(163条消息) ThreadX驱动编写(基于ARM处理器)_arm7star的博客-CSDN博客_threadx的arm7port

blog.csdn.net/arm7star/article/details/115706496

1、参考文档及代码

参考《Azure RTOS ThreadX User Guide》"Chapter 5: Device Drivers for ThreadX"

ThreadX 6.1.2 Versatile/PB代码参考https://github.com/arm7star/ThreadX(未添加驱动)

2、驱动框架介绍

ThreadX驱动框架比较简单，与linux驱动比较类似，中断服务程序进行简单的中断处理(外设硬件中断处理、清除外设及中断控制器中断，类似linux中断上半部)，然后唤醒驱动input/output线程(通过put信号量的方式幻想input/output线程，因此每中断一次信号量加1，类似linux的中断下半部)，input/output线程读写外设。

驱动程序编写流程即为创建中断服务程序与input/output线程之间同步的信号量，编写input/output线程(等待硬件中断，等待信号量)，编写中断服务程序(释放信号量)。

3、ThreadX官网驱动示例

3.1、信号量创建

```
VOID tx_sdriver_initialize(VOID)
{
        /* Initialize the two counting semaphores used to control
                the simple driver I/O. */
        tx_semaphore_create(&tx_sdriver_input_semaphore,
                "simple driver input semaphore", 0);
        tx_semaphore_create(&tx_sdriver_output_semaphore,
                "simple driver output semaphore", 1);
        /* Setup interrupt vectors for input and output ISRs.
                The initial vector handling should call the ISRs
                defined in this file. */
        /* Configure serial device hardware for RX/TX interrupt
        generation, baud rate, stop bits, etc. */
}
```

3.2、input线程

```
UCHAR tx_sdriver_input(VOID)
{
        /* Determine if there is a character waiting. If not,
                suspend. */
        tx_semaphore_get(&tx_sdriver_input_semaphore,
                TX_WAIT_FOREVER;
        /* Return character from serial RX hardware register. */
        return(*serial_hardware_input_ptr);
}
```

3.3、中断服务程序

```
VOID tx_sdriver_input_ISR(VOID)
{
        /* See if an input character notification is pending. */
        if (!tx_sdriver_input_semaphore.tx_semaphore_count)
        {
                /* If not, notify thread of an input character. */
                tx_semaphore_put(&tx_sdriver_input_semaphore);
        }
}
```

output驱动与此类似。

4、中断代码修改

ThreadX官网代码的IRQ中断处理函数在tx_initialize_low_level.S文件中，官网代码仅处理了定时器中断，默认都走定时器处理函数，代码如下。

```
    .global __tx_irq_handler
    .global __tx_irq_processing_return
__tx_irq_handler:
@
@    /* Jump to context save to save system context.  */
    B       _tx_thread_context_save
__tx_irq_processing_return:
@
@    /* At this point execution is still in the IRQ mode.  The CPSR, point of
@       interrupt, and all C scratch registers are available for use.  In
@       addition, IRQ interrupts may be re-enabled - with certain restrictions -
@       if nested IRQ interrupts are desired.  Interrupts may be re-enabled over
@       small code sequences where lr is saved before enabling interrupts and
@       restored after interrupts are again disabled.  */
@
@    /* Interrupt nesting is allowed after calling _tx_thread_irq_nesting_start
@       from IRQ mode with interrupts disabled.  This routine switches to the
@       system mode and returns with IRQ interrupts enabled.
@
@       NOTE:  It is very important to ensure all IRQ interrupts are cleared
@       prior to enabling nested IRQ interrupts.  */
#ifdef TX_ENABLE_IRQ_NESTING
    BL      _tx_thread_irq_nesting_start
#endif
@
@    /* For debug purpose, execute the timer interrupt processing here.  In
@       a real system, some kind of status indication would have to be checked
@       before the timer interrupt handler could be called.  */
@
    BL     _tx_timer_interrupt                  @ Timer interrupt handler
@
```

```
@
@    /* If interrupt nesting was started earlier, the end of interrupt nesting
@       service must be called before returning to _tx_thread_context_restore.
@       This routine returns in processing in IRQ mode with interrupts disabled.  */
#ifdef TX_ENABLE_IRQ_NESTING
    BL      _tx_thread_irq_nesting_end
#endif
@
@    /* Jump to context restore to restore system context.  */
    B       _tx_thread_context_restore
```

所有中断都调用_tx_timer_interrupt，为了能够处理所有中断，需要将_tx_timer_interrupt替换为所有中断处理函数，例如irq_handle，在irq_handle中获取中断号，调用对应的中断处理函数，例如tx_sdriver_input_ISR、tx_timer_ISR；

