**MCU编译与运行浅谈**

2014-02-18 21:15:15| 分类： [LINUX](http://masust.blog.163.com/blog/#m=0&t=1&c=fks_084068082085081069085081085095085082087074081082086075080) | 标签：[mcu](http://masust.blog.163.com/blog/#m=0&t=3&c=mcu) [编译](http://masust.blog.163.com/blog/#m=0&t=3&c=编译) [运行](http://masust.blog.163.com/blog/#m=0&t=3&c=运行) |举报 |字号大中小 订阅

学电子的人都知道：使用C语言编写代码后，我们必须将其处理成机器码，才能使之在MCU中执行，这其中的步骤基本概括为“编译->汇编->链接->加载->启动”。

编译和汇编很容易理解，不是我想讲解的重点，这里主要讲“链接->加载->启动”三部分，我会尽量使用浅显的文字进行概括性阐述。很多电子工程师做了半辈子MCU可能还不能完全理解这三部分，不是说这些工程师们脑残，而是即使他们不搞懂这些也不影响他们做出好的设计，而且真正走上工作岗位后很少有机会像在大学期间有大把大把的时间去系统的学习知识了，企业里的设计讲究团队合作，很多电子工程师仅仅精通某一方面。我现在所在的公司里有很多SOC设计的高手，但相当一部分人并不是很清楚ARM体系架构与启动流程方面的知识。因此，我希望实验室内的学生可以常思考、相互交流心得，尽量在大学阶段打扎实基础，搞定每个细节，往往越细微以及容易忽视的知识点理解起来也越难。

**一、编译和汇编**

我们知道C代码经过编译、汇编的过程后可以得到目标文件，比较常用的编译/汇编器为Cx51、icc、gcc、armcc / armasm等。以ARM为例，我们一般使用armcc / armasm，它的编译效率以及代码的执行效率大大高于gcc，但很遗憾的是armcc / armasm是收费软件，如ADS、MDK、RVDS等都是使用的armcc / armasm，IAR for ARM使用的是自己设计的编译/汇编器，有兴趣的可以自己去网上查找相关信息。armcc对应命令行代码简略写为(以S3C6410为例)：

armcc ：armcc --debug --cpu=ARM1176JZF-S

armasm：armasm --debug --keep --cpu=ARM1176JZF-S

但是我们的工程一般都有很多c文件以及s文件，它们每个都会对应生成一个o文件（目标文件）（注意，如果说h文件没有实体定义就不会生成o文件，但如果h文件中有函数定义，那么目标文件就会很乱，因此我们一般遵守“h文件声明、c文件定义”的规则，也就即“函数定义写在c文件中，函数声明写在h文件中”，这也证明了对于一个函数“可以多次声明，但不可以多次定义”，比如对于一个h文件，我们可以不关注它是否被重复包含，但如果你想include一个c文件，就一定要看清楚这个c文件有没有被别的文件也包含了，否则的话就会出现“重复定义”错误）。那么链接的目的就是要把编译、汇编得到的目标文件合并成可执行程序，当然这里所说的目标文件也包括库文件以及一些命令行参数。

**二、合并（链接）和加载流程**

**链接器是对每一个程序的各个段进行绑定并分配相对地址，而加载器完成最后的重定位步骤并赋予实际地址**。

再通俗一点讲就是：编译器和汇编器通常为每个目标文件创建程序地址从0开始的目标代码，如果一个程序是由多个子程序组成的，那么所有的子程序必须被加载到互不重叠的地址上；链接器将多个子程序构建成一个程序，并生成一个链接好的起始地址为0的输出程序，各个子程序通过重定位在大程序中确定位置；当这个程序被加载时，系统会选择一个加载地址，而连接好的程序会作为整体被重定位到加载地址。

