState

原型

void DrvUSB_SetUsbState(E_DRVUSB_STATE eUsbState)

描述

改变当前 USB 的状态。关于可用状态，请参考 DrvUSB_GetUsbState.

参数

eUsbState

USB 状态

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Set current USB state */
DrvUSB_SetUsbState(eDRVUSB_DETACHED);
```

DrvUSB_GetEpIdentity

原型

uint32_t DrvUSB_GetEpIdentity(uint32_t u32EpNum, uint32_t u32EpAttr)

描述

根据端点号和方向取得端点索引。端点索引可以用来定位 USB 硬件的端点源。端点索引可以是 0 ~ 5。端点号由软件分配，根据 USB 标准可以是 0 ~ 15。主机通过端点号访问 USB 设备。

参数

u32EpNum

端点号(0 ~ 15)

u32EpAttr

端点属性。可以是 EP_INPUT 或者 EP_OUTPUT

头文件

Driver/DrvUSB.h

返回值

0~5　指定的端点地址对应的端点索引

otherwise　根据输入的端点地址不能得到相应的端点索引

示例

```
/* Get the hardware endpoint identity of USB OUT endpoint 3 */
u32EpId = DrvUSB_GetEpIdentity(3, EP_OUTPUT);
```

DrvUSB_GetEpId

原型

uint32_t DrvUSB_GetEpId(uint32_t u32EpNum)

描述

根据端点地址得到端点索引。参数“u32EpNum”和 DrvUSB_GetEpIdentity 的参数不同，因为参数“u32EpNum”的比特 7 包含方向信息。例如：0x81。如果比特 7 是高，表示这个端点是 EP_INPUT 的，否则是 EP_OUTPUT 的。

参数

u32EpNum

比特 7 带方向信息的端点地址

头文件

Driver/DrvUSB.h

返回值

0~5　指定的端点地址对应的端点索引

otherwise　根据输入的端点地址不能得到相应的端点索引

示例

```
/* Get the hardware endpoint identity of USB IN endpoint 4 */
u32EpId = DrvUSB_GetEpIdentity(0x84);
```

DrvUSB_DataOutTrigger

原型

int32_t DrvUSB_DataOutTrigger(uint32_t u32EpNum, uint32_t u32Size)

描述

写寄存器 MXPLD 来触发数据输出就绪标志。这表示相应的端点缓冲区就绪，可以接收输出的数据包。

参数

u32EpNum

端点号(0 ~ 15)

u32Size

想从 USB 接收的最大包大小

头文件

Driver/DrvUSB.h

返回值

0　　成功

<0　根据输入的端点地址不能得到相应的端点索引

示例

```
/* Trigger endpoint number 2 to receive OUT packet of host and the maximum packet size is
64 bytes */
DrvUSB_DataOutTrigger(2, 64);
```

DrvUSB_GetOutData

原型

uint8_t * DrvUSB_GetOutData(uint32_t u32EpNum, uint32_t *u32Size)

描述

这个函数将返回端点 u32EpNum 的输出 USB SRAM 缓存指针。用户可以用这个指针来得到输出数据包的数据。

参数

u32EpNum

端点号(0 ~ 15)

u32Size

从 USB 收到的数据包大小

头文件

Driver/DrvUSB.h

返回值

返回 USB SRAM 地址

示例

```
/* Get the buffer address and size of received data of endpoint number 2 */
pu8EpBuf = DrvUSB_GetOutData(2, &u32Size);
```

DrvUSB_DataIn

原型

int32_t DrvUSB_DataIn(uint32_t u32EpNum, const uint8_t * u8Buffer, uint32_t u32Size)

描述

触发数据发送就绪标志，从主机收到 IN 令牌以后，USB 控制器将发送数据给主机。如果 u8Buffer == NULL && u32Size == 0 则总是发送数据长度是 0 的 DATA1，否则交替发送 DATA0 和 DATA1。

参数

u32EpNum

端点号(0 ~ 15)

u8Buffer

DATA IN 令牌的数据缓存

u32Size

数据缓存大小

头文件

Driver/DrvUSB.h

返回值

E_SUCCESS	成功
E_DRVUSB_SIZE_TOO_LONG	u32Size 大于设定的最大包大小

示例

```
/* Prepare 2 bytes data for endpoint number 0 IN transaction. */
DrvUSB_DataIn(0, au8Data, 2);
```

DrvUSB_BusResetCallback

原型

void DrvUSB_BusResetCallback(void * pVoid)

描述

总线复位处理函数。收到总线复位信号之后，这个函数将被调用。它将复位 USB 地址，设定 SETUP 缓存地址并初始化端点。

参数

pVoid

由结构 g_sBusOps[]传递的参数.

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* bus event call back */
S_DRVUSB_EVENT_PROCESS g_sBusOps[6] =
{
  {NULL, NULL}, /* attach event callback */
  {NULL, NULL}, /* detach event callback */
  {DrvUSB_BusResetCallback, &g_HID_sDevice}, /* bus reset event callback */
  {NULL, NULL}, /* bus suspend event callback */
  {NULL, NULL}, /* bus resume event callback */
  {DrvUSB_CtrlSetupAck, &g_HID_sDevice}, /* setup event callback */
};
```

DrvUSB_InstallClassDevice

原型

void * DrvUSB_InstallClassDevice(S_DRVUSB_CLASS *sUsbClass)

描述

注册 USB 设备类到 USB 驱动。

参数

sUsbClass

USB 设备类结构指针.

头文件

Driver/DrvUSB.h

返回值

返回 USB 驱动指针

示例

```
/* Register USB class device to USB driver. */
g_HID_sDevice.device = (void *)DrvUSB_InstallClassDevice(&sHidUsbClass);
```

DrvUSB_InstallCtrlHandler

原型

```
int32_t DrvUSB_InstallCtrlHandler(
    void *                          *device,
    S_DRVUSB_CTRL_CALLBACK_ENTRY    *psCtrlCallbackEntry,
    uint32_t                        u32RegCnt
)
```

描述

注册控制管道处理函数，包括对 Standard/Vendor/Class 命令的处理。每个命令包括对 SETUP ACK , IN ACK, OUT ACK 的处理函数。

参数

device

USB 设备驱动指针

psCtrlCallbackEntry

处理函数结构指针

u32RegCnt

处理函数结构大小

头文件

Driver/DrvUSB.h

返回值

0	成功
E_DRVUSB_NULL_POINTER	处理函数结构指针为空

示例

```
/* Register ctrl pipe handler. */
i32Ret = DrvUSB_InstallCt rlHandler(g_HID_sDevice.device, g_asCtrlCallbackEntry,
sizeof(g_asCtrlCallbackEntry) / sizeof(g_asCtrlCallbackEntry[0]));
```

DrvUSB_CtrlSetupAck

原型

void DrvUSB_CtrlSetupAck(void * pArgu)

描述

当 SETUP ack 中断发生时，这个函数将被调用。它将根据收到的命令调用 DrvUSB_InstallCtrlHandler 注册的 SETUP 处理函数。

参数

pArgu

由结构 g_sBusOps[]传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* bus event call back */
S_DRVUSB_EVENT_PROCESS g_sBusOps[6] =
{
  {NULL, NULL}, /* attach event callback */
  {NULL, NULL}, /* detach event callback */
  {DrvUSB_BusResetCallback, &g_HID_sDevice}, /* bus reset event callback */
  {NULL, NULL}, /* bus suspend event callback */
  {NULL, NULL}, /* bus resume event callback */
```

```
    {DrvUSB_CtrlSetupAck, &g_HID_sDevice}, /* setup event callback */
};
```

DrvUSB_CtrlDataInAck

原型

void DrvUSB_CtrlDataInAck(void * pArgu)

描述

当 IN ack 中断发生时，这个函数将被调用。它将根据收到的命令调用 DrvUSB_InstallCtrlHandler 注册的 IN ACK 处理函数。

参数

pArgu

由结构 g_sBusOps[]传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* USB event call back */
S_DRVUSB_EVENT_PROCESS g_sUsbOps[12] =
{
    {DrvUSB_CtrlDataInAck, &g_HID_sDevice},/* ctrl pipe0 (EP address 0) In ACK callback */
    {DrvUSB_CtrlDataOutAck, &g_HID_sDevice},/* ctrl pipe0 (EP address 0) Out ACK callback */
    {HID_IntInCallback, &g_HID_sDevice}, /* EP address 1 In ACK callback */
    {NULL, NULL}, /* EP address 1 Out ACK callback */
    {NULL, NULL}, /* EP address 2 In ACK callback */
    {HID_IntOutCallback, &g_HID_sDevice}, /* EP address 2 Out ACK callback */
    {NULL, NULL}, /* EP address 3 In ACK callback */
    {NULL, NULL}, /* EP address 3 Out ACK callback */
    {NULL, NULL}, /* EP address 4 In ACK callback */
    {NULL, NULL}, /* EP address 4 Out ACK callback */
    {NULL, NULL}, /* EP address 5 In ACK callback */
    {NULL, NULL}, /* EP address 5 Out ACK callback */
```

```
};
```

DrvUSB_CtrlDataOutAck

原型

void DrvUSB_CtrlDataOutAck(void * pArgu)

描述

当 OUT ack 中断发生时，这个函数将被调用。它将根据收到的命令调用由 DrvUSB_InstallCtrlHandler 注册的 OUT ACK 处理函数。

参数

pArgu

由结构 g_sBusOps[]传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* USB event call back */
S_DRVUSB_EVENT_PROCESS g_sUsbOps[12] =
{
  {DrvUSB_CtrlDataInAck , &g_HID_sDevice},/* ctrl pipe0 (EP address 0) In ACK
   callback */
  {DrvUSB_CtrlDataOutAck , &g_HID_sDevice},/* ctrl pipe0 (EP address 0) Out ACK
   callback */
  {HID_IntInCallback , &g_HID_sDevice}, /* EP address 1 In ACK callback */
  {NULL, NULL}, /* EP address 1 Out ACK callback */
  {NULL, NULL}, /* EP address 2 In ACK callback */
  {HID_IntOutCallback , &g_HID_sDevice},/* EP address 2 Out ACK callback */
  {NULL, NULL}, /* EP address 3 In ACK callback */
  {NULL, NULL}, /* EP address 3 Out ACK callback */
  {NULL, NULL}, /* EP address 4 In ACK callback */
  {NULL, NULL}, /* EP address 4 Out ACK callback */
  {NULL, NULL}, /* EP address 5 In ACK callback */
  {NULL, NULL}, /* EP address 5 Out ACK callback */
};
```

DrvUSB_CtrlDataInDefault

原型

void DrvUSB_CtrlDataInDefault(void * pVoid)

描述

IN ACK 缺省处理函数，用于返回下一次 OUT 令牌的 ACK。

参数

pVoid

由函数 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* If no control data IN callback installed, just use default one */
if (psEntry->pfnCtrlDataInCallback == NULL)
  psEntry->pfnCtrlDataInCallback = DrvUSB_CtrlDataInDefault;
```

