嵌入式第二次作业　2024.6 向胤兴 计算机2102 2215012469

1、轮询代码及分析

1.1 PPT中给出的轮询代码

```
#include "2410addr.h"
#include "option.h"
#include "def.h"
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

//初始化，输入PCLK、波特率——放在复位程序中
void Uart_Init(int pclk,int baud)
{
    int i;
    rGPHCON|=0xa0; //GPH2,GPH3 as TXD0,RXD0
    rGPHUP = 0x0;    //GPH2,GPH3内部上拉
    if(pclk == 0)
    pclk = PCLK;
    rUFCON0 = 0x0; //禁止3个通道的FIFO控制寄存器
    rUFCON1 = 0x0;
    rUFCON2 = 0x0;
    rUMCON0 = 0x0;
    rUMCON1 = 0x0;
    rUMCON2 = 0x0;    //初始化3个通道的MODEM控制寄存器，禁止AFC
    //Line control register 0: Normal,No parity,1 stop,8 bits.
    rULCON0=0x3;
    // Control register 0
    rUCON0  = 0x5;
    //Baud rate divisior register 0
    rUBRDIV0=( (int)(pclk/16./baud+0.5) -1 );
}
//接收一字节数据
char Uart_Getch(void)
{
        //读错误状态寄存器UERSTATn
        if (rUERSTAT0 & 0x4)
            return -1;
        return RdURXH0;  //0x50000027
}
//接收一个字符串
void Uart_GetString(char *string)
{
    char *string2 = string;
    char c;
    while((c = Uart_Getch())!='\r')   //回车符
           *string++ = c;
```

```
}
//接收一个字符串，转换为数字
int Uart_GetIntNum(void)
{
    char str[30];
    char *string = str;
    int base     = 10;  //进制
    int minus    = 0;   //是否负数
    int result   = 0;    //转换的结果
    int lastIndex;       //字符串长度
    int i;
    Uart_GetString(string);
if(string[0]=='-')
    {
        minus = 1;
        string++;
    }

if(string[0]=='0' && (string[1]=='x' || string[1]=='X'))
    {
        base    = 16;
        string += 2;
    }

lastIndex = strlen(string) - 1;

if(lastIndex<0)
        return -1;
if(string[lastIndex]=='h' || string[lastIndex]=='H' )
    {
        base = 16;
        string[lastIndex] = 0;
        lastIndex--;
    }
if(base==10)
{
     result = atoi(string);
     result = minus ? (-1*result):result;
}
else {
     for(i=0;i<=lastIndex;i++) {
        if(isalpha(string[i]))
        {
            if(isupper(string[i]))
                result = (result<<4) + string[i] - 'A' + 10;
            else
                result = (result<<4) + string[i] - 'a' + 10;
        }
        else
            result = (result<<4) + string[i] - '0';
        }
        result = minus ? (-1*result):result;
     }
 return result;
}
//发送一字节
```

```
//#define WrUTXH0(ch)  ( * (volatile unsigned char * )0x50000023)=(unsigned char)(ch)
void Uart_SendByte(int data)
{
       WrUTXH0(data);
}
//发送一个字符串
void Uart_SendString(char *pt)
{
    while(*pt)
        Uart_SendByte(*pt++);
}
```

1.2 主要代码分析

1. `Uart_Init(int pclk, int baud)`

- **功能**：初始化UART通信模块。
- **初始化操作**：
  - 设置 `rGPHCON` 的特定位，将 GPH2 和 GPH3 配置为 UART0 的 TXD 和 RXD。
  - 禁用内部上拉 `rGPHUP`。
  - 设置波特率相关寄存器 `rUBRDIV0`，计算方法为 `(PCLK / 16 / baud) - 1`。
  - 其他相关寄存器的初始化，如 FIFO 控制寄存器和 MODEM 控制寄存器。

2. `void Uart_GetString(char *string)`

- **功能**：从UART接收一个字符串，直到遇到回车符 `\r`。
- **实现**：使用 `Uart_Getch()` 循环接收字符，直到接收到回车符为止，将字符存储到 `string` 中。

