# KinetisK 系列固件库户手册

# 32位基于ARM_CortexM4微控制器KinetisK系列

**固件库版本:V2.4**

## 介 绍

本手册介绍了32位基于ARM_CortexM4微控制器KinetisK系列的固件函数库。

该函数库是一个固件函数包，它由程序、数据结构和宏组成，包括了微控制器所有外设的性能特征。该函数库还包括每一个外设的驱动描述和应用实例。通过使用本固件函数库，无需深入掌握细节，用户也可以轻松应用每一个外设。因此，使用本固件函数库可以大大减少用户的程序编写时间，进而降低开发成本。

该固件库使用C语言编写，主要使用KEIL软件进行编写，具有通用性，可以兼容IAR、CW等软件，并且包括了所有外设的功能，所以应用程序代码的大小和执行速度可能不是最优的。对大多数应用程序来说，用户可以直接使用之，对于那些在代码大小和执行速度方面有严格要求的应用程序，该固件库驱动程序可以作为如何设置外设的一份参考资料，根据实际需求对其进行调整。

此份固件库用户手册的整体架构如下：

- 定义、文档约定和固件函数库规则。
- 固件函数库概述（包的内容，库的架构），库使用实例。
- 固件库具体描述：设置架构和每个外设的函数。

KinetisK系列微处理器在整个文档中被写作K60。

**注：**此版本为第一版库函数，由前期各模块函数整理而来，如遇本资料与代码实例冲突的情况，请按照实例代码操作，带来不便还请见谅，如有疑问和建议请加入 qq 群进行交流：138899875、236155715。通过本群也可获得最新的模块开发代码。

鉴于作者水平有限，资料中难免存在不足和错误之处，恳请读者提出宝贵意见和建议，以便我们及时改进。

中国石油大学（华东）
飞思卡尔 MCU/DSP 实验室
超核电子

**日 期：2013年8月**

http://upcmcu.taobao.com/

# (如有不详之处请参考源文件代码)

# 目　　录

# 1. 文档和库规范

本用户手册和固件函数库按照以下章节所描述的规范编写。

## 1.1 缩写

**Table 1. 本文档所有缩写定义**

| 缩写 | 外设/单元/作用 |
|---|---|
| **SYS** | 系统时钟 NVIC 中断函数声明 和其他构件需要的常用宏和函数定义 |
| **DELAY** | 使用 Systick 构成的精确延时模块 |
| **WDOG** | 看门狗模块 |
| **DMA** | 直接内存存取控制器 |
| **GPIO** | 通用输入输出口 包括外部引脚中断 |
| **FLASH** | 片内闪存存储器 |
| **UART** | 通用异步串行口(串口) |
| **PIT** | 周期性中断定时器 |
| **RTC** | 实时时钟 |
| **ADC** | 模数转换器 |
| **DAC** | 数模转换器 |
| **SPI** | 串行外设接口 |
| **I2C** | 内部集成电路总线 |
| **SD** | SDIO 总线控制模块 |
| **CAN** | CAN 总线控制模块 |
| **TSI** | 电容触摸控制模块 |
| **FTM** | 灵活定时器(PWM 波产生 定时中断 AB 相正交解码 等) |
| **PDB** | 可编程延时模块(产生多路定时中断 可触发 ADC DAC 等) |
| **LPTM** | 低功耗定时器（产生定时中断 单相脉冲计数等） |
| **ENET** | 以太网 MAC\PHY 层驱动 |

## 1.2 命名规则

固件函数库遵从以下命名规则：

PPP表示任一外设缩写，例如：ADC。
常量仅被应用于一个文件中，定义于该文件中；被应用于多个文件的，在对应头文件中定义。所有常量都由英文字母大写书写。

寄存器作为常量处理。他们的命名都由英文字母大写书写。在大多数情况下，他们采用与缩写规范与本用户手册一致。

外设函数的命名以该外设的缩写加下划线为开头。每个单词的第一个字母都由英文字母大写书写，例如：**UART_SendData**。在函数名中，只允许存在一个下划线，用以分隔外设缩写和函数名的其它部分。

名为**PPP_Init**的函数，其功能是根据**PPP_InitTypeDef**中指定的参数，初始化外设PPP，例如**GPIO_Init**.

名为**PPP_StructInit**的函数，其功能为通过设置**PPP_InitTypeDef** 结构中的各种参数来定义外设的功能，例如：**GPIO_StructInit.**

名为**PPP_ITConfig**的函数，其功能为使能或者失能来自外设PPP某中断源，例如：**UART_ITConfig**.

名为**PPP_GetITStatus**的函数，其功能为判断来自外设PPP的中断发生与否，例如：**I2C_GetITStatus**.

名为**PPP_ClearITPendingBit**的函数，其功能为清除外设PPP中断待处理标志位，例如：**I2C_ClearITPendingBit**.

## 1.3 编码规则

本章节描述了固件函书库的编码规则。

### 1.3.1 变量

固件函数库定义了多种变量类型，他们的类型和大小是固定的。在文件***stdint.h***中定义了这些变量：

```
typedef    signed              char   int8_t;
typedef    signed short        int    int16_t;
typedef    signed              int    int32_t;
typedef    signed              __int64   int64_t;
typedef unsigned               char   uint8_t;
typedef unsigned short         int    uint16_t;
typedef unsigned               int    uint32_t;
typedef unsigned               __int64   uint64_t;
```

### 1.3.2 布尔型

在文件**sys.h**中，布尔形变量被定义如下：

```
typedef enum {FALSE = 0, TRUE = !FALSE} ErrorState;
```

### 1.3.3 功能状态类型

在文件**sys.h**中，我们定义功能状态类型（**FunctionalState** type）的2个可能值为“使能”与“失能”（**ENABLE** or **DISABLE**）。

```
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
```

# 2. 固件函数库

## 2.1 压缩包描述

K60固件函数库被压缩在一个zip文件中。解压该文件会产生一个文件夹，此文件夹中包含3个文件夹分别为OBJ、PRJ、SRC，OBJ文件存放编译器产生的文件，PRJ是建立工程的文件，SRC是工程源文件，里面存放各个模块的源代码。

### 2.1.1 文件夹 FWLIB

文件夹Fwlib，对应每一个K60外设，都包含一个子文件夹。这些子文件夹包含了整套文件，组成固件函数库，每个模块中都包含来两个文件：

***xxx.h***：xxx模块的相关变量声明和宏定义。

***xxx.c***：xxx模块的底层驱动源码。

注：所有的例程的使用，都不受不同软件开发环境的影响。

### 2.1.2 文件夹 HARDWARE

文件夹HARDWARE包含一些与硬件相关的函数，子文件夹包含了程序源代码，用户可根据实际需要进行参考，

**xxx.h**：xxx功能函数的相关变量声明和宏定义。

**xxx.c**：xxx功能函数的底层驱动源码。

注：所有代码都按照Strict ANSI-C标准书写，都不受不同软件开发环境的影响。

### 2.1.3 文件夹 STARTUP

文件夹STARTUP包含了工程启动的引导文件和芯片工作频率等的设置。

**startup_MK60DZ10.s**：Kinetis系类芯片的启动引导文件，由KEIL软件自带。

**startup_MK60DZ10.c**：Kinetis系类芯片时钟配置源代码，由KEIL软件自带。

### 2.1.4 文件夹 USER

文件夹USER包含了四个文件。

**isr.c**：Kinetis系类芯片的启动引导文件，由KEIL软件自带。

**isr.h**：Kinetis系类芯片时钟配置源代码，由KEIL软件自带。

**main.c**：Kinetis系类芯片的启动引导文件，由KEIL软件自带。

RESET **Abstract.txt**：简单的说明文档。

## 2.2 外设的初始化和设置

本节按步骤描述了如何初始化和设置任意外设。这里PPP代表任意外设。

1.在主应用文件中，声明一个结构PPP_InitTypeDef，例如：

PPP_InitTypeDef　PPP_InitStructure;

这里PPP_InitStructure是一个位于内存中的工作变量，用来初始化一个或者多个外设PPP。

2.为变量PPP_InitStructure的各个结构成员填入允许的值。可以采用以下2种方式：

a）按照如下程序设置整个结构体

```
PPP_InitStructure.member1=val1;
PPP_InitStructure.member2 = val2;
PPP_InitStructure.memberN = valN;
/* N代表结构体的成员数量*/
```

以上步骤可以合并在同一行里，用以优化代码大小：

```
PPP_InitTypeDef PPP_InitStructure = { val1, val2,.., valN}
```

b）仅设置结构体中的部分成员：这种情况下，用户应当首先调用函数PPP_SturcInit(..)来初始化变量PPP_InitStructure，然后再修改其中需要修改的成员。这样可以保证其他成员的值（多为缺省值）被正确填入。

```
PPP_StructInit(&PPP_InitStructure);
PP_InitStructure.memberX = valX;
PPP_InitStructure.memberY = valY;
/*X、Y 代表您期望配置的参数*/
```

3. 调用函数PPP_Init(..)来初始化外设PPP。

4. 在这一步，外设PPP已被初始化。可以调用函数PPP_Cmd(..)来使能之。

```
PPP_Cmd(PPP, ENABLE);
```

可以通过调用一系列函数来使用外设。每个外设都拥有各自的功能函数。更多细节参阅外设固件概述。

注： 在外设设置完成以后，继续修改它的一些参数，可以参照如下步骤：

```
PPP_InitStucture.memberX = valX;
PPP_InitStructure.memberY = valY; /*仅仅修改X、Y
PPP_Init(PPP, &PPP_InitStructure); //配置模块工作
```

## 2.3 位段（Bit-Banding）

Cortex™-M4 存储器映像包括两个位段(bit-band)区。这两个位段区将别名存储器区中的每个字映射到位段存储器区的一个位，在别名存储区写入一个字具有对位段区的目标位执行读-改-写操作的相同效果。所有K60外设寄存器都被映射到一个位段(bit-band)区。这个特性在各个函数中对单个比特进行置1/置0操作时被大量使用，用以减小和优化代码尺寸。2.3.1和2.3.2给出了外设固件函数库中如何实现位段访问的描述。

### 2.3.1 映射公式

映射公式给出了别名区中的每个字是如何对应位带区的相应位的，公式如下：

bit_word_offset = (byte_offset x 32) + (bit_number × 4)

bit_word_addr = bit_band_base + bit_word_offset

其中：

bit_word_offset是目标位在存取器位段区中的位置。

bit_word_addr 是别名存储器区中字的地址，它映射到某个目标位。

bit_band_base 是别名区的起始地址。

byte_offset 是包含目标位的字节在位段里的序号。

bit_number 是目标位所在位置（0-31）。

### 2.3.2 应用实例

下例展现了如何将GPIO的各个引脚映射到别名区，实现类似C51单片机一样的操作：

```
//IO口操作宏定义
#define BITBAND(addr,bitnum)       ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)
                                    +(bitnum<<2))
#define MEM_ADDR(addr)             *((volatile unsigned long    *)(addr))
#define BIT_ADDR(addr, bitnum)     MEM_ADDR(BITBAND(addr, bitnum))
//IO口地址映射
#define GPIOA_ODR_Addr      (PTA_BASE+0) //0x4001080C
#define GPIOB_ODR_Addr      (PTB_BASE+0) //0x40010C0C
#define GPIOC_ODR_Addr      (PTC_BASE+0) //0x4001100C
#define GPIOD_ODR_Addr      (PTD_BASE+0) //0x4001140C
#define GPIOE_ODR_Addr      (PTE_BASE+0) //0x4001180C
#define GPIOF_ODR_Addr      (PTF_BASE+0) //0x40011A0C
#define GPIOG_ODR_Addr      (PTG_BASE+0) //0x40011E0C
#define GPIOA_IDR_Addr      (PTA_BASE+0x10) //0x40010808
#define GPIOB_IDR_Addr      (PTB_BASE+0x10) //0x40010C08
#define GPIOC_IDR_Addr      (PTC_BASE+0x10) //0x40011008
#define GPIOD_IDR_Addr      (PTD_BASE+0x10) //0x40011408
#define GPIOE_IDR_Addr      (PTE_BASE+0x10) //0x40011808
#define GPIOF_IDR_Addr      (PTF_BASE+0x10) //0x40011A08
```

```
#define GPIOG_IDR_Addr       (PTG_BASE+0x10) //0x40011E08
 //IO口操作，只对单一的IO口
#define PAout(n)     BIT_ADDR(GPIOA_ODR_Addr,n)   //输出
#define PAin(n)      BIT_ADDR(GPIOA_IDR_Addr,n)   //输入
#define PBout(n)     BIT_ADDR(GPIOB_ODR_Addr,n)   //输出
#define PBin(n)      BIT_ADDR(GPIOB_IDR_Addr,n)   //输入
#define PCout(n)     BIT_ADDR(GPIOC_ODR_Addr,n)   //输出
#define PCin(n)      BIT_ADDR(GPIOC_IDR_Addr,n)   //输入
#define PDout(n)     BIT_ADDR(GPIOD_ODR_Addr,n)   //输出
#define PDin(n)      BIT_ADDR(GPIOD_IDR_Addr,n)   //输入
#define PEout(n)     BIT_ADDR(GPIOE_ODR_Addr,n)   //输出
#define PEin(n)      BIT_ADDR(GPIOE_IDR_Addr,n)   //输入
#define PFout(n)     BIT_ADDR(GPIOF_ODR_Addr,n)   //输出
#define PFin(n)      BIT_ADDR(GPIOF_IDR_Addr,n)   //输入
#define PGout(n)     BIT_ADDR(GPIOG_ODR_Addr,n) //输出
#define PGin(n)      BIT_ADDR(GPIOG_IDR_Addr,n)   //输入
```

## 2.4 运行时间检测

固件函数库通过检查库函书的输入来实现运行时间错误侦测。通过使用宏**assert_param**来实现运行时间检测。所有要求输入参数的函数都使用这个宏。它可以检查输入参数是否在允许的范围之内。

如果传给宏**assert_param**的参数为**false**，则调用函数**assert_failed**并返回被错误调用的函数所在的文件名和行数。如果传给宏**assert_param**的参数为**true**，则无返回值。

宏assert_param编写于文件sys.h中：

```
#ifdef   USE_FULL_ASSERT
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__, __LINE__))
  void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif /* USE_FULL_ASSERT */
```

函数**assert_failed**编写于文件**main.c**或者其他用户C文件：

```
#ifdef   USE_FULL_ASSERT
  void assert_failed(uint8_t* file, uint32_t line)
   {
          期望发生错误后所做的处理
       printf("assert_failed:line:%d %s\r\n",line,file);//打印错误的地方
       while(1);
   }
  #endif
```

**注：**运行时间检查，即宏**assert_param**应当只在库在Debug模式下编译时使用。建议在用户应用代码的开发和调试阶段使用运行时间检查，在最终的代码中去掉它们以改进代码尺寸和速度。

如果用户仍然希望在最终的代码中保留这项功能，可以在调用库函数前，重新使用宏**assert_param**来测试输入参数。

# 3. 外设固件概述

本节系统描述了每一个外设固件函数库。完整地描述所有相关函数并提供如何使用他们的例子，详见每个模块的章节。 函数的描述按如下格式进行：

**表 3. 函数描述格式**

| **函数名** | **外设函数的名称** |
|---|---|
| **函数原形** | 原形声明 |
| **功能描述** | 简要解释函数是如何执行的 |
| **输入参数{x}** | 输入参数描述 |
| **输出参数{x}** | 输出参数描述 |
| **返回值** | 函数的返回值 |
| **先决条件** | 调用函数前应满足的要求 |
| **被调用函数** | 其他被该函数调用的库函数 |

# 4. 通用输入输出模块(GPIO)

GPIO模块作为一般的通用输入输出接口，可以用作多个用途，Kinetis芯片的每个IO口最大输出电流为9ma，最大忍受电压为5V，支持5种输入输出状态，每个引脚支持独立的中断信号，支持多种中断源，并支持引脚滤波功能。

## 4.1 GPIO 模块主要寄存器结构

**GPIO**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| PDOR | 端口数据输出配置寄存器 |
| PSOR | 端口配置高寄存器 |
| PCOR | 端口配置低寄存器 |
| PTOR | 端口输出数据翻转寄存器 |
| PDIR | 端口输入读取寄存器 |
| PDDR | 数据输入输出方向寄存器 |

## 4.2 GPIO 库函数

GPIO 库函数

| 函数名 | 描述 |
|---|---|
| **GPIO_Init** | 填入 GPIO 初始化结构 初始化一个 GPIO 端口 |
| **GPIO_WritetBit** | 当 GPIO 设置为输出时 设置一个端口位的输出电平 |
| **GPIO_SetBits** | 当 GPIO 设置为输出时 设置一个端口位为高电平 |
| **GPIO_ResetBits** | 当 GPIO 设置为输出时 设置一个端口位为低电平 |
| **GPIO_ToggleBit** | 当 GPIO 设置为输出时 翻转一个端口位的电平 |
| **GPIO_Write** | 当 GPIO 设置为输出时 设置一个 GPIO 端口的电平 |
| **GPIO_ReadOutputDataBit** | 当 GPIO 设置为输出时 读取一个 GPIO 端口位的电平 |
| **GPIO_ReadOutputData** | 当 GPIO 设置为输出时 读取一个 GPIO 端口的电平 |
| **GPIO_ReadInputDataBit** | 当 GPIO 设置为输入时 读取一个 GPIO 端口位的电平 |
| **GPIO_ReadInputData** | 当 GPIO 设置为输入时 读取一个 GPIO 端口的电平 |
| **GPIO_GetITStates** | 获得外部中断标志位状态 |
| **GPIO_ClearITPendingBit** | 清除外部中断的中断待处理位 |

### 4.2.1 GPIO_Init

| 函数名 | **GPIO_Init** |
|---|---|
| **函数原形** | void GPIO_Init(GPIO_InitTypeDef* GPIO_InitStruct); |
| **功能描述** | 根据 GPIO_InitStruct 的参数初始化 GPIO |
| **输入参数** | GPIO_InitStruct：GPIO 初始化结构 |

| | |
|---|---|
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**GPIO_InitTypeDef** 位于**gpio.h**中，用于设置GPIO模块中指定引脚的工作状态，具体参数如下。

**GPIO_Pin**

该参数选择待设置的GPIO管脚，如下表

| GPIO_Pin | 描述 |
|---|---|
| **GPIO_Pin_0** | 选中管脚 0 |
| **GPIO_Pin_1** | 选中管脚 1 |
| **GPIO_Pin_2** | 选中管脚 2 |
| **GPIO_Pin_n** | 选中管脚 n |
| **GPIO_Pin_31** | 选中管脚 31 |

**GPIO_InitState**

该参数用来设置当GPIO为输出时，初始化后的电平状态，设置为输入时无效

| GPIO_InitState | 描述 |
|---|---|
| **Bit_SET** | 输出高电平 |
| **Bit_RESET** | 输出低电平 |

**GPIO_Mode**

该参数用来设置GPIO的工作模式

| GPIO_Mode | 描述 |
|---|---|
| **GPIO_Mode_IN_FLOATING** | 模拟输入 |
| **GPIO_Mode_IPD** | 下拉输入 |
| **GPIO_Mode_IPU** | 上拉输入 |
| **GPIO_Mode_OOD** | 开漏输出 |
| **GPIO_Mode_OPP** | 推挽输出 |

**GPIO_IRQMode**

该参数用来设置GPIO的中断源

| GPIO_IRQMode | 描述 |
|---|---|
| **GPIO_IT_DISABLE** | 禁止产生中断 |
| **GPIO_IT_DMA_FALLING** | 下降沿时产生 DMA 请求 |
| **GPIO_IT_DMA_RASING_FALLING** | 上升和下降沿时都产生 DMA 请求 |
| **GPIO_IT_LOW** | 低电平触发中断 |
| **GPIO_IT_RISING** | 上升沿触发中断 |
| **GPIO_IT_FALLING** | 下降沿触发中断 |
| **GPIO_IT_RISING_FALLING** | 上升和下降沿都触发中断 |
| **GPIO_IT_HIGH** | 高电平触发中断 |

