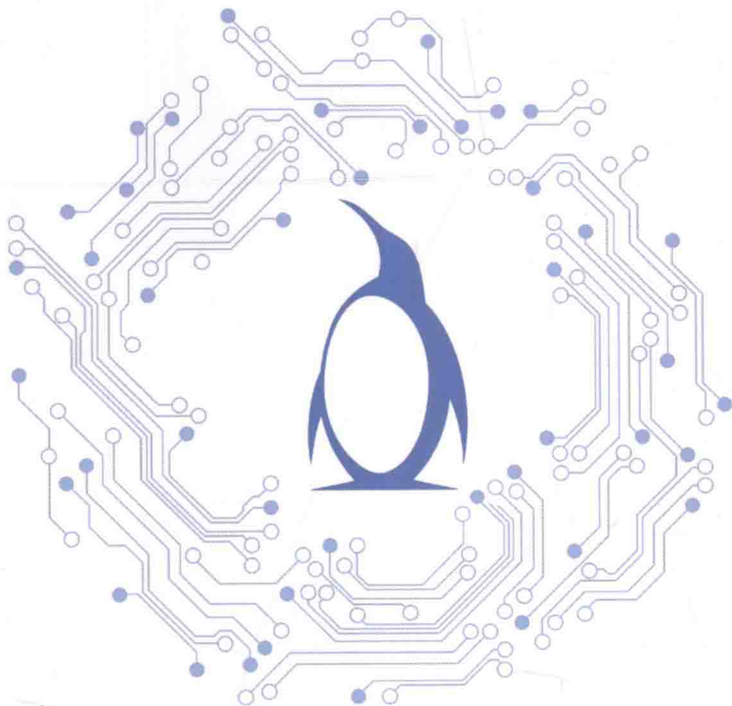
HZ BOOKS
华章科技

本书系统讲解BootLoader开发理论、流程与实例。
本书以ARM和Linux为蓝本，详细介绍了如何实现可用的BootLoader。
本书是嵌入式Linux移植开发领域的工程师不可多得的参考书。

电子与嵌入式系统
设计丛书

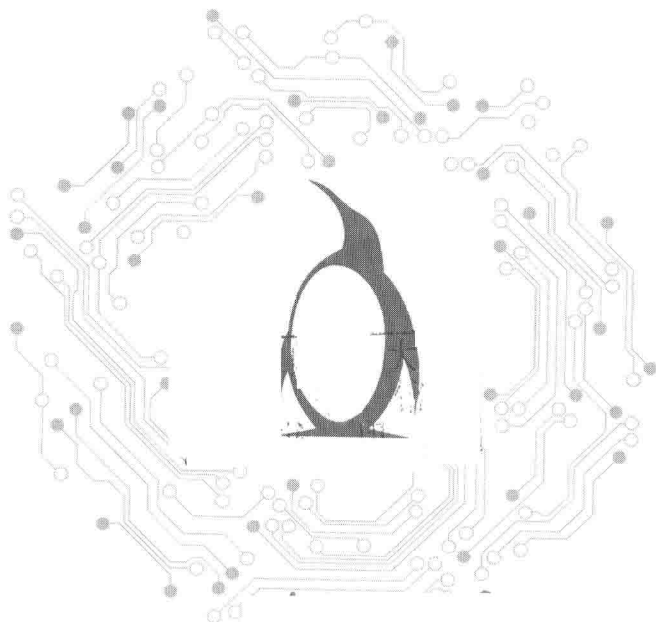


# 深入理解 BootLoader

胡尔佳 编著

机械工业出版社
China Machine Press

电子与嵌入式系统
设计丛书



# 深入理解 BootLoader

胡尔佳 编著

机械工业出版社
China Machine Press

## 图书在版编目（CIP）数据

深入理解 BootLoader/ 胡尔佳编著 . —北京：机械工业出版社，2015.1
（电子与嵌入式系统设计丛书）

ISBN 978-7-111-48570-4

I. 深…　II. 胡…　III. 单片微型计算机　IV. TP368.1

中国版本图书馆 CIP 数据核字（2014）第 267294 号


本书主要介绍 BootLoader 的开发理论、流程与实例，以当前流行的 ARM 和 Linux 为蓝本，详细介绍了如何一步步实现可用的 BootLoader。该书既对 Linux 开发环境、编译器使用、处理器架构以及编译和链接、链接脚本的细节做了较全面的理论介绍，又结合具体的环境向读者说明了 BootLoader 的原理和开发流程，使读者真正懂得 BootLoader 是如何工作的，即便今后遇到其他处理器或者引导其他操作系统，也能熟知开发或者移植 BootLoader 的思路。

