基于 NET+50 ARM7 的 DeltaOS 操作系统内核移植

作者：王 磊

2004 年 2 月 22 日

摘要

本文主要研究如何在同一 ARM 体系结构下对嵌入式实时操作系统 DeltaOS 的内核进行移植。

本文首先对操作系统内核移植涉及到的各个方面做了简单的介绍，然后详细描述了内核移植的整个过程，包括系统引导的移植、设备驱动程序的移植和严格的测试。系统引导的移植是内核移植的主体部分，保证操作系统内核的主体能够运行起来。设备驱动程序的移植是内核移植的补充，扩展了操作系统内核对外围设备的控制。测试部分以两个实际的测试用例为例，说明了整个严格的测试流程。

本文还特别对开发过程中遇到的一些问题专门进行了分析，并提出一些解决问题的方法。最后本文对整个移植工作做了一点简单的总结，并作出本人对操作系统内核移植未来的展望。

【关键词】 内核移植　　操作系统　　嵌入式系统　　实时

ABSTRACT

The thesis mainly discusses how to port the core of the embedded real time operating system DeltaOS under the same architecture.

First of all, the thesis simply introduces each fields that the porting of operating system core is involved in and the related work. Then it explaines the whole process of core porting detailly including the porting of system booting, the poring of device drivers and the strict tests. The porting of system booting is the main part, which ensures the normal running of the main part of the operating system core. The poring of device drivers is additional, which extends the operating system core's control over external devices. The test part takes two real test case as examples to show the whole process of strict test.

The thesis especially analyzes the problem met in the development and brings up some solutions. Finally it gets a simple summary of the entire porting work and shows my opinion of the perspective of the operating system core poring's future.

【Key words】 Core Port Operating System Embedded System Real Time

目　录

引言

本项目的主要任务是将嵌入式实时操作系统内核 DeltaCORE 移植到基于 ARM 体系结构的 NET+50 开发板，也就是同一体系结构下的嵌入式实时操作系统内核移植。

随着信息技术的不断发展和进步，嵌入式系统得到了越来越广泛的应用。特别是对嵌入式实时操作系统的需求不断增多，然而其相关技术的发展却比较缓慢。嵌入式系统的硬件根据应用需求而千差万别，为了满足市场的需求，就需要充分利用现有的比较成熟的嵌入式实时操作系统，将其移植到不同的硬件平台上。

本项目在移植过程中需要完成的主要工作包括：编写 BOOT 启动程序，存储器的初始化，堆栈的建立，外围设备的驱动程序（包括串口、网络和时钟），内存映射的分配以及严格的测试。

本文首先在第一章简单介绍了一下本系统的背景、意义和开发的环境，然后在第二章介绍了系统相关的一些理论知识，随后在第三章、第四章和第五章详细描述了整个系统的实现和测试，最后做了一下简单的总结和对未来的展望。

第一章 系统简介

1.1 项目背景及意义

世界上第一台计算机问世后的短短几十年时间里，计算机技术得到了迅猛发展，真可谓是日新月异。为了适应一些行业对产品体积、成本因素的要求，计算机的控制部分被安置在了设备内部，占着非常小的空间，给处理器提供非常有限的内存。这样的系统就是嵌入式系统。

目前，嵌入式计算机系统发展迅速，被广泛地应用于办公自动化、消费、通信、汽车、工业以及军事等领域，其中，办公自动化、消费电子和网络通信领域占的份额最大。

嵌入式实时操作系统作为嵌入式系统中极其重要的一员，其应用范围也是非常广泛的，而且通常应用于安全性、稳定性要求比较高的领域。因此，在实际的应用中，我们应该尽可能地选用那些能够经受住考验的、优秀的嵌入式实时操作系统。然而，一个优秀的、成熟的嵌入式实时操作系统的诞生通常需要十多年的时间，而且需要耗费大量的人力、物力和财力。因此，我们应该珍惜已经出现的优秀的嵌入式实时操作系统，让其发挥更大的作用。

综上所述，对嵌入式实时操作系统的内核进行移植是不仅非常有意义，而且是十分必要的。

1.2 项目开发环境简介

1.2.1 DeltaCORE 简介

DeltaOS 是北京科银京成技术有限公司具有自主知识产权的的嵌入式实时操作系统。这个实时操作系统可以嵌入到以 32 位中央处理器为核心的各种电子设备中。

DeltaOS 由内核 DeltaCORE、文件系统 DeltaFILE 和网络协议 DeltaNET 组成，如图 1-1 所示。

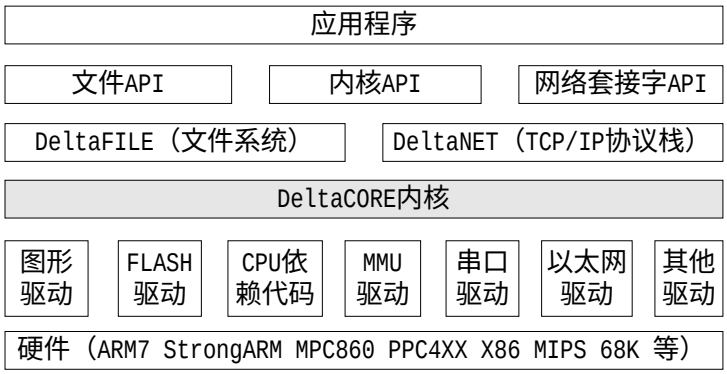


图1-1 DeltaOS的组成

从中可以看出，DeltaCORE 在 DeltaOS 中有着极其重要的作用。

DeltaCORE 支持多种目标平台，具有良好的实时性和可靠性，提供丰富的功能，性能优异，为嵌入式应用开发提供了理想的平台支持，其结构如图 1-2 所示。

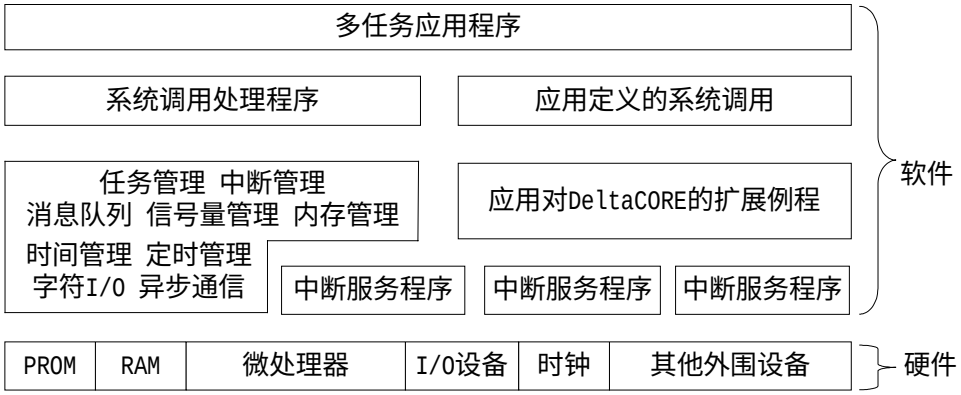


图1-2 DeltaCORE基本体系结构

DeltaCORE 的特点如下：

- 实时确定

DeltaCORE 在设计和实现上采用了多种方法保证内核具有良好的实时性。

- 调度机制：DeltaCORE 提供基于优先级的可抢占式调度和时间片轮转调度算法，兼顾相同优先级任务的平等运行权利。
- 可抢占内核：DeltaCORE 是一个可抢占内核，有利于提高及时响应能力。

- 尽量短的内核关中断时间。
- 允许中断嵌套，提高实时性能。
- 提供优先级天花板和优先级继承两种机制解决优先级反转问题。
- 浮点数的优化处理：只有真正需要时才进行协处理器上下文切换，提高运行效率。
- 系统调用确定性：为了保证系统执行时间即使在最不利情况下也可预测，DeltaCORE 在算法和数据结构上都做了细致的考虑，如采用优先级位图算法保证调度时间和就绪任务数无关、在数据结构上采用双向链表保证表项操作与位置无关、资源的有限等待等。

- 结构简捷、灵活

如图 1-3 所示，在设计上，DeltaCORE 采用三层软件体系结构，从下至上为硬件抽象层、内核层和应用层。硬件抽象层是最贴近硬件的软件层，向上对内核层提供抽象的硬件操作，向下操作具体的目标硬件，硬件抽象层可以显著减少 DeltaCORE 在硬件平台上移植的工作量。第二层是内核层，为应用程序提供任务管理、同步、通信与互斥机制、中断及内存管理等各种服务；最上层是应用层，开发人员通过系统调用接口使用内核层的服务。

由于在整个软件体系结构中各个层次所处的地位不同，其面向的开发人员也不一样。硬件抽象层主要面向操作系统内核的移植人员，操作系统的内核层主要面向操作系统的维护人员，根据应用层的需要添加或去除一些系统的服务，应用层主要面向应用程序的开发人员。其中，操作系统的维护人员和应用程序的开发人员完全不需要了解具体硬件设备的相关信息。

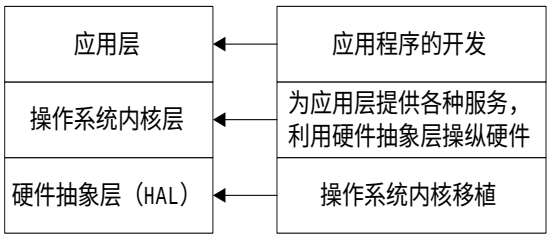


图1-3 DeltaCORE的三层软件体系结构

除了内核提供的服务以外，DeltaCORE 还提供了内核扩展机制，可以使开发人员对 DeltaCORE 的功能进行扩充。

- 可配置

用户可以对 DeltaCORE 的各种内核对象进行配置。利用科银京成提供的 Lambda 开发工具，开发人员根据应用程序的需求，对 DeltaCORE 提供的各种参数进行配置，从而使满足整个嵌入式软件系统在尺寸方面的。

- C/C++支持

可以在 C 和 C++程序中直接使用 DeltaCORE 的应用编程接口；此外，科银京成还为 DeltaCORE 提供了一个基于 C++语言的封装类库－Delta++，借助这个库，开发人员可以采用面向对象的编程方法来使用 DeltaCORE。

- 可靠

DeltaCORE 是迄今为止国内唯一经过第三方测试的实时操作系统内核。DeltaCORE 具有优异的性能和可靠性，已经在国防、航空、雷达、通讯、终端等各种电子设备中得到广泛应用。

DeltaCORE 的三层软件体系结构、90%以上的代码用 C 语言编写、函数具有可重入性等等特征保证了 DeltaCORE 具有良好的可移植性。

1.2.2 硬件平台简介

NET+50 微处理器是 NetSilicon 公司出品的一个 32 位的高性能片上系统（System-on-Chip），它内部集成了微处理器和常用外围组件，主要应用于网络设备和因特网互连。

