摘要

功能强大的嵌入式产品通常需要复杂的操作系统给予支持，系统启动模块(BootLoader)作为系统软件的重要组成部分，起到引导和加载操作系统内核镜像的作用。嵌入式系统低成本启动方案的设计实现，对降低产品的成本具有重要的实用价值。

本文介绍了 BootLoader 的基本概念、工作原理以及和 Boot 硬件的关系，并给出 BootLoader 通用架构。低成本启动方案的 BootLoader 以 NOR Flash、NAND Flash 和 SD 卡为启动存储介质设计实现。方案 1：NOR Flash+NAND Flash 的 Boot 方案，以 NOR Flash 环境下的启动为基础，采用 NOR Flash 存储启动代码，NAND Flash 存储内核镜像的方式。方案 2：NAND Flash 的启动方案根据 NAND Flash 的不同类型，以及启动过程是否通过片上 SRAM，结合 NAND Flash 控制器，设计系统直接从 NAND Flash 启动。方案 1 和方案 2 都采用汇编语言实现，以达到短小精悍的目的。方案 3：SD 卡启动方案以 SD 卡控制器为硬件基础，设计 SD 卡启动控制器并采用硬件描述语言实现，启动的软件流程用 C 语言设计实现，这样代码具有更好的可读性和可移植性。最后在 ARM9 内核的 SEP5010 环境中，采用 EDA 仿真工具 VCS 完成 SD 卡启动设计的功能验证，并在 Altera Stratix III 开发平台，完成三种启动方案的 FPGA 验证。三种设计方案的启动时间依次为：85.6s，111.3s 和 174.1s。根据验证结果：SD 卡启动成本最低，速度最慢；NAND Flash 启动方案成本低，需要 NAND 控制器的硬件支持；NOR Flash+NAND Flash 启动速度快，成本最高。

本文考虑各个 BootLoader 设计中的特点，完成嵌入式系统三种低成本启动方案的设计实现。论文最后部分总结了本文的工作，并进一步探讨 BootLoader 的研究方向。

关键词：嵌入式启动，低成本，NAND Flash 控制器，SDHC/SD 卡控制器改造，FPGA 验证

Abstract

Embedded products usually need an operating system. As the important ingredient of the embedded software, BootLoader's basic function is to boot and load the operating system image. The design and implementation Start low-cost embedded system design to achieve, to reduce the cost of the product has an important practical value.

The paper introduces BootLoader's concepts, working principles, as well as the relationship with Boot hardware, and finally gives its general framework.The design and implementation of low-cost BootLoader is basing on NOR Flash, NAND Flash and SD card storage medium. Scheme 1: NOR Flash + NAND Flash.Basing on the boot process under NOR Flash environment,NOR Flash is used to store code and NAND Flash is for kernel image . Scheme 2: BootLoader basing on NAND Flash is carried out according to the different types of NAND Flash and the usage of ESRAM.Combined with NAND Flash controller, scheme2 is designed to boot from NAND Flash directly.The schemes discussed above are realized in assemble language in order to achieve the purpose of dapper.Scheme 3:the hardware foundation of this scheme is SD card controller.SD card boot controller is designed and carried out in hardware description language.Boot process is realized in C language which brings better readability and portability. Function validation about SD card boot under ARM9 core SEP5010 chip environment has been done by VCS development kit. FPGA verification is basing on Altera Stratix III development platform. The time cost about three boot schemes is: 85.6s, 111.3s and 174.1s. According to results,SD card boot has the lowest cost and the loweast boot speed; NAND Flash boot with low cost but need the hardware support of NAND Controller; NOR Flash + NAND Flash boot has high speed and high cost.

In this thesis, full consideration is given to the characteristics of various BootLoader, and actual design of BootLoader aiming at low-cost boot on different storage media in embedded system is completed. The last chapter of the thesis touches upon the development direction of BootLoader and also reaches conclusion of the present research.

Key Words: Embeded System, Low-cost, NAND Flash Controller, SDHC/SD Card Controller Rebuild, FPGA Verification

东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

研究生签名：王清 日期：09.8.25

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括以电子信息形式刊登）论文的全部内容或中、英文摘要等部分内容。论文的公布（包括以电子信息形式刊登）授权东南大学研究生院办理。

研究生签名：王清 导师签名：[illegible] 日期：09.8.25

第一章 绪论

1.1 课题研究背景

在嵌入式系统的开发中，嵌入式软件是实现各种系统功能的关键，也是计算机技术最活跃的研究方向之一。不同应用对嵌入式软件系统有不同的要求，并且随着计算机技术的发展，这些要求也在不断变化。通常，应用系统对嵌入式软件的基本要求是体积小、执行速度快、具有较好的可剪裁性和可移植性。特别地，现在对嵌入式软件来说，都需要操作系统的支持。简单的嵌入式系统没有操作系统，只是一个控制循环，但是当系统变得越来越复杂时，就需要一个嵌入式操作系统来支持，否则应用软件就会变得过于复杂，使开发难度过大，安全性和可靠性都难以保证。

现在，专门为嵌入式产品开发的各种操作系统层出不穷，各界关注也特别多。然而，如何加载操作系统这个问题却研究得比较少，这就产生了另一个相关主题 BootLoader。BootLoader 的功能之一就是引导与加载内核镜像。BootLoader 是与系统高度相关的初始化软件，其广泛用于有操作系统的手持终端设备、智能家电、机顶盒等嵌入式设备上[1]。BootLoader 不仅担负着初始化硬件和引导操作系统的双重责任，还负责完成系统调试、系统配制等功能。每种不同的 CPU 体系结构都有不同的 BootLoader。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，如 RAM 的大小、RAM 在整个系统空间中的位置、ROM 的大小和位置、与主机的通信方式等[2][3]。BootLoader 依据量体裁衣、定身制作的原则，将满足要求的最小化代码存放在启动介质中。

从嵌入式系统实际开发角度讲，嵌入式操作系统的引导配置甚至应用程序的运行状况都和 BootLoader 有一定的关联，BootLoader 的重要性主要表现在以下几个方面：

1、功能的多样性。对于常见的的PC来说，操作系统内核主要存放在磁盘上，其位置、大小都存在一定的差异。这个时候，BootLoader的主要任务就是根据位置、大小等信息从磁盘读入操作系统内核并加以引导。由于PC具有很长的发展历史，其硬件体系结构和操作系统引导过程已经形成规范，另外，PC机中BIOS的存在，在某种程度上也简化了BootLoader对硬件进行检测和初始化的过程。因此，PC机中的BootLoader和硬件的关系就不如嵌入式系统中的那么紧密。而对嵌入式系统来说，由于其系统的多样性，所以BootLoader的主要工作不仅要包括从存储介质中将操作系统内核读入内存并加以执行，还必须要担负起对硬件的初始化和检测过程，有些甚至需要从串并口或者网卡上将操作系统内核下载到本机中，其复杂程度可想而知。嵌入式系统的BootLoader一般从某个固定的地址处开始执行，这个固定的地址可能是一段ROM或Flash空间，并且可以直接运行[4]。

2、在产品开发中的必要性。对于嵌入式系统的硬件提供商，所面对的是系统的二次开发商而不是最终用户，往往并不清楚二次开发商所要使用的操作系统，不同的操作系统内核其大小、引导方式、内核参数传递方法等都存在一定的差异。显然，要求二次开发商来开发针对硬件厂商的 BootLoader 是不现实的，其涉及到研发成本、技术机密等多方面因素。因此，对于硬件厂商来说，在其硬件平台上提供一个灵活性很高的引导程序以方便二次开发商是其必须要完成的工作。因此，对于硬件提供商来说，BootLoader 的开发是其产品开发中的重要组成部分。实际上大多数的嵌入式

处理器生产商在自己产品的开发包中都会提供一个或多个 BootLoader，并附带源代码，以方便二次开发商的移植工作。

3、为系统的可升级性和可调试性提供方便。任何软件的开发都离不开调试过程，对嵌入式系统的软件，尤其是操作系统内核，需要反复地将其加载到目标机上。此外，即使在系统的运行过程中，也难免会遇到需要对系统进行升级的情况。这就需要提供一种方法以方便对系统进行替换和升级。这个时候，解决这些问题的一个自然的方法就是设计一个包含这些功能的 BootLoader。

1.2 BootLoader 研究现状

由于同一种内核，不同厂家生产的芯片差别很大，因此不易编写出统一的 BootLoader 代码。很多公司针对这一问题而采取的策略是，不提供完整的 BootLoader 代码（例如 ARM 公司的开发工具 ADS 提供了一些功能代码），BootLoader 代码不足的部分由芯片厂商提供或者由用户自己编写。虽然也可以自行编写 BootLoader，但从可利用的资源和实际项目开发考虑，采用移植和修改已有的通用 BootLoader 源码来解决这一问题更符合大多数项目的开发要求。

目前通用的 BootLoader 很多，功能强大，开发容易，而且由专人维护升级。现在比较常见的嵌入式的 BootLoader 主要有以下几种[5-14]：

1．U-boot U-boot（Universal BootLoader）是遵循 GPL 条款的开放源码项目，从 FADSROM，8xxROM，PPCBOOT 逐步发展演化而来，其源码目录、编译形式与 Linux 内核很相似。事实上不少 U-boot 源码就是相应 Linux 内核源程序的简化，尤其是一些设备的驱动程序。但是 U-boot 不仅仅支持嵌入式 Linux 系统的引导，当前，它还支持 NetBSD，VxWorks， ARTOS，LynxOS 等嵌入式操作系统，并且支持 PowerPC，MIPS，x86，ARM，Nios，Xscale 等诸多常用系列的处理器。

2．Blob Blob 是 BootLoaderObject 的缩写，是由著名的开源嵌入式 Linux 工程 LART 发展起来的一个开源 BootLoader 程序，是一款功能强大的 BootLoader。它遵循 GPL 源代码完全开放的原则。Blob 既可以用来简单的调试，也可以启动 Linux kernel。目前它主要支持 Intel 的 StrongARM 体系结构和 Xscale 结构的 ARM 芯片。

3．Open BIOS OpenBIOS 是一个开源的、可移植的固件程序，其目标是实现一个与 IEEE1275-1994（即 Open Firmware，开发固件标准）标准 100%兼容的固件程序。OpenBIOS 目前支持 x86、Alpha、AMD64 和 IPF 等体系结构的处理器。

4．Linux BIOS Linux BIOS是由LosAlamos的高级计算实验室中的集群研究室在1999年发起的一个研究项目。发起这个项目的最初动机是为了让操作系统在启动后可以控制集群系统的节点机。LinuxBIOS的启动速度很快，其最快的启动时间号称只需3秒。

5．Angel 和 Angelboot 这是一种典型的目标板十主机型的 BootLoader，支持 Intel 的 Assabct。目标板的部分存放在 Flash 中，可以通过串口将内核加载到 Flash 中，主机端通过简单的 RC 文件来加以配置。在正常的启动过程中它把内核和 Raln Disk 加载到 RAM 中再从内核的入口处执行。Angel 和 Angelboot 是一个总的 BootLoader 的称呼。Angel 是目标板上运行的部分，在启动的时候执行，而 Angelboot 是运行在主机上的部分，它和目标机上的 Angel 配合来控制对内核的加载操作。Angel

实际上还可以用做调试器代理来用。

1.3 课题的主要内容

嵌入式产品的开发和使用，需要操作系统和应用软件的支持，嵌入式产品的市场定位决定了它的低成本要求。本课题的意义即以上面两点为基础，嵌入式系统低成本启动方案的设计与实现对降低嵌入式产品的成本具有实际的帮助。本文的内容主要包括以下几个部分：

第一章：绪论。介绍了嵌入式系统中 BootLoader 的重要性和多样性，以及对不同系统的不同选择，由此得出本课题的来源和本文的主要研究工作。

第二章：BootLoader 功能分析。阐述了 BootLoader 的基本原理，并归纳出其通用架构，并讨论了基于三种不同存储介质 NOR Flash、NAND Flash 和 SD 卡的启动方式的差异以及各自的适用范围。

第三章：为本文的重点章节，主要内容为低成本启动方案的设计。介绍了现有的软硬件条件，以及在此基础上完成启动方案的设计。在 NOR Flash 启动代码的基础上实现 NOR Flash+NAND Flash 的启动方案；基于 NAND Flash 的启动设计则按照是否使用 ESRAM（片上 SRAM）分成两种不同的设计实现；基于 SD 卡的启动则通过设计 SD 卡启动控制器和软件启动流程完成。

第四章：本文设计方案的测试工作，仍为本文的重点部分。根据现有的软件和硬件条件， 首先在服务器上面验证基于各种存储介质基本的读写功能是否正确，然后搭建 FPGA 平台，通过设计外围相关硬件电路，完成不同 BootLoader 的 FPGA 验证，以证实设计的正确性和可行性。并根据 FPGA 测试结果，比较各个启动方案的速度，分析成本，确定各自的适用环境。

第五章：为全文的总结与展望，总结本文所做的工作，并对本文的深入研究提出目标。

第二章 BootLoader 功能分析

2.1 BootLoader 基本概念

一个嵌入式系统从软件的角度看通常可以分为四个层次：引导加载程序、操作系统内核、文件系统、用户应用程序，如图 2-1 所示[1]。

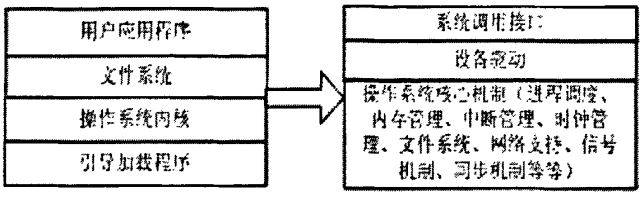

图2-1 操作系统层次图

引导加载程序是系统加电后运行的第一段代码。常用的 PC 中的引导程序一般由 BIOS 和位于 MBR 的 OS BootLoader(例如 LILO 或者 GRUB)一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 BootLoader 读到系统的 RAM 中，然后将控制权交给 OS BootLoader。BootLoader 的主要运行任务就是将内核镜象从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。 而在嵌入式系统中，通常并没有像 BIOS 那样的固件程序（有的嵌入式 CPU 也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 BootLoader 来完成。

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，可以初始化必要的硬件设备，创建内核需要的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[15][16]。

2.2 BootLoader 的共性

通常，BootLoader 是严重依赖于硬件而实现的，特别是在嵌入式系统中[17-19]。不同的体系结构需求的 BootLoader 是不同的。因此，在嵌入式世界里建立一个通用的 BootLoader 几乎是不可能的。尽管如此，仍然可以对 BootLoader 归纳出一些通用的概念，以指导用户特定的 BootLoader 设计与实现：

（1）BootLoader所支持的CPU和嵌入式板

每种不同的 CPU 体系结构都有不同的 BootLoader。有些 BootLoader 也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。这也就是说，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，通常都需要修改 BootLoader 的源程序。

（2）BootLoader的安装媒介（Installation Medium）

系统加电或复位后，所有的 CPU 通常从某个由 CPU 制造商预先安排的地址上取指令。比如，基于 ARM7TDMI Core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备（比如：ROM、EEPROM 或 Flash 等）被映射到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 BootLoader 程序。图 2-2 就是一个同时装有 Bootloader、内核的启动参数、内核镜像和根文件系统镜像的固态存储设备的典型空间分配结构图。

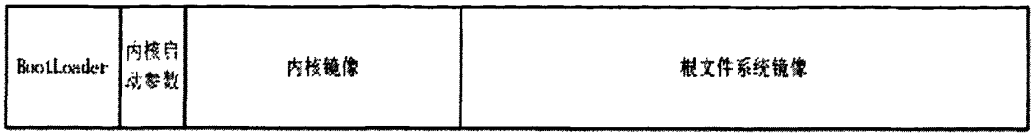

图2-2 固态存储设备的典型空间分配结构

（3）BootLoader的启动过程是单阶段（Single Stage）还是多阶段（Multi-Stage）

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 大多都是两个阶段的启动过程，也即启动过程可以分为 stage1 和 stage2 两部分。至于在 stage1 和 stage2 具体完成哪些任务将在下面做详细讨论。

（4）BootLoader的操作模式

大多数 BootLoader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别对于开发人员才有意义。但从最终用户的角度看，BootLoader 的作用永远就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载模式：这种模式也称为"自主"模式，即 BootLoader 从目标机上的某个固体存储设备上将操作系统加载到 RAM 中运行，整个过程没有用户的介入。这种模式是 BootLoader 的正常工作模式，因此当以嵌入式产品发布的时候，BootLoader 必须工作在这种模式下。

下载模式：在这种模式下，目标机上的 BootLoader 将通过串口或者网络连接或者其它通信手段从主机下载文件，比如：下载内核镜像和根文件系统镜像等。从主机下载的文件通常首先被 BootLoader 保存到目标机的 RAM 中、然后被 BootLoader 写到目标机上的 Flash 类固态存储设备中。BootLoader 的这种模式通常在第一次安装内核与根文件系统时使用。此外，以后的系统更新也会使用 BootLoader 的这种工作模式。工作于这种模式下的 BootLoader 通常都会向它的中断用户提供一个简单的命令行接口。

（5）BootLoader 与主机之间进行文件传输所用的通信设备及协议最常见的情况就是，目标机上的 BotLoader 通过串口与主机之间进行文件传输，传输协议通常是 xmodel/ymodel/zmodel 协议中的一种。但是，串口传输的速度是有限的，因此通过以太网连接并借助 TFTP 协议来下载文件是个更好的选择。

此外，系统更新也会使用 BootLoader 的这种工作模式。工作于这种模式下的 BootLoader 通常都会向它的终端用户提供一个简单的命令行接口。

（6）BootLoader的通用性

BootLoader 的设计与实现是与具体的 CPU 以及具体的硬件系统紧密相关的，要实现一个通用的 BootLoader，就要适合具体的微处理器以及硬件系统。另外，不同的操作系统，可能对具体的 BootLoader 还会有另外额外的要求。同时，对于 BootLoader 的这些共同特性，理论上只局限于 BootLoader 的基本功能，因为扩展功能众多，可以有串口、USB、以太网接口、IDE, CF 等，无法进行归纳与总结。

对于一个系统来说， BootLoader 作为引导与加载内核镜像的工具，必须提供以下几个功能：

- 初始化硬件设备：为后面程序的运行以及Kernel的加载准备一些基本的硬件环境。
- 初始化RAM：BootLoader必须能够初始化RAM，因为将来系统要通过它保存一些Volatile数据，但具体地实现要依赖与具体的CPU以及硬件系统。
- 初始化串口：BootLoader应该要初始化以及使能至少一个串口，通过它与控制台联系进行一些debug的工作以及与PC通信。
- 创建内核参数列表。
- 启动内核镜像：这是必须的，因为BootLoader的最终任务就是加载内核并将控制权交与它。