3. `void Uart_SendByte(int data)`

- **功能**：发送一个字节的数据到UART。

4. `void Uart_SendString(char *pt)`

- **功能**：发送一个以 null 结尾的字符串到UART。

2、改为中断模式

2.1 中断模式要求

PPT中给出的改为中断模式的要求

采用loopback模式，发送一个字符串，并接收
发送和接收都采用中断方式
编写完整程序，包括中断向量表、复位程序、IRQ中断服务程序等。全汇编或混合编程均可。
应编译通过。
设中断服务函数入口地址，把中断服务函数入口地址填入中断散转表
进入中断后：
（1）先屏蔽发送和接收中断，防止新来中断干扰我们的正常发送和接收
（2）查询寄存器INTOFFSET，根据散转表调用子程序
（3）在散转程序中查询挂起寄存器： SUBSRCPND
（4）清除几个挂起寄存器
（5）正常发送和接收
（6）取消中断屏蔽，等下一次中断。

2.2 关键代码实现

ResetHandler

```
ResetHandler
        BL Clock_Init    ; 初始化时钟
        LDR SP, =SvcStackSpace ; 设置管理模式堆栈
        MSR CPSR_c, #Mode_IRQ
        LDR SP, =IrqStackSpace ; 设置IRQ模式堆栈
        MSR CPSR_c, #Mode_FIQ
        LDR SP, =FiqStackSpace ; 设置FIQ模式堆栈
        MSR CPSR_c, #Mode_USR
        LDR SP, =UsrStackSpace ; 设置用户模式堆栈

        LDR R0, =pclk    ; PCLK和波特率
        LDR R1, =baud

        BL Uart_Init     ; 初始化UART

        LDR R0, =pINT_UART  ; INT_UART的入口地址
        LDR R1, =INT_UART   ; 中断服务程序入口地址
        STR R1, [R0]        ; 写入中断向量表
```

1. **ResetHandler**：
   - 这是程序的入口点，通常是系统上电后第一个执行的代码。在这里，它负责初始化时钟和堆栈，并设置不同处理模式的堆栈空间。
2. **Clock_Init**：
   - 这是一个函数调用（`BL Clock_Init`），用于初始化系统的时钟。时钟初始化是嵌入式系统中非常关键的一步，因为它确定了系统的时序和运行速度。
3. **设置堆栈空间**：
   - 使用 `LDR SP, =SvcStackSpace`、`LDR SP, =IrqStackSpace`、`LDR SP, =FiqStackSpace`、`LDR SP, =UsrStackSpace` 分别设置了管理模式、IRQ模式、FIQ模式和用户模式的堆栈空间。这些堆栈空间通常是预先分配好的内存区域，用于处理不同模式下的中断和函数调用。
4. **初始化 UART**：
   - 使用 `BL Uart_Init` 调用了 `Uart_Init` 函数，用于初始化 UART（串口）。UART 初始化包括设置引脚、波特率、数据位数、校验位等参数，以便进行串口通信。

5. **设置中断向量表：**
    - 使用 `LDR R0, =pINT_UART` 将中断向量表中 `INT_UART` 中断服务程序的入口地址存储到 `R0` 寄存器。
    - 使用 `LDR R1, =INT_UART` 将 `INT_UART` 中断服务程序的实际代码地址存储到 `R1` 寄存器。
    - 最后，使用 `STR R1, [R0]` 将 `R1` 寄存器中的地址写入 `R0` 寄存器指向的内存地址，实现了将 `INT_UART` 的中断服务程序入口地址写入中断向量表的操作。

INT_UART

```
.global INT_UART

INT_UART:
    ; 判断接收缓冲器是否有数据
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    TST     R5, #0x1
    BEQ     send_check    ; 如果接收缓冲器无数据，检查发送器

    ; 清除接收缓冲器有数据标志
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    ORR     R5, R5, #0x1
    STR     R5, [R4]

    MOV     R0, #1  ; 设置接收状态标志
    B       INT_UART_dispatch

send_check:
    ; 判断发送器是否为空
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    TST     R5, #0x2
    BEQ     INT_UART_exit    ; 如果发送器非空，退出中断处理

    ; 清除发送器空标志
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    ORR     R5, R5, #0x2
    STR     R5, [R4]

    MOV     R0, #2  ; 设置发送状态标志

INT_UART_dispatch:
    CMP     R0, #1
    BEQ     INT_UART0
    CMP     R0, #2
    BEQ     INT_UART1

INT_UART_exit:
    BX      LR
```