本书是初涉 BootLoader 移植开发领域读者的一本不可多得的参考书。书中介绍的理论不仅仅对理解 BootLoader 有帮助，而且对理解计算机系统（嵌入式系统）也有启发意义，适合广大嵌入式系统爱好者和开发人员参考使用。

# 深入理解 BootLoader

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）

责任编辑：缪　杰　　张梦玲　　　　责任校对：殷　虹

印　　刷：中国电影出版社印刷厂　　版　　次：2016 年 7 月第 1 版第 1 次印刷

开　　本：186mm×240mm　1/16　　印　　张：17.25

书　　号：ISBN 978-7-111-48570-4　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com



版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

# 前　言

BootLoader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射，从而将系统的软硬件环境设置成一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

在嵌入式系统中，通常没有像 PC 中的 BIOS 那样的固件程序，因此整个系统的加载启动任务就完全由 BootLoader 来完成。BootLoader 是 CPU 上电后运行的第一段程序，它的作用就是对嵌入式系统中的硬件进行初始化，创建内核需要的一些信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，真正起到引导和加载内核的作用。实际上，一个功能比较强大的 BootLoader 已经相当于一个微型的操作系统了。

不同的 CPU 体系结构有不同的 BootLoader。有些 BootLoader 支持多种体系结构的 CPU，比如 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外，BootLoader 实际上也依赖于具体的嵌入式板级设备的配置。也就是说，对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要想让运行在一块板子上的 BootLoader 程序也能运行在另一块板子上，通常需要修改 BootLoader 的源程序。因此每款嵌入式产品的 BootLoader 都是独一无二的，但我们可以总结出开发或者维护特定 BootLoader 需要哪些背景知识，掌握了这些背景知识，我们就可以做到以不变应万变。

为了引导操作系统，BootLoader 与 CPU 体系结构和操作系统有着非常紧密的联系。在本书中，我们以 ARM 体系结构和嵌入式 Linux 操作系统为原型讲述 BootLoader 的原理。通过理论联系实践的方法论，让读者理解 BootLoader 的概念，掌握开发 BootLoader 的方法。本书适合从单片机向 ARM 过渡并希望了解嵌入式开发的在校学生以及想要从事 BootLoader 开发移植工作的工程师参考使用。

下面来梳理一下需要哪些背景知识：

1）因为我们引导的操作系统是Linux，所以需要熟悉Linux开发环境。BootLoader也是一串代码，我们必须了解Linux下编辑器和编译器的用法；作为Linux下的开发者，对shell脚本和Make工程管理工具也要有必要的了解。第1章对BootLoader做了基本介绍，第2章则详细介绍了Linux开发环境。

2）因为我们仅仅以ARM体系结构作为原型，所以会在第3～5章讲述ARM体系结构、ARM指令集和ARM独特的寻址模式。

3）很多嵌入式开发人员都是通信、电子和自动化等专业的，对于计算机的编译和链接掌握得不够深入，因此第6章介绍编译和链接。控制链接行为的链接脚本是特有的脚本语言，第7章比较详细地介绍了该脚本的细节。

4）第8章采用流水灯的实验融会贯通前面各章节的理论，接着第9章对U-Boot代码展开分析，最后在第10章一步一步地实现了BootLoader。在此过程中，我们更注重ARM体系结构、编译链接等知识的融会贯通，不会花太多篇幅讲解DRAM、MMC等模块的驱动。

本书能够成书，要感谢LinkSprite团队的资源支持和左宝柱在pcDuino v3平台上的帮助。最后感谢家人和朋友对我的支持，鼓励我完成这项值得用心去做的工作。因作者水平有限且时间仓促，书中难免存在疏漏，还望广大读者不吝赐教。