每个输入文件都包含一系列的段（segments），即会被连续存放在输出文件中的代码或数据块。一个ARM程序(指正在执行的程序，而非保存在ROM中的bin映像(image)文件)包含3部分：RO，RW和ZI,，但ARM映像文件只包含了RO和RW数据，这是因为ZI数据都是0，没必要包含，只要程序运行之前将ZI数据所在的区域一律清零即可：

1、 RO是程序中的指令和常量；

2、 RW是程序中的已初始化变量；

3、 ZI是程序中的未初始化的变量。

一般在汇编文件中可以看到，下例说明：AREA是指定区域的关键字，INITARM1136为段名，而且是RO类型的（PRESERVE8 指令用于指定当前文件保持堆栈八字节对齐)。

每一个输入文件至少还包含一个符号表（symbol table），一般由链接器定义，在当前文件中定义并在其他文件中使用，这也使得可以通过这些符号方便的进行目标文件的重组，如（region\_nane代表某个段，在下面的scatter文件中会涉及）：

Load$$region\_nane$$Base代表region\_nane（段名）加载时的起始地址

Image$$region\_nane$$Base代表region\_nane运行时的起始地址

Image$$region\_nane$$Length代表region\_nane运行时的长度

现在的链接器一般会进行两遍扫描，首先对输入文件进行扫描，得到各个段的大小，并收集对所有符号的定义和引用，创建一个含有所有段的段表，和包含所有符号的符号表。

利用第一遍扫描得到的数据，链接器可以为符号分配数字地址，决定各个段在输出地址空间中的大小和位置，并确定每一部分在输出文件中的布局，也即得到每个段的相对输出文件头的偏移地址。

第二遍扫描会利用第一遍扫描中收集的信息来控制实际的链接过程。它会读取并重定位目标代码，为符号引用替换数字地址，调整代码和数据的内存地址以反映重定位的段地址，并将重定位后的代码写入到输出文件中。（注意，第二遍扫描其实已经是加载器的功能了）

以X86处理器为例，在链接过后得到的机器语言文件中，每条机器指令包含了一个字节的操作码和其后4个字节的地址（因为是32位的处理器，所以地址为32位的）。假如链接器执行完成后得到下面的代码：

| 机器指令 | 汇编代码 |

| A1 34 12 00 00 | mov a，%eax |

| A3 12 9A 00 00 | mov %eax，b |

此处地址的存放为小印第安序，那么第一条指令的地址为0x1234，第二条为0x9A12。要注意，此处机器指令中的地址为相对地址，并不是执行时每条指令存放的实际地址。那么再通过加载器为其指定实际地址就完成整个应用程序的制作工作了。

假如说加载器指定程序加载的地址为0x40000000，那么加载器会自动的为之前得到的机器语言中每段的第一个指令中的地址加上0x40000000，这时就得到最终的机器语言了。然后再使用一些烧录工具将得到的机器码烧录到加载器指定的程序加载地址0x40000000处，再将ARM的程序指针PC指向0x40000000，程序就可以正常执行了。

Linker的命令行代码为：

armlink：armlink --scatter="E:\S3C6410\EB6410\_Test\6410\_scatter.txt"

或者armlink --ro\_base=0x8000000 --rw\_base=0x50200000

第一个是使用scatter文件，一般用于比较复杂的连接和加载操作，第二个仅仅指定了“只读区”以及“读写区”的起始地址。

Scatter文件一般在ADS、MDK、RVDS会用到，IAR等IDE并不常见。

注意，scatter文件里面的地址仅仅是为了改变最后生成的机器码中的地址，并不是说这里就已经将程序加载到单片机的内存里了，这需要专门的flash烧录工具或者其它下载方式将应用程序下载到链接时指定的地址处，如果链接时指定的地址和真正下载的地址不相符，程序会无法执行。

**三、程序烧录**

程序烧录也就是平时所说的下载，单片机常用的下载方式有ISP(In System Program)、IAP(In Appplication Program)和JTAG下载，下面将介绍这三种下载方式。