**GPIOx**

该参数用来选择待设置的GPIO端口号

| GPIOx | 描述 |
|---|---|
| **PTA** | 选择 A 端口 |
| **PTB** | 选择 B 端口 |
| **PTC** | 选择 C 端口 |
| **PTD** | 选择 D 端口 |
| **PTE** | 选择 E 端口 |

**例：**配置PORTD端口的1引脚为输出，输出的初始值为高电平，使用推挽方式输出，不使用中断功能，具体使用如下：

```
GPIO_InitTypeDef GPIO_InitStruct1;//申请结构变量
GPIO_InitStruct1.GPIO_Pin = GPIO_Pin_1;//使用1引脚
GPIO_InitStruct1.GPIO_InitState = Bit_RESET;//输出为高电平
GPIO_InitStruct1.GPIO_IRQMode = GPIO_IT_DISABLE;//关闭中断
GPIO_InitStruct1.GPIO_Mode = GPIO_Mode_OPP;//使用推免方式输出
GPIO_InitStruct1.GPIOx = PTD;//使用PORTD端口
GPIO_Init(&GPIO_InitStruct1);//带入初始化函数
```

## 4.2.2 GPIO_WriteBit

| 函数名 | **GPIO_WriteBit** |
|---|---|
| **函数原形** | void GPIO_WriteBit(GPIO_Type *GPIOx,uint16_t GPIO_Pin, BitAction BitVal) |
| **功能描述** | 设置一个端口的指定引脚为高电平或者低电平 |
| **输入参数 1** | GPIOx:端口号 可以是 PTA PTB PTC PTD PTE |
| **输入参数 2** | GPIO_Pin:端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| **输入参数 3** | 端口位值: Bit_RESET:低电平 Bit_SET:高电平 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | GPIO 已经被设置为输出模式 |
| **被调用函数** | 无 |

**例：**设置PORTD端口的1引脚为高电平，具体使用如下：

```
GPIO_WriteBit(PTD, GPIO_Pin_1, Bit_SET);
```

## 4.2.3 GPIO_SetBit

| 函数名 | **GPIO_SetBit** |
|---|---|
| **函数原形** | void GPIO_SetBits(GPIO_Type* GPIOx, uint16_t GPIO_Pin) |

| | |
|---|---|
| **功能描述** | 设置一个端口引脚的状态为高电平 |
| **输入参数 1** | GPIOx:端口号 可以是 PTA PTB PTC PTD PTE |
| **输入参数 2** | GPIO_Pin:端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | GPIO 已经被设置为输出模式 |
| **被调用函数** | 无 |

**例：**设置PORTD端口的1引脚为高电平，具体使用如下：

```
GPIO_SetBit(PTD, GPIO_Pin_1);
```

## 4.2.4 GPIO_ResetBits

| **函数名** | **GPIO_ResetBit** |
|---|---|
| **函数原形** | void GPIO_ResetBits(GPIO_Type* GPIOx, uint16_t GPIO_Pin) |
| **功能描述** | 设置一个端口的指定引脚输出电平状态为低电平 |
| **输入参数 1** | GPIOx:端口号 可以是 PTA PTB PTC PTD PTE |
| **输入参数 2** | GPIO_Pin:端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | GPIO 已经被设置为输出模式 |
| **被调用函数** | 无 |

**例：**设置PORTD端口的1引脚为低电平，具体使用如下：

```
GPIO_ResetBit(PTD, GPIO_Pin_1);
```

## 4.2.5 GPIO_ToggleBit

| **函数名** | **GPIO_ToggleBit** |
|---|---|
| **函数原形** | GPIO_ToggleBit(GPIO_Type *GPIOx,uint16_t GPIO_Pin) |
| **功能描述** | 翻转一个端口指定引脚的电平状态 |
| **输入参数 1** | GPIOx:端口号 PTA、PTB、PTC、PTD、PTE |
| **输入参数 2** | GPIO_Pin:端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | GPIO 已经被设置为输出模式 |
| **被调用函数** | 无 |

**例：**改变PORTD端口的1引脚电平状态，如果引脚为高电平会变成低电平，如果为低电平则转换为高电平，具体使用如下：

```
GPIO_ToggleBit(PTD, GPIO_Pin_1);
```

## 4.2.6 GPIO_Write

| 函数名 | **GPIO_Write** |
|---|---|
| 函数原形 | GPIO_Write(GPIO_Type *GPIOx,uint32_t PortVal) |
| 功能描述 | 设置一个端口引脚的输出状态 |
| 输入参数 1 | GPIOx :端口号 PTA、PTB、PTC、PTD、PTE |
| 输入参数 2 | PortVal:设置端口的输出值，32 位 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | GPIO 已经被设置为输出模式 |
| 被调用函数 | 无 |

**例：**设置PORTD端口的数据输出为高电平，具体使用如下：

```
GPIO_Write(PTD,0xFFFFFFFF);
```

## 4.2.7 GPIO_ReadOutputDataBit

| 函数名 | **GPIO_ReadOutputDataBit** |
|---|---|
| 函数原形 | GPIO_ReadOutputDataBit(GPIO_Type* GPIOx, uint16_t GPIO_Pin) |
| 功能描述 | 获取指定端口引脚的输出电平状态 |
| 输入参数 1 | GPIOx: 端口号 PTA、PTB、PTC、PTD、PTE |
| 输入参数 2 | GPIO_Pin: 端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| 输出参数 | 无 |
| 返回值 | 有 |
| 先决条件 | GPIO 已经被设置为输出模式 |
| 被调用函数 | 无 |

**例：**获取PORTD端口的1引脚的输出电平状态，将状态存放在status中，具体使用如下：

```
Status = GPIO_ReadOutputDataBit(PTD, GPIO_Pin_1);
```

## 4.2.8 GPIO_ReadOutputData

| 函数名 | **GPIO_ReadOutputData** |
|---|---|
| 函数原形 | GPIO_ReadOutputData(GPIO_Type* GPIOx) |
| 功能描述 | 获取指定端口所有引脚的输出电平状态 |
| 输入参数 | GPIOx: 端口号 PTA、PTB、PTC、PTD、PTE |
| 输出参数 | 无 |
| 返回值 | 有 |
| 先决条件 | GPIO 已经被设置为输出模式 |
| 被调用函数 | 无 |

**例：**获取PORTD端口的32个引脚的输出电平状态，将状态存放在status中，具体使用如下：

Status = GPIO_ReadOutputDataBit(PTD);

## 4.2.9 GPIO_ReadInputDataBit

| 函数名 | **GPIO_ReadInputDataBit** |
|---|---|
| 函数原形 | GPIO_ReadInputDataBit(GPIO_Type* GPIOx, uint16_t GPIO_Pin) |
| 功能描述 | 获取指定端口引脚的输入电平状态 |
| 输入参数 1 | GPIOx：端口号 PTA、PTB、PTC、PTD、PTE |
| 输入参数 2 | GPIO_Pin：端口位号 GPIO_Pin_0～GPIO_Pin_31 |
| 输出参数 | 无 |
| 返回值 | 有 |
| 先决条件 | GPIO 已经被设置为输入模式 |
| 被调用函数 | 无 |

例：获取指定端口PORTD的1引脚的输入电平状态，将状态存放在status中，具体使用如下：

Status = GPIO_ ReadInputDataBit (PTD, GPIO_Pin_1);

## 4.2.10 GPIO_ReadInputData

| 函数名 | **GPIO_ReadInputData** |
|---|---|
| 函数原形 | GPIO_ReadInputData(GPIO_Type *GPIOx) |
| 功能描述 | 获取指定端口的输入电平状态 |
| 输入参数 | GPIOx：端口号 PTA、PTB、PTC、PTD、PTE |
| 输出参数 | 无 |
| 返回值 | 有 |
| 先决条件 | GPIO 已经被设置为输入模式 |
| 被调用函数 | 无 |

例：获取指定端口PORTD的32个引脚的输入电平状态，将状态存放在status中，具体使用如下：

Status = GPIO_ ReadInputData (PTD);

## 4.2.11 GPIO_GetITStates

| 函数名 | **GPIO_GetITStates** |
|---|---|
| 函数原形 | GPIO_GetITStates(GPIO_Type *GPIOx,uint16_t GPIO_Pin) |
| 功能描述 | 获取指定端口引脚的中断标志状态 |
| 输入参数 1 | GPIOx：端口号 PTA、PTB、PTC、PTD、PTE |
| 输入参数 2 | GPIO_Pin：端口位号 GPIO_Pin_0～GPIO_Pin_31 |
| 输出参数 | 无 |

| | |
|---|---|
| **返回值** | 有 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**获取指定PORTD端口1引脚的中断标志状态，将状态存放在status中，具体使用如下：

```
Status = GPIO_ GetITStates (PTD, GPIO_Pin_1);
```

### 4.2.12 GPIO_ClearITPendingBit

| **函数名** | **GPIO_ClearITPendingBit** |
|---|---|
| **函数原形** | GPIO_ClearITPendingBit(GPIO_Type *GPIOx,uint16_t GPIO_Pin) |
| **功能描述** | 清除指定端口引脚的中断标志状态 |
| **输入参数 1** | GPIOx: 端口号 PTA、PTB、PTC、PTD、PTE |
| **输入参数 2** | GPIO_Pin: 端口位号 GPIO_Pin_0 ~ GPIO_Pin_31 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**清除指定PORTD端口1引脚的中断标志状态，具体使用如下：

```
GPIO_ ClearITPendingBit (PTD, GPIO_Pin_1);
```

# 5. 通用异步收发器 (UART)

通用异步收发器(UART)是一种通用串行数据总线，用于异步通信，为双向通信。K60 的通用异步收发器支持全双工的数据传输，K60 包括 6 个相同独立的 UART 模块，每个模块都含有相对独立的发送器和接收器。UART 发送器的硬件可产生并发送奇偶校验位，而接收器奇偶校验硬件能确保接收数据的完整性。此外，UART 具有接收器帧错误检测功能，带有 DMA 接口。

## 5.1 UART 模块主要寄存器结构

**UART**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| BDH | 串口通信波特率设置寄存器(高位) |
| BDL | 串口通信波特率设置寄存器(低位) |
| C1 | 串口控制寄存器 1 |
| PTOR | 端口输出数据翻转寄存器 |
| PDIR | 端口输入读取寄存器 |
| PDDR | 数据输入输出方向寄存器 |

## 5.2 UART 库函数

UART 库函数

| 函数名 | 描述 |
| --- | --- |
| **UART_Init** | 初始化 UART 模块 |
| **UART_SendData** | 串口发送一个字节 |
| **UART_ReceiveData** | 使用串口接收一个字节 |
| **UART_SendDataInt** | 使用中断方式发送数据 |
| **DisplayCPUInfo** | 使用串口打印芯片信息 |
| **UART_SendDataIntProcess** | 使用中断方式发送数据处理函数（存放在中断函数中） |
| **UART_DMACmd** | 启用串口 DMA 功能 |
| **UART_DebugPortInit** | 串口调试端口初始化配置 |
| **UART_ITConfig** | 串口中断配置 |
| **UART_GetITStatus** | 获得串口中断标志 |

### 5.2.1 UART_Init

| 函数名 | **UART_Init** |
| --- | --- |
| 函数原形 | UART_Init(UART_InitTypeDef* UART_InitStruct) |
| 功能描述 | 初始化 UART 模块 |
| 输入参数 | UART_InitStruct： 串口初始化结构体 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**UART_InitTypeDef** 位于**uart.h**中，用于设置UART模块的工作状态，具体参数如下。

**UARTxMAP**

该参数选择待设置串口通道管脚，如下表

| UARTxMAP | 描述 |
| --- | --- |
| **UART0_RX_PA1_TX_PA2** | UART0 PA1 引脚作为 RX PA2 引脚作为 TX |
| **UART0_RX_PA14_TX_PA15** | UART0 PA14 引脚作为 RX PA15 引脚作为 TX |
| **UART0_RX_PB16_TX_PB17** | UART0 PB16 引脚作为 RX PB17 引脚作为 TX |
| **UART0_RX_PD6_TX_PD7** | UART0 PD6 引脚作为 RX PD7 引脚作为 TX |
| **UART1_RX_PE0_TX_PE1** | UART1 PE0 引脚作为 RX PE1 引脚作为 TX |
| **UART1_RX_PC3_TX_PC4** | UART1 PA14 引脚作为 RX PA15 引脚作为 TX |
| **UART2_RX_PD2_TX_PD3** | UART2 PD2 引脚作为 RX PD3 引脚作为 TX |
| **UART3_RX_PB10_TX_PB11** | UART3 PB10 引脚作为 RX PB11 引脚作为 TX |
| **UART3_RX_PC16_TX_PC17** | UART3 PC16 引脚作为 RX PC17 引脚作为 TX |
| **UART3_RX_PE4_TX_PE5** | UART3 PE4 引脚作为 RX PE5 引脚作为 TX |
| **UART4_RX_PE24_TX_PE25** | UART4 PE24 引脚作为 RX PE25 引脚作为 TX |

| | |
|---|---|
| **UART4_RX_PC14_TX_PC15** | UART4 PC14 引脚作为 RX　PC15 引脚作为 TX |

**UART_BaudRate**

该参数用来设置串口的通信速率

| **UART_BaudRate** | **描述** |
|---|---|
| **十进制数字** | 通信波特率 |

**例：**设置UART0 PA1引脚作为RX　PA2引脚作为TX，通信速度为115200，具体使用如下：

```
UART_InitTypeDef     UART_InitStruct1;
UART_InitStruct1. UARTxMAP = UART0_RX_PA1_TX_PA2;
UART_InitStruct1. UART_BaudRate = 115200;
UART_Init(& UART_InitStruct1);
```

## 5.2.2 UART_SendData

| **函数名** | **UART_SendData** |
|---|---|
| **函数原形** | UART_SendData(UART_Type* UARTx,uint8_t Data) |
| **功能描述** | 串口发送一个字节 |
| **输入参数 1** | UARTx: UART0~ UART4 |
| **输入参数 2** | Data:一字节数据 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**使用0端口进行数据发送，数据存储在data中，具体使用如下：

```
UART_SendData(UART0, Data);
```

## 5.2.3 UART_ReceiveData

| **函数名** | **UART_ReceiveData** |
|---|---|
| **函数原形** | UART_ReceiveData(UART_Type *UARTx,uint8_t *ch) |
| **功能描述** | 使用串口接收一个字节 |
| **输入参数 1** | UARTx: UART0~ UART4 |
| **输入参数 2** | ch:  一字节数据 |
| **输出参数** | 无 |
| **返回值** | 0：　失败；1：　成功 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**使用0端口进行数据接收，数据存储在data中，具体使用如下：

```
UART_SendData(UART0, Data);
```

## 5.2.4 UART_SendDataInt

| 函数名 | **UART_SendDataInt** |
|---|---|
| 函数原形 | UART_SendDataInt(UART_Type* UARTx,uint8_t* DataBuf,uint8_t Len) |
| 功能描述 | 使用中断方式发送数据 |
| 输入参数 1 | UARTx: UART0~ UART4 |
| 输入参数 2 | DataBuf:需要发送的数据地址 |
| 输入参数 3 | Len：数据长度，单位字节 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**使用端口0发送10字节的数据，数据存放在数组data[10]中，具体使用如下：

UART_SendDataInt(UART0, Data,10);

（**说明：**需要接收中断处理函数配合使用，具体使用请参见使用实例。）

## 5.2.5 DisplayCPUInfo

| 函数名 | **DisplayCPUInfo** |
|---|---|
| 函数原形 | DisplayCPUInfo(void) |
| 功能描述 | 使用串口打印芯片信息 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 使用 UART_DebugPortInit 函数对端口进行了配置 |
| 被调用函数 | 无 |

**例：**直接调用此函数即可。

## 5.2.6 UART_SendDataIntProcess

| 函数名 | **UART_SendDataIntProcess** |
|---|---|
| 函数原形 | UART_SendDataIntProcess(UART_Type* UARTx) |
| 功能描述 | 使用中断方式发送数据处理函数（存放在中断函数中） |
| 输入参数 | UARTx： UART0~ UART4 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**放在中断处理函数里即可。

## 5.2.7 UART_DMACmd

| 函数名 | **UART_DMACmd** |
|---|---|
| 函数原形 | UART_DMACmd(UART_Type* UARTx, uint16_t UART_DMAReq, FunctionalState NewState) |
| 功能描述 | 启用串口 DMA 功能 |
| 输入参数 1 | UARTx: UART0~ UART4 |
| 输入参数 2 | UART_DMAReq: UART_DMAReq_Tx：发送 DMA 功能<br>UART_DMAReq_Rx：接收 DMA 功能 |
| 输入参数 3 | NewState：ENABLE（使能），DISABLE（禁止） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | DMA 模块需配合使用 |
| 被调用函数 | 无 |

**例：**设置UART0模块发送触发DMA功能传输，（需DMA模块相关函数配合使用）。使用实例如下：

```
UART_DMACmd(UART0, UART_DMAReq_Tx, ENABLE);
```

## 5.2.8 UART_DebugPortInit

| 函数名 | **UART_DebugPortInit** |
|---|---|
| 函数原形 | UART_DebugPortInit(uint32_t UARTxMAP,uint32_t UART_BaudRate) |
| 功能描述 | 串口调试端口初始化配置 |
| 输入参数 1 | UARTxMAP: 引脚配置参数 |
| 输入参数 2 | UART_BaudRate: 串口通信速率设置 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**参见UART_Init函数的使用，主要用于使用printf函数。

## 5.2.9 UART_ITConfig

| 函数名 | **UART_ITConfig** |
|---|---|
| 函数原形 | UART_ITConfig(UART_Type* UARTx, uint16_t UART_IT, FunctionalState NewState) |
| 功能描述 | 串口中断配置 |
| 输入参数 1 | UARTx: UART0~ UART4 |
| 输入参数 2 | UART_IT:中断类型 |

| | |
|---|---|
| **输入参数 3** | NewState：ENABLE（使能），DISABLE（禁止） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**配置UART0端口使用接收中断，具体使用如下：

UART_ITConfig(UART0, UART_IT_RDRF, ENABLE);

## 5.2.10 UART_GetITStatus

| **函数名** | **UART_GetITStatus** |
|---|---|
| **函数原形** | UART_GetITStatus(UART_Type* UARTx, uint16_t UART_IT) |
| **功能描述** | 获得串口中断标志 |
| **输入参数 1** | UARTx: UART0~ UART4 |
| **输入参数 2** | UART_IT：中断类型 |
| **输出参数** | 无 |
| **返回值** | 中断状态 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**获得UART0端口的接收中断标志，存储在statue中，具体使用如下：

Statue = UART_GetITStatus(UART_Type* UARTx, uint16_t UART_IT);

# 6. 周期中断定时器(PIT)

周期中断寄存器（PIT）是一组可以产生中断和触发 DMA 通道的定时器。K60 中共包含 4 路 PIT，它的的 PIT 周期中断定时器可以产生 DMA 请求脉冲和定时中断，所有中断都是可屏蔽的，而且每个定时器都有各自的中断溢出周期。

## 6.1 PIT 模块主要寄存器结构

**PIT**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| MCR | 模块控制寄存器 |
| LDVAL0 | 定时器加载寄存器 |
| CVAL0 | 定时器当前值寄存器 |
| TCTRL0 | 定时器控制寄存器 |
| TFLG0 | 定时器标志寄存器 |

## 6.2 PIT 库函数

PIT 库函数

| 函数名 | 描述 |
| --- | --- |
| **PIT_Init** | PIT 模块初始化配置 |
| **PIT_GetLoadValue** | 获得预设计时器的值 |
| **PIT_GetCurrentValue** | 获得当前计时器的值 |
| **PIT_SetLoadValue** | 设置预设计时器的值 |
| **PIT_Start** | 开启 PIT 模块计时器 |
| **PIT_Stop** | 停止 PIT 模块计时器 |
| **PIT_ITConfig** | PIT 模块中断配置 |
| **PIT_GetITStatus** | 获得 PIT 模块中断标志 |
| **PIT_ClearITPendingBit** | 清除 PIT 模块中断标志位 |