胡尔佳

2014年11月

试读结束：需要全本请在线购买： www.ertongbook.com

# 目　录


前言

第1章　BootLoader的概念 ········ 1

1.1　BootLoader的角色 ········ 1

1.2　BootLoader的来历 ········ 2

1.3　BootLoader的概念 ········ 6

1.3.1　MCU下的BootLoader ········ 10

1.3.2　嵌入式ARM和Linux下的BootLoader ········ 17

1.3.3　PC下的引导流程 ········ 18

1.4　本章小结 ········ 20

第2章　Linux开发环境 ········ 21

2.1　编辑器Vim ········ 21

2.1.1　Vim介绍 ········ 22

2.1.2　Vim的两个基本模式 ········ 22

2.1.3　Vim的两个常用模式 ········ 24

2.1.4　Vim的启动与退出 ········ 25

2.1.5　Vim下光标的移动 ········ 26

2.1.6　Vim下的复制、粘贴和删除 ········ 28

2.1.7　Vim下的撤销和重复 ········ 28

2.1.8　Vim下的查找和替换 ········ 28

2.1.9　Vim下的文件恢复 ········ 29

2.1.10　Vim下的插件 ········ 29

2.2　编译器GCC和交叉编译器 ········ 30

2.2.1　GCC的编译流程 ········ 30

2.2.2　GCC的常用编译选项 ········ 31

2.2.3　交叉编译器 ········ 35

2.3　常用shell命令和脚本 ········ 37

2.3.1　find命令 ········ 38

2.3.2　grep命令 ········ 42

2.3.3　管道与重定向 ········ 46

2.4　工程管理Make和Makefile ········ 49

2.4.1　Make和Makefile ········ 49

2.4.2　Makefile中的变量 ········ 52

2.4.3　自动推导规则 ········ 53

2.4.4　嵌套的Makefile ········ 53

2.4.5　Make伪目标 ········ 54

2.4.6　自动化变量 ········ 56

2.4.7　Make的内嵌函数 ········ 56

2.5　本章小结 ········ 60

第3章　ARM体系结构 ········ 61

3.1　处理器模式 ········ 61

3.2　异常 ········ 62

3.3　ARM寄存器 ········ 63

3.4　通用寄存器 ········ 64

3.4.1 未分组的寄存器：R0 ～ R7 …… 65
3.4.2 分组的寄存器：R8 ～ R14 …… 65
3.4.3 寄存器 R15：程序计数器 …… 66
3.5 程序状态寄存器 …… 66
3.5.1 PSR 位的类型 …… 66
3.5.2 条件标志位 …… 67
3.5.3 中断禁止位 …… 68
3.5.4 模式位 …… 68
3.6 本章小结 …… 68

第 4 章 ARM 指令集 …… 69

4.1 数据处理指令 …… 69
4.2 分支指令 …… 73
4.3 软中断指令 …… 74
4.4 程序状态寄存器指令 …… 75
4.5 协处理器指令 …… 76
4.6 加载常量的伪指令 …… 78
4.7 本章小结 …… 78

第 5 章 ARM 寻址模式 …… 79

5.1 寻址模式 1——数据处理指令的寻址模式 …… 79
5.1.1 编码格式 …… 80
5.1.2 移位器操作数 …… 80
5.1.3 数据处理操作——立即数 …… 81
5.1.4 数据处理操作——寄存器 …… 82
5.1.5 数据处理操作——立即数的逻辑左移 …… 83
5.1.6 数据处理操作——寄存器的逻辑左移 …… 84
5.1.7 数据处理操作——立即数的逻辑右移 …… 85
5.1.8 数据处理操作——寄存器的逻辑右移 …… 85
5.1.9 数据处理操作——立即数的算术右移 …… 86
5.1.10 数据处理操作——寄存器的算术右移 …… 87
5.1.11 数据处理操作——立即数的循环右移 …… 88
5.1.12 数据处理操作——寄存器的循环右移 …… 89
5.1.13 数据处理操作——扩展的循环右移 …… 90
5.2 寻址模式 2——字或无符号字节的 load/store 指令 …… 90
5.2.1 编码格式 …… 91
5.2.2 立即数偏移 …… 92
5.2.3 寄存器偏移 …… 93
5.2.4 比例寄存器偏移 …… 94
5.2.5 立即数的前变址寻址 …… 95
5.2.6 寄存器的前变址寻址 …… 96
5.2.7 比例寄存器的前变址寻址 …… 97
5.2.8 立即数的后变址寻址 …… 99
5.2.9 寄存器的后变址寻址 …… 100
5.2.10 比例寄存器的后变址寻址 …… 101
5.3 寻址模式 3——杂类 load/store 指令的寻址方式 …… 103
5.3.1 编码格式 …… 103
5.3.2 杂类 load/store——立即数偏移 …… 104

5.3.3 杂类 load/store——寄存器偏移 ········ 105
5.3.4 杂类 load/store——立即数的前变址寻址 ········ 106
5.3.5 杂类 load/store——寄存器的前变址寻址 ········ 107
5.3.6 杂类 load/store——立即数的后变址寻址 ········ 108
5.3.7 杂类 load/store——寄存器的后变址寻址 ········ 109
5.4 寻址模式 4——批量 load/store ··· 110
5.4.1 编码格式 ········ 110
5.4.2 批量 load/store——执行后增加 ········ 111
5.4.3 批量 load/store——执行前增加 ········ 112
5.4.4 批量 load/store——执行后减少 ········ 112
5.4.5 批量 load/store——执行前减少 ········ 113
5.4.6 用于栈操作的批量 load/store ········ 114
5.5 寻址模式 5——协处理器的 load/store ········ 115
5.5.1 编码格式 ········ 115
5.5.2 协处理器的 load/store——立即数偏移 ········ 116
5.5.3 协处理器的 load/store——立即数的前变址寻址 117
5.5.4 协处理器的 load/store——立即数的后变址寻址 118
5.5.5 协处理器的 load/store——无索引 ········ 119
5.6 本章小结 ········ 119