**(1)、ISP下载：**

一般芯片内部都固化了一个程序，也即自举程序，一般在出厂前用编程器烧录好，如果芯片使用过程中不小心将自举程序破坏，就只能使用编程器再次烧录，否则无法使用ISP进行下载应用程序。当然有需要时也可以使用编程器来升级自举程序。单片机上电后首先执行自举程序，如果ISP上位机有下载请求，则立即建立与上位机的通信，开始一边接收上位机传来的数据，一边将这些数据存放在芯片内部存储器(这个流程就叫做“下载”，注意，上位机不是直接将代码下载到芯片内部的flash中的，而是以自举程序作为中转站)；否则就会开始执行之前已经烧录的代码。因此使用ISP下载时一般先将ISP软件配置好，点击Download后，然后才能给芯片上电进行下载，否则如果先给芯片上电，就会出现自举程序检测不到下载请求而直接跳过去执行应用程序的情况。

这种下载方式仅仅是将代码下载到芯片内部flash中，但没有调试的功能，在开发稍微大点的项目时就会捉肘见襟，因此除了51、AVR等低级单片机还在使用之外，稍微高端的单片机基本上已不再使用。但是在芯片内部预留自举程序的理念一直延续了下去。

**(2)、IAP下载：**

顾名思义，就是在系统运行的过程中动态编程，这种编程是对程序执行代码的动态修改，而且毋须借助于任何外部力量，也毋须进行任何机械操作。这一点有别于ISP。目前IAP下载用的并不广泛，此处不再赘言。

**(3)、JTAG调试与下载：**

对于JTAG接口而言，其强大的调试接口早已盖过所谓的下载功能。JTAG可以说是目前MCU市场上最流行的调试接口，它可以提供单步、断点、半主机等很多有用的调试手段。JTAG可以取得指令执行的控制权，这是因为在正常运行状态下，处理器内核由MCLK（Memory Clock）驱动，正常运行；在调试状态下，处理器内核由内部的调试时钟DCLK（Debug Clock）驱动，这个时钟一般可以在Debug软件中手动修改，**时钟是芯片运作的原动力**，因此我们可以实现单步、断点等功能。

DCLK比MCLK慢很多，所以在调试状态下，插入的指令的运行速度会比较慢（现对而言）。在完成需要的操作后，可以用RESTART JTAG指令让处理器内核返回到正常运行状态，恢复原来的运行。在调试状态下，处理器内核的正常运行被打断，并且和系统的其它部分隔离开来；我们可以通过插入特定的指令来读写处理器内核的内部寄存器和修改内存的内容。

JTAG的下载功能相对于ISP有很大不同，它可以通过TAP接口控制所有IO口以及总线，也就是说通过JTAG可以模拟出任何一种接口协议，可以访问CPU总线上的所有设备。一般JTAG可分为RAM下载和Flash下载。

**1、RAM下载：** JTAG可以通过TAP接口直接读RAM，因此不需要自举程序在芯片内先运行也可以下载程序到RAM(使用JTAG下载可以没有自举程序)。一般编译生成的目标文件是ELF格式，或类似的格式，包含有目标码运行地址，运行地址在Link时候确定。Debug下载程序时根据ELF文件中的地址信息下载程序到指定的地址。如果在把RAM的基地址设置为0x10000000， 而在编译的时候指定的开始地址为0x02000000，下载时目标码将被下载到0x02000000，显然此时下载会失败。

还有一个问题，一般的RAM(如SDRAM或者DDR)在使用前需要初始化，比如通过配置RAM的寄存器设置其基地址，总线宽度，访问速度等等，否则RMA相关的寄存器会处在复位值，此时无法正确读写数据，下载必然要失败。有的芯片还需要Remap(重映射)，才能正常工作。因此使用JTAG向RAM中下载程序前先要想办法设置RAM，而在ISP下载中，上位机是通过自举程序做中转站，并不是真正访问Flash或者RAM。