### 6.2.1 PIT_Init

| 函数名 | **PIT_Init** |
| --- | --- |
| 函数原形 | PIT_Init(PIT_InitTypeDef* PIT_InitStruct) |
| 功能描述 | PIT 模块初始化配置 |
| 输入参数 | PIT_InitStruct： 存储 PIT 模块工作参数的结构体 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**PIT_InitTypeDef** 位于**pit.h**中，用于设置PIT模块的工作状态，具体参数如下。

**PITx**

该参数选择待设置的模块通道，如下表

| PITx | 描述 |
| --- | --- |
| **PIT0~ PIT3** | PIT 模块的 0~3 通道 |

**PIT_Interval**

该参数用来设置PIT通道的周期时间间隔，单位为毫秒

| PIT_Interval | 描述 |
| --- | --- |
| **1~++** | 填入数字 1~正无穷，单位为毫秒 |

**例：**配置PIT2模块产生500毫秒的周期时间，具体使用如下：

```
PIT_InitTypeDef   PIT_InitStruct1;//申请的结构体变量
PIT_InitStruct1.PITx = PIT2;        //使用PIT中的2通道
PIT_InitStruct1.PIT_Interval = 500;    //设定时间间隔为500毫秒
PIT_Init(&PIT_InitStruct1); //将数据传递到函数中
```

## 6.2.2 PIT_GetLoadValue

| 函数名 | **PIT_GetLoadValue** |
|---|---|
| 函数原形 | PIT_GetLoadValue(uint8_t PITx) |
| 功能描述 | 获得预设计时器的值 |
| 输入参数 | PITx： PIT0~PIT3 |
| 输出参数 | 无 |
| 返回值 | 当前的预设计时值 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获取PIT0模块的预设计时器的值，存储在value中，具体使用情况如下：

```
Value = PIT_GetLoadValue(PIT0);
```

## 6.2.3 PIT_GetCurrentValue

| 函数名 | **PIT_GetCurrentValue** |
|---|---|
| 函数原形 | PIT_GetCurrentValue(uint8_t PITx) |
| 功能描述 | 获得当前计时器的值 |
| 输入参数 | PITx： PIT0~PIT3 |
| 输出参数 | 无 |
| 返回值 | 当前的计时值 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获取PIT0模块的当前计时器的值，存储在value中，具体使用情况如下：

```
Value = PIT_ GetCurrentValue (PIT0);
```

## 6.2.4 PIT_SetLoadValue

| 函数名 | **PIT_SetLoadValue** |
|---|---|
| 函数原形 | PIT_SetLoadValue(uint8_t PITx, uint32_t Value) |
| 功能描述 | 设置预设计时器的值 |
| 输入参数 **1** | PITx: PIT0~PIT3 |
| 输入参数 **2** | Value:计时时间 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**设置PIT2模块的周期时间为500毫秒，具体使用如下：

PIT_SetLoadValue(PIT0, 500);

## 6.2.5 PIT_Start

| 函数名 | **PIT_Start** |
|---|---|
| 函数原形 | PIT_Start(uint8_t PITx) |
| 功能描述 | 开启 PIT 模块计时器 |
| 输入参数 | PITx:　PIT0~PIT3 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | PIT 模块的时间已经设置好 |
| 被调用函数 | 无 |

**例：**开启PIT2模块，进行计时，具体使用如下：

PIT_Start(PIT2);

## 6.2.6 PIT_Stop

| 函数名 | **PIT_Stop** |
|---|---|
| 函数原形 | PIT_Stop(uint8_t PITx) |
| 功能描述 | 停止 PIT 模块计时器 |
| 输入参数 | PITx：　PIT0~PIT3 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**关闭PIT2模块，进行计时，具体使用如下：

PIT_ Stop (PIT2);

## 6.2.7 PIT_ITConfig

| 函数名 | **PIT_ITConfig** |
|---|---|
| 函数原形 | PIT_ITConfig(uint8_t PITx, uint16_t PIT_IT, FunctionalState NewState) |
| 功能描述 | PIT 模块中断配置 |
| 输入参数 1 | PITx: PIT0~PIT3 |
| 输入参数 2 | PIT_IT:　PIT_IT_TIF（定时中断） |
| 输入参数 3 | NewState：ENABLE（使能），DISABLE（禁止） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 中断处理函数已经写好 |
| 被调用函数 | 无 |

**例：**开启PIT2模块的周期性定时中断，具体使用如下：

PIT_ITConfig(PIT2,PIT_IT_TIF,ENABLE);

**注：**此函数需要配合中断处理函数配合使用，具体函数存储在 isr.c 文件中，函数的名字是固定的不可随意更改，用户可根据自己的需求在相应的函数中加入自己的程序。中断处理函数名字为：**void PITn_IRQHandler(void)**（注：n 值为 0~3 对应的模块通道）。

### 6.2.8 PIT_GetITStatus

| 函数名 | **PIT_GetITStatus** |
|---|---|
| 函数原形 | PIT_GetITStatus(uint8_t PITx, uint16_t PIT_IT) |
| 功能描述 | 获得 PIT 模块中断标志 |
| 输入参数 1 | PITx: PIT0~PIT3 |
| 输入参数 2 | PIT_IT: PIT_IT_TIF（定时中断） |
| 输出参数 | 无 |
| 返回值 | 指定中断标志的状态 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获得PIT2模块的定时中断中断标志状态，存储在statue中，具体使用如下：

Statue = PIT_GetITStatus(PIT2, PIT_IT_TIF);

### 6.2.9 PIT_ClearITPendingBit

| 函数名 | **PIT_ClearITPendingBit** |
|---|---|
| 函数原形 | PIT_ClearITPendingBit(uint8_t PITx,uint16_t PIT_IT) |
| 功能描述 | 清除 PIT 模块中断标志位 |
| 输入参数 1 | PITx: PIT0~PIT3 |
| 输入参数 2 | PIT_IT: PIT_IT_TIF（定时中断） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**清除PIT2模块的定时中断中断标志状态，具体使用如下：

PIT_ ClearITPendingBit (PIT2, PIT_IT_TIF);

## 7. 实时时钟(RTC)

实时时钟（RTC）模块是一个独立供电的模块，主要包含一个外部晶体振荡器，一个 POR 块，RTC 计时器以及自身的软件复位控制位。K60 的 RTC 实时时钟自带一个具有补偿功能

的 16 位预分频器和一个 32 位警告的秒计数器，此外具有寄存器写入保护功能，访问控制寄存器时需要系统复位，一边进行读取或者写入访问。

# 7.1 RTC 模块主要寄存器结构

**RTC**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| TSR | 实时时钟秒寄存器 |
| TPR | 定时器加载寄存器 |
| TAR | 定时器当前值寄存器 |
| CR | 定时器控制寄存器 |
| SR | 定时器标志寄存器 |

# 7.2 RTC 库函数

RTC 库函数

| 函数名 | 描述 |
|---|---|
| **RTC_Init** | RTC 时钟初始化配置 |
| **RTC_SecondIntProcess** | RTC 秒中断处理 |
| **RTC_ReadData** | 获取时间 |
| **RTC_SetData** | 设置时间 |

## 7.2.1 RTC_Init

| 函数名 | **RTC_Init** |
|---|---|
| **函数原形** | RTC_Init(void) |
| **功能描述** | RTC 实时时钟初始化配置 |
| **输入参数** | 无 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 具有必要的硬件资源 |
| **被调用函数** | 无 |

**例：**具体使用详见使用实例。

## 7.2.2 RTC_SecondIntProcess

| 函数名 | **RTC_SecondIntProcess** |
|---|---|
| **函数原形** | RTC_SecondIntProcess(void) |
| **功能描述** | 实时时钟秒中断处理 |
| **输入参数** | 无 |

| | |
|---|---|
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 已经进行了秒中断配置，放在秒中断中使用 |
| **被调用函数** | 无 |

**例：**具体使用详见使用实例。

## 7.2.3 RTC_ ReadData

| **函数名** | **RTC_ ReadData** |
|---|---|
| **函数原形** | RTC_ ReadData (RTC_CalanderTypeDef * RTC_CalanderStruct) |
| **功能描述** | 获取时间 |
| **输入参数** | RTC_CalanderStruct： 存储时间的结构体 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**RTC_CalanderTypeDef** 位于**rtc.h**中，用于存储实时时钟的工作状态和日期，具体参数如下。

| **参数** | **描述** |
|---|---|
| **Hour** | 存储小时 |
| **Minute** | 存储分钟 |
| **Second** | 存储秒 |
| **Month** | 存储月 |
| **Date** | 存储日 |
| **Week** | 存储周 |
| **Year** | 存储年 |
| **TSRValue** | 存储 RTC 计数器计数值 |

**例：**具体使用详见使用实例。

## 7.2.4 RTC_ SetData

| **函数名** | **RTC_ SetData** |
|---|---|
| **函数原形** | RTC_ SetData (RTC_CalanderTypeDef * RTC_CalanderStruct) |
| **功能描述** | 设置时间 |
| **输入参数** | RTC_CalanderStruct： 设置时间的结构体 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**具体使用详见使用实例。

## 7.3 RTC 使用实例

此实例展示RTC时钟的使用方法，时钟的时间数据每个一秒自动更新一次无需干涉，用户仅需读取存储时间的结构体即可。具体使用如下：

1、 完成 RTC 模块初始化配置

RTC_CalanderTypeDef myRTC;//申请时间结构体变量

RTC_Init( );

2、 设置时间为2013年7月29日12点12分12秒

myRTC.Hour =12;

myRTC.Minute = 12;

myRTC.Second = 12;

myRTC.Year = 2013;

myRTC.Month = 7;

myRTC.Date = 29;

带入函数中：

RTC_ SetData (myRTC)；

3、 读取myRTC结构体中的相应数据即可获得时间。

（**注：**需要把**RTC_SecondIntProcess()**函数放在RTC秒中断处理函数中，具体见文件irs.c文件的**void RTC_IRQHandler(void)**函数中。）

# 8. 内部集成电路总线(I2C)

I2C 总线主要用于与其它 IC 设备通信，采用串行总线方式进行数据的传输。I2C 总线一共有两根通信线，分别为双向串行数据线 SDA 和双向串行时钟线 SCL，K60 微处理器中包含 2 个 I2C 模块，分别为 I2C0 和 I2C1，芯片通信引脚可通过相关的模块配置来设置复用情况。

## 8.1 I2C 模块主要寄存器结构

**I2C**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| A1 | 设置 I2C 模块的从地址寄存器 |
| F | I2C 通信频率设置寄存器 |
| C1 | I2C 控制寄存器 1 |
| S | I2C 状态寄存器 |
| D | I2C 发送与接收数据寄存器 |
| C2 | I2C 控制寄存器 1 |
| D | I2C 发送与接收数据寄存器 |
| C2 | I2C 控制寄存器 2 |

## 8.2 I2C 库函数

I2C 库函数

| 函数名 | 描述 |
| --- | --- |
| **I2C_Init** | 初始化配置 I2C 模块工作模式 |
| **I2C_GenerateSTART** | 控制 I2C 模块产生一个开始信号 |
| **I2C_GenerateRESTART** | 控制 I2C 模块产生再次开始信号（用于通信中） |
| **I2C_GenerateSTOP** | 控制 I2C 模块产生一个停止信号 |
| **I2C_SendData** | 控制 I2C 模块发送一字节数据 |
| **I2C_Send7bitAddress** | 控制 I2C 模块发送 7 位从地址 |
| **I2C_WaitAck** | 控制 I2C 模块等待应答信号 |
| **I2C_SetMasterMode** | 设置 I2C 模块工作在主机模式 |
| **I2C_GenerateAck** | 控制 I2C 模块产生一个应答信号 |
| **I2C_EnableAck** | 控制 I2C 模块使能应答信号，读取一字节后返回应答信号 |
| **I2C_ITConfig** | 配置 I2C 模块的中断情况 |
| **I2C_GetITStatus** | 获得 I2C 模块的中断标志位状态 |
| **I2C_DMACmd** | 使能 I2C 模块的 DMA 功能 |
| **I2C_ClearITPendingBit** | 清除 I2C 模块的中断标志位 |

## 8.2.1 I2C_Init

| 函数名 | **I2C_Init** |
| --- | --- |
| 函数原形 | I2C_Init(I2C_InitTypeDef* I2C_InitStruct) |
| 功能描述 | 初始化配置 I2C 模块工作模式 |
| 输入参数 | I2C_InitStruct： 存储控制 I2C 模块配置的数据结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**I2C_InitTypeDef** 位于**i2c.h**中，用于设置i2c模块的工作状态，具体参数如下。

**I2CxMAP**

该参数选择待设置的通道管脚映射，如下表

| I2CxMAP | 描述 |
| --- | --- |
| **I2C0_SCL_PB0_SDA_PB1** | I2C0 模块，SCL 信号 PORTB 端口的 0 引脚；SDA 信号 PORTB 端口的 1 引脚 |
| **I2C0_SCL_PB2_SDA_PB3** | I2C0 模块，SCL 信号 PORTB 端口的 2 引脚；SDA 信号 PORTB 端口的 3 引脚 |
| **I2C1_SCL_PE1_SDA_PE0** | I2C1 模块，SCL 信号 PORTE 端口的 1 引脚；SDA 信号 PORTE 端口的 0 引脚 |
| **I2C1_SCL_PC10_SDA_PC11** | I2C1 模块，SCL 信号 PORTC 端口的 10 引脚；SDA 信号 PORTC 端口的 11 引脚 |

**I2C_ClockSpeed**

该参数用来设置I2C模块的通信速度，具体参数如下：

| I2C_ClockSpeed | 描述 |
|---|---|
| **I2C_CLOCK_SPEED_50KHZ** | 设置通信速度为 50KHz |
| **I2C_CLOCK_SPEED_100KHZ** | 设置通信速度为 100KHz |
| **I2C_CLOCK_SPEED_150KHZ** | 设置通信速度为 150KHz |
| **I2C_CLOCK_SPEED_200KHZ** | 设置通信速度为 200KHz |
| **I2C_CLOCK_SPEED_250KHZ** | 设置通信速度为 250KHz |
| **I2C_CLOCK_SPEED_300KHZ** | 设置通信速度为 300KHz |

**例：**配置I2C1模块工作在主机模式，通信速率为200KHz，使用芯片的PORTC端口的10和11引脚作为SCL 、SDA信号。具体使用如下：

```
I2C_InitTypeDef    I2C_InitStruct1; //申请I2C结构变量，存储I2C模块的工作方式
I2C_InitStruct1.I2CxMAP   =   I2C1_SCL_PC10_SDA_PC11;//配置芯片的SCL/SDA信号引脚
I2C_InitStruct1.I2C_ClockSpeed   =   I2C_CLOCK_SPEED_200KHZ;//设置模块的通信速
                                     度为200KHz,模块的时钟源为BUS_Clock。
I2C_Init(&I2C_InitStruct1); //调用初始化函数，将申请的结构体地址带入函数。
```

## 8.2.2 I2C_GenerateSTART

| 函数名 | **I2C_GenerateSTART** |
|---|---|
| **函数原形** | I2C_GenerateSTART(I2C_Type *I2Cx) |
| **功能描述** | 控制 I2C 模块产生一个开始信号 |
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块产生一个开始信号，一般配合其它函数使用。具体使用见I2C使用实例

8.2.3 I2C_GenerateRESTART

| 函数名 | **I2C_GenerateRESTART** |
|---|---|
| **函数原形** | I2C_GenerateRESTART(I2C_Type *I2Cx) |
| **功能描述** | 控制 I2C 模块产生再次开始信号（用于通信中） |
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | 无 |

| | |
|---|---|
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块再次产生一个开始信号，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.4 I2C_GenerateSTOP

| **函数名** | **I2C_GenerateSTOP** |
|---|---|
| **函数原形** | I2C_GenerateSTOP(I2C_Type *I2Cx) |
| **功能描述** | 控制 I2C 模块产生一个停止信号 |
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块产生一个停止信号，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.5 I2C_SendData

| **函数名** | **I2C_SendData** |
|---|---|
| **函数原形** | I2C_SendData(I2C_Type *I2Cx,uint8_t data8) |
| **功能描述** | 控制 I2C 模块发送一字节数据 |
| **输入参数 1** | I2Cx: I2C0，I2C1 |
| **输入参数 2** | data8: 要发送的一字节数据 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块发送一字节的数据，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.6 I2C_Send7bitAddress

| **函数名** | **I2C_Send7bitAddress** |
|---|---|
| **函数原形** | I2C_Send7bitAddress(I2C_Type* I2Cx, uint8_t Address, uint8_t I2C_Direction) |
| **功能描述** | 控制 I2C 模块发送 7 位从地址 |
| **输入参数 1** | I2Cx: I2C0，I2C1 |
| **输入参数 2** | Address: 需要通信的 IC 芯片地址 |
| **输入参数 3** | I2C_Direction：I2C_MASTER_WRITE（写数据），I2C_MASTER_READ（读数据） |

| | |
|---|---|
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块发送7位从地址，及数据的流向，是写数据还是读数据，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.7 I2C_WaitAck

| **函数名** | **I2C_WaitAck** |
|---|---|
| **函数原形** | I2C_WaitAck(I2C_Type *I2Cx) |
| **功能描述** | 控制 I2C 模块等待应答信号 |
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | TRUE（1），接收到应答信号，FALSE（0），接收应答信号失败 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**控制I2C模块等待应答信号，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.8 I2C_SetMasterMode

| **函数名** | **I2C_SetMasterMode** |
|---|---|
| **函数原形** | I2C_SetMasterMode(I2C_Type* I2Cx,uint8_t I2C_Direction) |
| **功能描述** | 设置 I2C 读写模式 |
| **输入参数 1** | I2Cx: I2C0，I2C1 |
| **输入参数 2** | I2C_Direction: I2C_MASTER_WRITE（主机写），I2C_MASTER_READ（主机读） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置I2C模块读写模式，一般配合其它函数使用。具体使用见I2C使用实例

## 8.2.9 I2C_GenerateAck

| **函数名** | **I2C_GenerateAck** |
|---|---|
| **函数原形** | I2C_GenerateAck(I2C_Type *I2Cx) |
| **功能描述** | 设置 I2C 模块产生一个应答信号 |

| | |
|---|---|
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置I2C模块读写模式，一般配合其它函数使用。具体使用见I2C使用实例。

## 8.2.10 I2C_EnableAck

| **函数名** | **I2C_EnableAck** |
|---|---|
| **函数原形** | I2C_EnableAck(I2C_Type *I2Cx) |
| **功能描述** | 设置 I2C 模块读取一字节数据后返回一个应答信号 |
| **输入参数** | I2Cx： I2C0，I2C1 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置I2C模块读取一字节数据后返回一个应答信号，一般配合其它函数使用。具体使用见I2C使用实例。

## 8.2.11 I2C_ITConfig

| **函数名** | **I2C_ITConfig** |
|---|---|
| **函数原形** | I2C_ITConfig(I2C_Type* I2Cx, uint16_t I2C_IT, FunctionalState NewState) |
| **功能描述** | 配置 I2C 模块的中断情况 |
| **输入参数 1** | I2Cx: I2C0，I2C1 |
| **输入参数 2** | I2C_IT: I2C_IT_TCF 传输完成中断 |
| **输入参数 3** | NewState： ENABLE（使能），DISABLE（禁止） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置I2C模块的中断情况，数据发送完成时可产生中断信号，一般配合其它函数使用。具体使用见I2C使用实例。