本项目中使用的是基于 NET+50 微处理器的嵌入式开发板，其特点主要有：

- 32 位的 ARM7TDMI 核 RISC 处理器，支持 16 位的 Thumb 模式
- 8KCache 或者 16K RAM
- 5 个可编程的片选
- 4MB FlashROM，32MB SDRAM，没有 MMU
- 8、16、32 位动态数据总线宽度，28 位外部地址总线宽度
- 4 个定时器，包括 2 个可编程的定时器（2μs 到 20.7 小时），1 个可编程的看门狗定时器，1 个可编程的总线定时器
- 10 个通道的 DMA 控制器
- 2 个串口，2 个 IEEE1284 并口，1 个 10/100 网络 MAC

- 工作的核心电压为 2.25~2.75 伏 44MHz，最多 100mW，功耗低

1.2.3 开发平台简介

嵌入式实时软件的开发需要独立的开发平台，常常需要在专门的交叉开发环境中进行，如图 1-4 所示。

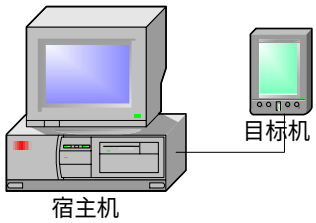


图1-4　交叉开发环境

嵌入式系统的一个特点在于其开发的特殊性与困难性。通用计算机具有完善的人机接口界面，在上面增加一些开发应用程序的环境即可进行对其自身的开发；而嵌入式系统本身不具备自举开发能力，即使设计完成以后通常用户也不能对其中的程序进行修改，只有具备一套开发工具和环境才能进行开发。目前常用的嵌入式系统的开发工具平台主要包括：实时在线仿真系统 ICE（In-Circuit Emulator)、高级语言交叉编译器（Cross Compiler Tools)、源程序模拟器（Simulator)、实时多任务操作系统、集成开发环境。本项目的开发环境是北京科银京成技术有限公司提供的 LambdaTools（包括 LambdaIDE、LambdaGCC 和 LambdaGDB）以及 BDI2000JTAG 调试器。

1.3 项目目标

整个项目的目标就是要使嵌入式实时操作系统的内核 DeltaCORE 能够在 NET+50 这个目标开发板上正确地运行起来，从而使基于 DeltaCORE 的应用程序能够正常、稳定地工作，并且能够保证其实时性。

第二章 系统相关的理论基础

2.1 ARM 体系结构

NET+50 处理器内核属于 ARM（Advanced RISC Machines）系列中的 ARM7TDMI。ARM7TDMI 核是冯·诺依曼（Von Neumann）体系结构，使用单一 32 位数据总线传送指令和数据。

2.1.1 ARM 指令系统

ARM 处理器是典型的 RISC（Reduced Instruction Set Computer）处理器，只有加载和存储指令可以访问存储器，并且数据处理指令只能对寄存器的内容进行操作。

RISC 指令系统的特点如下：

- 多个通用寄存器
- Load/Store 结构
- 寻址方式简单
- 指令定长，格式统一

ARM 指令系统继承了 RISC 结构，因此它除了具有 RISC 指令系统的特点之外，还有其自身独特的优点：

- 每条数据处理指令都能同时操作 ALU 和移位寄存器，提高了它们的利用率
- 自动增量模式提高循环程序效率
- Load/Store 多个寄存器，提高数据存取速度
- 所有的指令都可以条件执行

2.1.2 ARM 寄存器组织

ARM 处理器总共有 37 个寄存器，其中 30 个通用寄存器，1 个程序计数器（PC），6 个程序状态寄存器（PSR），长度都是 32 位。这些寄存器被安排成部分重叠的组（overlapping bank）。每种处理器模式都有不同的寄存器组。分组的寄存器在处理处理器异常和特权操作时可得到快速的上下文切换。在任一时刻，可访问的只有当前模式的一组寄存器：R0 到 R15、CPSR（当前程序状态寄存器）。不处于用户和系统模式时，还可以访问当前模式的 SPSR（保存的程序状态寄存器）。

R15 为程序计数器（PC），为当前的程序运行地址。在 ARM 状态下每条指令加一个字（4 个字节），或在 Thumb 状态下每条指令加 2 个字节。分支指令把目的地址加载到程序计数器中。也可以使用数据操作指令直接加载程序计数器。在执行时，R15 不包含当前执行指令的地址。典型情况下，当前正在执行指令的地址对于 ARM 状态是 PC-8，或对于 Thumb 状态是 PC-4。R0 到 R14 都是通用寄存器，程序中可以随意使用。R13 和 R14 通常用于特殊用途。R13 称为 SP（Stack Pointer），用作堆栈指针。R14 称为 LR（Link Register），在用户模式下，当子程序调用时它用来保存返回地址，若返回地址保存在通信堆栈中，则它也可用做通用寄存器；在异常处理模式下，它用来保存异常的返回地址。

表 2-1 不同工作模式下的寄存器分组

Mode					
User/System	Supervisor	Abort	Undefined	Interrupt	Fast Intr
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R8	R8	R8	R8	R8	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABT	R13_UND	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABT	R14_UND	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABT	SPSR_UND	SPSR_IRQ	SPSR_FIQ

= banked register

如表 2-1 所示，USR 和 SYS 模式共用一组寄存器，所有模式下的 PC 都是相同的。除了 FIQ 模式，不同模式之间的 R0 到 R12 是相同的，但每个模式都有自己独立的 SP 和 LR。对 FIQ 模式来说，它的 R8 到 R12 也是独立的。这样的好处是中断来临时，FIQ 模式下可以直接使用 R8 到 R12 而无须进行现场保存工作，加快了中断处理速度，节约了存储空间。

2.2 嵌入式实时操作系统

所谓嵌入式系统（Embedded System），就是以应用为中心，以计算机技术为基础，并且软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积、功耗等有严格要求的专用计算机系统。它一般由硬件环境、嵌入式操作系统和用户应用程序三个部分组成。

操作系统（Operating System）是介于编程者与机器硬件之间的一个软件层，可简单地被定义为：使得计算机系统的硬件成为可用的、由软件或固件（FIRMWARE）所实现的用于控制应用程序执行的程序集。

实时操作系统简称实时系统，是操作系统的一种，与普通操作系统的区别

在于它的正确性不仅依赖于系统计算的逻辑结果，还依赖于产生这个结果的时间。实时系统这一领域的基本特征是实时操作模式。实时操作模式是指：在计算机系统内部，用于处理从外部到达的数据的程序总处于就绪状态，而这些程序的运行结果只在确定的时间范围内有效；根据不同的应用，数据到达时间可以是随机的，或是预先就已确定了的。

根据实时系统的场合和开发过程，实时操作系统可以分为两种：一般实时操作系统和嵌入式实时操作系统。一般实时操作系统应用于实时查询等实时性较弱的系统，并且开发、调试、运行环境一致。而嵌入式实时操作系统应用于实时性要求高的控制系统，采用交叉开发环境，即开发环境与调试、运行环境不一致。

在嵌入式系统中，操作系统和应用软件集成于计算机硬件系统之中，即系统的应用软件与系统的专用硬件一体化。因此，嵌入式实时操作系统具有规模小（一般在几十 K 内）、实时性强（在毫秒或微秒数量级上）、可固化等特点。

2.3 LambdaTRA 简介

LambdaTRA 的全称是 Lambda Target ROM Agent，即 Lambda 目标监控器。目标监控器是调试器对目标机上运行的应用程序进行控制的代理程序，事先被固化在目标机的 Flash Memory、硬盘或启动软盘中，在目标机上电后自动启动目标机，并等待宿主机方调试器发来的命令，配合调试器完成应用程序的下载、运行和基本的调试功能。

LambdaTRA 由初始化模块、内核模块、接口模块和通信模块组成，如图 2-1 所示，其中内核模块以及通信模块部分以二进制库的形式提供给用户，其它模块均以源码形式提供给用户，内核模块与其它模块通过接口模块实现耦合。

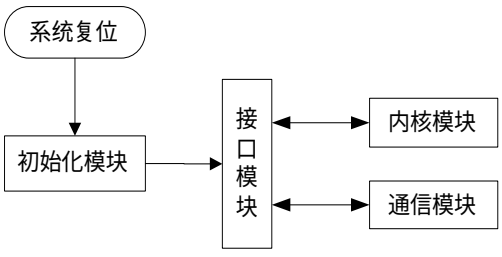


图2-1 LambdaTRA的总体结构

初始化模块负责板级的初始化，其中包括：初始化内存、配置各种复用的引脚、设置所需的 MPU 时钟以及 LambdaTRA 所用到的设备的低级初始化等工作；通信模块负责设置通信外设正确的通信模式、通信速率以及通信传输等工作；内核模块负责解析并执行 LambdaGDB 的命令来控制应用程序的执行，在有事件(例如：断点事件、内存访问越界事件等)发生时主动地向 LambdaGDB 报告；接口模块的主要功能是将 LambdaTRA 中与具体硬件有关的各种调用抽象为间接调用，使内核模块与具体的硬件无关，从而使 LambdaTRA 具有可配置性和可移植性。

Lambda 安装目录下的 tra-arm 子目录结构及内容如下：

- config 配置目录，包含可供用户配置的文件
- init 启动程序目录，包含 LambdaTRA 所支持的 ARM 系列目标板的初始化程序
- include 头文件目录，主要包含 LambdaTRA 所支持设备的驱动程序头文件
- lib 库目录，包含生成 TRA 链接所需的库文件
- targets 通信设备目录，包含 LambdaTRA 所支持设备的驱动程序源文件
- tools 工具目录，包含固化 LambdaTRA 的相关工具

第三章 系统引导的移植

移植就是使一个内核在某个处理器平台上运行。对于本项目来说，就是要使 DeltaCORE 能够在 NET+50 开发板上运行。

嵌入式实时系统作为一类特殊的计算机系统自下而上包含三个部分：硬件环境、嵌入式操作系统、嵌入式实时应用程序。由于嵌入式系统应用的硬件环境差异较大，因此，如何简洁有效地使嵌入式系统能够应用于各种不同的应用环境是嵌入式系统发展中所必须解决的关键问题。这就需要将嵌入式操作系统中与硬件相关的部分相对独立出来，形成一个单独的层次。这一层次的独立性也就决定了该嵌入式操作系统的可移植性。这一层次包含了操作系统中与硬件相关的大部分功能。通过特定的上层接口与操作系统进行交互，向操作系统提供底层的硬件信息，并根据操作系统的要求完成对硬件的直接操作。由于引入了这一层次，屏蔽了底层硬件的多样性，操作系统不再直接面对具体的硬件环境，而是面向由这个中间层次所代表的逻辑硬件环境，因此把这一层次叫做硬件抽象层 HAL（Hardware Abstraction Layer），在目前的嵌入式领域中通常也叫做板级支持包 BSP（Board Support Package）。硬件抽象层的引入大大推动了嵌入式实时操作系统的通用化，从而为嵌入式系统的广泛应用提供了可能，所以操作系统内核的移植实际上就是 BSP 的改写。