在AXD(ADS的调试软件)中可以使用JTAG通过Set命令设置，并进行Remap，代码如下。使用其他debug，大体类似，只是命令和命令的格式不同。

setmem 0xfffff124,0xFFFFFFFF,32 －－－关闭所有中断

setmem 0xffe00000,0x0100253d,32 －－－设置CS0

setmem 0xffe00004,0x02002021,32 －－－设置CS1

setmem 0xffe00008,0x0300253d,32 －－－设置CS2

setmem 0xffe0000C,0x0400253d,32 －－－设置CS3

setmem 0xffe00020,1,32 －－－Remap

如果要在ADW（SDT带的DEBUG）中使用，则只需要将所有的setmem更改为let就行了。

为了方便使用，可以将上述命令保存为一个文件config.ini或者Measure.ini，可以在IDE中的Debug软件中进行设置，如下图所示：

这个文件在程序加载到RAM前通过JTAG的TAP接口执行，主要用于为RAM下载提供必要的初始化。

**2、Flash下载：** 理论上通过JTAG可以访问CPU总线上的所有设备，所以应该可以写Flash。但因为RAM在初始化后可以进行字节寻址，也即通过\*(**volatile** **unsigned** **long**\*)(reg)的语句进行读写，但是Flash写入方式和RAM大不相同，需要特殊的命令，而且不同的Flash擦除，编程命令不同，而且块的大小，数量也不同，很难提供这一项功能，所以一般Debug不提供写Flash功能，或者仅支持少量几种Flash。

通过TAP进行读写Flash并不常用，我们比较常用的Flash下载方式类似于ISP下载，也即预先在内存中启动自举程序，这里可以称为bootloader程序，然后使用JTAG将代码下载至SRAM或者SDRAM，然后再由bootloader将代码写入Flash。还有一种方法是将bootloader代码写在要烧录的代码开头，主要是将紧跟其后的需要写入Flash的代码写入Flash，将此代码下载至内存后，令程序指针PC指向此代码的开头，也即开始执行“将后面的程序写入Flash”的功能。

南京博芯公司写的一篇**《使用Jlink烧写UB4020EVB的NorFlash.pdf》**感觉很有启发性。

再比如做嵌入式linux的人都用过u-boot下载系统镜像以及文件系统到NandFlash，我将其看作是一种高级的ISP下载，芯片先运行u-boot代码，等待上位机传送镜像，当然此时的下载接口不一定非得是串口了，现在的u-boot可以支持串口、网口、USB口下载。有的人可能还会有疑问：u-boot是不是也是出厂是烧录进去的呢？

其实这就要讲到ARM的启动方式问题了，我们预先把u-boot代码烧录在NandFlash和SD卡里(这件事很容易，特别是烧录代码到SD卡)，ARM上电后会自动从SD卡或者NOR/NAND Flash中读取前8K的代码到SRAM执行(当然具体从哪个存储器取数据是可以配置的)，这段代码主要是初始化堆栈、SDRAM / DDR、打开MMU/Cache功能以及搬运其它剩余的u-boot，然后就是我前面所说的ISP下载了。当然这只是简单介绍，关于ARM详细的启动流程可参考**《ARM启动释疑.pdf》**，文章下面我将会主要讲述芯片启动。

**四、芯片上电启动**

目前市面上的芯片种类成千上万，启动流程细节上虽然各不相同，但基本上还是可以归结为两类：

对于低级单片机来说，芯片上电后都是先运行自举程序，然后再从存储空间的零地址开始执行代码，内部flash是不是NOR？

对于高端一点的芯片，上电后会通过一些手段将预先写好的启动代码读入SRAM中执行。这里所说的“手段”可以是在MCU设计时做在硬件里(这在当今EDA技术如此发达的情况下是很容易做到的)，芯片上电后会自动初始化Nand或者SD卡，然后DMAC总线去这两个存储器搬运前8k的启动代码到SRAM；也可以是在设计MCU时在芯片内部添加一个片内ROM外设，用于存储自举程序，主要功能就是初始化芯片的基本功能，搬运少量启动代码到内存运行，接着内存中的部分启动代码再搬运所有的代码到内存运行。