## 8.2.12 I2C_GetITStatus

| **函数名** | **I2C_GetITStatus** |
|---|---|

| | |
|---|---|
| **函数原形** | I2C_GetITStatus(I2C_Type* I2Cx, uint16_t I2C_IT) |
| **功能描述** | 获得 I2C 模块的中断标志位状态 |
| **输入参数 1** | I2Cx:　I2C0，I2C1 |
| **输入参数 2** | I2C_IT: I2C_IT_TCF 中断源 |
| **输出参数** | 无 |
| **返回值** | 0，未发生；　1，产生中断信号 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**获得I2C模块的中断标志位状态，作为辅助函数使用，不经常使用，主要用于通信故障诊断。

## 8.2.13 I2C_DMACmd

| | |
|---|---|
| **函数名** | **I2C_DMACmd** |
| **函数原形** | I2C_DMACmd(I2C_Type* I2Cx, uint16_t I2C_DMAReq, FunctionalState NewState) |
| **功能描述** | 使能 I2C 模块的 DMA 功能 |
| **输入参数 1** | I2Cx:　I2C0，I2C1 |
| **输入参数 2** | I2C_DMAReq: I2C_DMAReq_TCF　触发 DMA 中断源 |
| **输入参数 3** | NewState：ENABLE（使能），DISABLE（禁止） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 设置好 DMA 模块的工作方式。 |
| **被调用函数** | 无 |

**例：**设置I2C1模块发送完成后自动触发DMA功能继续传输，（需DMA模块相关函数配合使用）。使用实例如下：

```
I2C_DMACmd(I2C1, I2C_DMAReq_TCF, ENABLE);
```

## 8.2.14 I2C_ClearITPendingBit

| | |
|---|---|
| **函数名** | **I2C_ClearITPendingBit** |
| **函数原形** | I2C_ClearITPendingBit(I2C_Type* I2Cx, uint16_t I2C_IT) |
| **功能描述** | 清除 I2C 模块的中断标志位 |
| **输入参数 1** | I2Cx:　I2C0，I2C1 |
| **输入参数 2** | I2C_IT:　清除指定的中断标志位，I2C_IT_TCF |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

例：清除I2C1模块的传输完成中断标志位，作为辅助函数使用，一般不使用，具体使用如下：

I2C_ClearITPendingBit(I2C1, I2C_IT_TCF)；

## 8.3 I2C 使用实例：

对于 I2C 的使用遵循具体的 IC 通信手册，在这里通过配合上述函数封装成 I2C 通信的常用函数，具体步骤及方法如下：

1. 配置 I2C 模块通讯类型

   I2C_Init(&I2C_InitStruct1);（具体使用参见上文）

2. 向指定 IC 的寄存器中写数据

   实现上述功能需要多个函数配合使用，具体如下：

   I2C_GenerateSTART(I2C1);//使用 I2C1 模块产生开始信号

   I2C_Send7bitAddress(I2C1,ADDRESS, WRITE);//使用 I2C1 模块需找指定地址的 IC 器件

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_SendData(I2C1, RegisterAddress);//指定准备向 IC 器件写入数据的寄存器地址

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_SendData(I2C1,Data); //向刚才指定的 IC 的寄存器写入数据

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_GenerateSTOP(I2C1);//控制 I2C1 模块产生停止信号，结束通信。

   注：按照上述顺序即可封装成一个 I2C 写数据函数

3. 读取 IC 中指定寄存器的数据

   实现上述功能需要多个函数配合使用，具体如下：

   I2C_GenerateSTART(I2C1);//使用 I2C1 模块产生开始信号

   I2C_Send7bitAddress(I2C1,ADDRESS, WRITE);//使用 I2C1 模块需找指定地址的 IC 器件

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_SendData(I2C1, RegisterAddress);//指定准备读取 IC 器件的寄存器地址

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_GenerateRESTART(I2C1);//再次产生开始信号

   I2C_Send7bitAddress(I2C1,ADDRESS,READ);//读取 I2C 器件的数据

   I2C_WaitAck(I2C1);// 等待指定 IC 器件产生应答信号

   I2C_SetMasterMode(I2C1,I2C_MASTER_READ);//设置 I2C 模块处于接收模式

   I2C_GenerateAck(I2C1);//产生一个应答信号

   I2C_WaitAck(I2C1); // 等待指定 IC 器件产生应答信号

   I2C_GenerateSTOP(I2C1); //控制 I2C1 模块产生停止信号，结束通信。

   result = I2Cx->D;//读取 I2C 模块接收到的数据，存储在 result 中。

# 9. 串行外设总线(SPI)

SPI 总线是一种四线制的同步串行总线接口，主要用于和外围扩展芯片之间的信息交流。它由串行时钟线 SCLK、主机输入/从机输出数据线 MISO、主机输出／从机输入数据线 MOSI 和从机片选信号 CS 组成。SPI 总线采用的是单端非平衡的传输方式，传输数据位的电压电平是以公共地作为参考端的。Kinetis 芯片中支持 3 个 SPI 模块，每个模块可支持 6 个外设芯片。

# 9.1 SPI 模块主要寄存器结构

**SPI**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| MCR | SPI 模块控制寄存器 |
| CTAR0 | 速率和传输属性寄存器 0 |
| CTAR1 | 速率和传输属性寄存器 1 |
| SR | 状态寄存器 |
| RSER | 中断类型选择和使能寄存器 |
| PUSHR | 数据发送和设置寄存器 |
| POPR | 数据接收寄存器 |

**注：**此列表主要针对 SPI 主模式下所需的主要寄存器

# 9.2 SPI 库函数

SPI 库函数

| 函数名 | 描述 |
|---|---|
| **SPI_Init** | SPI 模块初始化设置 |
| **SPI_ReadWriteByte** | SPI 模块读写一次数据 |
| **SPI_ITConfig** | SPI 中断配置寄存器 |
| **SPI_GetITStatus** | 获得 SPI 模块中断标志 |
| **SPI_ClearITPendingBit** | 清除中断标志位 |
| **SPI_DMACmd** | 设置 SPI 处于 DMA 传输模式 |

## 9.2.1 SPI_Init

| 函数名 | **SPI_Init** |
|---|---|
| 函数原形 | SPI_Init(SPI_InitTypeDef* SPI_InitStruct) |
| 功能描述 | SPI 模块初始化设置 |
| 输入参数 | SPI_InitStruct （配置 SPI 工作的数据结构体） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**SPI_InitTypeDef** 位于**spi.h**中，用于设置SPI模块的工作状态，具体参数如下。

**SPIxDataMap**

该参数选择待设置的数据管脚映射，如下表

| SPIxDataMap | 描述 |
|---|---|
| SPI0_SCK_PA15_SOUT_PA16_SIN_PA17 | SPI0 模块，SCK 信号 PORTA 端口的 15 引脚；SOUT 信号 PORTA 端口的 16 引脚；SIN 信号 PORTA 端口的 17 引脚 |
| SPI0_SCK_PC5_SOUT_PC6_SIN_PC7 | SPI0 模块，SCK 信号 PORTC 端口的 5 引脚；SOUT 信号 PORTC 端口的 6 引脚；SIN 信号 PORTC 端口的 7 引脚 |
| SPI0_SCK_PD1_SOUT_PD2_SIN_PD3 | SPI0 模块，SCK 信号 PORTD 端口的 1 引脚；SOUT 信号 PORTD 端口的 2 引脚；SIN 信号 PORTD 端口的 3 引脚 |
| SPI1_SCK_PE2_SOUT_PE1_SIN_PE3 | SPI1 模块，SCK 信号 PORTE 端口的 2 引脚；SOUT 信号 PORTE 端口的 1 引脚；SIN 信号 PORTE 端口的 3 引脚 |
| SPI1_SCK_PB11_SOUT_PB16_SIN_PB17 | SPI1 模块，SCK 信号 PORTB 端口的 11 引脚；SOUT 信号 PORTB 端口的 16 引脚；SIN 信号 PORTB 端口的 17 引脚 |
| SPI2_SCK_PB21_SOUT_PB22_SIN_PB23 | SPI2 模块，SCK 信号 PORTB 端口的 21 引脚；SOUT 信号 PORTB 端口的 22 引脚；SIN 信号 PORTB 端口的 23 引脚 |

**SPIxPCSMap**

该参数用来设置SPI模块的片选引脚，具体参数如下：

| SPIxPCSMap | 描述 |
|---|---|
| SPI0_PCS0_PA14 | SPI0 模块片选通道 0，PORTA 端口 14 引脚 |
| SPI0_PCS1_PC3 | SPI0 模块片选通道 1，PORTC 端口 3 引脚 |
| SPI0_PCS2_PC2 | SPI0 模块片选通道 2，PORTC 端口 2 引脚 |
| SPI0_PCS3_PC1 | SPI0 模块片选通道 3，PORTC 端口 1 引脚 |
| SPI0_PCS4_PC0 | SPI0 模块片选通道 4，PORTC 端口 0 引脚 |
| SPI1_PCS0_PB10 | SPI1 模块片选通道 0，PORTB 端口 10 引脚 |
| SPI1_PCS1_PB9 | SPI1 模块片选通道 1，PORTB 端口 9 引脚 |
| SPI1_PCS2_PE5 | SPI1 模块片选通道 2，PORTE 端口 5 引脚 |
| SPI1_PCS3_PE6 | SPI1 模块片选通道 3，PORTE 端口 6 引脚 |
| SPI2_PCS0_PB20 | SPI2 模块片选通道 0，PORTB 端口 20 引脚 |

**SPI_DataSize**

该参数用来设置SPI模块的数据大小，具体参数如下：

| SPI_DataSize | 描述 |
|---|---|
| 8 | 一次数据发送或接收一字数据 |

**SPI_CPOL**

该参数用来设置SPI模块的时钟信号极性，具体参数如下：

| SPI_CPOL | 描述 |
|---|---|
| SPI_CPOL_Low | SPI 模块时钟信号低电平有效 |
| SPI_CPOL_High | SPI 模块时钟信号高电平有效 |

**SPI_Mode**

该参数用来设置SPI模块的主从模式，具体参数如下：

| SPI_Mode | 描述 |
|---|---|

| | |
|---|---|
| **SPI_Mode_Master** | SPI 模块处于主机模式 |
| **SPI_Mode_Slave** | SPI 模块处于从机模式 |

**SPI_CPHA**

该参数用来设置SPI模块的时钟相位，具体参数如下：

| **SPI_CPHA** | **描述** |
|---|---|
| **SPI_CPHA_1Edge** | SPI 模块的时钟相位第一边缘 |
| **SPI_CPHA_2Edge** | SPI 模块的时钟相位第二边缘 |

**SPI_BaudRatePrescaler**

该参数用来设置SPI模块的波特率，具体参数如下：

| **SPI_BaudRatePrescaler** | **描述** |
|---|---|
| **SPI_BaudRatePrescaler_2** | SPI 模块通信速率进行 2 分频 |
| **SPI_BaudRatePrescaler_4** | SPI 模块通信速率进行 4 分频 |
| **SPI_BaudRatePrescaler_6** | SPI 模块通信速率进行 6 分频 |
| **SPI_BaudRatePrescaler_8** | SPI 模块通信速率进行 8 分频 |
| **SPI_BaudRatePrescaler_16** | SPI 模块通信速率进行 16 分频 |
| **SPI_BaudRatePrescaler_32** | SPI 模块通信速率进行 32 分频 |
| **SPI_BaudRatePrescaler_64** | SPI 模块通信速率进行 64 分频 |
| **SPI_BaudRatePrescaler_128** | SPI 模块通信速率进行 128 分频 |
| **SPI_BaudRatePrescaler_256 …** | SPI 模块通信速率进行 256 分频。。。 |
| **… SPI_BaudRatePrescaler_2048** | 。。。SPI 模块通信速率进行 2048 分频 |

**SPI_FirstBit**

该参数用来设置SPI模块数据高位还是低位发送，具体参数如下：

| **SPI_FirstBit** | **描述** |
|---|---|
| **SPI_FirstBit_MSB** | SPI 模块的高位数据优先发送 |
| **SPI_FirstBit_LSB** | SPI 模块的低位数据优先发送 |

**例：**使用SPI0模块中的PORTA14、15、16、17引脚，数据尺寸为8位，高位优先，速度2分频，主机模式，数据在第一个时钟沿有效，设置时钟信号在空闲时为低电平，具体使用如下：

```
SPI_InitTypeDef SPI_InitStruct1;//申请结构变量
SPI_InitStruct1.SPIxDataMap = SPI0_SCK_PA15_SOUT_PA16_SIN_PA17;
SPI_InitStruct1.SPIxPCSMap = SPI0_PCS0_PA14; //选择通信引脚
SPI_InitStruct1.SPI_DataSize = 8;//设置8位数据结构
SPI_InitStruct1.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;//设置速度为2分频
SPI_InitStruct1.SPI_Mode = SPI_Mode_Master;//设置SPI为主模式
SPI_InitStruct1.SPI_CPHA = SPI_CPHA_1Edge;//设置SPI在1个时钟沿数据有效
SPI_InitStruct1.SPI_CPOL = SPI_CPOL_Low;//设置时钟线在空闲时为低电平
SPI_InitStruct1.SPI_FirstBit = SPI_FirstBit_MSB;//设置SPI通信为高位优先原则
```

SPI_Init(&SPI_InitStruct1); //将上述数据传入SPI初始化函数，完成工作配置

## 9.2.2 SPI_ReadWriteByte

| 函数名 | **SPI_ReadWriteByte** |
|---|---|
| 函数原形 | SPI_ReadWriteByte(uint32_t SPICSMap,uint16_t Data,uint16_t PCS_State) |
| 功能描述 | SPI 读写一次数据 |
| 输入参数 1 | SPICSMap: 芯片的 SPI 模块引脚复用配置 |
| 输入参数 2 | Data: 需要发送的一字节数据 |
| 输入参数 3 | PCS_State：数据发送完成后，片选信号的电平状态 |
| 输出参数 | 无 |
| 返回值 | 读取的一字节数据 |
| 先决条件 | 首先对 SPI 模块初始化设置 |
| 被调用函数 | 无 |

例：1.写一字节数据，结束后片选信号为低电平

SPI_ReadWriteByte(SPI0_SCK_PA15_SOUT_PA16_SIN_PA17, Data, SPI_PCS_Inactive);

2.读取一字节数据，结束后片选信号为低电平

Read = SPI_ReadWriteByte(SPI0_SCK_PA15_SOUT_PA16_SIN_PA17, 0, SPI_PCS_Inactive);

（读取的数据通过函数返回，存储在变量 Read 里）

## 9.2.3 SPI_ITConfig

| 函数名 | **SPI_ ITConfig** |
|---|---|
| 函数原形 | SPI_ ITConfig (SPI_Type* SPIx, uint16_t SPI_IT, FunctionalState NewState) |
| 功能描述 | SPI 模块中断配置 |
| 输入参数 | SPIx： SPI0、SPI1、SPI2<br>SPI_IT：SPI 模块中断类型<br>NewState：设置中断类型状态，ENABLE（开启）<br>DISABLE（关闭） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：配置SPI0模块数据发送完成时产生发送完成中断

SPI_ ITConfig (SPI0, SPI_IT_TCF, ENABLE);

## 9.2.4 SPI_GetITStatus

| 函数名 | **SPI_GetITStatus** |
|---|---|
| 函数原形 | SPI_ GetITStatus (SPI_Type* SPIx, uint16_t SPI_IT) |
| 功能描述 | 获得 SPI 模块的中断标志 |
| 输入参数 | SPIx： SPI0、SPI1、SPI2<br>SPI_IT：SPI 模块中断类型 |
| 输出参数 | 无 |
| 返回值 | 中断标志状态 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获取SPI0模块数据的发送完成中断标志位状态

Status = SPI_ GetITStatus (SPI0, SPI_IT_TCF);
（status 中存储着中断标志的状态，0 或者 1）

## 9.2.5 SPI_ClearITPendingBit

| 函数名 | **SPI_ClearITPendingBit** |
|---|---|
| 函数原形 | SPI_ ClearITPendingBit (SPI_Type* SPIx, uint16_t SPI_IT) |
| 功能描述 | 清除 SPI 模块的中断标志 |
| 输入参数 | SPIx： SPI0、SPI1、SPI2<br>SPI_IT：SPI 模块中断类型 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**清除SPI0模块数据的发送完成中断标志位状态

SPI_ ClearITPendingBit (SPI0, SPI_IT_TCF);

## 9.2.5 SPI_DMACmd

| 函数名 | **SPI_DMACmd** |
|---|---|
| 函数原形 | SPI_DMACmd(SPI_Type* SPIx, uint16_t SPI_DMAReq, FunctionalState NewState) |
| 功能描述 | 使能 SPI 模块的 DMA 功能 |
| 输入参数 1 | SPIx: SPI0，SPI1，SPI2 |
| 输入参数 2 | SPI_DMAReq: SPI_DMAReq_TCF 触发 DMA 中断源 |
| 输入参数 3 | NewState：ENABLE（使能），DISABLE（禁止） |
| 输出参数 | 无 |
| 返回值 | 无 |

| | |
|---|---|
| **先决条件** | 设置好 DMA 模块的工作方式。 |
| **被调用函数** | 无 |

**例：**设置SPI0模块发送完成后自动触发DMA功能继续传输，（需DMA模块相关函数配合使用）。使用实例如下：

```
SPI_DMACmd(SPI0, SPI_DMAReq_TCF, ENABLE);
```

# 10. 模数转换器(ADC)

模拟到数字量转换模块称为 ADC，K60 的 ADC 模块包含 2 个模块，分别为 ADC0、ADC1，采用 16 位精度的线性逐次逼近算法，具有 4 对差分模拟输入和 24 个单端模拟输入引脚，而且具有 64 倍增益的可编程增益放大器。

## 10.1 ADC 模块主要寄存器结构

**ADC**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| SC1A | ADC 模块状态和控制寄存器 1A |
| SC1B | ADC 模块状态和控制寄存器 1B |
| CFG1 | ADC 模块配置寄存器 1 |
| CFG2 | ADC 模块配置寄存器 2 |
| RA | 数据结果寄存器 A |
| RB | 数据结果寄存器 B |
| SC2 | ADC 模块状态和控制寄存器 2 |
| SC3 | ADC 模块状态和控制寄存器 3 |

## 10.2 ADC 库函数

ADC 库函数

| 函数名 | 描述 |
|---|---|
| **ADC_Init** | 配置 ADC 模块工作方式 |
| **ADC_GetConversionValue** | 获得模数转换结果 |
| **ADC_ITConfig** | ADC 模块中端配置 |
| **ADC_GetITStatus** | 获得 ADC 模块中断状态 |
| **ADC_DMACmd** | 使能 ADC 模块的 DMA 功能 |

### 10.2.1 ADC_Init

| 函数名 | **ADC_Init** |
|---|---|
| **函数原形** | ADC_Init(ADC_InitTypeDef* ADC_InitStruct) |
| **功能描述** | 配置 ADC 模块工作方式 |

| | |
|---|---|
| **输入参数** | ADC_InitStruct，存储有关 ADC 模块的工作设置 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**ADC_InitTypeDef** 位于**adc.h**中，用于设置ADC模块的工作状态，具体参数如下。

**ADCxMap**

该参数选择AD模块工作引脚和类型，如下表：

| **ADCxMap** | **描述** |
|---|---|
| **ADC0_DP0_DM0** | 差分模式下，ADC0 模块下的 DP0、DM0 引脚 |
| **ADC0_SE0_DP0** | 单端模式下，ADC0 模块下 0 通道的 DP0 引脚 |
| **ADC0_SE1_DP1** | 单端模式下，ADC0 模块下 1 通道的 DP1 引脚 |
| **ADC1_SE0_DP0** | 单端模式下，ADC1 模块下 0 通道的 DP0 引脚 |

（说明：因引脚和通道太多，在这里不再一一列出，具体参见spi.h文件）

**ADC_Precision**

该参数选择AD模块转换精度，如下表：

| **ADC_Precision** | **描述** |
|---|---|
| **ADC_PRECISION_8BIT** | 设置 AD 转换精度为 8 位 |
| **ADC_PRECISION_10BIT** | 设置 AD 转换精度为 10 位 |
| **ADC_PRECISION_12BIT** | 设置 AD 转换精度为 12 位 |
| **ADC_PRECISION_16BIT** | 设置 AD 转换精度为 16 位 |