BSP 由于在系统中的特殊位置而具有以下主要特点：

- 硬件相关性：因为嵌入式实时系统的硬件环境具有应用相关性，所以作为高层软件与硬件之间的接口，必须为操作系统提供操作和控制硬件的方法。
- 操作系统相关性：不同的操作系统具有各自的软件层次结构，因此不同的操作系统具有特定的硬件接口形式。

在实现上，BSP 是一个介于操作系统和底层硬件之间的软层次，包括了系统中大部分与硬件相关的软件模块。在功能上包含两部分：系统初始化及与硬件相关的设备驱动。系统初始化完成的基本功能有：对 MPU 进行低级初始化、对开发板的硬件进行初始化、加载操作系统。

在 DeltaCORE 中，内核移植主要涉及到 LambdaTRA 初始化模块和外围设备驱动程序的修改。

内核移植的大致步骤为：

- 按照硬件资料的详细说明编写 BOOT 启动程序，一般包括对 chipselect 的初始化，MPU 时钟的配置，其它的各种复用引脚的配置等。
- 不同的目标板，内存分布可能不一样，需要更改或添加 LambdaTRA 的链接定位文件。
- 更改 makefile 文件。
- 在 config.h 文件中添加新的目标机配置。
- 根据目标机的具体情况修改调试通信接口。在 config.h 文件中添加新的目标机配置。

3.1 BOOT 启动

BOOT 启动程序是 BSP（或硬件抽象层）的一部分，其基本功能包括对 MPU 的低级初始化（如寄存器的配置）、对开发板的硬件进行初始化、加载操作系统。

我们先来看看 PC 的 BOOT 启动过程。在 LINUX 系统中，当 PC 上电之后，合理的硬件电路能够产生上电复位信号给 MPU，紧接着 MPU 通常会跳往某个地址单元取得指令并执行。在普通的 PC 中，该地址单元存放着一条跳转指令直接跳到 BIOS 中，由 BIOS 负责系统硬件初始化，然后读取主硬盘的第一个扇区（MBR）的信息，MBR 将会指向 LILO，由 LILO 负责加载 LINUX 内核并解压，并把 MPU 的控制权交给解压后的 LINUX 内核，随后逐步启动 LINUX 内核，直到整个操作系统正常地工作起来。

嵌入式开发板的 BOOT 启动过程与 PC 类似，但还是有一些不同的地方。由于开发板可能会应用在不同的环境下，所以开发板上通常都有许多的跳线和开关。当开发板上电之后，合理的硬件电路能够先根据跳线和开关的设置情况对相应的寄存器赋值，将开发板配置成相应的环境（比如 FLASH 数据位的宽度、RAM 存储器数据位的宽度数据以及存储的大小端等等），然后产生上电复位信号给 MPU，紧接着 MPU 通常会跳往复位异常中断向量地址处，通过在此

中断向量安放合适的启动代码就可以引导 MPU 的运行。本开发板中，上电复位之后，NET+50 首先从 FLASH 的 0x0 处执行第一条指令，通常这条指令是一条跳转指令，跳到对开发板硬件进行初始化的程序执行，当硬件初始化完成之后，就启动操作系统的核心程序，创建操作系统的第一个进程。

从上面可以看出，BOOT 启动程序的编写是操作系统内核移植的第一步，也是最重要的一步。如果 BOOT 启动程序不能够使开发板的硬件正常工作起来，那么内核移植就没有办法进行下去。同时，由于在开发板正常工作之前，其外设都是不可用的，没有办法与主机通信，所以对其进行调试是非常困难的，只能通过指示灯等一些原始的方法观察 BOOT 启动程序的执行情况。

硬件系统上电时，硬件特别是内存没有初始化，C 函数库没有装入内存，系统此时不支持 C 语言程序，只支持它自己的 32 位汇编指令，所以 BOOT 启动程序只能用汇编指令来初始化硬件，为后续的操作系统包括 C 语言支持做准备。在存储器被成功初始化之后，就可以调用 C 语言函数了。

3.2 建立 ROM 中断向量

中断技术是计算机技术中非常重要的一种技术，它的出现给计算机世界带来了重大的进步。

3.2.1 中断定义

所谓中断，是指 MPU 在正常运行程序时，由于内部/外部事件或由于程序的预先安排引起的 MPU 中断正在运行的程序，而转到为内部/外部事件或预先安排事件服务的程序中去。服务完毕，再返回去继续执行被暂时中断的程序。也就是说，MPU 在执行当前程序的过程中，插入另外一段程序运行。

虽然不同的微型计算机的中断系统有所不同，但实现中断时都有一个相同的中断过程。它包括中断请求、中断响应、中断服务和中断返回 4 个阶段。

3.2.2 中断服务程序入口地址的获得

发出中断请求的外部设备或引起中断的内部原因称为中断源，而 MPU 识

别中断或获取中断服务程序入口地址的方法有两种：

- 向量中断：中断服务程序的入口地址（或入口地址的指针）是在 MPU 响应中断后，发出中断应答信号 INTA 时，由中断控制器通过数据线输入到 MPU 中。
- 查询中断：是采用软件查询技术来确定发出中断请求的中断源。

在实际应用的系统中，MPU 正在处理某个中断源，又可能会出现更高的中断源请求中断服务，为了使更紧急的中断源及时得到服务，MPU 需要暂时挂起正在处理的中断，而去服务更高级别的中断，这样就要求系统支持多重中断，也就是中断嵌套。

3.2.3 中断处理

当中断产生时会触发一系列的事件，包括硬件部分和软件部分。

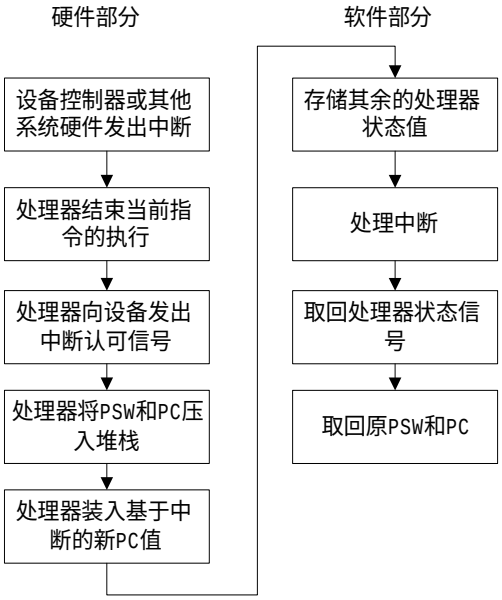


图3-1 简单中断处理

如图 3-1 所示，一个 I/O 设备完成一次 I/O 操作需要完成如下的工作：

(1). 设备向处理器发出一个中断信号。

- 处理器在响应中断前，完成当前指令的执行。
- 处理器检测中断，确定中断源，并发送一个响应信号给发出中断的设备，允许设备取消中断。

- 现在处理器需要准备传送控制给中断例程。首先须保存中断的现场。需要的最小信息是：保存在一个称为程序状态字（PSW）寄存器中的处理器状态和保存在一个程序计数器中的下一个要执行指令的位置。这些信息都被压入到系统控制栈中。
- 处理器将中断处理程序的入口地址装入程序计数器。取决于计算机结构及操作系统设计，或是一个单一程序，或每个中断类型对应一个程序，或每个设备和每个中断类型对应一个程序。如果有一个以上的中断处理程序，处理器必须确定调用哪一个程序。这个信息可能包含在原先的中断信号中，或处理器发送一个请求到请求中断的设备，以得到包含所需信息的响应。

一旦程序计数器装入，处理器进入下一个指令周期，并获取一条指令。因为指令获取是由程序计数器的内容决定的。所以控制权转到了中断处理程序。程序执行以下几步操作：

(2). 与中断程序有关的程序计数器和 PSW 已存入系统栈。但是，还有其它一些执行程序的“状态”信息要考虑，特别是，处理器寄存器的内容必须保存，因为这些寄存器会被中断处理程序使用。总之，必须保存所有这些值及其它状态信息。通常，中断处理程序一开始就在栈中保存所有寄存器的内容，然后将程序计数器指向中断服务程序的起点。

(3). 中断处理程序现在可以进行中断处理。它包括：检查与 I/O 操作有关或引起中断的其它事件的状态信息，以及向 I/O 设备发送附加的命令或确认信号。

(4). 当中断处理完成后，将所保存的寄存器值从栈中取出，并恢复到寄存器中。

(5). 从栈中恢复 PSW 和程序计数器的值。结果是，下一条要执行的指令从原先的中断点继续执行下去。

注意，保存所有关于中断程序的状态信息对以后的恢复工作很重要。这是因为，中断不是被程序调用的例程，而是在任何时刻，用户程序执行到的任何点都可能发生的事件，它的出现是随机的。

3.2.4 ARM 中断向量

ARM 支持 7 种类型的异常，如表 3-1 所示，表中列出了异常的类型以及处理该异常的处理器工作模式。当异常发生的时候，正在执行的程序将被中断，转到相应异常类型固定对应的存储器地址执行。这些固定对应的存储器地址被称为异常向量。

表 3-1　ARM 异常处理模式及向量

异常类型	处理器模式	普通地址	高位地址
复位	SVC	0x00000000	0xFFFF0000
未定义指令	UND	0x00000004	0xFFFF0004
软中断（SWI）	SVC	0x00000008	0xFFFF0008
指令预取错误	ABT	0x0000000C	0xFFFF000C
取数据错误	ABT	0x00000010	0xFFFF0010
IRQ 中断	IRQ	0x00000018	0xFFFF0018
FIQ 中断	FIQ	0x0000001C	0xFFFF001C

当异常发生时，LR 寄存器用于存储程序跳转之前所在指令的下一条指令的地址，SPSR 寄存器用于存储处理器模式转换之前的程序状态。

这一点和 PC 上 X86 的结构很类似，只是中断向量的数目少了许多。在移植的过程中，我们严格地遵循这一结构建立了自己的中断向量表和相应的中断处理函数。

```
.section ".rom_vectors", #alloc, #execinstr
    B       _reset
    B       _Undefined_Handler
    B       _SWI_Handler
    B       _Prefetch_Handler
    B       _Abort_Handler
    NOP             // Reserved vector
    B       _IRQ_Handler
    B       _FIQ_Handler
```

目标机上电之后执行的第一条指令就是 B　_reset，这同时也是系统软件复位时执行的第一条指令。这条指令跳转到_reset 标号处，也就是对开发板进行初始化的程

序段开始执行。

_IRQ_Handler 是所有中断处理函数的入口点。不管 MPU 接收到什么样的中断信号，它都会马上执行 B　_IRQ_Handler 这条指令。

```
_IRQ_Handler:
    SUB   sp, sp, #4
    STMFD    sp!, {r0}
    LDR   r0, =HandleIrq
    LDR   r0, [r0]
    STR   r0, [sp, #4]
    LDMFD    sp!, {r0, pc}
```