第 6 章 编译和链接 ········ 120

6.1 ELF 文件结构描述 ········ 122
6.2 段表 ········ 128
6.3 符号表结构 ········ 133
6.4 存储空间分配 ········ 138
6.4.1 简单的存储布局 ········ 138
6.4.2 实际采用的空间布局 ······ 139
6.5 重定位信息 ········ 140
6.5.1 重定位表项 ········ 140
6.5.2 重定位类型 ········ 142
6.6 静态链接和重定位 ········ 143
6.6.1 符号和符号表 ········ 144
6.6.2 符号解析 ········ 145
6.6.3 重定位 ········ 146
6.7 本章小结 ········ 150

第 7 章 链接脚本 ········ 151

7.1 链接脚本的基本概念 ········ 152
7.2 链接脚本格式 ········ 152
7.3 简单的链接脚本示例 ········ 153
7.4 简单的链接脚本命令 ········ 154
7.4.1 入口点 ········ 154
7.4.2 文件命令 ········ 155
7.4.3 格式命令 ········ 156
7.4.4 区域别名 ········ 157
7.4.5 杂类命令 ········ 159
7.5 为符号分配值 ········ 160
7.5.1 简单的分配 ········ 161
7.5.2 HIDDEN ········ 161
7.5.3 PROVIDE ········ 162

7.5.4 PROVIDE_HIDDEN ······ 162

7.5.5 源代码引用 ······ 162

7.6 段命令 ······ 164

7.6.1 输出段描述 ······ 164

7.6.2 输出段名称 ······ 165

7.6.3 输出段地址 ······ 165

7.6.4 输入段描述 ······ 166

7.6.5 输出段数据 ······ 171

7.6.6 输出段关键字 ······ 172

7.6.7 输出段丢弃 ······ 172

7.6.8 输出段属性 ······ 172

7.6.9 覆盖描述 ······ 175

7.7 内存命令 ······ 177

7.8 链接脚本的表达式 ······ 178

7.8.1 常量 ······ 178

7.8.2 符号常量 ······ 179

7.8.3 符号名称 ······ 179

7.8.4 孤立的段 ······ 179

7.8.5 位置计数器 ······ 180

7.8.6 运算符 ······ 182

7.8.7 赋值 ······ 183

7.8.8 内建函数 ······ 183

7.9 本章小结 ······ 187

**第 8 章 Linux 下开发流水灯 ···· 188**

8.1 GNU ARM 汇编简介 ······ 188

8.2 流水灯的硬件描述 ······ 191

8.3 流水灯的汇编实现 ······ 193

8.4 流水灯的编译和链接 ······ 195

8.5 本章小结 ······ 196

**第 9 章 U-Boot 代码的分析 ····· 197**

9.1 U-Boot 简介 ······ 197

9.2 U-Boot 目录结构 ······ 198

9.3 U-Boot 配置和编译 ······ 200

9.4 U-Boot 代码分析 ······ 207

9.4.1 SPL 代码追踪 ······ 208

9.4.2 U-Boot 代码追踪 ······ 233

9.5 本章小结 ······ 245

**第 10 章 实现简单的 BootLoader ······ 246**

10.1 STM32 下的 BootLoader 设计 ······ 246

10.2 硬件平台 pcDuino 简介 ······ 251

10.2.1 pcDuino nano 配置 ······ 252

10.2.2 pcDuino nano 的接口和外设 ······ 253

10.2.3 平台和主芯片介绍 ······ 254

10.3 三种方式实现代码复制和跳转 ······ 255

10.3.1 方式一 ······ 255

10.3.2 方式二 ······ 258

10.3.3 方式三 ······ 259

10.4 实现 BootLoader ······ 261

10.4.1 广义上的 BootLoader ····· 261

10.4.2 如何引导 Linux ······ 264

10.4.3 引导代码实现 ······ 266

10.4.4 BootLoader 引导 Linux 总结 ······ 267

10.5 本章小结 ······ 267

# 第 1 章 BootLoader 的概念

**本章导读**

本书讲述的主题是 BootLoader，那么 BootLoader 是什么呢？首先从字面上来看，BootLoader 是一个英文单词，那它是什么意思呢？如果使用词典查询一下，会得到这样的结果：“对不起，词库中没有您查询的单词！您要查找的是不是：Boot Loader Boot-Loader” Boot Loader [ 计 ] 引导装载程序。

可以看到 BootLoader 应该是 Boot 和 Loader 这两个单词的组合。Boot 的意思是“[ 计算机科学 ] 引导”，而 Loader 是“加载器”的意思。因此可以推断 BootLoader 这个词是随着计算机科学的发展而衍生出现的。

