9. 基于Cortex-A9 LED汇编、C语言驱动编写

原创 土豆居士 一口Linux 2020-12-22 11:45

收录于合集
#从0学arm 27 #所有原创 206

ARM系列文章合集如下：

《从0学arm合集》

0. 前言

一般我们购买一个开发板，厂家都会给出对应的电路图文件，我们可以通过搜索对应名称来查找到对应的外设。对于驱动工程师来说，我们只需要知道外设与SOC交互的一些数据线和信号线即可。

用主控芯片控制这些外设的一般步骤：

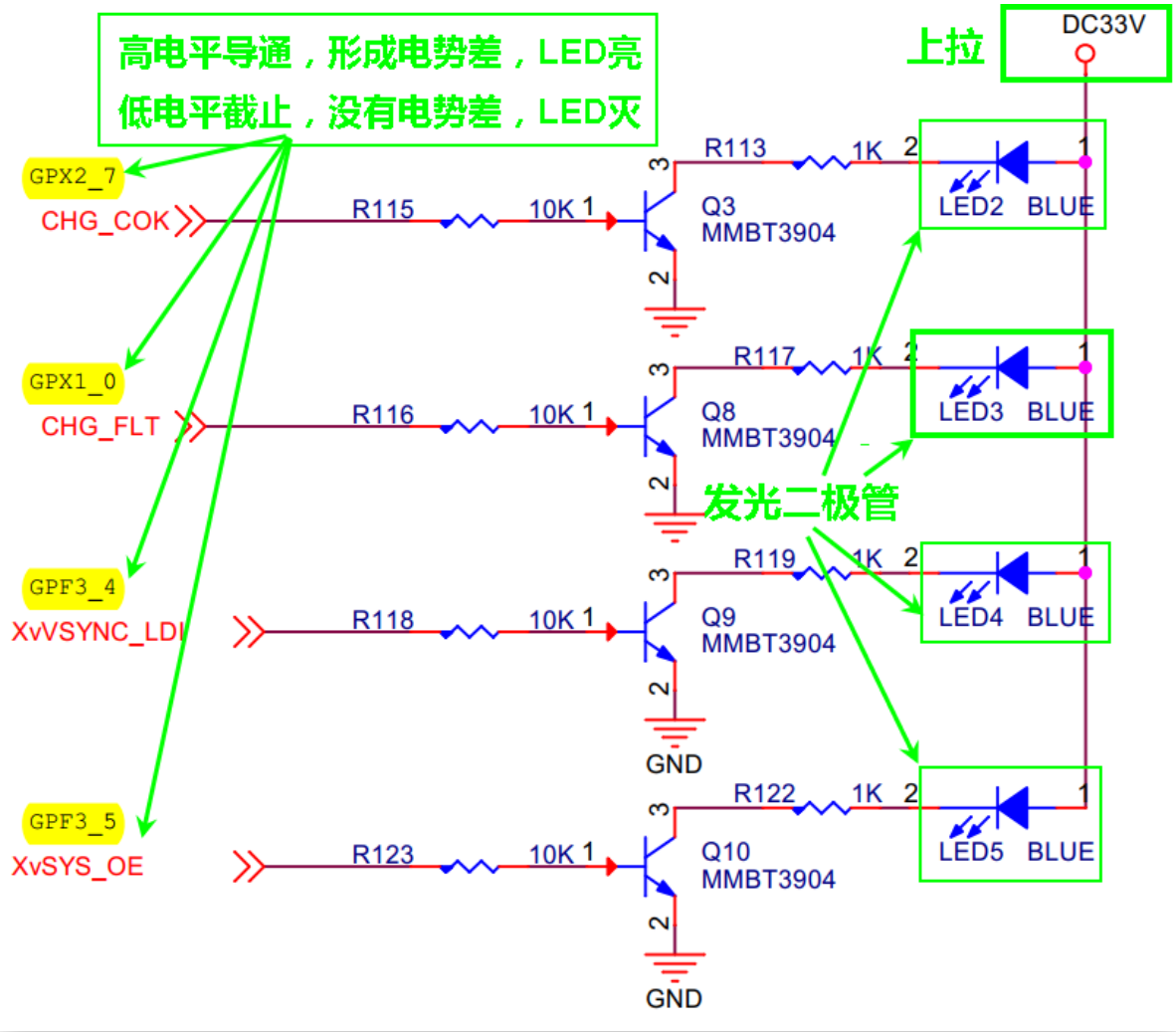
1. 看电路原理图，弄明白主控芯片和外设是怎么连接的，对于驱动工程师来说，主要是看外设的一些clk、数据引脚、控制引脚是如何连接的；
2. 外设一般都会连接到SOC的1个或者多个控制器上，比如i2c、spi、gpio等，有的是数据线有的是信号线，中断线等；
3. 根据电路连接和需求对主控芯片进行设置，往往对外设的设置都是通过寄存器操作实现；
4. 书写相应代码，实现功能，不同类型的外设，代码结构也不尽相同，比如按键，我们既可以通过轮询方式读取按键信息，也可以通过中断方式来读取。

