深入理解嵌入式Linux设备驱动程序

曹国辉　曾志鹏　著

站在驱动开发工程师的角度，以实际开发板硬件为基础，循序渐进地讲述了嵌入式Linux驱动程序开发的方法

不但讲解了怎么做，更重要的是分析了为什么要这么做，在分析Linux内核驱动源码的基础上，提炼出Linux驱动架构模型的核心思想及实现思路

在理解整个Linux设备驱动流程和实现思路的基础上，阐述了编写设备驱动和移植设备驱动的基本思路和方法

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深入理解嵌入式 Linux 设备驱动程序

曹国辉　曾志鹏　著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

前　言

随着计算机技术及相关技术的发展，嵌入式技术已经在通信、网络、工控、航空航天、医疗电子等领域得到了广泛的应用。近些年来，三网融合、智能电网、物联网等战略性新兴产业的快速发展也为嵌入式产业带来巨大的发展机遇。而嵌入式操作系统，作为驱动嵌入式系统硬件的灵魂，在嵌入式技术中占据着极其重要的地位。

智能化、网络化是当今电子设备发展最重要的趋势。这种趋势给设备软件带来的最大挑战就是开发成本和复杂度激增，对于任何一家企业，使用自己的嵌入式操作系统根本不可能适应市场竞争的需求和压力。与此同时，Linux 正在成为嵌入式软件平台的重要选择，由于免费且开放源代码，不仅显著降低了嵌入式操作系统的使用成本，而且可以从广泛的开放源代码社区获得丰富的资源。因此越来越多的嵌入式企业开始选择 Linux 系统作为公司产品的软件平台。

但是，Linux 因为其开源特性，使得开放的源码质量难以进行控制，同时相关的开发工具也没有商业操作系统便利，这就对嵌入式 Linux 开发人员的技术水平提出了更高的要求。嵌入式驱动程序开发是嵌入式软件开发中的最底层，与硬件密切相关，也是最难的一部分，因此，广大嵌入式开发人员迫切需要一本详细讲解如何进行嵌入式 Linux 驱动开发的教程，本书正是为了满足广大嵌入式开发人员这一需求而编写的。

本书站在一个驱动开发工程师的角度，以实际开发板硬件为基础，循序渐进地详细讲述了嵌入式 Linux 驱动程序开发的方法。全书主要分为三大部分：

第一部分（1～4 章）阐述了在没有操作系统情况下嵌入式系统的硬件体系架构，裸机程序的启动运行过程，BootLoader 的功能及 U-boot 引导系统的过程。

第二部分（5～8 章）重点讲解了嵌入式 Linux 设备驱动开发基础及基本理论，包括嵌入式 Linux 模块编程、Linux 设备及设备驱动模型、Platform 平台设备驱动、设备驱动程序中常用的并发控制，阻塞与非阻塞，轮询、异步通知等基本 Linux 设备驱动程序开发核心技术。

第三部分（9～15 章）详细分析了常用的硬件设备驱动程序，包括 LCD 屏设备驱动、触摸屏设备驱动、网卡设备驱动、I^2C 设备驱动、USB 设备驱动等。这是对 Linux 设备驱动程序架构模型及基本理论的具体应用。

全书思路清晰，环环相扣，融为一体。

本书的前身是南京英贝得嵌入式培训中心的高级班嵌入式 Linux 设备驱动班的讲义，在编写时，立足于实践，理论和实践相结合。书中所有提到的相关程序，笔者都亲自在电脑里调试运行过，在阐述每一章节时，遵循提出问题、分析问题、解决问题的思路，以问题为中

心，一步步分析 Linux 设备驱动程序架构，分析 Linux 设备驱动架构是如果解决现实中的问题及其优缺点。在阐述每一个具体设备驱动程序时，遵循从硬件原理分析、无操作系统的驱动程序、Linux 下的驱动程序的思路进行讲解分析，符合读者接受新知识的思维。

本书的特色主要在于不但详细讲解了怎么做，更重要的是分析了为什么要这么做，在分析 Linux 内核驱动源码的基础上，提炼出了 Linux 驱动架构模型的核心思想及实现思路；基于对每个具体设备的 Linux 设备驱动源码的分析，详细阐述了整个设备驱动程序的工作流程及实现思路，在理解整个 Linux 设备驱动流程和实现思路的基础上，阐述了编写设备驱动和移植设备驱动的基本思路和方法。

