DSP启动过程

|  |  |
| --- | --- |
| 编 制 人： | Lizy |
| 发布日期： | 2023.3.4 |
| 页 数： |  |

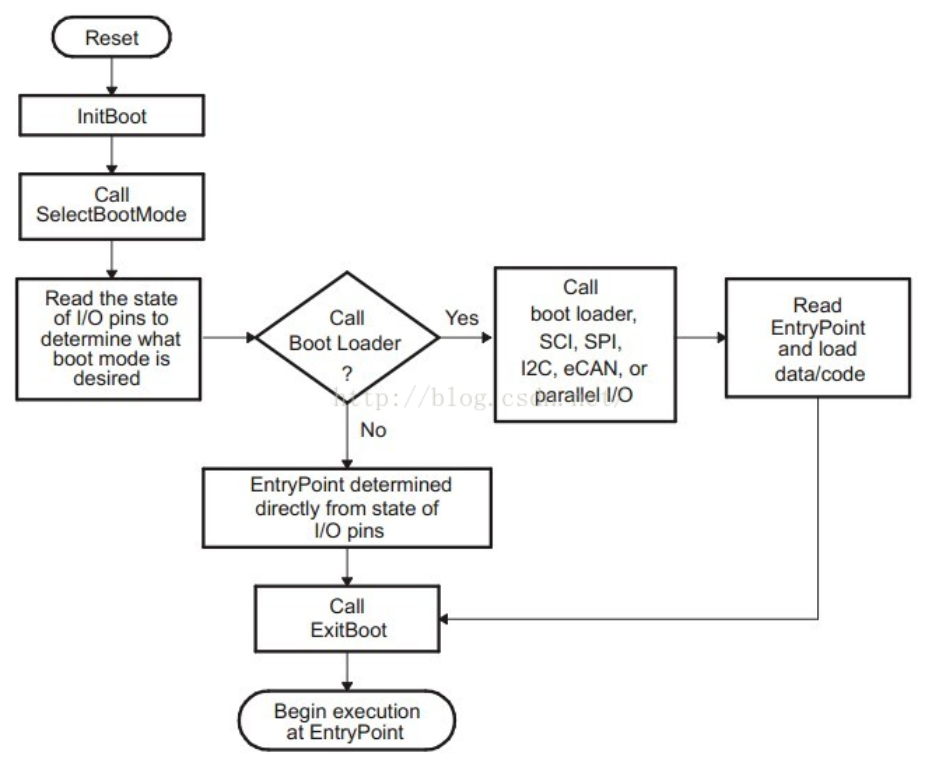
更改记录

|  |  |
| --- | --- |
| **发布日期** | **更改描述** |
| 20230304 | 第一版发布 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# DSP F28335启动过程

## 启动基本流程

TI支持很多方式的boot，内部RAM（SARAM）、FLASH、SCI等，但无论哪种启动方式都要遵循下面的流程：



流程中的reset、initboot、call selectbootmode、read i/o state、call boot loader这些步骤都是固化在芯片内部的程序自己执行的，也就是说这些代码在出厂时已经在芯片内部了。

我们写的程序是从codestart处开始执行，其代码在DSP2833x\_CodeStartBranch.asm内，其指定位置在CMD文件内定义。如果程序下载到FLASH中，codestart在flash的BEGIN:0x33FFF6；如果程序加载到RAM中，codestart在RAM的BEGIN:0x000000中（RAM引导不需要执行bootloader程序）。

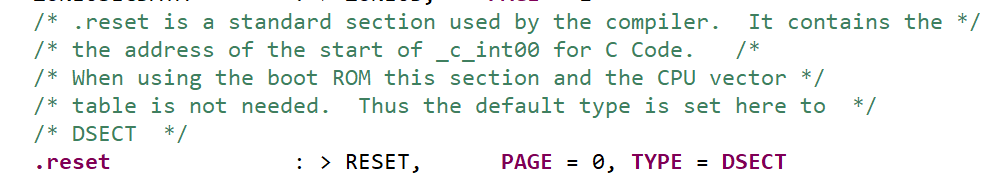
## 启动过程



根据上面流程图，详细解释一下启动过程。

1. 第一步：BROM Vector（0x3FFFC0 – 0x3FFFFF）存放的是中断向量表，系统一上电（上电或芯片Reset后，VMAP = 1、ENPIE = 0，BROM向量表区被激活），处于reset中断，因此直接跳到reset的地方执行（0x3FFFC0）。而这个地方的两个字节只放了一条跳转到0x3FF34C（Boot ROM区）的跳转指令，运行ROM内的固化引导程序bootloader；