下面我们就以华清远见的fs4412开发板为例来看如何编写led的裸机程序。SOC exynos 4412 datahseet 下载地址：

https://download.csdn.net/download/daocaokafei/12533438

一、LED灯电路图

首先看下led电路图：

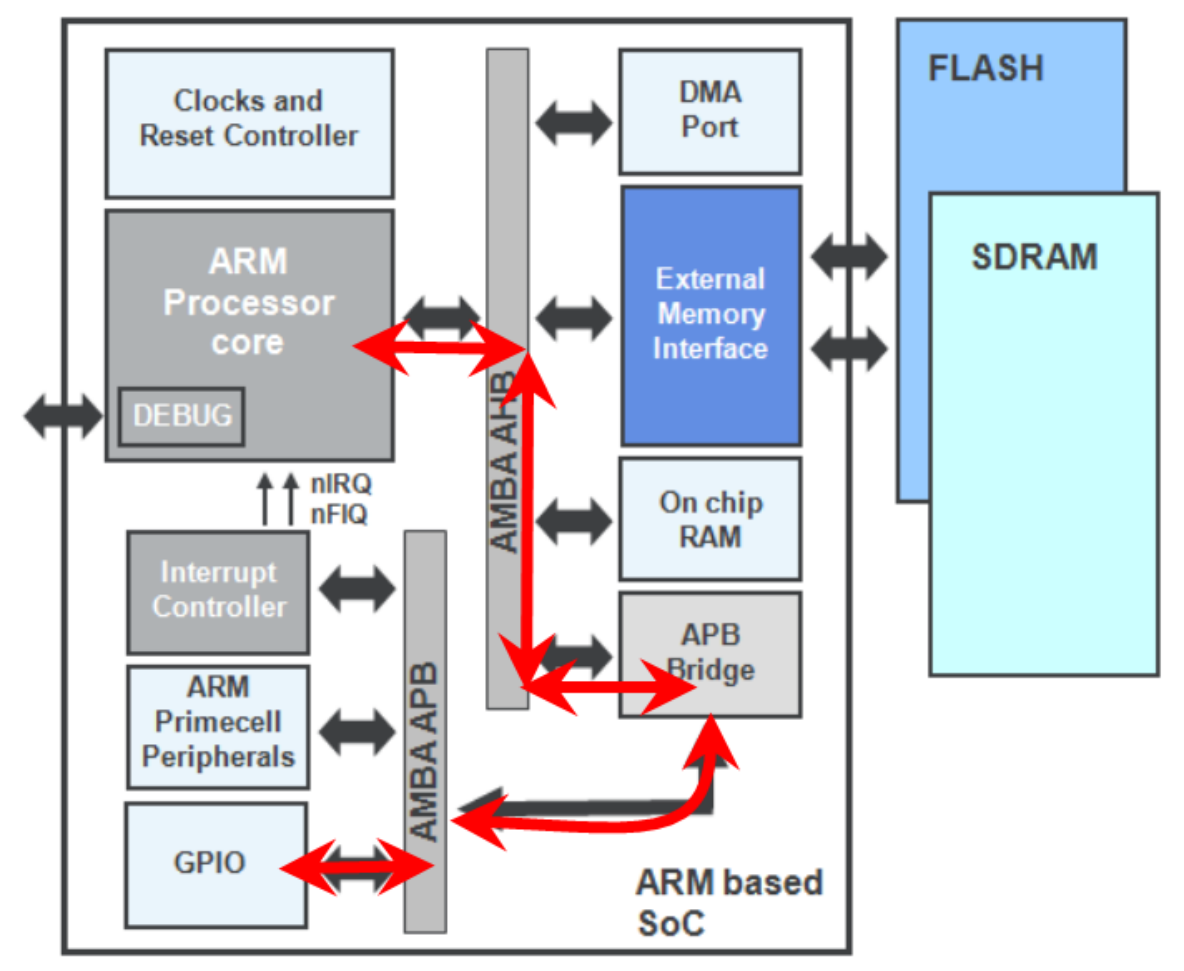


LED电路图

1. 该板子有4个LED，是发光二极管，有电流是为蓝色；
2. led都接了上拉电阻；
3. 三极管的基极接了SOC的某个GPIO引脚；
4. 比如GPX1_0,当该引脚为高电平是，三极管pn结导通，于是LED3两侧就有了电势差，LED3被点亮，如果该引脚为低电平，pn结截止，LED3两侧就没有了电势差，LED3熄灭。

下面是CPU核访问GPIO控制器的数据通路：

1. AHB：高速总线
2. APB Bridge:APB总线桥
3. APB：外设总线，低速总线
4. GPIO挂载在APB总线上



GPIO 与 SOC

由上图可知，cpu要访问GPIO的寄存器需要经过的路径。

二、GPIO

GPIO（General Purpose I/O Ports）意思为通用输入/输出端口，通俗地说，就是一些引脚，可以通过它们输出高低电平或者通过它们读入引脚的状态–是高电平或是低电平。

用户可以通过GPIO口和硬件进行数据交互(如UART)，控制硬件工作(如LED、蜂鸣器等),读取硬件的工作状态信号（如中断信号）等。GPIO口的使用非常广泛。

1. GPIO的优点

- 低功耗：GPIO具有更低的功率损耗(大约1μA，μC的工作电流则为100μA)。
- 集成I^2C从机接口：GPIO内置I^2C从机接口，即使在待机模式下也能够全速工作。
- 小封装：GPIO器件提供最小的封装尺寸—3mm x 3mm QFN!
- 低成本：您不用为没有使用的功能买单！
- 快速上市：不需要编写额外的代码、文档，不需要任何维护工作！

- 灵活的灯光控制：内置多路高分辨率的PWM输出。
- 可预先确定响应时间：缩短或确定外部事件与中断之间的响应时间。
- 更好的灯光效果：匹配的电流输出确保均匀的显示亮度。
- 布线简单：仅需使用2条I^2C总线或3条SPI总线。

2. exynos4412 GPIO特性

1. 172 个外部中断
2. 32个外部可唤醒中断
3. 252个多功能 input/output ports
4. 在休眠模式下也可以控制GPIO引脚，但不包括 GPX0, GPX1, GPX2, and GPX3

