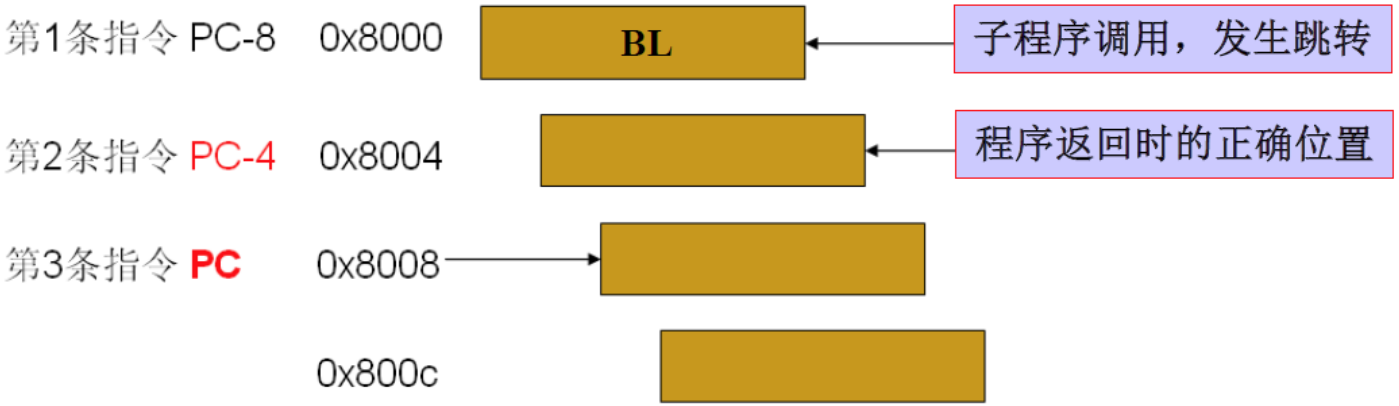
子程序调用时的LR

- 执行第1条指令（地址0x8000）时，PC正指向第3条指令（地址0x8008）。第1条指令是调用指令BL。
- 执行完子程序返回时，PC应指向第2条指令（地址0x8004）。
- 第1条指令执行时，ARM硬件自动把PC（=0x8008）保存至LR，并将LR自动调整为：LR=LR-0x4（即LR=0x8004）。
- 最终保存在LR中的是第2条指令的地址，正好是正确的返回地址。
- 由于各种异常中断响应的过程不同，因此，保存在LR中的地址是不同的。大多数情况下，保存在LR中的地址是：LR=PC-4。



SWI和未定义指令时的LR

- SWI 和未定义指令异常中断是由当前执行的指令自身产生的。
- 当SWI和未定义指令异常中断产生时，PC的值还未更新，它指向当前执行指令后面的第2条指令。
- 当SWI和未定义指令异常中断发生时，ARM将值（PC-4）保存到异常模式下的LR（R14_svc或R14_und）中。这时（PC-4）即指向当前指令的下一条指令。
- 返回操作：可通过指令“MOVS PC, LR”来实现。同时，SPSR_svc或SPSR_und的内容被复制到当前程序状态寄存器中。

图 SWI的返回地址

指令预取中止时的LR

- 指令预取中止异常中断是由当前执行的指令自身产生的。发生异常后，程序要返回到这个有问题的指令处，重新读取并执行该指令。
- 指令预取中止异常中断产生时，PC的值还未更新，它指向当前执行指令后面的第2条指令。此时，ARM将值（PC-4）保存到异常模式下的LR（R14_abt）中。这时（PC-4）即指向当前指令的下一条指令。
- 返回操作：可通过指令“SUBS PC, LR, #4”来实现。同时，SPSR_abt的内容被复制到当前程序状态寄存器中。

图 指令预取异常的返回地址

数据访问中止时的LR

- 数据访问中止异常中断是由数据访问指令自身产生的。发生异常后，程序要返回到这个有问题的数据访问处，重新访问该数据。
- 数据访问中止异常中断产生时，PC的值已经更新，它指向当前执行指令后面的第3条指令。此时，ARM将值（PC-4）保存到异常模式下的LR（R14_abt）中。这时（PC-4）即指向当前指令后的第2条指令。
- 返回操作：可通过指令“SUBS PC, LR, #8”来实现。同时，SPSR_abt的内容被复制到当前程序状态寄存器中。

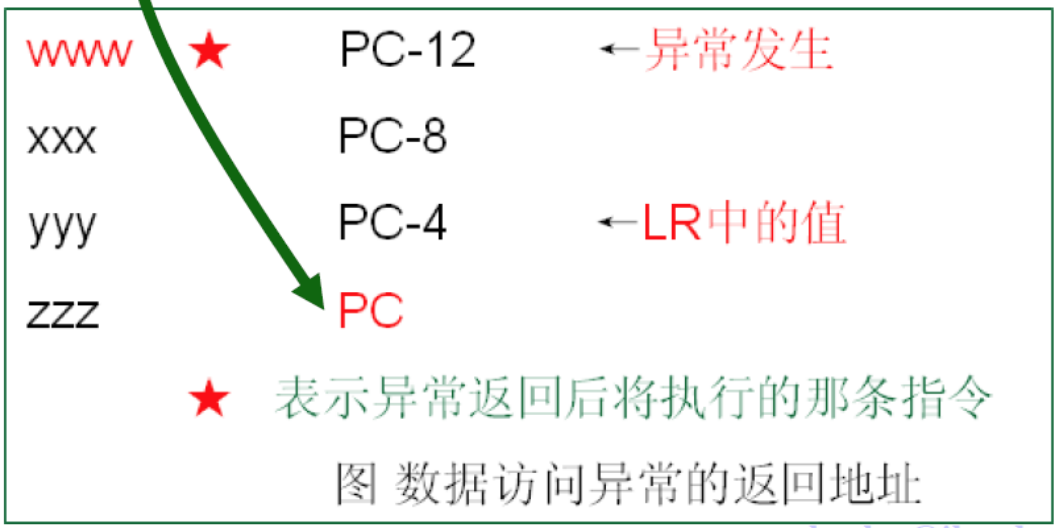


图 数据访问异常的返回地址

IRQ与FIQ时的LR

- ARM执行完当前指令后，查询IRQ与FRQ，若有中断请求，且CPU允许中断，将产生IRQ中断或FIQ中断。
- 当IRQ或FIQ中断产生时，PC的值已经更新，它指向当前执行指令后面的第3条指令。IRQ或FIQ中断发生时，ARM将值（PC-4）保存到异常模式下的LR（R14_irq或R14_fiq）中。这时（PC-4）即指向当前指令后的第2条指令。
- 返回操作：可通过指令“SUBS PC, LR, #4”来实现。同时，SPSR_irq或SPSR_fiq的内容被复制到当前程序状态寄存器中。

图 IRQ/FRQ的返回地址

转移到一级中断源的中断服务程序

```
IRQ_Handler:
    sub     sp, sp, #4          ; 在堆栈中预留一个字单元，保存转移目标的入口地址
    stmfd   sp!, {r8-r9}        ; r8、r9进栈，保护r8、r9
    ldr     r9, =INTOFFSET      ; 将INTOFFSET寄存器地址送r9
    ldr     r9, [r9]            ; 将INTOFFSET中的中断偏移值送r9，
                                ; 该值代表被响应的中断
    ldr   r8, =HandleEINT0      ; 将一级中断源的中断向量表首地址送r8
    add   r8, r8, r9, lsl #2    ; 计算被响应中断的中断向量地址， r8=r8+r9×4
    ldr     r8, [r8]            ; 读取被响应中断的中断向量（即中断服务程序入口地址）
    str     r8, [sp, #8]        ; 将获取的中断向量送入堆栈预留的字单元
    ldmfd  sp!, {r8-r9, pc}     ; 恢复r8、r9，中断向量弹出到PC，
                                ; ARM按PC取指令，转入被响应中断的中断服务程序
```