1.1 节介绍 BootLoader 在嵌入式系统或者计算机系统中所扮演的角色。

1.2 节结合半导体芯片技术、计算机技术以及软件工程技术的发展历史来探究 BootLoader 的出现过程，从而说明 BootLoader 的来历和它在整个计算机系统中的定位、概念以及作用。

1.3 节阐述 BootLoader 的概念，并以 MCU 下的 BootLoader、嵌入式 ARM 和 Linux 下的 BootLoader 和 PC 下的引导流程为例进行说明。

## 1.1 BootLoader 的角色

当一个嵌入式开发板上电时，哪怕执行最简单的程序，都要初始化非常多的硬件。每种体系结构、处理器都有一组预定义的动作和配置，它们包含从单板的存储设备获取初始化代码的功能。最初的初始化代码是 BootLoader 的一部分，它负责启动处理器和相关硬件设备。

在上电复位时，大多数处理器都有一个获取第一条执行指令的默认地址。硬件设计人员利用该信息来进行存储空间的布局。这样一来，上电的时候可从一个通用的已知地址获取代码，然后建立软件的控制。

BootLoader 提供最初的初始化代码，并初始化单板，这样就可以执行其他的程序。最初的初始化代码都是由该处理器体系结构下的汇编语言写成。当然，在 BootLoader 已经执行完基本的处理器和平台的初始化之后，它的主要工作就是引导完整的操作系统。它将定位、加载操作系统，并将控制权移交给操作系统。另外，BootLoader 可能含有一些高级特性，比如校验 OS 镜像、升级 OS 镜像、从多个 OS 镜像中选择性引导。与传统的 PC-BIOS 不同，当操作系统获取控制权后，嵌入式下的 BootLoader 就不复存在了。

## 1.2 BootLoader 的来历

我们从 4 个角度来阐述 BootLoader 的来历，第一个是从半导体芯片技术（特别是处理器的发展）来看 Boot 的出现：

在集成电路只读存储器出现之前，早期的计算机 ENIAC 在存储中并没有程序，而是通过连接电缆的配置来解决问题。ENIAC 中并没有自举电路或引导程序，因为它只要上电，其硬件配置就开始解决问题。在早期的计算机中，根本就没有 BootLoader 的概念，连 Boot 这个名称都没有。

稍晚时间出现的 IBM 701 计算机（1952 ～ 1956 年）有一个“Load”按键，按下它，可以从外部储存中加载最初的 36 位的字到主存中。有一个加载选择开关来决定外部储存是穿孔卡片、磁带，还是磁鼓（看到这里是不是觉得它和现代的嵌入式系统有些类似，在有些嵌入式电路板中，同样用一些拨码开关来控制电路板从 SD 卡、Flash 等多种存储设备中选择某一种存储设备进行启动）。接下来的 18 位半字作为一条指令执行，它通常会读取更多的字到主存中。这时开始执行 Boot 程序，它将依次从外部存储媒质加载更大的程序到主存中。在 1958 年，“Boot”这个计算机名词便开始使用了。请读者注意，这里的“ Load”按键和 Loader 是有本质区别的：“ Load”是通过外部操作来进行加载的，而 Loader 是软件实现的加载器，它可以自行完成加载的动作。

Boot 这个词的原意是靴子，那靴子怎么会与计算机系统中的引导发生关联呢？

试读结束：需要全本请在线购买： www.ertongbook.com

原来，这里的Boot是Bootstrap（鞋带）的缩写，它来自于一句谚语："Pull oneself up by one's own bootstraps"，直译的意思是"拽着鞋带把自己拉起来"，这当然是不可能的事情。最早的时候，工程师用它来做比喻，比如自举电路振荡器（Bootstrap Generator）就是指，不外加激励信号而自行产生恒稳和持续的振荡。对于早期计算机的启动，也存在这样一个问题：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序。于是，当时的人们想尽各种办法，他们把一小段程序装进内存，之后计算机就能正常运行了。因此，工程师们将这个过程称为"拉鞋带"，久而久之就简称为Boot了。而且随着处理器的发展，Boot处理的事务越来越多，比如对CPU运行模式的设置，比如对CPU内部时钟的配置，还包括对Cache和MMU的配置。

第二个是从存储技术的发展来看Loader的出现：

首先介绍两类半导体存储器：ROM和RAM。ROM是Read Only Memory的缩写，RAM是Random Access Memory的缩写。ROM在系统停止供电的时候仍然可以保持数据，而RAM通常在掉电之后就丢失数据，但是其存储单元的内容可按需随意取出或存入，且存取的速度与存储单元的位置无关。