1. **判断接收缓冲器是否有数据：**
    - `LDR R4, =rUTRSTAT0`：将 `rUTRSTAT0` 的地址加载到 `R4` 寄存器。

- `LDR R5, [R4]`：从 `rUTRSTAT0` 寄存器读取数据到 `R5` 寄存器，这个寄存器包含了 UART 接收和发送的状态信息。
- `TST R5, #0x1`：对 `R5` 寄存器的值进行与操作，检查接收缓冲器是否有数据。位运算 `TST` 指令会更新状态寄存器的条件标志位。
- `BEQ send_check`：如果接收缓冲器无数据，则跳转到 `send_check` 标签处继续执行。

2. **清除接收缓冲器有数据标志：**
   - 如果接收缓冲器有数据，会执行以下操作：
     - `ORR R5, R5, #0x1`：将 `R5` 寄存器与 `0x1` 进行或运算，用于清除接收缓冲器有数据的标志。
     - `STR R5, [R4]`：将更新后的 `R5` 寄存器的值写回 `rUTRSTAT0` 寄存器，完成标志位的更新。
3. **设置接收和发送状态标志：**
   - 根据接收和发送的情况设置 `R0` 寄存器，用于后续的中断处理。
4. **分发到具体的中断处理函数：**
   - `INT_UART_dispatch` 标签处根据 `R0` 寄存器的值，决定跳转到相应的中断处理函数：
     - 如果 `R0` 等于 `1`，跳转到 `INT_UART0` 处理接收中断。
     - 如果 `R0` 等于 `2`，跳转到 `INT_UART1` 处理发送中断。
5. **INT_UART_exit：**
   - 这是处理结束的地方，通过 `BX LR` 指令返回到调用 `INT_UART` 的地方，继续执行主程序或其他中断处理。

INT_UART0

```
.global INT_UART0

INT_UART0:
    ; 循环读取字符，直到遇到回车符 '\r'
INT_UART0_loop:
    BL      Uart_Getch     ; 从UART接收一个字符
    CMP     R0, #0x0D      ; 比较是否接收到回车符 '\r'
    BEQ     INT_UART0_exit ; 如果是回车符，则退出循环

    ; 存储到 recvBuffer 中
    LDR     R4, =recvBuffer
    LDR     R5, =recvIndex
    LDRB    R6, [R4, R5]
    STRB    R0, [R4, R6]

    ; 更新 recvIndex
    ADD     R6, R6, #1
    STRB    R6, [R4, R5]

    ; 继续循环读取下一个字符
    B       INT_UART0_loop

INT_UART0_exit:
    ; 清除中断挂起
```

```
    LDR     R4, =rSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 清除子挂起寄存器
    LDR     R4, =rSUBSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #0x1
    STR     R5, [R4]

    ; 清除中断请求
    LDR     R4, =rINTPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    BX      LR
```

1. **循环读取字符，直到遇到回车符 '\r'：**
   - `INT_UART0_loop` 标签标识了一个循环的开始。
   - `BL Uart_Getch`：调用 `Uart_Getch` 函数从 UART 接收一个字符，结果存储在 `R0` 寄存器中。
   - `CMP R0, #0x0D`：将 `R0` 中的字符与回车符（`0x0D`）比较，检查是否接收到回车符。
   - `BEQ INT_UART0_exit`：如果接收到回车符，则跳转到 `INT_UART0_exit` 标签，退出循环。
2. **存储到 recvBuffer 中：**
   - 如果未接收到回车符，则执行以下操作：
     - `LDR R4, =recvBuffer`：将 `recvBuffer` 的地址加载到 `R4` 寄存器。
     - `LDR R5, =recvIndex`：将 `recvIndex` 的地址加载到 `R5` 寄存器。
     - `LDRB R6, [R4, R5]`：从 `recvBuffer` 中读取 `recvIndex` 处的值到 `R6` 寄存器，这个值用于确定存储位置。
     - `STRB R0, [R4, R6]`：将接收到的字符 `R0` 存储到 `recvBuffer` 中 `recvIndex` 处的位置。
     - `ADD R6, R6, #1`：更新 `recvIndex`。
     - `STRB R6, [R4, R5]`：将更新后的 `recvIndex` 存回 `recvBuffer`。
3. **继续循环读取下一个字符：**
   - `B INT_UART0_loop`：无条件跳转回到 `INT_UART0_loop` 标签，继续循环读取下一个字符。
4. **INT_UART0_exit：**
   - 如果接收到回车符或其他退出条件，会执行以下清理操作：
     - 清除中断挂起：通过设置 `rSRCPND` 寄存器的相应位，清除当前中断的挂起状态。
     - 清除子挂起寄存器：通过设置 `rSUBSRCPND` 寄存器的相应位，清除子中断的挂起状态。
     - 清除中断请求：通过设置 `rINTPND` 寄存器的相应位，清除中断请求。
5. **返回：**
   - `BX LR`：返回到调用 `INT_UART0` 的地方。