3. 6 General Purpose Input/Output (GPIO) Control

Exynos 4412 SCP 包括304个多功能 input/output端口引脚和164 存储端口引脚. 总共 37 个端口分组和两个存储端口分组.。

下图为GPIO模块图：

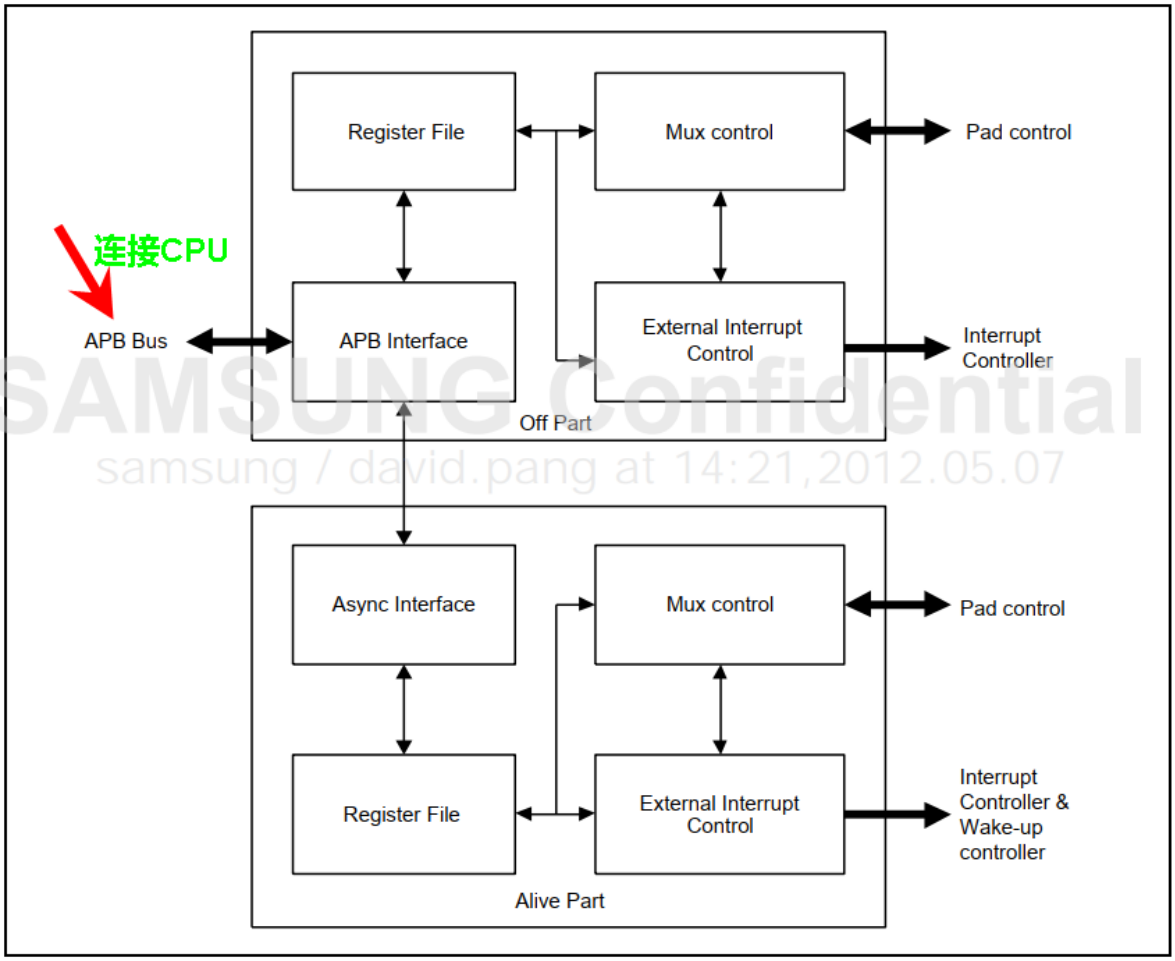


Figure 6-1 GPIO Block Diagram

三、如何操作GPIO?

主要通过寄存器来操作GPIO引脚。

GPxCON用于选择引脚功能，GPxDAT用于读/写引脚数据；另外，GPxUP用于确定是否使用内部上拉电阻。其中x为A、B.....H、J等。

1. GPxCON寄存器

从寄存器的名字可以看出，它用于配置（Configure）–选择引脚功能。

LED3是连接到GPX1_0,该引脚说明如下：

6.2.3.198 GPX1CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
GPX1CON[0]	[3:0]	RW	0x0 = Input 0x1 = Output 0x2 = Reserved 0x3 = KP_COL[0] 0x4 = Reserved 0x5 = ALV_DBG[4] 0x6 to 0xE = Reserved 0xF = WAKEUP_INT1[0]	0x00

GPX1CON

由上图所示，

1. GPX1CON地址为0x1100C20;
2. LED3是输出设备，所以需要将GPX1CON[3:0]设置为0x1，但是能修改其他的bite。

2. GPxDAT寄存器

GPxDAT用于读/写引脚；当引脚被设为输入时，读此寄存器可知相应引脚的电平状态是高还是低；当引脚被设为输出时，写此寄存器相应位可以令此引脚输出高电平或是低电平。

6.2.3.199 GPX1DAT

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C24, Reset Value = 0x00

Name	Bit	Type	Description	Reset Value
GPX1DAT[7:0]	[7:0]	RWX	When you configure port as input port then corresponding bit is pin state. When configuring as output port then pin state should be same as corresponding bit. When the port is configured as functional pin, the undefined value will be read.	0x00