全书是笔者多年项目开发和教学经验的总结，可作为高校嵌入式专业教材以及广大嵌入式爱好者的参考书。

本书由曹国辉和曾志鹏共同完成，第 1、8、10、11、12、13、14、15 章由曹国辉编写，第 2、3、4、5、6、7、9 章由曾志鹏编写，全书由曹国辉整理定稿。

本书在编写过程中引用了一些互联网上的文献资料，无法一一注明出处，在此向原作者表示感谢。由于笔者水平有限，书中难免存在错误与不妥之处，欢迎广大读者朋友不吝赐教。

联系邮箱：guohuicao@126.com。

本书源代码可在电子工业出版社官网“在线资源”中下载。

曹国辉　曾志鹏

目录

Contens

第 1 章
嵌入式 ARM 系统开发基础

本章目标

- 掌握嵌入式 ARM 集成开发环境 ADS1.2 的使用方法
- 掌握嵌入式 ARM 应用系统的启动过程
- 掌握嵌入式 ARM 裸机中断处理过程
- 理解 ARM 应用程序加载地址和运行地址的概念
- 掌握嵌入式 ARM 应用系统的基本程序设计方法

1.1 嵌入式 ARM 系统开发概述

嵌入式 ARM 系统开发是指在 ARM 裸机上进行开发，在 ARM 上没有“跑”任何操作系统和驱动，全部由自己编程实现，相当于把 ARM 当做高级单片机来使用。下面介绍嵌入式 ARM 系统开发的一些基础知识。

1.1.1 ARM 系统可执行映像文件格式

ARM 系统的可执行映像文件格式主要有 ELF（.elf）、AXF（.axf），和 BIN（.bin），下面对这三种映像格式文件分别进行介绍。

ELF：Linux 操作系统下可执行映像文件格式，在 Linux 环境下用 GCC 编译器生成的可执行映像文件格式即为 ELF 格式，在 Linux 操作系统下可直接运行。

AXF：ARM 的调试文件，由 ARM 集成开发工具 ADS 生成，除了包括可执行代码外，还包括其他调试信息，用于 ADS 调试。

BIN：真正的可执行文件，包括 ARM 可执行的指令和数据，可以使用相关工具（Objcopy）从 ELF 文件中生成，写到 FLASH 或 RAM 中可直接运行。

ARM 中的各种源文件（包括汇编文件、C 语言程序及 C++程序等）经过 ARM 编译器编译后生成 ELF(Executable and Linking Format)格式的目标文件。这些目标文件和相应的 C/C++运行时所用到的库经过 ARM 连接器处理后，生成 ELF 格式的映像文件（Image），这种 ELF

格式的映像文件是一种可执行文件。

BIN 文件是真正的可执行文件。AXF 文件是 ARM 的调试文件，除了包含 BIN 的内容之外，还附加了其他的调试信息。这些调试信息加在可执行的二进制数据的前面，所以把 AXF 文件写到 ARM 的指令执行地址（一般是 0x0）将不能运行；因为在此地址前几十个字节的数据不是可执行的二进制数据，而是头部的调试信息。而 BIN 文件正是去掉了调试信息的可以执行的“精华”部分。

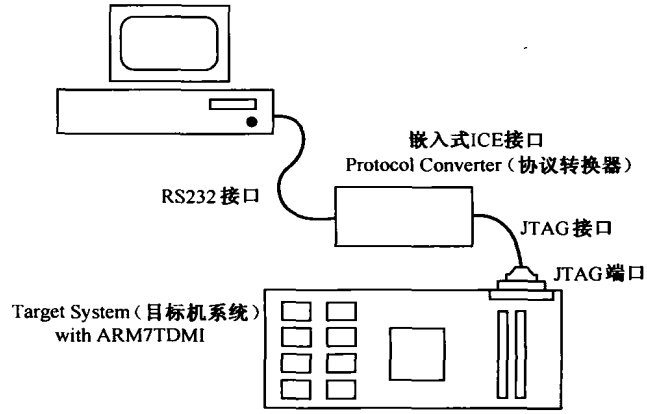
可执行映像文件主要分为 3 个段，即 RO 段、RW 段和 ZI 段，如下图所示。

- RO 段：只读代码段；
- RW 段：RW 区域放的是已赋值（赋 0 除外）的全局变量；
- ZI 段：ZI 区域放的是未赋值的全局变量或初始化为 0 的全局变量。

1.1.2 ARM 开发调试工具

ARM 公司为 ARM 系统的开发提供了图形化的集成开发环境——ADS，目前使用的版本是 ADS1.2；ADS 提供了项目代码管理、编辑、编译等功能。关于 ADS 的使用，读者可参考其他相关资料，这里主要介绍一些 ARM 的调试系统框架及基本原理。