16.冒泡排序和二分查找算法

```
;首地址保存在r0寄存器中，数组长度保存在r1寄存器中。
BubbleSort
   STMFD sp!, {r4-r6, lr}
   SUB r4, r1, #1
OuterLoop
   MOV r5, r4
InnerLoop
   LDR r2, [r0, r5, LSL #2]
   SUB r3, r5, #1
   LDR r6, [r0, r3, LSL #2]
   CMP r2, r6
   BLE NoSwap
   STR r2, [r0, r3, LSL #2]
   STR r6, [r0, r5, LSL #2]
NoSwap
   SUBS r5, r5, #1
   BPL InnerLoop
   SUBS r4, r4, #1
   BPL OuterLoop
   LDMFD sp!, {r4-r6, pc}

;假设数组已经排序，搜索键保存在r2寄存器中。如果找到，r0将被设置为数组中的位置，否则r0将被设置为-1。
BinarySearch
   STMFD sp!, {r4-r6, lr}
   MOV r4, #0;r4=头
   SUB r5, r1, #1;r5=尾-1
SearchLoop
   ADD r1, r4, r5
   MOV r1, r1, ASR #1;mid/2
   LDR r3, [r0, r1, LSL #2];r3=a[mid/2]
   CMP r2, r3;
   BEQ Found;找到
   BLT Lower;头~mid
   ADD r4, r1, #1;mid+1~尾，r4=头=mid+1
   B Continue
Lower
   MOV r5, r1;r5=尾=mid
Continue
   CMP r4, r5;比较头尾，是否结束
   BLE SearchLoop
   MOV r0, #-1;失败
   B End
Found
   MOV r0, r1
End
   LDMFD sp!, {r4-r6, pc}

;测试程序
start
   LDR r0, =array
   LDR r1, =5
   BL BubbleSort
   MOV r2, #15
   BL BinarySearch
   B end

array   DCD 20, 15, 10, 5, 0

end
   MOV r0, #0x18
   LDR r1, =0x20026
   SWI 0x123456
```

19.将连续1000个字节单元数据写入文件

```
#include <stdio.h>
#define START_ADDR 0x40003200
#define BYTE_COUNT 1000
int main() {
   FILE *file =
fopen("MemoryData.dat", "wb");
   if (file == NULL) {
      printf("无法打开文件\n");
      return 1;
   }
   unsigned char *mem_ptr =
(unsigned char *)START_ADDR;
   for (int i = 0; i < BYTE_COUNT; i++) {
      fputc(*(mem_ptr + i), file);
   }
   fclose(file);
   printf("内存数据已写入文件\n");
   return 0;
}
```

22. 内存复制子程序，考虑重叠和不重叠

```
; 输入参数：r0 - 源地址, r1 - 目标地址, r2 - 字节数
; 输出参数：无
memcpy:
   cmp r0, r1          ; 比较源地址和目标地址
   beq finish          ; 如果相等则不需要复制
   blt reverse_copy    ; 如果源地址小于目标地址，需要反向复制

forward_copy:
   ldrb r3, [r0], #1   ; 从源地址加载一个字节，并后移源指针
   strb r3, [r1], #1   ; 将字节存储到目标地址，并后移目标指针
   subs r2, r2, #1     ; 减少字节数
   bne forward_copy    ; 如果还有字节需要复制，继续循环
   b finish            ; 完成正向复制

reverse_copy:
   add r0, r0, r2      ; 将源指针移动到最后一个字节
   add r1, r1, r2      ; 将目标指针移动到最后一个位置
reverse_loop:
   subs r0, r0, #1     ; 后移源指针
   subs r1, r1, #1     ; 后移目标指针
   ldrb r3, [r0]       ; 从源地址加载一个字节
   strb r3, [r1]       ; 将字节存储到目标地址
   subs r2, r2, #1     ; 减少字节数
   bne reverse_loop    ; 如果还有字节需要复制，继续循环

finish:
   bx lr               ; 返回调用者
```

23.100个压缩BCD码求和

二进制 转 BCD
BCD 转 二进制

```
 AREA BCDADD, CODE, READONLY
   ENTRY

start:
   MOV r1, #100        ; 设置循环计数器
   LDR r2, =0x40003200  ; 压缩BCD码的内存地址
   LDR r3, =0x40003400  ; 存放结果的内存地址
   MOV r4, #0          ; 初始化结果为0

addition:
   LDR r0, [r2], #4     ; 加载一个字的BCD码，并将指针后移
   BL bcd_to_bin        ; 将BCD码转换为二进制
   ADD r4, r4, r0       ; 进行加法运算
   SUBS r1, r1, #1      ; 减少循环计数器
   BNE addition         ; 如果还没有处理完所有的BCD字，继续循环

   BL bin_to_bcd        ; 将加法结果从二进制转换为BCD码
   STR r0, [r3]         ; 将结果存储到目标地址

   MOV r0, #0x18        ; 正常退出
   LDR pc, =_sys_exit

; _sys_exit 这部分代码根据你的系统环境来编写，这只是一个示例，你可能需要根据你的具体环境来更改。
_sys_exit:
   SWI 0x11

; 子程序：bcd_to_bin
; 输入参数：r0 - BCD码
; 输出参数：r0 - 二进制
bcd_to_bin:
   AND r1, r0, #0x000F ; 获取低4位
   MOV r2, r0, LSR #4  ; 获取高4位
   MUL r0, r2, #10     ; 高4位乘以10
   ADD r0, r0, r1      ; 加上低4位
   BX lr               ; 返回调用者

; 子程序：bin_to_bcd
; 输入参数：r0 - 二进制
; 输出参数：r0 - BCD码
bin_to_bcd:
   MOV r1, #0          ; 初始化BCD码为0
   MOV r2, #1000       ; 设置除数为1000（因为最大可能的二进制数是9999）
bin_to_bcd_loop:
   MOV r3, r0, LSR #28 ; 获取高4位
   CMP r3, r2          ; 比较二进制数和除数
   BLT next_digit      ; 如果二进制数小于除数，处理下一个数字
   SUB r0, r0, r2      ; 从二进制数中减去除数
   ADD r1, r1, #0x1000 ; 在BCD码中添加一个1（因为我们是从高位开始的，所以是0x1000而不是0x0001）
   B bin_to_bcd_loop   ; 继续循环，直到二进制数小于除数
next_digit:
   MOV r1, r1, LSR #4  ; 将BCD码右移4位，为下一个数字预留空间
   MOV r2, r2, LSR #4  ; 将除数右移4位，为下一个数字预留空间
   CMP r2, #0          ; 检查是否处理了所有的数字
   BNE bin_to_bcd_loop ; 如果没有，继续循环
   MOV r0, r1          ; 将结果存储到r0
   BX lr               ; 返回调用者

   END; 返回调用者
```

23. 二进制 转 BCD

```
; 子程序：bin_to_bcd
; 输入参数：r0 - 二进制
; 输出参数：r0 - BCD码

bin_to_bcd:
        MOV R2,#0 ;R2存结果
        MOV R1,#0 ;R1保存每一位的计数
LoopK:
        CMP R0,#1000 ;和千位比较
        ADDGT R1,R1,#1
        SUBS R0,R0,#1000
        ORRLT R2,R2,R1,LSL#12 ;将计数值
移到对应位，注意百位和十位移动8、
4位
        MOVLT R1,#0 ;清零计数值
        BLT LoopB
        BGT LoopK
;下面百位十位个位略去
end:
        MOV R0,R2
        MOV PC,LR ;返回
```