注意_tx_timer_interrupt没有清除中断，需要增加代码清除中断。

5、s3c6410中断代码示例

5.1、__tx_irq_handler

(修改中断处理函数为handle_irq)

```
    .global __tx_irq_handler
    .global __tx_irq_processing_return
__tx_irq_handler:
@
@    /* Jump to context save to save system context.  */
    B       _tx_thread_context_save
__tx_irq_processing_return:
@
@    /* At this point execution is still in the IRQ mode.  The CPSR, point of
@       interrupt, and all C scratch registers are available for use.  In
@       addition, IRQ interrupts may be re-enabled - with certain restrictions -
@       if nested IRQ interrupts are desired.  Interrupts may be re-enabled over
@       small code sequences where lr is saved before enabling interrupts and
@       restored after interrupts are again disabled.  */
@
@    /* Interrupt nesting is allowed after calling _tx_thread_irq_nesting_start
@       from IRQ mode with interrupts disabled.  This routine switches to the
@       system mode and returns with IRQ interrupts enabled.
@
@       NOTE:  It is very important to ensure all IRQ interrupts are cleared
@       prior to enabling nested IRQ interrupts.  */
#ifdef TX_ENABLE_IRQ_NESTING
    BL      _tx_thread_irq_nesting_start
#endif
@
@    /* For debug purpose, execute the timer interrupt processing here.  In
@       a real system, some kind of status indication would have to be checked
@       before the timer interrupt handler could be called.  */
@
    BL     handle_irq @ /* BL     _tx_timer_interrupt                  @ Timer interrupt handler */
@
```

```
@
@    /* If interrupt nesting was started earlier, the end of interrupt nesting
@       service must be called before returning to _tx_thread_context_restore.
@       This routine returns in processing in IRQ mode with interrupts disabled.  */
#ifdef TX_ENABLE_IRQ_NESTING
    BL      _tx_thread_irq_nesting_end
#endif
@
@    /* Jump to context restore to restore system context.  */
    B       _tx_thread_context_restore
```

5.2、handle_irq

硬件相关中断处理代码。

```c
#include "s3c6410.h"
// 中断处理函数指针类型定义
typedef void (*irq_handler_ptr)(void);
// 中断处理函数表(数组索引即为硬件中断号)
static irq_handler_ptr irq_handler_table[64] = {
};
/*
 * 功能：注册中断处理函数
 * 输入：hw_irq，需要屏蔽的中断号；handler_ptr中断处理函数指针
 * 输出：无
 * 返回：void
 */
void request_irq(unsigned int hw_irq, irq_handler_ptr handler_ptr)
{
    irq_handler_table[hw_irq] = handler_ptr;
}
/*
 * 功能：取消注册的中断处理函数
 * 输入：hw_irq，需要取消注册的硬件中断号
 * 输出：无
 * 返回：void
 */
void free_irq(unsigned int hw_irq)
{
    irq_handler_table[hw_irq] = (void (*)(void))0;
}
/*
 * 功能：c语言中断处理函数入口(中断上下文保存及恢复由上上一级函数实现)
 * 输入：无
 * 输出：无
 * 返回：void
 */
void handle_irq(void)
{
    int hw_irq = ffs(VIC0IRQSTATUS) - 1;
    if ((hw_irq >= 0)&& (irq_handler_table[hw_irq] != 0))
    {
        irq_handler_table[hw_irq]();
    }
}
```

5.3、按键中断服务程序

```c
/*
 * key.c
 */
#include   "s3c6410.h"
#include   "tx_api.h"
extern TX_SEMAPHORE            semaphore_0;
void key_isr(void)
{
    static int i = 0;
    UINT    status;
    printf("key_isr %d, %d\r\n", i++, GPNDAT);
    status =  tx_semaphore_put(&semaphore_0);
    printf("tx_semaphore_put %d\r\n", status);
    while ((~GPNDAT) & 0x3f);
    EINT0PEND = 0x3f;
    VIC0ADDRESS = 0;
}
void key_init(void)
{
    GPNCON &= ~(0xfff);
    GPNCON |= 0xaaa;
    EINT0CON0 &= ~(0xfff);
    EINT0CON0 |= 0x444; // 上升沿触发中断
    request_irq(INT_EINT0, key_isr);
    EINT0MASK &= ~(0x3f);
    VIC0INTEnable(INT_EINT0);
}
```

5.4、input线程

(获取信号量)

```
void    thread_0_entry(ULONG thread_input)
{
UINT    status;
    /* This thread simply sits in while-forever-sleep loop.  */
    while(1)
    {
        printf("thread 0 obtained semaphore:   %d\r\n", thread_0_counter);
        /* Get the semaphore with suspension.  */
        status =  tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);
        /* Check status.  */
        if (status != TX_SUCCESS)
            break;
        /* Increment the thread counter.  */
        thread_0_counter++;
    }
}
```