NV32F100 KBI 键盘中断编程

第一章 所有库函数简介

库函数列表

void KBI_Init(KBI_Type *pKBI, KBI_ConfigType *pConfig)
KBI 初始化函数
void KBI_SetCallback(KBI_Type *pKBI, KBI_CallbackType pfnCallback)
KBI 回调函数
寄存器操作的内联函数，调用内联函数和直接操作寄存器效率一样高
void KBI_EnableInt(KBI_Type *pKBI)
打开 KBI 中断使能
void KBI_DisableInt(KBI_Type *pKBI)
关闭 KBI 中断使能
void KBI_ClrFlags(KBI_Type *pKBI)
清除 KBI 中断标志位

KBI 特性说明

*最多 16 个具有单个使能位的键盘中断引脚
*每个键盘中断引脚可编程为：
仅下降沿触发
仅上升沿触发
下降沿和低电平都触发
上升沿和高电平都触发

KBI 使用说明

打开 KBI 系统时钟，KBI 使能，KBI 中断使能，中断触发方式，配置好目标引脚就可以 KBI 正常工作了，并在中断函数里执行想要的操作。

1.1 KBI 模块初始化

函数名	KBI_Init
函数原形	KBI_Init(KBI_Type *pKBI, KBI_ConfigType *pConfig)
功能描述	以配置结构体 pConfig 来初始化 KBI
输入参数	配置结构体 pConfig，模块结构体 KBI_Type
输出参数	无

返回值	无
先决条件	无
函数使用实例	先设置配置结构体，KBI_Init(KBI0, &sKBIConfig);

字段	描述
7-4 保留	此字段为保留字段 此只读字段为保留字段且值始终为 0。
3 KBF	KBI 中断标志 表示检测到 KBI 中断请求。写操作对 KBF 无效。 0 未检测到 KBI 中断请求。 1 检测到 KBI 中断请求。
2 KBACK	KBI 应答 在 KBACK 中写入 1 是标志清零机制的一部分。
1 KBIE	KBI 中断使能 确定 KBI 中断是否已使能： 0 KBI 中断未使能。 1 KBI 中断已使能。
0 KBMOD	KBI 检测模式 KBMOD(与 ES[KBEDG]寄存器）一起控制 KBI 中断引脚的检测模式： 0 键盘仅检测边沿。 1 键盘检测边沿和电平。

```
/*****************************************************************************//*!
*
* @简介  打开 KBI 时钟  配置 KBI 对应中断引脚  配置沿触发方式
*
*
* @无返回
*
*****************************************************************************/

void KBI_Init(KBI_Type *pKBI, KBI_ConfigType *pConfig)
{
#if defined(CPU_NV32)
    uint16_t    i;
    uint8_t     sc = 0;
    uint8_t     u8Port;
    uint8_t     u8PinPos;
    uint16_t    u16PinMapping[KBI_MAX_NO][8] =
    {
        {
            0, 1, 2, 3, 8, 9, 10, 11        /* KBI0 对于的脚位，对应关系可从芯片规格书里的管脚说明表里查
询 */
```

```
        },
        {
            24, 25, 26, 27, 28, 29, 30, 31          /* KBI1 对于的脚位，对应关系可从芯片规格书里的管脚说明表里查询 */
        }
    };

    if(KBI0 == pKBI)
    {
        SIM->SCGC     |= SIM_SCGC_KBI0_MASK;                  /* 打开 KBI0 的系统时钟 */
        u8Port        =  0;
    }
    else if (KBI1 == pKBI)
    {
        SIM->SCGC     |= SIM_SCGC_KBI1_MASK;                  /* 打开 KBI1 的系统时钟 */
        u8Port        =  1;
    }

    /* 配置 KBI 的中断方式 */
    sc              = pConfig->sBits.bMode;
    pKBI->SC        = sc;

    /* configure KBI pin polarity and others */
    for (i = 0; i < KBI_MAX_PINS_PER_PORT; i++)
    {
        if(pConfig->sPin[i].bEn)
        {
            pKBI->PE      |= (1<<i);                          /* 打开对应 KBI 中断脚的使能 */
            pKBI->ES      = (pKBI->ES & ~(1<<i)) | (pConfig->sPin[i].bEdge << i); //配置中断方式
            u8PinPos = u16PinMapping[u8Port][i];
            ASSERT(!(u8PinPos & 0x80));
        #if defined(CPU_NV32)|| defined(CPU_NV32M3)
            FGPIOA->PIDR   &= ~(1<<u8PinPos);               /* 使能 GPIO */
            FGPIOA->PDDR   &= ~(1<<u8PinPos);               /* 配置引脚为输入 */
            PORT->PUEL   |= (1<<u8PinPos);                  /* 配置引脚内部上拉 */
        #elif defined(CPU_NV32M4)
            if (u8Port == 0)     /* KBI0 */
            {
            FGPIOA->PIDR   &= ~(1<<u8PinPos);               /* 使能 GPIO */
            FGPIOA->PDDR   &= ~(1<<u8PinPos);               /* 配置引脚为输入 */
            PORT->PUE0   |= (1<<u8PinPos);                  /*配置引脚内部上拉 */
            }
            else if (u8Port == 1)     /* KBI1 */
            {
```

```
            FGPIOB->PIDR    &= ~(1<<u8PinPos);                  /* 使能 GPIO */
            FGPIOB->PDDR    &= ~(1<<u8PinPos);                   /* 配置引脚为输入 */
            PORT->PUE1    |= (1<<u8PinPos);                    /*配置引脚内部上拉 */
          }
        #endif
        }
    }

    #if defined(CPU_NV32M4)
    /*重置 KBI_SP 寄存器*/
    sc = pConfig->sBits.bRstKbsp<<KBI_SC_RSTKBSP_SHIFT;
    pKBI->SC       |= sc;

    /*Real KBI_SP register enable*/
    sc = pConfig->sBits.bKbspEn<<KBI_SC_KBSPEN_SHIFT;
    pKBI->SC       |= sc;
    #endif

    /* 清除中断位 */
    pKBI->SC       = sc;

    /* 使能 KBI 中断 */
    if(pConfig->sBits.bIntEn)
    {
        pKBI->SC       |=    KBI_SC_KBIE_MASK;

        if(KBI0 == pKBI)
        {
            NVIC_EnableIRQ(KBI0_IRQn);
        }
        else
        {
            NVIC_EnableIRQ(KBI1_IRQn);
        }
    }
}
```