在 ARM 汇编语言中，“ B 标号 ”是相对偏移量的跳转指令。在对存储器地址进行重新定位的时候，这样的相对偏移量的跳转指令会带来许多不必要的麻烦，因此通常会用类似于上面这段程序的方法将相对偏移量的跳转指令转换为绝对偏移量的跳转指令。上面这段程序中 HandleIrq 就是一个绝对地址，其具体的值可以根据需要自己指定。HandleIrq 的值就是中断处理程序的统一入口点。这个入口点的主要工作就是判断中断的来源，如果是调试端口发出的中断，就由 LambdaTRA 处理，转到具体的调试端口的发送、接收程序；如果是其他外围设备发出的中断，就交给上层的操作系统处理，由操作系统再根据中断号调用被注册的中断服务程序。

```
delta_isr_entry current_irqfiq[ARM_EXCEPTION_IRQ_AND_FIQ_NUMBER];
```

被注册的中断服务程序的起始地址就存放在 current_irqfiq 这个数组里面。

3.3 屏蔽中断

ARM 体系结构支持 7 种处理器模式，如表 3-2 所示：

表 3-2　　ARM 处理器模式

处理器模式		说明
用户	USR	正常程序执行模式
快速中断	FIQ	支持高速数据传送或通道处理
中断	IRQ	用于通用中断处理
管理	SVC	操作系统保护模式

终止	ABT	实现虚拟存储器或存储器保护
未定义	UND	支持硬件协处理器的软件仿真
系统	SYS	运行特权操作系统任务（ARMv4 及以上版本）

大多数应用程序在用户模式下执行。当处理器工作在用户模式时，正在执行的程序不能访问某些被保护的系统资源，如某些特殊的寄存器，同时也不能改变自己运行的模式，除非有异常发生。这样就有利于操作系统保护系统资源。

除用户模式外的其它模式称为特权模式，它们可以自由地访问系统资源和改变自身运行的模式。

程序跳转到标号_reset 处之后，首先要做的就是将 MPU 的工作模式转为特权模式，同时在 MPU 这一级屏蔽所有的中断。

MPU 工作模式的切换主要涉及到 CPSR 和 SPSR 两个寄存器。

CPSR 和 SPSR 是程序状态寄存器，它们的结构如图 3-2 所示。

图3-2 程序状态寄存器结构

状态寄存器的低 8 位为控制位，只能由软件来进行修改。其中，M[4..0]为模式编码，码表如表 3-1 所示。T 用于指示是否工作在 Thumb 指令模式下，I 和 F 用于屏蔽中断。程序状态寄存器的高 4 位为条件标志位，分别为 N（Negative）、Z（Zero）、V（oVerflow）和 C（Carry）。

表 3-1 模式编码

M[4..0]	**模式**
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

具体的代码如下：

```
MOV     R0, #SUP_MODE
ORR     R0, R0, #I_Bit
ORR     R0, R0, #F_Bit
MSR     cpsr_all, R0
```

3.4 存储器初始化

工作模式设定完成后，MPU 有了一个基本的运行环境，但是这个环境是相当“恶劣”的，我们还应该扩展这个环境，最大限度地充分发挥 MPU 的作用。

存储器的初始化是整个 BOOT 启动程序中最重要、也是最难以调试的部分。之所以说其重要，是因为所有的程序都需要在装载在存储器中，并且一直在存储器中执行；而之所以说其难以调试，是因为其表现只有两种：工作正常和工作异常，而且导致工作异常的因素是非常多的。这就要求我们不但要对存储器硬件的型号和特性有足够的了解，而且要对其硬件文档中描述的初始化序列十分清楚，同时还要注意某些开发板为纠正错误而要求的特定的初始化序列。ROM 的初始化相对来说比较简单，相关的参数比较少，而 RAM 的初始化就要困难一些。

在对 RAM 进行初始化的过程中就遇到了一些莫名其妙的问题。例如，在存储器的初始化完成之后，发现在调试过程中在单步跟踪模式下，后序的程序能够正常地运行，能够输出正确的结果；然而在直接运行整个程序的时候，程序总是没有响应，对程序强制结束后，屏幕上弹出的警告框中提示的错误信息并不是每次都一样。

遇到这个问题之后，经过自己反复地思考，总结出可能的原因有：

- 在运行程序之前，对开发板寄存器的赋值数目还不够多，应该增加对一些相关寄存器的赋值。
- 在对 RAM 进行读写操作时延时不够，应该增加一些延时来保证读写周期的稳定。
- 由于屏幕上弹出的警告框中提示的错误信息多数是访问非法的内存空间，因此可能是对内存初始化的一些参数设置不太合理，导致内存工

作不稳定。

基于上面几种考虑，分别对程序进行修改和测试。最后，发现真正的原因是 RAM 的 Burst Access Size in Beats 这个参数的值设置得太小，导致 RAM 工作不稳定。以前这个参数的值设为 4 System Bus cycles in burst，现在设为 16 System Bus cycles in burst 后程序就能够正常地工作了。

这只是遇到的众多问题中的一个，从中我们也可以看出，存储器相关的复杂参数特别多，在对存储器进行初始化的时候必须十分小心。

存储器初始化工作完成后需要进行严格的测试。基本的方法是先向 RAM 区域写入数据，然后再读出，通过比较写入和读出的数据是否相等就可以判断对 RAM 的初始化是否成功。

3.5 RAM 区准备工作

由于程序在 ROM 中运行的速度比在 RAM 中运行的速度要慢一些，并且 ROM 区是只读的，不能满足应用程序读写的要求，因此现在就要为程序在 RAM 中运行做一些必要的准备工作。

首先，为了方便以后的存储器重新映射，在 RAM 区也要建立中断向量，其具体的值和 ROM 中的中断向量是一致的，只需要做简单的拷贝工作就可以了。

```
ADRL    r0, SystemHandlerData
LDMIA   r0, {r1-r8}
LDR     r0, =0x00000000   /* HANDLER TABLE*/
STMIA   r0, {r1-r8}
```

其次，要清零 BSS 段。BSS 段中存放的是没有被初始化的数据。

```
ldr     r4,=_bss_size
cmp     r4,#0
beq     no_bss

ldr     r3,=_bss_start
add     r4,r4,r3
```

```
    mov    r5,#0x0

    clear_bss_loop:
    str    r5,[r3],#+4
    cmp    r3,r4
    bne    clear_bss_loop
    no_bss:
```

最后，就是将 ROM 中的数据拷贝到 RAM 中。

```
    ldr r4,=_data_size
    cmp r4,#0
    beq no_data
    ldr r3,=_data_start
    ldr r6,=_rom_data_start
mov_data_loop:
    ldr r0,[r6]
    str r0,[r3]

    ldr r0,[r3]
    ldr r5,[r6]
    cmp r0,r5
    beq ram_right
    b ram_error
ram_right:
    sub r4,r4,#4
    cmp r4,#0
    beq no_data

    add r3,r3,#4
    add r6,r6,#4
    b mov_data_loop
```

```
no_data:
```

3.6 建立堆栈

由于 ARM 体系结构具有 7 种处理器模式，所以必须为每一种模式建立各自独立的堆栈空间，这就涉及到处理器模式的切换问题。

模式切换完之后把相应的堆栈地址赋给堆栈指针寄存器 SP 就完成相应模式堆栈的建立。

下面这段示例代码用来建立 IRQ 模式和 FIQ 模式下的堆栈，其他模式堆栈的建立与此类似。

```
MRS     r0, cpsr
BIC     r0, r0, #LOCK_MASK | MODE_MASK
ORR     r2, r0, #USR_MODE

ORR     r1, r0, #LOCKOUT | FIQ_MODE
MSR     cpsr, r1
MSR     spsr, r2
LDR     sp, =_tra_fiq_stack_top

ORR     r1, r0, #LOCKOUT | IRQ_MODE
MSR     cpsr, r1
MSR     spsr, r2
LDR     sp, =_tra_irq_stack_top
```

其中，LOCK_MASK、MODE_MASK、USR_MODE、LOCKOUT、FIQ_MODE、IRQ_MODE 都是宏定义。_tra_fiq_stack_top 和_tra_irq_stack_top 是连接文件中的变量。

3.7 存储器映射

出于安全性和可维护性的考虑，整个存储器空间被划分为若干个段。段在整个空间的定位可以是动态的，也可以是静态的。在本项目中，由于程序要被

固化到 FLASH 中，所以采用静态定位的方式。

```
ENTRY(_reset) ;
MEMORY
{
RamVectorSpace :  org = 0x0,          len = 0x100
RamSpace :        org = 0x100,        len = 0x10000
RomVectorSpace :  org = 0x2000000,    len = 0x100
RomCodeSpace :    org = 0x2000100,    len = 0x10000
}
```

这里，RAM 空间是 0x0~0x10100，被划分为 RamVectorSpace 和 RamSpace 两个区域；RAM 空间是 0x2000000~0x2010100，被划分为 RomVectorSpace 和 RomSpace 两个区域。其中，RamSpace 和 RomSpace 再细分为若干个段。

```
.text :
{
. = ALIGN (16);
_text_start = . ;
*(.text) ;
*(.rodata) ;
*(.glue_7t);
*(.glue_7) ;
. = ALIGN (16);
_text_end = . ;
} >RomCodeSpace
```

上面这一段程序是.text 段的组成和定位。从中我们可以看出，.text 段由各个目标文件的.text 段、.rodata 段、.glue_7t 段和.glue_7 段组成，定位在 RomCodeSpace 区域。

3.8 改写 MAKEFILE

当一个项目包含大量文件时，对这些文件进行编译、连接是一件非常复杂

的事情。利用 make 工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 makefile 工具就可以简洁明快地理顺各个源文件之间纷繁复杂的相互关系。而且如此多的源文件，如果每次都要键入 gcc 命令进行编译的话，简直就是一场灾难。而 make 工具则可自动完成编译工作，并且可以只对在上次编译后修改过的部分进行编译。因此，有效的利用 make 和 makefile 工具可以大大提高项目开发的效率。

Make 工具最主要也是最基本的功能就是通过 makefile 文件来描述源程序之间的相互关系并自动维护编译工作。而 makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，还要定义源文件之间的依赖关系。