2.3 BootLoader 典型结构框架

上一小节在分析 BootLoader 共性的基础上，从理论上对 BootLoader 的通用性作了总结，而这些理论仍然比较抽象，难以客观理解，需要提出具体的实现模型才可以起到指导作用。

由于 BootLoader 的实现依赖于 CPU 的体系结构，因此将 BootLoader 的实现分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。而 stage2 则通常用 C 语言来实现，这样可以实现复杂的功能，而且代码会具有更好的可读性和可移植性[20]。

BootLoader 的 stage1 通常包括以下步骤（以执行的先后顺序）：

- 硬件设备初始化。
- 为加载BootLoader的stage2准备RAM空间。
- 加载BootLoader的stage2到RAM空间中。
- 设置好堆栈。
- 跳转到stage2的C入口点。

BootLoader的stage2通常包括以下步骤（以执行的先后顺序）：

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射（Memory Map）。
- 将Kernel镜像和根文件系统镜像从Flash上读到RAM空间中。
- 为内核设置启动参数。
- 调用内核。

2.3.1 BootLoader 的 stage1

如图 2-3 所示，stage1 流程如下：

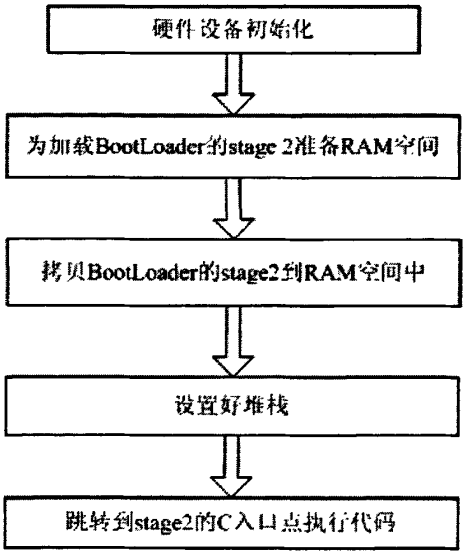

图2-3 stage1流程图

1、基本的硬件初始化

在 stage1 中要完成一些准备工作，目的是为 stage2 的执行以及随后的 Kernel 的执行准备好一些基本的硬件环境。它通常包括以下步骤：

（1）屏蔽所有的中断。为中断提供服务通常是 OS 设备驱动程序的责任，因此在 BootLoader 的执行全过程中可以不必响应任何中断。中断屏蔽可以通过写 CPU 的中断屏蔽寄存器或状态寄存器（比如 ARM 的 CPSR 寄存器）来完成。

（2）设置 CPU 的速度和时钟频率。

（3）RAM 初始化。包括正确地设置系统的内存控制器的功能寄存器以及各内存库控制寄存器等。

（4)通过 GPIO 来驱动 LED，其目的是表明系统的状态是 OK 还是 Error。如果板子上没有 LED，那么也可以通过初始化 UART 向串口打印 BootLoader 的 Logo 字符信息来完成这一点。

（5）关闭 CPU 内部指令/数据 Cache。

2、为加载 stage2 准备 RAM 空间

为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，因此必须为加载 BootLoader 的 stage2 准备好一段可用的 RAM 空间范围。

由于 stage2 通常是 C 语言执行代码，因此在考虑空间大小时，除了 stage2 可执行镜象文件的大小外，还必须把堆栈空间也考虑进来。此外，空间大小最好是 Memory Page 大小(通常是 4KB)的倍数。一般而言，1M 的 RAM 空间已经足够了。具体的地址范围可以任意安排，但是，必须确保所安排的地址范围的确是可读写的 RAM 空间。

3、加载 stage2 到 RAM 中时要确定两点：（1）stage2 的可执行镜象在固态存储设备的存放起始地址和终止地址；（2）RAM 空间的起始地址。

4、设置堆栈指针 sp 堆栈指针的设置是为了执行 C 语言代码作好准备。

5、跳转到 stage2 的 C 入口点：在上述一切都就绪后，就可以跳转到 BootLoader 的 stage2 去执行了。比如，在 ARM 系统中，这可以通过修改 PC 寄存器为合适的地址来实现。

2.3.2 BootLoader 的 stage2

stage2 的代码通常用 C 语言来实现，以便于实现更复杂的功能和取得更好的代码可读性和可移植性。其基本功能流程图如图 2-4 所示：

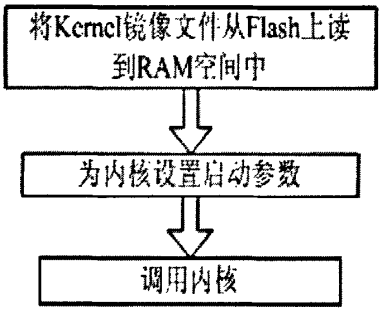

图2-4 stage2流程图

阶段 2 的实现首先要规划内存占用的布局，为 Kernel 镜像文件从 Flash 中拷贝过来准备 RAM 空间，之后就开始 stage2 的执行步骤：

1、从 Flash 上拷贝

通常嵌入式 CPU 通常都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，因此从 Flash 上读取数据与从 RAM 单元中读取数据基本相同。用一个简单的循环就可以完成从 Flash 设备上拷贝镜像的工作。

2、为内核设置启动参数

在将内核镜像和根文件系统镜像拷贝到 RAM 空间中后，就可以准备启动系统内核了。但是在调用内核之前，应该作一步准备工作，即设置系统内核的启动参数。

3、调用内核

BootLoader 调用系统内核的方法是直接跳转到内核的第一条指令处。在调用内核时，需要满足一些条件：

（1）CPU 寄存器的设置：R0=0；R1=机器类型；R2=启动参数标记列表在 RAM 中的起始地址。这三个寄存器的设置是在最后启动内核时通过启动参数来传递完成的。

（2）CPU 模式：关闭中断；属于 SVC 模式。

（3）Cache 和 MMU 的设置：MMU 必须关闭；数据 Cache 必须关闭；指令 Cache 可以关闭也可以开启。

2.4 基于三种不同存储介质的 BootLoader 介绍

2.4.1 嵌入式存储介质

2.4.1.1 Flash 存储器

在嵌入式系统中，被广泛使用的存储设备为随机访问存储器（RAM）和闪存。RAM 用来存放系统运行时的数据，掉电后所存放的所有数据全部丢失。而闪存是用来存放需要永久保存的程序和数据，掉电后不会丢失。闪存以半导体为存储媒介，分为多种类型，其中 NOR Flash 和 NAND Flash 是目前主流的类型[21-29]。

表 2-1 NOR Flash 和 NAND Flash 的比较

Flash 类型 / 权衡角度	NOR Flash	NAND Flash
容量	1MB~32MB	16MB~2GB
是否支持 XIP eXecute In Place	支持	否
性能比较	擦写速率很慢（5s） 写速率很慢，读速率快	擦写速率很快（4ms） 写速率快，读速率快
可靠性	可靠性比较好 位翻转的现象很少见	可靠性比较低 需使用 1-4 位的 EDC/ECC 校验来解决位翻转的现象，同时需要坏块管理
擦写周期	10,000-100,000	100,000-1,000,000
使用寿命	10 万次的擦写次数	是 NOR Flash 使用寿命的 10 倍
接口特性	具有 RAM 接口	具有 I/O 接口特性，由控制信号控制
访问方式	可随机访问	顺序访问
合理用途	存储代码，因容量受限于价格，可存储少量数据	存储数据，因具有复杂的 Flash 管理

NOR Flash的特点是芯片内执行（XIP，eXecute In Place），这样应用程序可以直接在Flash闪存内运行，不必再把代码读到系统RAM中。NOR Flash的传输效率很高，在1～4MB的小容量时具有很高的成本效益，但是很低的写入和擦除速率大大影响了它的性能。

NAND Flash结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速率也很快。应用NAND Flash的困难在于Flash的管理和需要特殊的系统接口。M-system公司在NOR Flash和NAND Flash的主要区别进行了一系列比较。主要区别如表2-1所示。

从表2-1中可以看出，NAND Flash和NOR Flash的比较可以从以下几个方面进行：

（1）容量和成本

NOR Flash占据了容量为1MB~32MB闪存市场的大部分，而NAND Flash只是用在16MB~2GB的

产品当中，这也说明NOR Flash主要应用在代码存储介质中，NAND Flash适合于数据存储，NAND Flash在Compact Flash、Secure Digital、PC Cards和MMC存储卡市场上所占份额最大。NAND Flash的单元尺寸较小，由于生产过程简单，NAND Flash结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

（2）性能比较

Flash闪存是非易失存储器，可以对存储器单元块进行擦写和再编程。任何Flash器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND Flash器件执行擦除操作是十分简单的，而NOR Flash则要求在进行擦除前先要将目标块内所有的位都写为0。

由于擦除NOR Flash器件时是以64～128KB的块进行的，执行一个写入/擦除操作的时间为1～5s，与此相反，擦除NAND Flash器件是以8～32KB的块进行的，执行相同的操作最多只需要4ms。执行擦除操作时块尺寸的不同进一步拉大了NOR Flash和NAND Flash之间的性能差距，统计表明，对于给定的一套写入操作(尤其是更新小文件时)，更多的擦除操作必须在基于NOR Flash的单元中进行。这样，当选择存储解决方案时，需要权衡以下的各项因素。

- NOR Flash的读速率比NAND Flash稍快一些
- NAND Flash的写入速率比NOR Flash快很多
- NAND Flash的4ms擦除速率远比NOR Flash的5s快
- 大多数写入操作需要先进行擦除操作
- NAND Flash擦除电路更少

（3）可靠性和耐用性

耐用性：从表中可以看到，在 NAND Flash 闪存中，每个块的最大擦写次数是 100 万次，而 NOR Flash 的擦写次数是 10 万次。NAND Flash 存储器除了具有 10：1 的块擦除周期优势外，每个 NAND Flash 存储器块在相同时间内的删除次数要少一些。

所有 Flash 器件都受位交换现象的困扰。在某些情况下，一个比特位会发生反转或被报告反转了。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障就可能导致系统停机。如果只是报告有问题，多读几次就可能解决。

位反转的问题更多见于 NAND Flash 闪存，NAND Flash 的供应商建议使用 NAND Flash 闪存的时候，同时使用 EDC/ECC 算法。当然，如果用本地存储设备来存储操作系统、配置文件或其它敏感信息时，必须使用 EDC/ECC 系统以确保可靠性。

坏块处理：NAND Flash 器件中的不坏块是随机分布的。以前做过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。NAND Flash 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理。将导致高故障率。

（4）接口差别

NOR Flash带有RAM接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。可以实现数据的随机读取。NAND Flash器件使用复杂的I/O口来串行地存取数据，各个产品或厂商的方法可能各不相同。8个引脚用来传送控制、地址和数据信息。 NAND Flash读和写操作采用512字

节的块，这一点有点像硬盘管理此类操作，很自然地，基于NAND Flash的存储器就可以取代硬盘或其他块设备。

（5）用途

考虑到成本的问题，一般系统中使用 NOR Flash 型闪存来存储代码，也可以存储少量的数据。因 NAND Flash 型闪存具有复杂的管理数据的方法，所以一般在 NAND Flash 型闪存上存储数据。

2.4.1.2 SD 存储卡

SD 存储卡最初是从 MMC 卡基础上发展起来的，可以与 MMC 卡实现兼容。SD 存储卡的主机控制器接口可以支持常规的 MMC 卡操作，实际上 SD 存储卡和 MMC 卡的主要区别在于初始化过程。相对于 MMC 卡，SD 存储卡拥有更快的数据存取速度以及更大的存储容量，并且 SD 存储卡符合安全标准版权保护机制（Secure Digital Music Initiative，SDMI），SDMI 是音乐界及有关人士组成的组织，该组织旨在推广 SDMI 方案来有效地保护唱片业的利益。SD 存储卡的版权保护机制具体实现为可记录媒体内容保护(Content protection for Recordable Media, CPRM)功能，用于卡内数据的授权访问，实现内容保护。

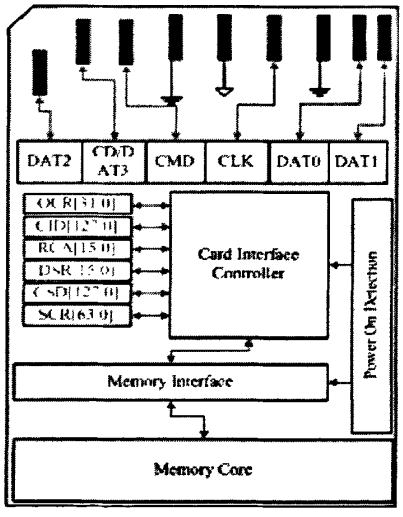

图 2-5 SD 存储卡内部结构

如图 2-5 所示，SD 存储卡内部主要包括大容量 Flash 存储芯片和一个片上控制器。片上控制器单元主要完成接口通信协议、版权保护的安全算法、数据存储和恢复、ECC 校验算法、缺陷诊断和处理的实现，并且负责电源管理和时钟控制。这样就易化了软件设计人员对 Flash 存储芯片的管理，SD 存储卡内部的 Nand Flash 不需要外部参与进行缺陷处理和诊断以及 ECC 校验。标准 SD 存储卡背面共有 9 个引脚，包含 4 根数据线、1 根命令线、一根时钟线、1 根电源线以及两根地线。它的尺寸为 32mm*24mm*2.1mm，相当于邮票的大小。

2006 年，SDA 协会发布了 SD2.0 规范，符合此新规范的 SD 存储卡容量为 4GB 到 32GB。符合新规范的 SD 存储卡，称为 SD 高容量（SD high capacity, SDHC）卡。SDHC 卡外形维持与 SD 存储卡一致，但是文件系统从 FAT12、FAT16 改为 FAT32 型。除了 SDHC 卡外，还有 Mini SDHC，Micro SDHC 类型的卡。SDHC 卡与标准 SD 存储卡不再兼容，必须符合 SD2.0 规范的设备才能支持 SDHC

卡，而支持 SDHC 卡的设备可以向下兼容 SD 存储卡。为了充分发挥 SDHC 卡的性能，保证兼容性，SDA 协会为 SDHC 卡定义了 3 个速度等级：2，4，6；其含义是各等级分别可以接受的传输速率至少是 2MB/s，4MB/s，6MB/s。速度等级定义中使用的是数据写速率，数据读速率要比数据写速率快。

2.4.2 三种不同 BootLoader 特点及使用范围

目前市面上常用的嵌入式系统的 BootLoader 方案一般是基于 NOR Flash，NOR Flash 芯片内执行的特点使得设计基于 NOR Flash 的 BootLoader 比较简单且容易实现。但它的缺点则是写入数据的速度过慢，加上每位的成本相对于 NAND Flash 和 SD 卡来说是昂贵的。因此，NOR Flash 主要应用在代码存储方面。

NAND 型 Flash 价格比 NOR 型 Flash 廉价，尺寸小，能提供极高的单元密度，并且写入和擦除的速度也很快。因此可以考虑设计基于 NAND Flash 的嵌入式 BootLoader，相比较 NOR Flash 而言，可以支持更大体积的嵌入式操作系统内核。NAND Flash 的应用困难在于 Flash 的管理和需要特殊的系统接口。

当前很多嵌入式处理器都已经集成了与 SD 存储卡通信的接口，因此本文设计出一种新的启动方式——嵌入式系统实现从 SD 卡的启动。由于 SD 卡内嵌有 NAND 控制器，所以从嵌入式系统应用的角度上看，不用考虑 NAND 坏块的问题，使用更加方便。而且，SD 卡的可移动性以及方便的实时更新性，是 NAND Flash 和 NOR Flash 无法比拟的。所以，基于 SD 卡的 BootLoader 适用于启动大规模操作系统且系统内核的更新频率较高的环境中[30][31]。

2.5 本章小结

本章介绍了 BootLoader 的基本概念、作用，对 BootLoader 的概念作了扩展，总结了 BootLoader 的共性；在此基础上，提出了通用 BootLoader 的设计模型。这一模型屏蔽了不同体系结构在硬件上的差异，采用分阶段的实现方式，详细阐述了每一阶段功能的设计框架。然后介绍了三种的嵌入式存储介质 NOR Flash、NAND Flash 和 SD 卡，讨论了各自不同的特点，分析了基于这三种介质的 BootLoader 所对应的适用环境。下一章将在本章介绍的内容基础之上，以 ARM 处理器为基础进行基于不同存储介质的 BootLoader 代码的设计实现。

第三章 低成本启动方案的设计实现

本章作为本文的重点，主要介绍基于三种不同存储介质：NOR Flash、NAND Flash 和 SD 卡的启动方案的设计实现。SEP5010 是一款基于 ARM9 内核的嵌入式处理器，也是本文设计启动方案的硬件基础。基于 NOR Flash 的启动方案是目前最常见的一种，有成熟的 BootLoader 软件代码。本文从 SEP5010 的内部架构出发，介绍了其内部和启动设计相关的硬件模块的功能，并结合 NOR Flash 环境下的 BootLoader 软件代码和三种不同的存储介质，提出了对应的启动方案：

（1）在给予 SEP5010 芯片的基础上，设计基于 NOR+NAND+SDRAM 存储方案的 BootLoader

该方案结合了 NOR Flash 和 NAND Flash 两种存储介质各自的优点，以 NOR Flash 环境下的 BootLoader 代码为基础，结合 SEP5010 内部硬件模块：NAND 控制器，EMI 模块，PMC 模块等，选用汇编语言完成新方案的 BootLoader 代码的编写。

（2）在给予 SEP5010 的芯片的基础上，设计基于 NAND+SDRAM 存储方案的 BootLoader

该启动方案，仍然以 NOR Flash 环境下的 BootLoader 为基础，研究其启动方式和流程，再结合 SEP5010 处理器中的 NAND 控制器、DMA、ESRAM 等硬件条件，设计基于 NAND Flash 的 BootLoader。在 BootLoader 设计过程中，考虑到 ESRAM 在 ARM 芯片中的特性，以启动是否用到 ESRAM 为标准，整个设计又分成两种。整个方案设计采用汇编语言完成实现。

（3）在 SEP5010 芯片基础上，改造 SDIO 控制器，使其支持 SD 卡+SDRAM 的启动方案，并设计相应的 BootLoader

该启动方案分为硬件改造和软件流程设计两部分。硬件改造是在 SEP5010 内部的 SD 卡控制器基础上完成，使其支持新的启动方案，采用硬件描述语言完成该工作。在硬件改造成功的基础上，仍选用汇编语言完成基于 SD 卡+SDRAM 的 BootLoader 设计。