DrvUSB_CtrlDataOutDefault

原型

void DrvUSB_CtrlDataOutDefault(void * pVoid)

描述

OUT ACK 缺省处理函数。收到主机的下一次 IN 令牌之后，返回 0 长度的数据包给主机

参数

pVoid

由函数 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* If no control data OUT callback installed, just use default one */
if (psEntry->pfnCtrlDataOutCallback == NULL)
    psEntry->pfnCtrlDataOutCallback = DrvUSB_CtrlDataOutDefault;
```

DrvUSB_Reset

原型

void DrvUSB_Reset(uint32_t u32EpNum)

描述

根据端点号恢复指定的 CFGx 和 CFGPx 寄存器的缺省值。

参数

u32EpNum

要复位的端点号

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Reset endpoint number 2 */
DrvUSB_Reset(2);
```

DrvUSB_ClrCtrlReady

原型

void DrvUSB_ClrCtrlReady(void)

描述

清除控制管道就绪标志。这个标志由写寄存器 MXPLD 来设定。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Clear control endpoint ready flag */
DrvUSB_ClrCtrlReady();
```

DrvUSB_ClrCtrlReadyAndTrigStall

原型

```
void DrvUSB_ClrCtrlReadyAndTrigStall(void);
```

描述

清除控制管道就绪标志(这个标志由写寄存器 MXPLD 来设定)，并发送 STALL。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Clear control pipe ready flag that was set by MXPLD and send STALL. */
DrvUSB_ClrCtrlReadyAndTrigStall();
```

DrvUSB_GetSetupBuffer

原型

```
uint32_t DrvUSB_GetSetupBuffer(void)
```

描述

取得 USB SRAM 的 setup 缓存地址，来读取接收到的 setup 包数据。

参数

无

头文件

Driver/DrvUSB.h

返回值

Setup 缓存地址

示例

```
/* Get setup buffer address of USB SRAM. */
SetupBuffer = (uint8_t *)DrvUSB_GetSetupBuffer();
```

DrvUSB_GetFreeSRAM

原型

uint32_t DrvUSB_GetFreeSRAM(void)

描述

取得 EP 在根据 DrvUSB_Open 中 sEpDescription[i].u32MaxPacketSize 分配之后，空闲的 USB SRAM 缓存地址。用户可以取得这个地址用于双缓存。

参数

无

头文件

Driver/DrvUSB.h

返回值

空闲的 USB SRAM 地址

示例

```
/* Get the base address of free USB SRAM */
u32BaseAddr = DrvUSB_GetFreeSRAM();
```

DrvUSB_EnableSelfPower

原型

void DrvUSB_EnableSelfPower(void)

描述

使能 USB 设备的自供电属性。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

/* Set a flag to note the USB device is self-power */

DrvUSB_EnableSelfPower();

DrvUSB_DisableSelfPower

原型

void DrvUSB_DisableSelfPower(void)

描述

禁止 USB 设备的自供电属性。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

/* Clear the flag to note the USB device is not self-power */

DrvUSB_ DisableSelfPower ();

DrvUSB_IsSelfPowerEnabled

原型

int32_t DrvUSB_IsSelfPowerEnabled(void)

描述

查看自供电是使能的还是禁止的。

参数

无

头文件

Driver/DrvUSB.h

返回值

TRUE	USB 设备是自供电的
FALSE	USB 设备是总线供电的

示例

```
/* Check if the USB device is self-power */
if(DrvUSB_IsSelfPowerEnabled())
{
   /* The USB device is self-power */
}
```

DrvUSB_EnableRemoteWakeup

原型

void DrvUSB_EnableRemoteWakeup(void)

描述

使能 USB 设备的远程唤醒属性。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Set the flag to note the USB device supports remote wakeup */
DrvUSB_EnableRemoteWakeup();
```

DrvUSB_DisableRemoteWakeup

原型

void DrvUSB_DisableRemoteWakeup(void)

描述

禁止远程唤醒属性。

参数

无

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Clear the flag to note the USB device doesn"t support remote wakeup */
DrvUSB_DisableRemoteWakeup();
```

DrvUSB_IsRemoteWakeupEnabled

原型

int32_t DrvUSB_IsRemoteWakeupEnabled (void)

描述

查看远程唤醒是使能的还是关闭的。

参数

无

头文件

Driver/DrvUSB.h

返回值

TRUE	USB 设备支持远程唤醒
FALSE	USB 设备不支持远程唤醒

示例

```
/* Check if the USB device supports remote wakeup. */
if(DrvUSB_ IsRemoteWakeupEnabled ())
{
   /* Remote wakeup enable flag is set */
}
```

DrvUSB_SetMaxPower

原型

int32_t DrvUSB_SetMaxPower(uint32_t u32MaxPower)

描述

配置最大电流，单位 2mA。MaxPower 的最大值是 0xFA (500mA), 缺省值是 0x32 (100mA)

参数

u32MaxPower

最大电流值

头文件

Driver/DrvUSB.h

返回值

0：成功

<0：最大值错误

示例

```
/* Set the maximum power is 150mA */
DrvUSB_SetMaxPower(75);
```

DrvUSB_GetMaxPower

原型

int32_t DrvUSB_GetMaxPower(void)

描述

取得当前最大电流值，单位 2mA, 也就是说 0x32 = 100mA.

参数

无

头文件

Driver/DrvUSB.h

返回值

最大电流值. (单位 2mA)

示例

```
/* Get the maximum power */
i32Power = DrvUSB_GetMaxPower();
```

DrvUSB_EnableUSB

原型

void DrvUSB_EnableUSB(S_DRVUSB_DEVICE *psDevice)

描述

使能 USB，PHY 和远程唤醒。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Enable USB, PHY and remote wakeup function. */
DrvUSB_EnableUSB(psDevice);
```

DrvUSB_DisableUSB

原型

void DrvUSB_DisableUSB(S_DRVUSB_DEVICE * psDevice)

描述

禁止 USB 和 PHY，但是保持远程唤醒功能开启。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Enable USB, PHY and remote wakeup function. */
DrvUSB_DisableUSB(psDevice);
```

DrvUSB_PreDispatchWakeupEvent

原型

void DrvUSB_PreDispatchWakeupEvent(S_DRVUSB_DEVICE *psDevice)

描述

预分派唤醒事件。这个函数是预留的。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

N/A

DrvUSB_PreDispatchFDTEvent

原型

void DrvUSB_PreDispatchFDTEvent(S_DRVUSB_DEVICE * psDevice)

描述

预分派插入/拔出事件。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Pre-dispatch float-detection event. */
DrvUSB_PreDispatchFDTEvent(&gsUsbDevice);
```

DrvUSB_PreDispatchBusEvent

原型

void DrvUSB_PreDispatchBusEvent(S_DRVUSB_DEVICE *psDevice)

描述

预分派总线事件。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Pre-dispatch bus event. */
DrvUSB_PreDispatchBusEvent(&gsUsbDevice);
```

DrvUSB_PreDispatchEPEvent

原型

void DrvUSB_PreDispatchEPEvent(S_DRVUSB_DEVICE * psDevice)

描述

预分派端点事件，包括 IN ACK/IN NAK/OUT ACK/ISO 端点事件。这个函数用来识别端点事件并记录它们，将来函数 DrvUSB_DispatchEPEvent()将做进一步处理。所有端点事件的缺省处理函数定义在 g_sUsbOps[]中。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Clear USB events individually instead of in total. Otherwise, incoming USB events may be
cleared mistakenly. Pre-dispatch USB event. */
DrvUSB_PreDispatchEPEvent(&gsUsbDevice);
```

DrvUSB_DispatchWakeupEvent

原型

void DrvUSB_DispatchWakeupEvent(S_DRVUSB_DEVICE *psDevice)

描述

分派唤醒事件。这个函数是预留的。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

N/A

DrvUSB_DispatchMiscEvent

原型

void DrvUSB_DispatchMiscEvent(S_DRVUSB_DEVICE * psDevice)

描述

分派杂项事件。这个事件包含连接/脱离/总线复位/总线暂停/总线重新开始和 setup ACK。杂项事件的处理函数定义在结构 g_sBusOps[]中。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

/* Parsing the MISC events and call relat ive handles */

DrvUSB_DispatchMiscEvent(&gsUsbDevice);

DrvUSB_DispatchEPEvent

原型

void DrvUSB_DispatchEPEvent(S_DRVUSB_DEVICE * psDevice)

描述

分派端点事件, 它处理由函数 DrvUSB_PreDispatchEPEvent()分派的事件。包括 IN ACK/IN NAK/OUT ACK/ISO end。端点事件的处理函数定义在结构 g_sUsbOps[]中。

参数

psDevice

USB 设备驱动指针

头文件

Driver/DrvUSB.h

返回值

无

示例

/* Parsing the endpoint events and call relat ive handlers */

DrvUSB_ DispatchEPEvent (&gsUsbDevice);

DrvUSB_CtrlSetupSetAddress

原型

void DrvUSB_CtrlSetupSetAddress(void * pVoid)

描述

用于设定地址命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/*ctrl pipe call back.*/
/*it will be call by DrvUSB_CtrlSetupAck, DrvUSB_CtrlDataInAck and
DrvUSB_CtrlDataOutAck*/
/*if in ack handler and out ack handler is 0, default handler will be called */
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
   {REQ_STANDARD, SET_ADDRESS, DrvUSB_CtrlSetupSetAddress,
   DrvUSB_CtrlDataInSetAddress, 0, &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupClearSetFeature

原型

void DrvUSB_CtrlSetupClearSetFeature(void * pVoid)

描述

用于清除特性命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
   {REQ_STANDARD, CLEAR_FEATURE, DrvUSB_CtrlSetupClearSetFeature, 0, 0,
   &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupGetConfiguration

原型

void DrvUSB_CtrlSetupGetConfiguration(void * pVoid)

描述

用于获取配置命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
  {REQ_STANDARD, GET_CONFIGURATION, DrvUSB_CtrlSetupGetConfiguration, 0,
  0,  &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupGetStatus

原型

void DrvUSB_CtrlSetupGetStatus(void * pVoid)

描述

用于获取状态命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
```

```
    {REQ_STANDARD, GET_STATUS, DrvUSB_CtrlSetupGetStatus, 0, 0,
    &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupGetInterface

原型

void DrvUSB_CtrlSetupGetInterface(void * pVoid)

描述

用于获取接口命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
    {REQ_STANDARD, GET_INTERFACE, DrvUSB_CtrlSetupGetInterface, 0, 0,
    &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupSetInterface

原型

void DrvUSB_CtrlSetupSetInterface(void * pVoid)

描述

用于设定接口命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
  {REQ_STANDARD, SET_INTERFACE, DrvUSB_CtrlSetupSetInterface, 0, 0,
  &g_HID_sDevice}
};
```