在本项目中，需要向原来的 makefile 文件中添加本目标机的编译选项和各个文件之间的依赖关系。

```
netarm: tra-netarm.coff
    ${OBJCOPY} ${OBJCOPYFLAG} tra-netarm.coff tra.bin
    cp tra.bin ./tools/netarm/tra-netarm.bin
tra-netarm.coff: TMP_NETARM_BUILD
    ${LD} ${LDFLAGS-BIG} -L./lib -L/host/dcore/lib/lib_big -T
        ${NETARM_LINKCMDS} ${NETARM_START_OBJ} ${TRA_COMM_LIB_BIG}
        libtmp.a ${TRA_COMM_LIB_BIG}    -lgcc -o tra-netarm.coff
    rm -rf libtmp.a
TMP_NETARM_BUILD:
    rm -rf ${TARGET_ALWAYS_CLEAN_OBJS}
    cd ./;${MAKE} all --file=Makefile-netarm
    ${AR} ${ARFLAGS} libtmp.a ${NETARM_OBJS}
```

在调试过程中，编译的时候要加上-g 这个编译选项，这样就能在生成的 coff 文件中添加一些调试信息，从而方便调试。调试成功之后要去掉-g 这个编译选项，使生成的文件小巧一些。

在整个编译、连接过程中，遇到了一个小问题。编译能通过，但是连接过不了。经过检查，发现程序中用到了取模运算，而连接时并没有包括取模运算

的库文件，解决的方法就是在上面的语句中加入那两个加粗的字段。

上面的这些工作完成之后，可以生成 LambdaTRA 的一个 bin 文件和一个 coff 文件，coff 文件用于在 RAM 中调试，bin 文件用于烧制 FLASH。

3.9 使用 LambdaGCC 编译

LambdaGCC 是一个支持多种语言的编译工具，它根据输入文件的后缀自动选择相应的编译程序进行处理，把它编译成汇编文件；然后用 dcore-as 交叉汇编器将这个汇编文件编译为目标文件；最后调用 dcore-ld 交叉链接器来生成可执行文件或将多个目标文件合并为一个目标文件。图 3-3 简要展示了 LambdaGCC 的编译流程。

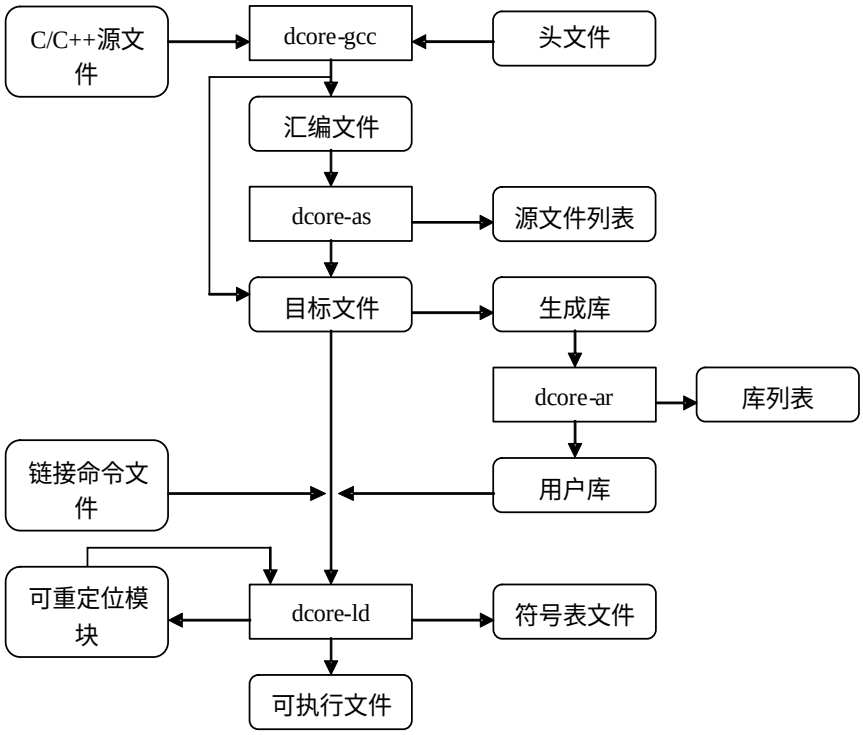


图3-3 程序的编译过程

3.10 使用 LambdaGDB 调试

LambdaGDB 是北京科银京成技术有限公司的一种专门用于嵌入式应用系统开发的交叉调试工具。此工具的功能非常强大，提供了许多非常复杂的调试

功能。

在实际的项目开发中，主要使用的调试功能有：

- LIST——在调试器中查看代码
- BREAK——设置断点
- STEP——单步跟踪
- NEXT——下一步跟踪
- PRINT——监视程序中变量的值
- SET——在程序运行过程中手动改变变量的值

使用这些命令能够很方便地调试程序中的 BUG。

第四章 设备驱动程序的移植

4.1 设备驱动程序

设备驱动程序是设备提供给操作系统或者应用软件的一套接口，主要负责对硬件寄存器的读写操作和设备的逻辑控制。它的出现，把操作系统和应用软件与设备隔离开来，屏蔽了硬件的细节，方便了用户对设备的读写和控制。同时它也使得一种硬件设备只要配备不同的驱动程序，就可以在不同的系统上使用；一种操作系统只要配备不同的驱动程序，就可以使用不同设备。

设备驱动程序是内核的一部分，它完成以下的功能：

- 对设备初始化和释放
- 把数据从内核传送到硬件和从硬件读取数据
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据
- 检测和处理设备出现的错误

驱动程序对大多数人看来觉得非常神秘，其实从程序员的观点来看，它本质上就是一些包含了 I/O 操作的子函数。

与其它的应用程序相比，驱动程序具有显著的特点，即它不能自动执行，只能被动调用。而且，在调用方法上也有独特的地方。调用驱动程序有多种方法，其中主要的有三种：

- 任务直接调用驱动程序，例如 linux 操作系统中，对文件读写，直接调用低级接口 read、write
- 任务通过操作系统调用驱动程序，例如，在 linux 操作系统中对文件读写，调用高级接口 fread、fwrite
- 任务调用功能模块的服务，再通过功能模块调用驱动程序，例如，在 linux 操作系统中对 socket 的调用

由于调用驱动程序有三种方法，因此提供给用户的对驱动程序的调用的接口也就抽象为三个层次，如图 4-1 所示：

- 驱动程序常规操作接口，该接口是提供给用户直接调用驱动程序的接口
- 操作系统提供的接口
- 功能模块服务提供的接口

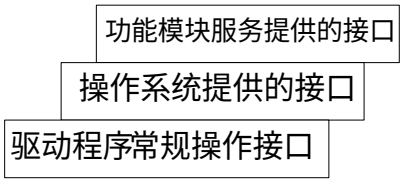


图4-1 驱动程序的三层抽象

这三种接口形成了调用驱动程序的不同层次，这样就为不同层次的用户提供了方便。

在本项目中主要涉及到的是驱动程序常规操作接口的接口。

设备通过驱动程序对外界呈现为宏观和微观两种不同的表现：在微观上，对程序员来看，表现为一组 API 的调用，其实质就是由 MPU 读写设备的控制寄存器来控制设备和配置设备，读写设备的状态寄存器来了解设备的状态和运行情况，读写数据寄存器来和设备交换数据，MPU 通过控制这些寄存器，来完成对设备的控制。在宏观上，设备正确接收了 MPU 的各种指示后，通过设备内部的数字电路、I/O 处理器和控制电路，把这些逻辑上的信息转换为相应的电信号，或者直接输出，或者再通过 D/A 转换装置，变为光、电等模拟信号，最终完成设备的功能。反过来，当设备接收到外界的电、光等信号时，它会把这些信号通过与上述过程的反过程进行转换，把收集到的数据交给 MPU 处理。具体的过程如图 4-2 所示。

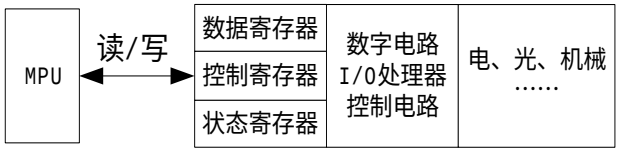


图4-2 MPU与设备的关系

4.2 串口驱动移植

串口是各种外围设备中比较简单的一种，同时也是最常用的。本项目中采用的是串口的异步通信方式。在这种工作方式下，一个字符一个字符地传输，每个字符一位一位地传输，传输一个字符时，以起始位开始，然后传输字符本身的各位，接着传输校验位，最后以停止位结束该字符的传输。一次传输的起始位、字符各位、校验位、停止位构成一组完整的信息，称为帧（Frame）。帧与帧之间可有任意个空闲位。

根据目标板串口相关的寄存器分布，实现 LambdaTRA 要求的标准串口接口函数，包括：

uart_init () ——串口初始化

com_putchar() ——串口发送一个字符

com_getchar() ——串口接收一个字符

turn_on_com_receive_interrupt() ——打开串口接收中断

turn_off_com_receive_interrupt() ——关闭串口接收中断

下面是串口初始化的程序片段，其主要的工作是设定串口的波特率、数据位的长度、停止位的位数、奇偶校验和中断允许。

```
void uart_init(int baudrate, int serial)
{
    int BAUDRATE_DIV;
    long ValueInCTRLReg=0;
    /* to get right baudrate value */
    switch(baudrate){
       case 19200:     BAUDRATE_DIV=0x80000005; break;
       case 38400:   BAUDRATE_DIV=0x80000002; break;
       case 57600:   BAUDRATE_DIV=0x80000001; break;
       case 115200: BAUDRATE_DIV=0x80000000; break;
       default:          BAUDRATE_DIV=0x81000001;
    }
#ifdef NET40_COM1
```

```
        /*set base address*/
        net40_uart1.io_base = BASE_ADDR1;
        ValueInCTRLReg = UART_REG_READ( &net40_uart1, UART_CTRL_REG );
        ValueInCTRLReg &= 0xFFFF0000;
        /*disable all interrupt*/
        UART_REG_WRITE( &net40_uart1, UART_CTRL_REG, ValueInCTRLReg );
        /*PORTA*/
        REG_WRITE( PORTA_REG, 0xeff000ef );
        /*Set baud rate:use X16 mode, the value of N register is 1 then, set bit-rate
generator to enable */
      UART_REG_WRITE( &net40_uart1, UART_BITRATE_REG, BAUDRATE_DIV );
        /* word length=8bit, stop bit 1,transmit interrupt request*/
        UART_REG_WRITE( &net40_uart1, UART_CTRL_REG, 0x83038a08 );
#endif
        ……
}
```

在串口驱动的实际开发过程，也遇到了一些问题，具体的现象是：串口不能正常地输入和输出，数据寄存器接收不到任何数据。经过不断地分析、实践，最后发现错误产生的原因在于，自己原来以为串口的工作只与其控制、状态和数据寄存器有关，实际上它还与端口 PORTA 有关系。