GPX1DAT

1. GPX1DAT的地址是0x1100C24
2. LED3对应的输出引脚是GPX1DAT[0]，点灯只需要将该引脚设置为1即可，灭灯将bite0置0。

3. GPxUP寄存器

GPxUP：某位为1时，相应引脚无内部上拉电阻；为0时，相应引脚使用内部上拉电阻。

上拉电阻的作用在于：当GPIO引脚处于第三态（即不是输出高电平，也不是输出低电平，而是呈高阻态，即相当于没接芯片）时，它的电平状态由上拉电阻、下拉电阻确定。

本例不用设置。

四、驱动编写

下面我们分别用汇编和C语言来给LED编写驱动程序。

1. 汇编代码

大家如果掌握了我之前讲解的汇编指令的知识点，那么这个代码很容易就能看明白：

```
.globl _start
.arm
_start:
 LDR R0,=0x11000C20 @将配置寄存器GPX1CON的地址写入到R0
 LDR R1,[R0]  @读取寄存器GPX1CON的值保存到R1
 BIC R1,R1,#0x0000000f @将R1的3:0位清0，目的是不覆盖到其他bit的值
 ORR R1,R1,#0x00000001 @将R1的3:0位置1
 STR R1,[R0]  @将R1的值写回寄存器GPX1CON
loop:
 LDR R0,=0x11000C24 @将data寄存器GPX1DAT的地址写入到R0
```

```
 LDR R1,[R0] @读取寄存器GPX1DAT的值保存到R1
 ORR R1,R1,#0x01 @将R1的值bite0 设置为1，即拉高，点灯
 STR R1,[R0]  @将R1的值写回寄存器GPX1DAT
 BL delay  @调用延时函数
 LDR R1,[R0]
 BIC R1,R1,#0x01 @将R1的值bite0 设置为0，即拉低，灭灯
 STR R1,[R0]
 BL delay
 B loop
delay:     @delay延时函数
 LDR R2,=0xffffffff
loop1:
 SUB R2,R2,#0x1
 CMP R2,#0x0
 BNE loop1
 MOV PC,LR @返回
.end
```

Makefile

```
TARGET=gcd
all:
 arm-none-linux-gnueabi-gcc -O0 -g -c -o $(TARGET).o $(TARGET).s
 arm-none-linux-gnueabi-ld   $(TARGET).o -Ttext 0x40008000 -N -o $(TARGET).elf
 arm-none-linux-gnueabi-objcopy -O binary -S $(TARGET).elf $(TARGET).bin
clean:
 rm -rf *.o *.elf *.dis *.bin
```

程序功能很简单，就是让LED3呈现一闪一闪的效果。

执行make，最终生成的gcd.bin文件。

2. c语言实现

如果要进入C语言执行环境，那么就必须为设置栈空间，函数调用参数和返回值会压栈。

start.s

```
.text
.global _start
_start:
```

```
  ldr  sp,=0x70000000         /*get stack top pointer*/
  b  main
```

main.c

```
/* GPX1 */
typedef struct {
    unsigned int CON;
    unsigned int DAT;
    unsigned int PUD;
    unsigned int DRV;
}gpx1;
#define GPX1 (* (volatile gpx1 *)0x11000C20 )

void led_init(void)
{
 GPX1.CON = GPX1.CON & (~(0x0000000f)) | 0x00000001;
}
void led_on(int n)
{
 GPX1.DAT = GPX1.DAT|0x01;
}
void led_off()
{
 GPX1.DAT = GPX1.DAT&(~(0x01));
}
void  delay_ms(unsigned int num)
{   int i,j;
     for(i=num; i>0;i--)
      for(j=1000;j>0;j--)
         ;
 }
int main(void)
{
 led_init ();
 while (1) {
  led_on();
  delay_ms(500);
  led_off();
  delay_ms(500);
 }
 while(1);
```

```
    return 0;
}
```

