# 编译链接原理与芯片启动学习

何思虎、康全杰、彭鑫

2021/8/2

目录

[编译链接原理与芯片启动学习 1](#_Toc78808735)

[围绕问题：C代码如何在芯片上运行？ 2](#_Toc78808736)

[第一部分： 2](#_Toc78808737)

[编译、预编译、汇编 2](#_Toc78808738)

[交叉编译 3](#_Toc78808739)

[链接 4](#_Toc78808740)

[.text段、.data段、.bss段 4](#_Toc78808741)

[链接脚本（布局、参数） 5](#_Toc78808742)

[Makefile 7](#_Toc78808743)

[gcc编译参数： 7](#_Toc78808744)

[ld链接参数： 8](#_Toc78808745)

[第二部分： 8](#_Toc78808746)

[一、启动初始化 start.c 8](#_Toc78808747)

[1、栈顶指针及栈大小分配 8](#_Toc78808748)

[2、中断向量表初始化 9](#_Toc78808749)

[3、复位中断服务函数 9](#_Toc78808750)

[二、主程序实现 main.c 10](#_Toc78808751)

[1、查原理图 10](#_Toc78808752)

[2、找到绝对地址 10](#_Toc78808753)

[3、时钟使能 11](#_Toc78808754)

[4、操作寄存器输出 11](#_Toc78808755)

# 围绕问题：C代码如何在芯片上运行？

应当包含两个部分：

1. 将代码转换为嵌入式芯片能识别的机器语言（编译链接过程）
2. 芯片启动

# 第一部分：

## 编译、预编译、汇编

C代码转换成芯片能识别的机器语言通常包含编译、预编译、汇编及链接四个步骤。使用GCC编译工具可以根据自身需要在编译的任一阶段结束，能够更好地控制整个编译过程。其中，编译、预编译、汇编过程及gcc指令如下：

(1)预编译 gcc -E hello.c -o hello.i .c >>> .i

将宏定义#define全部替换掉

条件编译 #if

将包含的文件插入指定位置 #include

删除所有注释

添加行号，方便后面调试

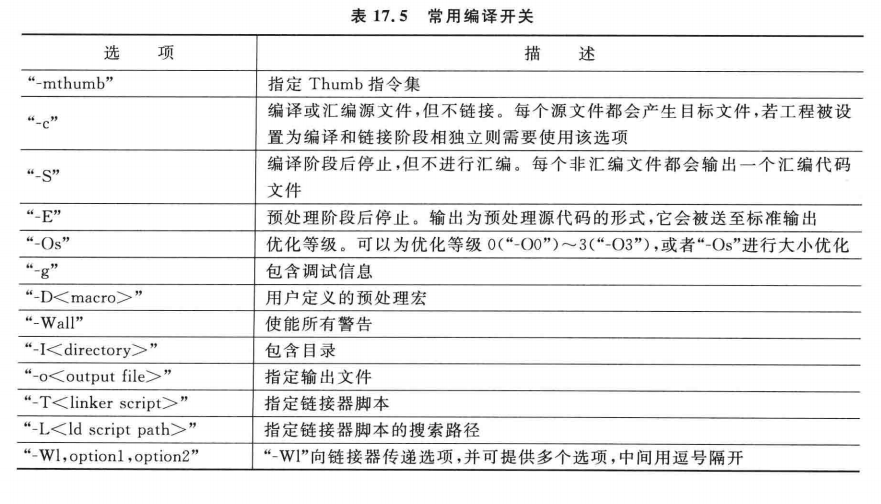
(2)编译 gcc -S hello.i -o hello.s .i >>> .s 汇编语言