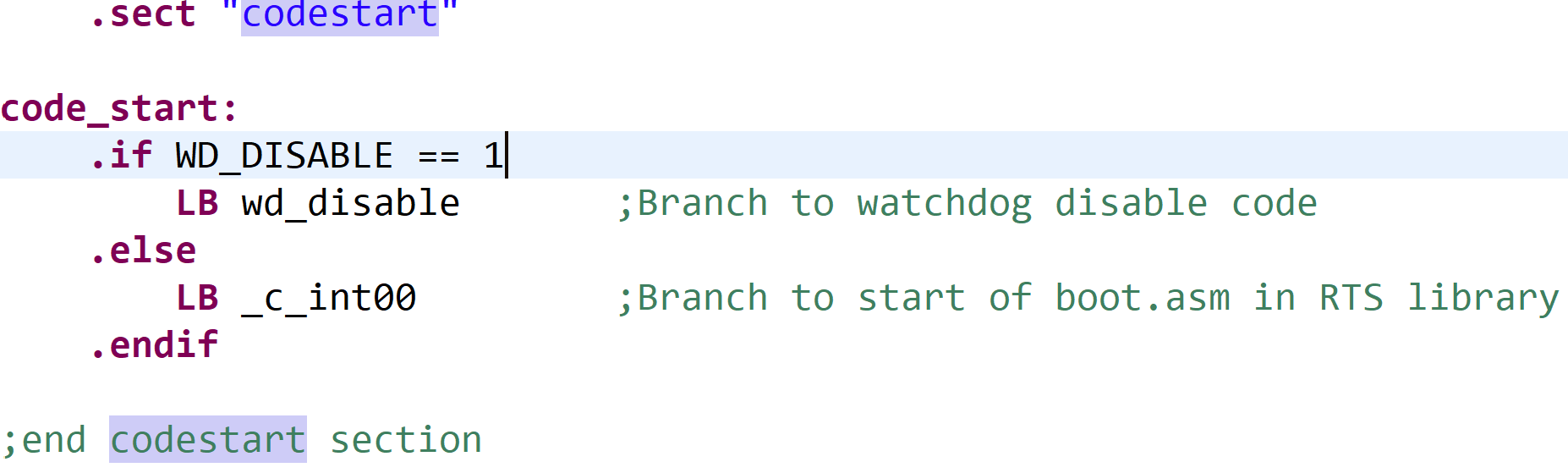


.reset和reset是不同的，一个是memory，一个是section。

.reset只包含一个32位的中断矢量，指向实时支持库rts2800\_ml.lib中的C编译器引导函数，即\_c\_int00子程序。通常我们不用此块，而是另外创建分支指向开始代码。如4）中介绍

1. 第二步：在0x3FF34C中的bootloader程序，主要有initboot、SelectBootMode、read GPIO、外设引导等函数。SelectBootMode根据芯片的硬件或软件设置来判断芯片该区哪里寻找程序的入口，直接目的就是如何找到main函数，然后执行应用程序。Bootloader操作会去检测外部GPIO口的状态，从而判断是哪种方式的启动；
2. 第三步：然后根据相应的启动方式调至相应的入口地址：比如FLASH启动就是0x33FFF6，内部SARAM启动就是0x000000；
3. 第四步：而这里的入口地址就是cmd文件中定义的BEGIN段。因此对于FLASH启动和SARAM启动，BEGIN段的定义是不同的，在FLASH启动时BEGIN就是0x33FFF6，而内部SARAM启动时BEGIN就是0x000000。这个2个字符的区间也就是我们程序最初执行的第一条指令（通常是code\_start）,它里面是一条长跳转指令LB刚好占两个字节。