24.ASCII码数据串115200和24相乘

ASCII 转 二进制

```
 AREA MULT, CODE, READONLY
   ENTRY

start:
   LDR r1, =str1        ; 字符串1的地址
   BL ascii_to_bin      ; 将ASCII转换为二进制
   MOV r3, r0           ; 保存结果到r3

   LDR r1, =str2        ; 字符串2的地址
   BL ascii_to_bin      ; 将ASCII转换为二进制

   MUL r0, r0, r3       ; 将两个结果相乘

   LDR r1, =0x40003100  ; 结果的内存地址
   STR r0, [r1]         ; 将结果存储到内存

   MOV r0, #0x18        ; 正常退出
   LDR pc, =_sys_exit

; _sys_exit 这部分代码根据你的系统环境来编写，这只是一个示例，你可能需要根据你的具体环境来更改。
_sys_exit:
   SWI 0x11


; 子程序：ascii_to_bin
; 输入参数：r1 - ASCII字符串的地址
; 输出参数：r0 - 二进制
ascii_to_bin:
   MOV r0, #0           ; 初始化结果为0
ascii_to_bin_loop:
   LDRB r2, [r1], #1    ; 加载一个字节，并将指针后移
   CMP r2, #'0'         ; 检查字符是否为'0'
   BLT end_of_string    ; 如果小于'0'，则字符不是数字，字符串结束
   CMP r2, #'9'         ; 检查字符是否为'9'
   BGT end_of_string    ; 如果大于'9'，则字符不是数字，字符串结束
   SUB r2, r2, #'0'     ; 将字符转换为数字
   MUL r0, r0, #10      ; 将结果乘以10
   ADD r0, r0, r2       ; 将数字添加到结果
   B ascii_to_bin_loop  ; 继续循环，直到字符串结束
end_of_string:
   BX lr                ; 返回调用者

section .data
str1: .asciz "115200"    ; 字符串1
str2: .asciz "24"        ; 字符串2

   END
```

38. 跳转表

复制

```
AREA    Jump, CODE, READONLY
        CODE32
num       EQU     2                ; 跳转表的入口数目
        ENTRY                      ; 程序入口
start
  MOV       r0, #0              ; 传递给子程序的参数
  MOV       r1, #3    ; 传递给子程序的参数
  MOV       r2, #2    ; 传递给子程序的参数
  MOV       R3, #0             ; 调用散列表中的子程序序号
  BL        arithfunc           ; 调用计算地址的子程序
stop
  MOV       r0, #0x18           ; 执行中止
  LDR       r1, =0x20026
  SWI       0x123456
arithfunc
  CMP       r3, #num           ; 比较参数
  MOVHS     pc, lr              ; HS 无符号大于
  ADR       r4, JumpTable      ; 装载地址表首地址
  LDR       pc, [r4, r3, LSL#2]          ; 跳转到相应子程序入口地址处
JumpTable
  DCD      DoAdd
  DCD      DoSub
DoAdd
          ADD       r0, r1, r2         ; =0时的操作
          MOV       pc, lr             ; 返回
DoSub
          SUB       r0, r1, r2         ; =1时的操作
          MOV       pc, lr              ; 返回
          END                         ; 程序结尾
```

40. Arm调用C语言实现阶乘

```
//factorial.c
#include <stdint.h>

uint64_t factorial(int n) {
   uint64_t result = 1;
   for(int i = 2; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

```
;main.s
AREA FAC,CODE,READONLY
IMPORT FACTORIAL

_start:
   MOV r0, #20        ; 加载参数到r0
   BL factorial       ; 调用C函数factorial

   ; 返回值在r0和r1中，其中r0是低32位，r1是高32位
```

41.100个字节填充，然后相加

```c
#include <stdint.h>
uint64_t add100(unsigned char* addr)
{
    int n=0;
    for (i=0;i<=99;i++)
        n=n+*(addr+i);
    return n;
}
```

```
AREA ADD100,CODE,READONLY
IMPORT add100
ENTRY
start:
    MOV R0,#1;
LOOP:
    LDR R1,=0X4000F000
    STR R0,[R1],#1
    ADD R0,R0,#1
    CMP R0,#101
    BNE LOOP
add100:
    LDR R0=0x4000F000
    BL add100
stop:
    MOV r0, #0x18
    LDR r1, =0x20026
    SWI 0x123456
END
```

43.课堂作业　编程

1、编写汇编程序实现1x2+2x3+3x4+4x5…9x10表达式的功能

```
 AREA HomeWork,CODE,READONLY
 ENTRY
N   EQU   9
start
      MOV R0,#0    ;乘加和
      MOV R1,#1    ;第一个乘数
      MOV R2,#2    ;第二个乘数
      LDR R3,=N    ；循环次数
loop  MLA R0,R1,R2,R0;   R0=R0+R1xR2 乘加和
      ADD R1,R1,#1  ;第一个数+1
      ADD R2,R2,#1  ；第二个数+1
      SUBS R3,R3,#1 ；循环次数-1
      BNE loop      ；循环次数<>0，转loop
 END
```

MLA　Rd, Rm, Rs, Rn	32 位乘加指令	Rd←Rm*Rs+Rn　　(Rd≠Rm)

44.45.课堂作业　编程

2、用汇编程序实现C程序功能

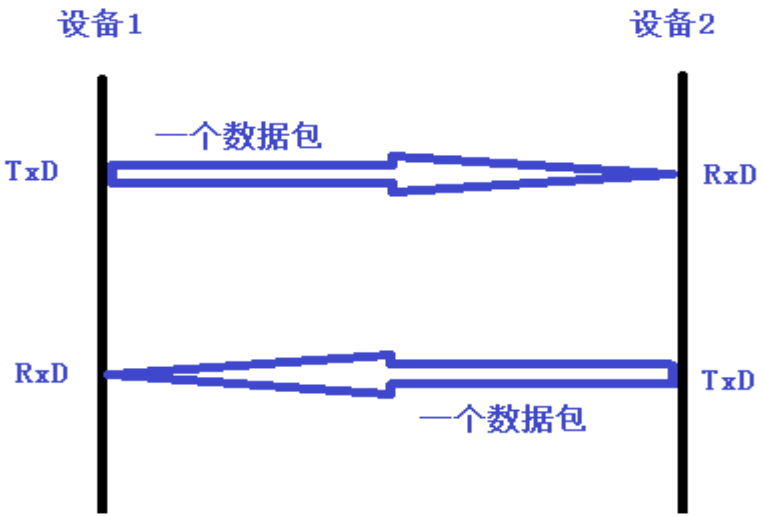
```
char Sendbuf[256];
unsigned char  SendLen;
unsigned char pack(unsigned char CMD,char *buf,unsigned char buflen)
{
 unsigned char i,sum=0;
 if ((buflen<=0)||(buflen>255)) return 0;
 SendLen=0;
 Sendbuf[SendLen++]=0xAA;
 Sendbuf[SendLen++]=CMD;
 Sendbuf[SendLen++]=buflen;
 for (i=0;i<buflen;i++) Sendbuf[SendLen++]=buf[i];
 for (i=0;i<SendLen;i++) sum+=Sendbuf[i];
 Sendbuf[SendLen++]=sum;
 return 0;
}
```

课堂作业　编程

串行通信用户自定义格式

数据头	命令	数据长度	数据	字节校验和
Head	CMD	Datalength	Data	Checksum
1Byte	1Byte	1Byte		1Byte
0xAA 或0x55				前4部分字节和

数据包格式



课堂作业　编程

```
AREA HomeWork2,CODE,READONLY
ENTRY
EXPORT pack
;r0==>CMD    r1==>buf  r2==buflen 入口参数
pack  LDR R4,= addrSendBuf
    LDR R4,[R4]        ;R4===>SendBuf


    MOV R5,#0          ;SendLen
    ;Head
    LDR R6,=0xAA
    STRB R6,[R4],#1
    ADD R5,R5,#1       ;缓冲区字节数
    ;CMD
    STRB R0,[R4],#1
    ADD R5,R5,#1
    ;buflen
    STRB R2,[R4],#1
    ADD R5,R5,#1
    ;buf
LOOP  LDRB R6,[R1],#1
    STRB R6,[R4],#1
    ADD R5,R5,#1
    SUBS R2,R2,#1
    BNE LOOP
    ;CheckSum
    LDR R7,=addrSendBuf
    LDR R7,[R7]        ;R7===>SendBuf
    MOV R8,R5          ;前4部分数据字节数
    MOV R9,#0          ;CheckSum
LOOP1 LDRB R10,[R7],#1
    ADD R9,R9,R10
    SUBS R8,R8,#1
    BNE LOOP1
    STRB R9,[R4],#1
    ADD R5,R5,#1
    ;返回R0
    LDR R7= addrSendLen
    LDRB R7,[R7]
    STRB R5,[R7]     ;SendLen
    MOV R0,R5        ;return value
    MOV PC,LR