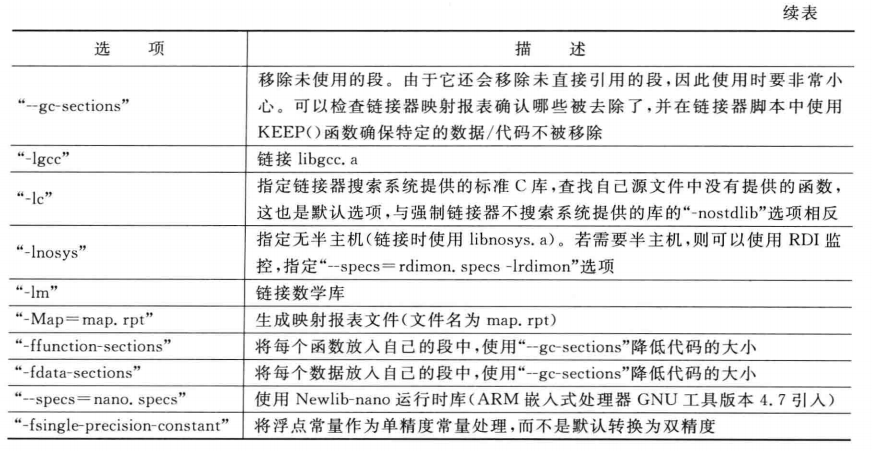
词法分析、语法分析、语义分析、优化

(3)汇编（目标文件）gcc -c hello.s -o hello.o .s >>> .o 机器语言

将汇编代码转化为机器可以执行的指令，将每一条汇编语句按对照表翻译成相应的机器指令就行。

其他常用gcc指令还有：





## 交叉编译

使用交叉编译器的原因在于：

(1)性能充足：目标平台的运行速度往往比主机慢得多，许多专用的嵌入式硬件被设计为低成本和低功耗，没有太高的性能。

(2)资源丰富：整个编译过程是非常消耗资源的，嵌入式系统往往没有足够的内存或磁盘空间。

(3)统一平台：即使目标平台资源很充足，可以本地编译，但是第一个在目标平台上运行的本地编译器总需要通过交叉编译获得。

(4)移植便捷：一个完整的Linux编译环境需要很多支持包，交叉编译使我们不需要花时间将各种支持包移植到目标板上。

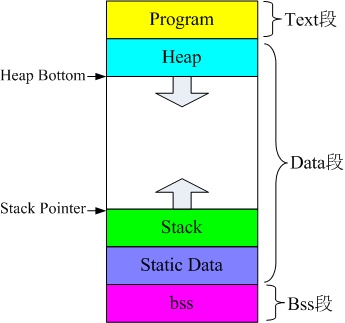
|  |  |  |
| --- | --- | --- |
| 工具 | 通用命令 | ARM嵌入式处理器GNU工具命令 |
| C编译器 | gcc | arm-none-eabi-gcc |
| 汇编器 | as | arm-none-eabi-as |
| 链接器 | ld | arm-none-eabi-ld |
| 二进制文件生成 | objcopy | arm-none-eabi-objcopy |
| 反汇编器 | objdump | arm-none-eabi-objdump |

## 链接

### .text段、.data段、.bss段

编译器编译源代码生成目标文件后，目标文件包含的是编译后的机器指令代码，这些机器指令代码包含着多种信息，比如说符号表，字符串等。因此，根据不同的属性，将目标文件的机器指令代码以“段”的形式存储，有时候也可称“节”。