ROM分为ROM、PROM、EPROM和EEPROM等。只读存储器（ROM）是只能读取资料的内存。其资料内容在写入后就不能更改。此种内存的制造成本极低，常用于电脑中的开机启动。可编程只读存储器（PROM）一般可编程一次。PROM存储器在出厂时各个存储单元皆为1，或皆为0，用户在使用时，再采用编程的方法使PROM存储所需要的数据。可擦除可编程存储器（EPROM）可多次编程，这是一种便于用户根据需要来写入，并能把已写入的内容擦去后再改写，即为一种多次改写的ROM。由于能够改写，因此能对写入的信息进行校正，修改错误后再重新写入。电子可擦除可编程只读存储器（EEPROM）的运作原理类似于EPROM，但是擦除的方式更加方便。

RAM按照存储单元的工作原理分为静态随机存储器（SRAM）和动态随机存储器（DRAM）。SRAM的存取速度要比DRAM快，同时价格也更高。在计算机中SRAM常用来作为Cache，而DRAM常作为内存来使用。

闪存（Flash Memory）是一种高密度、非易失性的读/写半导体存储器。它既有EEPROM的特点，又有RAM的特点，是一种全新的存储结构。Nor Flash和Nand

Flash 是市场上两种主要的闪存技术。Intel 公司于 1988 年首先开发出 Nor Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 Nand Flash 结构，强调降低每比特的成本，提供更高的性能，并且 Nand Flash 像磁盘一样可以通过接口轻松升级。Nor Flash 和 Nand Flash 有几个重要的区别：Nor Flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取内部的每一个字节，可以做到芯片内执行（XIP），应用程序可以直接在 Nor Flash 内运行。Nand Flash 使用复杂的 I/O 口来串行地存取数据，并且读写操作都是块操作，因此它无法做到 XIP。Nor Flash 的读取速度比 Nand Flash 稍快，但是写入速度却要慢很多。Nand Flash 的存储密度更高，成本更低。因此，很多嵌入式单板都使用小容量的 Nor Flash 来运行启动代码，而使用 Nand Flash 存储其他信息。

假如说当时已经有一种既可以运行程序，而且价格低廉、存储空间大的非易失性存储的设备，那么我们的所有程序都可以存放在此类存储设备中，这样就不存在加载的问题了。然而实际上，一般的计算机系统都使用多级存储器层次结构，既有存储空间大、价格低廉、运行速度慢的非易失性存储设备（磁盘），也有空间小、价格高、运行速度略快的非易失性存储设备（PROM 等），还有价格高，运行速度非常快的易失性存储设备（RAM）。这样的层次结构会保证硬件最佳的性价比，然而从软件上，我们必须引入 Loader。它将自动完成从非易失性存储设备向易失性存储设备加载程序的功能。

在现实的发展过程中，随着集成电路只读存储器时代的到来，由于集成电路技术的发展和 ROM 的出现（包括可掩码编程的 ROM、可编程的 ROM（PROM）、可擦除可编程的 ROM 和 Flash 存储器），引导流程发生了极大的改变。它们使得固件引导程序开始安装到计算机中。

通常情况下，在上电和复位的条件下，每个微型处理器会执行下面两种可能的启动流程：①开始执行从某一特定地址开始的代码；②在特定地址处查询代码，然后跳转到指定地址，开始执行代码。运用这种微处理器的系统使用 ROM 来保存这些特定的位置，如此，系统就不需要操作人员的干预就能开始运转。举个例子，Intel x86 处理器一直从位于 FFFF：0000 的指令处开始运行，而 MOS 6502 处理器，读取位于 $FFFD（MS byte）和 $FFFC（LS byte）的两个字节长度的向量地址，然后跳转到该地址开始运行引导程序。

Apple 的第一代计算机（1976 年推出的 Apple1）使用了 PROM 芯片而不再需要前

面板的配置。因此当时 Apple 的宣传广告是“ No More Switches, No More Lights ... the firmware in PROMS enables you to enter, display and debug programs（all in hex）from the keyboard”。

由于当时只读存储器的造价很高，Apple II 系列采用多种存储器来构建存储器系统，以保证性能和价格的最优化。它使用一串小步骤来引导磁盘上的操作系统，每一个小步骤都将控制权递交到愈加复杂的引导流程中。我们可以预见，每个步骤都必须能控制下一个步骤所用的存储器的读取操作，从而才能将其内容加载到特定的位置以执行，这样就有了 Loader（加载器）的概念。结合 Boot 和 Loader 的概念，BootLoader 是既能“自举”又能“举人”的一段程序，由此，BootLoader 在整个计算机系统中可以说起着举足轻重的作用，任何一个计算机系统都需要一个稳定强大的 BootLoader。