4.3 网络驱动移植

网络驱动程序涉及到 OSI 的数据链路层和网络层，是网络芯片和高层协议之间的桥梁和接口。网络驱动程序把网卡如何对来自和发往高层的包所使用的方法进行了屏蔽，使高层不必了解收发操作的复杂性，而网络驱动程序本身则必须对网卡的操作有详细的了解，如网卡上的各种控制寄存器和状态寄存器，DMA 和 I/O 端口等。符合 LAN 标准的网络芯片，尽管厂商不同，但因为是按照同一标准所生产的，所以必定能够通过 LAN 进行通信。例如，中断请求 IRQ，DMA 和 I/O 端口尽管有不同的分配，但不会影响通信。由于对标准的具体实

现不同，网络驱动程序也就不同。正因为这样，对所使用的特定的网络芯片必须选择相对应的驱动程序。本项目使用的网络芯片是 Cirrus Logic 公司的 10Base-T 10M 以太网芯片 CS8900a。

首先，将 MAC 地址和 IP 地址写到相应的寄存器中。这样，如图 4-3 所示，当网络芯片向网络发送数据包时，便会自动将 MAC 地址和 IP 地址附加在数据包的头部，组成一个完整的网络包。

图4-3 网络数据包的组成

接下来，最重要的工作就是实现数据的收发，也就是实现下面这两个收发数据的函数。

void lan_putpkt(char *buf,unsigned short len);

unsigned short lan_getpkt(char*buf);

网络芯片 CS8900a 内部有两个先进先出（First In First Out, FIFO）数据缓冲区，分别是输入数据缓冲区和输出数据缓冲区。对这两个数据缓冲区的维护工作直接影响到网络芯片收发数据包的速度。NET+50 收发数据包的工作方式有三种可供选择，分别是轮询、中断和 DMA。不同工作方式的复杂程度是不同的，其收发数据包的速度也有差别。DMA 方式收发数据包的速度最快，但其驱动程序实现起来也最为复杂。轮询方式的驱动程序实现起来也最简单，但其收发数据包的速度也最慢。由于在 LambdaTRA 中，数据的流量是很小的，因此从总体上考虑，采用轮询方式是最为合适的。

在发送数据的时候需要特别注意数据的对齐问题。CS8900a 网络芯片是半字（16 位）对齐的。如果在数据发送时有单个的字节，应该在这个单字节的后面填充一个字节的 0，保证数据能够正确地传送出去，否则就会造成数据的丢失。

4.4 时钟驱动移植

如果说操作系统是一台计算机的大脑的话，那么时钟就是这个大脑运行的节奏控制器。没有了时钟，操作系统就不能正常地运行，更不能合理地调度任务。而实时操作系统对时钟的要求更高，不但要求时钟能够工作，而且要求时钟能够精确地工作。

操作系统为了能够准时调度任务，需要有一种能保证调度准时进行的机制。这种机制是通过定时器来实现的。从硬件上来讲，支持各种操作系统的微处理器必须包含一个可周期性中断、可编程的间隔定时器。这个周期性中断被称为系统时钟滴答，它就像节拍器一样来组织系统任务。从软件上来讲，必须有一个软件上的定时器在硬件中断到来时处理任务调度。

NET+50 这个硬件平台提供了两种计时的方式，一种是可编程的硬件定时器（TIMER），另一种是硬件实时时钟（RTC）。

硬件定时器单纯地提供一个规则的脉冲序列，每个脉冲间隔计数完成，产生一次定时器硬件中断。硬件定时器中断的频率通过对时钟相关的寄存器编程来设置。一个系统时基与一个微秒的关系由系统配置工具设置，同时系统时基的大小为实时时钟硬件中断周期的倍数。例如，实时时钟硬件中断的周期为 1 毫秒，系统时基的大小就只能是 1 毫秒的倍数。

为了计准时间间隔，一个很重要的问题是 MPU 与实时时钟的同步。常用的方法是用程序启动、停止时钟工作，设置时基的大小，并在启动后利用硬件定时器中断信号的方法来对准系统的时钟。每当定时器的时基到时，它就引起中断，中断响应完成后定时器又开始工作，时基到时又引起中断，这样达到与 MPU 的同步。显然，软件方法具有简单、灵活、易实现和低成本的优点，可以很方便地修改实时时钟的设置和系统时间的表示，且可以在不增加硬件的基础上非常灵活地用软件模拟多个“ 定时时钟 ”，因此，在实时系统中广泛采用此种方法。本项目也采用了这种方法。

但由于中断的延迟，对系统时钟可能会造成一定的误差，因此在设计中通常将实时时钟中断的优先级设置得很高，一般仅次于掉电中断。系统的时间精度要求越高，时钟中断的频度就越高，这样执行时钟 ISR 的时间就会增多，系统的开销就会增大，影响系统的其他工作。因此，应使时钟 ISR 程序尽可能的

短，同时还要考虑时间精度。

由于嵌入式实时系统硬件设备的多样化，内核提供的实时时钟的服务就要适应这种灵活性的要求，它通常不是以 ISRs 的身份出现，而只是作为提供给 ISRs 的系统调用 delta_clock_tick 出现。

下面这段代码是定时器中断的服务程序。

```
void timer0_isr(void)
{
    //关中断
    *(unsigned int *)0xfffff014 = 0x20000;
    //使 MPU 与实时时钟同步
    delta_clock_tick();
   //开中断
   *(unsigned int *)0xfffff010 = 0x20000;
}
```

定时器驱动的实际开发过程同样不是一帆风顺的。错误的现象是，定时器的中断服务程序总是不停地被执行，而非中断服务程序不能正常地继续执行。

经过分析发现，产生这种现象的原因是，定时器每 0.001 秒产生一次中断，而中断服务程序本身的执行时间超过了 0.001 秒，刚刚从中断服务程序退出，马上又进入中断服务程序执行。将定时器产生中断的时间间隔改为 0.01 秒之后，程序的执行就正常了。

使用硬件定时器可以使时间的精度达到毫秒，这样的精度是很高的，但是由于在断电的情况下是不工作的，所以通常用于相对时间的记录。硬件实时时钟的精度通常为秒，并且自己维护绝对时间，通常用于绝对时间的记录，在目标机断电的时候采用电池供电，保证日历绝对时间的正确。

第五章 系统测试

信息技术的飞速发展，使软件产品应用到社会的各个领域，软件产品的质量自然成为人们共同关注的焦点。不论软件的生产者还是软件的使用者，均生存在竞争的环境中，软件开发商为了占有市场，必须把产品质量作为企业的重要目标之一，以免在激烈的竞争中被淘汰出局。用户为了保证自己业务的顺利完成，当然希望选用优质的软件。质量不佳的软件产品不仅会使开发商的维护费用和用户的使用成本大幅增加，还可能产生其他的责任风险，造成公司信誉下降，继而冲击股票市场。在一些关键应用（如民航订票系统、银行结算系统、证券交易系统、自动飞行控制软件、军事防御和核电站安全控制系统等）中使用质量有问题的软件，还可能造成灾难性的后果。

软件危机曾经是软件界甚至整个计算机界最热门的话题。现在人们已经逐步认识到所谓的软件危机实际上仅是一种状况，那就是软件中有错误，正是这些错误导致了软件开发在成本、进度和质量上的失控。有错是软件的属性，而且是无法改变的，因为软件是由人来完成的，所有由人做的工作都不会是完美无缺的。我们进行软件测试的目的也就是发现软件中的错误和缺陷，并尽可能地使之减少。

4.1 单元测试

单元测试的对象是软件设计的最小单位——模块。单元测试的依据是详细设计描述，单元测试应对模块内所有重要的控制路径设计测试用例，以便发现模块内部的错误。单元测试主要采用白盒测试技术，辅之以黑盒测试，系统内多个模块可以并行地进行测试。

单元测试任务包括：模块接口测试、模块局部数据结构测试、模块边界条件测试、模块中所有独立执行通路测试和模块的各条错误处理通路测试。

一般认为单元测试应紧接在编码之后，当源程序编制完成并通过复审和编译检查，便可开始单元测试。测试用例的设计应与复审工作相结合，根据设计

信息选取测试数据，将增大发现上述各类错误的可能性。在确定测试用例的同时，应给出期望结果。

基于 NET+50 平台的 LambdaTRA 在编译、连接工作完成后，就可以进行单元调试。

4.1.1 白盒测试

白盒测试也称结构测试或逻辑驱动测试，它是知道产品内部工作过程，可通过测试来检测产品内部动作是否按照规格说明书的规定正常进行，按照程序内部的结构测试程序，检验程序中的每条通路是否都有能按预定要求正确工作，而不顾它的功能，白盒测试的主要方法有逻辑驱动、基路测试等，主要用于软件验证。

“白盒”法全面了解程序内部逻辑结构、对所有逻辑路径进行测试。“白盒”法是穷举路径测试。在使用这一方案时，测试者必须检查程序的内部结构，从检查程序的逻辑着手，得出测试数据。贯穿程序的独立路径数是天文数字。但即使每条路径都测试了仍然可能有错误。第一，穷举路径测试决不能查出程序违反了设计规范，即程序本身是个错误的程序。第二，穷举路径测试不可能查出程序中因遗漏路径而出错。第三，穷举路径测试可能发现不了一些与数据相关的错误。

首先对 LambdaTRA 代码进行分析。由于模块的接口简单，而且采用 ARM 汇编语言，所以模块接口测试和模块局部数据结构测试都很简单。由于此模块是整个程序的初始模块，所以不需要模块边界条件测试。由此可以看出，测试的重点在于模块中所有独立执行通路测试和模块的各条错误处理通路测试，同时由于程序中条件分支语句很少，所以采用了覆盖率最高的路径测试，分析了每一条可能路径的执行情况。经过严格的分析，LambdaTRA 通过了白盒测试。

4.1.2 黑盒测试

白盒测试完成后，就应该进行黑盒测试。

黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用，在测试时，把程序看作一个

不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数锯而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。“ 黑盒 ”法是穷举输入测试，只有把所有可能的输入都作为测试情况使用，才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

测试的主要过程如下：

- 修改 JTAG 调试器 BDI2000 需要调用的初始化文件，包括目标板工作频率的设定、内存的初始化、大小端的设定、BDI2000 IP 地址的设定等等。
- 将 BDI2000 的 JTAG 调试口与目标板的 JTAG 调试口连接，BDI2000 的网络口与主机的网络口连接。目标板的串口 1 与主机的串口 1 进行连接。
- 然后主机就可以利用 TFTP 工具与 BDI2000 进行交互，通过 JTAG 端口将自己定制的 LambdaTRA 下载到目标板的 RAM 中，然后利用 LambdaGDB 工具监视目标板的运行状态，进行远程跟踪调试。