DrvUSB_CtrlSetupSetConfiguration

原型

void DrvUSB_CtrlSetupSetConfiguration(void * pVoid)

描述

用于设定配置命令的 Setup ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
  {REQ_STANDARD, SET_CONFIGURATION, DrvUSB_CtrlSetupSetConfiguration, 0,
  0,  &g_HID_sDevice}
};
```

DrvUSB_CtrlDataInSetAddress

原型

void DrvUSB_CtrlDataInSetAddress(void * pVoid)

描述

用于设定地址命令的 IN ACK 处理函数。

参数

pVoid

由 DrvUSB_InstallCtrlHandler 传递的参数

头文件

Driver/DrvUSB.h

返回值

无

示例

```
S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[] =
{ //request type,command,setup ack handler, in ack handler,out ack handler, parameter
  {REQ_STANDARD, SET_ADDRESS, DrvUSB_CtrlSetupSetAddress,
  DrvUSB_CtrlDataInSetAddress, 0, &g_HID_sDevice}
};
```

DrvUSB_memcpy

原型

void DrvUSB_memcpy(uint8_t *pi8Dest, uint8_t *pi8Src, uint32_t u32Size)

描述

推荐 USB 缓存使用字节访问，因为这个函数通过字节访问方式实现。

参数

pi8Dest

目标指针

pi8Src

源指针

u32Size

数据大小。单位是字节

头文件

Driver/DrvUSB.h

返回值

无

示例

```
/* Copy 64 bytes data from USB SRAM to SRAM */
DrvUSB_memcpy(0x20000800, 0x40060100, 64);
```

15. PDMA 驱动

15.1. PDMA 介绍

NuMicro™ NUC100 系列包含一个外设直接内存访问(PDMA)控制器，可以读/写内存或者读/写APB。PDMA 有 9 个 DMA 通道(外设到内存或者内存到外设或者内存到内存)。对于每个 PDMA 通道(PDMA CH0~CH8)，有一个 4 字节的缓存用于外设 APB IP 和内存之间的传输缓存。

软件可以通过禁止 PDMA[PDMACEN]来停止 PDMA 操作。CPU 可以通过软件轮询或者接收中断的方式来识别一次 PDMA 的完成。PDMA 控制器能增加源或者目标地址，也能固定源/目标地址。

15.2. PDMA 特性

PDMA 包含下面的特性：

- 增强型微控制器总线架构增强型高性能总线(AMBA AHB)主/从接口兼容，用于数据传输和寄存器读写
- PDMA 支持 32 比特源/目标寻址范围，地址增加/固定
- 多达 9 通道 PDMA，PDMA 通道号请参考附录 NuMicro™ NUC100 系列产品选型指导

15.3. 常量定义

常量名	值	描述
CHANNEL_OFFSET	0x100	PDMA 通道寄存器偏移量

15.4. 类型定义

E_DRVPDMA_CHANNEL_INDEX

枚举标识符	值	描述
eDRVPDMA_CHANNEL_0	0	PDMA 通道 0
eDRVPDMA_CHANNEL_1	1	PDMA 通道 1
eDRVPDMA_CHANNEL_2	2	PDMA 通道 2
eDRVPDMA_CHANNEL_3	3	PDMA 通道 3
eDRVPDMA_CHANNEL_4	4	PDMA 通道 4
eDRVPDMA_CHANNEL_5	5	PDMA 通道 5
eDRVPDMA_CHANNEL_6	6	PDMA 通道 6
eDRVPDMA_CHANNEL_7	7	PDMA 通道 7
eDRVPDMA_CHANNEL_8	8	PDMA 通道 8

E_DRVPDMA_DIRECTION_SELECT

枚举标识符	值	描述
eDRVPDMA_DIRECTION_INCREMENTED	0	源/目标地址增长
eDRVPDMA_ DIRECTION_FIXED	2	源/目标地址固定

E_DRVPDMA_TRANSFER_WIDTH

枚举标识符	值	描述
eDRVPDMA_WIDTH_32BITS	0	IP 到内存/内存到 IP 模式下，每次 PDMA 操作传输一个字
eDRVPDMA_WIDTH_8BITS	1	IP 到内存/内存到 IP 模式下，每次 PDMA 操作传输半个字
eDRVPDMA_WIDTH_16BITS	2	IP 到内存/内存到 IP 模式下，每次 PDMA 操作传输一个字节

E_DRVPDMA_INT_ENABLE

枚举标识符	值	描述
eDRVPDMA_TABORT	1	目标中止中断/标志
eDRVPDMA_BLKD	2	传输完成中断/标志

E_DRVPDMA_APB_DEVICE

枚举标识符	值	描述

eDRVPDMA_SPI0	0	PDMA 源/目的 APB 设备是 SPI0
eDRVPDMA_SPI1	1	PDMA 源/目的 APB 设备是 SPI1
eDRVPDMA_SPI2	2	PDMA 源/目的 APB 设备是 SPI2
eDRVPDMA_SPI3	3	PDMA 源/目的 APB 设备是 SPI3
eDRVPDMA_UART0	4	PDMA 源/目的 APB 设备是 UART0
eDRVPDMA_UART1	5	PDMA 源/目的 APB 设备是 UART1
eDRVPDMA_ADC	7	PDMA 源/目的 APB 设备是 ADC
eDRVPDMA_I2S	8	PDMA 源/目的 APB 设备是 I2S

E_DRVPDMA_APB_RW

枚举标识符	值	描述
eDRVPDMA_READ_APB	0	从 APB 设备读取数据到内存
eDRVPDMA_WRITE_APB	1	从内存写数据到 APB 设备

E_DRVPDMA_MODE

枚举标识符	值	描述
eDRVPDMA_MODE_MEM2MEM	0	PDMA 模式是内存到内存
eDRVPDMA_MODE_APB2MEN	1	PDMA 模式是 APB 设备到内存
eDRVPDMA_MODE_MEM2APB	2	PDMA 模式是内存到 APB 设备

15.5. 函数

DrvPDMA_Init

原型

void

DrvPDMA_Init (void);

描述

该函数用于使能 AHB PDMA 时钟。

参数

无

头文件

Driver/DrvPDMA.h

返回值

无

示例

```
/* Enable AHB PDMA engine clock */
DrvPDMA_Init();
```

DrvPDMA_Close

原型

void DrvPDMA_Close (void);

描述

这个函数可以用来禁止所有的 PDMA 通道时钟和 AHB PDMA 时钟。

参数

无

头文件

Driver/DrvPDMA.h

返回值

无

示例

```
/* Disable all PDMA channel clock and AHB PDMA clock */
DrvPDMA_Close();
```

DrvPDMA_CHEnableTransfer

原型

```
int32_t
DrvPDMA_CHEnablelTransfer(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来使能 PDMA 指定通道和指定通道数据读/写功能。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

示例

```
/* Enable PDMA channel0 and enable channel0 data read/write transfer */
DrvPDMA_CHEnableTransfer(eDRVPDMA_CHANNEL_0);
```

DrvPDMA_CHSoftwareReset

原型

```
int32_t
DrvPDMA_CHSoftwareReset(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来软件复位指定的通道。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

Note

该函数将会复位指定的通道的内部状态机和指针。控制寄存器中的内容不会被清除。

示例

```
/* Software reset PDMA channel0 and get returned value */
int32_t i32RetVal_CH0SoftwareReset;
i32RetVal_CH0SoftwareReset =
DrvPDMA_CH0SoftwareReset(eDRVPDMA_CHANNEL_0);
```

DrvPDMA_Open

原型

```
int32_t
DrvPDMA_Open(
    E_DRVPDMA_CHANNEL_INDEX sChannel,
    STR_PDMA_T *sParam
);
```

描述

这个函数可以用来配置 PDMA 设定值。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

sParam [in]

配置 PDMA.的结构体参数，

包括

sSrcCtrl.u32Addr：源地址(必须是字对齐)

sSrcCtrl.eAddrDierction：源地址方向

eDRVPDMA_DIRECTION_INCREMENTED：源地址增长

eDRVPDMA_ DIRECTION_FIXED：源地址固定

sDestCtrl.u32Addr：目的地址(必须是字对齐)

sSrcCtrl.eAddrDierction：目的地址方向。可以是

eDRVPDMA_DIRECTION_INCREMENTED：目的地址增长

eDRVPDMA_ DIRECTION_FIXED：目的地址固定

u8TransWidth：外围设备传输宽度。该位域只有在操作模式设定为 APB 到内存或者内存到 APB 时才有意义。传输宽度可以是：

eDRVPDMA_WIDTH_8BITS/ eDRVPDMA_WIDTH_16BITS/ eDRVPDMA_WIDTH_32BITS

eDRVPDMA_WIDTH_8BITS：每次 PDMA 操作传输一个字节(8 比特)

eDRVPDMA_WIDTH_16BITS：每次 PDMA 操作传输半个字(16 比特)

eDRVPDMA_WIDTH_32BITS：每次 PDMA 操作传输一个字(32 比特)

u8Mode：操作模式

eDRVPDMA_MODE_MEM2MEM：内存到内存模式

eDRVPDMA_MODE_APB2MEM：APB 到内存模式

eDRVPDMA_MODE_MEM2 APB：内存到 APB 模式

i32ByteCnt：PDMA 传输字节数

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

示例