ARM 芯片本身提供了在线调试功能，这使得我们可以在线进行实时调试，不需要再像其他单片机那样使用仿真器进行开发调试。ARM CPU 提供了 JTAG 接口，通过 JTAT 接口我们可以给 ARM 发送调试命令，实现访问 ARM 内部的指令寄存器和数据寄存器，暂停程序的执行等调试功能。同时通过 JTAG 接口，我们可以烧写程序到系统的 Nor FLASH 或 Nand FLASH 上。ARM 的在线调试功能正是通过 JTAG 口来实现的，典型的 ARM 系统调试框图如下图所示。



典型的 ARM 调试系统主要分为三部分：调试主机、协议转换器（调试代理）、目标机系统（调试目标）。调试主机一般是一台运行调试软件（如 ADS）的计算机。调试主机可以发出一些高层的调试命令，如设置断点、访问内存等。

协议转换器（如 MULTI-ICE）用来将调试主机发出的高层调试命令转换为底层的 ARM JTAG 调试命令。

调试目标一般就是指基于 ARM 内核 MCU 目标开发板。经过协议转换器进行命令解释，主机上运行的调试软件就可以通过 JTAG 接口直接和 ARM 内核对话。通过扫描链，可以把 ARM/THUMB 指令插入到 ARM 的指令流水线当中去执行。通过插入特定的 ARM 指令，我们可以检查、保存或者改变内核和系统的状态。为了支持底层的调试，ARM 处理器提供了硬件上的调试扩展。这些调试扩展包括：

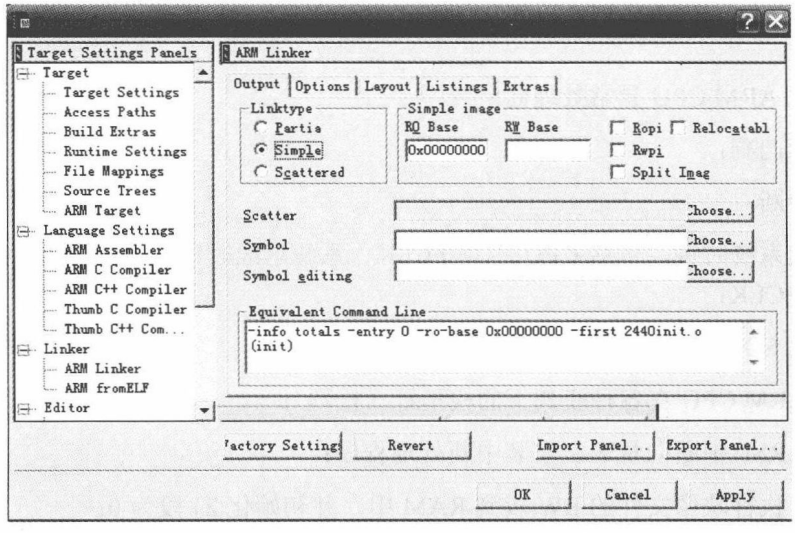
- 停止程序的运行；
- 检查和修改 ARM920T 的内核状态；
- 观察和修改内存；
- 恢复程序的运行。

1.1.3 加载地址和运行地址

由于 ARM 系统开发的应用程序在运行和没有运行时所在的位置可能不一样，例如，应用程序没有运行时存放在 Nand FLASH 中，运行时被搬到 RAM 中执行，所以就引出了加载地址（应用程序的加载）和运行地址（应用程序的运行）的概念。

- 加载地址：映像文件位于存储器（还没有运行，一般在 ROM 中，也可以在 RAM 中）时的地址；
- 运行地址：映像文件运行时的地址。

应用程序的加载和运行地址在应用程序链接时通过链接选项指定，如下图所示。



看看 ADS 开发文档 ARM Developer suite 1.2 中的 ADS_CodeWarriorIDEGuide.pdf 怎么说的：

RO Base This text field sets both the load address and execution address of the region containing the RO section. If you do not enter a value, the value defaults to 0x8000。

从 ADS 的帮助文档知道，RO Base 设置的是加载地址和运行地址，在这里加载地址和运行地址是一致的，通过 ADS 调试裸机程序时，映像文件首先要加载到目标板的内存中。

ADS 链接器预定义如下变量来表示应用程序运行时的地址，ADS 的预定义变量和含义分别如下：

- |Image$$RO$$Base| ：RO 段起始地址；
- |Image$$RO$$Limit| ：RO 段结束地址加 1；
- |Image$$RW$$Base| ：RW 段起始地址；
- |Image$$RW$$Limit| ：RW 段结束地址加 1；
- |Image$$ZI$$Base| ：ZI 段起始地址；
- |Image$$ZI$$Limit| ：ZI 段结束地址加 1。