map.lds

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x40008000;      ;从该地址开始
 . = ALIGN(4);
 .text      :       ;指定代码段
 {
  gcd.o(.text)  ;代码的第一个部分，绝对不能错
  *(.text)
 }
 . = ALIGN(4);
    .rodata :              ;只读数据段
 { *(.rodata) }
    . = ALIGN(4);
    .data :              ;读写数据段
 { *(.data) }
    . = ALIGN(4);
    .bss :
     { *(.bss) }
}
```

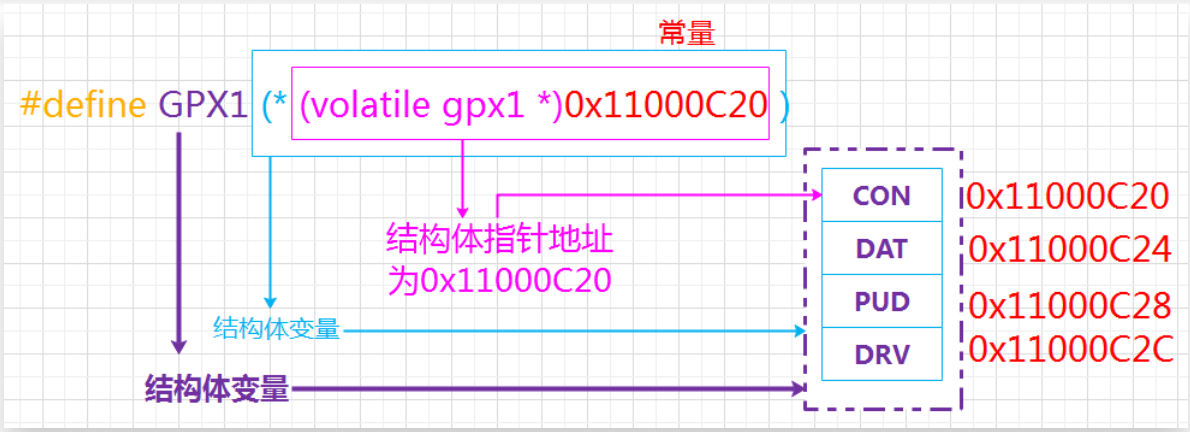
Makefile

```
TARGET=gcd
TARGETC=main
all:
 arm-none-eabi-gcc -O0 -g -c -o $(TARGETC).o  $(TARGETC).c
 arm-none-eabi-gcc -O0 -g -c -o $(TARGET).o $(TARGET).s
 arm-none-eabi-gcc -O0 -g -S -o $(TARGETC).s  $(TARGETC).c
 arm-none-eabi-ld $(TARGETC).o $(TARGET).o -Tmap.lds -o  $(TARGET).elf
 arm-none-eabi-objcopy -O binary -S $(TARGET).elf $(TARGET).bin
clean:
 rm -rf *.o *.elf *.dis *.bin
```

执行make命令，最终生成的gcd.bin文件。

这段代码中，读者可能不能理解的是下面的定义：