1.2 KBI 回调函数

```
/*****************************************************************************//*!
*
* @简介 可将该回调函数指向用户自定义的回执函数.
*
* @无返回
```

```
*****************************************************************************/
void KBI_SetCallback(KBI_Type *pKBI, KBI_CallbackType pfnCallback)
{
    if(KBI0 == pKBI)
    {
        KBI_Callback[0] = pfnCallback;
    }
    else
    {
        KBI_Callback[1] = pfnCallback;
    }
}
```

第二章 样例程序

2.1 使用 KBI 中断来实现按键控制 LED 灯的亮灭

```
/*****************************************************************************************
*
*
* @简介   此例程配置 PA0 和 PD0 引脚为 KBI 中断，均为上升沿触发模式，引脚每有一次上升沿，就进入
*         回执函数 LED 灯进行一次反转，注意 KBI 反应速度很慢，不适用高频信号，100kHZ
*         沿变化触发最佳。
*
*****************************************************************************************/

#include "common.h"
#include "ics.h"
#include "rtc.h"
#include "uart.h"
#include "kbi.h"
#include "sysinit.h"


*****************************************************************************/
int main (void);
void RTC_Task(void);
void KBI0_Task(void);
void KBI1_Task(void);
```

```
/*****************************************************************/
int main (void)
{
    uint8_t              u8Ch,i,j;
    ICS_ConfigType   sICSConfig;
    RTC_ConfigType   sRTCConfig;
    RTC_ConfigType   *pRTCConfig = &sRTCConfig;
    KBI_ConfigType   sKBIConfig;

    /* 系统初始化 */
    sysinit();
    printf("\nRunning the KBI_demo project.\n");
LED0_Init();
LED1_Init();
LED2_Init();

    /* 初始化 RTC 为 1Hz 中断频率 */
    pRTCConfig->u16ModuloValue = 9;
    pRTCConfig->bInterruptEn     = RTC_INTERRUPT_ENABLE;       /* 打开 RTC 中断使能 */
    pRTCConfig->bClockSource     = RTC_CLKSRC_1KHZ;            /*时钟源为 1khz*/
    pRTCConfig->bClockPresaler = RTC_CLK_PRESCALER_100;      /*分频系数 100*/

    RTC_SetCallback(RTC_Task);
    RTC_Init(pRTCConfig);

    printf("\nin FEE mode now,");
    UART_WaitTxComplete(TERM_PORT);

    /* 将时钟由 FEI 模式 转换为 FEE 模式 */
    sICSConfig.u32ClkFreq = 32;
    ICS_SwitchMode(FEE,FEI, &sICSConfig);

    printf("switch to FEI mode.\n");

    OSC_Enable();

    /* 关闭所有引脚的 KBI 中断 */
   for (i = 0; i < KBI_MAX_PINS_PER_PORT; i++)
   {
     sKBIConfig.sPin[i].bEn        = 0;
   }

    sKBIConfig.sBits.bMode     = KBI_MODE_EDGE_ONLY;   //配置为沿触发
    sKBIConfig.sPin[0].bEdge = KBI_RISING_EDGE_HIGH_LEVEL; //配置为上升沿触发
```

```
    sKBIConfig.sBits.bIntEn    = 1;     //打开中断使能
    sKBIConfig.sPin[0].bEn     = 1;     //打开对应引脚的 KBI 功能

    KBI_Init(KBI0, &sKBIConfig);
    KBI_SetCallback(KBI0, &KBI0_Task);//定义 KBI0 中断执行的函数为 KBI0_Task


    KBI_Init(KBI1, &sKBIConfig);
    KBI_SetCallback(KBI1, &KBI1_Task);

    while(1);
}

/*****************************************************************************//*!
*
* @简介  RTC 中断回调的对应函数
*
* @无返回
*****************************************************************************/

void RTC_Task(void)
{
    /* toggle LED1 */
    LED0_Toggle();
}

/*****************************************************************************//*!
*
* @简介  KBI0 回调的对应操作函数
*
* @无返回
*****************************************************************************/

void KBI0_Task(void)
{
    LED1_Toggle();
    printf("KBI0 routinue.\n");
}

/*****************************************************************************//*!
*
* @简介  KBI1 回调的对应操作函数.
*
* @无返回
```

```
*****************************************************************************/

void KBI1_Task(void)
{
    LED2_Toggle();
    printf("KBI1 routinue.\n");
}
/*****************************************************************************/
```