1.2 嵌入式 ARM 系统的启动代码分析

1.2.1 ARM 上电启动概述

ARM 系统上电启动后，从 0x0 地址处开始执行，根据系统的配置 0x0 地址可以映射在 Nor FLASH 或 SDRAM 中，在 Nor FLASH 或 Nand FLASH 中存放有系统的启动初始化程序，与计算机的 BIOS 类似，完成系统最底层的初始化工作。

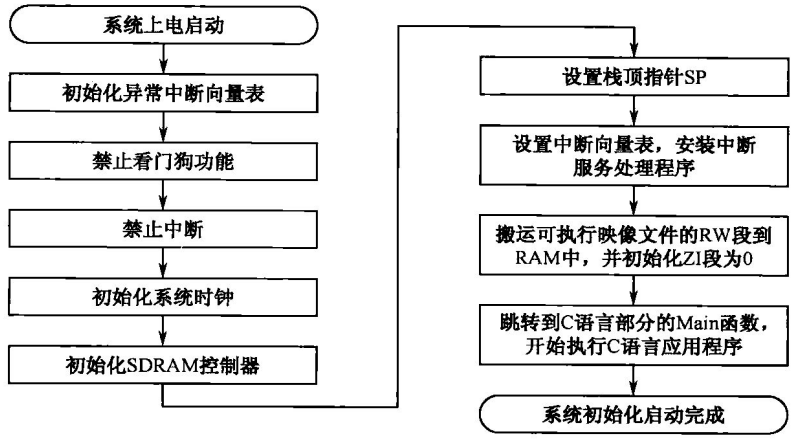
ARM 系统上电后，首先就运行系统初始化程序，系统初始化程序主要完成系统最基本的硬件初始化，为后面的 C 语言应用程序提供运行环境。ARM 系统初始化启动代码完成的主要功能如下：

- 初始化 ARM CPU 异常处理向量表；
- 禁止看门狗；
- 禁止中断；
- 初始化系统时钟，包括 CPU 主频 FCLK、系统总线时钟频率 HCLK、外设总线时钟频率 PCLK；
- 初始化 SDRAM 控制器；
- 设置 ARM CPU 在各种模式下的栈指针（栈顶）；
- 设置 ARM 中断向量表，安装中断处理程序；
- 搬运可执行映像文件的 RW 段到 RAM 中，并初始化 ZI 段为 0；

- 跳转到 C 语言应用程序的 Main 函数，开始执行 C 语言应用程序。

到此，系统的初始化启动程序就完成了 ARM 系统的启动过程。

ARM 系统的初始化启动代码一般用汇编语言编写，根据以上的分析，我们知道 ARM 系统初始化启动程序的流程如下图所示。



1.2.2 ARM 上电初始化启动代码分析

ARM 系统的初始化启动代码源文件为 2440init.s，是汇编语言源文件。下面对汇编启动代码的主要部分进行分析讲解。

```
GET     option.inc
GET     memcfg.inc
GET     2440addr.inc
```

程序的开始是通过 GET 汇编伪指令设置该源码文件要用到的一些头文件的，其中 option.inc 定义 ARM 系统的一些配置选项，在这里主要定义中断向量起始地址_ISR_STARTADDRESS 和系统时钟的分频系数 M_MDIV、M_PDIV、M_SDIV 等。GET 伪指令也可以用 INCLUDE 伪指令替代。

```
UserStack   EQU     (_STACK_BASEADDRESS-0x3800)   ;0x33ff4800 ~
SVCStack    EQU     (_STACK_BASEADDRESS-0x2800)   ;0x33ff5800 ~
UndefStack  EQU     (_STACK_BASEADDRESS-0x2400)   ;0x33ff5c00 ~
AbortStack  EQU     (_STACK_BASEADDRESS-0x2000)   ;0x33ff6000 ~
IRQStack    EQU     (_STACK_BASEADDRESS-0x1000)   ;0x33ff7000 ~
FIQStack    EQU     (_STACK_BASEADDRESS-0x0)      ;0x33ff8000 ~
```

接下来程序定义 ARM CPU 在各种工作模式下的栈顶位置，在系统初始化程序的最后阶段需要设置 CPU 在各种工作模式下的栈顶指针（SP 寄存器），为 C 语言程序的运行做准备。

_STACK_BASEADDRESS 定义在 option.inc 文件中，如下所示。

```
; Start address of each stacks,
_STACK_BASEADDRESS EQU 0x33ff8000
```