对于上述三种存储介质，由不同的硬件模块来实现对其数据的存储和读取，因此在启动方案设计的过程中会对使用的硬件模块做个大概的介绍，以便更好的解释各个启动方案的设计思想。

不管是从上述三种介质中的哪一种启动，启动代码都需要完成本质上相同的操作：必要的硬件设置和代码搬运，在系统上要真正运行的是启动程序所搬运的代码。这些代码是在开发完成后在调试状态下生成的内存映象，它被保存在一个二进制镜像文件中，并和启动代码一起烧录到存储介质中。烧录时需要把启动代码放在首地址，程序代码则应放在启动代码所指定的搬运首地址。而且程序代码的首地址必须存放的是异常中断向量表。这些都是启动方案设计过程所要考虑到的因素。

3.1 启动设计硬件环境

本论文研究工作的硬件平台是基于东南大学 ASIC 工程中心自主研发的一款嵌入式处理器芯片 SEP5010。该处理器内嵌 ARM926EJ 内核，其主频可达到 260MHz。SEP5010 处理器适用于高端多媒体手持终端应用，并有竞争力较强的设计方案，以及低功耗设计以适合电池供电设备。芯片内置 EMI，支持 SRAM/SDRAM/NOR Flash/NAND Flash，支持低成本的 Nand Flash 控制器并可从其直接

启动；支持时钟和功耗管理模块（PMC）；支持中断控制器（INTC）；支持 MMC/SD 控制器，用户可以扩展系统的存储能力和外设功能；支持 6 通道 DMA 控制器，为用户提供了高速的数据传输通道。SEP5010 的结构如图 3-1 所示：

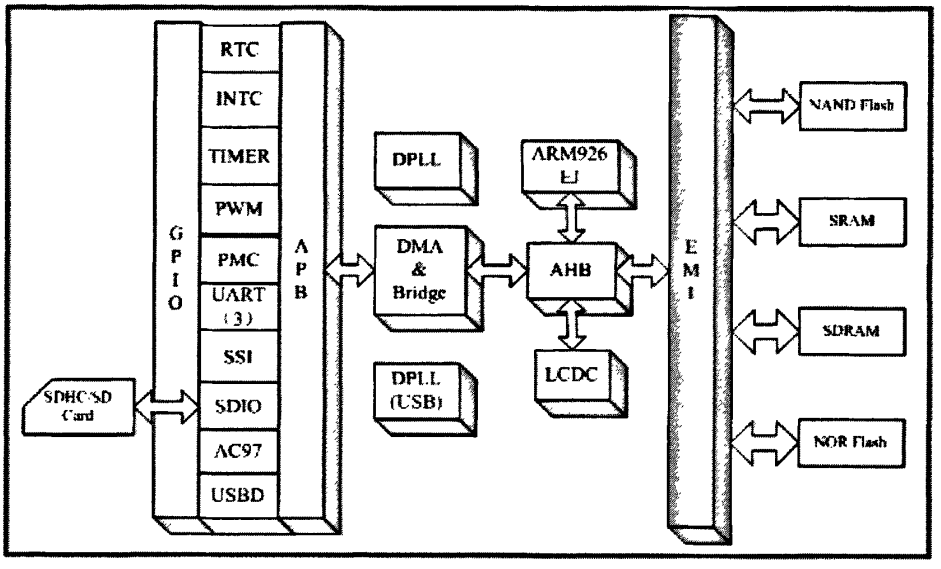

图3-1 SEP5010整体框架

该处理器中的部分模块为本文启动方案的设计提供硬件条件，因此，有必要对相关的硬件模块进行功能介绍[32-35]：

（1）处理器内核－ARM926EJ，特点如下：

- 32/16位RISC架构（ARMv5TEJ）
- 支持高性能和灵活性的32位ARM指令集
- 支持代码紧凑的16位Thumb指令集
- DSP指令扩展与单周期乘加（MAC）操作
- ARM Jazelle技术实现Java实时性能，高效的Java字节码的执行和超低Java功耗
- 支持下列操作系统（Symbian OS, Windows CE & Linux）的存储器管理单元MMU
- 16 K的指令缓存和16K的数据缓存
- 支持实时调试的EmbeddedICE-RT逻辑

（2）外部存储器接口控制器（EMI）：

支持 SRAM/SDRAM/Nor Flash/Nand Flash 存储器。SRAM 存储器接口可支持 8/16/32 位方式，SDRAM 可支持 16/32 位方式；支持 6 个片选，片选起始地址均可配。SEP5010 外部存储接口模块（EMI,EXTERNAL MEMORY INTERFACE），它的功能是为外部存储器提供读写接口。它支持地址的 REMAP 功能，即两个逻辑地址指向同一个物理地址，且支持从外部 NOR Flash 启动以及从 NAND Flash 启动。本处理器支持的片外存储器包括：SRAM、ROM、NOR Flash、SDRAM 及 NAND Flash。

EMI 整体框图如图 3-2 所示：

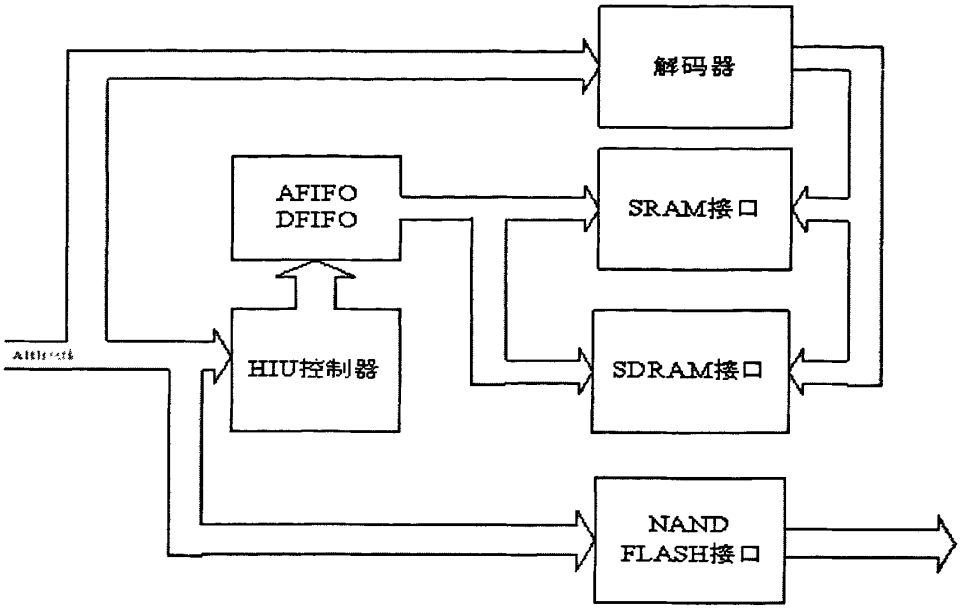

图3-2 EMI整体框图

EMI 分为两大子模块，一个是 HIU 模块（AHB 总线接口单元），另外一个是 MIU 模块（存贮器接口单元）。其中 HIU 模块中包含 HIU 控制器，数据 FIFO（DFIFO），地址 FIFO（AFIFO）三个单元模块；MIU 模块中包含解码器单元，SRAM 接口单元，SDRAM 接口单元，NAND Flash 接口单元。

（3）时钟和功耗管理模块（PMC）：

支持芯片主频软件可配；所有外围模块时钟独立控制；支持低功耗模式，共有四种功耗模式可供切换：

- NORMAL：系统正常工作，不进行低功耗处理
- SLOW：时钟不经过PLL，由外部晶振直接或者分频提供
- IDLE：CPU时钟关闭
- SLEEP：除实时时钟模块（RTC）外，其它模块都停止工作

（4）中断控制器（INTC）：

- 支持IRQ中断和FIQ快速中断
- 共33个中断源，其中15个内部中断，15个外部中断，3个外部快速中断
- 外部中断支持沿出发、电平出发，极性可配
- 普通中断优先级过滤配置
- 支持软件强制中断
- 支持所有中断的软件优先级配置

（5）DMA控制器（DMAC）：

- 6个独立的DMA通道，支持双向传输
- 支持存储器到存储器，存储器到外设，外设到存储器的DMA传输

- 硬件DMA通道优先级
- 地址产生可配置为递增或非递增，不支持卷址
- 软件配置Burst请求支持穿越1KB地址边界
- 支持基于链表配置的DMAC
- 支持锁通道功能
- 支持硬件连线和软件配置的握手机制，支持Single和Burst请求
- 6通道共用一个16×32bit FIFO
- 自动对数据进行打包、解包以适合FIFO宽度
- 可配置的传输控制方：外设或者DMA控制器本身
- 两个中断请求：DMA传输错误和DMA传输完成中断请求

（6）MMC/SD控制器（MMC/SD）：

- 兼容SD Spec ver1.01/1.10和MultiMediaCard Spec ver4.X/3.X
- 支持SD/MMC 1bit/4bit/8bit modes，不支持SPI模式
- 支持MMCplus and MMCmobile，支持CEATA specifications（ver1.0）
- 支持所有命令集，包括MMCA stream write and read
- 支持任意block数据长度
- SD时钟的最高工作为25MHz，不支持快速SD卡（50MHz）
- 支持SD卡的热插拔
- 内嵌数据CRC16和命令CRC7校验

3.2 基于 NOR Flash+NAND Flash 的启动方案设计实现

基于 NOR Flash 的启动方案是目前最常见的一种，有成熟的 BootLoader 软件代码[36][37]。本启动方案，以现有 NOR Flash BootLoader 为基础，并结合 NAND Flash 存储介质特点以及现有的硬件条件：ARM 处理器中的 NAND 控制器，采用 NOR Flash+NAND Flash 的方式，将 BootLoader 第一阶段的启动代码写入小容量的 NOR Flash 完成第一步的引导工作，而将 BootLoader 第二阶段的代码和较大的内核镜像存储于大容量的 NAND Flash 中，从成本、性能及可行性上上分析，这种方案相比较单独的 NOR Flash 启动更优。

本文在现有硬件模块支持的基础上，选用汇编语言完成该启动方案的 BootLoader 代码的编写。

3.2.1 BootLoader 方案设计

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 能提供更复杂的功能，以及更好的可移植性[26][29]。本文中基于 NOR Flash+NAND Flash 存储的设计将 BootLoader 分两个阶段来执行：Stage I 和 Stage II。考虑因素如下：

（1）基于编程语言的考虑。Stage I 主要用汇编语言实现，它主要进行与 CPU 核以及 NOR Flash 存储设备密切相关的处理工作，进行一些必要的初始化工作，是一些依赖于 CPU 体系结构的代码，

为了增加效率，而且因为涉及到协处理器的设置，用汇编语言实现，这部分代码可以直接在NOR Flash中执行。但是考虑到执行效率，同时避免在调试时进行NOR Flash的反复烧写，本文设计把它搬运至SDRAM中执行，这样将更加方便。Stage II主要用一般的C语言，来实现一般的流程以及对板级的一些驱动支持，这部分代码也在SDRAM中执行。

（2）代码具有更好的可读性与移植性。若对于相同的CPU以及存储设备，要增加外设支持的时候，Stage I的代码可以维护不变，只对Stage II的代码进行修改；若要支持不同的CPU，则只需在Stage I中修改。

（3）方便更新镜像。单阶段的BootLoader整体存放在固态存储设备的某一块连续的固定空间，不方便更新，而且容易造成数据被覆盖等问题；分两阶段执行的设计则不存在这个问题，两个阶段的程序分别存储于固态存储设备的不同空间中，方便更新。

综合Stage I和Stage II两部分的功能，基于NOR Flash+NAND Flash设计的BootLoader总体结构如图3-3所示：

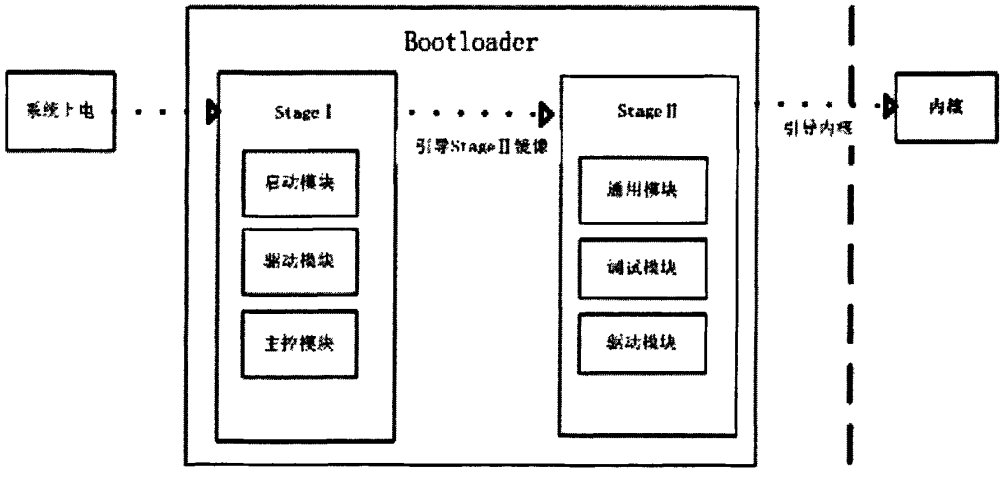

图3-3 BootLoader总体结构图

3.2.2 地址设计

图3-4为本论文镜像加载地址的布局图。对于本论文的硬件平台，外接2M的NOR Flash，由片选信号CSA选择，而SEP5010中是将片选CSA直接映射到0x00000000地址的，所以当选通CSA片选后，默认NOR Flash的起始地址就是0x00000000。在ARM体系结构中，上电或系统复位后将跳转到地址0x00000000处执行，该处存放的是复位异常中断的中断向量表。因此，将BootLoader的Stage I镜像存储于NOR Flash中，符合该要求。

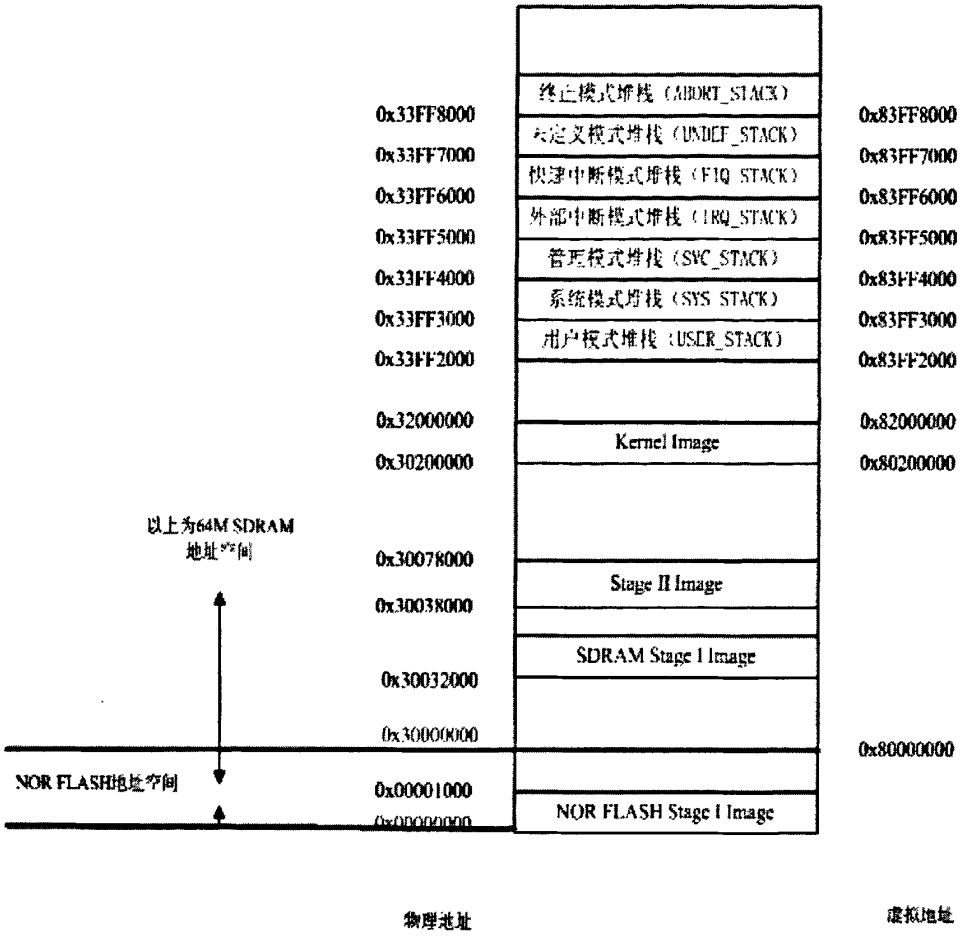

图 3-4 加载地址布局图

对于 SDRAM，则由 SEP5010 的片选型号 CSE 选择。Stage I 执行完成后，将自动加载存储于 NAND Flash 中的 Stage II 镜像到 SDRAM 中运行，Stage II 在 SDRAM 中的加载地址如图 3-4 所示。

3.2.3 基于 NOR Flash+NAND Flash 的 BootLoader 设计实现

BootLoader 启动第二阶段主要是关于内核镜像加载以及执行内存中的代码，不在本文的研究范围之内，因此关于 BootLoader 的启动设计主要是关于 Stage I。

当选用 NOR Flash+NAND Flash 作为外部启动存储介质时，上电后，PC 指针指向了 NOR Flash 的首地址，从这里开始执行代码。Stage I 主要完成从 NOR Flash 启动系统，初始化 CPU，然后读出存储于 NAND Flash 中的 Stage II 代码到 SDRAM 中运行的工作。其中 CPU 的初始化主要由两部分组成：将 EMI 控制器中的 SDRAM 配置为使能，PMC 控制器设置系统时钟频率和打开各个模块。其流程图如下所示：