AREA HomeWorkD,DATA,READWRITE
addrSendBuf   DCD        SendBuf
addrSendLen   DCD        SendLen

IMPORT SendBuf           ;C变量,直接用就可以了
IMPORT Sendlen           ;C变量

END
```

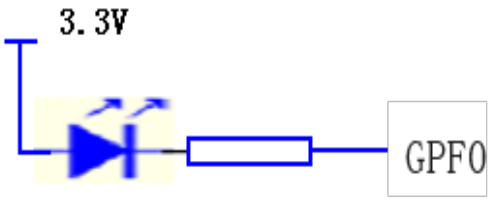
44.45.课堂作业　编程

第二小题，接收数据的实现

```
#include <stdint.h>
#define FRAME_START 0xAA

char RcvBuf[1024];
unsigned int RcvLen;

unsigned char extract_cmd(void) {
    if (RcvLen < 3) return 0; // 如果接收到的数据长度小于3，那么数据帧格式不正确
    return RcvBuf[1]; // 返回命令字节
}

unsigned char checksum(void) {
    unsigned char sum = 0;
    for (unsigned int i = 0; i < RcvLen - 1; i++) {
        sum += RcvBuf[i];
    }
    return sum;
}

int extract_data(char* buf, unsigned int buflen) {
    if (RcvLen < 4 || RcvBuf[0] != FRAME_START || RcvBuf[2] > buflen || RcvBuf[RcvLen - 1] != checksum()) {
        return -1; // 数据帧格式不正确或校验失败
    }
    for (unsigned int i = 0; i < RcvBuf[2]; i++) {
        buf[i] = RcvBuf[3 + i]; // 提取数据
    }
    return RcvBuf[2]; // 返回数据长度
}

void receive_frame(char* frame, unsigned int len) {
    if (len > sizeof(RcvBuf)) return; // 如果帧太大，那么忽略它
    for (unsigned int i = 0; i < len; i++) {
        RcvBuf[i] = frame[i]; // 保存接收到的数据帧
    }
    RcvLen = len; // 更新接收到的数据长度
}
```

硬件应用题　16

已知电路图如下:



使用S3C2440 的F口GPF0经电阻与LED发光管输出极连接，发光管输入极连接到电源正3.3V。.

要求回答下列问题:

（1） 说明LED接口电路的工作原理.

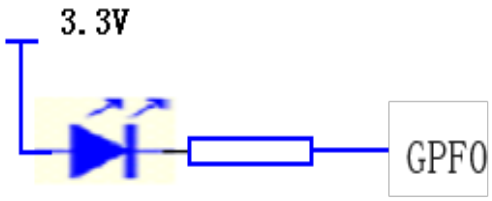
（2） 编程利用定时器0（PCLK为50MHZ）实现LED闪亮。

硬件应用题　16　01

1. 说明LED接口电路的工作原理.

当GPF0输出高电平时，LED两端都是高电平,LED不亮;

当GPF0输出低电平时，LED输入端高电平，输出端低电平,LED亮;



2.编程利用定时器0（PCLK为50MHZ）实现LED闪亮。

(1)实现方法:

第1步：初始化：配置定时器定时1S；配置定时器定时到时，启动中断。

第2步：LED闪亮函数:

在中断服务程序内，控制LED亮;

延时一段时间,

控制LED不亮;

第3步:中断服务程序

LED闪亮函数

清除中断记录标志，定时器可继续中断

硬件应用题　16　02

(2)引脚初始化

```
#define rGPFCON (*((volatile unsigned char *) 0x56000050))
#define rGPFDAT (*((volatile unsigned char *) 0x56000054))
#define rGPFUP (*((volatile unsigned char *) 0x56000058))
GFPCON=2_0000 0000 0000 0001=0x0001
GPFUP=0x00;
```

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input　01 = Output 10 = EINT[7]　11 = Reserved
GPF6	[13:12]	00 = Input　01 = Output 10 = EINT[6]　11 = Reserved
GPF5	[11:10]	00 = Input　01 = Output 10 = EINT[5]　11 = Reserved
GPF4	[9:8]	00 = Input　01 = Output 10 = EINT[4]　11 = Reserved
GPF3	[7:6]	00 = Input　01 = Output 10 = EINT[3]　11 = Reserved
GPF2	[5:4]	00 = Input　01 = Output 10 = EINT2]　11 = Reserved
GPF1	[3:2]	00 = Input　01 = Output 10 = EINT[1]　11 = Reserved
GPF0	[1:0]	00 = Input　01 = Output 10 = EINT[0]　11 = Reserved

```
ProtF_Init()
{
rGFPCON=0x0001;
rGPFUP=0x00;

}
```

硬件应用题　16　02

(2)LED闪亮函数

- 延时子程序

```
void Delay ( unsigned int x )                    //延时程序
{    unsigned int i,j,k;
     for(i=0;i<=x;i++)
                for(j=0;j<=0xff;j++)
                     for(k=0;k<=0xff;k++);
}
```

- 闪亮子程序

```
void  shanliang()
{
   rGPFDAT=0x00;       /GPF0=0  LED亮
    Delay(10);
  rGPFDAT=0x01;       /GPF0=1  LED不亮
    Delay(10);

}
```

硬件应用题　16　03

(3)Timer0初始化子程序

```
void Timer0_Iint(void)
{
rTCFG0 = 0xff;                  // 预分频计数器 = 255
rTCFG1 = 0x03;                  // 0011 : 1/16设定为16分频
rTCON = 0x02;                   // 00  1  0，手动更新
}

Void Timer0_start()
{rTCNTB0 = 5*2440;              // 计数初值，(TCNT + 1) * 81.92 = 1
 rTCON |= 0x01;       //TCON[0]=1 启动定时器
}
Void Timer0_stop()
{
rTCON &= 0xFFFFFE;       //TCON[0]=0 停止定时器
}
```

TCON	Bit	Description	Initial state
Reserved	[7:5]	Reserved	
Dead zone enable	[4]	Determine the dead zone operation. 0 = Disable　1 = Enable	0
Timer 0 auto reload on/off	[3]	Determine auto reload on/off for Timer 0. 0 = One-shot　1 = Interval mode(auto reload)	0
Timer 0 output inverter on/off	[2]	Determine the output inverter on/off for Timer 0. 0 = Inverter off　1 = Inverter on for TOUT0	0
Timer 0 manual update (note)	[1]	Determine the manual update for Timer 0. 0 = No operation　1 = Update TCNTB0 & TCMPB0	0
Timer 0 start/stop	[0]	Determine start/stop for Timer 0. 0 = Stop　1 = Start for Timer 0	0

TCFG1	Bit	Description	Initial State
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt)　0001 = Timer0 0010 = Timer1　0011 = Timer2 0100 = Timer3　0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK0	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK0	0000

硬件应用题 16 04

INTOFFSET	偏移量	INTOFFSET	偏移量	INTOFFSET	偏移量	INTOFFSET	偏移量
INT_ADC	31	INT_UART1	23	INT_UART2	15	nBATT_FLT	7
INT_RTC	30	INT_SPI0	22	INT_TIMER4	14	INT_CAM	6
INT_SPI1	29	INT_SDI	21	INT_TIMER3	13	EINT8_23	5
INT_UART0	28	INT_DMA3	20	INT_TIMER2	12	EINT4_7	4
INT_IIC	27	INT_DMA2	19	INT_TIMER1	11	EINT3	3
INT_USBH	26	INT_DMA1	18	INT_TIMER0	10	EINT2	2
INT_USBD	25	INT_DMA0	17	INT_WDT_AC97	9	EINT1	1
INT_NFCON	24	INT_LCD	16	INT_TICK	8	EINT0	0

(4)Timer0中断服务子程序