```
typedef struct {
    unsigned int CON;
    unsigned int DAT;
    unsigned int PUD;
    unsigned int DRV;
}gpx1;
#define GPX1 (* (volatile gpx1 *)0x11000C20 )
```



GPX1宏定义

由上图所示：

1. (volatile gpx1 *)0x11000C20)：将常量0x11000C20 强转成struct gpx1类型指针
2. (* (volatile gpx1 *)0x11000C20)：查找指针对应的内存驱动，即对应整个结构体变量，结构体变量地址为0x11000C20
3. #define GPX1 (* (volatile gpx1 *)0x11000C20)：GPX1等价于地址为0x11000C20的结构体变量

这样我们要想操作GPX1的寄存器，就可以像结构体变量一样操作即可。

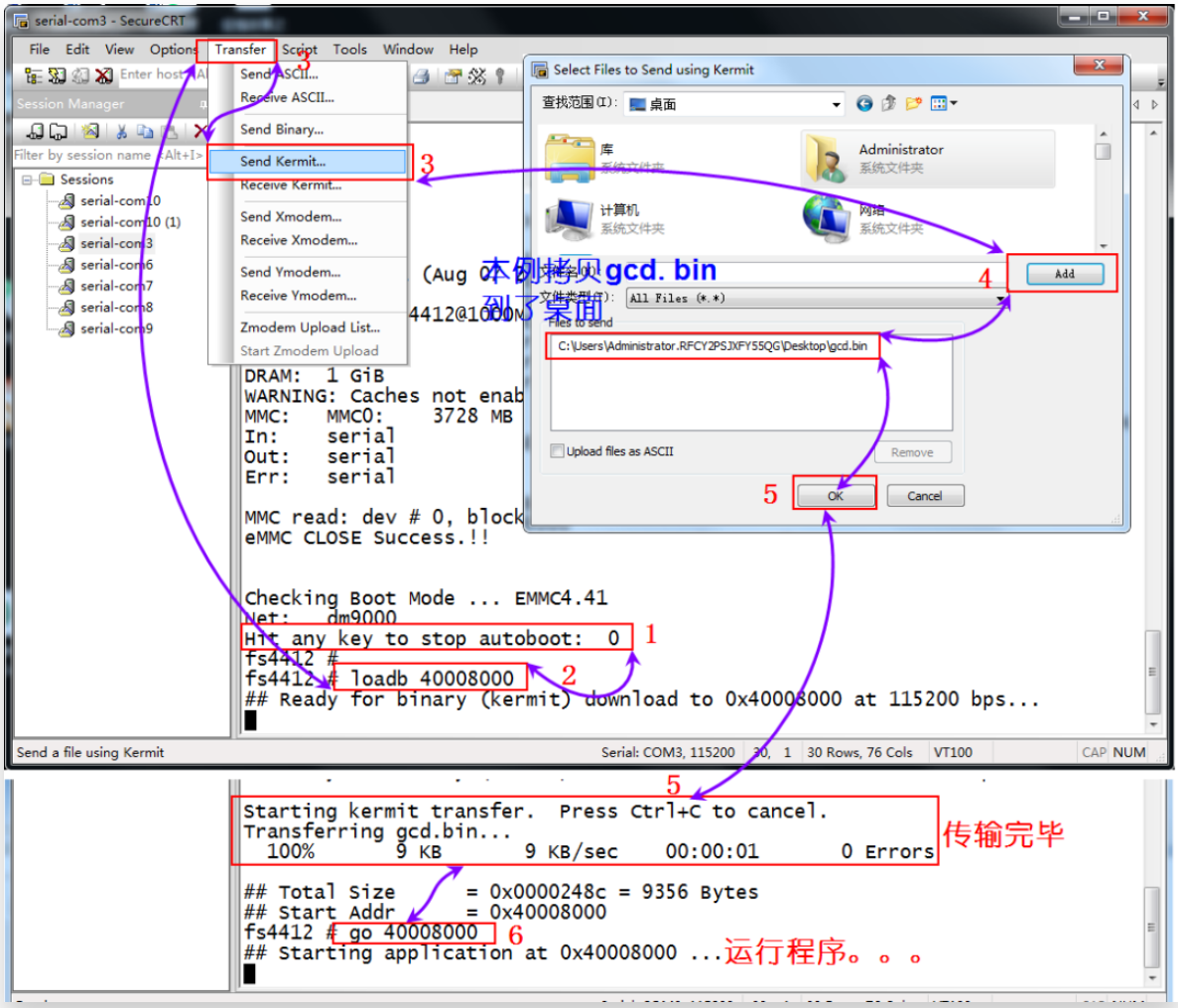
3. 测试

采用UBOOT自带的命令loadb,通过串口以baud速率下载binary(.bin)至SDRAM中某一地址中，然后用go 命令从某地址处开始执行程序。