**ADC_TriggerSelect**

该参数选择AD模块转换触发方式，如下表：

| **ADC_TriggerSelect** | **描述** |
|---|---|
| **ADC_TRIGGER_HW** | 设置 AD 转换为硬件触发 |
| **ADC_TRIGGER_SW** | 设置 AD 转换为软件触发 |

**例：**使用ADC0模块的单端模式进行模数转换，转换精度为16位，引脚为DP0，使用0通道，软件触发，具体使用情况如下：

```
ADC_InitTypeDef  ADC_InitStruct；//申请结构体变量
ADC_InitStruct . ADCxMap  =  ADC0_SE0_DP0;//使用引脚配置
ADC_InitStruct . ADC_Precision = ADC_PRECISION_16BIT;//16 位转换精度
ADC_InitStruct . ADC_TriggerSelect = ADC_TRIGGER_SW；//选择软件触发
ADC_Init（ADC_InitStruct）;//调用初始化函数
```

## 10.2.2 ADC_GetConversionValue

| | |
|---|---|
| **函数名** | **ADC_GetConversionValue** |
| **函数原形** | ADC_GetConversionValue(uint32_t ADCxMap) |

| | |
|---|---|
| **功能描述** | 获得模数转换结果 |
| **输入参数** | ADCxMap：AD 模块引脚复用 |
| **输出参数** | 无 |
| **返回值** | 模数转换结果 |
| **先决条件** | ADC 模块已经完成初始化配置 |
| **被调用函数** | 无 |

**例：**获得上述模块的模数转换结果，存储在value中。

```
value = ADC_GetConversionValue(ADC0_SE0_DP0);
```

## 10.2.3 ADC_ITConfig

| **函数名** | **ADC_ITConfig** |
|---|---|
| **函数原形** | ADC_ITConfig(ADC_Type* ADCx,uint8_t ADC_Mux, uint16_t ADC_IT, FunctionalState NewState) |
| **功能描述** | ADC 模块中断配置 |
| **输入参数 1** | ADCx: ADC0、ADC1 |
| **输入参数 2** | ADC_Mux: A/B 通道选择 |
| **输入参数 3** | ADC_IT：中断源选择 |
| **输入参数 4** | NewState：ENABLE（开启），DISABLE（关闭） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**配置ADC0模块的A通道在模数转换完成时产生中断信号。

```
ADC_ITConfig(ADC0, A, ADC_IT_AI, ENABLE);
```

## 10.2.4 ADC_GetITStatus

| **函数名** | **ADC_GetITStatus** |
|---|---|
| **函数原形** | ADC_GetITStatus(ADC_Type* ADCx, uint8_t ADC_Mux, uint16_t ADC_IT) |
| **功能描述** | 获得 AD 模块中断标志状态 |
| **输入参数 1** | ADCx: ADC0、ADC1 |
| **输入参数 2** | ADC_Mux: A/B 通道选择 |
| **输入参数 3** | ADC_IT：中断类型 |
| **输出参数** | 无 |
| **返回值** | 中断的标志状态 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**获得ADC0模块的A通道的转换完成中断标志状态，status存储标志状态。

status = ADC_GetITStatus(ADC0, A, ADC_IT_AI);

### 10.2.5 ADC_DMACmd

| 函数名 | **ADC_DMACmd** |
|---|---|
| 函数原形 | ADC_DMACmd(ADC_Type* ADCx, uint16_t ADC_DMAReq, FunctionalState NewState) |
| 功能描述 | 使能 ADC 模块的 DMA 功能 |
| 输入参数 1 | ADCx: ADC0、ADC1 |
| 输入参数 2 | ADC_DMAReq: 触发 DMA 中断源 |
| 输入参数 3 | NewState：ENABLE（开启），DISABLE（关闭） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 设置好 DMA 模块的工作方式 |
| 被调用函数 | 无 |

**例**：设置ADC0模块发送完成后自动触发DMA功能继续传输，（需DMA模块相关函数配合使用）。使用实例如下：

ADC_DMACmd(ADC0, ADC_DMAReq_COCO, ENABLE);

## 11. 数模转换器(DAC)

DAC 是将数字编码量转换为模拟信号的设备，是 ADC 的逆转换。K60 的 DAC 模块共有 16 位的数据缓冲区，同时 DAC 可以被配置为正常开模式、摆动模式、一次扫描模式。该模块支持可配置阈值的 16 字数据缓冲器和多操作模式。

### 11.1 DAC 模块主要寄存器结构

**DAC**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| DATnL | DAC 模块数字电压低八位 |
| DATnH | DAC 模块数字电压高八位 |
| SR | DAC 模块状态寄存器 |
| C0 | DAC 模块控制寄存器 0 |
| C1 | DAC 模块控制寄存器 1 |
| C2 | DAC 模块控制寄存器 2 |

### 11.2 DAC 库函数

DAC 库函数

| 函数名 | 描述 |
|---|---|

| | |
|---|---|
| **DAC_Init** | DAC 模块初始化配置 |
| **DAC_StructInit** | DAC 模块默认配置 |
| **DAC_DMACmd** | DAC 模块 DMA 功能设置 |
| **DAC_ITConfig** | DAC 模块中断配置 |
| **DAC_GetITStatus** | 获得 DAC 模块中断状态 |
| **DAC_SoftwareTrigger** | 软件触发 DAC 模块 |
| **DAC_SetBuffer** | 设置 DAC 模块数模转换数据缓存 |
| **DAC_SetValue** | 设置 DAC 模块电压输出值 |

## 11.2.1 DAC_Init

| 函数名 | **DAC_Init** |
|---|---|
| 函数原形 | DAC_Init(DAC_InitTypeDef* DAC_InitStruct) |
| 功能描述 | DAC 模块初始化配置 |
| 输入参数 | DAC_InitStruct　DAC 模块工作配置 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**DAC_InitTypeDef** 位于**dac.h**中，用于设置DAC模块的工作状态，具体参数如下。

**DAC_TrigerMode**

该参数选择DA模块的工作触发方式，如下表：

| **DAC_TrigerMode** | **描述** |
|---|---|
| **DAC_TRIGER_MODE_NONE** | 不触发转换 |
| **DAC_TRIGER_MODE_SOFTWARE** | 软件触发转换 |
| **DAC_TRIGER_MODE_HARDWARE** | 硬件触发转化 |

**DAC_BufferMode**

该参数选择DA模块的缓存方式，如下表：

| **DAC_ BufferMode** | **描述** |
|---|---|
| **BUFFER_MODE_DISABLE** | 关闭缓存模式 |
| **BUFFER_MODE_NORMAL** | 正常缓存模式 |
| **BUFFER_MODE_SWING** | SWING 缓存模式 |
| **BUFFER_MODE_ONETIMESCAN** | 一次浏览模式 |

**DAC_WaterMarkMode**

该参数选择DA模块的水平缓存方式，如下表：

| **DAC_WaterMarkMode** | **描述** |
|---|---|
| **WATER_MODE_1WORD** | 设置水平缓存为 1 字节 |
| **WATER_MODE_2WORD** | 设置水平缓存为 2 字节 |
| **WATER_MODE_3WORD** | 设置水平缓存为 3 字节 |
| **WATER_MODE_4WORD** | 设置水平缓存为 4 字节 |

**例：**具体使用见使用实例。

## 11.2.2 DAC_StructInit

| 函数名 | **DAC_StructInit** |
|---|---|
| 函数原形 | DAC_StructInit(DAC_InitTypeDef* DAC_InitStruct) |
| 功能描述 | DAC 模块默认配置 |
| 输入参数 | DAC_InitStruct （DAC 模块工作配置） |
| 输出参数 | DAC_InitStruct（DAC 模块默认配置输出） |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**使用默认配置启动DAC模块，配置它的工作方式。使用步骤如下：

```
DAC_InitTypeDef   * DAC_InitStruct ;//申请结构体
DAC_StructInit(DAC_InitStruct);//设置默认配置
DAC_Init(DAC_InitStruct);//调用初始化配置
DAC_SetValue(DAC0,1000);//通过调用设置函数设置输出电压值
DAC_SoftwareTrigger(DAC0);//软件触发一次输出
```

## 11.2.3 DAC_DMACmd

| 函数名 | **DAC_DMACmd** |
|---|---|
| 函数原形 | DAC_DMACmd(DAC_Type* DACx, uint16_t DAC_DMAReq, FunctionalState NewState) |
| 功能描述 | DAC 模块 DMA 功能设置 |
| 输入参数 1 | DACx： DAC0 |
| 输入参数 2 | DAC_DMAReq：DAC_DMAReq_DAC |
| 输入参数 3 | NewState：ENABLE(开启)，DISABLE(关闭) |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 设置好 DMA 模块的工作方式 |
| 被调用函数 | 无 |

**例：**设置DAC0模块缓冲区完成后自动触发DMA功能，（需DMA模块相关函数配合使用）。

使用实例如下：

```
DAC_DMACmd (DAC0, DAC_DMAReq_DAC, ENABLE);
```

## 11.2.4 DAC_ITConfig

| 函数名 | **DAC_ITConfig** |
|---|---|

| | |
|---|---|
| 函数原形 | DAC_ITConfig(DAC_Type* DACx, uint16_t DAC_IT, FunctionalState NewState) |
| 功能描述 | DAC 模块中断配置 |
| 输入参数 1 | DACx： DAC0 |
| 输入参数 2 | DAC_IT：DAC_IT_POINTER_BUTTOM(指针到达底部触发)<br>DAC_IT_POINTER_TOP(指针到达顶部触发)<br>DAC_IT_WATER_MARK(指针到达水平触发) |
| 输入参数 3 | NewState：ENABLE(开启)，DISABLE(关闭) |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：配置DAC模块在指针到达底部时触发中断。

```
DAC_ITConfig(DAC0, DAC_IT_POINTER_BUTTOM, ENABLE);
```

## 11.2.5 DAC_GetITStatus

| | |
|---|---|
| 函数名 | **DAC_GetITStatus** |
| 函数原形 | DAC_GetITStatus(DAC_Type* DACx, uint16_t DAC_IT) |
| 功能描述 | 获得 DAC 模块中断状态 |
| 输入参数 1 | DACx： DAC0 |
| 输入参数 2 | DAC_IT：DAC_IT_POINTER_BUTTOM(指针到达底部触发)<br>DAC_IT_POINTER_TOP(指针到达顶部触发)<br>DAC_IT_WATER_MARK(指针到达水平触发) |
| 输出参数 | 无 |
| 返回值 | 0：未发生、1：发生 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：获得指针到达底部的中断标志状态，存储在status中。

```
status = DAC_GetITStatus(DAC0, DAC_IT_POINTER_BUTTOM);
```

## 11.2.6 DAC_SoftwareTrigger

| | |
|---|---|
| 函数名 | **DAC_SoftwareTrigger** |
| 函数原形 | DAC_SoftwareTrigger(DAC_Type *DACx) |
| 功能描述 | 软件触发 DAC 模块 |
| 输入参数 | DACx： DAC0 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |

| 被调用函数 | 无 |
|---|---|

例：触发DAC模块进行一次数据转换。

```
DAC_SoftwareTrigger(DAC0);
```

### 11.2.7 DAC_SetBuffer

| 函数名 | **DAC_SetBuffer** |
|---|---|
| 函数原形 | DAC_SetBuffer(DAC_Type *DACx, uint16_t* DACBuffer,uint8_t NumberOfBuffer) |
| 功能描述 | 设置 DAC 模块数模转换数据缓存 |
| 输入参数 1 | DACx： DAC0 |
| 输入参数 2 | DACBuffer：数模输出缓存值 |
| 输入参数 3 | NumberOfBuffer：缓存数值顺序（0~15） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：设置缓存值为100，序号为10

```
DAC_SetBuffer(DAC0, 100,10);
```

### 11.2.8 DAC_SetValue

| 函数名 | **DAC_SetValue** |
|---|---|
| 函数原形 | DAC_SetValue(DAC_Type *DACx,uint16_t DAC_Value) |
| 功能描述 | 设置 DAC 模块电压输出值 |
| 输入参数 1 | DACx： DAC0 |
| 输入参数 2 | DAC_Value：设置输出电压值（0~4095） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：设置DAC模块的数模输出值为1000，使用情况如下：

```
DAC_SetValue(DAC0,1000);
```

# 12. 看门狗模块(WDOG)

看门狗模块主要用于防止程序跑飞为设定的，对于保障程序的正常运行具有重要的作用，当程序未按照规定运行时，看门狗模块不能及时喂狗，引起芯片复位信号，重新启动芯片工作。

## 12.1 WDOG 模块主要寄存器结构

**WDOG**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| STCTRLH | WDOG 模块状态和控制寄存器（高位） |
| STCTRLL | WDOG 模块状态和控制寄存器（低位） |
| TOVALH | WDOG 模块超时寄存器（高位） |
| TOVALL | WDOG 模块超时寄存器（低位） |
| REFRESH | WDOG 模块刷新寄存器 |
| UNLOCK | WDOG 模块解锁寄存器 |
| PRESC | WDOG 模块分频寄存器 |

## 12.2 WDOG 库函数

WDOG 库函数

| 函数名 | 描述 |
|---|---|
| **WDOG_Init** | WDOG 模块初始化配置 |
| **WDOG_Open** | 开启看门狗功能 |
| **WDOG_Close** | 关闭看门狗功能 |
| **WDOG_Feed** | 喂看门狗 |

### 12.2.1 WDOG_Init

| 函数名 | **WDOG_Init** |
|---|---|
| 函数原形 | WDOG_Init(uint16_t FeedInterval) |
| 功能描述 | WDOG 模块初始化配置 |
| 输入参数 | FeedInterval 看门狗喂狗间隔 ms |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**设置看门狗喂狗间隔时间为100ms，具体使用如下：

```
WDOG_Init(100);
```

### 12.2.2 WDOG_Open

| 函数名 | **WDOG_Open** |
|---|---|
| 函数原形 | WDOG_Open(void) |
| 功能描述 | 开启看门狗功能 |

| | |
|---|---|
| **输入参数** | 无 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 已经配置好了看门狗的超时时间 |
| **被调用函数** | 无 |

**例：**开启看门狗功能，具体使用如下：

```
WDOG_Open();
```

### 12.2.3 WDOG_Close

| **函数名** | **WDOG_Close** |
|---|---|
| **函数原形** | WDOG_Close(void) |
| **功能描述** | 关闭看门狗功能 |
| **输入参数** | 无 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**关闭看门狗功能，具体使用如下：

```
WDOG_Close();
```

### 12.2.4 WDOG_Feed

| **函数名** | **WDOG_Feed** |
|---|---|
| **函数原形** | WDOG_Feed(void) |
| **功能描述** | 喂狗 |
| **输入参数** | 无 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**喂狗，防止芯片复位，具体使用如下：

```
WDOG_Feed();
```

# 13. 灵活定时器(FTM)

K60 的 FTM 模块具有两个 8 通道的定时器，支持输入捕捉、输出比较功能，并且可以生成 PWM 信号用于控制电机或电源管理等场合。FTM 的时间参考是一个 16 位的计数器，可以设置为无符号型或有符号型。目前本版本的库函数使用 FTM 模块主要用于产生 PWM 波形。

# 13.1FTM 模块主要寄存器结构

**FTM** 主要寄存器表

| 寄存器 | 描述 |
|---|---|
| SC | FTM 模块状态和控制寄存器 |
| CNT | FTM 模块计数器寄存器 |
| CnSC | FTM 模块 n 通道状态控制寄存器 |
| CnV | FTM 模块 n 通道值寄存器 |
| CNTIN | FTM 模块初始计数器 |

# 13.2 FTM 库函数

FTM 库函数

| 函数名 | 描述 |
|---|---|
| **FTM_Init** | FTM 模块初始化配置（用于产生 PWM 波形） |
| **FTM_PWM_ChangeDuty** | FTM 模块波形占空比设置 |

## 13.2.1 FTM_Init

| 函数名 | **FTM_Init** |
|---|---|
| 函数原形 | FTM_Init(FTM_InitTypeDef *FTM_InitStruct) |
| 功能描述 | FTM 模块初始化配置（用于产生 PWM 波形） |
| 输入参数 | FTM_InitStruct：FTM 模块配置结构体 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**FTM_InitTypeDef** 位于**ftm.h**中，用于设置FTM模块的工作状态，具体参数如下。

**FTMxMAP**

该参数选择FTM模块工作引脚和类型，如下表：

| FTMxMAP | 描述 |
|---|---|
| **FTM0_CH0_PC1** | FTM0 模块的 0 通道，PORTC 端口的 1 引脚 |
| **FTM0_CH5_PD5** | FTM0 模块的 5 通道，PORTD 口的 5 引脚 |
| **FTM1_CH0_PA12** | FTM1 模块的 0 通道，PORTA 端口的 12 引脚 |
| **FTM2_CH1_PB19** | FTM2 模块的 1 通道，PORTB 端口的 19 引脚 |

（说明：因引脚和通道太多，在这里不再一一列出，具体参见ftm.h文件）

**FTM_Mode_TypeDef**

该参数选择FTM模块输出的PWM波形，如下表：

| FTM_Mode_TypeDef | 描述 |
|---|---|
| **FTM_Mode_EdgeAligned** | FTM 模块 PWM 波形为边沿对齐 |

| | |
|---|---|
| **FTM_Mode_CenterAligned** | FTM 模块 PWM 波形为中央对齐 |
| **FTM_Mode_Combine** | FTM 模块 PWM 波形为组和模式 |
| **FTM_Mode_Complementary** | FTM 模块 PWM 波形为互补模式 |

**例：**配置芯片的PORTC端口的1引脚为PWM输出，占空比为40%，频率为1KHz，使用边沿对齐模式，具体使用如下：

```
FTM_InitTypeDef    FTM_InitStruct1;//申请结构体变量
FTM_InitStruct1.Frequency = 1000;                    // 1KHZ
FTM_InitStruct1.FTMxMAP = FTM0_CH0_PC1;              //FTM0_CH0 PC1引脚
FTM_InitStruct1.FTM_Mode = FTM_Mode_EdgeAligned; //边沿对齐模式
FTM_InitStruct1.InitalDuty = 4000;                   //占空比为40%
FTM_Init(&FTM_InitStruct1);                  //带入函数初始化
```

### 13.2.2 FTM_PWM_ChangeDuty

| 函数名 | **FTM_PWM_ChangeDuty** |
|---|---|
| 函数原形 | void FTM_PWM_ChangeDuty(uint32_t FTMxMAP,uint32_t PWMDuty) |
| 功能描述 | FTM 模块波形占空比设置 |
| 输入参数 | FTMxMAP：FTM 模块引脚<br>PWMDuty：FTM 波形占空比，精度为 0.1 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**设置PTM模块的波形输出占空比为50%，具体使用如下：

```
FTM_PWM_ChangeDuty(FTM0_CH0_PC1,5000);
```

## 14. 直接内存存取控制器(DMA)

DMA 是一种无需 CPU 干预就可以实现数据快速传输的技术。K60 的 DMA 由通道复用管理模块和 DMA 控制模块两部分组成，主要功能包括源和目标地址计算、数据移动操作、每一通道都包含传输控制描述符。(目前此模块还不完善，仅能保障少量的数据传输稳定。)

### 14.1 DMA 模块主要寄存器结构

**DMA及DMAMUX**主要寄存器表

| 寄存器 | 描述 |
|---|---|

| | |
|---|---|
| CHCFGn（DMAMUX 模块） | 通道配置寄存器（n=0-15） |
| CR | 控制寄存器 |
| ES | 错误状态寄存器 |
| ERQ | 使能请求寄存器 |
| INT | 中断请求寄存器 |