这样就通过汇编语言实现了字符串的发送。

INT_UART1

```
.global INT_UART1

INT_UART1:
    ; 循环读取和发送字符，直到遇到回车符 '\r'
INT_UART1_loop:
    ; 读取字符
    LDR     R4, =sendBuffer
    LDR     R5, =sendIndex
    LDRB    R6, [R4, R5]

    ; 检查是否为回车符 '\r'
    CMP     R6, #0x0D
    BEQ     INT_UART1_exit  ; 如果是回车符，则退出循环

    ; 发送字符
    BL      Uart_SendByte

    ; 更新 sendIndex
    ADD     R5, R5, #1
    STR     R5, [R4, R5]

    ; 继续循环读取下一个字符
    B       INT_UART1_loop

INT_UART1_exit:
    ; 清除中断挂起
    LDR     R4, =rSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 清除子挂起寄存器
    LDR     R4, =rSUBSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #0x2
    STR     R5, [R4]

    ; 清除中断请求
    LDR     R4, =rINTPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    BX      LR
```

1. **循环读取和发送字符，直到遇到回车符 '\r'：**
    - `INT_UART1_loop` 标签标识了一个循环的开始。
    - `LDR R4, =sendBuffer`：将 `sendBuffer` 的地址加载到 `R4` 寄存器。
    - `LDR R5, =sendIndex`：将 `sendIndex` 的地址加载到 `R5` 寄存器。

- `LDRB R6, [R4, R5]`：从 `sendBuffer` 中读取 `sendIndex` 处的值到 `R6` 寄存器，这个值即将发送的字符。

2. **检查是否为回车符 '\r'：**
   - `CMP R6, #0x0D`：将 `R6` 中的字符与回车符（`0x0D`）比较，检查是否需要退出循环。
   - `BEQ INT_UART1_exit`：如果接收到回车符，则跳转到 `INT_UART1_exit` 标签，退出循环。
3. **发送字符：**
   - `BL Uart_SendByte`：调用 `Uart_SendByte` 函数发送 `R6` 中的字符到 UART。
4. **更新 sendIndex：**
   - `ADD R5, R5, #1`：将 `sendIndex` 加 1，准备指向下一个要发送的字符位置。
   - `STR R5, [R4, R5]`：将更新后的 `sendIndex` 存回 `sendBuffer`。
5. **继续循环读取下一个字符：**
   - `B INT_UART1_loop`：无条件跳转回到 `INT_UART1_loop` 标签，继续循环读取下一个字符并发送。
6. **INT_UART1_exit：**
   - 如果接收到回车符或其他退出条件，会执行以下清理操作：
     - 清除中断挂起：通过设置 `rSRCPND` 寄存器的相应位，清除当前中断的挂起状态。
     - 清除子挂起寄存器：通过设置 `rSUBSRCPND` 寄存器的相应位，清除子中断的挂起状态。
     - 清除中断请求：通过设置 `rINTPND` 寄存器的相应位，清除中断请求。
7. **返回：**
   - `BX LR`：返回到调用 `INT_UART1` 的地方。

2.3 代码：

2.3.1 主函数代码 `test.c`

```
#include "2410addr.h“
#include "option.h"
#include "def.h"
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

extern void Uart_Init(int pclk, int baud);
extern void INT_UART(void);
extern void INT_UART0(void);
extern void INT_UART1(void);
extern void Uart_SendByte(int data);
extern char Uart_Getch(void);

char recvBuffer[100];
char sendBuffer[100];
int recvIndex = 0;
int sendIndex = 0;

int main() {
```

```
    int pclk = 0;  // Initialize pclk and baud as needed
    int baud = 0;

    Uart_Init(pclk, baud);

    while (1) {
        // Main program loop
        // Handle other tasks if needed
    }

    return 0;
}

// UART interrupt handler
void __attribute__((interrupt("IRQ"))) IRQ_Handler(void) {
    INT_UART();
}

// UART0 interrupt handler
void __attribute__((interrupt("IRQ"))) UART0_Handler(void) {
    INT_UART0();
}

// UART1 interrupt handler
void __attribute__((interrupt("IRQ"))) UART1_Handler(void) {
    INT_UART1();
}
```

