



(1)uboot主要作用是用来启动操作系统内核。

(2)uboot还要负责部署整个计算机系统。

(3)uboot中还有操作Flash等板子上硬盘的驱动。

(4)uboot还得提供一个命令行界面供人来操作。

(2)uboot本质上是一个裸机程序（不是操作系统），一旦uboot开始SoC就会单纯运行uboot（意思就是uboot运行的时候别的程序是不可能同时运行的），一旦uboot结束运行则无法再回到uboot（所以uboot启动了内核后uboot自己本身就死了，要想再次看到uboot界面只能重启系统。重启并不是复活了刚才的uboot，重启只是uboot的另一生）

(3)uboot的入口和出口。uboot的入口就是开机自动启动，uboot的唯一出口就是启动内核。uboot还可以执行很多别的任务（譬如烧录系统），但是其他任务执行完后都可以回到uboot的命令行继续执行uboot命令，而启动内核命令一旦执行就回不来了。

1. 在器件上电运行后，处理器自动开始Stage-0 Boot，也就是执行片内BootROM中的代码

2. BootROM会初始化CPU和一些外设，以便读取下一个启动阶段所需的程序代码，FSBL（First Stage Bootloader）。

不过这又有一个问题了----之前说到，Zynq支持多种启动设备，BootROM怎么知道从哪个启动设备里去加载FSBL？这就得靠几个特殊的MIO引脚来选择了，

BootROM会去读取MIO[2..8],从而确定启动设备，将选定设备的头192Kbyte内容，也就是FSBL，复制到OCM(On Chip Memory)中，并将控制器交给FSBL。

3. FSBL启动时可以使用整块256Kb的OCM，当FSBL开始运行后，器件就正式由咱自己控制了。Xilinx提供了一份FSBL代码，如果没什么特殊要求，可以直接使用。

按照手册说明，FSBL应该完成以下几件事。

1). 使用XPS提供的代码，继续初始化PS

2). 将bitstream写入PL（配置FPGA），不过这一步也可以以后再做

3). 将接下来启动用的Second Stage Bootloader（SSBL，一般就是U-Boot一类的东西)，或者裸奔程序，复制到内存中

4). 跳到SSBL运行去

4. 接下来的步骤就没啥特别了，Uboot开始运行，会自动加载devicetree、zImage和rootfs，初始化好Linux启动环境，然后开始运行Linux系统。U-boot完成linux内核启动之前所必须的硬件初始化，例如串口，DDR控制器等并且支持从本地网络中加载linux内核，设备镜像树到内存中。之后U-Boot将控制权交给linux内核。

集齐u-boot.elf、system.bit和fsbl.elf就可以生成BOOT.BIN文件

*# zImage  - compressed kernel image  
# uImage  - zImage plus U-Boot header  
uImage是在zImage之前加上一个长度为0x40的“头”，说明这个映像文件的类型、加载位置、生成时间、大小等信息。换句话说，如果直接从uImage的0x40位置开始执行，zImage和uImage没有任何区别。*

设备树编译

Device Tree：一种描述硬件资源的数据结构。可以描述的信息包括CPU的数量和类别、内存基地址和大小、总线和桥、外设连接、中断控制器和中断使用情况，GPIO控制器和GPIO使用情况、Clock控制器和Clock使用情况。  
设备树的编译器是dtc，只要编译过内核，就会生成dtc。dtc编译器可以把dts文件编译成为dtb，也可把dtb编译成为dts文件，编译命令格式如下：  
dtc [-I input-format] [-O output-format][-o output-filename] [-V output\_version] input\_filename  
参数说明  
   
input-format：   
- “dtb”: “blob” format   
- “dts”: “source” format.   
- “fs” format.  
   
output-format：   
- “dtb”: “blob” format   
- “dts”: “source” format   
- “asm”: assembly language file  
   
output\_version：   
定义”blob”的版本，在dtb文件的字段中有表示，支持1 2 3和16,默认是3,在16版本上有许多特性改变  
   
(1)  Dts编译生成dtb  
./dtc -I dts -O dtb -o B\_dtb.dtb A\_dts.dts  
把A\_dts.dts编译生成B\_dtb.dtb  
   
(2)  Dtb编译生成dts  
./dtc -I dtb -O dts -o A\_dts.dts A\_dtb.dtb  
把A\_dtb.dtb反编译生成为A\_dts.dts

.dts文件是一种ASCII文本对Device Tree的描述，放置在内核的/arch/arm/boot/dts目录。一般而言，一个.dts文件对应一个ARM的machine。   
由于一个SOC可能有多个不同电路板，而每个电路板拥有一个.dts。这些dts势必会存在许多共同部分，为了减少代码的冗余，设备树讲这些共同部分提炼保存在.dtsi文件中，供不同的dts共同使用。.dtsi的使用方法，类似于C语言的头文件，在dts文件中需要进行include .dtsi文件。当然，dtsi本身也支持include另一个dtsi文件。

当默认的环境变量不符合要求时，可以用uEnv.txt设置新的环境变量

**console**

        指定控制台的设备以及波特了

**root**

        指定挂载的根文件系统，这里是/dev/mmcblk0p2，当时在创建linaro的sd卡镜像时会特别提示需要在SD卡中创建两个分区，第一个分区是FAT文件系统，存放内核，设备树，BOOT.bin等，而第二个就是存放Linaro文件系统了，这里也就是对应的mmcblk0p2设备。

**rootfstype**

        制定跟文件系统的类型，这里是ext4

**rw**

        rw参数告诉内核以读写方式加载根文件系统。 ro参数告诉内核以只读方式加载根文件系统，以便进行文件系统完整性检查，比如运行fsck；

**earlyprintk**

        在console设备注册前（也就是printk注册之前）提供对打印函数的支持，这个之前就可以使用early\_printk()函数来代替printk()函数

**rootwait**

        让内核等待所有设备都被初始化完成后，再去执行root文件系统的挂载工作。这样可以避免根文件系统驱动初始化成功之前就挂载根文件系统。详细的说明可以参考<http://blog.csdn.net/liujixin8/article/details/5704991>

**devtmpfs.mount**

        是否挂载devtmpfs，1是挂载，0是不挂载。