关于启动部分比较重要的就是程序运行的内存地址必须和链接时指定的地址相同，否则即使程序烧录进内存，但因指令内的地址与实际运行地址不匹配，程序依然无法运行。

还有一个概念比较重要，不过只会在ARM中碰到，就是**处理器运行模式**问题，ARM有七种运行模式(注意和ARM的两种工作状态(ARM状态和Thumb状态)区分)：

1. 用户模式：在执行完CPU启动代码文件startup.s后正常的程序执行状态，

2. 系统模式：运行一些操作系统核，

3. IRQ (中断)模式：通用的中断处理模式，

4. FIQ (快速中断)模式：快速中断，处理一些特殊的中断源，

5. 管理模式：进入保护状态的执行；通常在复位或使用SWI指令时进入此模式，

6. 异常模式：在数据或指令预取失败时进入此模式，

7. 未定义模式：当执行一个未定义的指令时进入此模。

七种运行模式下的寄存器组占用也有所不同(要注意此处的寄存器并不是指其它片内外设的功能寄存器，而是ARM内核里的寄存器)，可以使用软件修改CPSR 的M[4:0]位来决定处理器的工作模式。

( R0~R12一般作为通用寄存器使用，也即存放临时变量以及函数形参等，比如R0~R3用于存放C语言中子函数的形参，R4~R7用于存放C代码中定义的变量，因此如果子函数的形参超过四个，R0~R3不够用，处理器就会将多出来的形参保存在内存里，而不是通用寄存器里，这样就会影响子函数的执行效率，比较好的解决方式是定义一个结构体，将所有形参放入此结构体中，以这个结构体的指针作为形参，这样使用一个通用寄存器即可。当然，R0~R7与R8~R12又有所不同，有兴趣的可以自己去网上详查。

比较特殊的寄存器有：

**1、**寄存器R13通常用作堆栈指针，称作SP；

**2、**寄存器R14用作子程序链接寄存器，也称为链接寄存器LK，仅当执行带链接分支（BL跳转）指令时备份R15(PC指针)中的值(也即保存程序跳转前的地址)，其他情况下将R14当做通用寄存器使用；

**3、**R15为程序指针，可以使用JTAG调试时，可以在内存窗口手动设置PC的值，然后点击运行按钮后芯片就会从设置的地址处开始运行代码；

**4、**寄存器R16用作程序状态寄存器CPSR(当前程序状态寄存器)，在所有处理器模式下都可以访问CPSR。CPSR包含条件码标志、中断禁止位、当前处理器模式以及其他状态和控制信息；每种异常模式都有一个程序状态保存寄存器SPSR，当异常出现SPSR用于备份CPSR的状态 )

除**用户模式**以外，其余的所有6种模式称之为非用户模式或**特权模式**；其中除去用户模式和系统模式以外的5种又称为**异常模式**，常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

异常我们会在下面的“中断响应”部分讲，这里主要区分用户模式和系统模式。

一般情况下，如果仅对于前后台运行的程序，可以不用关心处理器是用户模式还是系统模式的问题。如果对于运行操作系统，这时就有必要进行区分：

系统模式具有和用户模式完全一样的资源(可见寄存器)，但具有可以直接切换到其它模式的特权，而且也可访问受限制的资源，比如对CPSR，SPSR进行访问。而用户模式不能进行这样的操作，以免误了操作系统的设置。

系统模式并不是通过异常过程进入的，它和用户模式具有完全一样的寄存器。但是系统模式属于特权模式，可以访问所用的系统资源，也可以直接进行处理器模式切换。它主要供操作系统任务使用。通常操作系统的任务需要访问所有的系统资源，同时该任务仍然使用用户模式的寄存器组，而不是使用异常模式下相应的寄存器组，这样可以保证当异常中断发生时任务状态不被破坏。