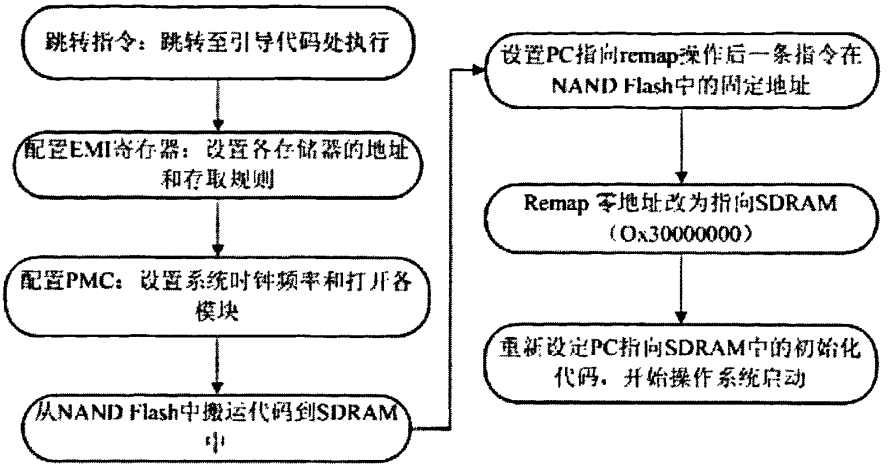

图 3-5 NOR Flash 的启动代码（BootLoader）流程图

Stage I 的设计用汇编语言进行编写，按照图 3-5 所示的启动流程图，其中的主要步骤用代码实现如下：

（1）建立异常跳转

ARM 有七种异常向量，在此只需要能够跳转到下面的代码执行即可，不需要其它异常设置，因此，将异常向量统一写为 RST_DO，这样 PC 指针即指向 BootLoader 代码处开始执行。

```
bal   RST_DO
mov r1，r1；  reserved exception
bal   RST_DO
```

（2）配置EMI寄存器

SEP5010 所使用的 SDRAM 由两片 32M 大小的 W982516BH75L 并联而成，其工作频率小于 135MHZ，行地址 13 位，列地址 9 位。主要为配置 SRAM 的读写周期、片选 CSA 设置（总线宽度 32bit，只读 Flash）、片选 CSF 设置（外接 SDRAM）以及 SDRAM 的时序配置。

```
ldr     r1，  =0x11000000 ；  THE ADD OF EMIADDR_SMCONF
ldr     r2，  =0xbb3337ff
str     r2，  [r1]              存储器参数配置
ldr     r1，  =0x11000014 ；  THE ADD OF EMIADDR_SDCONF1
ldr     r2，  =0x0118a077
str     r2，  [r1]              EMI的SDRAM 时序寄存器1
ldr     r1，  =0x11000014 ；  THE ADD OF EMIADDR_SDCONF1
ldr     r2，  =0x0118a077
str     r2，  [r1]              SDRAM 控制器的时序寄存器
```

（3）配置PMC模块

按顺序对时钟进行配置，ARM926EJ 的工作频率小于 300MHZ，SDRAM 的工作频率小于

135MHZ，基于这些原则，对 SEP5010 PMC 模块的相应寄存器进行配置（以下配置方案为典型配置方案之一：设置 ARM926EJ 的工作频率为 200MHZ，分频比为 1：2，总线频率为 100MHZ）：

```
ldr  r1,  =0x1000100c   ;  // PCSR 寄存器的配置
ldr  r2,  =0xffff       ;  // 首先打开所有模块
str  r2,  [r1]
ldr  r1,  =0x10001014   ;  // PMDR寄存器的配置
ldr  r2,  =0x1          ;  // 设置系统状态为Normal
str  r2,  [r1]
ldr  r1,  =0x10001000   ;  //PLTR寄存器的配置
ldr  r2,  =0x018000cd   ;  //该配置是一个参考值，PLL的稳定过渡时间
str  r2,  [r1]
ldr  r1,  =0x10001004   ;  //PMCR 寄存器的配置
ldr  r2,  =0X4199       ;  //配置PLL的PV，PD参数，设置系统时钟为200MHz
str  r2,  [r1]
ldr  r1,  =0x10001004   ;  //PMCR 寄存器的配置
ldr  r2,  =0X1228       ;  //最后使能PMC
str  r2,  [r1]
```

（4） 搬运代码

该步骤实现内核镜像的搬运工作，将主程序代码和内核镜像从 NAND Flash 中搬运到 SDRAM 以地址 0x30002000 开始的地方。

```
copy (kernel) to SDRAM
ldr  r3, =0x00000000
ldr  r1, =0x30002000   ;  搬运到SDRAM的目标地址
ldr  r2, =0x11000100   ;  代码在NAND Flash中的地址
```

在上述代码中，主程序的代码放在 NAND Flash 中，起始地址为 0x11000100，要讲其搬运到 SDRAM 内存中，它的起始地址为：0x30002000

```
LOOP
ldr   r4,  [r2],  #4
str   r4,  [r1],  #4
add   r3,  r3,    #1
cmp   r3,  #0x54000
bne   LOOP
```

（5） 地址重映射

代码和内核镜像搬运工作结束之后，进行地址重映射，实现将 0 地址从 NAND Flash 定位到 SDRAM，PC 指针指向 SDRAM，开始启动操作系统。

```
REMAP
```

```
ldr    r1，=0x11000010 ；//REMAP 0 ADDRESS TO SDRAM
ldr    r2，=0x0000000b ；//将CSE映射到0地址
str    r2，[r1]
ldr    PC，=0x30002000 ；//将PC指针指向函数入口处（即0x30002000）
```

3.3 基于 NAND Flash 的启动方案设计实现

NAND Flash 作为目前使用比较广泛的非易挥发存储介质，因其较低廉的每比特价格和较快的擦除和写入速度，获得了较广泛的应用。跟 NOR Flash 的总线数据读取方式不同，NAND Flash 采用 I/O 口和一些控制信号来实现数据的存取。EMI 模块中具备了 NAND Flash 接口单元，因此支持对 NAND Flash 的基本操作。

相比 NOR Flash，NAND Flash 中可以加载较大规模的代码和系统镜像，解决了单独使用 NOR Flash 启动时存储容量较小的问题，降低了应用成本。本文中基于 NAND Flash 的启动方案，采用了 NOR Flash 中 BootLoader 的启动流程。但是，由于 NAND Flash 不具备 NOR Flash 的 XIP 特性，CPU 不能直接读取运行，而且，NAND Flash 中的数据只能顺序地进行访问，因此在进行基于 NAND Flash 的 BootLoader 设计时要考虑到上述因素。

由于 NAND Flash 的 Page 大小分为 512 和 2K Bytes 两种不同类型，而且考虑到启动过程中对芯片内部片上 SRAM 的利用，因此，本文对基于 NAND Flash 的 BootLoader 设计分为是否用到 ESRAM，在通过 ESRAM 的启动设计过程又会考虑到 NAND Flash Page 大小不同之分。

3.3.1 NAND Flash 控制器

NAND Flash 控制器作为 AMBA 总线上的一个从设备，集成于 AHB 总线上[24]。如图 3-6 所示，主要模块包括总线接口模块、FIFO 缓冲模块、ECC 编码模块以及逻辑控制模块。

总线接口模块主要的功能是转换 AMBA 总线上的控制和数据信号：将总线上的数据送入 FIFO 或将数据从 FIFO 读出到总线上，将总线上的控制信号转换时序后送到控制模块。

NAND 控制器包含一个宽度为 32bit，深度为 4 的缓冲 FIFO，用于解决高速总线与低速设备之间数据传输速度的匹配问题。为提高总线的传输效率，以及控制器设计的便利性，NAND Flash 的数据传输采用 DMA 的方式来完成。譬如在读取 Flash 一页数据时，数据持续写入控制器 FIFO，FIFO 满时发出 DMA 传输的请求，同时暂停 Flash 的数据读取，控制信号 nRE 拉高，直至 DMA 响应请求即 FIFO 不满时，Flash 的数据传输重新开始。当选择应用的 Flash 位宽为 8，页大小为（512+16）B 时，控制器需要发出（32+1）次 4 拍字宽度的 DMA 传输请求来完成数据和校验信息的读取。

控制模块的工作主要是将总线接口转换的控制信号，按照 NAND Flash 的接口协议，将片选、地址、命令、读写使能按照所配置的时序要求，发送到 NAND Flash 中，并且控制数据的传输个数，以及 DMA 请求、数据传输完成中断、数据错误中断等系统信号。

NAND Flash 可靠性相对较差，存储器芯片中有坏块的存在，会导致存储数据出错。ECC 校验模块针对 NAND Flash 的可靠性问题，提供了一种查错、纠错的机制。ECC 校验码在数据读入时，

由硬件计算完成后写入到 Flash 的校验位中，当此页数据读出时，校验码再次生成与存储器校验位中的数据进行比较，若相同则没有损坏位，若不同，则给出出错中断，软件通过检查比较结果，判断出错位的位置进行纠错处理。纠错功能仅针对单 bit 位的出错，当一个以上位同时在一页中出现时，ECC 校验不能给出出错位正确的位置。

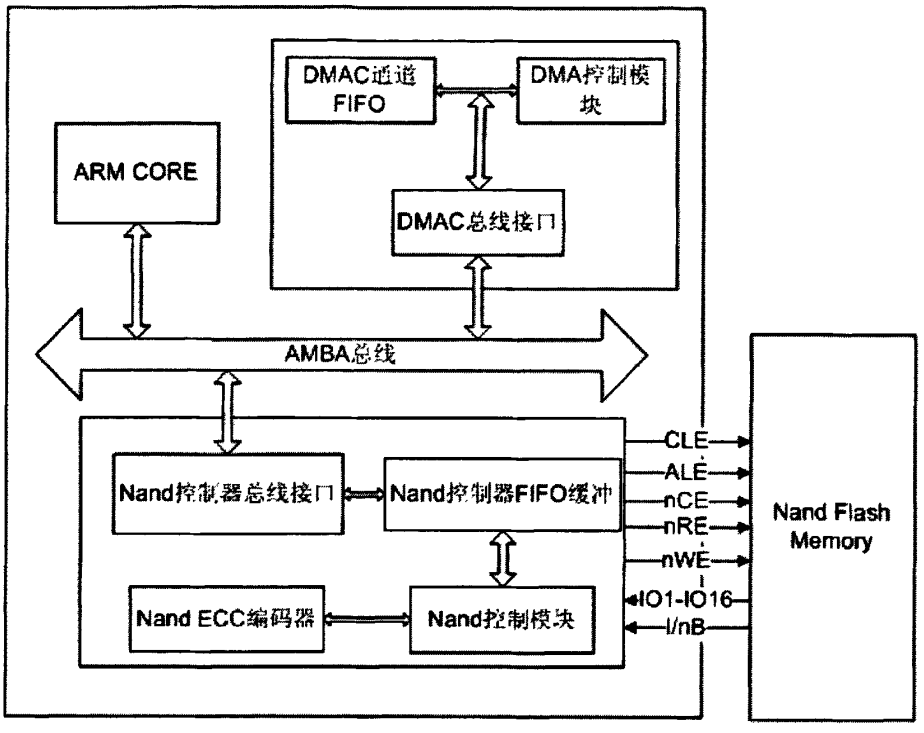

图 3-6 NAND Flash 控制器及系统架构

3.3.2 通过 ESRAM 的 NAND Flash 启动方案设计实现

如果选用 NAND Flash 作为外部存储介质，由于 NAND Flash 不具备指令直接执行的功能，所以默认的零地址不能置于 NAND Flash 所在片选。为了解决这个问题，本设计中提供了如下的解决方案：上电启动时，芯片中的 EMI 把 NAND Flash 的第一个 Page 的内容（即 BootLoader 代码）通过 DMA 搬到 ESRAM 中，并将 PC 指针指到 ESRAM 上开始执行。因此，在 NAND Flash 的第一个 Page 的内容便成为启动代码。和 NOR Flash 驱动一样，它在完成系统初始化后也要把放在 NAND Flash 中的其它待执行代码载到内存中，然后从内存中开始执行程序。

基于 NAND Flash 的总流程图如图 3-7 所示：

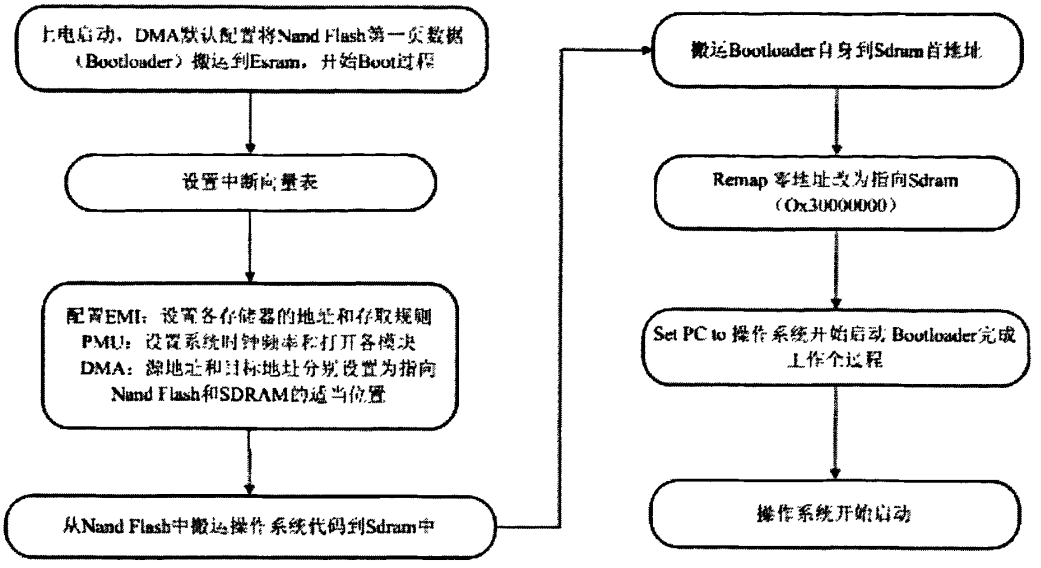

图 3-7 NAND Flash 经过 ESRAM 的启动代码流程图

下面依次介绍 NAND Flash 中 BootLoader 主要完成的工作：

（1）中断向量表和配置GPIO口

AREA text、CODE、READONLY

ENTRY

b RESET

b DEAD_LOOP；其他异常

中断向量表位于 BootLoader 的顶端，用于处理中断异常。

（2）配置DMA、PMC、EMI各参数；

首先配置 DMA 通道，清除传输完成和传输出错中断，然后屏蔽所有中断，配置 PMC 模块，打开所有模块的时钟。

ldr r1，=0x11000014；

ldr r2，=0x00004077；配置 SMCONF1（CSF）寄存器

str r2，[r1] ；

ldr r1，=0x11000018；配置 SMCONF2（SDRAM）寄存器

ldr r2，=0x80000500；

str r2，[r1] ；

EMI 中 NAND Flash 主要配置读写等待周期、读写周期、地址模式，SDRAM 配置：SDRAM 初始化、刷新时钟周期以及 Self_Reresh 期间刷新所有的行。

（3）从NAND Flash指定位置搬运操作系统代码到SDRAM中；

ldr r11，= 0x2000； 存放代码的 NAND 内部地址

ldr r8， = 0x30002000；读出代码的 SDRAM 目标地址

```
WAIT ldr r13，[r12]；    读出空闲寄存器内容
and r13，r13，#0x01
cmp r13，#0x01
bne WAIT；               还未完成页传输，继续等待
add r11，r11，#0x200
add r8，r8，  #0x200；   更新 NAND 内部页地址
```

这一步是 BootLoader 最重要的一部分，将操作系统代码从 NAND Flash 中读出到初始化完成后的 SDRAM 的指定位置，以备在 Boot 完成时把控制权交给操作系统，系统启动就完成了任务。

（4）复制启动代码本身到SDRAM首地址处

```
ldr   r0，=0x0
ldr   r1，=0x30000000；搬运目标地址（SDRAM）
mov r2，#0x200；         搬运数据量大小（Word）
```

这一部分代码是因为如果之前配置 Remap 寄存器的话，PC 寄存器值的值依然保持了从 0 开始计数的数值（比如 0x100），而 Remap 后零地址将指向 SDRAM 中 0x100 处，而此处没有任何代码，将导致启动失败。所以在 Remap 之前先将启动代码复制到 SDRAM 首地址，Remap 后启动程序得以继续进行。

（5）配置EMI Remap 寄存器，Set PC。

```
Remap
ldr   r1，=0x11000010
mov r2，#0x0b
str   r2，[ r1 ]；REMAP 第一步
ldr   r1，=0x1100001c
mov r2，#0x1
str   r2，[ r1 ]；REMAP 第二步
```

配置 Remap 寄存器为 0xb，使得零地址指向 SDRAM，PC 值指向 SDRAM 中启动代码后继位置，启动本身不受影响。此时 ESRAM 首地址为 0x1fff0000，而 SDRAM 首地址 0x0 或 0x30000000；

Set PC 为操作系统代码首地址处，操作系统开始启动。BootLoader 任务完成，系统正式开始启动。

3.3.3 NAND 启动新方法—不通过 ESRAM 的启动方案设计实现

一般情况下，片上存储器在作为启动代码转移阶石的同时，往往在启动后也有其特殊的作用。可以作为特殊的程序区，譬如在进行 MP3 解码过程中，核心解码函数作为频繁调用的程序，可以安排在片上 SRAM 中，以提高读取速度，提升系统性能。在 SoC 芯片开发过程中，在整体架构以及模块功能的变化之后，这块内嵌的 SRAM 失去了原来的作用，而仅作为 NAND Flash 启动时的代码跳板，对于整个芯片而言，付出的代价比较大。于是提出了在没有片上存储器的架构下，从 NAND Flash 启动的一种新模式。

在上述一般模式启动过程中，片上 SRAM 所起到的作用，就是执行 NAND Flash 中第一页的代码，将真正的启动代码引入到 SDRAM，最后将 PC 指针指向 SDRAM。在失去片上 SRAM 的支持后，可以在控制器的 FIFO 中去执行此段代码，这需要在硬件以及软件代码中作出适当的改变。

（1）首先需要改变的是地址映射机制，系统上电后，ARM 即从零地址开始执行指令，零地址映射到 NAND Flash 的 FIFO 入口地址，地址的译码过程由 AMBA 总线模块完成。在外部硬线 NAND Boot 拉高的条件下，AMBA 从设备地址译码模块在启动过程中，将零地址的设备选择权给到缓冲 FIFO。在第一页的指令执行完毕后，PC 指针也指向 SDRAM。

（2）其次是 NAND Flash 控制器在启动过程中，对数据的读取方式。鉴于 NAND Flash 大批量数据读写的特性，往往采用 DMA 方式对数据进行操作。启动过程中，由 ARM Core 直接向 FIFO 读取数据，在 FIFO 读空的情况下，将从设备 Ready 信号拉低，等待 NAND 中的数据读出。并且在此读取过程中，DMA 的请求被屏蔽。