```
/*---------------------------------------------------------------------------------------*/
/* Set PDMA channel1 to UART1 TX---------------------------------------- */
/* Set PDMA transfer done callback function and trigger PDMA function. */
/*---------------------------------------------------------------------------------------*/
/* PDMA Setting */
UARTPort = UART1_BASE;
DrvPDMA_SetCHForAPBDevice(eDRVPDMA_CHANNEL_1,eDRVPDMA_UART1,eDR
VPDMA_WRITE_APB);
/* CH1 TX Setting */
sPDMA.sSrcCtrl.u32Addr = (uint32_t)SrcArray;
sPDMA.sDestCtrl.u32Addr = UARTPort;
sPDMA.u8TransWidth = eDRVPDMA_WIDTH_8BITS;
sPDMA.u8Mode = eDRVPDMA_MODE_MEM2APB;
sPDMA.sSrcCtrl.eAddrDirect ion = eDRVPDMA_DIRECTION_INCREMENTED;
sPDMA.sDestCtrl.eAddrDirection = eDRVPDMA_DIRECTION_FIXED;
sPDMA.i32ByteCnt = UART_TEST_LENGTH;
DrvPDMA_Open(eDRVPDMA_CHANNEL_1,&sPDMA);
/* Enable INT */
DrvPDMA_EnableInt(eDRVPDMA_CHANNEL_1, eDRVPDMA_BLKD );
/* Install Callback function */
DrvPDMA_InstallCallBack(eDRVPDMA_CHANNEL_ 1,eDRVPDMA_BLKD,(PFN_DRV
PDMA_CALLBACK));
/* Enable UART PDMA and Trigger PDMA specified Channel */
DrvPDMA_CHEnableTransfer(eDRVPDMA_CHANNEL_ 1);
```

DrvPDMA_ClearIntFlag

原型

```
void
DrvPDMA_ClearIntFlag(
    E_DRVPDMA_CHANNEL_INDEX eChannel,
    E_DRVPDMA_INT_FLAG eIntFlag
) ;
```

描述

这个函数可以用来清除指定通道的中断状态。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntFlag [in]

中断源：

eDRVPDMA_TABORT：读/写目标中止

eDRVPDMA_BLKD：块传输完成

头文件

Driver/DrvPDMA.h

返回值

无

示例

```
/* Clear channel0 block t ransfer done interrupt flag. */
DrvPDMA_ClearIntFlag(eDRVPDMA_CHANNEL_0, eDRVPDMA_BLKD_FLAG);
/* Clear channel1 read/write target abort interrupt flag */
DrvPDMA_ClearIntFlag(eDRVPDMA_CHANNEL_1, eDRVPDMA_TABORT);
```

DrvPDMA_PollInt

原型

```
int32_t
DrvPDMA_PollInt(
    E_DRVPDMA_CHANNEL_INDEX eChannel,
    E_DRVPDMA_INT_FLAG eIntFlag
```

) ;

描述

这个函数可以用来轮询通道中断状态

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntFlag [in]

中断源：

eDRVPDMA_TABORT：读/写目标中止

eDRVPDMA_BLKD：块传输完成

头文件

Driver/DrvPDMA.h

返回值

True：中断状态置位

False：中断状态清除

示例

```
/* Get Channel 5 transfer done interrupt status */
int32_t i32Channel5TransferDone;
/* Enable INT */
DrvPDMA_EnableInt(eDRVPDMA_CHANNEL_ 5, eDRVPDMA_BLKD );
…
/* Check channel5 transfer done interrupt flag */
if(DrvPDMA_PollInt(eDRVPDMA_CHANNEL_5, eDRVPDMA_BLKD_FLAG)==TRUE)
   printf(“Channel5 block t ransfer done interrupt flag is set!!\n”)
else
   printf(“Channel5 block t ransfer done interrupt flag is not set!!\n”)
```

DrvPDMA_SetAPBTransferWidth

原型

```
int32_t
DrvPDMA_SetAPBTransferWidth(
   E_DRVPDMA_CHANNEL_INDEX eChannel,
   E_DRVPDMA_TRANSFER_WIDTH eTransferWidth
```

) ;

描述

这个函数可以用来设定指定通道的 APB 传输宽度

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eTransferWidth [in]

eDRVPDMA_WIDTH_32BITS：每次 PDMA 操作传输一个字(32 比特)

eDRVPDMA_WIDTH_8BITS：每次 PDMA 操作传输一个字节(8 比特)

eDRVPDMA_WIDTH_16BITS：每次 PDMA 操作传输半个字(16 比特)

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

Note

该函数仅在 PDMA 操作模式是 APB 到内存活着内存到 APB 时才有意义。

示例

```
/* Set chaneel 7 peripheral bus width to 8 bits.*/
DrvPDMA_SetAPBTransferWidth(eDRVPDMA_CHANNEL_7,
eDRVPDMA_WIDTH_8BITS);
```

DrvPDMA_SetCHForAPBDevice

原型

```
int32_t
DrvPDMA_SetCHForAPBDevice(
    E_DRVPDMA_CHANNEL_INDEX   eChannel,
    E_DRVPDMA_APB_DEVICE          eDevice,
    E_DRVPDMA_APB_RW          eRWAPB
) ;
```

描述

这个函数可以用来为 APB 设备选择 PDMA 通道

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eDevice [in]

APB 设备通道。包括：

eDRVPDMA_SPI0~3, eDRVPDMA_UART0~1, eDRVPDMA_ADC, eDRVPDMA_I2S

eRWAPB [in]

PDMA 传输方向：

eDRVPDMA_WRITE_APB：PDMA 从内存传输数据到指定的 APB

eDRVPDMA_READ_APB：PDMA 从指定的 APB 传输数据到内存

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口

E_DRVPDMA_FALSE_INPUT：无效 APB 设备

示例

```
/*Set PDMA channel1 to UART1 TX port*/
DrvPDMA_SetCHForAPBDevice(eDRVPDMA_CHANNEL_1,eDRVPDMA_UART1,eDRVPDMA_WRITE_APB);
/*Set PDMA channel0 to SPI0 RX port*/
DrvPDMA_SetCHForAPBDevice(eDRVPDMA_CHANNEL_ 0,eDRVPDMA_SPI0,eDRVPDMA_READ_APB);
```

DrvPDMA_DisableInt

原型

```
int32_t
DrvPDMA_DisableInt(
    E_DRVPDMA_CHANNEL_INDEX eChannel,
    E_DRVPDMA_INT_ENABLE eIntSource
);
```

描述

这个函数可以用来禁止指定通道的中断

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntSource [in]

中断源：

eDRVPDMA_TABORT：读/写目标中止

eDRVPDMA_BLKD：块传输完成

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口

示例

```
/*Disable channel3 read/write target abort interrupt*/
DrvPDMA_DisableInt(eDRVPDMA_CHANNEL_3, eDRVPDMA_TABORT);
```

DrvPDMA_EnableInt

原型

```
int32_t
DrvPDMA_EnableInt(
   E_DRVPDMA_CHANNEL_INDEX eChannel,
   E_DRVPDMA_INT_ENABLE eIntSource
);
```

描述

这个函数可以用来使能指定通道的中断

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntSource [in]

中断源：

eDRVPDMA_TABORT：读/写目标中止

eDRVPDMA_BLKD：块传输完成

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

E_DRVPDMA_ERR_PORT_INVALID：无效端口

示例

```
/*Enable channel0 block transfer done interrupt.*/
DrvPDMA_EnableInt(eDRVPDMA_CHANNEL_0, eDRVPDMA_BLKD);
```

DrvPDMA_GetAPBTransferWidth

原型

```
int32_t
DrvPDMA_GetAPBTransferWidth(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来从指定的通道获取外围设备的传输宽度

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

0：每次 PDMA 操作传输一个字(32 比特)

1：每次 PDMA 操作传输一个字节(8 比特)

2：每次 PDMA 操作传输半个字(16 比特)

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

Note

该函数仅在 PDMA 操作模式是 APB 到内存活着内存到 APB 时才有意义。

示例

```
/*Get peripheral t ransfer width from channel3*/
int32_t i32Channel3APBTransferWidth;
i32Channel3APBTransferWidth =
DrvPDMA_GetAPBTransferWidth(eDRVPDMA_CHANNEL_3);
```

DrvPDMA_GetCHForAPBDevice

原型

```
int32_t
DrvPDMA_GetCHForAPBDevice(
    E_DRVPDMA_APB_DEVICE eDevice,
    E_DRVPDMA_APB_RW eRWAPB
);
```

描述

这个函数可以用来取得指定 APB 设备使用的 PDMA 通道

参数

eDevice [in]

APB 设备通道。包括：

eDRVPDMA_SPI0~3, eDRVPDMA_UART0~1, eDRVPDMA_ADC, eDRVPDMA_I2S

eRWAPB [in]

指定 APB 方向：

eDRVPDMA_READ_APB：APB 到内存

eDRVPDMA_WRITE_APB：内存到 APB

头文件

Driver/DrvPDMA.h

返回值

0：通道 0

1：通道 1

2：通道 2

3：通道 3

4：通道 4

5：通道 5

6：通道 6

7：通道 7

8：通道 8

E_DRVPDMA_FALSE_INPUT：参数错误

其他：保留

Note

如果 APB 设备没有被分配到任何一个通道，默认返回值是 15(0xF)

示例

```
/* Get UART0 RX PDMA channel*/
int32_t i32GetChannel4APBDevice;
i32GetChannel4APBDevice = DrvPDMA_GetCHForAPBDevice(eDRVPDMA_UART0,
eDRVPDMA_READ_APB);
```

DrvPDMA_GetCurrentDestAddr

原型

```
uint32_t
DrvPDMA_GetCurrentDestAddr(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来取得指定 PDMA 通道的当前目标地址

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

当前目标地址

Note

当前目标地址表示 PDMA 传输正在进行的目标地址

示例

```
/*Get Channel5 current destination address;*/
uint32_t u32Channel5CurDestAddr;
```

```
u32Channel5CurDestAddr =
DrvPDMA_GetCurrentDestAddr(eDRVPDMA_CHANNEL_5);
```

DrvPDMA_GetCurrentSourceAddr

原型

```
uint32_t
DrvPDMA_GetCurrentSourceAddr(
    E_DRVPDMA_CHANNEL_INDEX eChannel
)
```

描述

这个函数可以用来取得指定 PDMA 通道的当前源地址.

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

当前源地址寄存器表示 PDMA 传输正在进行的源地址

示例

```
/*Get channel7 current source address.*/
uint32_t u32Channel7CurrentSourceAddress;
u32Channel7CurrentSourceAddress =
DrvPDMA_GetCurrentSourceAddr(eDRVPDMA_CHANNEL_7);
```

DrvPDMA_GetRemainTransferCount

原型

```
uint32_t
DrvPDMA_GetRemainTransferCount(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来取得指定 PDMA 通道的当前余下的字节数

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

当前余下的字节数

Note

如果用户设定传输字节数为 64，在数据传输开始时，当前余下的字节数是 64。在 PDMA 传输 4 字节数据到内存之后，用户可以调用这个 API 获取到当前余下的字节数是 60 自节。