```
static void_irq Timer0_ISR(void)
{
shanliang()
 rSRCPND=1<<10; //SRCPND[10]=>Timer0,写入1，清除Timer0中断请求记录
 rINTPND=1<<10; //INTPND[10]=>Timer0,写入1，清除Timer0中断服务记录
 Timer0_start();          //再次启动 Timer0
}
```

(5)Timer0向量地址变量

中断向量表首地址=_ISR_STARTADDRESS+0x20 //EINT0的中断向量地址

Timer0中断向量地址=中断向量表首地址+10*4

```
#define pISR_Timer0 (*(unsigned *)(_ISR_STARTADDRESS+0x20+10*4))
```

硬件应用题　16　05

(6)Timer0中断初始化

```
Void Timer0_int_Init()
{rTCFG1&=~(0xF<20);  //清除TCFG1[23:20],开发Timer中
 rPRIORITY = 0x0000 007F;  // 使用默认的循环优先级
 rINTMOD = 0x0000 0000;  // 所有中断均为默认的IRQ中断
 rINTMSK&=~(1<<10);          //Timer0开中断
 pISR_Timer0= (unsigned)Timer0_ISR;
rSRCPND=1<<10;
 rINTPND=1<<10;
}
```

TCFG1	Bit	Description	Initial State
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt)　0001 = Timer0 0010 = Timer1　0011 = Timer2 0100 = Timer3　0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK1	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK0	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2　0001 = 1/4　0010 = 1/8 0011 = 1/16　01xx = External TCLK0	0000

DMA Mode	DMA Request	Timer0 INT	Timer1 INT	Timer2 INT	Timer3 INT	Timer4 INT
0000	No select	ON	ON	ON	ON	ON

(7)程序

```
#define rGPFCON (*((volatile unsigned char *) 0x56000050))
#define rGPFDAT (*((volatile unsigned char *) 0x56000054))
#define rGPFUP (*((volatile unsigned char *) 0x56000058))
#define pISR_Timer0 (*(unsigned *)(_ISR_STARTADDRESS+0x20+10*4))
#define rSRCPND (*((volatile unsigned char *) 0X4A000000))
#define rINTMOD (*((volatile unsigned char *) 0X4A000004))
#define rINTMSK (*((volatile unsigned char *) 0X4A000008))
#define rPRIORITY (*((volatile unsigned char *) 0x4A00000C ))
#define rINTPND (*((volatile unsigned char *) 0X4A000010))
ProtF_Init()    //端口F初始化
{rGFPCON=0x0001;
rGPFUP=0x00;
}
void Delay ( unsigned int x )                    //延时程序
{          unsigned int i,j,k;
           for(i=0;i<=x;i++)
                      for(j=0;j<=0xff;j++)
                                 for(k=0;k<=0xff;k++);
}
void  shanliang()   //闪亮子程序
{
   rGPFDAT=0x00;       /GPF0=0   LED亮
   Delay(10);
  rGPFDAT=0x01;       /GPF0=1   LED不亮
   Delay(10);
}
void Timer0_Iint(void) //定时器初始化
{
rTCFG0 = 0xff;                  // 预分频计数器 = 255
rTCFG1 = 0x03;                  // 0011 : 1/16设定为16分频
rTCNTB0 = 5*2440;               // 计数初值，(TCNT + 1) * 81.92 = 1 秒
rTCON = 0x02;                                 // 00   1   0，手动更新
}
```

```
Void Timer0_start()  //启动Timer0定时
{rTCNTB0 = 5*2440;              // 计数初值，(TCNT + 1) * 81.92 = 1秒
 rTCON |= 0x01;       //TCON[0]=1 启动定时器
}
Void Timer0_stop()  //结束Timer0定时
{
rTCON &= 0xFFFFFE;       //TCON[0]=0 停止定时器
}

//Timer0中断服务程序
static void_irq Timer0_ISR(void)
{
shanliang()
 rSRCPND=1<<10;  //SRCPND[10]=>Timer0,写入1，清除Timer0中断请求记录
 rINTPND=1<<10;  //INTPND[10]=>Timer0,写入1，清除Timer0中断服务记录
 Timer0_start();           //再次启动 Timer0
}
//Timer0中断初始化
Void Timer0_Ini_inti()
{rTCFG1&=~(0xF<<20);
 rPRIORITY = 0x0000 007F;   // 使用默认的循环优先级
  rINTMOD = 0x0000 0000;   // 所有中断均为默认的IRQ中断
 rINTMSK&=~(1<<10);         //Timer0开中断
 pISR_Timer0= (unsigned)Timer0_ISR;
rSRCPND=1<<10;                  //SRCPND[10]=>Timer0,写入1，清除Timer0中断请求记录
 rINTPND=1<<10;                 //INTPND[10]=Timer0,写入1，清除Timer0中断服务记录
}
```

硬件应用题　16　05