（3）存储器首页的代码是在缓冲 FIFO 中执行的，FIFO 的人口地址是一个高 24 位的选通地址，因此当系统启动时，零地址开始增加，对 FIFO 中读出的指令而言，低 8 位地址的变化是无关的，FIFO 始终被选通，指令的输出是默认的顺序输出。这就要求首页的代码中不可以出现循环、跳转等语句，并且要求在 128 条指令内完成需要的操作。

启动的流程如图3-8所示：

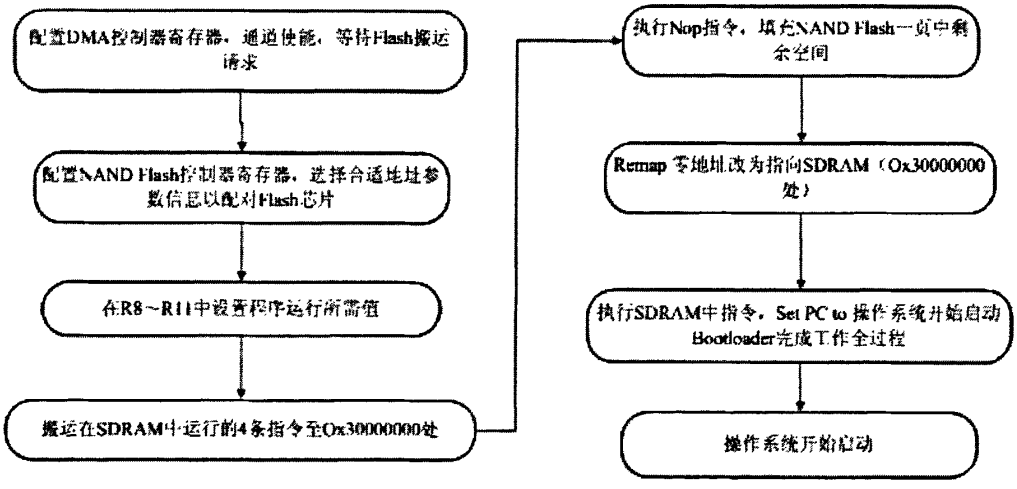

图 3-8 NAND Flash 不经过 ESRAM 的启动代码流程图

（1）配置 DMA 控制器的 4 个寄存器，通道使能后，等待 Flash 发出的搬运请求；

```
mov   r1，#0xl1000000    // 设置 DMA 寄存器基址
mov   r0，#0xl000
add   r1，r2，  r1        // DMA 源地址寄存器地址
str   r1，[r0]，#0x8      // DMA 源地址 0xl1000200
add   r1，r1，  r2
str   r1，[r0]，#0x8      // DMA 目标地址 0x30001000
```

```
add  r1，r1，  #0x9b
str   r1，[r0]，#0x4      // DMA 控制寄存器 0x21249b
add  r1，r1，  #0x5
str   r1，[r0]              // 配置 DMA 通道使能寄存器 0x305
```

（2）配置 NAND Flash 控制器的 3 个寄存器，选择适合的地址、时序参数与所用的 Flash 芯片吻合；

```
mov  r0，#0x11000000
add  r0，r0，  #0x100  //地址寄存器地址 0x11000100
mov  r1，#0x100
str   r1，[r0]，#0x18   //首页地址 0x100
add  r1，r1，  #0x57
str   r1，[r0]，#0x10   //配时序寄存器 1 0x6202857
add  r1，r1，  #0x53
str   r1，[r0]              // 配时序寄存器 2 0x514353
```

（3）将需要在 SDRAM 中运行的指令搬入 SDRAM 0x30000000 处，寄存器的值运行后 NAND Flash 开始传输；

```
mov r1，#0xe2000000
mov r2，#0x590000
add r1，  r1，  #0x1
str  r1，[r0]，#0x4     //0x30000004   0xe2599001
add r1，r1，  #0xfd
str  r1，[r0]，#0x4     // 0x30000008   0x1afffffd
add r1，r1，  #0x8
str  r1，[r0]，#0x4     // 0x3000000c   0xe1a0f008
Nop
…                          // 插入 Nop 指令填满一页
```

（4）执行在页末的指令，将 PC 指针指向 SDRAM 的 0x30000000 处；

```
mov r0，#0x30000000
mov pc，r0     // PC 指向 0x30000000 SDRAM
```

执行 SDRAM 中的指令，首先启动 NAND Flash 的数据传输，将程序搬往 SDRAM 的 0x30001000 处。其次执行一个循环语句，等待第一页的程序搬完，之后将 PC 指针指向 0x30001000 处，启动程序从 0x30001000 处正式开始执行。

3.4 基于 SDHC/SD 卡的启动方案设计实现

随着消费类电子产品音视频等多媒体功能的不断增强，系统对于存储介质的安全、容量、性能的需求不断提高，SD 存储卡支持符合 SDMI 标准的版权保护机制，并且具有更快的传输速度和更大

的存储容量，因此越来越多的消费类 SoC 集成了 SD 卡控制器，以扩展系统对移动存储介质的支持。

SD 卡多用于 MP3/MP4 随声听、数码相机等各种消费类电子嵌入式产品中。SD 卡容量大，读写速度快，而随着目前用户视频体验的增强，对存储卡容量和速度要求的提高，传统 SD 卡低于 2GB 的容量已经不能满足使用要求。最新公布的 SDHC（SD High Capacity）技术规范，主要针对 2GB 容量以上大容量高速数据存储和读取的需求。该技术标准采用 FAT32 文件系统（传统 SD 卡为 FAT12/16），可以支持最高 32GB 的存储容量。SDHC 卡延续了传统 SD 卡的外形特点以及对 CPRM 保护技术的支持，支持 FAT32 文件系统的新设备除了能使用 SDHC 之外，仍可以使用原有的 SD 卡。

目前，对于涉及到大容量数据应用的系统，一般选择使用 NOR Flash+SD 卡+SDRAM 的存储方案：NOR Flash 用来完成系统的启动功能，SDRAM 运行应用程序，SD 卡存放各种类型的数据。如果系统可以直接从 SD 卡启动，就可以省去 NOR Flash，降低系统的成本和硬件的复杂度。所以，本文提出一种系统直接从 SD 卡启动的方案。该方案实现的难点在于，对于目前常见 SoC 集成的 SD 卡控制器，一般不能支持直接启动。因此，本方案设计过程中，需要对 SD 卡控制器进行一定的修改，使其能够支持直接启动。同时配合相应的 BootLoader 代码，就可以实现 SDRAM+SD 卡的低成本大容量的存储方案。

3.4.1 常见 SD 卡控制器介绍

SD 控制器集成于芯片内部，用以与 SD 卡进行通信，实现系统的 SD 功能扩展 SD 卡控制器通过 SD 总线对 SD 卡进行初始化以及读写操作，SD 卡控制器结构如图 3-9 所示：

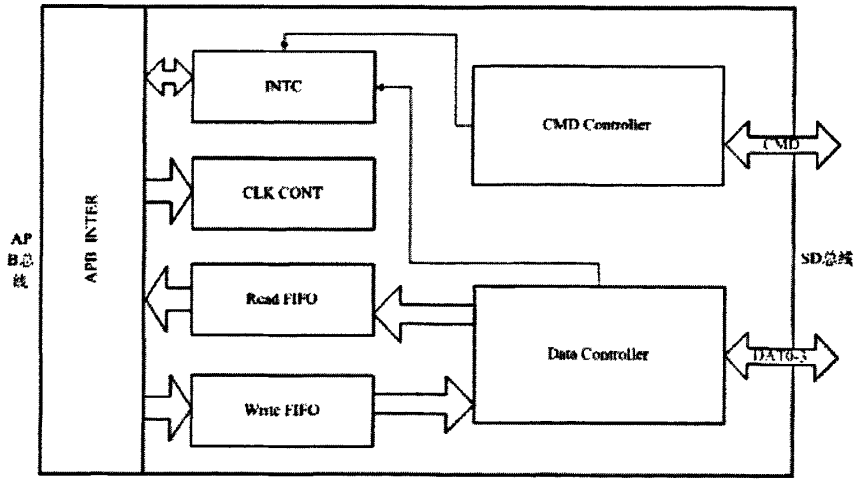

图 3-9 SD 卡控制器模块结构

SD 总线包括时钟线 CLK，命令线 CMD，数据线 DAT0-3 等。SD 总线上的通信基于命令和数据的比特流，命令和数据的第一位是起始位，最后一位是停止位。控制器通过发出命令发起操作，命令通过命令线传递给 SD 卡，SD 卡发出响应作为对于命令的回应，响应也在命令线上传输。控制器接受并分析响应以确定 SD 卡的状态，并根据 SD 卡的状态进行下一步的操作。其主要特性如下所示：

- 与 SD Memory Card Specification Version 1.01 完全兼容；

- 支持 SD 卡 1/4 位传输模式；
- 支持单/多 Block 传输，Block 的长度以及数目可以配置；
- 读数据超时的周期数可以配置；
- 响应超时周期固定配置为 64 个周期；
- 读写数据以及响应的 CRC 检查，CRC 出错产生相应中断；

根据工作原理和功能分析，SD 卡控制器可以分成如图 3-9 所示的 7 个子模块：SD 时钟控制模块（CLK CONT）、命令收发控制器（CMD Controller）、命令收发控制器（CMD Controller）、数据收发控制器（Data Controller）、中断控制器（INTC）、读数据缓冲器（Read FIFO）、写数据缓冲器（Write FIFO）和 AMBA 总线接口模块（APB Interface）。

（1）SD 时钟控制模块

由于 APB 总线频率一般在 70MHz 以上，而普通 SD 卡的正常工作的最大频率为 25MHz，因此 SD 卡时钟需要对 APB 时钟进行分频。SD 卡在初始化模式下时钟频率不能超过 25MHz，驱动软件配置分频控制寄存器使分频电路产生这两个频率范围的时钟，为了降低 SD 卡控制器的动态功耗，在控制器不工作时应关闭 SD 时钟，时钟的开关也由驱动软件控制。

（2）命令收发控制器

命令收发控制器控制发送命令和接收响应的时序并完成串并转换。在发送命令时，将命令由并行转换为串行并以 SD 时钟同步发出；在接收响应时，将串行的响应由串行转换为并行并存储。所有的命令和响应都采用 CRC7 校验。CRC7 校验的硬件电路由 7 级移位寄存器和两个加法器组成，各移位寄存器初始化为“0”，命令随着时钟同步串行的移入，当命令全部移入后，从寄存器输出 CRC7 校验码。

（3）数据收发控制器

读操作时，数据控制器将接收的串行数据转换为并行数据并存入读缓冲区；写操作时，数据控制器从写缓冲区中取出并行数据后串行发出。数据读写也需要经过 CRC 校验，数据采用 CRC16 校验，CRC16 和 CRC7 的算法本质相同，只是增加了 CRC 校验的位数从而提高了 CRC 校验的精度。对 SD 卡进行写操作时，数据收发控制器还要检测 SD 卡发回的 CRC 状态来判断写操作是否成功，若操作失败则需要重传。

（4）读/写数据缓冲器

由于 APB 总线上连接多个功能模块，如 I2C、SPI 等都需要占用总线进行数据传输，SD 卡控制器只能通过发出总线请求在有限时间占有总线。这使得在进行 SD 卡读写操作时可能由于申请不到总线来不及存储已经收到的数据或者来不及获取新的发送数据，增加读/写数据缓冲器实现数据暂存可以很好的解决这个问题。缓冲器的大小需要根据 SD 总线与 APB 总线的频率对比以及 APB 总线的繁忙程度来选择，在本 SD 卡控制器中读/写数据缓冲器都采用 8 级 16 位宽的 FIFO。

（5）中断控制器

当 SD 卡控制器完成了某项操作或者工作异常的时候，中断控制器会产生相应的中断，软件可对不同的中断做出相应的处理。中断包括读/写缓冲区已满/空、读/写缓冲区读写出错、命令发送完毕、数据传输完成、响应超时、读写数据 CRC 出错等。软件可以通过配置中断控制器屏蔽或使能每

个中断。

（6）AMBA 总线接口模块

ARM 内核通过 APB 总线读写 SD 卡控制器的寄存器和数据缓冲区，而该模块作为 APB 总线与 SD 卡控制器的接口，它将控制器内部的寄存器和数据缓冲区映射到统一的地址空间，使内核可以通过地址访问。

综上所述，SD 卡控制器要为 SD 卡提供频率可配置的时钟，控制器还要负责发出命令接收响应以及发送/接收数据，为保证传输的正确性控制器还要对命令、响应、数据进行 CRC 校验。

3.4.2 系统架构的设计

基于 SDHC/SD 卡存储器的启动方案由软、硬件两部分组成，在将启动代码由 DMA 控制器载入到系统的片上存储空间 ESRAM 之前需要通过硬件设计完成 SD 控制器对 SDHC/SD 卡的初始化操作；软件代码实现系统的启动配置，完成硬件初始化，设置中断向量表，配置系统时钟、中断控制器、内存控制器、DMA 控制器等，并且完成堆栈设置以及地址重映射等功能。如图 3-10 所示，整个系统采用 AMBA 总线架构，芯片内置 ESRAM，通过 EMI 控制器外接片外存储器，DMA 模块实现数据在外设和存储器之间的直接传输，PMC 产生系统运行所需要的时钟并进行系统功耗的管理。

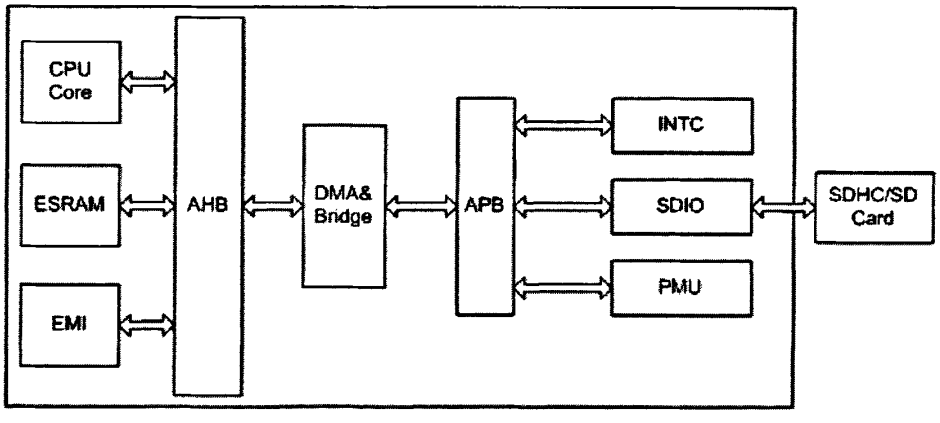

图 3-10 系统构架框图

3.4.3 SD 卡控制器的改造

为实现从 SDHC/SD 存储卡的启动，SD 卡控制器的改造过程中需要考虑以下几个功能的实现：

（1）对于不同的启动模式，在上电复位以后，需要对 AMBA 总线的地址译码电路进行不同的设计，实现不同的零地址映射。

（2）在功耗管理模块中应保证系统启动时与启动相关的功能模块如 ESRAM、DMA、SD 控制器的时钟是默认打开的。

（3）SDHC/SD 卡在上电之后首先要进行初始化配置，因而在上电之后，默认由 DMA 控制总线，直到完成数据载入的传输之后释放总线，CPU Core 从零地址开始执行程序。

（4）由于启动过程中涉及到通过 DMA 实现代码的搬运，因而 DMA 的初始参数配置和时序配置应满足完成代码传输的需要。

（5）由于 SDHC/SD 卡启动对于程序员的设计而言是一个黑盒子，因而在保证 Boot Controller 设计健壮性的同时，应该考虑设计中友善的接口，在启动失败的时候应有指示信号以便于程序员调试。

上电启动以后，SD 控制器需要对 SDHC/SD 存储卡进行初始化以及数据载入操作，为了更好的实现该功能，在已有的代码基础上加入 SD Boot Controller，新建 SD _TOP 实现两个模块的连接，如图 3-11 所示。作为整个芯片设计的启动方案之一，模块设计中采用 Boot_en 信号作为使能信号，并在启动失败时输出 Boot_error 信号作为芯片调试信号。

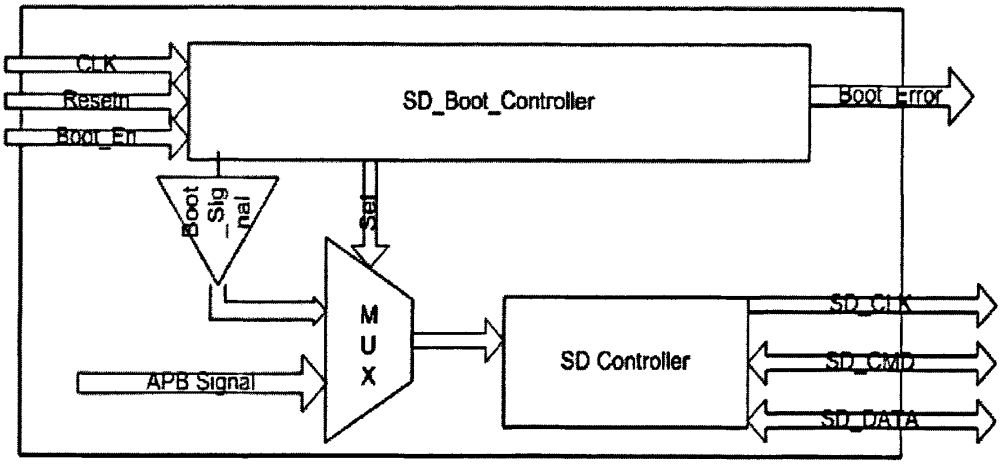

图 3-11 SDHC/SD 卡启动控制器的顶层设计

硬件设计的启动控制器对存储卡进行初始化，将 SDHC/SD 卡设置从卡确认状态转换进入数据传输状态，完成单线多块（4K）数据的传输。

如图 3-12 SDHC/SD 卡初始化流程图及图 3-13 硬件启动控制器的状态转换图所示，在系统上电复位和时钟稳定之后，硬件启动控制器进入 BOOT_IDLE 状态对 Boot_en 信号采样，当信号有效则进入 START 状态，否则进入 IDLE 状态。