最基本的就是.text段(代码段)、.data段(数据段)、.bss段。

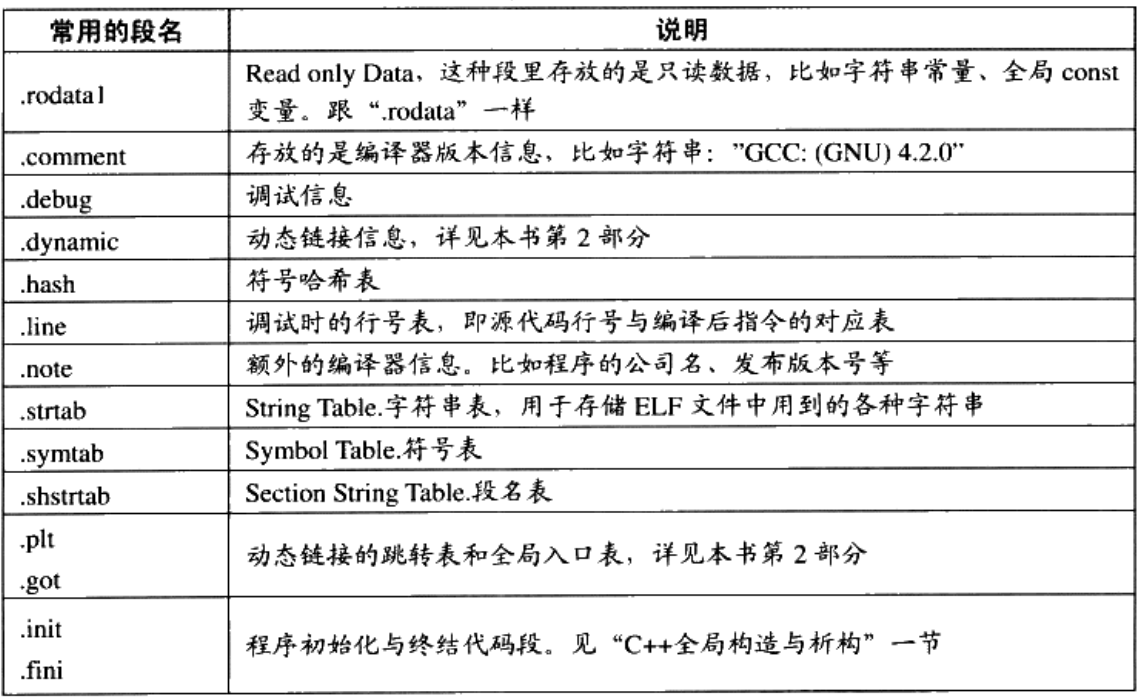


(1).text段: 用于存放程序代码，指令，函数，常量的区域， 编译时确定，如果有本文件未定义的函数引用，会搁置分配地址，等链接时会通过链接器更新函数地址。存放在Flash中，只读。

(2).data段 :用于存放已赋初值的全局变量，采用静态内存分配，为了掉电不丢失其值，存放在FLASH中，运行时将数据复制到RAM中使用。

(3).bss段：用来存放未赋初值的全局变量，采用静态内存分配，在可执行文件中只占用一个位用于存放变量的实际地址，程序初始化时会进入变量实际地址进行清零操作，初始化后存放在RAM中。

除了.text段、.data段、.bss段这3个常用的段之外，目标文件还包含其他的段，用来保存与程序相关的其他信息，如下。



### 链接脚本（布局、参数）

虽然目标文件已经是可执行文件的格式，但是还没有经过链接的过程，比如说有些符号或者有些地址还未被调整。使用链接工具可以将多个不同的目标文件之间相互连接在一起，目标文件之间的相互拼合实际上是目标文件之间对地址的引用，即队函数和变量的地址的引用。此外，因为不同的芯片所设计的地址空间是不同的，所以我们需要对目标文件的各个段进行设计排布，以符合芯片可寻址的空间。

链接脚本的作用就是规定如何把输入文件（即目标文件）的section放入输出文件（即最终的可执行文件），并且控制输出文件内各部分在地址空间内的布局。链接器一般有自己的内置链接脚本，但不符合我们的需要。使用-T选项可以指定自己的链接脚本以代替默认的链接脚本。

MEMORY

{

  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 0x10000

  RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 0x10000

  /\*定义内存布局与大小\*/

}

\_estack = 0x20010000;    /\*指定内存外栈指针地址\*/

SECTIONS          /\*设置段链接分布\*/

{

    .text :

    {

        . = ALIGN(4);    /\*四字节对齐\*/

        \_\_text\_start\_\_ = .;        /\*记录段开始地址\*/

        KEEP(start.o(\*.cortex\_vectors))/\*优先加载start.o中text段\*/

        KEEP(start.o(.text.Reset\_Handler))

        \*(.text\*)            /\*将所有.o文件的(.text\*)段汇集成一段\*/

        KEEP(\*(.init))