大多数的用户程序运行在用户模式下。这时，应用程序不能够访问一些受操作系统保护的系统资源。应用程序也不能直接进行处理器切换。当需要进行处理器模式切换时，应用程序可以产生异常处理，在异常处理过程中进行处理器模式的切换。

**五、中断响应**

MCU里比较流行的中断有两种：向量中断、非向量中断。向量中断就是不同的中断有不同的入口地址，非向量中断就只有一个入口地址，进去了在判断中断标志来识别具体是哪个中断，其实两种中断方式的编程方法一模一样，只不过中断响应效率上面有些差别，向量中断实时性好，非向量中断简单。

下面说一下产生异常(中断)后芯片的执行流程，以ARM为例：

当一个异常或中断发生时，处理器会把PC(程序指针)设置为一个特定的存储器地址，这个地址放在一个被称为向量表(vector table)的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

一般情况下**存储器映射地址**0x00000000是为向量表保留的(对于低级单片机来说就是物理地址，而ARM芯片是有存储器映射功能的)。但在有时向量表可以选择定位在存储空间的更高地址(从偏移量0xffff0000开始)，其实WinCE就是采用0xffff0000作为中断向量表的地址，估计产生异常后的跳转地址是由编译器决定的。

当一个异常或中断发生时，处理器挂起当前正在执行的操作，转而从向量表装载指令。每一个向量表入口包含一条指向一个特定子程序的跳转指令：

0x00000000(0xffff0000) - 复位

0x00000004(0xffff0004) - 未定义指令

0x00000008(0xffff0008) - 软件中断

0x0000000c(0xffff000c) - 预取指令中断

0x00000010(0xffff0010) - 数据中断

0x00000014(0xffff0014) - 保留

0x00000018(0xffff0018) - 中断请求

0x0000001c(0xffff001c) - 快速中断请求

有兴趣的可以去查看工程代码中的start.s文件或者其它名字的启动文件，里面一般都会有这么几句：

里面的RST\_DO以及IRQ\_DO都是汇编代码里的标号，类似与C语言中的函数名，里面就是各个异常的处理代码，意思就是跳转到相应的异常处理代码中去运行。

为什么还要“跳转到其他地方去执行处理代码”这么麻烦？这是因为(由上面的向量表也可以看出)芯片默认只分配4字节的空间用于处理每个异常，这样，也即只能放一条指令(ARM是32位处理器，一条指令占四个字节)，但异常处理代码肯定不止一条指令，所以就把处理代码放在其他地址，再使用一条跳转指令跳转到那个地址去执行。

为什么“默认只分配4字节的空间用于处理每个异常”呢？这是因为在产生异常时PC自动跳转到对应地址去执行，也就是说这个地址必须是事先固定好的，否则无法得知跳转到哪里，也就是说每个异常的处理空间是固定的，但是每个异常的处理代码大小都是可变的，我们就算给每个异常固定分配1K的空间，也不能保证一定够用。因此在设计时就规定每个异常只分配四字节用于跳转，**像上面写的“bal”跳转代码一定是在0地址(**也可以为重映射后的0地址**)依次存放，每条指令占用四字节，即使某个异常没有处理程序，也要用一条空语句或者无用语句代替，否则产生异常时程序会跑飞或者进入错误的处理程序中**。

说到这里才只说到异常，没有提到中断，其实异常包括IRQ中断以及FIQ中断。当产生中断后，肯定会进入前图所示的向量表，然后跳转到IRQ或者FIQ去执行处理程序，注意， IRQ的处理程序依旧是个向量表，主要是跳转到ARM芯片的各个中断处理函数，比如UART\_ISR等，这些函数应该很熟悉，都是我们自己定义和修改的。也就是中断处理其实经过了两级跳跃。其它的像进入中断前后的现场保护和恢复，以及执行中断时的使能、屏蔽等操作，有兴趣的自己去查阅资料，不是这里讲解的重点。