示例

```
/* Get Channel0 Current remained byte count */
uint32_t u32CurrentRemainedByteCount;
u32CurrentRemainedByteCount =
DrvPDMA_GetRemainTransferCount(eDRVPDMA_CHANNEL_0);
```

DrvPDMA_GetInternalBufPointer

原型

```
uint32_t
DrvPDMA_GetInternalBufPointer(
    E_DRVPDMA_CHANNEL_INDEX eChannel
);
```

描述

这个函数可以用来取得指定通道的内部缓存指针。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

E_DRVPDMA_ERR_PORT_INVALID：无效端口

0x1：内部指针指向 byte1(PDMA 缓存中剩余一个字节)

0x3：内部指针指向 byte2(PDMA 缓存中剩余两个个字节)

0x7：内部指针指向 byte3(PDMA 缓存中剩余三个字节)

0xF：内部指针指向 byte4(PDMA 缓存中没有剩余数据)

示例

```
/*Get channel0 internal buffer data point to know how many bytes remained in PDMA shared
buffer and print the internal buffer values.*/
uint32_t u32PdmaInternalBufferPoint ;
uint32_t u32PdmaSharedBufferData;
uint8_t au8Effect iveSharedBufferData[4] ;
u32PdmaInternalBufferPoint =
DrvPDMA_GetInternalBufPointer(eDRVPDMA_CHANNEL_0)
if(u32PdmaInternalBufferPoint==0x01)
{
   printf("Because the Pdma Internal bufer point is 0x01 which indicates that there is only
       one byte data remained in PDMA buffer!"""")
   u32PdmaSharedBufferData =
   DrvPDMA_GetSharedBufData(eDRVPDMA_CHANNEL_0) ;
   au8Effect iveSharedBufferData [0] =                    (uint8_t)
(u32PdmaSharedBufferData&0x000000FF) ;
   printf("PDMA Shared buffer data is %x\n""", au8EffectiveSharedBufferData [0]) ;
}
else if(u32PdmaInternalBufferPoint==0x03)
{
   printf(""Because the Pdma Internal bufer point is 0x03 which indicates that there is two
       bytes data remained in PDMA buffer!"""")
   u32PdmaSharedBufferData =
   DrvPDMA_GetSharedBufData(eDRVPDMA_CHANNEL_0) ;
   au8Effect iveSharedBufferData [0] =       (uint8_t)
(u32PdmaSharedBufferData&0x000000FF) ;
   au8Effect iveSharedBufferData [1] =       (uint8_t)
(u32PdmaSharedBufferData&0x0000FF00) ;
   printf(""PDMA Shared buffer data are %x and %x\n""", au8EffectiveSharedBufferData
[0],
            au8Effect iveSharedBufferData [1]) ;
}
else if(u32PdmaInternalBufferPoint==0x07)
{
```

```
    printf("Because the Pdma Internal bufer point is 0x07 which indicates that there is three
        bytes data remained in PDMA buffer!"""")

    u32PdmaSharedBufferData =
    DrvPDMA_GetSharedBufData(eDRVPDMA_CHANNEL_0) ;

    au8Effect iveSharedBufferData [0] =       (uint8_t)
(u32PdmaSharedBufferData&0x000000FF) ;

    au8Effect iveSharedBufferData [1] =       (uint8_t)
(u32PdmaSharedBufferData&0x0000FF00) ;

    au8Effect iveSharedBufferData [2] =       (uint8_t)
(u32PdmaSharedBufferData&0x00FF0000) ;

    printf("PDMA Shared buffer data are %x,%x and
%x\n"""",au8EffectiveSharedBufferData[0],

            au8Effect iveSharedBufferData [1] , au8Effect iveSharedBufferData [2]) ;

}

else if(u32PdmaInternalBufferPoint==0x0F)

{

    printf(""Because the Pdma Internal bufer point is 0x0F which indicates that there is no data
        in PDMA buffer!"""")

}
```

DrvPDMA_GetSharedBufData

原型

```
uint32_t

DrvPDMA_GetSharedBufData(

    E_DRVPDMA_CHANNEL_INDEX eChannel,

)
```

描述

这个函数可以用来取得指定 PDMA 通道的共享缓存的内容

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

共享缓存的内容

示例

请参考 DrvPDMA_GetInternalBufPointer()的示例

DrvPDMA_GetTransferLength

原型

```
int32_t
DrvPDMA_GetTransferLength(
    E_DRVPDMA_CHANNEL_INDEX eChannel,
    uint32_t* pu32TransferLength
);
```

描述

这个函数可以用来取得 PDMA 通道的传输长度设定值。* pu32TransferLength 的单位是字节。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

pu32TransferLength [in]

指向存放传输长度的缓存指针

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

示例

```
/* Get the transfer byte count setting of channel0.*/
uint32_t u32GetTransferByteCountSetting;
DrvPDMA_GetTransferLength(eDRVPDMA_CHANNEL_0,
&u32GetTransferByteCountSetting)
```

DrvPDMA_InstallCallBack

原型

```
int32_t
DrvPDMA_InstallCallBack(
    E_DRVPDMA_CHANNEL_INDEX eChannel,
    E_DRVPDMA_INT_ENABLE eIntSource,
```

PFN_DRVPDMA_CALLBACK pfncallback

);

描述

这个函数可以用来给指定 PDMA 通道和中断源安装回调函数。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntSource [in]

中断源：

eDRVPDMA_TABORT：读/写目标中止

eDRVPDMA_BLKD：块传输完成

pfncallback [in]

回调函数指针

头文件

Driver/DrvPDMA.h

返回值

E_SUCCESS：成功

示例

请参考 DrvPDMA_Open()示例代码

DrvPDMA_IsCHBusy

原型

int32_t

DrvPDMA_IsCHBusy(

E_DRVPDMA_CHANNEL_INDEX eChannel

);

描述

这个函数可以用来获取 PDMA 通道使能/禁止状态。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

头文件

Driver/DrvPDMA.h

返回值

TRUE：通道正忙

FALSE：通道没有使用

E_DRVPDMA_ERR_PORT_INVALID：无效端口号

示例

```
/* Get channel0 bus status.*/
int32_t i32Channel0BusStatus;
i32Channel0BusStatus = if(DrvPDMA_IsCHBusy(eDRVPDMA_CHANNEL_0);
if(i32Channel0BusStatus== TRUE)
   printf("Channel0 bus is busy!!\n");
else if(i32Channel0BusStatus== FALSE)
   printf("Channel0 bus is not busy!!\n");
else if(i32Channel0BusStatus== E_DRVPDMA_ERR_PORT_INVALID)
   printf(" invalid port!!\n");
```

DrvPDMA_IsIntEnabled

原型

```
int32_t
DrvPDMA_IsIntEnabled(
   E_DRVPDMA_CHANNEL_INDEX eChannel,
    E_DRVPDMA_INT_ENABLE eIntSource
);
```

描述

这个函数可以用来检查指定 PDMA 通道的指定的中断是否使能。

参数

eChannel [in]

指定 PDMA 通道 eDRVPDMA_CHANNEL_0~8

eIntSource [in]

中断源：eDRVPDMA_TABORT/eDRVPDMA_BLKD

头文件

Driver/DrvPDMA.h

返回值

TRUE：指定通道的指定中断源使能

FALSE：指定通道的指定中断源禁止

示例

```
int32_t i32IsIntEnable;

i32IsIntEnable=DrvPDMA_IsIntEnabled(eDRVPDMA_CHANNEL_0,
eDRVPDMA_BLKD)

if(i32IsIntEnable == TRUE )
   printf("Channel0 Block transfer Done interrupt is enable!\n");
else if(i32IsIntEnable == FALSE )
   printf("Channel0 Block transfer Done interrupt is disable!\n");
```

DrvPDMA_GetVersion

原型

int32_t

DrvPDMA_GetVersion (void);

描述

返回驱动当前版本号。

参数

无

头文件

Driver/DrvPDMA.h

返回值

PDMA 驱动当前版本号：

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

示例

```
/* Get PDMA driver current version number */

int32_t i32PDMAVersionNum;

i32PDMAVersionNum = DrvPDMA_GetVersion();
```

16. I2S 驱动

16.1. I2S 介绍

I2S 控制器包含 IIS 协议，用于连接外部音频 CODEC。两个 8 字的 FIFO 分别用于读与写通道，可以处理 8 比特，16 比特，24 比特和 32 比特数据。DMA 控制器处理数据在 FIFO 和内存之间传输。

16.2. I2S 特性

- 主机或从机模式
- 可以处理 8，16，24 和 32 比特数据
- 支持单声道和立体声音频数据
- 支持 I2S 和 MSB 校验数据格式
- 提供两个 8 字 FIFO 数据缓存。一个用于发送，一个用于接收
- 当 Tx/Rx FIFO 缓冲超过可编程边界时，产生中断请求
- 两个 DMA 请求。一个用于发送，一个用于接收

16.3. 常量定义

常量名	值	描述
DRVI2S_DATABIT_8	0x00	数据大小是 8 比特
DRVI2S_DATABIT_16	0x01	数据大小是 16 比特
DRVI2S_DATABIT_24	0x02	数据大小是 24 比特
DRVI2S_DATABIT_32	0x03	数据大小是 32 比特
DRVI2S_MONO	0x01	数据是单声道格式
DRVI2S_STEREO	0x00	数据是立体声格式
DRVI2S_FORMAT_MSB	0x01	MSB 校验数据格式
DRVI2S_FORMAT_I2S	0x00	I2S 数据格式
DRVI2S_MODE_SLAVE	0x01	I2S 工作在从机模式
DRVI2S_MODE_MASTER	0x00	I2S 工作在主机模式
DRVI2S_FIFO_LEVEL_WORD_0	0x00	FIFO 阈值水平是 0 个字
DRVI2S_FIFO_LEVEL_WORD_1	0x01	FIFO 阈值水平是 1 个字
DRVI2S_FIFO_LEVEL_WORD_2	0x02	FIFO 阈值水平是 2 个字
DRVI2S_FIFO_LEVEL_WORD_3	0x03	FIFO 阈值水平是 3 个字
DRVI2S_FIFO_LEVEL_WORD_4	0x04	FIFO 阈值水平是 4 个字
DRVI2S_FIFO_LEVEL_WORD_5	0x05	FIFO 阈值水平是 5 个字
DRVI2S_FIFO_LEVEL_WORD_6	0x06	FIFO 阈值水平是 6 个字
DRVI2S_FIFO_LEVEL_WORD_7	0x07	FIFO 阈值水平是 7 个字
DRVI2S_FIFO_LEVEL_WORD_8	0x08	FIFO 阈值水平是 8 个字
DRVI2S_EXT_12M	0	I2S 时钟源来自外部 12M Crystal 时钟
DRVI2S_PLL	1	I2S 时钟源来自 PLL 时钟
DRVI2S_HCLK	2	I2S 时钟源来自 HCLK
DRVI2S_INTERNAL_22M	3	I2S 时钟源来自内部 22M RC 时钟