```
//主函数
#include <stdio.h>
int main()
{
ProtF_Init();  //引脚初始化
Timer0_Init(); //Timer0初始化
Timer0_Int();     //timer0中断初始化
Timer0_start(); //启动定时器
While(1)  //循环，等待Timer0中断
{
 Dealy(1);
}
return 0;
}
```

NAND Flash和NOR Flash都是非易失性的存储设备，它们都使用浮动门晶体管来存储数据，但是它们在结构、性能和使用场景上有一些主要的区别：

1. **结构**：NAND Flash的结构更加紧凑，所以在同样的面积内，NAND Flash可以存储更多的数据，进而达到更高的存储密度和更低的每位成本。而NOR Flash则有更大的单元面积，但每个单元可以独立寻址，因此NOR Flash在读取时具有更低的延迟。

2. **读写性能**：NOR Flash提供了快速的随机读取能力，适合用作程序/代码存储（例如在嵌入式系统中）。NAND Flash则优化了顺序读取和写入的性能，适合用作大量数据的存储（例如在USB闪存盘和固态硬盘中）。

3. **耐用性**：NAND Flash具有更高的写入/擦除耐用性。NOR Flash在这方面表现较差，但由于其随机读取性能优秀，通常用于执行代码，这种用途对写入/擦除耐用性的要求不高。

4. **接口**：NOR Flash通常提供并行接口，而NAND Flash通常提供串行接口。并行接口具有更高的速度，但需要更多的I/O线，而串行接口则在速度和I/O线数量之间取得了平衡。

5. **价格**：由于结构上的差异，NAND Flash通常比NOR Flash更便宜，这也是为什么NAND Flash在大容量存储设备中更常见的原因。

总的来说，NAND Flash和NOR Flash各有其优势和适用场景，设计者需要根据具体的需求来选择使用哪种类型的Flash。

1. DMA请求方式与请求源

- DMA请求方式：2种。

  （1）硬件请求：H/W请求模式，由选择的DMA请求源触发DMA操作。

  （2）软件请求：S/W请求模式，通过软件设置请求DMA操作。

- DMA请求源：在H/W请求模式（硬件请求）有效，每个DMA通道有7个DMA请求源，可以选择其中的一个。

- 说明：虽然每个DMA通道有7个DMA请求源，但各不相同。

通道	请求源0	请求源1	请求源2	请求源3	请求源4	请求源5	请求源6
Ch-0	nXDREQ0	UART0	SDI	Timer	USB device EP1	I2SSDO	PCMIN
Ch-1	nXDREQ1	UART1	I2SSDI	SPI0	USB device EP2	PCMOUT	SDI
Ch-2	I2SSDO	I2SSDI	SDI	Timer	USB device EP3	PCMIN	MICIN
Ch-3	UART2	SDI	SPI1	Timer	USB device EP4	MICIN	PCMOUT

注：
nXDREQ0和nXDREQ1代表两个外部源（外部设备），IISSDO和IISSDI分别代表IIS的发送和接收。

DMA传输方向

- **DMA传输方向：** **4种。**
- **在“源←→目的”之间实现“高性能总线（AHB）←→外设总线（APB）”的传送，情况如下：**

  （1）源在高性能总线（AHB）→

  目的在高性能总线（AHB）

  （2）源在高性能总线（AHB）→

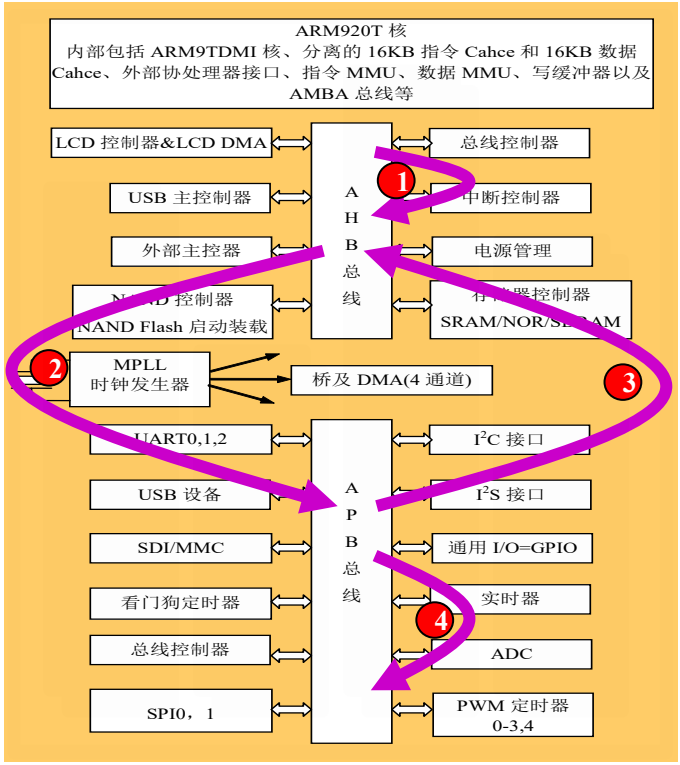
  目的在外设总线（APB）

  （3）源在外设总线（APB）→

  目的在高性能总线（AHB）

  （4）源在外设总线（APB）→

  目的在外设总线（APB）



2. DMA操作状态- FSM操作

- **S3C2440 DMA采用三态FSM（有限状态机，Finite State Machine）操作：**

  （1）状态1：等待DMA 请求（初始状态）。一旦有请求，则进入状态2。初始状态下，DMA ACK信号（DMA应答信号）和 INT REQ信号（中断请求信号）都为0。

  （2）状态 2：计数器加载。在此状态下，DMA ACK信号变成1，计数器（CURR_TC）从DCON控制寄存器的[19:0]位加载计数初值。

  （3）状态 3：数据传输。在此状态下，进行DMA基本操作。从源地址读取数据并写入目的地址。

- **DMA计数器：20位，减1型计数器，每次当前传输结束后计数器的值减1。计数器值减到0时， 表示DMA操作结束。**

3. DMA工作模式

- DMA工作模式：2种。

  （1）单服务模式：单数据传送，一次DMA请求完成一个数据单元的DMA传送。

  （2）全服务模式：连续数据传送，一次DMA请求完成一批数据单元的DMA传送。

- DMA传输数据单元：一次DMA操作传送的数据宽度，包括3种：字节、半字、字。
- DMA传输增量方式：一次DMA传送后，源、目的地址的变化情况，2种：
- （1）增加：在传输后地址增加（地址根据传输模式中数据大小的不同情况而增加）。
- （2）固定：在传输后地址不改变（突发模式中，地址只在突发传输期间增加，但在传输后又回到其第一个值）。
- 外部DMA请求/应答协议：3种。

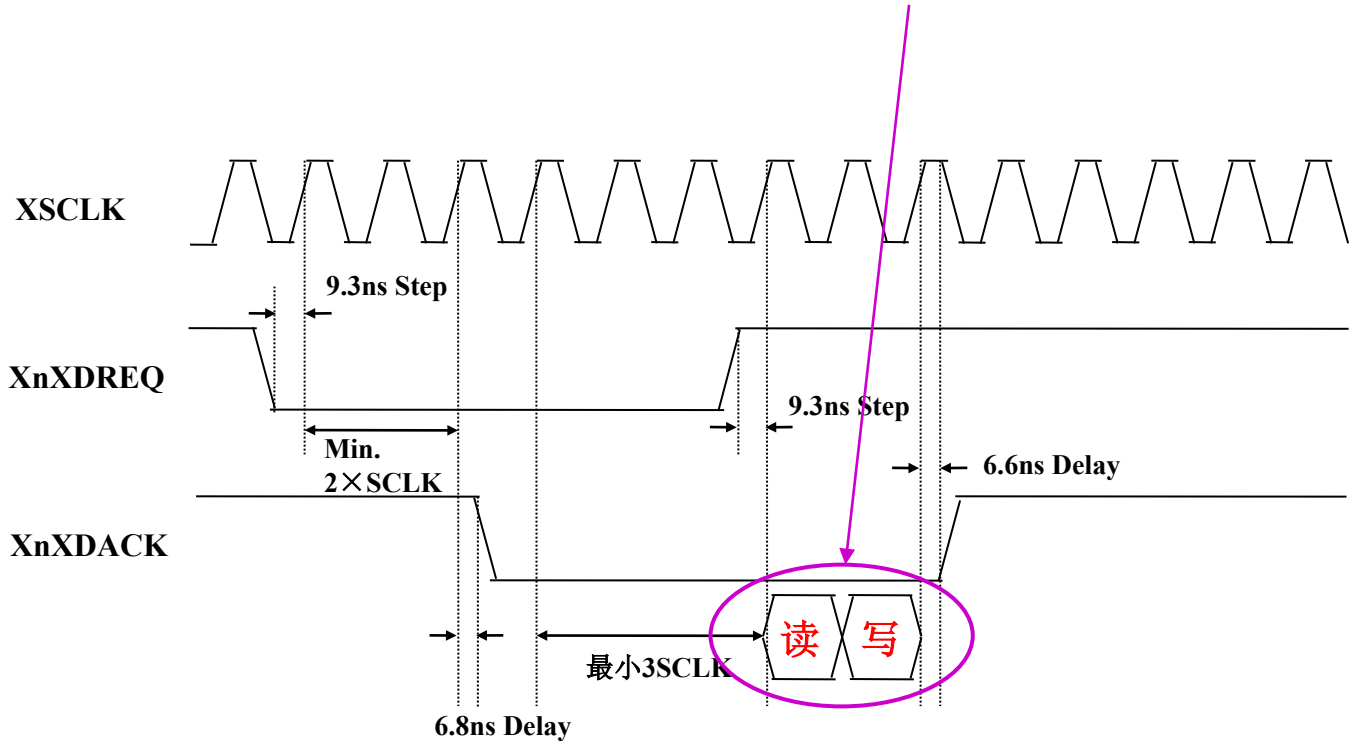
  （1）单服务请求模式。

  （2）单服务握手模式。

  （3）全服务握手模式。

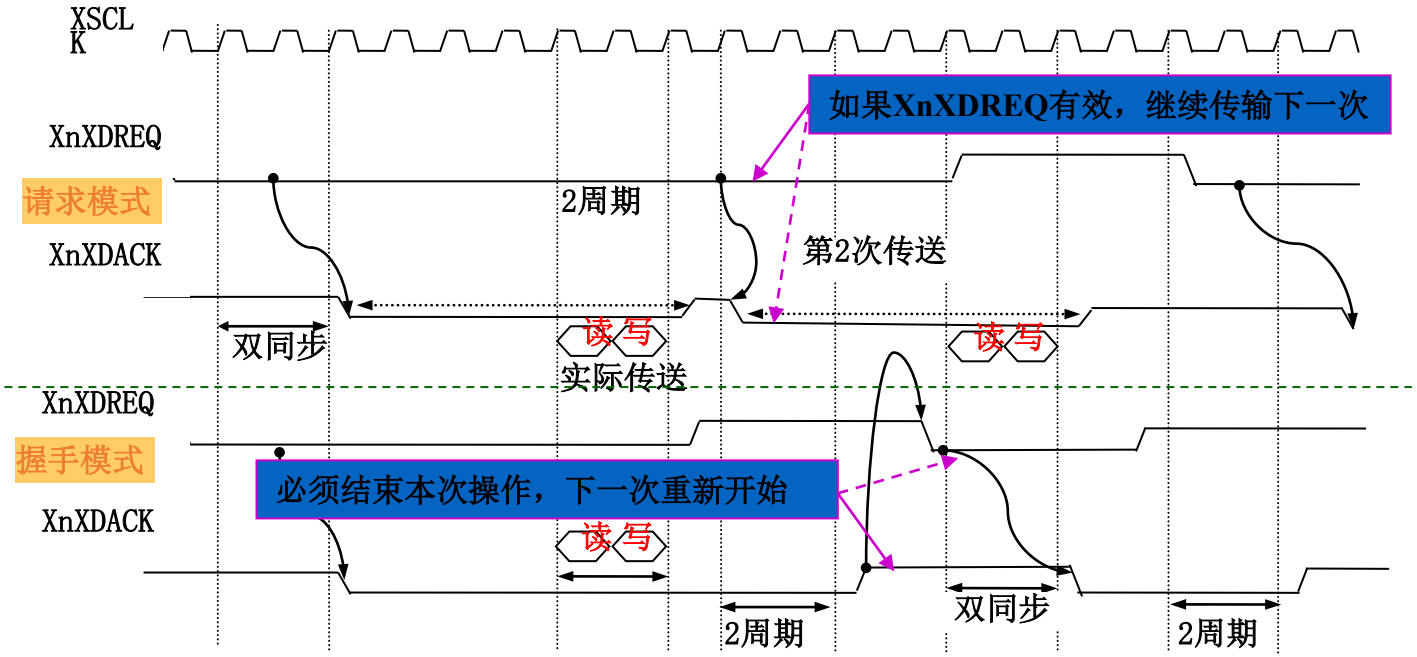