在 START 状态首先检查是否有 SD 卡连接在卡槽上，之后完成模块输入时钟配置、清除状态寄存器、更新输出时钟等操作，此时应注意将初始化过程中的输出时钟频率设定在低于 400KHz。在完成以上设置之后发送 CMD0 命令，应考虑到在上电后存储卡需要初始化延时（74 个输出时钟周期，最大时间不超过 1 毫秒），因而在 CMD0 发送之前等待 80 个 clock 时钟周期。由于 CMD0 没有返回值，因而在命令发送完成后需要等待至少 8 个输出时钟周期，进入 CARD_CHECK 状态，发送 CMD8：如果在规定 CLK 内 CMD8 没有返回 Response，则判断该卡为符合 SD Version1.X 协议的普通 SD 卡，如果有返回 Response，则判断为符合 Version2.0 协议的 SD 卡，即为 SDHC；进入 CRESET_SDAPPCMD 状态后，发送 CMD55 命令，如果接收到响应，则是 SD 卡连接在卡槽上，进入 CRESET_SDOPCOND 状态，继续完成 ACMD41 命令的发送。

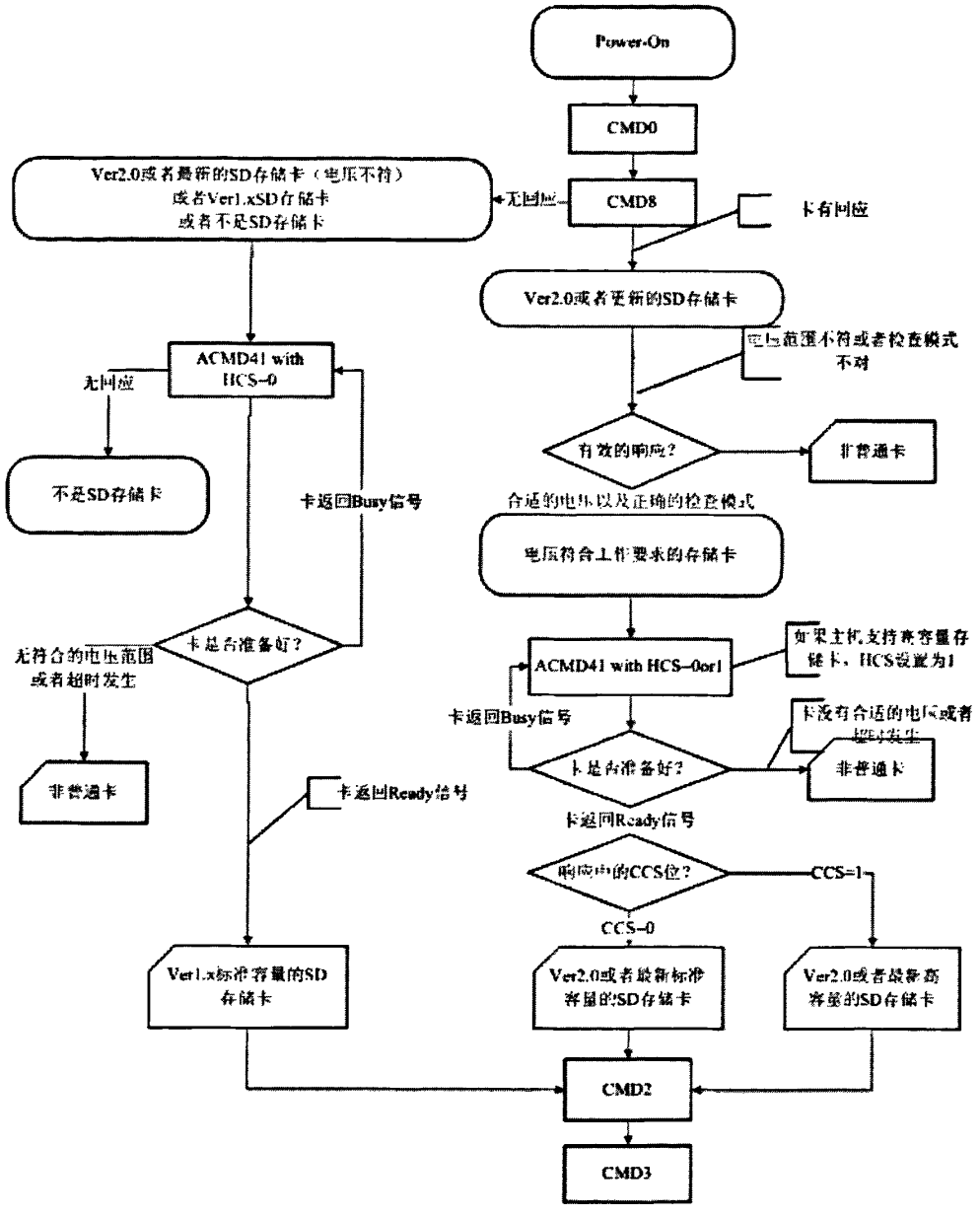

图 3-12 SDHC/SD 卡初始化流程

对 ACMD41 在命令线上返回的 OCR（Operation condition register）寄存器值，如果包含 Busy 位（置 0），显示卡仍然在上电启动或者重启过程，还没有为后面的初始化准备好通信，需要重新发送命令，直到 Busy 位被清除（置 1），进入 READY 状态。

对于单张卡，上电启动过程的最大周期不可以超过一秒，所以当命令返回错误的时间或者 Busy 未被清除的时间超过一秒，则判定启动过程失败，进入 IDLE 状态并且显示 Boot_error。在 READY 状态发送 CMD2 命令，获取存储卡的 CID（Card identification number）信息，之后进入 IDENT 状态，针对已经判定的存储卡类型，对 SD 存储卡发送 CMD3 命令获取 RCA（Relative card address）。当

完成 CMD3 的发送后，状态机进入 STANDBY 状态，此时可以更新输出时钟，使存储卡工作于更高的工作频率，并发送 CMD7 命令将存储卡的状态由等待（STANDBY）转换到传输（TRANSFER）。在 TRANS 状态，设置 SD 控制器的发送 FIFO 和接收 FIFO 阈值、发送数据的大小，并发送 CMD17 及 CMD12 用于传输 4K 数据，如果数据传输过程中出现数据 CRC 校验错误或读数据超时将会显示 Boot_error。

整个启动过程中所涉及到的 SD 命令如下：

CMD0：GO_IDLE_STATE　设置所有存储卡进入 IDLE 状态；

CMD8：SEND_IF_CON　该命令用来判断用于数据存储的 SD 卡类型；

ACMD41：SD_APP_OP_COND　设置存储卡返回 OCR（Operation Condition Register）参数；

CMD2：ALL_SEND_CID 存储卡发送 CID（Card Identification Number）至控制器；

CMD3：SEND_RELATIVE_ADDR　存储卡返回 RCA 相对地址；

CMD7：SELECT/DESELECT CARD 状态转换命令，设置存储卡进入等待/传输状态；

CMD18：READ_MULTIPLE_BLOCK 读取 SD 卡中多块数据命令；

CMD12：STOP_TRASMISSION　存储卡数据读取停止命令；

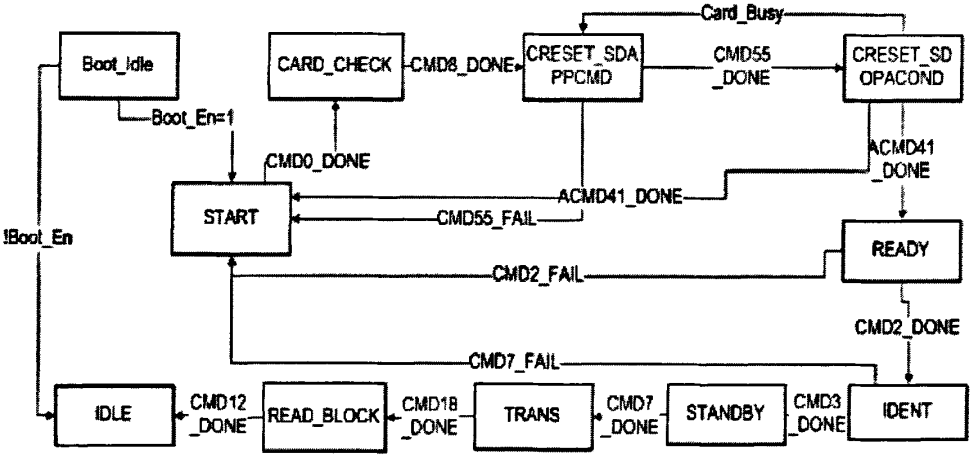

图 3-13 硬件启动控制器的状态转换图

3.4.4 启动方案的软件设计

图 3-14 所示为基于 SD 卡启动的软件流程。在 SD 控制器完成数据的载入之后，CPU Core 获得 AMBA 总线的控制权，开始执行启动程序。首先在零地址读取中断向量异常表，显示切换到 SVC 模式，这时禁止所有中断，包括中断控制器的 IRQ 和 FIQ 中断，以及屏蔽 CPU Core 中的中断使能位。由于启动数据载入之后，SD 控制器和 DMA 控制器会指示出相应中断状态，此时应先清除中断状态，再配置 PMC 和 EMI 控制器，使系统切换到正常工作状态。此时可以从 SD 存储器将体积更大的可执行镜像文件载入到速度和容量更大的 DDRAM 中，配置重映射寄存器之后，在新的存储介质 DDRAM 中运行可执行文件。

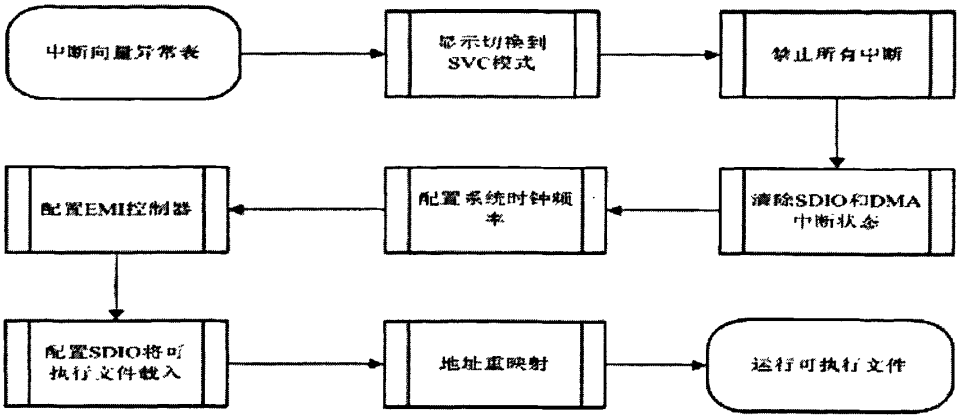

图 3-14 基于 SDHC/SD 卡启动的软件流程

3.5 本章小结

本章结合课题研究现有的硬件资源，从最常见的基于 NOR Flash 的 BootLoader 入手，考虑到 NOR Flash 芯片内执行的特性以及 NAND Flash 低成本、大容量存储的特点，设计一种基于 NOR Flash+NAND Flash 的启动方案。整个设计分为 BootLoader 的设计，代码和内核镜像加载地址的设计，并选用汇编语言完成 BootLoader 代码的编写。

基于大容量存储介质 NAND Flash 的启动方案，结合 NAND Flash 控制硬件模块，则根据是否使用芯片内部 ESRAM，分为通过 ESRAM 和不通过 ESRAM 的 NAND Flash 启动，两种启动方案适用于不同的 ARM 芯片，采用汇编语言设计实现。

SD 卡具有大容量、读写速度快以及可移动性等优点，使用现有的 SD 卡控制器模块，设计基于 SD 卡的启动方案。该启动设计分为软件流程设计和硬件启动控制两个部分。软件流程介绍了 BootLoader 的实现过程，用 C 语言设计实现。硬件启动控制器的功能主要是完成 SD 卡控制器的改造，以实现改控制器对直接启动的支持，该部分用硬件描述语言设计实现。

本章以第二章介绍的 BootLoader 相关知识为基础，分析三种不同存储介质的特点，从理论上完成各自的启动方案设计，并用语言描述实现，为下一步的验证工作做好铺垫。

第四章 设计验证与评测

4.1 设计验证

设计验证，顾名思义就是通过仿真、时序分析、上板调试等手段检验设计正确性的过程。在复杂的FPGA/IC设计流程中，其他环节由于需要人为干预的东西比较少，例如综合、布局布线等流程，基本所有的工作都由工具完成，设置好工具的参数之后，结果很快就可以出来，因此所花的时间精力要比验证少的多。而前仿真和后仿真涉及验证环境的建立，需要耗费大量的时间，因此验证在整个设计流程中占用了60%～70%的时间。FPGA/IC的设计流程中，属于验证的有功能验证仿真和时序验证两个步骤。

相比较时序验证而言，功能验证更受重视，特别是现在FPGA/IC设计都朝向SoC（System on Chip，片上系统）的方向发展，设计的复杂度都大大提高，如何保证这些复杂系统的功能是正确的成了至关重要的问题。功能验证对所有功能进行充分的验证，尽早地暴露问题，保证所有功能完全正确，满足设计的需要。

本文的设计验证部分分为仿真验证和FPGA验证两个部分：

- 在仿真验证中，基于NOR Flash+NAND Flash和NAND Flash的启动设计主要内容为BootLoader软件代码编写，这两种方案是根据实验室现有的硬件模块条件完成。由于硬件模块的基本功能已经历过多次功能验证，稳定性可靠，因此无需对模块再次验证。基于SD卡的启动方案设计由于采用硬件语言描述了了一个新的SD Boot控制器，该控制器主要完成SD卡的初始化以及数据的读取操作，所以有必要在实验室现有VCS软件环境下，验证新建SD Boot控制器的功能是否正确和完善。
- FPGA验证，这是本文验证工作的重点。基于实验室现有的FPGA平台，从验证硬件平台的搭建开始，经过BootLoader和系统内核的烧录，到最后启动是否顺利实现，完成三种基于不同存储介质的BootLoader的验证，可以检验设计的启动方案的正确性和可靠性。在完成启动功能的前提下，并测试三种启动方案的速度，分析测试结果，讨论不同方案各自的适用环境

4.2 启动设计的仿真验证

做仿真验证时，需要建立验证环境，以便对设计施加特定的输入，然后对设计的输出进行检查，确实其是否正确。在实际验证工作中，一般采用由Testbench和被测模块组成的验证体系，如图4-1所示：

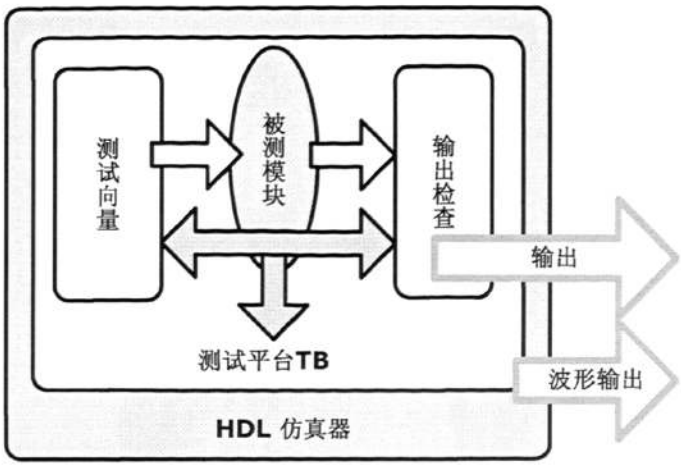

图 4-1 仿真测试平台示意图

这是验证系统普遍适用的模型，Testbench 为被测模块提供输入，然后监视输出，从而判断测试模块的工作是否正确。验证工作的难度在于确定应该输入何种激励，相应的正确的输出应该是怎样的。

基于 SD 卡启动设计的功能验证是在 SEP5010 集成环境下进行，如图 4-2 和图 4-3 所示，SD Boot 控制器的功能仿真结果。

进行模拟时，采用系统级仿真，通过加载可执行的二进制代码，仿真整个系统启动过程的实际环境。由于受仿真模型的限制，在测试过程中首先需要将可执行的代码写入 SD Model 中，然后重新启动系统。在重置 Reset 信号的同时，更改系统启动模式，通过图 4-2 可见，在 15000000ns 时刻，系统重置，SD Boot 控制器在完成对卡的初始化之后，在 18000000ns，DMA 开始数据载入。

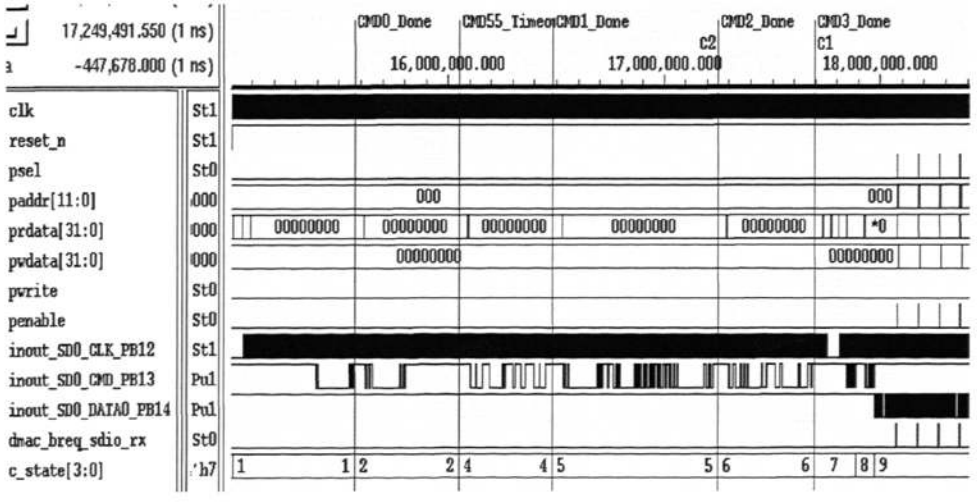

图 4-2 硬件初始化及数据载入仿真波形图

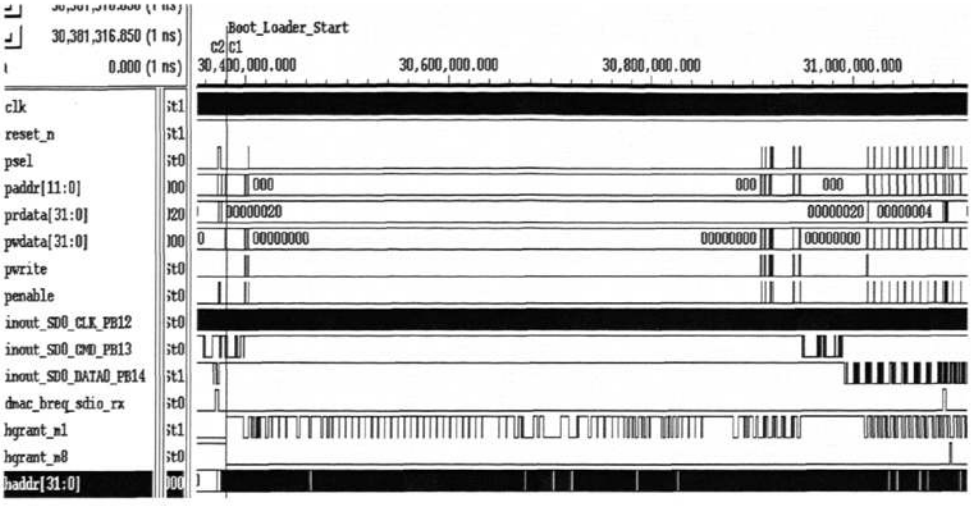

图 4-3 软件启动仿真波形图

由图 4-3 可见，在 30400000ns，数据传输完成，AMBA 的控制权由 DMA（图中 hgrant_m8）转交给 CPU Core（图中 hgrant_m1），在完成系统的初始化配置之后，再次从 SDHC/SD 存储器中读出可执行的内核操作代码。由仿真波形可见，模拟结果正确，SD Boot 控制器的 SDHC/SD 卡初始化以及数据读取功能都能正确实现。