16.4. 类型定义

E_I2S_CHANNEL

枚举标识符	值	描述
I2S_LEFT_CHANNEL	0	I2S 左通道
I2S_RIGHT_CHANNEL	1	I2S 右通道

E_I2S_CALLBACK_TYPE

枚举标识符	值	描述
I2S_RX_UNDERFLOW	0	RX FIFO 下溢出中断
I2S_RX_OVERFLOW	1	RX FIFO 溢出中断
I2S_RX_FIFO_THRESHOLD	2	RX FIFO 阈值水平中断

I2S_TX_UNDERFLOW	8	TX FIFO 下溢出中断
I2S_TX_OVERFLOW	9	TX FIFO 溢出中断
I2S_TX_FIFO_THRESHOLD	10	TX FIFO 阈值水平中断
I2S_TX_RIGHT_ZERO_CROSS	11	TX 右通道过零中断
I2S_TX_LEFT_ZERO_CROSS	12	TX 左通道过零中断

16.5. 宏

_DRVI2S_WRITE_TX_FIFO

原型

```
static __inline
void _DRVI2S_WRITE_TX_FIFO (
    uint32_t u32Data
);
```

描述

写字数据到 Tx FIFO。

参数

u32Data [in]

要写到 Tx FIFO 的字数据

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Write word data 0x12345678 into I2S Tx FIFO */
_DRVI2S_WRITE_TX_FIFO (0x12345678);
```

_DRVI2S_READ_RX_FIFO

原型

```
static __inline
uint32_t
_DRVI2S_READ_RX_FIFO (
    void
```

);

描述

从 Rx FIFO 读取字数据。

参数

无

头文件

Driver/DrvI2S.h

返回值

从 Rx FIFO 读取的字数据

示例

```
uint32_t u32data;
/* Read word data from I2S Rx FIFO */
u32data = _DRVI2S_READ_RX_FIFO ();
```

_DRVI2S_READ_TX_FIFO_LEVEL

原型

```
static __inline
uint32_t
_DRVI2S_READ_TX_FIFO_LEVEL (
   void
);
```

描述

获取 Tx FIFO 中字数据数量。

参数

无

头文件

Driver/DrvI2S.h

返回值

0~8：Tx FIFO 中字数据数量

示例

```
uint32_t u32len;
```

```
/* Get word data number in Tx FIFO */
u32len = _DRVI2S_READ_TX_FIFO_LEVEL ();
```

_DRVI2S_READ_RX_FIFO_LEVEL

原型

```
static __inline
uint32_t
_DRVI2S_READ_RX_FIFO_LEVEL (
   void
);
```

描述

获取 Rx FIFO 中字数据数量。

参数

无

头文件

Driver/DrvI2S.h

返回值

0~8：Rx FIFO 中字数据数量

示例

```
uint32_t u32len;
/* Get word data number in Rx FIFO */
u32len = _DRVI2S_READ_RX_FIFO_LEVEL ();
```

16.6. 函数

DrvI2S_Open

原型

int32_t DrvI2S_Open(S_DRVI2S_DATA_T *sParam);

描述

该函数用于使能 I2S 时钟和功能，并配置数据长度/数据格式/FIFO 阈值水平/BCLK(比特时钟)。数据和音频格式在 TRM 的 I2S 章节中 I2S 操作和 FIFO 操作部分给出。对于主机模式，*I2S_BCLK* 和 *I2S_LRCLK* 引脚工作在输出模式；对于从机模式，*I2S_BCLK* 和

I2S_LRCLK 引脚工作在输入模式。I2S 信号(*I2S_BCLK* 和 *I2S_LRCLK*)在 TRM 的 I2S 章节中 I2S 模块图部分给出。

参数

***sParam [in]**

包含如下参数：

u32SampleRate：采样率。当 I2S 工作在主机模式时设定值有效。

u8WordWidth：8，16，24，或者 32 比特数据大小 -DRVI2S_DATABIT_8/ DRVI2S_DATABIT_16/ DRVI2S_DATABIT_24/ DRVI2S_DATABIT_32

u8AudioFormat：支持单声道或立体声音频数据 -DRVI2S_MONO/ DRVI2S_STEREO

u8DataFormat：支持 I2S 和 MSB 校验数据格式 -DRVI2S_FORMAT_I2S/ DRVI2S_FORMAT_MSB

u8Mode：主机或从机操作模式 -DRVI2S_MODE_MASTER/ DRVI2S_MODE_SLAVE

u8TxFIFOThreshold：Tx FIFO 阈值水平 -

DRVI2S_FIFO_LEVEL_WORD_0/

DRVI2S_FIFO_LEVEL_WORD_1/

DRVI2S_FIFO_LEVEL_WORD_2/

DRVI2S_FIFO_LEVEL_WORD_3/

DRVI2S_FIFO_LEVEL_WORD_4/

DRVI2S_FIFO_LEVEL_WORD_5/

DRVI2S_FIFO_LEVEL_WORD_6/

DRVI2S_FIFO_LEVEL_WORD_7

u8RxFIFOThreshold：Rx FIFO 阈值水平 -

DRVI2S_FIFO_LEVEL_WORD_1/

DRVI2S_FIFO_LEVEL_WORD_2/

DRVI2S_FIFO_LEVEL_WORD_3/

DRVI2S_FIFO_LEVEL_WORD_4/

DRVI2S_FIFO_LEVEL_WORD_5/

DRVI2S_FIFO_LEVEL_WORD_6/

DRVI2S_FIFO_LEVEL_WORD_7/

DRVI2S_FIFO_LEVEL_WORD_8

头文件

Driver/DrvI2S.h

返回值

0：成功

示例

```
S_DRVI2S_DATA_T st;
st.u32SampleRate = 16000; /* Sampling rate is 16ksps */
st.u8WordWidth = DRVI2S_DATABIT_16; /* Data length is 16-bit */
st.u8AudioFormat = DRVI2S_STEREO; /* Stereo format */
st.u8DataFormat = DRVI2S_FORMAT_I2S; /* I2S data format */
st.u8Mode = DRVI2S_MODE_MASTER; /* Operate as master mode */
/* Tx FIFO threshold level is 0 word data */
st.u8TxFIFOThreshold = DRVI2S_FIFO_LEVEL_WORD_0;
/* Rx FIFO threshold level is 8 word data */
st.u8RxFIFOThreshold = DRVI2S_FIFO_LEVEL_WORD_8;
/* Enable I2S and configure its settings */
DrvI2S_Open (&st);
```

DrvI2S_Close

原型

void DrvI2S_Close (void);

描述

关闭 I2S 控制器并禁止 I2S 时钟。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
DrvI2S_Close ( ); /* Disable I2S */
```

DrvI2S_EnableInt

原型

int32_t DrvI2S_EnableInt (E_I2S_CALLBACK_TYPE Type, I2S_CALLBACK callbackfn);

描述

使能 I2S 中断功能并安装 I2S 中断处理函数中相关的回调函数。

参数

Type [in]

有八种回调函数类型

I2S_RX_UNDERFLOW：Rx FIFO 下溢出

I2S_RX_OVERFLOW：Rx FIFO 溢出

I2S_RX_FIFO_THRESHOLD：Rx FIFO 中字数据数高于 Rx 阈值水平

I2S_TX_UNDERFLOW：Tx FIFO 下溢出

I2S_TX_OVERFLOW：Tx FIFO 溢出

I2S_TX_FIFO_THRESHOLD：Tx FIFO 中字数据数低于 Tx 阈值水平

I2S_TX_RIGHT_ZERO_CROSS：Tx 右通道过零

I2S_TX_LEFT_ZERO_CROSS：Tx 左通道过零

callbackfn [in]

指定中断事件的回调函数名

头文件

Driver/DrvI2S.h

返回值

0：成功

<0：失败

示例

```
/* Enable Rx threshold level interrupt and install its callback function */
DrvI2S_EnableInt (I2S_RX_FIFO_THRESHOLD, Rx_thresholdCallbackfn);
/* Enable Tx threshold level interrupt and install its callback function */
DrvI2S_EnableInt (I2S_TX_FIFO_THRESHOLD, Tx_thresholdCallbackfn);
```

DrvI2S_DisableInt

原型

int32_t DrvI2S_DisableInt (E_I2S_CALLBACK_TYPE Type);

描述

禁止 I2S 中断功能并卸载 I2S 中断处理函数中相关的回调函数。

参数

Type [in]

有八种回调函数类型

I2S_RX_UNDERFLOW：Rx FIFO 下溢出

I2S_RX_OVERFLOW：Rx FIFO 溢出

I2S_RX_FIFO_THRESHOLD：Rx FIFO 中字数据数高于 Rx 阈值水平

I2S_TX_UNDERFLOW：Tx FIFO 下溢出

I2S_TX_OVERFLOW：Tx FIFO 溢出

I2S_TX_FIFO_THRESHOLD：Tx FIFO 中字数据数低于 Tx 阈值水平

I2S_TX_RIGHT_ZERO_CROSS：Tx 右通道过零

I2S_TX_LEFT_ZERO_CROSS：Tx 左通道过零

callbackfn [in]

指定中断事件的回调函数名

头文件

Driver/DrvI2S.h

返回值

0：成功

<0：失败

示例

```
/* Disable Rx threshold level interrupt and uninstall its callback function */
DrvI2S_DisableInt (I2S_RX_FIFO_THRESHOLD);
/* Disable Tx threshold level interrupt and uninstall its callback function */
DrvI2S_DisableInt (I2S_TX_FIFO_THRESHOLD);
```

DrvI2S_GetBCLKFreq

原型

uint32_t DrvI2S_GetBCLKFreq (void);

描述

获取 I2S BCLK(比特时钟)频率。

BCLK = I2S source clock / (2 x *(BCLK divider + 1))*

参数

无

头文件

Driver/DrvI2S.h

返回值

I2S BCLK 频率。单位是 Hz

示例

```
uint32_t u32clock;
u32clock = DrvI2S_GetBCLKFreq ( ); /* Get I2S BCLK clock frequency */
```

DrvI2S_SetBCLKFreq

原型

void DrvI2S_SetBCLKFreq (uint32_t u32Bclk);

描述

配置 I2S BCLK(比特时钟)频率。当 I2S 工作在主机模式时，BCLK 起作用。

BCLK = I2S source clock / (2 x *(BCLK divider + 1))*

参数

u32Bclk [in]