        KEEP(\*(.fini))

        \*(.rodata\*)

        . = ALIGN(4);    /\*四字节对齐\*/

        \_\_text\_end\_\_ = .;        /\*记录段结束位置\*/

    }> FLASH                    /\*保存内存属性\*/

    /\*加一条AT使其虚拟地址直接连在.text段后面\*/

    .data :AT (\_\_text\_end\_\_)

    {

        . = ALIGN(4);    /\*四字节对齐\*/

        \_\_data\_start\_\_ = .;

        /\*记录data段实际物理地址，启动后需将数据复制到实际地址中\*/

        \*(.data\*)

        . = ALIGN(4);    /\*四字节对齐\*/

        /\* All data end \*/

        \_\_data\_end\_\_ = .;

    }> RAM

    /\*禁止bss段产生内容，直接连在data段后面\*/

    .bss (NOLOAD):

    {

        . = ALIGN(4);    /\*四字节对齐\*/

        \_\_bss\_start\_\_ = .;    /\*bss段初始化后被分配的地址空间的地址\*/

        \*(.bss\*)

        . = ALIGN(4);    /\*四字节对齐\*/

        \_\_bss\_end\_\_ = .;

    }> RAM

}

AT（addr）：实现存放地址和加载地址不一致的功能，AT表示在文件中存放的位置，而在内存里呢，按照普通方式存储。

KEEP 关键字：在连接命令行内使用了选项–gc-sections后，连接器可能将某些它认为没用的section过滤掉，此时就有必要强制连接器保留一些特定的 section，可用KEEP()关键字达此目的。如KEEP(\* (.text))或KEEP(SORT(\*)(.text))

## Makefile

Makefile 可以简单的认为是一个工程文件的编译规则，描述了整个工程的编译和链接等规则。其中包含了那些文件需要编译，那些文件不需要编译，那些文件需要先编译，那些文件需要后编译，那些文件需要重建等等。编译整个工程需要涉及到的，在 Makefile 中都可以进行描述。换句话说，Makefile 可以使得我们的项目工程的编译变得自动化，不需要每次都手动输入一堆源文件和参数。

cc = arm-none-eabi-gcc

ld = arm-none-eabi-ld

objcopy = arm-none-eabi-objcopy

lds = link.ld

input = main.o start.o

output = main.hex

$(output): main

    $(objcopy) -O ihex main $(output)

main: $(input)

    $(ld) -nostdlib --gc-section -T $(lds) -Map main.map $^ -o main

$(input): %.o: %.c

    $(cc) -mthumb -mcpu=cortex-m3 -fdata-sections -ffunction-sections -O1 -c $< -o $@

.PHONY : clean

clean:

    -rm \*.o main \*.map \*.hex

对以上设计的makefile一些参数的解释：

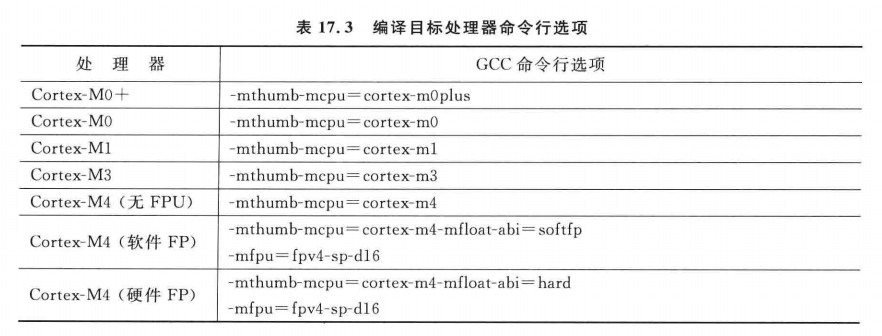
### gcc编译参数：

**-mthumb：**使用这个编译选项生成的目标文件是Thumb指令集。

**-mcpu：**表示对于源代码，指定针对指定的处理器进行编译。如：–mcpu=cortex-a53 : 指定处理器是cortex-a53，这样编译的指令，就是目标在该处理器上执行的指令。使用armclang –target=aarch64-arm-none-eabi –mcpu=list查看 aarch64下支持的处理器。