4.3 启动设计的 FPGA 验证

FPGA 具有速度快、易擦写、结果直观等优点，FPGA 技术的快速发展使得很多 ASIC 设计可以在流片前进行系统原型的验证。FPGA 开发板使设计者可以完成板级的软硬件协同调试验证，节省 SoC 芯片的研发时间，降低流片失败的风险。采用 FPGA 进行系统功能验证，相比 RTL 级软件环境下的逻辑仿真速度要快好几个数量级。

FPGA 快速原型体现了芯片设计的硬化，所以 FPGA 验证成功可以确保硬件设计方案的可行性，给流片提供依据。

4.3.1 FPGA 验证环境

FPGA 验证平台采用基于 Stratix III EP3SL150 的开发板。Stratix III FPGA 采用了 TSMC 的 65nm 工艺技术,其突破性创新包括硬件体系结构提升和 Quartus II 软件改进,与前一代 Stratix II 器件相比,这些新特性使功耗降低了 50%,性能提高了 25%,密度是其两倍。EP3SL150 逻辑单元达到 150K，在所有高密度、高性能可编程逻辑器件中，其功耗最低，适合高性能计算、新一代基站、网络基础设施以及高级成像设备等多种应用。开发板如图 4-4 所示：

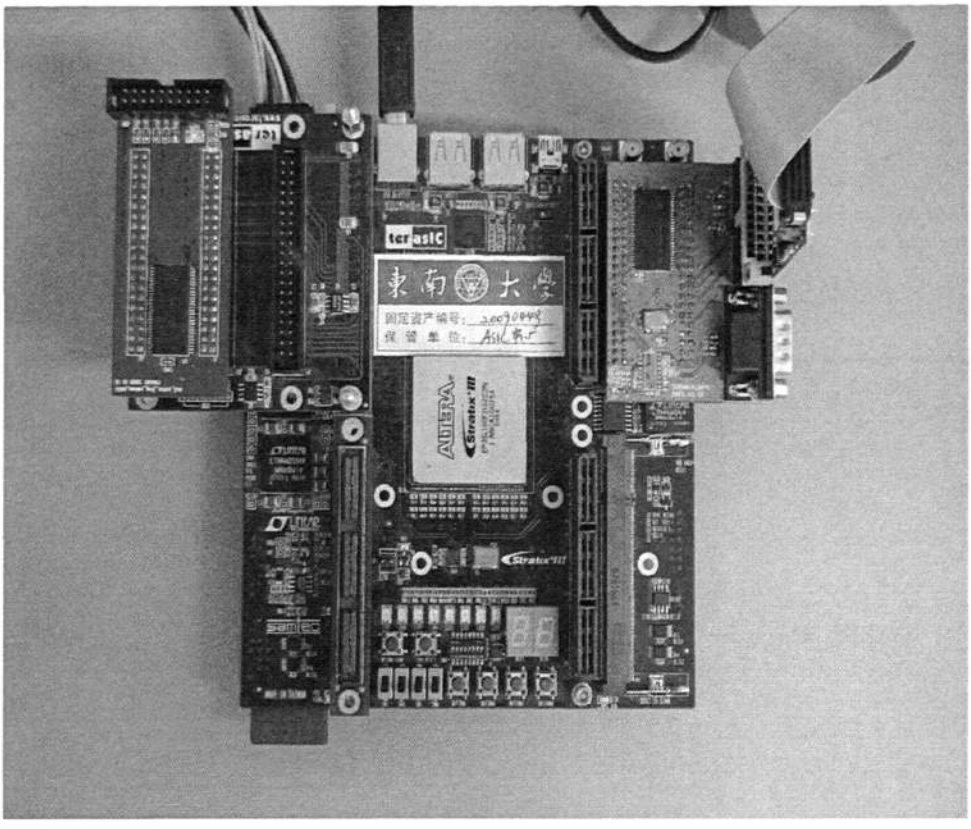

图 4-4 FPGA Stratix III EP3SL150 开发板

根据照片所示可以看到，左上角为扩展接口，验证平台所需的 NOR Flash 和 NAND Flash 均焊接在自制的 PCB 板上，通过扩展接口来实现和系统的连接。照片左下角部分为 FPGA 电路板的 SD 卡插槽部分，FPGA 平台通过该插槽来扩展对移动存储介质的支持。图 4-5 为 Stratix III 平台和 SD 卡连接示意图。

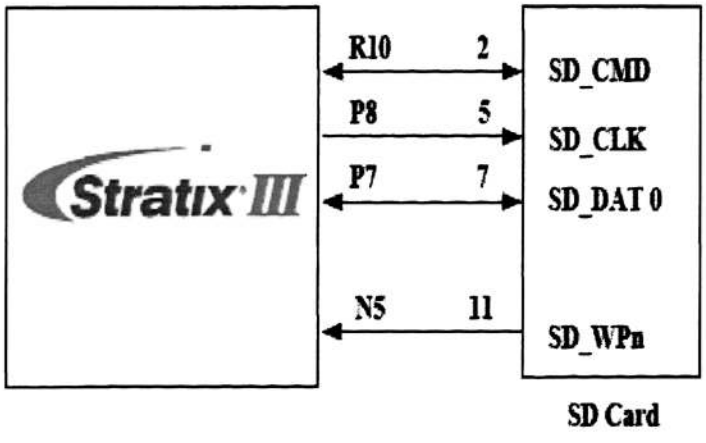

图 4-5 Stratix III 平台和 SD 卡连接示意图

（1）FPGA 硬件调试环境：

- Stratix III EP3SL150 开发板
- NOR Flash 型号：TE28F160C3

该 Flash 存储器件提供兼容 16 位数据总线的封装以及高性能的异步读取能力。支持单个 Block 可擦写的特性更适于代码编辑和数据存储。内存映射的顶部或底部分布了 8×4K 的参数块，其余的内存阵列分为 32K 字的主要块。该器件可以在不同的 I/O 电压（1.8V 和 3V）下支持阵列阅读的工作模式，以及在 3V 和 12V 的外接电压下支持擦除和编程操作。

- NAND Flash 型号：K9F1208

K9F1208 是 Samsumg 公司生产的 512Mb(64M×8 位)NAND Flash 存储器。该存储器的工作电压为 2.7～3.6V，内部存储结构为 528 字节×32 页×4096 块，页大小为 528 字节，块大小为 (16KB+512 字节)；可实现程序自动擦写、页程序、块擦除、智能的读 / 写和擦除操作，一次可以读 / 写或者擦除 4 页或者块的内容，内部有命令寄存器。

- Kingston 1G SD 卡

支持CPRM。两个可选的通信协议：SD模式和SPI模式；可变时钟频率0－25MHz；通信电压范围：2.0-3.6V；工作电压范围：2.0-3.6V；低电压消耗：自动断电及自动睡醒，智能电源管理；高速串行接口带随即存取，支持双通道闪存交叉存取快写技术；最大读写速率：10Mbyte/s；数据寿命：10万次编程/擦除。

（2）FPGA 软件调试环境：

- ARM RealView Development Suite（RVDS）开发套件

RVDS 是 ARM 公司继 SDT 与 ADS1.2 之后主推的新一代开发工具。RVDS 集成的 RVCT 是业内公认的能够支持所有 ARM 处理器，并提供最好的执行性能的编译器；RVD 是 ARM 系统调试方案的核心部分，支持含嵌入式操作系统的单核和多核处理器软件开发，可以同时提供相关联的系统级模型构建功能和应用级软件开发功能，为不同用户提供最为合适的调试功效。

RVDS 的突出特性是：支持 ARM 新架构下的编译和调试，包括支持 V7 指令集和 NEON 技术，支持 Cortex A8 和 M3；RVD 可以直接连接到 SoC Designer；支持 CoreSight 调试技术； Eclipse / Codewarrior 集成开发环境；丰富的项目管理系统：基于 Eclipse 的项目管理器，能支持 Linux，Windows 平台。

- OrCAD+ Power PCB

该软件用来实现 NOR Flash 和 NAND Flash 扩展电路的设计：NOR Flash 和 NAND Flash 的连接原理图如图 4-6 和 4-7 所示：

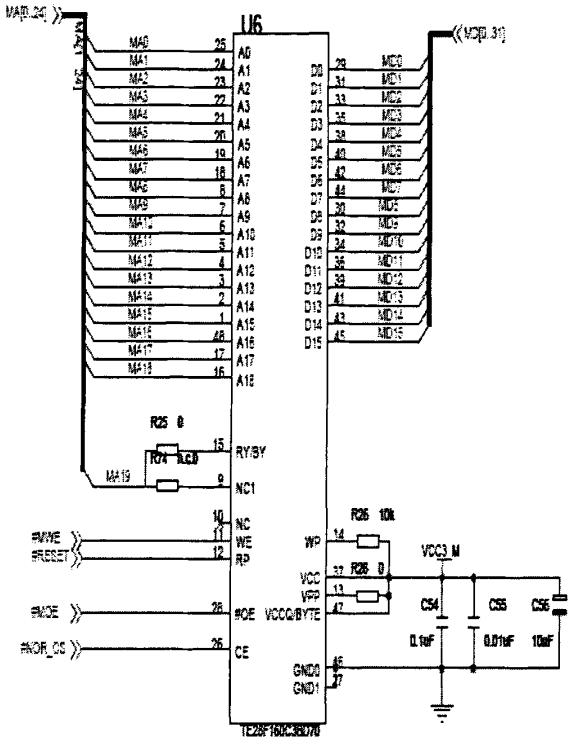

图 4-6 NOR Flash 连接原理图

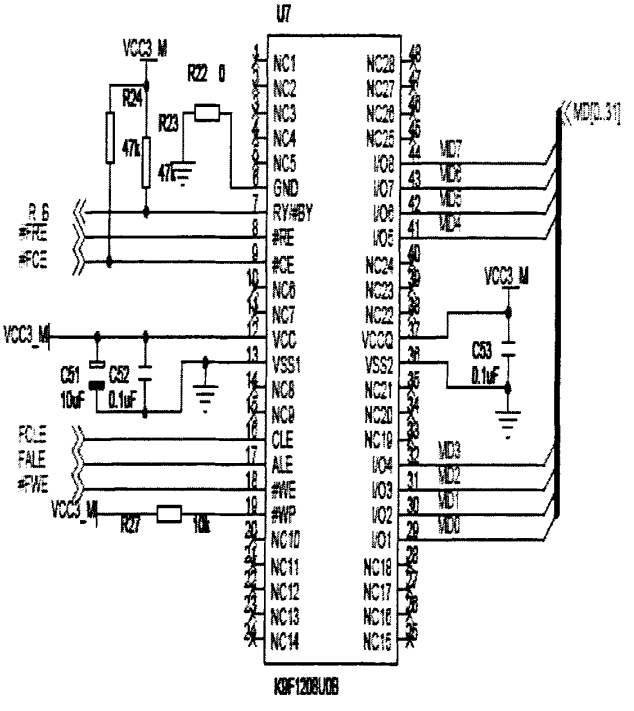

图4-7 NAND Flash连接原理图

4.3.2 基于 NOR Flash+NAND Flash 启动方案的 FPGA 验证

NOR Flash 在系统中的物理地址是 0x20000000，如图 4-8 所示：通过 Jlink 工具将 BootLoader 烧录到 NOR Flash 中：

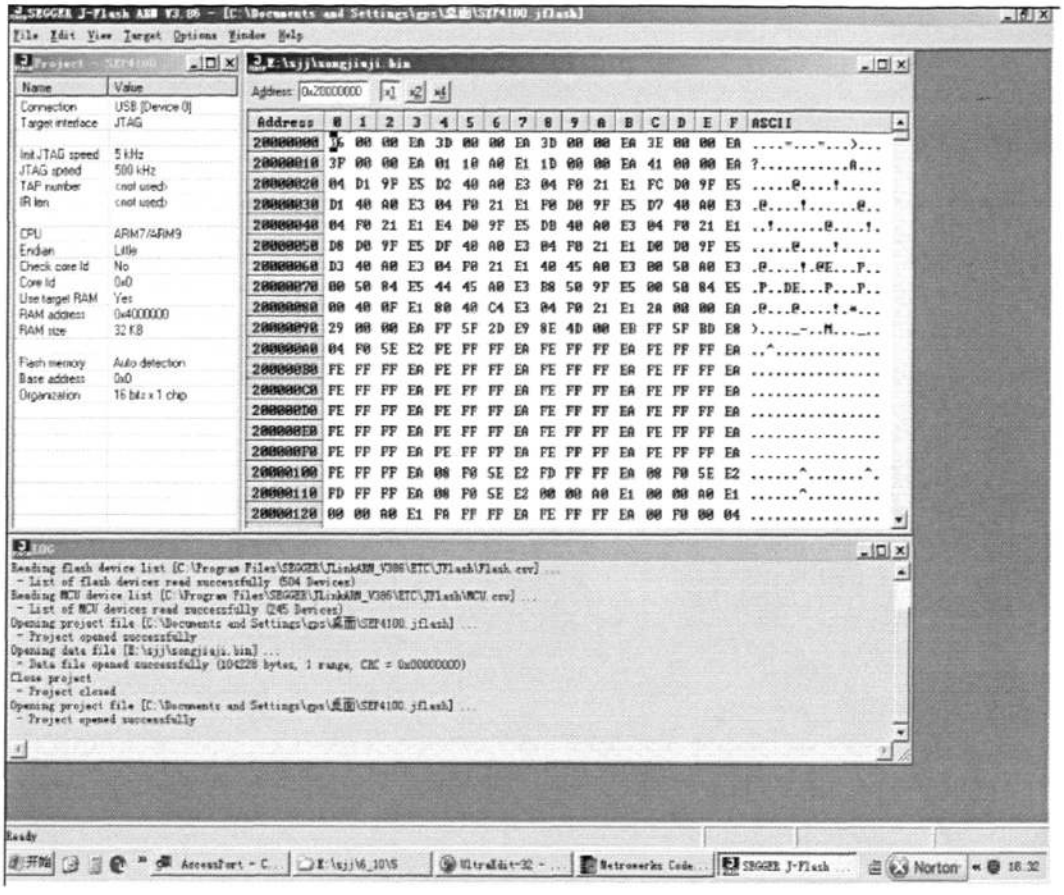

图 4-8 BootLoader 烧录示意图

设置系统启动为 NOR Flash 启动模式，系统直接将 NOR Flash 映射到 0 地址，将 BootLoader 烧录到 NOR Flash 后，重新上电或者重新启动系统，系统直接在 NOR Flash 中运行。系统执行 BootLoader，将放在 NAND Flash 中的大小为 8M 字节的系统内核搬运到 SDRAM 中，然后再将 SDRAM 映射到 0 地址，完成系统启动。

在进行 FPGA 验证时，为了确定 BootLoader 是否可以正常执行，在 BootLoader 中嵌入了一段计算计算 PI 的程序，如果 BootLoader 可以正常运行，则会将 PI 的计算结果按每四位一组写入 ESRAM 从 0x5000f000 开始的地方。同时为了测试这种启动方式所用的时间，在 BootLoader 中设置了 TIMER 用其来计时，当将 8M 字节的系统内核搬运完成后，再读取 TIMER 的计数值，来确认整个启动用时。

经过测试，采用 NOR Flash+NAND Flash 方案的启动时间为：**85.6 秒**

4.3.3 基于 NAND Flash 启动方案的 FPGA 验证

设置系统启动为 NAND Flash 启动模式，NAND Flash 数据寄存器的地址为 0x11000200，基于 NAND Flash 的两种不同启动方式验证流程如下：

1、通过 DMA 搬运

由 DMA 配合 NAND FLASH 控制器去搬运 NAND Flash 中的前 4kByte 代码到 ESRAM 中，然后由 CPU 执行启动代码从而将真正的启动代码引入到 SDRAM，最后将 PC 指针指向 SDRAM。

2、不经过 ESRAM 由 CPU 进行搬运

程序 Load 到 NAND Flash。启动过程中，由 CPU 直接从 FIFO 读取数据，在 FIFO 读空的情况下，将从设备 READY 信号拉低，等待 NAND Flash 中的数据读出。并且在此读取过程中，DMA 的请求被屏蔽。NAND Flash 中的第 1 页程序，要求是顺序执行的，内容为配置 EMI 和 DMA，执行将 NAND Flash 的第 2 页程序传输到 SDRAM。NAND Flash 的第 2 页程序将在 SDRAM 中执行，内容为将后续的所有程序传输到 SDRAM。最后在 SDRAM 中执行后续程序。

经过测试，两种基于 NAND Flash 的设计方案的启动时间相差无几，均为 **111.3 秒**。

4.3.4 基于 SD 卡启动方案的 FPGA 验证

完整的基于 SD 卡启动设计的验证流程如下：

1、设计一个启动程序，启动程序完成圆周率 PI 的计算，计算 800 位，以四位为单位，将计算结果写到 ESRAM 的 0x5000f000 处；

2、编译基于 SD 卡的启动程序，生成镜像文件 boot.axf，将 boot.axf 文件转化成二进制文件；

3、将生成的二进制文件做成一个数组，通过 SD 多块写命令将其写到 SD 卡的 0 地址处。至此，启动的前期准备工作完成。

4、通过开发板的硬件设置，将系统启动模式选择为 SD 启动。图 4-9 为启动前 ESRAM 中 0x5000f000 处的状态，值全部为 0。按复位键，系统复位。由于系统处于 SD 卡启动模式，根据系统内部硬件的设计，复位后，系统内部硬件首先会完成 SD 卡的初始化及 SD 卡多块读命令的发送。上述命令正确执行后，DAM 通道 0 将 SD 卡从 0 地址开始的 4K 数据搬运至系统的 0 地址处。4K 数据搬运完毕后，DMA 交出总线使用权，ARM 从 0 地址处开始执行程序，如果可以正常启动，并且启动后系统可以正常运行，那么启动后 ESRAM 从 0x5000f000 开始的地址处将会被写入数据，数据即是启动程序运行的结果 PI 的值，从图 4-10 可以看出，启动后 ESRAM 中 Ox5000f000 处已经写入新的数据，SD 卡启动正常，并且在启动后程序可以正常运行，基于 SD 卡的启动设计正确实现。