4. 基本的DMA时序

- 基本DMA传输操作：在DMA操作期间执行成对的读写周期。



5. DMA传输模式

- DMA传输模式：2种。
- （1）请求模式：也叫询问模式、查询模式， 一个DMA数据单元传输结束后，如果XnXDREQ信号保持有效，就立刻进行下一次的DMA传输。
- （2）握手模式：一个DMA数据单元传输结束后，结束本次DMA操作。若还想传输下一个数据，需要重新进行DMA请求。

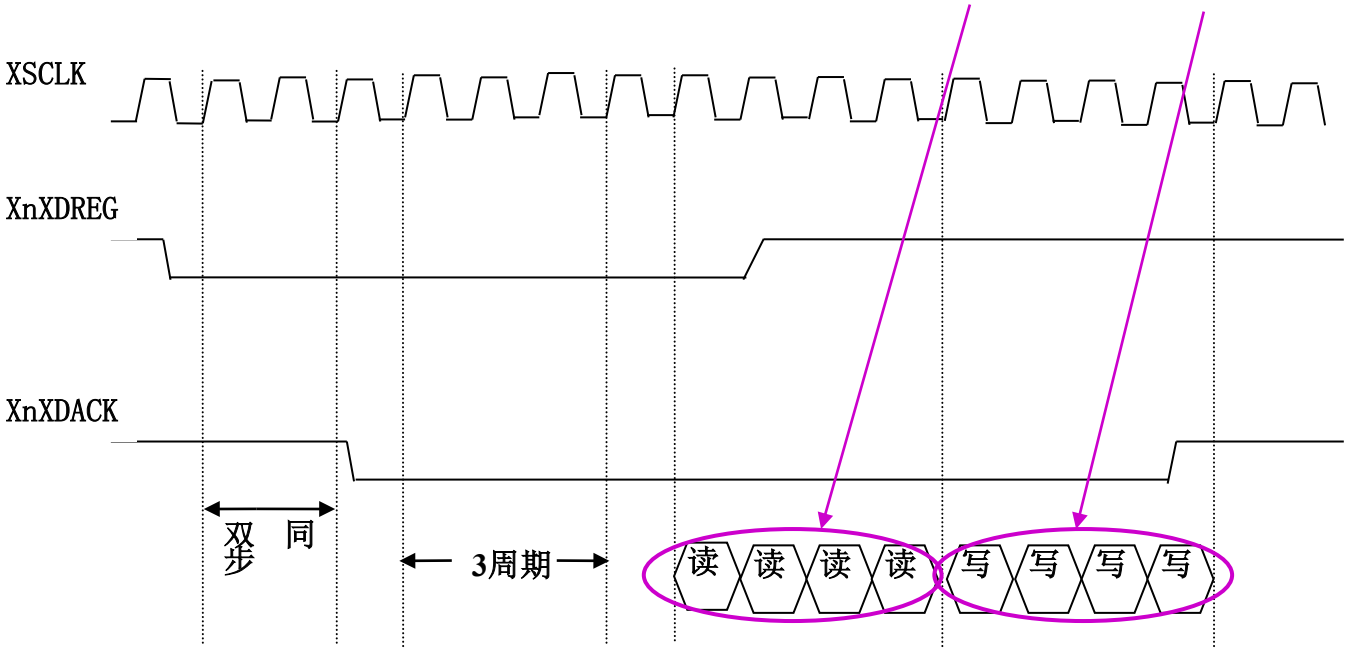


6．DMA基本传输模式

- **DMA基本传输模式：在一个基本DMA传输操作中，根据传输的数据大小，分2种。**

  （1）单次传输：一个基本的DMA传输操作完成一次读和一次写。

  （2）突发传输：一个基本的DMA传输操作完成四次连续读和四次连续写。



5.4.2 S3C2440芯片的DMA寄存器

- 每个DMA通道有9个寄存器：6个控制寄存器（控制DMA的传输），3个状态寄存器（反映DMA的状态）。
- S3C2440有4个DMA通道 。 4个DMA通道共计有36寄存器。
- 6个DMA控制传输的寄存器，可以进行读/写：
  - （1）DMA初始源寄存器DISRCn（DMA INITIAL SOURCE）
  - （2）DMA初始源控制寄存器DISRCCn（DMA INITIAL SOURCE CONTROL）
  - （3）DMA初始目的寄存器DIDSTn（DMA INITIAL DESTINATION）
  - （4）DMA初始目的控制寄存器DIDSTCn（DMA INITIAL DESTINATION CONTROL）
  - （5）DMA控制寄存器DCONn（DMA CONTROL）
  - （6）DMA屏蔽触发寄存器DMASKTRIGn（DMA MASK TRIGGER）
- 3个状态监视的寄存器，只能进行读：
  - （1）DMA状态寄存器DSTATn（DMA STATUS）
  - （2）DMA当前源寄存器DCSRCn（DMA CURRENT SOURCE）
  - （3）DMA当前目的寄存器DCDSTn（CURRENT DESTINATION）

1. 传输控制寄存器

- DMA传输控制寄存器：6个，主要用来控制DMA传输时的源地址起始值、目的地址起始值、数据块长度、DMA模式等信息。

- （1）DMA初始源寄存器
- DISRCn功能：存放要传输的源数据起始地址。

DISRC	位	描述	初始值
S_ADDR	[30:0]	用于传输的源数据基址（开始地址）。如果CURR_SRC为0 且DMA ACK为1，该位值将被装载到CURR_SRC。	0

寄 存 器	地 址	读/写	描 述	复 位 值
DISRC0	0x4B000000	读/写	DMA0初始源地址寄存器	0x00000000
DISRC1	0x4B000040		DMA1初始源地址寄存器	
DISRC2	0x4B000080		DMA2初始源地址寄存器	
DISRC3	0x4B0000C0		DMA3初始源地址寄存器	

（2）DMA初始源控制寄存器 DISRCCn

- **DISRCC功能：选择源数据在高性能总线（AHB）上，还是在外部总线（APB）上，以及源地址的增量方式。**

DISRCC	位	描述	初始值
LOC	[1]	位1用来选择DMA源的位置。 0=源在高性能总线(AHB)上；1=源在外设总线(APB)上。	0
INC	[0]	位0被用于选择地址增加。0=增加；1=固定。 如果为 0，每次传输以后，地址增加相应的数据大小； 如果为 1，每次传输以后，地址保持不变。	0

寄 存 器	地 址	读/写	描 述	复 位 值
DISRCC0	0x4B000004	读/写	DMA0初始源控制寄存器	0x00000000
DISRCC1	0x4B000044		DMA1初始源控制寄存器	
DISRCC2	0x4B000084		DMA2初始源控制寄存器	
DISRCC3	0x4B0000C4		DMA3初始源控制寄存器	

（3）DMA初始目的寄存器 DIDSTn

- **DIDSTn功能：** 存放要传输的**目的数据起始地址**。

寄 存 器	地 址	读/写	描 述	复 位 值
DIDST0	0x4B000008	读/写	DMA0初始目标地址寄存器	0x00000000
DIDST1	0x4B000048		DMA1初始目标地址寄存器	
DIDST2	0x4B000088		DMA2初始目标地址寄存器	
DIDST3	0x4B0000C8		DMA3初始目标地址寄存器	

DIDST	位	描述	初始值
D_ADDR	[30:0]	要传输的目的数据基本地址（开始地址）。当且仅当CURR_DST为0并且DMA ACK为1时将此位的值锁存到CURR_DST中	0

（4）DMA初始目的控制寄存器 DIDSTCn

- **DIDSTC功能：选择目的数据在高性能总线（AHB）上，还是在外部总线（APB）上，以及源地址的增量方式。**

寄 存 器	地 址	读/写	描 述	复 位 值
DIDSTC0	0x4B00000C	读/写	DMA0初始目标控制寄存器	0x00000000
DIDSTC1	0x4B00004C		DMA1初始目标控制寄存器	
DIDSTC2	0x4B00008C		DMA2初始目标控制寄存器	
DIDSTC3	0x4B0000CC		DMA3初始目标控制寄存器	

DIDSTCn寄存器格式

DIDSTC	位	描述	初始值
CHK_INT	[2]	当自动重载被设置，选择中断出现时间。 0=若TC为0，中断出现；1=自动重载被执行后，中断出现。	0