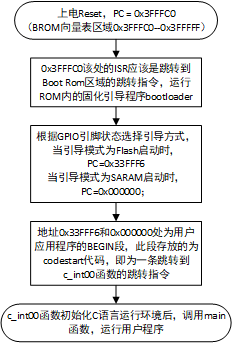
Bootloader程序执行完毕后会跳到FLASH的BEGIN（0x33FFF6），而codestart就在此处。即执行的是DSP2833x\_CodeStartBranch.asm代码；



codestart包含一条长跳转指令，指向实时支持库rts2800\_ml.lib中的C编译器引导函数，即\_c\_int00子程序。

1. 第五步：\_c\_int00 定义在 rtsxxx.lib 库中，c\_int00() 函数完成：初始化堆栈指针和页指针；初始化全局变量；最后调用 main()。

**小结：**



DSP F28335启动过程：

1. 上电Reset后，程序指针PC = 0x3FFFC0（RESET中断），而BROM向量表区域0x3FFFC0–0x3FFFFF；
2. 0x3FFFC0处的ISR应该是跳转到Boot Rom区域的跳转指令，运行ROM内的固化引导程序bootloader；
3. bootloader根据GPIO引脚状态选择引导方式，当引导模式为Flash启动时，PC=0x33FFF6；当引导模式为SARAM启动时，PC=0x000000；
4. 地址0x33FFF6或0x000000处为用户应用程序的BEGIN段，此段存放的为codestart代码，即为一条跳转到c\_int00函数的跳转指令；
5. c\_int00函数初始化C语言运行环境后，调用main函数，运行用户程序。

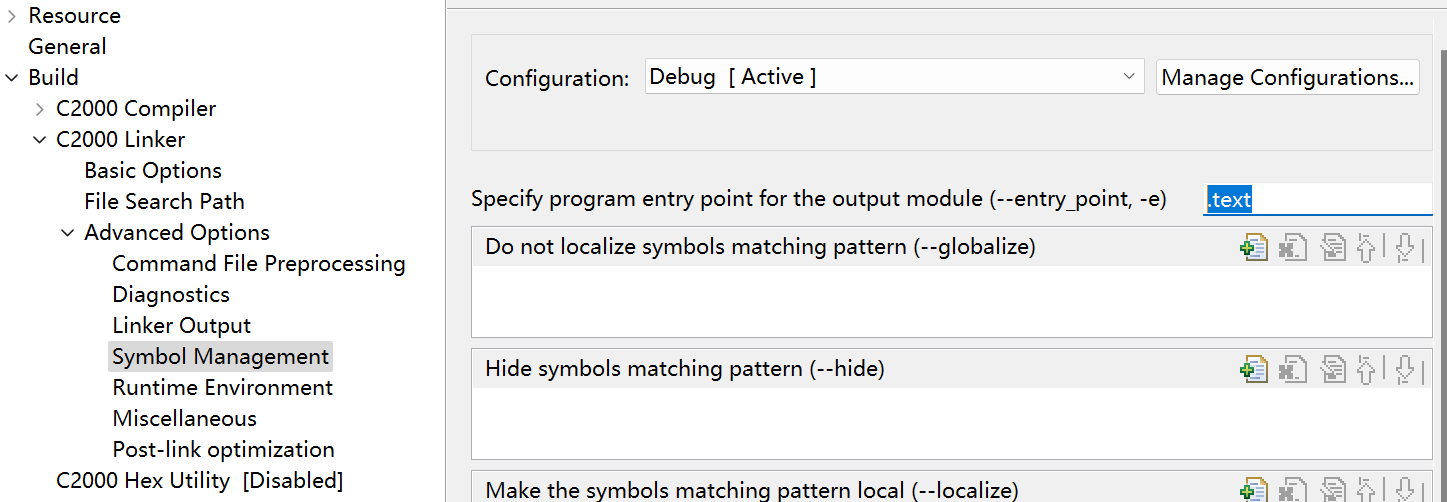
## 补充

在28335的TI提供的例程中，程序的起始位置是code\_start，在这里先禁止看门狗，然后再跳转至c\_int00处运行；而在一般的C工程中，这个起始位置一般是c\_int00；这个起始位置在编译选项（build options）中设定，在C环境建立之前将看门狗禁止，使得程序更可靠。