其中以“;”开始的行代表注释。

```
MACRO
$HandlerLabel HANDLER $HandleLabel
$HandlerLabel
    sub   sp,sp,#4              ;decrement sp(to store jump address)
    stmfdsp!,{r0}               ;PUSH the work register to stack
    ldr       r0,=$HandleLabel  ;load the address of HandleXXX to r0
    ldr       r0,[r0]           ;load the contents(service routine start address) of HandleXXX
    str       r0,[sp,#4]        ;store the contents(ISR) of HandleXXX to stack
    ldmfd     sp!,{r0,pc}       ;POP the work register and pc(jump to ISR)
MEND
```

上面这段代码定义了一个宏，该宏实现的主要功能是把中断服务程序首地址 HandleLabel 装载到指令寄存器（PC 寄存器）中，当中断产生时，系统能正确执行中断服务处理程序。在后面有关的中断系统处理的章节中，我们会详细分析该宏的使用方法。

接下来我们通过 IMPORT 伪指令导入启动代码中需要用到的外部符号。

```
; declare   ARM-linker internel self-define variable and Main
    IMPORT   |Image$$RO$$Limit|      ; End of ROM code (=start of ROM data)
    IMPORT   |Image$$RW$$Base|      ; Base of RAM to initialise
    IMPORT   |Image$$ZI$$Base|      ; Base and limit of area
    IMPORT   |Image$$ZI$$Limit|     ; to zero initialise

    IMPORT   Main
```

Main 为 C 语言程序的入口函数，也可以改成别的。

接下来才是真正系统初始化程序的开始。首先使用 AREA 伪指令定义代码段，段名为 Init，在链接时，通过链接选项把 Init 段链接到可执行映像文件的第一个段，如下所示。

```
AREA    Init,CODE,READONLY
    ENTRY

    b    ResetHandler
    b    HandlerUndef       ;handler for Undefined mode
    b    HandlerSWI         ;handler for SWI interrupt
    b    HandlerPabort      ;handler for PAbort
    b    HandlerDabort      ;handler for DAbort
    b    .                  ;reserved
    b    HandlerIRQ         ;handler for IRQ interrupt
    b    HandlerFIQ         ;handler for FIQ interrupt
;@0x20
    b    .                  ; Must be @0x20.
```

ENTRY 伪指令定义代码段的入口。系统启动程序的前 32 字节用来存放 ARM 异常向量表。当异常发生时，CPU 自动跳转到异常向量表处执行异常处理程序。当系统上电时，首先执行第一条指令“b ResetHandler”，该代码通过一条跳转指令跳到 ResetHandler 处执行复位处理程序。

代码中“b.”表示跳到当前位置，“.”表示当前指令位置。

```
ResetHandler
    ldr   r0,=WTCON          ;watch dog disable
    ldr   r1,=0x0
    str   r1,[r0]

    ldr   r0,=INTMSK
    ldr   r1,=0xffffffff   ;all interrupt disable
    str   r1,[r0]

    ldr   r0,=INTSUBMSK
    ldr   r1,=0x3ff          ;all sub interrupt disable
    str   r1,[r0]

    ;To reduce PLL lock time, adjust the LOCKTIME register
    ldr   r0,=LOCKTIME
    ldr   r1,=0xffffff
    str   r1,[r0]

    ; Added for confirm clock divide. for 2440. set pll
    ; Setting value Fclk:Hclk:Pclk
    ldr   r0,=CLKDIVN
    ldr   r1,=CLKDIV_VAL            ; 3=1:2:4
    str   r1,[r0]

    ;Configure UPLL
    ldr   r0,=UPLLCON
    ldr   r1,=((U_MDIV<<12)+(U_PDIV<<4)+U_SDIV)
    str   r1,[r0]
```

系统复位处理程序 ResetHandler 首先禁止看门狗，再禁止中断，然后设置 CPU 系统时钟和时钟分频系数，最后设置 USB 时钟和相应的分频系数。

设置好系统时钟后，紧接着开始初始化 SDRAM 控制器，主要设置 SDRAM 的总线宽度和操作时序等。设置完 SDRAM 控制器后，SDRAM 就可以正常工作了。

```
;Set memory control registers
    ldr   r0,=SMRDATA
    ldr   r1,=BWSCON    ;BWSCON Address
    add   r2, r0, #52  ;End address of SMRDATA
0
    ldr   r3, [r0], #4
    str   r3, [r1], #4
    cmp  r2, r0
    bne   %B0
```