## 14.2 DMA 库函数

DMA 库函数

| 函数名 | 描述 |
|---|---|
| **DMA_Init** | DMA 模块初始化配置 |
| **DMA_SetEnableReq** | 设置 DMA 模块请求 |
| **DMA_IsComplete** | 检测 DMA 是否传输完成 |
| **DMA_SetCurrDataCounter** | 设置数据传输量 |
| **DMA_GetCurrDataCounter** | 获得剩余数据传输量 |
| **DMA_ClearITPendingBit** | 清除中断标志位 |
| **DMA_ITConfig** | DMA 中断配置 |

### 14.2.1 DMA_Init

| 函数名 | **DMA_Init** |
|---|---|
| 函数原形 | DMA_Init(DMA_InitTypeDef *DMA_InittStruct) |
| 功能描述 | DMA 模块初始化配置 |
| 输入参数 | DMA_InittStruct：初始化配置结构体 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

（说明：参数结构体的变量使用，详见dma.h文件，不在这里一一叙述。）

**例：**使用DMA模块的0通道传输256个字节，每一次传输一字节，具体使用如下：

```
DMA_InitTypeDef DMA_InitStruct1;     //DMA结构变量申请
DMA_InitStruct1.Channelx = DMA_CH0;          //使用0通道
DMA_InitStruct1.DMAAutoClose = ENABLE;     //传输完毕后自动关闭
DMA_InitStruct1.EnableState = ENABLE;        //初始化完成后立即开始传输
DMA_InitStruct1.MinorLoopLength = 256;       //一个传输256个字节
DMA_InitStruct1.PeripheralDMAReq    = DMA_MUX2;       //不需触发源
DMA_InitStruct1.TransferBytes = 1;         //每次传输一个字节
```

```
//配置目的地址传输参数
DMA_InitStruct1.DestBaseAddr = (uint32_t)DMADestBuffer;  //指向目的地址
DMA_InitStruct1.DestDataSize = DMA_DST_8BIT;    //数据接收1字节
DMA_InitStruct1.DestMajorInc = 0;          //只执行一次大循环
DMA_InitStruct1.DestMinorInc = 1;          //每次传输完地址加1
//配置源地址传输参数
DMA_InitStruct1.SourceBaseAddr = (uint32_t)DMASrcBuffer;//设置源地址
DMA_InitStruct1.SourceDataSize = DMA_SRC_8BIT; //数据接收1字节
DMA_InitStruct1.SourceMajorInc = 0;  //只执行一次大循环
DMA_InitStruct1.SourceMinorInc = 1;  //每次传输完地址加1
DMA_Init(&DMA_InitStruct1);    //带入函数初始化配置。
```

## 14.2.2 DMA_SetEnableReq

| 函数名 | **DMA_SetEnableReq** |
|---|---|
| 函数原形 | DMA_SetEnableReq(uint8_t DMAChl,FunctionalState EnableState) |
| 功能描述 | 设置 DMA 模块请求 |
| 输入参数 1 | DMAChl：DMA0_CH0 - DMA_CH15 |
| 输入参数 2 | EnableState：ENABLE （开启传输）；DISABLE:（关闭传输） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**开启0通道进行数据传输，具体使用如下：

```
DMA_SetEnableReq(DMA0_CH0, ENABLE);
```

## 14.2.3 DMA_IsComplete

| 函数名 | **DMA_IsComplete** |
|---|---|
| 函数原形 | DMA_IsComplete(uint8_t DMAChl) |
| 功能描述 | 检测 DMA 是否传输完成 |
| 输入参数 | DMAChl：DMA0_CH0 - DMA_CH15 |
| 输出参数 | 无 |
| 返回值 | 1- TRUE,传输完成；0- FALSE,传输未完成 |

| | |
|---|---|
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：获取0通道的数据传输状态，返回值存储在status中，具体使用如下：

```
Status = DMA_IsComplete(DMA0_CH0);
```

## 14.2.4 DMA_SetCurrDataCounter

| 函数名 | DMA_SetCurrDataCounter |
|---|---|
| 函数原形 | DMA_SetCurrDataCounter(uint8_t DMAChl,uint16_t DataNumber) |
| 功能描述 | 设置数据传输量 |
| 输入参数 1 | DMAChl：DMA0_CH0 - DMA_CH15 |
| 输入参数 2 | DataNumber：数据传输量 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：设置0通道的数据传输量为256字节，具体使用如下：

```
DMA_SetCurrDataCounter(DMA0_CH0,256);
```

## 14.2.6 DMA_GetCurrDataCounter

| 函数名 | DMA_GetCurrDataCounter |
|---|---|
| 函数原形 | DMA_GetCurrDataCounter(uint8_t DMAChl) |
| 功能描述 | 获取剩余数据传输量 |
| 输入参数 | DMAChl：DMA0_CH0 - DMA_CH15 |
| 输出参数 | 无 |
| 返回值 | 有 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：获得通道0 数据传输的剩余字节量，存储在value中，具体使用如下：

```
Value = DMA_GetCurrDataCounter(DMA0_CH0);
```

## 14.2.7 DMA_ClearITPendingBit

| 函数名 | DMA_ClearITPendingBit |
|---|---|
| 函数原形 | DMA_ClearITPendingBit(DMA_Type* DMAx, uint16_t DMA_IT, uint8_t DMA_CH) |

| | |
|---|---|
| **功能描述** | 清除中断标志位 |
| **输入参数 1** | DMAx：DMA0 |
| **输入参数 2** | DMA_IT：DMA_IT_MAJOR-数据完成产生中断 |
| **输入参数 3** | DMA_CH：DMA0_CH0 - DMA_CH15 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**清除通道0在数据传输完成后产生中断信号，具体使用如下：

```
DMA_ClearITPendingBit(DMA0, DMA_IT_MAJOR, DMA0_CH0);
```

### 14.2.8 DMA_ITConfig

| **函数名** | **DMA_ITConfig** |
|---|---|
| **函数原形** | DMA_ITConfig(DMA_Type* DMAx, uint16_t DMA_IT, uint8_t DMA_CH, FunctionalState NewState) |
| **功能描述** | DMA 中断配置 |
| **输入参数 1** | DMAx：DMA0 |
| **输入参数 2** | DMA_IT：DMA_IT_MAJOR-数据完成产生中断 |
| **输入参数 3** | DMA_CH：DMA0_CH0 - DMA_CH15 |
| **输入参数 4** | NewState：ENABLE （开启）；DISABLE:（关闭） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**配置0通道数据完成时产生中断信号，具体使用如下：

```
DMA_ITConfig(DMA0, DMA_IT_MAJOR, DMA0_CH0, ENABLE);
```

**注：**如果开启中断功能的话，需要使用中断处理函数配合使用，具体函数 void DMA0_IRQHandler(void)存放在 isr.c 文件中，使用情况详见 isr.c 文件。

# 15. 系统设置 (SYS)

此单元不属于任何模块，主要是用于设置芯片工作的各种频率，例如 BUS_Clock、CORE_Clock、FLASH_Clock 等，通过该文件的一些函数可以获取当前芯片的一些工作参数和状态等信息。

## 15.1 主要寄存器结构

主要寄存器表

| 寄存器所在模块 | 描述 |
|---|---|

| | |
|---|---|
| MC | 模式控制器 |
| SIM | 系统综合模块 |
| MCG | 多功能时钟发生器 |
| NVIC | 嵌套式矢量中断控制器 |

## 15.2 SYS 函数

SYS 库函数

| 函数名 | 描述 |
|---|---|
| **SystemClockSetup** | 设置芯片工作的频率 |
| **SystemSoftReset** | 系统软件复位 |
| **GetCPUInfo** | 获取芯片的信息 |
| **EnableInterrupts** | 开启芯片内核中断 |
| **DisableInterrupts** | 关闭芯片内核中断 |
| **SetVectorTable** | 设置中断向量表起始地址 |
| **NVIC_Init** | 设置一个中断的优先级 |
| **GetFWVersion** | 获取本函数库的版本号 |

### 15.2.1 SystemClockSetup

| 函数名 | **SystemClockSetup** |
|---|---|
| **函数原形** | SystemClockSetup(uint8_t ClockOption,uint16_t CoreClock) |
| **功能描述** | 设置芯片工作的频率 |
| **输入参数 1** | ClockOption：选择时钟源<br>ClockSource_IRC（内部时钟源）<br>ClockSource_EX8M（外部 8MHz）<br>ClockSource_EX50M（外部 50MHz） |
| **输入参数 2** | CoreClock：期望的内核频率<br>CoreClock_48M（48MHz）<br>CoreClock_64M（64MHz）<br>CoreClock_72M（72MHz）<br>CoreClock_96M（96MHz）<br>CoreClock_100M（100MHz） |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置芯片选用外部50MHz时钟，内核频率为100MHz，具体使用如下：

SystemClockSetup(ClockSource_EX50M,CoreClock_100M);

## 15.2.2 SystemSoftReset

| 函数名 | **SystemSoftReset** |
|---|---|
| 函数原形 | SystemSoftReset(void) |
| 功能描述 | 系统软件复位 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**软件复位芯片一次，具体使用如下：

```
SystemSoftReset( );
```

## 15.2.3 GetCPUInfo

| 函数名 | **GetCPUInfo** |
|---|---|
| 函数原形 | GetCPUInfo(void) |
| 功能描述 | 获取当前芯片的信息 |
| 输入参数 | 无 |
| 输出参数 | 有，所获取的芯片信息都存储在 CPUInfo 的结构体中 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获取芯片的BUS_Clock信息，具体使用如下：

```
GetCPUInfo( );
CPUInfo. BusClock;//调用此数据即可获取 BUS 时钟频率
```

## 15.2.4 EnableInterrupts

| 函数名 | **EnableInterrupts** |
|---|---|
| 函数原形 | EnableInterrupts(void) |
| 功能描述 | 开启芯片内核中断 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**开启M4内核的中断处理功能，具体使用如下：

```
EnableInterrupts( );
```

## 15.2.5 DisableInterrupts

| 函数名 | **DisableInterrupts** |
|---|---|
| 函数原形 | DisableInterrupts(void) |
| 功能描述 | 关闭芯片内核中断 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**关闭M4内核的中断处理功能，具体使用如下：

```
DisableInterrupts ( );
```

## 15.2.6 SetVectorTable

| 函数名 | **SetVectorTable** |
|---|---|
| 函数原形 | SetVectorTable (uint32_t offset) |
| 功能描述 | 设置中断向量表的起始地址 |
| 输入参数 | Offset：中断向量表的起始地址 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**设置中断向量表的起始地址为0X20处，具体使用如下：

```
SetVectorTable(0x20);
```

**说明：**此函数请慎用，否则会造成中断失败。

## 15.2.9 NVIC_Init

| 函数名 | **NVIC_Init** |
|---|---|
| 函数原形 | NVIC_Init(IRQn_Type IRQn,uint32_t PriorityGroup,uint32_t PreemptPriority,uint32_t SubPriority) |
| 功能描述 | 设置一个中断源的优先级 |
| 输入参数 | IRQn：中断源 |
| 输入参数 | PriorityGroup：中断优先级分组 |
| 输入参数 | NVIC_PriorityGroup_0~ NVIC_PriorityGroup_4 |
| 输入参数 | PreemptPriority：抢占优先级<br>SubPriority：响应优先级 |
| 输出参数 | 无 |

| | |
|---|---|
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**设置PORTD端口的中断源的中断优先级分组为0，强占优先级为1，响应优先级为2，

具体使用如下：

```
NVIC_Init(PORTD_IRQn, NVIC_PriorityGroup_0,1,2);
```

### 15.2.10 GetFWVersion

| **函数名** | **GetFWVersion** |
|---|---|
| **函数原形** | uint16_t GetFWVersion(void) |
| **功能描述** | 获取本函数库的版本 |
| **输入参数** | 无 |
| **输出参数** | 无 |
| **返回值** | 有，库函数版本号 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**获取本函数库的版本号，存储在version中，具体使用如下：

```
Version = GetFWVersion( );//返回值为十进制数。
```

# 16. 延时模块(DELAY)

Cortex_M4 内核的处理器内部包含了一个 SysTick 定时器，可以实现系统的延时功能，利用内部 SysTick 来实现延时时，既不占用中断，也不占用系统定时器，可以提供短时间的延时功能，最小延时单位可以 1 微秒。

## 16.1 SysTick 模块主要寄存器结构

**SysTick**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| CTRL | SysTick 模块状态和控制寄存器 |
| LOAD | SysTick 模块定时寄存器 |
| VAL | SysTick 模块计时寄存器 |

## 16.2 DELAY 函数

DELAY 库函数

| 函数名 | 描述 |
|---|---|

| | |
|---|---|
| **DelayInit** | SysTick 模块初始化配置 |
| **DelayUs** | 微秒级延时函数 |
| **DelayMs** | 毫秒级延时函数 |

## 16.2.1 DelayInit

| 函数名 | **DelayInit** |
|---|---|
| 函数原形 | DelayInit(void) |
| 功能描述 | SysTick 模块初始化配置 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**详见使用实例。

## 16.2.2 DelayUs

| 函数名 | **DelayUs** |
|---|---|
| 函数原形 | DelayUs(uint32_t us) |
| 功能描述 | 微秒级延时函数 |
| 输入参数 | us：微秒单位的延时时间 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**详见使用实例。

## 16.2.3 DelayMs

| 函数名 | **DelayMs** |
|---|---|
| 函数原形 | DelayMs(uint32_t ms) |
| 功能描述 | 毫秒级延时函数 |
| 输入参数 | ms：毫秒单位的延时时间 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**详见使用实例。

## 16.3 使用实例

先进行 100us 的延时，随后进入 100ms 的延时，使用情况如下：

1. DelayInit();//延时模块初始化配置
2. DelayUs(100);//100us 延时
3. DelayMs(100);//100ms 延时

# 17. 低功耗计时器(LPTM)

低功耗计时器 LPTM 可以被配置成定时器，也可配置成脉冲计数器，实现脉冲计数功能，KinetisK 系列处理器中包含一个 LPTM 模块，一次仅可进行一路脉冲计数。

## 17.1 LPTM 模块主要寄存器结构

**LPTM**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| CSR | 模块状态和控制寄存器 |
| PSR | 分频寄存器 |
| CMR | 比较值寄存器 |
| CNR | 计数寄存器 |

## 17.2 LPTM 函数

LPTM 库函数

| 函数名 | 描述 |
|---|---|
| **LPTM_Init** | LPTM 模块初始化寄存器 |
| **LPTM_SetCompareValue** | 设置定时器比较值 |
| **LPTM_GetCompareValue** | 获得定时器比较值 |
| **LPTM_GetTimerCounterValue** | 获得计数器的值 |
| **LPTM_ITConfig** | 模块中断配置 |
| **LPTM_GetITStatus** | 获得模块的中断标志状态 |
| **LPTM_ClearITPendingBit** | 清除模块中断标志状态 |
| **LPTM_ResetTimeCounter** | 模块计数器重置 |

### 17.2.1 LPTM_Init

| 函数名 | **LPTM_Init** |
|---|---|
| 函数原形 | LPTM_Init(LPTM_InitTypeDef* LPTM_InitStruct) |
| 功能描述 | LPTM 模块初始化寄存器 |
| 输入参数 | LPTM_InitStruct：模块工作配置结构体 |

| | |
|---|---|
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**LPTM _InitTypeDef** 位于**lptm.h**中，用于设置LPTM模块的工作状态，具体参数如下。

**LPTMxMap**

该参数选择LPTM模块的引脚，如下表：

| **LPTMxMap** | **描述** |
|---|---|
| **LPTM_CH1_PA19** | 使用 1 通道，PORTA 端口的 19 引脚 |
| **LPTM_CH2_PC5** | 使用 2 通道，PORTC 端口的 5 引脚 |

**LPTM_Mode**

该参数选择LPTM模块工作模式，如下表：

| **LPTM_Mode** | **描述** |
|---|---|
| **LPTM_Mode_PC_RISING** | 脉冲计数模式，上升沿计数 |
| **LPTM_Mode_PC_FALLING** | 脉冲计数模式，下降沿计数 |
| **LPTM_Mode_TC** | 定时计数模式 |

**例：**配置LPTM模块工作在外部引脚计数模式，具体使用如下：

```
LPTM_InitTypeDef LPTM_InitStruct1; //申请结构体变量
LPTM_InitStruct1.LPTMxMap = LPTM_CH2_PC5; //使用 2 通道的 PORTC 端口的 5 引脚
LPTM_InitStruct1.LPTM_InitCompareValue = 200;      //在脉冲计数模式下无意义
LPTM_InitStruct1.LPTM_Mode = LPTM_Mode_PC_FALLING;   //下降沿触发脉冲计数
LPTM_Init(&LPTM_InitStruct1); //带入函数，启动工作。
```

## 17.2.2 LPTM_SetCompareValue

| **函数名** | **LPTM_SetCompareValue** |
|---|---|
| **函数原形** | LPTM_SetCompareValue(LPTMR_Type* LPTMx, uint32_t Value) |
| **功能描述** | 设置定时器比较值，在定时模式下用于设定触发时间 |
| **输入参数 1** | LPTMx：LPTMR0 |
| **输入参数 2** | Value：0~65535 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 已经设置好模块处于定时工作模块 |
| **被调用函数** | 无 |

**例：**设置定时时间为200毫秒，具体使用如下：

```
LPTM_SetCompareValue(LPTMR0, 200);
```

## 17.2.3 LPTM_GetCompareValue

| 函数名 | **LPTM_GetCompareValue** |
|---|---|
| 函数原形 | LPTM_GetCompareValue(LPTMR_Type* LPTMx) |
| 功能描述 | 获得定时器比较值 |
| 输入参数 | LPTMx：LPTMR0 |
| 输出参数 | 无 |
| 返回值 | 返回定时器的值 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：获得定时器比较值，存储在value中，具体使用如下：

```
Value = LPTM_GetCompareValue(LPTMR0);
```

## 17.2.4 LPTM_GetTimerCounterValue

| 函数名 | **LPTM_GetTimerCounterValue** |
|---|---|
| 函数原形 | LPTM_GetTimerCounterValue(LPTMR_Type* LPTMx) |
| 功能描述 | 获得计数器的值 |
| 输入参数 | LPTMx：LPTMR0 |
| 输出参数 | 无 |
| 返回值 | 返回当前计数器的值 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：获得计数器的值，存储在value中，具体使用如下：

```
Value = LPTM_ GetTimerCounterValue (LPTMR0);
```

## 17.2.5 LPTM_ITConfig

| 函数名 | **LPTM_ITConfig** |
|---|---|
| 函数原形 | LPTM_ITConfig(LPTMR_Type* LPTMx, uint16_t LPTM_IT, FunctionalState NewState) |
| 功能描述 | 模块中断配置 |
| 输入参数 1 | LPTMx：LPTMR0 |
| 输入参数 2 | LPTM_IT：LPTM_IT_TCF，定时器溢出中断 |
| 输入参数 3 | NewState：ENABLE （开启）；DISABLE:（关闭） |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：启动定时器溢出中断，具体使用如下：

LPTM_ITConfig(LPTMR0,LPTM_IT_TCF,ENABLE);

**注：**需要配合中断处理函数 **void LPTimer_IRQHandler(void)**使用，具体参见 **isr.c** 文件

## 17.2.6 LPTM_GetITStatus

| 函数名 | **LPTM_GetITStatus** |
|---|---|
| 函数原形 | LPTM_GetITStatus(LPTMR_Type* LPTMx, uint16_t LPTM_IT) |
| 功能描述 | 获得模块的中断标志状态 |
| 输入参数 1 | LPTMx：LPTMR0 |
| 输入参数 2 | LPTM_IT：LPTM_IT_TCF，定时器溢出中断 |
| 输出参数 | 无 |
| 返回值 | 指定中断标志的状态 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**获得模块的定时器溢出中断标志状态，存储在statue中，具体使用如下：

Statue = LPTM_GetITStatus(LPTMR0, LPTM_IT_TCF);