该命令使用了kermit protocol，嵌入式系统通常使用该协议与pc传送文件。

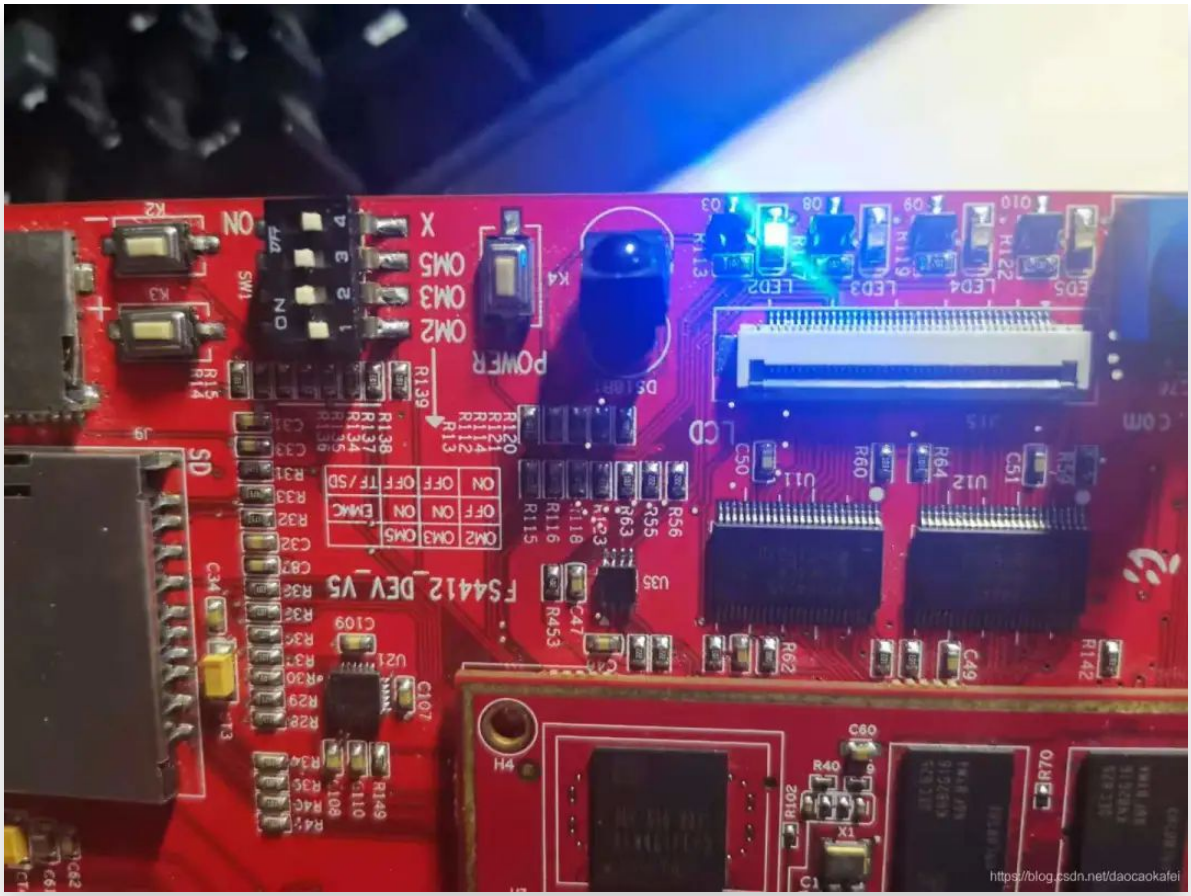
操作步骤如下：

1. 串口连接开发板，开发板启动后在读秒阶段，立即按下回车，进入uboot命令界面
2. 执行loadb 40008000 【该地址与Makefile 和map.lds文件中的地址保持一致】
3. 选择菜单transfer–>send Kermit,
4. 然后选择我们编译好的gcd.bin文件,
5. 点击OK，出现”Staring kermit transfer.”字样,
6. 执行 go 40008000，运行程序



运行裸机程序

执行结果：



led

可以看到LED闪烁的现象。

5. 注意

该种测试方法需要bootloader选用uboot，并且需要串口工具支持Kermit协议，一口君使用的是SecureCRT7.3.3版本【其他低一些的版本可能不支持该协议】，该软件的下载和安装方法【安装方法有点繁琐】可以公众号后台回复【SecureCRT】。

SecureCRT版本

• END •

其他网友提问汇总