LOC	[1]	用于选择DMA目的的位置。 0=DMA目的在高性能总线(AHB)上。 1=DMA目的在外设总线(APB)上。	0
INC	[0]	用于选择目的地址是否增加。0=增加；1=固定。 如果该位为0，每次传输以后，地址增加1（依据数据宽度）； 如果该位为1，每次传输以后，地址保持不变。	0

（5）DMA控制寄存器 DCONn

- DCONn功能：选择DMA传输模式（请求模式、握手模式）、DMA基本传输模式（单次传输、突发传输）、DMA工作模式（单服务模式、全服务模式）、DMA通道请求源、DMA请求方式（硬件请求、软件请求）、DMA传输数据单元（字节、半字、字），设置DMA计数值。

寄 存 器	地 址	读/写	描 述	复 位 值
DCON0	0x4B000010	读/写	DMA0控制寄存器	0x00000000
DCON1	0x4B000050		DMA1控制寄存器	
DCON2	0x4B000090		DMA2控制寄存器	
DCON3	0x4B0000D0		DMA3控制寄存器	

DCONn寄存器格式 1

DCON	位	描述	初始值
DMD_HS	[31]	用来选择DMA传输模式。 0=选择请求模式；1=选择握手模式。	0
SYNC	[30]	用来选择 DREQ/DACK同步信号。 0= DREQ / DACK与PCLK (APB时钟)同步。 1= DREQ /DACK与HCLK (AHB时钟)同步。	0
INT	[29]	用来使能/不使能CURR_TC产生中断。 0=不使能；1=使能。	0
TSZ	[28]	用来选择基本DMA传输的大小。 0=执行单元传输；1=执行阵发长度为4的DMA传输。	0
SERVMODE	[27]	用来选择服务模式。 0=单独服务模式；1=整体服务模式。	0
HWSRCSEL	[26:24]	各DMA通道请求源选择位。 通道0：000：nXDREQ0；001：UART0；010：SDI；011：Timer 100：USB device EP1；101：IISSDO；110：PCMIN 通道1：000：nXDREQ1；001：UART1；010：IISSDI；011：SPI 100：USB device EP2；101：PCMOUT；110：SDI 通道2：000：IISSDO；001：IISSDI；010：SDI；011：Timer 100：USB device EP3；101：PCMIN；110：MICIN 通道3：000：UART2；001：SDI；010：SPI；011：Timer 100：USB device EP4；101：MICIN；110：PCMOUT	000

DCONn寄存器格式 2

SWHW_SEL	[23]	用来选择S/W还是H/W模式。 0=S/W模式；1=H/W模式。	0
RELOAD	[22]	设定重装载开关选项 0=当前DMA传输完后，终点计数器自动重载； 1=当前DMA传输完后，DMA通道被关闭。	0
DSZ	[21:20]	传输数据大小单位。 00 =字节；01 =半字；10=字； 11 =保留。	00
TC	[19:0]	初始DMA传输计数值。 传输的实际字节数由以下公式计算：DSZ x TSZ x TC。	0x00000

（6）DMA 屏蔽触发寄存器 DMASKTRIGn

- **DMASKTRIGn功能：设置DMA通道的开启与关闭，设置软件DMA请求。**

寄 存 器	地 址	读/写	描 述	复 位 值
DMASKTRIG0	0x4B000020	读/写	DMA0屏蔽触发寄存器	0x00000000
DMASKTRIG1	0x4B000060		DMA1屏蔽触发寄存器	
DMASKTRIG2	0x4B0000A0		DMA2屏蔽触发寄存器	
DMASKTRIG3	0x4B0000E0		DMA3屏蔽触发寄存器	

DMASKTRIG	位	描述	初始值
STOP	[2]	用来停止DMA操作。 0=正常；1=当一个基本操作完成后立即停止DMA操作。	0
ON_OFF	[1]	DMA通道开关位。 0= 关闭DMA通道；1= 开启DMA通道。	0
SW_TRIG	[0]	用来在软件模式下触发DMA通道。 0=不触发；1=触发DMA操作。	0

2. 状态寄存器

- 状态寄存器：记录DMA传输的状态，有3个寄存器，可以通过这些寄存器来了解DMA传输时的信息，以便于进行控制。

- （1）DMA状态寄存器

- DMA状态寄存器（DSTATn）共有4个：DSTAT0、DSTAT1、DSTAT2、DSTAT3，分别对应4个独立的DMA通道。
- 功能：DMA就绪、忙状态，提供计数器值。
- 这4个寄存器地址分别为0x4B000014、0x4B000054、0x4B000094、0x4B0000D4，复位后的初值为0x00000000，且都是可读可写的。

DSTAT	位	描述	初始值
STAT	[21:20]	DMA控制器的状态。 00=指示DMA控制器准备好接收下一个DMA请求； 01=指示DMA控制器忙。	0
CURR_TC	[19:0]	当前DMA计数器的值。 注：在每一次当前传输结束后该值减1。	0x00000

（3）DMA当前目的寄存器DCDSTn

- DMA当前目的寄存器（DCDSTn）共有4个：DCDST0、DCDST1、DCDST2、DCDST3，分别对应4个独立的DMA通道。
- 功能：提供当前DMA通道的目的地址值。
- 这4个寄存器地址分别为0x4B00001C、0x4B00005C、0x4B00009C、0x4B0000DC，复位后的初值为0x00000000，且都是可读可写的。

DCDST	位	描述	初始值
CURR_DST	[30:0]	当前DMA通道的目的地址值。	0x00000000