## 17.2.7 LPTM_ClearITPendingBit

| 函数名 | **LPTM_ClearITPendingBit** |
|---|---|
| 函数原形 | LPTM_ClearITPendingBit(LPTMR_Type *LPTMx, uint16_t LPTM_IT) |
| 功能描述 | 清除模块中断标志状态 |
| 输入参数 1 | LPTMx：LPTMR0 |
| 输入参数 2 | LPTM_IT：LPTM_IT_TCF，定时器溢出中断 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**例：**清除模块定时器溢出中断标志状态，具体使用如下：

LPTM_ClearITPendingBit(LPTMR0, LPTM_IT_TCF);

## 17.2.9 LPTM_ResetTimeCounter

| 函数名 | **LPTM_ResetTimeCounter** |
|---|---|
| 函数原形 | LPTM_ResetTimeCounter(LPTMR_Type* LPTMx) |
| 功能描述 | 模块计数器重置 |
| 输入参数 | LPTMx：LPTMR0 |
| 输出参数 | 无 |

| | |
|---|---|
| **返回值** | 无 |
| **先决条件** | 一般用于脉冲计数模式下，获取计数值后调用此函数。 |
| **被调用函数** | 无 |

**例：**清空计数器，具体使用如下：

```
LPTM_ResetTimeCounter(LPTMR0);
```

# 18. 可编程延时模块(PDB)

可编程延时模块（PDB）可以提供从内部/外部触发源、可编程间隔到 A/D 转换的硬件触发，也可以提供 D/A 转换模块间隔触发的可控时延，可以为 ADC 转换和 DAC 输出提供精确的时间。

## 18.1 PDB 模块主要寄存器结构

**PDB**主要寄存器表

| 寄存器 | 描述 |
|---|---|
| SC | 模块状态和控制寄存器 |
| MOD | 最大值寄存器 |
| CNT | 计数寄存器 |
| IDLY | 中断延时寄存器 0 |
| CHnC1 | 通道 n 控制寄存器 1（n=0-15） |
| CHnS | 通道 n 状态寄存器（n=0-15） |
| CHnDLY0 | 通道 n 延时寄存器 0（n=0-15） |
| CHnDLY1 | 通道 n 延时寄存器 1（n=0-15） |

## 18.2 PDB 函数

PDB 库函数

| 函数名 | 描述 |
|---|---|
| **PDB_Init** | PDB 初始化 |
| **PDB_ADC_TriggerInit** | PDB 触发 ADC 功能初始化 |
| **PDB_ITConfig** | PDB 中断配置 |
| **PDB_GetITStatus** | PDB 获得中断标志位 |
| **PDB_DMACmd** | PDB DMA 功能控制 |
| **PDB_ClearITPendingBit** | PDB 清除中断等待位 |

### 18.2.1 PDB_Init

| 函数名 | **PDB_Init** |
|---|---|
| 函数原形 | PDB_Init(PDB_InitTypeDef * PDB_InitStruct) |

| | |
|---|---|
| **功能描述** | PDB 模块初始化 |
| **输入参数** | PDB_InitStruct：PDB 模块初始化结构 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**PDB_InitTypeDef** 位于**pdb.h**中，用于初始化PDB模块。

**PDB_TriggerSourceSelect**

PDB 触发源

| **PDB_TriggerSourceSelect 可选值** | **描述** |
|---|---|
| **TRIGGER_IN0** | 使用 0 通道触发 |
| **TRIGGER_INn** | 使用 n 通道触发 |
| **TRIGGER_IN15** | 使用 15 通道触发 |
| **TRIGGER_IN_SOFTWARE_TRIGER** | 使用软件触发 |

**PDB_ContinuousMode**

连续或者一次触发模式

| **PDB_ContinuousMode可选值** | **描述** |
|---|---|
| **PDB_CONT_MODE_ONESHOT** | 只执行一次 PDB 延时 |
| **PDB_CONT_MODE_CONTINUE** | 连续性的执行，类似 PIT 中断 |

**PDB_LoadMode**

PDB 装载模式选择

| **PDB_LoadMode可选值** | **描述** |
|---|---|
| **LDMOD0** | 当 LDOK=1 之后 立即加载 |
| **LDMOD1** | 当 LDOK=1 和 PDB 到达 MOD 后，立即加载 |
| **LDMOD2** | 当 LDOK=1 和一个输入时间设置为 1,立即加载 |
| **LDMOD3** | 当 LDOK=1 和倒计时为 0 时,立即加载 |

**PDB_Period**

PDB 定时周期 单位 MS

**例：**配置PDB模块为周期性中断计时模式，定时周期为 100毫秒，使用软件进行触发，具体使用情况如下：

```
PDB_InitTypeDef PDB_InitStruct1;
PDB_InitStruct1.PDB_ContinuousMode = PDB_CONT_MODE_CONTINUE;
PDB_InitStruct1.PDB_LoadMode = LDMOD0;
PDB_InitStruct1.PDB_Period = 100;
PDB_InitStruct1.PDB_TriggerSourceSelect = TRIGGER_IN_SOFTWARE_TRIGER;
PDB_Init(&PDB_InitStruct1);
```

## 18.2.2 PDB_ADC_TriggerInit

| 函数名 | **PDB_ADC_TriggerInit** |
|---|---|
| 函数原形 | PDB_ADC_TriggerInit(PDB_ADC_PreTriggerInitTypeDef * PDB_ADC_InitStruct) |
| 功能描述 | PDB 触发 ADC 初始化 |
| 输入参数 | PDB_ADC_InitStruct：初始化结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | PDB 模块已经被初始化 |
| 被调用函数 | 无 |

例：配置 PDB 模块周期性的延时触发 ADC 模块进行模数转换，具体使用如下：

```
PDB_ADC_TriggerInit(&PDB_ADC_InitStruct1);
```

## 18.2.3 PDB_ITConfig

| 函数名 | **PDB_ITConfig** |
|---|---|
| 函数原形 | PDB_ITConfig(PDB_Type* PDBx, uint16_t PDB_IT, FunctionalState NewState) |
| 功能描述 | PDB 中断配置 |
| 输入参数 1 | PDBx： |
| -可选值 | PDB0 PDB0 模块 |
| 输入参数 2 | PDB_IT： |
| -可选值 | PDB_ERR ： PDB 错误中断 |
| -可选值 | PDB_IT_IF PDB 计数器溢出中断 |
| 输入参数 3 | NewState： |
| -可选值 | ENABLE: 使能 |
| -可选值 | DISABLE: 禁止 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | PDB 已经被初始化 |
| 被调用函数 | 无 |

例：配置 PDB 模块计数到达中断，具体使用如下：

```
PDB_ITConfig(PDB0, PDB_IT_IF, ENABLE);
```

## 18.2.4 PDB_GetITStatus

| 函数名 | **PDB_GetITStatus** |
|---|---|
| 函数原形 | PDB_GetITStatus(PDB_Type* PDBx, uint16_t PDB_IT) |
| 功能描述 | 获得 PDB 中断标志位状态 |
| 输入参数 1 | PDBx： |
| -可选值 | PDB0 : PDB0 模块 |

| | |
|---|---|
| **输入参数 2** | PDB_IT： |
| **-可选值** | PDB_ERR ： PDB 错误中断 |
| **-可选值** | PDB_IT_IF PDB 计数器溢出中断 |
| **输出参数** | 无 |
| **返回值** | ITStatus |
| **-可选值** | RESET:标志位没有被职位 |
| **-可选值** | SET： 标志位已经被置位 |
| **先决条件** | PDB 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**获取中断标志状态，具体使用如下：

```
PDB_GetITStatus(PDB0, PDB_IT_IF);
```

## 18.2.5 PDB_DMACmd

| | |
|---|---|
| **函数名** | **PDB_DMACmd** |
| **函数原形** | PDB_DMACmd(PDB_Type* PDBx, uint16_t PDB_DMAReq, FunctionalState NewState) |
| **功能描述** | PDB DMA 控制 |
| **输入参数 1** | PDBx： |
| **-可选值** | PDB0 模块 |
| **输入参数 2** | PDB_DMAReq |
| **-可选值** | PDB_DMAReq_IF PDM DMA 中断 |
| **输入参数 3** | NewState： |
| **-可选值** | ENABLE : 使能 |
| **-可选值** | DISABLE：禁止 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | PDB 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**配置PDB模块使用中断触发DMA功能，需要DMA模块函数配合使用，具体使用如下：

```
PDB_DMACmd(PDB0, PDB_DMAReq_IF, ENABLE);
```

## 18.2.6 PDB_ClearITPendingBit

| | |
|---|---|
| **函数名** | **PDB_ClearITPendingBit** |
| **函数原形** | PDB_ClearITPendingBit(PDB_Type *PDBx,uint16_t PDB_IT) |
| **功能描述** | PDB 清除中断标志位 |
| **输入参数 1** | PDBx： |
| **-可选值** | PDB0 模块 |

| | |
|---|---|
| **输入参数 2** | PDB_IT： |
| **-可选值** | PDB_ERR ： PDB 错误中断 |
| **-可选值** | PDB_IT_IF PDB 计数器溢出中断 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | PDB 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**清除PDB模块中的计数器溢出中断标志，具体使用如下：

```
PDB_ClearITPendingBit(PDB0，PDB_IT_IF PDB);
```

# 19. 局域网控制器 (CAN)

CAN 总线是一种半双工的通信方法，通常况下是用两条总线 CANH 和 CANL 以及电平的差分方法进行数据信号的表达。CAN 属于总线型结构，采用同步、串行、多主、双向通信数据块的通信方式，不分主从，网络上每一个节点都可以主动发送信息，可以很方便地构成多机备份。

## 19.1 CAN 模块主要寄存器结构

| 寄存器 | 描述 |
|---|---|
| MCR | CAN 模块配置寄存器 |
| CTRL1 | 控制寄存器 1 |
| CTRL2 | 控制寄存器 2 |

## 19.2 CAN 函数

CAN 库函数

| 函数名 | 描述 |
|---|---|
| **CAN_Init** | CAN 初始化 |
| **CAN_EnableReceiveMB** | CAN 时能接收邮箱 |
| **CAN_Receive** | CAN 接收数据 |
| **CAN_Transmit** | CAN 发送数据 |
| **CAN_ITConfig** | CAN 中断配置 |
| **CAN_GetITStatus** | CAN 获得中断标志位状态 |
| **CAN_ClearITPendingBit** | CAN 清楚中断等待位 |
| **CAN_ClearAllITPendingBit** | CAN 清楚所有中断等待位 |

### 19.2.1 CAN_Init

| 函数名 | **CAN_Init** |
|---|---|

| | |
|---|---|
| **函数原形** | CAN_Init(CAN_InitTypeDef* CAN_InitStruct) |
| **功能描述** | 初始化 CAN 模块 |
| **输入参数** | CAN_InitStruct：CAN 初始化结构体 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**CAN_InitTypeDef** 位于**can.h**中，用于设置CAN初始化

**CANxMap**

CAN 通道及引脚定义

| **CANxMap 可选值** | **描述** |
|---|---|
| **CAN0_TX_PA12_RX_PA13** | CAN0 Tx:PA12 Rx:PA13 |
| **CAN0_TX_PB18_RX_PB19** | CAN0 Tx:PB18 Rx:PB19 |
| **CAN1_TX_PE24_RX_PE25** | CAN1 Tx:PE24 Rx:PE25 |
| **CAN1_TX_PC17_RX_PC16** | CAN1 Tx:PC17 Rx:PC16 |

**CAN_BaudRateSelect**

波特率选择

| **CAN_BaudRateSelect 可选值** | **描述** |
|---|---|
| CAN_SPEED_33K | 波特率 33K |
| CAN_SPEED_83K | 波特率 83K |
| CAN_SPEED_50K | 波特率 50K |
| CAN_SPEED_100K | 波特率 100K |
| CAN_SPEED_125K | 波特率 125K |
| CAN_SPEED_250K | 波特率 250K |
| CAN_SPEED_500K | 波特率 500K |
| CAN_SPEED_1000K | 波特率 1000K |

**FilterEnable**

是否时能 ID 过滤功能

| **FilterEnable 可选值** | **描述** |
|---|---|
| ENABLE | 使能 |
| DISABLE | 禁止 |

**例：**配置 CAN 模块的通信速度为 125KHz，不使用滤波功能，使用 PORTA 端口的 12 引脚作为发送引脚，使用 PORTA 端口的 13 引脚作为接收引脚

```
CAN_InitTypeDef   CAN_InitStruct1;
CAN_InitStruct1. CAN_BaudRateSelect = CAN_SPEED_125K;
CAN_InitStruct1. CANxMap = CAN0_TX_PA12_RX_PA13;
CAN_InitStruct1. FilterEnable = DISABLE;
CAN_Init(&CAN_InitStruct1);
```

## 19.2.2 CAN_EnableReceiveMB

| 函数名 | **CAN_EnableReceiveMB** |
|---|---|
| 函数原形 | CAN_EnableReceiveMB(CAN_Type*CANx,CAN_RxMsgTypeDef*RxMessage) |
| 功能描述 | CAN 使能接收邮箱 |
| 输入参数 1 | CANx： |
| -可选值 | CAN0 CAN0 模块 |
| -可选值 | CAN1 CAN1 模块 |
| 输入参数 2 | RxMessage: CAN 接收消息邮箱结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | CAN 已经被初始化 |
| 被调用函数 | 无 |

例：CAN_EnableReceiveMB(CAN0,CAN_RxMessage1);

## 19.2.3 CAN_Receive

| 函数名 | **CAN_Receive** |
|---|---|
| 函数原形 | CAN_Receive(CAN_Type* CANx,CAN_RxMsgTypeDef* RxMessage) |
| 功能描述 | CAN 接收数据 |
| 输入参数 1 | CANx： |
| -可选值 | CAN0 CAN0 模块 |
| -可选值 | CAN1 CAN1 模块 |
| 输入参数 2 | RxMessage: CAN 接收消息邮箱结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | CAN 已经被初始化 |
| 被调用函数 | 无 |

例：CAN_ Receive (CAN0,CAN_RxMessage1);

## 19.2.4 CAN_Transmit

| 函数名 | **CAN_Transmit** |
|---|---|
| 函数原形 | CAN_Transmit(CAN_Type* CANx, CAN_TxMsgTypeDef* TxMessage) |
| 功能描述 | CAN 发送数据 |
| 输入参数 1 | CANx： |
| -可选值 | CAN0 CAN0 模块 |
| -可选值 | CAN1 CAN1 模块 |
| 输入参数 2 | TxMessage:CAN 通讯发送结构 |

| | |
|---|---|
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | CAN 已经被初始化 |
| 被调用函数 | 无 |

例：CAN_Transmit(CAN0,CAN_TxMessage1);

## 19.2.5 CAN_ITConfig

| 函数名 | **CAN_ITConfig** |
|---|---|
| 函数原形 | CAN_ITConfig(CAN_Type* CANx, uint16_t CAN_IT, FunctionalState NewState) |
| 功能描述 | CAN 中断配置 |
| 输入参数 1 | CANx： |
| -可选值 | CAN1 CAN1 模块 |
| -可选值 | CAN1 CAN1 模块 |
| 输入参数 2 | CAN_IT: |
| -可选值 | CAN_IT_MB0 ： MB0 接收中断 |
| -可选值 | CAN_IT_MBn ： MBn 接收中断 |
| -可选值 | CAN_IT_MB15 ： MB15 接收中断 |
| 输入参数 3 | NewState： |
| -可选值 | ENABLE 使能 |
| -可选值 | DISABLE 禁止 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | CAN 已经被初始化 |
| 被调用函数 | 无 |

例：CAN_ITConfig(CAN0,CAN_IT_MB0, ENABLE);

## 19.2.6 CAN_GetITStatus

| 函数名 | **CAN_GetITStatus** |
|---|---|
| 函数原形 | CAN_GetITStatus(CAN_Type* CANx, uint16_t CAN_IT) |
| 功能描述 | 获得 CAN 模块中断标志位状态 |
| 输入参数 1 | CANx： |
| -可选值 | CAN0 CAN0 模块 |
| -可选值 | CAN1 CAN1 模块 |
| 输入参数 2 | CAN_IT: |
| -可选值 | CAN_IT_MB0 ： MB0 接收中断 |
| -可选值 | CAN_IT_MBn ： MBn 接收中断 |
| -可选值 | CAN_IT_MB15 ： MB15 接收中断 |
| 输出参数 | 无 |
| 返回值 | ITStates |

| | |
|---|---|
| **-可选值** | SET　　相应标志位被置位 |
| **-可选值** | RESET　相应标志位没有被职位 |
| **先决条件** | CAN 已经被初始化 |
| **被调用函数** | 无 |

**例：**CAN_GetITStatus(CAN0, CAN_IT_MB0)

## 19.2.7 CAN_ClearITPendingBit

| **函数名** | **CAN_ClearITPendingBit** |
|---|---|
| **函数原形** | CAN_ClearITPendingBit(CAN_Type* CANx, uint16_t CAN_IT) |
| **功能描述** | CAN 清除中断标志位 |
| **输入参数 1** | CANx： |
| **-可选值** | CAN0 CAN0 模块 |
| **-可选值** | CAN1 CAN1 模块 |
| **输入参数 2** | CAN_IT: |
| **-可选值** | CAN_IT_MB0 ： MB0 接收中断 |
| **-可选值** | CAN_IT_MBn ： MBn 接收中断 |
| **-可选值** | CAN_IT_MB15 ： MB15 接收中断 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | CAN 已经被初始化 |
| **被调用函数** | 无 |

**例：**CAN_ClearITPendingBit(CAN0, CAN_IT_MB0);

## 19.2.8 CAN_ClearAllITPendingBit

| **函数名** | **CAN_ClearAllITPendingBit** |
|---|---|
| **函数原形** | CAN_ClearAllITPendingBit(CAN_Type* CANx) |
| **功能描述** | CAN：清除所有中断标志位 |
| **输入参数** | CANx： |
| **-可选值** | CAN0 CAN0 模块 |
| **-可选值** | CAN1 CAN1 模块 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | CAN 已经被初始化 |
| **被调用函数** | 无 |

**例：**CAN_ClearAllITPendingBit(CAN0);

# 20. FLASH 存储器(FLASH)

FLASH 存储器具有电可擦除、无须后备电源来保存数据、可在线编程、存储密度高、功耗

低和成本低的特点。KinetisK 系列微控制器内部 FLASH 存储器模块包含程序 FLASH 存储器和 Flex 存储器。K60 系列微控制器所有芯片都包含程序 FLASH，指定类型芯片包含 FlexNVM 和 FlexRAM。

# 20.1 FLASH 模块主要寄存器结构

| 寄存器 | 描述 |
|---|---|
| FSTST | Flash 状态寄存器 |
| PFB0CR | FlashBank0 控制寄存器 |
| PFB1CR | FlashBank1 控制寄存器 |

# 20.2 FLASH 函数

**FLASH** 库函数

| 函数名 | 描述 |
|---|---|
| **FLASH_Init** | 初始化 Flash 控制器模块 |
| **FLASH_ReadByte** | Flash 读取字节 |
| **FLASH_WriteSector** | Flash 写入一个扇区 |
| **FLASH_EraseSector** | Flash 删除一个扇区 |

## 20.2.1 FLASH_Init

| 函数名 | **FLASH_Init** |
|---|---|
| 函数原形 | FLASH_Init(void) |
| 功能描述 | 初始化 Flash 模块控制器 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

例：FLASH_Init();

## 20.2.2 FLASH_ReadByte

| 函数名 | **FLASH_ReadByte** |
|---|---|
| 函数原形 | FLASH_ReadByte(uint32_t FlashStartAdd,uint32_t len,uint8_t *pbuffer) |
| 功能描述 | 读取 Flash 字节 |
| 输入参数 1 | FlashStartAdd：读取字节起始地址 |
| 输入参数 2 | len：读取长度 |
| 输入参数 3 | pbuffer：读取的内容缓冲区指针 |

| | |
|---|---|
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | Flash 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**FLASH_ReadByte（0, 512, pBuffer）;