汇编语言实现

1. `Uart_Init` 函数的汇编实现

```
.global Uart_Init

Uart_Init:
    ; 设置GPH2和GPH3为TXD0和RXD0
    LDR     R4, =rGPHCON
    LDR     R5, [R4]
    ORR     R5, R5, #0xA0
    STR     R5, [R4]

    ; 启用GPH2和GPH3的内部上拉
    LDR     R4, =rGPHUP
    MOV     R5, #0x0
    STR     R5, [R4]

    ; pclk = PCLK
    LDR     R4, =PCLK
    LDR     R5, [R4]
    MOV     R4, R0
    MOV     R5, R4  ; 将 PCLK 存入 pclk

    ; 禁用所有通道的FIFO控制寄存器
    LDR     R4, =rUFCON0
```

```
    MOV     R5, #0x0
    STR     R5, [R4]

    ; 禁用AFC，初始化MODEM控制寄存器
    LDR     R4, =rUMCON0
    MOV     R5, #0x0
    STR     R5, [R4]

    ; 8位数据，无校验位，1位停止位
    LDR     R4, =rULCON0
    MOV     R5, #0x3
    STR     R5, [R4]

    ; 配置控制寄存器
    LDR     R4, =rUCON0
    MOV     R5, #0x325
    STR     R5, [R4]

    ; 设置loopback模式
    LDR     R4, =rUCON0
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 5)
    STR     R5, [R4]

    ; 设置发送和接收中断
    LDR     R4, =rUCON0
    LDR     R5, [R4]
    ORR     R5, R5, #(0x3 << 8)
    STR     R5, [R4]

    ; 清除挂起寄存器
    LDR     R4, =rSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 清除子挂起寄存器
    LDR     R4, =rSUBSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #0x3
    STR     R5, [R4]

    ; 清除中断挂起寄存器
    LDR     R4, =rINTPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 取消UART0中断屏蔽
    LDR     R4, =rINTMSK
    LDR     R5, [R4]
    BIC     R5, R5, #(BIT_UART0)
    STR     R5, [R4]

    ; 取消发送和接收中断屏蔽
    LDR     R4, =rINTSUBMSK
```

```
    LDR     R5, [R4]
    BIC     R5, R5, #(BIT_SUB_TXD0 | BIT_SUB_RXD0)
    STR     R5, [R4]

    BX      LR
```

2. `INT_UART` 函数的汇编实现

```
.global INT_UART

INT_UART:
    ; 判断接收缓冲器是否有数据
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    TST     R5, #0x1
    BEQ     send_check    ; 如果接收缓冲器无数据，检查发送器

    ; 清除接收缓冲器有数据标志
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    ORR     R5, R5, #0x1
    STR     R5, [R4]

    MOV     R0, #1  ; 设置接收状态标志
    B       INT_UART_dispatch

send_check:
    ; 判断发送器是否为空
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    TST     R5, #0x2
    BEQ     INT_UART_exit    ; 如果发送器非空，退出中断处理

    ; 清除发送器空标志
    LDR     R4, =rUTRSTAT0
    LDR     R5, [R4]
    ORR     R5, R5, #0x2
    STR     R5, [R4]

    MOV     R0, #2  ; 设置发送状态标志

INT_UART_dispatch:
    CMP     R0, #1
    BEQ     INT_UART0
    CMP     R0, #2
    BEQ     INT_UART1

INT_UART_exit:
    BX      LR
```