I2S BCLK 频率。单位是 Hz

头文件

Driver/DrvI2S.h

返回值

无

示例

```
DrvI2S_SetBCLKFreq (512000); /* Set I2S BCLK clock frequency 512 KHz */
```

DrvI2S_GetMCLKFreq

原型

uint32_t DrvI2S_GetMCLKFreq (void);

描述

获取 I2S MCLK(主时钟)频率。

MCLK = I2S source clock / (2 x *MCLK divider))*

参数

无

头文件

Driver/DrvI2S.h

返回值

I2S MCLK 频率。单位是 Hz

示例

```
uint32_t u32clock;
u32clock = DrvI2S_GetMCLKFreq ( ); /* Get I2S MCLK clock frequency */
```

DrvI2S_SetMCLKFreq

原型

void DrvI2S_SetMCLKFreq (uint32_t u32Mclk);

描述

配置 I2S MCLK(主时钟)。

MCLK = I2S source clock / (2 x *MCLK divider))*

参数

u32Mclk [in]

I2S MCLK 频率。单位是 Hz

头文件

Driver/DrvI2S.h

返回值

无

示例

```
DrvI2S_SetMCLKFreq (12000000); /* Set I2S MCLK clock frequency 12MHz */
```

DrvI2S_SetChannelZeroCrossDetect

原型

int32_t DrvI2S_SetChannelZeroCrossDetect (E_I2S_CHANNEL channel, int32_t i32flag);

描述

使能或禁止右/左通道过零检测功能。

参数

channel [in]

I2S_LEFT_CHANNEL / I2S_RIGHT_CHANNEL

i32flag [in]

使能或禁止过零检测功能。(1：使能 0：禁止)

头文件

Driver/DrvI2S.h

返回值

0：成功

<0：失败

示例

```
/* Enable left channel zero cross detect */
DrvI2S_SetChannelZeroCrossDetect (I2S_LEFT_CHANNEL, 1);
/* Disable right channel zero cross detect */
DrvI2S_SetChannelZeroCrossDetect (I2S_RIGHT_CHANNEL, 0);
```

DrvI2S_EnableTxDMA

原型

void DrvI2S_EnableTxDMA (void);

描述

使能 I2S Tx DMA 功能。I2S 请求 DMA 传送数据到 Tx FIFO。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable I2S Tx DMA function */
DrvI2S_EnableTxDMA ( );
```

DrvI2S_DisableTxDMA

原型

void DrvI2S_DisableTxDMA (void);

描述

禁止 I2S Tx DMA 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable I2S Tx DMA function */
DrvI2S_DisableTxDMA ( );
```

DrvI2S_EnableRxDMA

原型

void DrvI2S_EnableRxDMA (void);

描述

使能 I2S Rx DMA 功能。I2S 请求 DMA 从 Rx FIFO 传输数据。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable I2S Rx DMA function */
DrvI2S_EnableRxDMA ( );
```

DrvI2S_DisableRxDMA

原型

void DrvI2S_DisableRxDMA (void);

描述

禁止 I2S Rx DMA 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable I2S Rx DMA function */
DrvI2S_DisableRxDMA ( );
```

DrvI2S_EnableTx

原型

void DrvI2S_EnableTx (void);

描述

使能 I2S Tx 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable I2S Tx function */
DrvI2S_EnableTx ( );
```

DrvI2S_DisableTx

原型

void DrvI2S_DisableTx (void);

描述

禁止 I2S Tx 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable I2S Tx function */
DrvI2S_DisableTx ( );
```

DrvI2S_EnableRx

原型

void DrvI2S_EnableRx (void);

描述

使能 I2S Rx 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable I2S Rx function */
DrvI2S_EnableRx ( );
```

DrvI2S_DisableRx

原型

void DrvI2S_DisableRx (void);

描述

禁止 I2S Rx 功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable I2S Rx function */
DrvI2S_DisableRx ( );
```

DrvI2S_EnableTxMute

原型

void DrvI2S_EnableTxMute (void);

描述

使能 I2S Tx 静音功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable I2S Tx  Mute function */
DrvI2S_EnableTxMute ( );
```

DrvI2S_DisableTxMute

原型

void DrvI2S_DisableTxMute (void);

描述

禁止 I2S Tx 静音功能。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable I2S Tx Mute function */
DrvI2S_DisableTxMute ( );
```

DrvI2S_EnableMCLK

原型

void DrvI2S_EnableMCLK (void);

描述

使能 I2S MCLK 从 GPA 引脚 15 输出。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Enable MCLK output */
DrvI2S_EnableMCLK ( );
```

DrvI2S_DisableMCLK

原型

void DrvI2S_DisableMCLK (void);

描述

禁止 I2S MCLK 从 GPA 引脚 15 输出。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
/* Disable MCLK output */
DrvI2S_DisableMCLK ( );
```

DrvI2S_ClearTxFIFO

原型

void DrvI2S_ClearTxFIFO (void);

描述

清除 Tx FIFO。Tx FIFO 的内部指针复位到初始位置。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

```
DrvI2S_ClearTxFIFO ( ); /* Clear Tx FIFO */
```

DrvI2S_ClearRxFIFO

原型

void DrvI2S_ClearRxFIFO (void);

描述

清除 Rx FIFO。Rx FIFO 的内部指针复位到初始位置。

参数

无

头文件

Driver/DrvI2S.h

返回值

无

示例

DrvI2S_ClearRxFIFO ( ); /* Clear Rx FIFO */

DrvI2S_SelectClockSource

原型

void DrvI2S_SelectClockSource (uint8_t u8ClkSrcSel);

描述

选择 I2S 时钟源，包括外部 12M，PLL 时钟，HCLK 和内部 22M。

参数

`u8ClkSrcSel [in]`

选择 I2S 时钟源，有四组时钟源可供 I2S 选择：

DRVI2S_EXT_12M：外部 12MHz crystal 时钟

DRVI2S_PLL：PLL 时钟

DRVI2S_HCLK：HCLK

DRVI2S_INTERNAL_22M：内部 22MHz oscillator 时钟

头文件

Driver/DrvI2S.h

返回值

无

示例

DrvI2S_SelClockSource (DRVI2S_EXT_12M); /* I2S clock source from external 12M */

DrvI2S_SelClockSource (DRVI2S_PLL); /* I2S clock source from PLL clock */

DrvI2S_SelClockSource (DRVI2S_HCLK); /* I2S clock source from HCLK */

DrvI2S_GetSourceClockFreq

原型

uint32_t DrvI2S_GetSourceClockFreq (void);

描述

获取 I2S 时钟源频率。

参数

无

头文件

Driver/DrvI2S.h

返回值

获取 I2S 时钟源频率。单位是 Hz

示例

```
uint32_t u32clock;
u32clock = DrvI2S_GetSourceClock ( ); /* Get I2S source clock frequency */
```

DrvI2S_GetVersion

原型

uint32_t DrvI2S_GetVersion (void);

描述

返回该模块的版本号。

参数

无

头文件

Driver/ DrvI2S.h

返回值

版本号：

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

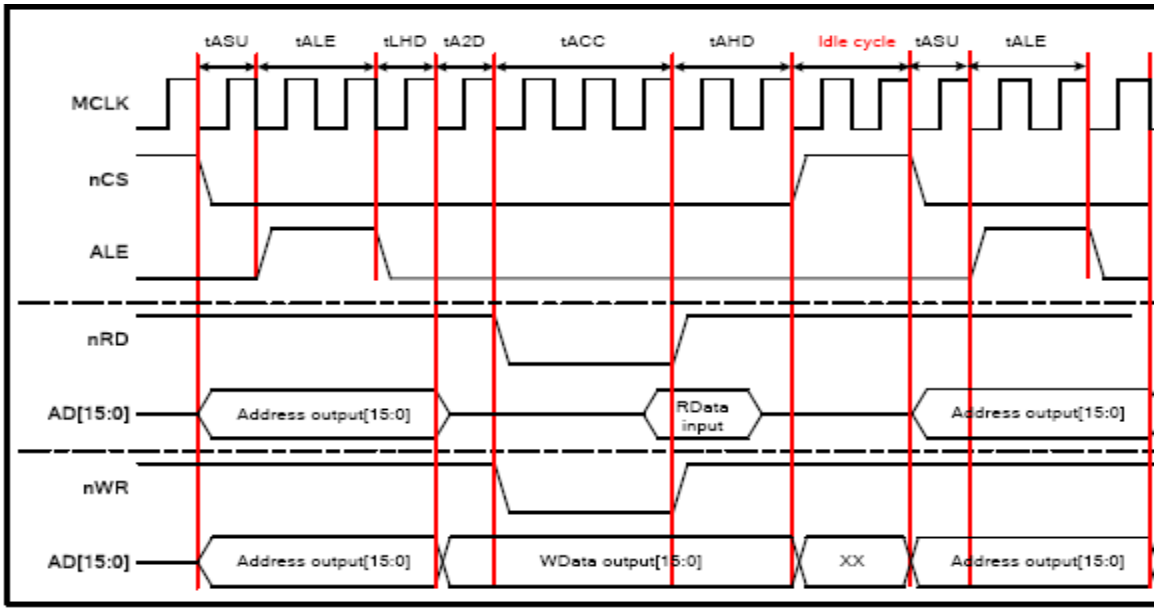
17. EBI 驱动

17.1. EBI 介绍

NUC100 低密度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。

17.2. 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，



17.3. 类型定义

E_DRVEBI_BUS_WIDTH

枚举标识符	值	描述
E_DEVEBI_BUS_8BIT	0x0	EBI 数据总线宽度是 8 比特
E_DEVEBI_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

17.4. 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

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

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 = u8Ext IW2X*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，详细信息请参考 DrvEBI_SetBusTiming