**-fdata-sections -ffunction-sections：**将每个函数或符号(数据)创建为一个sections，其中每个sections名与function或data名保持一致。而在链接阶段，gcc -Wl,–gc-sections 指示链接器去掉不用的section，这样就能减少最终的可执行程序的大小了。

**O1**：优化参数，对代码的分支，常量以及表达式等进行优化



### ld链接参数：

**-nostdlib：** 不连接系统标准启动文件和标准库文件，只把指定的文件传递给连接器。这个选项常用于编译内核、bootloader等程序，它们不需要启动文件、标准库文件。

**-gc-sections：**不链接没有使用到的函数、符号sections，以减小可执行文件的体积，某些需要保持的数据参数可在链接脚本中用keep指定

-T link.ld：指定自定义的链接脚本文件link.ld并按照自定义布局可执行文件

# 第二部分：

## 一、启动初始化 start.c

引用链接脚本文件中栈顶地址，利用地址和结构体分配一份空间存放复位函数和中断向量表，在复位函数中将data段数据进行拷贝，对Bss段数据进行清零，最后通过外部函数进入main函数，实现具体的程序设计内容

### 1、栈顶指针及栈大小分配

extern uint\_32 \_estack; //引用链接脚本中地址

typedef struct{ //定义空间结构体

    void\* estack;

    Handler vector[15];

} Vectors;

### 2、中断向量表初始化

const Vectors cortex\_vectors = {    //初始化结构体，赋予指针地址与内容

    &\_estack,

    {

         [0] = Reset\_Handler,

        // [1] = NMI\_Handler;

        // [2] = HardFault\_Handler;

        // [3] = MemManage\_Handler;

        // [4] = BusFault\_Handler;

        // [5] = UsageFault\_Handler;

        // [10] = SVC\_Handler;

        // [11] = DebugMon\_Handler;

        // [13] = PendSV\_Handler;

        // [14] = SysTick\_Handler;

    }};

### 3、复位中断服务函数

#### （1）将data段内容复制到实际地址

if(flash != ram)       //ram实际地址不是连在flash后面时，复制数据

    {

        for(;ram < &\_\_data\_end\_\_;)

        {

            \*ram = \*flash;

            flash ++;

            ram ++;

        }

    }

#### （2）清零bss段

for(ram = &\_\_bss\_start\_\_; ram < &\_\_bss\_end\_\_; ram ++)

    {

        \*ram = 0;    //进入bss段指向的地址中清零

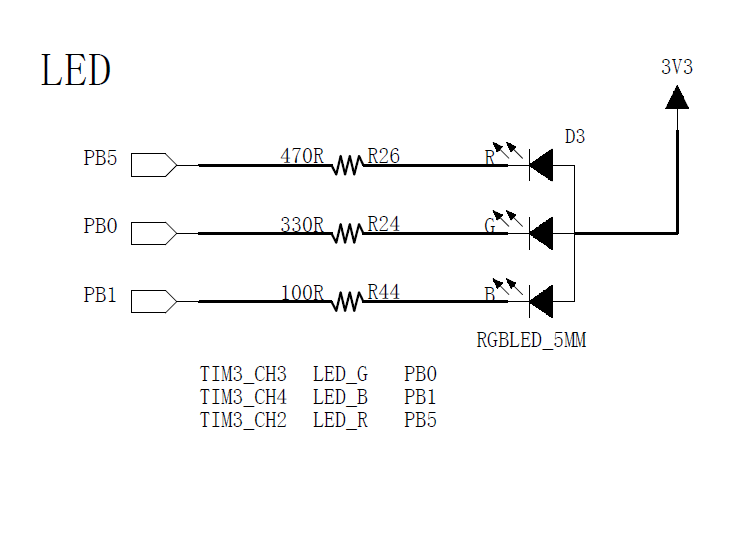
    }

#### （3）进入main()函数

## 二、主程序实现 main.c

### 1、查原理图

查询开发板原理图资料，可得出LED\_G端口为PB0，低电平点亮，接下来找到PB0的绝对地址，利用寄存器设置输出低电平



### 2、找到绝对地址

/\*片上外设基地址  \*/

#define PERIPH\_BASE           ((unsigned int)0x40000000)