第三个是从软件语言的发展（特别是 C 语言的出现）来看 BootLoader 的发展：

最早的计算机用于读取纸带的数据，上面记录着二进制格式的机器指令，计算机每读一条机器指令就执行一条机器指令。机器指令繁多，而且不适合人来阅读和使用，所以后来出现了汇编语言，汇编语言实质上是机器语言的助记符。汇编语言中出现了函数的概念，那么在编写汇编代码时，就需要对栈进行小心的设计，并注意保护好上下文现场。再往后，出现了 C 语言，C 语言是一种更抽象的语言，它是针对于程序员的语言，隔离了许多底层上的概念，比方说隔离函数调用栈的设计。并且 C 语言无法完成很多底层操作，因为只有汇编语言才能使用芯片架构上的一些特殊的指令。因此对于 BootLoader 来说，它是由汇编和 C 语言混合编码的，Boot 中的汇编代码完成只能用汇编指令完成的操作，并提供 C 语言的运行环境，还需要对栈指针和堆空间进行初始化和分配。C 语言的出现无疑加快了 BootLoader 的设计和开发，从而促进了 BootLoader 的发展。

第四个是从软件工程的角度来看 BootLoader 的发展：

最初的计算机软件统称为 Firmware，而随着软件的复杂度提高，操作系统的出现，我们相应地将 BootLoader 独立出来，并作为专门的一个固件。BootLoader 和操作系统分开设计，使得各自的扩展性、跨平台、可移植性都大大提高。另外从软件工程的模块化设计、分层设计来讲，像 UBoot 这样的 BootLoader 的设计越来越像 Linux 了，其代码布局和 Linux 十分类似，使用了按照模块划分的结构，并且充分考虑了体系结构和跨平台问题。

## 1.3 BootLoader 的概念

这一节我们全面地介绍 BootLoader 的基本概念。

简单地说，BootLoader 就是在操作系统内核运行前执行的一段小程序。通过这段小程序，可以对硬件设备，如 CPU、SDRAM、Flash、串口等进行初始化，也可以下载文件到系统板、对 Flash 进行擦除和编程，真正起到加载和引导内核镜像的作用。但是随着嵌入式系统的发展，BootLoader 已经逐渐在上述基本功能的基础上进行了扩展，BootLoader 可以增加更多的对具体系统的板级支持，即增加一些硬件模块功能上的使用支持，以方便开发人员进行开发和调试。

从这个层面上看，功能扩展后 BootLoader 可以虚拟地看成一个微小的系统级的代码包。BootLoader 是依赖于硬件实现的，特别是在嵌入式系统中。不同的体系结构下，对 BootLoader 的需求是不同的；除了体系结构，BootLoader 还依赖于具体的嵌入式板级设备的配置。也就是说，对于两块不同的嵌入式电路板而言，即使它们基于相同的 CPU 构建，运行在其中一块电路板上的 BootLoader，都未必能够运行在另一块电路板上。

当现代的计算机关机时，它的软件，包括操作系统、应用程序和数据，都会存放在非易失性存储设备上，比如硬盘、CD、DVD、Flash 存储卡（SD 卡）、USB 盘和软盘。当计算机上电时，在它的 RAM 中是没有操作系统和加载器的。计算机会首先执行存放在 ROM 中的相对较小的程序，读取小容量的必需数据，这样才能访问非易失性存储设备中的操作系统和数据，并将其加载到 RAM 中。

这个过程中的小程序被称为“Bootstrap Loader”“Bootstrap”或“Boot Loader”。这段小程序的目的是加载其余的数据和程序到 RAM 中并开始执行。若使用多阶段的 Boot Loader，它们会一个接一个的加载。

有一些计算机系统，当从操作人员或者外设接到一个引导信号后，会加载少量的固定指令到内存中的特定地址，初始化至少一个 CPU，然后让 CPU 开始执行这些指令。这些指令通常会对一些外围设备做输入操作。有一些系统会向外围设备或者 I/O 控制器直接发送硬件命令来进行非常简单的输入操作，加载一小部分 Boot Loader 指令到内存中，I/O 设备的一个信号会通知 CPU 开始执行指令。

小型的计算机经常使用不那么灵活但是更自动化的 Boot Loader 机制来保证计算

机在预定的软件配置下能够快速启动。在很多桌面型计算机中，比如，在 ROM（IBM PC 的 BIOS）的预定义地址中有一软件（有一些 CPU，包括 Intel 的 x86 系列，在复位后无需外部的辅助就可以执行这个软件），这个软件有查找 booting 过程中用到的设备的基本功能，然后从设备的特定部分（通常是 boot sector）加载一个小程序。

Boot Loader 会有一些特殊的约束，特别是大小。比如，在 IBM PC 和其兼容机上，引导扇区只能是系统存储中的 32KB（后来扩充到 64KB），而且不能使用 8088/8086 处理器不支持的指令。