3. `INT_UART0` 函数的汇编实现

```
.global INT_UART0

INT_UART0:
    ; 循环读取字符，直到遇到回车符 '\r'
INT_UART0_loop:
    BL      Uart_Getch      ; 从UART接收一个字符
    CMP     R0, #0x0D       ; 比较是否接收到回车符 '\r'
    BEQ     INT_UART0_exit ; 如果是回车符，则退出循环

    ; 存储到 recvBuffer 中
    LDR     R4, =recvBuffer
    LDR     R5, =recvIndex
    LDRB    R6, [R4, R5]
    STRB    R0, [R4, R6]

    ; 更新 recvIndex
    ADD     R6, R6, #1
    STRB    R6, [R4, R5]

    ; 继续循环读取下一个字符
    B       INT_UART0_loop

INT_UART0_exit:
    ; 清除中断挂起
    LDR     R4, =rSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 清除子挂起寄存器
    LDR     R4, =rSUBSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #0x1
    STR     R5, [R4]

    ; 清除中断请求
    LDR     R4, =rINTPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    BX      LR
```

4. `INT_UART1` 函数的汇编实现

```
.global INT_UART1

INT_UART1:
    ; 循环读取和发送字符，直到遇到回车符 '\r'
INT_UART1_loop:
    ; 读取字符
    LDR     R4, =sendBuffer
    LDR     R5, =sendIndex
```

```
    LDRB    R6, [R4, R5]

    ; 检查是否为回车符 '\r'
    CMP     R6, #0x0D
    BEQ     INT_UART1_exit  ; 如果是回车符，则退出循环

    ; 发送字符
    BL      Uart_SendByte

    ; 更新 sendIndex
    ADD     R5, R5, #1
    STR     R5, [R4, R5]

    ; 继续循环读取下一个字符
    B       INT_UART1_loop

INT_UART1_exit:
    ; 清除中断挂起
    LDR     R4, =rSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    ; 清除子挂起寄存器
    LDR     R4, =rSUBSRCPND
    LDR     R5, [R4]
    ORR     R5, R5, #0x2
    STR     R5, [R4]

    ; 清除中断请求
    LDR     R4, =rINTPND
    LDR     R5, [R4]
    ORR     R5, R5, #(0x1 << 28)
    STR     R5, [R4]

    BX      LR
```

5. `Uart_SendByte` 函数的汇编实现

```
.global Uart_SendByte

Uart_SendByte:
    ; 将数据发送到 UART
    LDR     R4, =0x50000023
    STRB    R0, [R4]

    BX      LR
```

6. `Uart_Getch` 函数的汇编实现

```
.global Uart_Getch

Uart_Getch:
    ; 从 UART 接收一个字符
    LDR     R0, =0x50000027
    LDRB    R0, [R0]

    BX      LR
```

2.3.2 中断向量表`Int_EntryTable.s`

```
pEINT0      DCD 0  ; 外部中断0-3的服务程序入口地址
pEINT1      DCD 0
pEINT2      DCD 0
pEINT3      DCD 0

pEINT4_7    DCD 0  ; 外部中断4-7的服务程序入口地址
pEINT8_23   DCD 0  ; 外部中断8-23的服务程序入口地址

pINT_CAM    DCD 0  ; 26个内部中断服务程序入口地址
pnBATT_FLT  DCD 0
pINT_TICK   DCD 0
pINT_WDT_AC97   DCD 0
pINT_UART   DCD 0

pINT_RTC    DCD 0
pINT_ADC    DCD 0
            END
```