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 ();
```

18. 附录

18.1. NuMicro™ NUC100 系列产品选型指导

NUC100 Advance Line Selection Guide (medium density)

Part number	Flash (KB)	SRAM (KB)	Connectivity			I2S	PWM	Comp.	ADC	Timer	RTC	ISP ICP	I/O	Package
			UART	SPI	I2C									
NUC100LD3AN	64 KB	16 KB	2	1	2	1	6	1	8x12-bit	4x32-bit	v	v	up to 35	LQFP48
NUC100LE3AN	128 KB	16 KB	2	1	2	1	6	1	8x12-bit	4x32-bit	v	v	up to 35	LQFP48
NUC100RD3AN	64 KB	16 KB	2	2	2	1	6	2	8x12-bit	4x32-bit	v	v	up to 49	LQFP64
NUC100RE3AN	128 KB	16 KB	2	2	2	1	6	2	8x12-bit	4x32-bit	v	v	up to 49	LQFP64
NUC100VD2AN	64 KB	8 KB	3	4	2	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100
NUC100VD3AN	64 KB	16 KB	3	4	2	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100
NUC100VE3AN	128 KB	16 KB	3	4	2	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100

NUC100 Advance Line Selection Guide (low density)

Part number	Flash	SRAM	Connectivity			I2S	PWM	Comp.	ADC	Timer	RTC	EBI	ISP ICP	I/O	Package
			UART	SPI	I2C										
NUC100LC1BN	32 KB	4 KB	2	1	2	1	4	1	8x12-Bit	4x32-bit	v	-	v	up to 35	LQFP48
NUC100LD1BN	64 KB	4 KB	2	1	2	1	4	1	8x12-Bit	4x32-bit	v	-	v	up to 35	LQFP48
NUC100LD2BN	64 KB	8 KB	2	1	2	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 35	LQFP48
NUC100RC1BN	32 KB	4 KB	2	2	2	1	4	2	8x12-Bit	4x32-bit	v	v	v	up to 49	LQFP64
NUC100RD1BN	64 KB	4 KB	2	2	2	1	4	2	8x12-Bit	4x32-bit	v	v	v	up to 49	LQFP64
NUC100RD2BN	64 KB	8 KB	2	2	2	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 49	LQFP64

NUC120 USB Line Selection Guide (medium density)

Part number	Flash	SRAM	Connectivity				I2S	PWM	Comp.	ADC	Timer	RTC	ISP ICP	I/O	Package
			UART	SPI	I2C	USB									
NUC120LD3AN	64 KB	16 KB	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 31	LQFP48
NUC120LE3AN	128 KB	16 KB	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 31	LQFP48
NUC120RD3AN	64 KB	16 KB	2	2	2	1	1	6	2	8x12-bit	4x32-bit	v	v	up to 45	LQFP64

NUC120RE3AN	128 KB	16 KB	2	2	2	1	1	6	2	8x12-bit	4x32-bit	v	v	up to 45	LQFP64
NUC120VD2AN	64 KB	8 KB	3	4	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100
NUC120VD3AN	64 KB	16 KB	3	4	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100
NUC120VE3AN	128 KB	16 KB	3	4	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100

NUC120 USB Line Selection Guide (low density)

Part number	Flash	SRAM	Connectivity				I2S	PWM	Comp.	ADC	Timer	RTC	EBI	ISP ICP	I/O	Package
			UART	SPI	I2C	USB										
NUC120LC1BN	32 KB	4 KB	2	1	2	1	1	4	1	8x12-Bit	4x32-bit	v	-	v	up to 31	LQFP48
NUC120LD1BN	64 KB	4 KB	2	1	2	1	1	4	1	8x12-Bit	4x32-bit	v	-	v	up to 31	LQFP48
NUC120LD2BN	64 KB	8 KB	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 31	LQFP48
NUC120RC1BN	32 KB	4 KB	2	2	2	1	1	4	2	8x12-Bit	4x32-bit	v	v	v	up to 45	LQFP64
NUC120RD1BN	64 KB	4 KB	2	2	2	1	1	4	2	8x12-Bit	4x32-bit	v	v	v	up to 45	LQFP64
NUC120RD2BN	64 KB	8 KB	2	2	2	1	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 45	LQFP64

NUC130 Automotive Line Selection Guide (medium density)

Part number	Flash	SRAM	Connectivity					I2S	PWM	Comp.	ADC	Timer	RTC	ISP ICP	I/O	Package
			UART	SPI	I2C	LIN	CAN									
NUC130LD3AN	64 KB	16 KB	3	1	2	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 35	LQFP48
NUC130LE3AN	128 KB	16 KB	3	1	2	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 35	LQFP48
NUC130RD3AN	64 KB	16 KB	3	2	2	2	1	1	6	2	8x12-bit	4x32-bit	v	v	up to 49	LQFP64
NUC130RE3AN	128 KB	16 KB	3	2	2	2	1	1	6	2	8x12-bit	4x32-bit	v	v	up to 49	LQFP64
NUC130VD2AN	64 KB	8 KB	3	4	2	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100
NUC130VD3AN	64 KB	16 KB	3	4	2	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100
NUC130VE3AN	128 KB	16 KB	3	4	2	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 80	LQFP100

NUC130 Automotive Line Selection Guide (low density)

Part number	Flash	SRAM	Connectivity					I2S	PWM	Comp.	ADC	Timer	RTC	EBI	ISP ICP	I/O	Package
			UART	SPI	I2C	LIN	CAN										
NUC130LC1BN	32 KB	4 KB	2	1	2	2	1	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 35	LQFP48
NUC130LD2BN	64 KB	8 KB	2	1	2	2	1	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 35	LQFP48
NUC130RC1BN	32 KB	4 KB	2	2	2	2	1	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 49	LQFP64
NUC130RD2BN	64 KB	8 KB	2	2	2	2	1	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 49	LQFP64

NUC140 Connectivity Line Selection Guide (medium density)

Part number	Flash	SRAM	Connectivity						I2S	PWM	Comp.	ADC	Timer	RTC	ISP ICP	I/O	Package
			UART	SPI	I2C	USB	LIN	CAN									
NUC140LD3AN	64 KB	16 KB	2	1	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 31	LQFP48
NUC140LE3AN	128 KB	16 KB	2	1	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	v	up to 31	LQFP48
NUC140RD3AN	64 KB	16 KB	3	2	2	1	2	1	1	4	2	8x12-bit	4x32-bit	v	v	up to 45	LQFP64
NUC140RE3AN	128 KB	16 KB	3	2	2	1	2	1	1	4	2	8x12-bit	4x32-bit	v	v	up to 45	LQFP64
NUC140VD2AN	64 KB	8 KB	3	4	2	1	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100
NUC140VD3AN	64 KB	16 KB	3	4	2	1	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100
NUC140VE3AN	128 KB	16 KB	3	4	2	1	2	1	1	8	2	8x12-bit	4x32-bit	v	v	up to 76	LQFP100

NUC140 Connectivity Line Selection Guide (low density)

Part number	Flash	SRAM	Connectivity						I2S	PWM	Comp.	ADC	Timer	RTC	EBI	ISP ICP	I/O	Package
			UART	SPI	I2C	USB	LIN	CAN										
NUC140LC1BN	32 KB	4 KB	2	1	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 31	LQFP48
NUC140LD2BN	64 KB	8 KB	2	1	2	1	2	1	1	4	1	8x12-bit	4x32-bit	v	-	v	up to 31	LQFP48
NUC140RC1BN	32 KB	4 KB	2	2	2	1	2	1	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 45	LQFP64
NUC140RD2BN	64 KB	8 KB	2	2	2	1	2	1	1	4	2	8x12-bit	4x32-bit	v	v	v	up to 45	LQFP64

NUC101 Selection Guide

Part number	Flash	SRAM	Connectivity						I2S	PWM	Comp.	ADC	Timer	RTC	ISP ICP	I/O	Package
			UART	SPI	I2C	USB	LIN	CAN									
NUC101LC1BN	32 KB	4 KB	1	3	1	1	-	-	1	4	1	-	4x32-bit	-	v	up to 31	LQFP48
NUC101LD2BN	64 KB	8 KB	1	3	1	1	-	-	1	4	1	-	4x32-bit	-	v	up to 31	LQFP48
NUC101YC1BN	32 KB	4 KB	1	2	1	1	-	-	1	1	1	-	4x32-bit	-	v	up to 31	QFP36
NUC101YD2BN	64 KB	8 KB	1	2	1	1	-	-	1	1	1	-	4x32-bit	-	v	up to 31	QFP36

18.2. PDID 表

NUC100 Advance Line PDID List (medium density)

Part number	PDID
NUC100LD3AN	0x00010003
NUC100LE3AN	0x00010000
NUC100RD3AN	0x00010012
NUC100RE3AN	0x00010009
NUC100VD2AN	0x00010022
NUC100VD3AN	0x00010021
NUC100VE3AN	0x00010018

NUC100 Advance Line PDID List (low density)

Part number	PDID
NUC100LC1BN	0x10010008
NUC100LD1BN	0x10010005
NUC100LD2BN	0x10010004
NUC100RC1BN	0x10010017
NUC100RD1BN	0x10010014
NUC100RD2BN	0x10010013

NUC120 USB Line PDID List (medium density)

Part number	PDID
NUC120LD3AN	0x00012003
NUC120LE3AN	0x00012000
NUC120RD3AN	0x00012012
NUC120RE3AN	0x00012009
NUC120VD2AN	0x00012022
NUC120VD3AN	0x00012021
NUC120VE3AN	0x00012018

NUC120 USB Line PDID List (low density)

Part number	PDID
NUC120LC1BN	0x10012008
NUC120LD1BN	0x10012005
NUC120LD2BN	0x10012004
NUC120RC1BN	0x10012017
NUC120RD1BN	0x10012014
NUC120RD2BN	0x10012013

NUC130 Automotive Line PDID List (medium density)

Part number	PDID
NUC130LD3AN	0x00013003
NUC130LE3AN	0x00013000
NUC130RD3AN	0x00013012
NUC130RE3AN	0x00013009
NUC130VD2AN	0x00013022
NUC130VD3AN	0x00013021
NUC130VE3AN	0x00013018

NUC130 Automotive Line PDID List (low density)

Part number	PDID
NUC130LC1BN	0x10013008
NUC130LD2BN	0x10013004
NUC130RC1BN	0x10013017
NUC130RD2BN	0x10013013

NUC140 Connectivity Line PDID List (medium density)

Part number	PDID
NUC140LD3AN	0x00014003
NUC140LE3AN	0x00014000
NUC140RD3AN	0x00014012
NUC140RE3AN	0x00014009
NUC140VD2AN	0x00014022

NUC140VD3AN	0x00014021
NUC140VE3AN	0x00014018

NUC140 Connectivity Line PDID List (low density)

Part number	PDID
NUC140LC1BN	0x10014008
NUC140LD2BN	0x10014004
NUC140RC1BN	0x10014017
NUC140RD2BN	0x10014013

NUC101 PDID List

Part number	PDID
NUC101LC1BN	0x10010108
NUC101LD2BN	0x10010104
NUC101YC1BN	0x10010147
NUC101YD2BN	0x10010143

19. Revision History

版本	日期	描述
V1.00.001	1. 8, 2009	• Created
V1.00.002	7. 30, 2010	• 修正错误 • 增加 API 示例

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.