BootLoader 的启动过程可以是单阶段的，也可以是多阶段的。通常多阶段的 BootLoader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 BootLoader 大多数是二阶段的启动过程，即启动过程可以分为 stage1 和 stage2 两部分。

第二阶段的 BootLoader，比如 GNU GRUB、BOOTMGR、Syslinux、NTLDR 或者 BootX，能够正确加载操作系统并移交控制权；接着操作系统会进行一系列初始化并可能加载额外的设备驱动。第二阶段的 Boot Loader 在自己的操作过程中不需要驱动，但是需要由系统固件（比如 BIOS 或者 Open Firmware）提供通用存储访问方法，然而这样会限制硬件的功能，降低其性能。

很多 Boot Loader（比如 GNU GRUB、Window 的 BOOTMGR 和 Windows NT/2000/XP 的 NTLDR）能够为用户提供多个引导选项。这些选项可以是多个不同的操作系统（不同分区或者不同驱动器上的多个引导），也可以是同一操作系统的不同版本，还可以是同一操作系统的不同加载选项（比如说进入恢复模式或者安全模式），甚至可以是一些不是操作系统的独立程序，比如内存测试程序（memtest86+）。有一些 BootLoader 会加载其他的 BootLoader，比如 GRUB 不会直接加载 Windows，而是加载 BOOTMGR。通常有一个倒计时选择可以让用户通过按键来进行选择。倒计时结束后，默认的选择是自动运行正常的引导程序。

当计算机能够与用户交互时，或者操作系统能够运行系统程序和应用程序时，引导流程可以看成结束了。通常现代的个人计算机在 1min 以内完成引导，其中 15s 用于上电自检和初步引导，剩下的时间用于加载操作系统和其他软件。操作系统加载完成后花费的时间可以缩短至 3s，仅仅只是启动核心的服务。而大型服务器可能需要数分钟来引导和启动服务。

很多嵌入式系统必须非常迅速地完成引导。比如说，花一分钟来等待数字电视或者 GPS 设备的启动是不被用户接受的。因此这类设备在 ROM 或者 Flash 中有特殊的软件系统，使得设备更快速地启动，几乎不需要加载，因为在生产设备的时候加载的内容已预先存在 ROM 中了。

其他类型的引导顺序如下。

一些现代的 CPU 和微处理器（例如 TI OMAP），甚至一些 DSP 在它们的芯片中含有已经烧录好引导代码的引导 ROM，那么这样的处理器能够执行相当复杂的引导流程，比如它能够从 Nand Flash、SD 或 MMC 卡等多种存储设备引导。由于很难用电路逻辑来处理这些设备，那么这种情况下就使用集成的引导 ROM。引导 ROM 能够提供比硬件逻辑更灵活的引导顺序。比如，引导 ROM 能够试着从多个引导源设备执行引导过程。另外，引导 ROM 也能够从串口，比如 UART、SPI 或者 USB 等接口来加载 BootLoader 或者诊断程序。当存放在非易失性存储中的引导程序被擦除时，这个特性可以用来恢复系统，而且当系统中的非易失性存储中没有程序可用时，也可以对非易失性存储进行编程。

有些嵌入式系统可能包含中间的引导流程步骤，由内建的引导 ROM 加载到系统 RAM 中。由于平台的限制，比如说 RAM 大小的限制，需要加一个中间的引导流程步骤，比如 Das U-Boot，它就有 SPL 的架构：由 SPL 引导 U-Boot，接着 U-Boot 引导系统。

我们有时会使用硬件的调试接口，比如 JTAG（JTAG 是一个标准的流行接口，许多 CPU 和微处理器都带有 JTAG 接口）来控制系统。这类调试接口可以用于向可引导的非易失性存储设备中烧写 Boot Loader 代码。另外，这类调试接口可以用于加载一些诊断或者引导代码到 RAM 中，并且使得处理器开始执行代码。当可引导的存储设备没有程序时，它可以用于恢复嵌入式系统，也可以用于不含有内建引导 ROM 的处理器。

有些微处理器包含不能获取系统控制权或者直接运行代码的特殊硬件接口，但是它们通过一些简单的协议可以将引导代码注入非易失性存储设备中。那么在生产过程中，使用这类接口向非易失性存储中注入引导代码。当系统复位后，微处理器开始执行非易失性存储中的代码，就像使用 ROM 引导一样。Atmel AVR 微处理器就使用了该项技术。很多情况下这类接口都是由硬件逻辑设计的。另外的一些情况下，这类接口可以通过配置 GPIO 运行内建的引导 ROM 中的代码来创建。

试读结束：需要全本请在线购买： www.ertongbook.com