1. 两个线程，两个互斥锁，怎么形成一个死循环?
2. 一个端口号可以同时被两个进程绑定吗?
3. 一个多线程的简单例子让你看清线程调度的随机性
4. 粉丝提问|c语言：如何定义一个和库函数名一样的函数，并在函数中调用该库函数
5. [网友问答5]i2c的设备树和驱动是如何匹配以及何时调用probe的?
6. [粉丝问答6]子进程进程的父进程关系
7. 【粉丝问答7】局域网内终端是如何访问外网？答案在最后

推荐阅读

【1】嵌入式工程师到底要不要学习ARM汇编指令？**必读**

【2】Modbus协议概念最详细介绍**必读**

【3】嵌入式工程师到底要不要学习ARM汇编指令？

【4】【从0学ARM】你不了解的ARM处理异常之道

【5】4. 从0开始学ARM-ARM汇编指令其实很简单

【6】为什么使用结构体效率比较高？ **必读**

进群，请加一口君个人微信，带你嵌入式入门进阶。

收录于合集 #从0学arm 27

上一篇
散装 vs 批发谁效率高？变量访问被ARM架构安排的明明白白

下一篇
10. 基于Cortex-A9的pwm详解

喜欢此内容的人还喜欢

面试连环问--操作系统
阿Q正砖

pinia
睡不着所以学编程

Konva实现图片自适应裁剪

A逐梦博客