当自己定制的 LambdaTRA 执行完之后，如果对存储器以及各种相关外围设备的读写都无误，就可以认为该 LambdaTRA 通过了黑盒测试。

4.2 集成测试

集成测试也叫做组装测试或联合测试，是组装软件的系统测试技术，按设计要求把通过单元测试的各个模块组装在一起之后，进行综合测试以便发现与接口有关的各种错误。

由于 DeltaCORE 的其他模块已经通过了测试，因此，在本项目中直接对已经通过了单元测试的 LambdaTRA 和 DeltaCORE 的其他模块进行集成测试。

4.2.1 测试环境的搭建

首先，根据 NET+50 的内存映射，修改应用程序内存分布的模板，为测试

用例编译后的代码段、数据段以及 BSS 段等指定在内存中的位置。

其次，针对 DeltaCORE 的各个功能组成部分，构造出测试用例集。在本项目中，针对进程调度、进程的同步与互斥、软件定时器等各个功能组成部分，一共构造了 16 个测试用例。

最后，就要根据测试用例的不同作用修改其运行环境相关的参数，编译、连接生成测试用例集。

4.2.2 测试过程

宿主机通过串口与目标机通信，在通过几次握手信号之后，双方建立连接，此时就可以通过 LambdaGDB 将宿主机上的测试用例下载到目标机上调试运行。

4.2.3 测试———时间片测试

测试目标：测试操作系统中时间片轮转算法的正确性以及串口是否工作正常。

测试程序：

```
delta_task Init(void)
{
    delta_status_code ret;
    usr_prep();
    printf("\n*** Enter Roundrobin test***\n");
    /* 创建并启动任务 1 */
    task_name[ 1 ] = delta_build_name( 'T', 'A', '1', ' ' );
    ret= delta_task_create( task_name[ 1 ],3,DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES|DELTA_TIMESLICE,
            DELTA_DEFAULT_ATTRIBUTES,&task_id[ 1 ]);
    ret= delta_task_start( task_id[ 1 ], task_1,0);
    /* 创建并启动任务 2 */
    task_name[ 2 ] = delta_build_name( 'T', 'A', '2', ' ' );
    ret=delta_task_create(task_name[2],3,DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES|DELTA_TIMESLICE,
            DELTA_DEFAULT_ATTRIBUTES,&task_id[ 2 ]);
    ret= delta_task_start( task_id[ 2 ], task_2, 0 );
    /* 创建并启动任务 3 */
```

```
    task_name[ 3 ] = delta_build_name( 'T', 'A', '3', ' ' );
    ret= delta_task_create(task_name[3],3,DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES|DELTA_TIMESLICE,
            DELTA_DEFAULT_ATTRIBUTES,&task_id[ 3 ]);
    ret = delta_task_start( task_id[ 3 ], task_3, 0 );
    /*删除任务自己*/
    delta_task_delete( DELTA_SELF );
}
delta_task task_1(void)
{
    while (1) {
        printf("\n\n*** task1 is running! ***");
    }
}
delta_task task_2(void)
{
    while (1) {
        printf("\n\n*** task2 is running! ***");
    }
}
delta_task task_3(void)
{
    while (1) {
        printf("\n\n*** task3 is running! ***");
    }

}
```

在这个测试用例中，时间片是 10 毫秒，任务 1、任务 2 和任务 3 平等地参与时间片的竞争。

预计结果：在 Init()函数中先后创建了三个任务。根据时间片轮转的算法，任务 1、任务 2 和任务 3 平等地享有时间片。时间片为 10 毫秒，这就意味着每 10 毫秒就会有一次任务调度。按照任务创建的先后次序，任务 1、任务 2 和任务 3 轮流获得时间片。

根据程序的流程，通过串口应该能够接收到下面的信息：

*** Enter Roundrobin test***

*** task1 is running! ***

*** task2 is running! ***

*** task3 is running! ***

*** task1 is running! ***

*** task2 is running! ***

*** task3 is running! ***

……

测试过程：用 LambdaIDE 将时间片轮转的测试用例编译、连接为 ELF 格式的文件，然后通过调试端口下载到 NET+50 目标机，用 LambdaGDB 在关键地方加上断点查看进程的各个状态。最后让程序直接运行，查看串口输出的结果，并与预计的结果进行比对。

测试结果：通过串口接收到了预计的字符。

测试结论：测试结果与预计结果相符合，可以证明时间片轮转算法在 NET+50 目标机上的执行是正确的，串口也工作正常。

4.2.4 测试二——定时器测试

测试目标：测试定时器是否工作正常。

测试程序：

```
delta_task Init(void)
{
    delta_status_code ret;
    delta_time_of_day time;
    delta_time_of_day time_buffer ;
    usr_prep();
    printf("\n*** Enter Task Wake test***\n");
    time_buffer.year   = 2001;
    time_buffer.month  = 1;
    time_buffer.day    = 1;
    time_buffer.hour   = 12;
    time_buffer.minute = 0;
    time_buffer.second = 0;
    time_buffer.ticks  = 0;
    /* 设置时钟*/
    ret = delta_clock_set( &time_buffer);
    ret = delta_clock_get( DELTA_CLOCK_GET_TOD, &time );
    printf("\n*****Task wake test start on %d/%d/%d, %d:%d:%d:%d.*****\n\r",
           time.year,time.month,time.day,time.hour,time.minute,time.second,time.ticks);
    /* 创建并启动任务 1 */
```

```
    task_name[ 1 ] = delta_build_name( 'T', 'A', '1', ' ' );
    ret= delta_task_create( task_name[ 1 ], 5, DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES,        DELTA_DEFAULT_ATTRIBUTES,
            &task_id[ 1 ] );
    ret= delta_task_start( task_id[ 1 ], task_1, 0 );
    if( ret != DELTA_SUCCESSFUL )
        printf("task1 error.\n");
    /* 创建并启动任务 2 */
    task_name[ 2 ] = delta_build_name( 'T', 'A', '2', ' ' );
    ret= delta_task_create( task_name[ 2 ], 10, DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES,        DELTA_DEFAULT_ATTRIBUTES,
            &task_id[ 2 ] );
    ret = delta_task_start( task_id[ 2 ], task_2, 0 );
    if( ret != DELTA_SUCCESSFUL )
        printf("task2 error.\n");
    /* 创建并启动任务 3*/
    task_name[ 3 ] = delta_build_name( 'T', 'A', '3', ' ' );
    ret= delta_task_create( task_name[ 3 ],15,DELTA_MINIMUM_STACK_SIZE * 4,
            DELTA_DEFAULT_MODES,DELTA_DEFAULT_ATTRIBUTES,
            &task_id[ 3 ]);
    ret = delta_task_start( task_id[ 3 ], task_3, 0 );
    if( ret != DELTA_SUCCESSFUL )
        printf("task3 error.\n");
    /*删除任务自己*/
    delta_task_delete( DELTA_SELF );
}
delta_task task_1(void)
{
    delta_status_code ret;
    delta_time_of_day wake_buffer;
    delta_time_of_day time;
    wake_buffer.year   = 2001;
    wake_buffer.month  = 1;
    wake_buffer.day    = 1;
    wake_buffer.hour   = 12;
    wake_buffer.minute = 0;
    wake_buffer.second = 5;
    wake_buffer.ticks  = 0;
    /*在指定时间唤醒任务*/
    ret = delta_task_wake_when( &wake_buffer );
    ret = delta_clock_get( DELTA_CLOCK_GET_TOD, &time );
    printf("\n*****task_1 wake on %d/%d/%d, %d:%d:%d:%d.*****\n\r",time.year,
```

```
            time.month,time.day,time.hour,time.minute,time.second,time.ticks);
        delta_task_delete( DELTA_SELF );
}
delta_task task_2(void)
{
        delta_status_code ret;
        delta_time_of_day wake_buffer;
        delta_time_of_day time;
        wake_buffer.year      = 2001;
        wake_buffer.month     = 1;
        wake_buffer.day       = 1;
        wake_buffer.hour      = 12;
        wake_buffer.minute = 0;
        wake_buffer.second = 10;
        wake_buffer.ticks     = 0;
        /*在指定时间唤醒任务*/
        delta_task_wake_when( &wake_buffer );
        ret = delta_clock_get( DELTA_CLOCK_GET_TOD, &time );
        printf("\n*****task_2 wake on %d/%d/%d, %d:%d:%d:%d.*****\n\r",time.year,
            time.month,time.day,time.hour,time.minute,time.second,time.ticks);
        delta_task_delete( DELTA_SELF );
}
delta_task task_3(void)
{
        delta_status_code ret;
        delta_time_of_day time;
        while (1)
        {
            /*得到系统日期时间*/
            ret = delta_clock_get( DELTA_CLOCK_GET_TOD, &time );
            printf("\nBeijing Date is    %d/%d/%d, %d:%d:%d:%d.\n\r",time.year,
                time.month,time.day,time.hour,time.minute,time.second,time.ticks);
            /*1000 tick  后唤醒任务*/
            delta_task_wake_after(1000);
        }
}
```

在这个测试用例中，同时采用了硬件实时时钟 RTC 和硬件定时器 TIMER。硬件实时时钟 RTC 负责维护时间秒的整数部分，硬件定时器 TIMER 负责维护时间秒的小数部分。

预计结果：在 Init()函数中首先设定系统的绝对时间为 2001 年 1 月 1 日 12

点 0 分 0 秒，随后创建了三个任务。任务 1 被设定为睡眠状态，在 2001 年 1 月 1 日 12 点 0 分 5 秒这一时刻被唤醒。任务 2 被设定为睡眠状态，在 2001 年 1 月 1 日 12 点 0 分 10 秒这一时刻被唤醒。任务 3 被设定为周期性的动作，首先进入睡眠状态，1000 个 tick 后被唤醒，随后又进入睡眠状态，如此周而复始。

根据执行的过程，每一秒钟都应该能够通过串口接收到当前的时间，而且当系统时间为 2001 年 1 月 1 日 12 点 0 分 5 秒的时候，串口应该接收到信息“ *****task_1 wake on 2001/1/1, 12:0:5:0.***** ”，当系统时间为 2001 年 1 月 1 日 12 点 0 分 10 秒的时候，串口应该接收到信息“ *****task_2 wake on 2001/1/1, 12:0:10:0.***** ”。

测试过程：与时间片轮转算法测试的过程基本一致。

测试结果：通过串口接收到了下面这些信息。

****** Preparation Done. ******

*** Enter Task Wake test***

*****Task wake test start on 2001/1/1, 12:0:0:0.*****

Beijing Date is 2001/1/1, 12:0:0:68.

Beijing Date is 2001/1/1, 12:0:1:130.

Beijing Date is 2001/1/1, 12:0:2:171.

Beijing Date is 2001/1/1, 12:0:3:212.

Beijing Date is 2001/1/1, 12:0:4:253.

*****task_1 wake on 2001/1/1, 12:0:5:0.*****

Beijing Date is 2001/1/1, 12:0:5:294.

Beijing Date is 2001/1/1, 12:0:6:335.

Beijing Date is 2001/1/1, 12:0:7:376.