2.3.3中断服务程序`ASM_Interrupt.s`

定义了中断方式

```
Mode_USR EQU 0x50  ; 用户模式，IRQ中断开放，FIQ中断关闭
Mode_FIQ EQU 0xD1  ; FIQ模式，关闭IRQ和FIQ中断
Mode_IRQ EQU 0xD2  ; IRQ模式，关闭IRQ和FIQ中断
Mode_SVC EQU 0xD3  ; 管理模式，关闭IRQ和FIQ中断

        GET 2440Reg_addr.inc
        AREA MyCode, CODE, READONLY
        IMPORT Uart_Init
        IMPORT INT_UART
        IMPORT INT_UART0
        IMPORT INT_UART1
        ENTRY

        B ResetHandler    ; 复位中断服务程序
        B .               ; 未定义指令异常处理
        B .               ; SWI中断处理
        B .               ; 指令预取中止异常处理
```

```
        B .               ; 数据访问中止异常处理
        B .               ; 保留
        B HandlerIRQ      ; IRQ中断处理
        B .               ; FIQ中断处理

ResetHandler
        BL Clock_Init     ; 初始化时钟
        LDR SP, =SvcStackSpace ; 设置管理模式堆栈
        MSR CPSR_c, #Mode_IRQ
        LDR SP, =IrqStackSpace ; 设置IRQ模式堆栈
        MSR CPSR_c, #Mode_FIQ
        LDR SP, =FiqStackSpace ; 设置FIQ模式堆栈
        MSR CPSR_c, #Mode_USR
        LDR SP, =UsrStackSpace ; 设置用户模式堆栈

        LDR R0, =pclk     ; PCLK和波特率
        LDR R1, =baud

        BL Uart_Init      ; 初始化UART

        LDR R0, =pINT_UART  ; INT_UART的入口地址
        LDR R1, =INT_UART   ; 中断服务程序入口地址
        STR R1, [R0]        ; 写入中断向量表

MAIN_LOOP
        NOP
        B MAIN_LOOP       ; 主循环，等待中断

Clock_Init
        GET Clock_Init.s ; 初始化时钟

MemSetup
        GET MemSetup.s    ; 初始化内存

Init_DATA
        GET Init_DATA.s   ; 初始化数据区

HandlerIRQ
        SUB LR, LR, #4    ; 调整返回地址
        STMFD SP!, {LR}   ; 保存LR到堆栈
        LDR LR, =Int_Return ; 中断返回地址
        LDR R0, =INTSUBMSK ; 屏蔽子中断
        LDR R1, [R0]
        ORR R1, R1, #0x3
        STR R1, [R0]
        LDR R0, =INTOFFSET ; 获取中断源编号
        LDR R1, [R0]
        LDR R2, =Int_EntryTable ; 中断向量表起始地址
        LDR PC, [R2, R1, LSL #2

] ; 跳转到中断服务程序

Int_Return
        LDMFD SP!, {PC}^ ; 恢复LR并返回

        END
```

3、实验总结

本次实验主要是关于嵌入式系统中UART通信的原理和实现方法，以及中断方式的优化和相关寄存器的作用。以下是实验中涉及到的关键点和总结：

1. UART初始化（`Uart_Init`）

在本实验中，使用了`Uart_Init`函数来配置UART通信的基本设置，包括波特率、数据格式、中断设置以及GPIO配置。具体步骤如下：

- 使用 `rGPHCON` 寄存器配置 TXD0 和 RXD0 引脚。
- 通过 `rGPHUP` 寄存器启用这些引脚的内部上拉。
- 配置 UART0 为 8 位数据位、无校验位、1 位停止位（`rULCON0` 寄存器）。
- 设置 UART0 的控制寄存器（`rUCON0`），包括启用循环测试模式和接收/发送中断。
- 清除中断挂起、子挂起和中断请求，确保中断环境的清晰和正常运行。

2. UART中断处理（`INT_UART`，`INT_UART0`，`INT_UART1`）

实验中实现了针对UART接收和发送的中断处理函数，分别为 `INT_UART`，`INT_UART0`，`INT_UART1`：

- `INT_UART` **函数**：用于处理通用的UART中断，根据接收和发送状态来区分具体的处理。
- `INT_UART0` **函数**：专门处理UART0的接收中断，从UART接收缓冲区中读取数据，并存储到接收缓冲区中。
- `INT_UART1` **函数**：专门处理UART1的发送中断，从发送缓冲区中读取数据，并发送到UART。

在这些函数中，使用了寄存器级的操作，如 `LDR`，`STR`，`CMP`，`B` 等指令来实现数据的传输和状态的检测，确保了UART通信的可靠性和效率。

3. 主程序（`main` 函数）

主程序负责初始化系统并进入一个无限循环，以便执行其他任务或等待中断的触发。主要步骤包括：

- 初始化系统时钟和其他必要的资源。
- 调用 `Uart_Init` 函数来初始化UART通信。
- 进入一个永久循环（`while (1)`），在这里可以添加额外的应用程序逻辑或任务处理。

通过本实验，深入学习了在实现UART通信和中断处理的相关知识和技能。掌握了如何配置和初始化UART、如何处理接收和发送中断，并且能够使用汇编语言编写有效的中断处理程序。这些技能对于嵌入式系统开发和实时通信应用具有重要意义，为进一步的硬件和软件开发奠定了坚实的基础。