### 20.2.3 FLASH_WriteSector

| **函数名** | **FLASH_WriteSector** |
|---|---|
| **函数原形** | FLASH_WriteSector(uint32_t sectorNo,uint16_t count,uint8_t const *buffer) |
| **功能描述** | Flash 写入一个扇区 |
| **输入参数 1** | sectorNo：扇区号 |
| **输入参数 2** | count：　写入的字节数 |
| **输入参数 3** | buffer：　结束数据的缓冲区指针 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | Flash 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**FLASH_WriteSector(0,512,pBuffer);

### 20.2.4 FLASH_EraseSector

| **函数名** | **FLASH_EraseSector** |
|---|---|
| **函数原形** | FLASH_EraseSector(uint32_t sectorNo) |
| **功能描述** | 擦除一个扇区 |
| **输入参数** | sectorNo：扇区号 |
| **输出参数** | 无 |
| **返回值** | 无 |
| **先决条件** | Flash 模块已经被初始化 |
| **被调用函数** | 无 |

**例：**FLASH_EraseSector(0);

## 21. SDIO 模块(SD)

SD 卡是一种常用于便携性设备的非易失性内存卡，根据封装或传输速度的不同， SD 卡可以分为不同的等级。SD 存储卡定义了 SD 接口和 SPI 接口两种通信协议。对 SD 模块的操作主要有初始化，读取数据块和写入数据块等操作。

本构件利用 Kinetis 的 SDIO 总线模块，完成 SD 卡的底层驱动函数操作。

## 21.1 SD 模块主要寄存器结构

| 寄存器 | 描述 |
|---|---|
| BLKATTR | 块传输特性控制器寄存器 |
| SYSCTL | 系统控制寄存器 |
| CMDARG | 命令参数寄存器 |
| XFERTYP | 传输类型寄存器 |
| PROCTL | 传输协议控制寄存器 |

## 21.2 SD 函数

SD 库函数

| 函数名 | 描述 |
|---|---|
| **SD_Init** | SD 卡初始化 |
| **SD_GetCapacity** | 获得 SD 卡容量 |
| **SD_ReadSingleBlock** | 读取 SD 卡一个块 |
| **SD_WriteSingleBlock** | 写入 SD 卡一个块 |

## 21.2.1 SD_Init

| 函数名 | **SD_Init** |
|---|---|
| **函数原形** | SD_Init(SD_InitTypeDef* SD_InitStruct) |
| **功能描述** | 初始化 SD 卡 |
| **输入参数** | SD_InitStruct：SD 卡初始化结构 |
| **输出参数** | SD_InitStruct：SD 卡初始化结构 |
| **返回值** | ESDHC_OK ： 初始化成功<br>ESDHC_ERROR_INIT_FAILED ： 初始化失败 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**SD_InitTypeDef** 位于**sd.h**中，用于设置或读取SD卡信息。

**SD_BaudRate**

该参数用于设置SDIO通讯速率 根据SD卡类型速率上限不同，初始化时可以先用较低的波特率测试。

**SD_CardType**

卡类型

| 宏 | 描述 |
|---|---|
| SD_CARD_TYPE_NONE | 无卡 |
| SD_CARD_TYPE_UNKNOW | 未能识别的卡 |
| SD_CARD_TYPE_SD | 普通 SD 卡 |
| SD_CARD_TYPE_SDHC | SDHC 卡 |

| | |
|---|---|
| SD_CARD_TYPE_SDIO | SDIO 设备 |
| SD_CARD_TYPE_SDCOMBO | COMBO 卡 |
| SD_CARD_TYPE_SDHCCOMBO | SDHC COMBO 卡 |
| SD_CARD_TYPE_MMC | MMC 卡 |
| SD_CARD_CEATA | CEATA 卡 |

**SD_Size** ： 卡大小 单位 MB
**OCR** ： 卡操作状态寄存器值
**CID** ： 卡 ID 识别寄存器值
**CSD** ： 卡数据类型寄存器
**RCA** ： 卡相对地址寄存器
**CSR** ： 卡配置寄存器
**例：** 初始化SD卡并打印SD卡大小

```
SD_InitTypeDef   SD_InitStruct1;
while(SD_Init(&SD_InitStruct1) != ESDHC_OK);
UART_printf("SDSize:%dMB", SD_InitStruct1.SD_Size);
```

## 21.2.2 SD_GetCapacity

| 函数名 | **SD_GetCapacity** |
|---|---|
| 函数原形 | SD_GetCapacity(SD_InitTypeDef* SD_InitStruct) |
| 功能描述 | 获得卡容量 单位 MB |
| 输入参数 | SD_InitStruct：SD 卡初始化结构 |
| 输出参数 | 无 |
| 返回值 | 卡容量 单位 MB |
| 先决条件 | SD 卡已经被初始化 |
| 被调用函数 | 无 |

**例：** SD_GetCapacity(SD_InitStruct);

## 21.2.3 SD_ReadSingleBlock

| 函数名 | **SD_ReadSingleBlock** |
|---|---|
| 函数原形 | SD_ReadSingleBlock(uint32_t sector, uint8_t *buffer) |
| 功能描述 | 读取 SD 卡一个块 |
| 输入参数 1 | sector：块编号 |
| 输入参数 2 | buffer：读取的缓冲区指针 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | SD 卡已经被初始化 |
| 被调用函数 | 无 |

**例：** SD_ReadSingleBlock(0,Buffer);

### 21.2.4 SD_WriteSingleBlock

| 函数名 | SD_WriteSingleBlock |
|---|---|
| 函数原形 | SD_WriteSingleBlock(uint32_t sector, const uint8_t *buffer) |
| 功能描述 | 写入 SD 卡一个块 |
| 输入参数 1 | sector：块编号 |
| 输入参数 2 | buffer：写入的缓冲区指针 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | SD 卡已经被初始化 |
| 被调用函数 | 无 |

例：SD_WriteSingleBlock(0,Buffer);

# 22. 触摸感应输入(TSI)

K60 的 TSI 模块提供的是一种电容感应输入接口，包括了 16 个 TSI 引脚。当有感应物接近与 TSI 引脚相连的电极的电容值变大，造成充放电时间延长，从而识别出触控动作。TSI 模块有 4 个 32 位功能设定与状态寄存器及 2 组通道寄存器。

## 22.1 TSI 模块主要寄存器结构

| 寄存器 | 描述 |
|---|---|
| GENCS | 通用控制与状态寄存器 |
| SCANC | 扫描控制寄存器 |
| PEN | 引脚时能寄存器 |
| STATUS | 状态寄存器 |

## 22.2 TSI 函数

TSI 库函数

| 函数名 | 描述 |
|---|---|
| TSI_Init | TSI 模块初始化 |
| TSI_SelfCalibration | TSI 模块自校准 |
| TSI_GetCounter | 获得 TSI 模块某个通道的计数值 |
| TSI_ITConfig | TSI 模块中断配置 |
| TSI_ClearAllITPendingFlag | TSI 清除所有中断标志位 |
| TSI_GetChannelOutOfRangleFlag | TSI 获得某个通道 超出范围标志位 |
| TSI_ClearITPendingBit | TSI 清除中断标志位 |
| TSI_GetITStatus | TSI 获得中断标志 |

## 22.2.1 TSI_Init

| 函数名 | **TSI_Init** |
|---|---|
| 函数原形 | TSI_Init(TSI_InitTypeDef* TSI_InitStruct) |
| 功能描述 | 初始化 TSI 一个通道 |
| 输入参数 | TSI_InitStruct：TSI 初始化结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**TSI_InitTypeDef** 位于**tsi.h**中，用于设置TSI初始化配置信息。

**TSIxMAP**

**TSI** 通道定义

| **TSIxMap 可选值** | **描述** |
|---|---|
| **TSI0_CH1_PA0** | TSI0 1 通道 PA0 引脚 |
| **TSI0_CH2_PA1** | TSI0 2 通道 PA1 引脚 |
| **TSI0_CH3_PA2** | TSI0 3 通道 PA2 引脚 |
| **TSI0_CH4_PA3** | TSI0 4 通道 PA3 引脚 |
| **TSI0_CH5_PA4** | TSI0 5 通道 PA4 引脚 |
| **TSI0_CH0_PB0** | TSI0 0 通道 PB0 引脚 |
| **TSI0_CH6_PB1** | TSI0 6 通道 PB1 引脚 |
| **TSI0_CH7_PB2** | TSI0 7 通道 PB2 引脚 |
| **TSI0_CH8_PB3** | TSI0 8 通道 PB3 引脚 |
| **TSI0_CH9_PB16** | TSI0 9 通道 PB16 引脚 |
| **TSI0_CH10_PB17** | TSI0 10 通道 PB17 引脚 |
| **TSI0_CH11_PB18** | TSI0 11 通道 PB18 引脚 |
| **TSI0_CH12_PB19** | TSI0 12 通道 PB19 引脚 |
| **TSI0_CH13_PC0** | TSI0 13 通道 PC0 引脚 |
| **TSI0_CH14_PC1** | TSI0 14 通道 PC1 引脚 |
| **TSI0_CH15_PC2** | TSI0 15 通道 PC2 引脚 |

**TSI_ITMode**

**TSI** 中断模式选择

| **TSI_ITMode 可选值** | **描述** |
|---|---|
| TSI_IT_MODE_END_OF_SCAN | TSI 扫描完成中断 |
| TSI_IT_MODE_OUT_OF_RANGE | TSI 超出预设范围中断 |

**例：**

```
    TSI_InitTypeDef TSI_InitStruct1;
  //初始化 PA4 引脚 用作 TSI 通道
    TSI_InitStruct1.TSIxMAP = TSI0_CH5_PA4;
    TSI_InitStruct1.TSI_ITMode = TSI_IT_MODE_END_OF_SCAN;
    TSI_Init(&TSI_InitStruct1);
    //初始化 PB16 引脚 用作 TSI 通道
```

```
  TSI_InitStruct1.TSIxMAP = TSI0_CH9_PB16;
  TSI_Init(&TSI_InitStruct1);
//配置 TSI 中断为扫描结束中断
  TSI_ITConfig(TSI0,TSI_IT_EOSF,ENABLE);
//开启对应的 NVIC 中断开关
  NVIC_EnableIRQ(TSI0_IRQn);
```

（**说明**：TSI 模块使用一个引脚通道时无法进行触控动作的识别，至少打开 2 个引脚才可有效）

## 22.2.2 TSI_SelfCalibration

| 函数名 | **TSI_SelfCalibration** |
|---|---|
| 函数原形 | TSI_SelfCalibration(uint8_t TSI_Ch) |
| 功能描述 | 完成一个通道的自校准 |
| 输入参数 1 | TSI_Ch: TSI 通道 |
| -可选值 | TSI0_CH0 : 0 通道 |
| -可选值 | TSI0_CHn : n 通道 |
| -可选值 | TSI0_CH15: 15 通道 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 该 TSI 通道已经被初始化 |
| 被调用函数 | 无 |

**例**：TSI_SelfCalibration(TSI_CH0);

## 22.2.3 TSI_GetCounter

| 函数名 | **TSI_GetCounter** |
|---|---|
| 函数原形 | TSI_GetCounter(uint8_t TSI_Ch) |
| 功能描述 | 获得 TSI 某个通道的计数值 |
| 输入参数 | TSI_Ch: TSI 通道 |
| -可选值 | TSI0_CH0 : 0 通道 |
| -可选值 | TSI0_CHn : n 通道 |
| -可选值 | TSI0_CH15: 15 通道 |
| 输出参数 | 无 |
| 返回值 | 计数值 |
| 先决条件 | 该 TSI 通道已经被初始化 |
| 被调用函数 | 无 |

**例**：TSI_GetCounter(TSI_CH0);

## 22.2.4 TSI_ITConfig

| 函数名 | **TSI_ITConfig** |
|---|---|
| 函数原形 | TSI_ITConfig(TSI_Type *TSIx,uint16_t TSI_IT,FunctionalState NewState) |
| 功能描述 | 配置 TSI 的中断 |
| 输入参数 1 | TSIx：TSI 模块号 |
| -可选值 | TSI0： TSI0 模块 |
| 输入参数 2 | TSI_IT： TSI 模块中断源 |
| -可选值 | TSI_IT_EOSF：TSI 扫描完成中断 |
| -可选值 | TSI_IT_OUTRGF：TSI 超出范围中断 |
| -可选值 | TSI_IT_EXTERF：TSI 外部短路中断 |
| -可选值 | TSI_IT_OVRF: TSI OverRun 中断 |
| 输入参数 3 | NewState： 使能或者禁止 |
| -可选值 | ENABLE:使能 |
| -可选值 | DISABLE：禁止 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | TSI 已经被初始化 |
| 被调用函数 | 无 |

例：TSI_ITConfig(TSI0,TSI_IT_EOSF,ENABLE);

## 22.2.5 TSI_ClearAllITPendingFlag

| 函数名 | **TSI_ClearAllITPendingFlag** |
|---|---|
| 函数原形 | TSI_ClearAllITPendingFlag(TSI_Type *TSIx) |
| 功能描述 | 清除所有中断等待位 |
| 输入参数 1 | TSIx：TSI 模块号 |
| -可选值 | TSI0：TSI0 模块 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | TSI 已经被初始化 |
| 被调用函数 | 无 |

例：TSI_ClearAllITPendingFlag(TSI0);

## 22.2.6 TSI_GetChannelOutOfRangleFlag

| 函数名 | **TSI_GetChannelOutOfRangleFlag** |
|---|---|
| 函数原形 | TSI_GetChannelOutOfRangleFlag(TSI_Type *TSIx,uint8_t TSI_CH) |
| 功能描述 | 获得某个 TSI 通道的超出范围标志 |
| 输入参数 1 | TSIx：TSI 模块号 |

| | |
|---|---|
| -可选值 | TSI0：TSI0 模块 |
| 输入参数 2 | TSI_Ch: TSI 通道 |
| -可选值 | TSI0_CH0 : 0 通道 |
| -可选值 | TSI0_CHn : n 通道 |
| -可选值 | TSI0_CH15: 15 通道 |
| 先决条件 | TSI 已经被初始化 |
| 被调用函数 | 无 |

例：TSI_GetChannelOutOfRangleFlag(TSI_CH0);

## 22.2.7 TSI_ClearITPendingBit

| 函数名 | **TSI_ClearITPendingBit** |
|---|---|
| 函数原形 | TSI_ClearITPendingBit(TSI_Type* TSIx,uint16_t TSI_IT) |
| 功能描述 | 清除 TSI 中断标志等待位 |
| 输入参数 1 | TSIx：TSI 模块号 |
| -可选值 | TSI0：TSI0 模块 |
| 输入参数 2 | TSI_IT： TSI 模块中断源 |
| -可选值 | TSI_IT_EOSF：TSI 扫描完成中断 |
| -可选值 | TSI_IT_OUTRGF：TSI 超出范围中断 |
| -可选值 | TSI_IT_EXTERF：TSI 外部短路中断 |
| -可选值 | TSI_IT_OVRF: TSI OverRun 中断 |
| 先决条件 | TSI 已经被初始化 |
| 被调用函数 | 无 |

例：TSI_ClearITPendingBit(TSI0, TSI_IT_EOSF);

## 22.2.8 TSI_GetITStatus

| 函数名 | **TSI_GetITStatus** |
|---|---|
| 函数原形 | TSI_GetITStatus(TSI_Type* TSIx, uint16_t TSI_IT) |
| 功能描述 | 获得 TSI 模块中断状态 |
| 输入参数 1 | TSIx：TSI 模块号 |
| -可选值 | TSI0：TSI0 模块 |
| 输入参数 2 | TSI_IT： TSI 模块中断源 |
| -可选值 | TSI_IT_EOSF：TSI 扫描完成中断 |
| -可选值 | TSI_IT_OUTRGF：TSI 超出范围中断 |
| -可选值 | TSI_IT_EXTERF：TSI 外部短路中断 |
| -可选值 | TSI_IT_OVRF: TSI OverRun 中断 |
| 输出参数 | 无 |
| 返回值 | ITStatues : 中断标志符 |
| -可选值 | SET： 标志位被置位 |
| -可选值 | RESET：标志位没有被置位 |

| 被调用函数 | 无 |
| --- | --- |

例：TSI_GetITStatus(TSI0,TSI_IT_EOSF);

# 23. 以太网控制器(ENET)

以太网控制器用于提供底层的网络通信控制，K60 芯片是不能通过引脚直接控制物理层收发器，必须通过 RMII/MII 接口间接控制物理层收发器。

## 23.1 ENET 模块主要寄存器结构

| 寄存器 | 描述 |
| --- | --- |
| ECR | 以太网控制寄存器 |
| MSCR | MII 速度控制寄存器 |
| RCR | 接收控制寄存器 |
| TCR | 发送控制寄存器 |
| PALR | 物理地址低寄存器 |
| PAUR | 物理地址高寄存器 |

## 23.2 ENET 函数

ENET 库函数

| 函数名 | 描述 |
| --- | --- |
| **ENET_Init** | 以太网初始化 |
| **ENET_MacSendData** | 以太网 MAC 层发送数据 |
| **ENET_MacRecData** | 以太网 MAC 层接收数据 |
| **ENET_MiiLinkState** | 指示以太网是否连接 |

### 23.2.1 ENET_Init

| 函数名 | **ENET_Init** |
| --- | --- |
| 函数原形 | ENET_Init(ENET_InitTypeDef* ENET_InitStrut) |
| 功能描述 | 初始化以太网模块 |
| 输入参数 | ENET_InitStrut：以太网初始化结构 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

**ENET_InitTypeDef** 位于**enet.h**中，用于设置ENET初始化配置信息。

**pMacAddress**

MAC 地址字符串指针

**例：**

```
const uint8_t LocalMacAddress[6] = {0x02,0x02,0x02,0x02,0x02,0x02};
ENET_InitTypeDef     ENET_InitStruct1;
ENET_InitStruct1.pMacAddress = (uint8_t*)LocalMacAddress;
ENET_Init(&ENET_InitStruct1);
```

## 23.2.2 ENET_MacSendData

| 函数名 | **ENET_MacSendData** |
|---|---|
| 函数原形 | ENET_MacSendData(uint8_t *ch, uint16_t len) |
| 功能描述 | 以太网 MAC 层发送数据 |
| 输入参数 1 | ch：数据指针 |
| 输入参数 2 | len：数据长度 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 以太网已经被初始化 |
| 被调用函数 | 无 |

**例：**ENET_MacSendData(pData,100);

## 23.2.3 ENET_MacRecData

| 函数名 | **ENET_MacRecData** |
|---|---|
| 函数原形 | ENET_MacRecData(uint8_t *ch) |
| 功能描述 | 以太网 MAC 层接收数据 |
| 输入参数 | ch：接收数据缓冲区指针 |
| 输出参数 | 收到的数据长度 |
| 返回值 | 有 |
| 先决条件 | 以太网已经被初始化 |
| 被调用函数 | 无 |

**例：**len = ENET_MacRecData(pData);

## 23.2.7 ENET_MiiLinkState

| 函数名 | **ENET_MiiLinkState** |
|---|---|
| 函数原形 | ENET_MiiLinkState(void) |
| 功能描述 | 判断网线是否已经正常连接 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | TRUE OR FALSE |

| | |
|---|---|
| **-可选值** | TRUE：已经连接 |
| **-可选值** | FALSE： 未连接 |
| **先决条件** | 无 |
| **被调用函数** | 无 |

**例：**LinkStates = ENET_MiiLinkState();

# 24 修订记录

本文档主要参考 STM32 的文档而来，依据 KinetisK 系类芯片的底层各个模块的主要功能进行相关函数的编写，函数的编写及完善主要依据前期编写的底层驱动改进而来。目前版本是库函数的第一版，如有问题和改进之处请 qq 联

# 中国石油大学(华东)

## 飞思卡尔嵌入式实验室

超核电子：http://upcmcu.taobao.com/

2013年8月