Beijing Date is 2001/1/1, 12:0:8:417.

Beijing Date is 2001/1/1, 12:0:9:458.

*****task_2 wake on 2001/1/1, 12:0:10:0.*****

Beijing Date is 2001/1/1, 12:0:10:499.

……

测试结论：测试结果与预计结果相符合，可以证明 NET+50 目标机上硬件

实时时钟 RTC 和硬件定时器 TIMER 的工作都正常。

类似地经过集成测试，16 个测试用例的执行全部正确，可以充分证明 DeltaCORE 已经成功地被移植到 NET+50 目标机上。

后记

现在回想起这一个多月的项目实践，觉得收获颇丰。

首先，对嵌入式系统的历史和发展有了一定的了解，对嵌入式系统的开发过程有了一定的感性认识。

其次，对嵌入式实时操作系统的内核有了初步的认识，消除了以前的恐惧感，使以前书本上学到的东西与实际开发结合起来了，并巩固了一些基础知识。

最后，使自己形成了一套良好的分析问题、解决问题的方法。

与此同时，也感觉到自己在很多方面也还有欠缺，需要进一步地努力加以改进。

完成整个移植工作之后，我真真切切地感受到 DeltaCORE 的可移植性是非常好的，移植起来非常方便、快捷。

随着嵌入式系统的不断发展，对嵌入式操作系统的内核进行移植可以使优秀的嵌入式操作系统能够应用在更广泛的领域，为人们的生产和生活带来了极大的方便。因此，内核移植工作是非常有意义，也是非常有发展前景的。

致谢

本文的最后我要向我的家人、指导者衷心地道一声感谢。我所取得的成绩离不开他们的大力支持和帮助。

指导者朱明远教授对我在嵌入式应用领域中的成长付出了极大的心血。他以全面的技术知识、开阔的眼界、实干家的气魄和对事业倾注的热情将我们北京科银京成技术有限公司的全体员工凝聚在一起，为了一个共同的事业，并肩作战，不断创造出奇迹。在此，我谨向朱老师表示我忠心的感谢。

指导者陈朔鹰老师是我步入嵌入式应用领域的引路人。他广博的知识、开阔的思路和对嵌入式事业的热情与执着深深地感染着我们每一个人。我在事业上所能取得的进步和成绩离不开陈老师对我的信赖、鼓励和支持。

诚挚地感谢我的家人自始至终给予的关心和鼓励。

在一个月左右的工程项目中，在项目组长张庆莉的悉心指导下，与小组的合作伙伴刘兆伟、孙梦、刘向前相互研究、相互讨论，一起度过了这段难忘的美好时光。合作伙伴们给予了我很大的帮助和支持，在这里也要对他们特别表示感谢！

最后祝北京科银京成技术有限公司在新世纪再创新佳绩。

参考文献

[1]. 马忠梅等．ARM 嵌入式处理器结构与应用基础．北京航空航天大学出版社．2002 年．北京

[2]. David Seal．ARM Architecture Reference Manual．Pearson Education limited．2001 年．英国

[3]. 何荣森 何希顺 张跃．从 ARM 体系看嵌入式处理器的发展．微电子与计算机 2002 年第 5 期．北京

[4]. 屈振新 曾庆伟 韩波．嵌入式实时操作系统核心的设计与实现．计算机工程与应用．2002 年第 9 期．北京

[5]. WALTER A.TRIEBEL．硬件、软件及接口技术．清华大学出版社．1999 年．北京

[6]. 王学龙．嵌入式 Linux 系统设计与应用．清华大学出版社．2001 年．北京

[7]. μC/OS-Ⅱ——源码公开的实时嵌入式操作系统．中国电力出版社．2001 年．北京

[8]. 徐国治等．全国大学生电子设计竞赛——嵌入式系统专题竞赛培训讲义．上海交通大学电子工程系．2002 年．上海

[9]. NET+Works for NET+ARM Hardware Reference Guide．NetSilicon Inc.．2001年．美国

[10]. 北京科银京成技术有限公司系统部内核组．DeltaCORE参考手册．北京科银京成技术有限公司．2001年．北京

[11]. 程红蓉．一种实时嵌入式操作系统内核DeltaCore的设计与实现．电子科技大学硕士论文．2001年．成都

[12]. 蒋本珊．电子计算机组成原理．北京理工大学出版社．1999 年．北京

[13]. 张丽芬．操作系统原理与设计．北京理工大学出版社．1997 年．北京

[14]. 李善平 陈文智等．边干边学——LINUX 内核指导．浙江大学出版社．2002 年．杭州

[15]. 毛德操 胡希明．LINUX 内核源代码情景分析．浙江大学出版社．2001 年．杭州

[16]. 郑纬民 汤志忠．计算机系统结构（第二版）．清华大学出版社．1998 年．北京

[17]. Alessandro Rubini Jonathan Corbet．LINUX 设备驱动程序（第二版）．中国电力出版社．2002 年．北京

[18]. Richard M. Stallman Roland McGrath．GNU Make A Program for Directing Recompilation describing make Version 3.74．GNU．1996 年．美国

[19]. Roger S.Pressman．软件工程——实践者的研究方法．机械工业出版社．2002 年．北京

[20]. 郑人杰 殷人昆．软件工程概论．清华大学出版社．1998 年．北京

[21]. 贺炘．单元测试的基本方法．http://www.51cmm.com/SoftTesting/-No009.htm．2003 年．长沙

[22]. 软件工程专家网．软件测试的基本方法．http://www.51cmm.com/-SoftTesting/No027.htm．2003 年．长沙

[23]. 软件工程专家网．软件测试概述．http://www.51cmm.com/-SoftTesting/No013.htm．2003 年．长沙

射 频 和 天 线 设 计 培 训 课 程 推 荐

易迪拓培训(www.edatop.com)由数名来自于研发第一线的资深工程师发起成立，致力并专注于微波、射频、天线设计研发人才的培养；我们于 2006 年整合合并微波 EDA 网(www.mweda.com)，现已发展成为国内最大的微波射频和天线设计人才培养基地，成功推出多套微波射频以及天线设计经典培训课程和 ADS、HFSS 等专业软件使用培训课程，广受客户好评；并先后与人民邮电出版社、电子工业出版社合作出版了多本专业图书，帮助数万名工程师提升了专业技术能力。客户遍布中兴通讯、研通高频、埃威航电、国人通信等多家国内知名公司，以及台湾工业技术研究院、永业科技、全一电子等多家台湾地区企业。

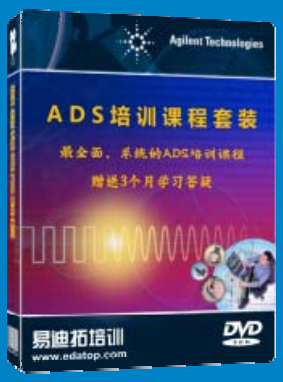
易迪拓培训课程列表：http://www.edatop.com/peixun/rfe/129.html

射频工程师养成培训课程套装

该套装精选了射频专业基础培训课程、射频仿真设计培训课程和射频电路测量培训课程三个类别共 30 门视频培训课程和 3 本图书教材；旨在引领学员全面学习一个射频工程师需要熟悉、理解和掌握的专业知识和研发设计能力。通过套装的学习，能够让学员完全达到和胜任一个合格的射频工程师的要求…

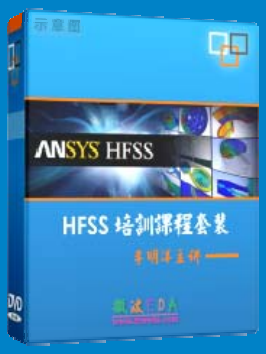
课程网址：http://www.edatop.com/peixun/rfe/110.html

ADS 学习培训课程套装



该套装是迄今国内最全面、最权威的 ADS 培训教程，共包含 10 门 ADS 学习培训课程。课程是由具有多年 ADS 使用经验的微波射频与通信系统设计领域资深专家讲解，并多结合设计实例，由浅入深、详细而又全面地讲解了 ADS 在微波射频电路设计、通信系统设计和电磁仿真设计方面的内容。能让您在最短的时间内学会使用 ADS，迅速提升个人技术能力，把 ADS 真正应用到实际研发工作中去，成为 ADS 设计专家...

课程网址： http://www.edatop.com/peixun/ads/13.html



HFSS 学习培训课程套装

该套课程套装包含了本站全部 HFSS 培训课程，是迄今国内最全面、最专业的 HFSS 培训教程套装，可以帮助您从零开始，全面深入学习 HFSS 的各项功能和在多个方面的工程应用。购买套装，更可超值赠送 3 个月免费学习答疑，随时解答您学习过程中遇到的棘手问题，让您的 HFSS 学习更加轻松顺畅…

课程网址：http://www.edatop.com/peixun/hfss/11.html

CST 学习培训课程套装

该培训套装由易迪拓培训联合微波 EDA 网共同推出，是最全面、系统、专业的 CST 微波工作室培训课程套装，所有课程都由经验丰富的专家授课，视频教学，可以帮助您从零开始，全面系统地学习 CST 微波工作的各项功能及其在微波射频、天线设计等领域的设计应用。且购买该套装，还可超值赠送 3 个月免费学习答疑…

课程网址：http://www.edatop.com/peixun/cst/24.html

HFSS 天线设计培训课程套装

套装包含 6 门视频课程和 1 本图书，课程从基础讲起，内容由浅入深，理论介绍和实际操作讲解相结合，全面系统的讲解了 HFSS 天线设计的全过程。是国内最全面、最专业的 HFSS 天线设计课程，可以帮助您快速学习掌握如何使用 HFSS 设计天线，让天线设计不再难…

课程网址：http://www.edatop.com/peixun/hfss/122.html

13.56MHz NFC/RFID 线圈天线设计培训课程套装

套装包含 4 门视频培训课程，培训将 13.56MHz 线圈天线设计原理和仿真设计实践相结合，全面系统地讲解了 13.56MHz 线圈天线的工作原理、设计方法、设计考量以及使用 HFSS 和 CST 仿真分析线圈天线的具体操作，同时还介绍了 13.56MHz 线圈天线匹配电路的设计和调试。通过该套课程的学习，可以帮助您快速学习掌握 13.56MHz 线圈天线及其匹配电路的原理、设计和调试…

详情浏览：http://www.edatop.com/peixun/antenna/116.html

我们的课程优势：

※ 成立于 2004 年，10 多年丰富的行业经验，

※ 一直致力并专注于微波射频和天线设计工程师的培养，更了解该行业对人才的要求

※ 经验丰富的一线资深工程师讲授，结合实际工程案例，直观、实用、易学

联系我们：

※ 易迪拓培训官网：http://www.edatop.com

※ 微波 EDA 网：http://www.mweda.com

※ 官方淘宝店：http://shop36920890.taobao.com