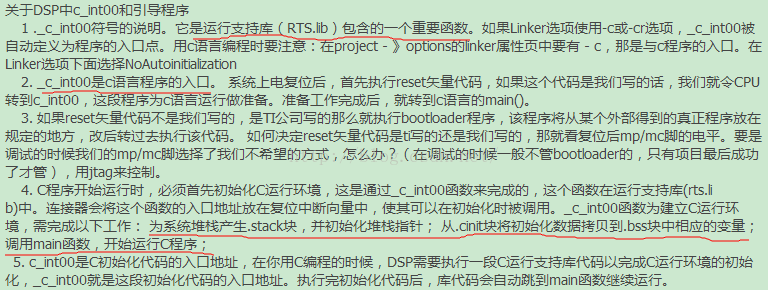
### 程序入口地址

CPU启动或复位后开始执行的程序地址。一般默认程序入口有三种：地址 0000、\_c\_int00 和 \_main。

此外还可以自行设置入口地址 Project properties-Built-C2000 Linker-Advance Options-Symbol Management：



### c\_int00介绍



\_c\_int00是C初始化代码的入口地址；

在你用C编程的时候，DSP需要执行一段C运行支持库代码以完成C运行环境的初始化，\_c\_int00就是这段初始化代码的入口地址，

\_c\_int00函数在运行支持库（rts,runtime-support library）中。连接器会将这个函数的入口地址放置在复位中断向量处，使其可以在初始化时被调用。c\_int00函数进行以下工作以建立C运行环境：

1. 为系统堆栈产生.stack块，并初始化堆栈指针；
2. 从.cinit块将初始化数据拷贝到.bss块中相应的变量；
3. 执行完初始化代码后，就跳转到main函数，开始运行C程序；

另外：

rts2800.lib:C/C++运行支持库;

rts2800\_ml.lib C/C++大内存模式运行支持库.

rts2800\_ml.lib中有大量浮点运算处理的函数而rts2800.lib没有

在指针的访问空间上有区别，rts2800.lib中库函数的指针为near，故不能访问3Fxxxx，rts2800\_ml.lib可以访问(大小内存模式故名思议就是可以访问的内存的大小有区别，小内存模式只能访问低64k地址，也就是16位地址线)

### 连接仿真器

假设我们配置cmd文件；

BEGIN: origin=0x082000,length=0x000002

我们连接仿真器在线运行时，程序是可以正常运行的，但是当我们不连接仿真器时，将DSP芯片上电，此时，程序的确已经烧写进入dsp的flash芯片当中，但是芯片无法正常运行；我们需要将程序的cmd文件设置为：

BEGIN : origin=0x080000,length=0x000002

codestart : > BEGIN

即将codestart执行地址通过cmd文件链接至0x80000处（不同型号芯片的地址不完全相同），程序才可以正常执行。仿真器其实就是一个小的嵌入式软件。理论上我们可以通过DSP来实现一个仿真器的功能。仿真器功能开发时有可能加入了这种begin位置自动识别跳转功能（在仿真模式下忽略了启动模式的选择，而是让仿真器直接将程序的起始地址赋给PC的）。

### 反汇编

通过调试界面的view中的memory broswer 选项，查看FLASH地址中的内容， FLASH中保存的都是数据，这些“数据”也就是可执行的机器码，这段汇编代码具体的意思，可以通过dsp的反汇编来查看；

在dsp在线调试状态下，进入debug界面，打开view菜单栏中的Disassembly选项（反汇编）；这里我们可以查看片内FLASH上不同地址中对应的汇编代码：