/\*APB2 总线基地址 \*/

#define APB2PERIPH\_BASE       (PERIPH\_BASE + 0x10000)

/\* AHB总线基地址 \*/

#define AHBPERIPH\_BASE        (PERIPH\_BASE + 0x20000)

/\*GPIOB外设基地址\*/

#define GPIOB\_BASE            (APB2PERIPH\_BASE + 0x0C00)

/\* GPIOB寄存器地址,强制转换成指针 \*/

#define GPIOB\_CRL           \*(unsigned int\*)(GPIOB\_BASE+0x00)

#define GPIOB\_CRH           \*(unsigned int\*)(GPIOB\_BASE+0x04)

#define GPIOB\_IDR           \*(unsigned int\*)(GPIOB\_BASE+0x08)

#define GPIOB\_ODR           \*(unsigned int\*)(GPIOB\_BASE+0x0C)

#define GPIOB\_BSRR          \*(unsigned int\*)(GPIOB\_BASE+0x10)

#define GPIOB\_BRR           \*(unsigned int\*)(GPIOB\_BASE+0x14)

#define GPIOB\_LCKR          \*(unsigned int\*)(GPIOB\_BASE+0x18)

/\*RCC外设基地址\*/

#define RCC\_BASE      (AHBPERIPH\_BASE + 0x1000)

/\*RCC的AHB1时钟使能寄存器地址,强制转换成指针\*/

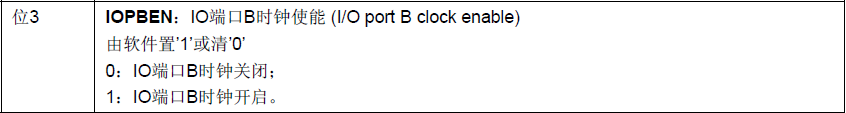
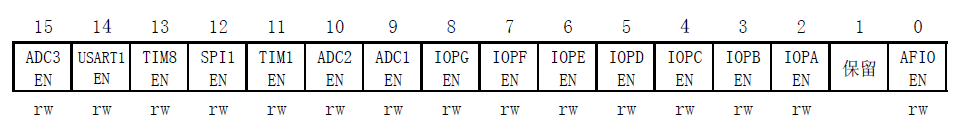
#define RCC\_APB2ENR      \*(unsigned int\*)(RCC\_BASE+0x18)

地址强制转换指针 #define GPIOB\_CRL \*(unsigned int\*)(GPIOB\_BASE+0x00)

(unsigned int\*)作用是将数据强制转换为地址指针，否则系统会将其认定为整形数据，转换后再用\*对地址进行操作

### 3、时钟使能

**APB2**外设时钟使能寄存器**(RCC\_APB2ENR)**



//开启GPIOB 端口时钟

RCC\_APB2ENR |= (1<<3);

### 4、操作寄存器输出

//清空控制PB0的端口位

GPIOB\_CRL &= ~( 0x0F<< (4\*0));

//配置PB0为通用推挽输出，速度为10M

GPIOB\_CRL |= (1<<4\*0);

//PB0 输出 低电平

GPIOB\_ODR &= ~(1<<0);

寄存器置位

置1：GPIOB\_CRL |= (1<<4\*0);

0000 0001左移4位后（0001 0000） 和寄存器(xxxx xxxx)进行【逻辑和】操作，最终使寄存器（xxx1 xxxx），不影响其他位数原始值

置0：GPIOB\_ODR &= ~(1<<0);

0000 0001左移0位后（0000 0001）取反（1111 1110）再和寄存器(xxxx xxxx)进行【逻辑与】操作，最终使寄存器（xxxx xxx0），不影响其他位数原始值