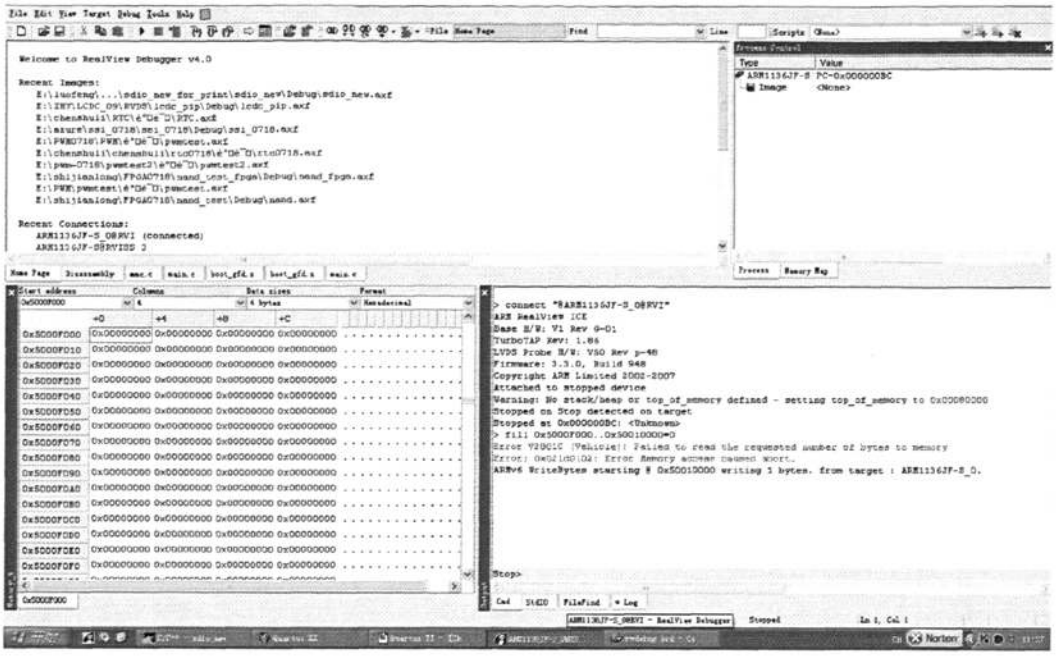

图4-9 SD卡启动前ESRAM中数据状态

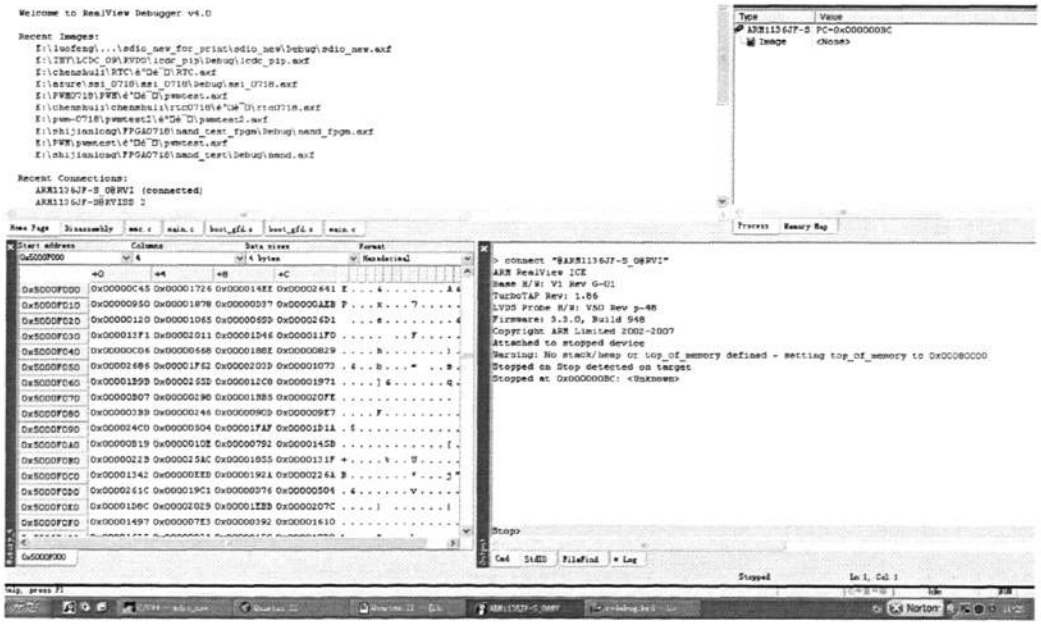

图4-10 SD卡启动后ESRAM中数据状态

经过测试，采用 SD 卡设计方案的启动时间为：174.1 秒

4.3.5 启动方案结果分析

本文设计的三种嵌入式系统低成本启动方案，都可以实现正常的启动要求。但是三种启动方式都有各自的特点：

表 4-1 不同启动方案的性能对比

方案 / 对比	NOR+SDRAM	NOR+NAND+SDRAM	NAND+SDRAM	SD Card+SDRAM
设计复杂度	NOR的XIP特性，设计简单	多片存储器，设计复杂	设计简单	需对SD卡控制器进行改造，设计复杂
所需硬件支持	EMI、PMC等	EMI、PMC、NAND控制器或者GPIO模式接口电路	需要专用的NAND控制器，且支持直接启动	EMI、PMC、INTC、在SD卡控制器基础上改造而成的SD卡启动控制器
容 量	存储少量的代码和数据	NOR存放少量代码，NAND储存大量的数据和系统内核镜像	可存储大容量数据和系统内核镜像	支持大容量数据的存储
成 本	高	较高	低	最低
启动速度	快	快	较慢	慢
适用场合	小量数据启动的系统	对成本有一定要求、而对系统硬件复杂度没有要求	低成本，具有NAND控制器的SoC环境	低成本，具有SD卡控制器的SoC，启动具有灵活性要求

（1）NOR Flash+SDRAM

由于 NOR Flash 的 XIP 特性，基于该方案的 BootLoader 简单且容易实现，几乎所有的 SoC 都支持，且启动速度快，但是 NOR Flash 的成本高，因此该方案只适用与少量代码和数据的的存储应用。

（2）NOR Flash+NAND Flash+SDRAM

该存储方案由于选择了多片存储器，设计变得较为复杂。NAND Flash 的应用可以通过 GPIO 口模拟实现，也可以通过专用的 NAND Flash 控制器。该启动方案结合了 NOR Flash 和 NAND Flash 各自的优点，相比与 NOR Flash 启动，可以支持大容量代码、数据的存储，应用范围更加广泛。启动速度较快，成本也高。

（3）NAND Flash+SDRAM

该方案的硬件条件需要 SoC 芯片内部有专用的 NAND Flash 控制器，并且控制器支持直接启动，适用于大容量的代码和数据的存储，NAND Flash 控制器的设计本身比较复杂，而且还需要专门的校验电路，因此支持该方案启动的 SoC 芯片设计比较复杂。该启动方案可以不需要 ESRAM 配合。由

于该方案中没有用到 NOR Flash 成本较低，成本相对较低，速度较慢。

（4）SD 卡+SDRAM

该方案的硬件条件需要芯片内部有 SD 卡控制器，通过对该控制器的专门改造来实现 SD 卡的直接启动。可以支持大容量的存储，相对而言，控制器的设计较为容易（校验已经在卡内完成），读写速度较慢，需要 ESRAM 配合。该方案适用于应用需要 SD 卡的 SoC 系统。该方案成本低，速度慢。

基于 NOR Flash 和 NAND Flash 的两种启动方式，都有共同的弊端：烧片的速度比较缓慢，调试的效率不高；硬件方面需要大容量的 Flash 的支持，增加了研发成本；进行内核更新时显得不够灵活。

基于 SD 卡可移动存储介质的启动方式，进行内核更新时显得更为灵活，只需把更新内核转存到指定目录，此外它的实现也比较简单。进行这方面的改进时只需做以下工作：在硬件方面，增加针对 SD 卡的硬件电路，而在中大型系统中，有关移动存储介质的硬件电路是现成的，所以硬件部分也可忽略；在 BootLoader 程序内部，只需添加对移动存储介质（如 CF 卡、SD 卡等）基于文件系统进行存储访问的指令。

4.4 本章小结

本章为本文的测试环节，属于本文的重要部分。首先从介绍验证的重要性开始，并结合实验室的软硬件资源，在 SUNSERVER 服务器上完成 SD 卡启动设计中 SD 启动控制器的功能验证，证明设计的正确性。

在功能验证的基础上，为了进一步验证各种启动方案，本文在 Altera Stratix III EP3SL150 开发板平台上完成设计的 FPGA 验证。以操作系统内核的载入和顺利启动为标识，证明了三种不同不同启动方案设计的正确性以及实用性。本文在 FPGA 测试过程中，设置 TIMER 各个启动方式所用时间，测试结果显示 NOR Flash+NAND Flash 的组合启动速度最快，SD 卡启动方案的成本最低。考虑三种不同启动方式的速度和成本，分析了各自在嵌入式系统中的应用场合。

第五章 总结与展望

本文针对嵌入式系统 BootLoader 特点，分析了 BootLoader 的结构和基本功能，并结合不同的存储介质，完成嵌入式系统低成本启动方案的设计和实现。这些工作对于降低嵌入式产品的成本具有重要现实意义。

1. 在论文的研究工作中，通过分析不同 BootLoader 的实现过程，总结了其共同的特性，归纳出 BootLoader 的通用功能，对 BootLoader 有较深入的认识。

2. 选取三种存储介质：NOR Flash、NAND Flash 和 SD/SDHC 卡作为分析对象，通过对它们结构、数据读取存储操作的分析，讨论基于这三种存储介质的 Boot 设计，并分别用汇编和硬件描述语言设计实现。

3. 结合实验室现有的软、硬件资源，在 SunServer 服务器上面完成各个设计的功能验证，并在 Altera StratixIII 平台上完成 FPGA 验证，进一步验证设计的正确性和实用性。

一个好 BootLoader，应该提供更多的功能以便开发人员的应用。它需要研究的地方还很多，主要有以下几个方面需要进一步的深入研究：

1. 在上面的开发过程中，一些问题需要在知道操作系统下才能得到比较好的解决，所以要设计一个好的 BootLoader，平时需要多分析学习操作系统的具体实现，才可以提高自身的设计能力。

2. 对更多存储介质和协议的支持。目前市面上除了文中介绍到的存储介质以外，还有 U 盘、CF 卡、网络节点等。另外随着现在各种协议的广泛使用，在已经实现的 BootLoader 的基础之上，可以进行进一步的工作，譬如增加对网络协议 TCP、TFTP 等的支持。BootLoader 如果实现对上述内容的支持，可以更加扩展嵌入式系统的应用范围。

通过本课题的研究，我对嵌入式领域的原理有了一个总体上的认识，对系统的具体实现过程也深入了一步。

致谢

本论文从课题选择、制定方案到具体实施，以及最后阶段论文大量复杂繁琐的修改工作，都是在导师陆生礼教授和刘新宁老师的悉心指导和大力帮助下完成的。他们丰富的知识、严谨的治学态度以及崇高的修养，真正体现了为人师表这四个字的含义，使我受益匪浅。特别是刘新宁老师，作为本人的责任导师，在本人整个硕士研究生学习期间，给予我以学习、生活、工作各个方面的关心与照顾，使我在 ASIC 中心的这三年个人获得极大的成长，其高尚的人品和道德，更是我未来人生路上的榜样。我会永远铭记陆老师和刘老师对我耐心细致的指导，在此致以衷心的感谢！

我也要感谢 ASIC 工程中心的杨军老师、王学香老师以及其他所有帮助过我的老师，感谢他们在我的硕士研究生期间对我的谆谆教诲和帮助，使得我在中心渡过了快乐的三年！

还要感谢我在东南大学这两年认识的所有同学和朋友。有了他们的相伴，以及在我需要的时候给我提供的帮助，使得我能顺利完成研究生阶段的学习。他们是：戴麟、罗峰、葛伟、李璐、朱彬、鲁顺、张其、秦奋、朱炜、邱华、鲍丹、时建龙、左成兵、史先强、肖建、张彬、周海燕等。

最后的也是最重要的，谢谢我的家人，我可爱的母亲和可敬的父亲，是你们默默的付出和无尽的关心爱护，让我一路顺利走来，完成研究生阶段的学习。感谢你们的养育之恩，在此，致以最衷心的感谢。

谨以此文献给我的家人和所有关心我的老师、同学、家人和朋友。

参考文献

[1] 严菊明. 基于 ARM 嵌入式系统的通用 Bootloader 设计与实现[D]:[硕士学位论文].南京：东南大学电子工程学院，2005

[2] 杜春雷. ARM 体系结构与编程[M].北京:清华大学出版社,2003. 2-37

[3] 马忠梅. ARM 嵌入式处理器结构与应用基础[M].北京: 北京航空航天大学出版社,2002. 3-19

[4] 时龙兴. 嵌入式系统——基于 SEP3203 微处理器的应用开发[M].电子工业出版社，2006. 18-57

[5] Tony. 几种 Bootloader 简介[DB/OL].http://www.cevx.com/bbs/viewthread.php?action=printable&tid=18439, 2004-4-3

[6] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, et al. Hill. C and tcc: a language and complier for dynamic code generation [J]. ACM Transactions on Programming Languages and Systems,1999,21(2):324-369

[7] SAMSUNG Corporation. K9F1208UOB.pdf[EB/OL].http://www.qic.com.cn/ic/k/137. html, 2004-3-1

[8] 白浪,张思东. WinCE 系统下 Bootloader 的开发[J].单片机及嵌入式系统,2004(2):2-4

[9] George, M.R. Windows CE for a reconfigurable system-on-a-chip processor[C]. 2004 IEEE International Conference, 2004:201-207

[10] 裴科,张刚,靳荣浩. 具有多重下载接口的 Bootloader 设计[J].计算机应用研究，2007,24(12):212-213

[11] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, et al. Hill. C and tcc: a language and complier for dynamic code generation [J]. ACM Transactions on Programming Languages and Systems,1999,21(2):324-369

[12] Microsoft Corporation. Microsoft Extensible Firmware Initiative FAT32 File System Specification[S], Version 1.03. December 6, 2000.

[13] SD Group. SD Memory Card Specifications[S], Version 1.0. March 2000.

[14] Intel Corporation. TE28F160C3BD70.pdf[EB/OL].http://4megaupload.com/download_file.php?file=2217949_&desc =Intel+TE28F160C3BD70.pdf, 2005-5

[15] 陈红展,吴非,封仲淹,等. 基于 Intel PXA272 的 Bootloader 的设计与实现[J].微计算机信息 (嵌入式与 SoC),2006,22(10-2) :158-159

[16] 马学文, 朱名日, 程小辉. 嵌入式系统中 Bootloader 的设计与实现[J].计算机工程,2005(7)：96-97

[17] 许杨. 基于 unicore 架构嵌入式系统的通用 Bootloader 的设计与实现[D]:[硕士学位论文].南京：东南大学电子工程学院，2007

[18] SAMSUNG Corporation. W982516BH .pdf[EB/OL]. http://www.winbond-usa.com/products/winbond_products/pdfs/Memory/w982516bh.pdf, 2005-7

[19] Pascal Stang, Bootloaders for the Atmel AVR series[DB/OL]. http://hubbard.engr.scu.edu/embedded/avr/Bootloader/index.html, 2005-7-3

[20] 李毅，李连云，张伟宏,等.Bootloader 面向不同结构 Flash 的实现[J].计算机工程,2008,34(4):82-83

[21] 於少峰. NAND FLASH 在嵌入式系统中的研究与应用[D]：[硕士学位论文].东南大学，2005

[22] 白伟平,包启亮. 基于 ARM 的嵌入式 Bootloader 浅析[J].微计算机信,2006(4):99-100

[23] 田泽. 嵌入式开发与应用教程[M].北京:北京航空航天大学出版社, 2004

[24] 葛伟，刘新宁.一种 SD/MMC 存储器启动方案的设计实现[C]. 东南大学校庆报告会, 2008. 93-95

[25] Hennessy John L, Patterson David A. Computer Archi-tecture-A Quantitative Approach[M]. 北京:机械工业出版社, 2002. 77-107

[26]蔡浩.一种 NAND FLASH 自启动的新方法[J].现代电子技术，2007（8）：141-143

[27]南京博芯电子技术有限公司.东芯 IV SEP3203F50 系统启动过程说明[DB/OL]. 2007.6

[28]汉泽西，吕飞. 大容量 NAND FLASH 在嵌入式系统中的应用[J].计算机应用，2006(2)：61-64

[29]马丰鍔，杨斌，卫洪春. 非易失存储器 NAND Flash 及其在嵌入式系统中的应用[J].计算机技术与发展, 2007(1)：203-207

[30]许海燕，付炎. 嵌入式系统技术与应用[M].北京：机械工业出版社. 2002. 7-19

[31]李驹光.ARM 应用系统开发详解[M].北京：清华大学出版社. 2003. 78-91

[32]Kirk Zurell，艾克武.嵌入式系统的 c 程序设计[M].北京：机械工业出版社. 2002. 66-69

[33]凌明，郑凯东，胡晨. 嵌入式操作系统内核的实现[J]. 电子器件, 1999(12)：262-269

[34]王田苗.嵌入式系统设计与实例开发[M].北京:清华大学出版社. 2002. 65-78

[35]丁岩军.基于嵌入式 ARM 的 BootLoader 研究与实现[D]：[硕士学位论文].华北电力大学，2007

[36]刘洋. 基于 S3C44B0 的 BootLoader 设计与实现[D]：[硕士学位论文].哈尔滨理工大学,2007

[37]齐欣，张家栋,霍凯. ARM 核微处理器 Bootloader 的分析与应用[J].现代电子技术,2006(11):2-4

攻读硕士期间发表论文清单

王清 第一作者，《SD 卡硬件启动和数据存储的控制逻辑的设计实现》2008．10《电脑知识与技术》录用