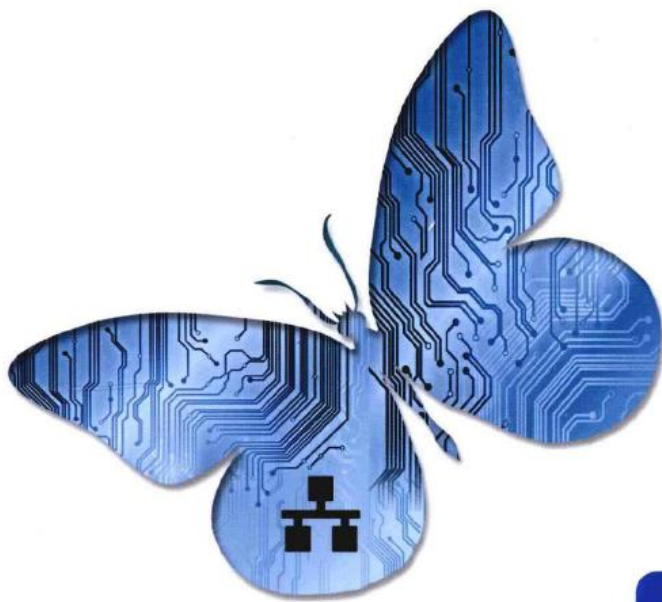




首本从系统视角讲解STM32开发规范和编码规范的书，学习嵌入式系统开发必读。
全方位讲解在STM32F107开发板上移植FreeRTOS和LwIP的全过程。
配套极具性价比的硬件开发平台，并提供完整工程文件和源代码，极具可操作性。



单片机与嵌入式



STM32

STM32嵌入式 系统开发实战指南

FreeRTC

联合移植

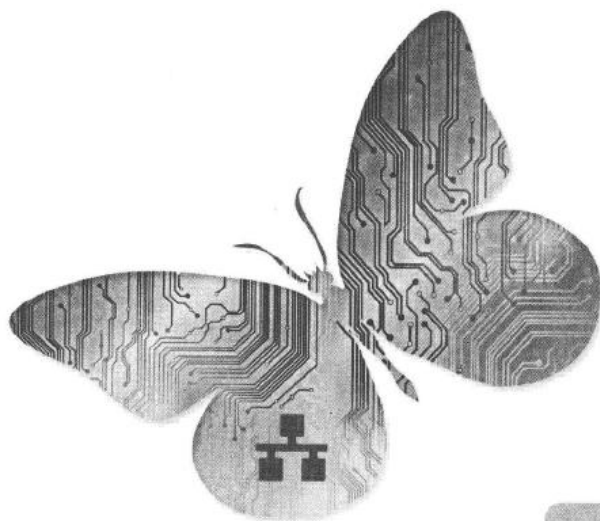
李志明 檀永 徐石明 等编著



机械工业出版社
China Machine Press



单片机与嵌入式



STM32

STM32嵌入式 系统开发实战指南

FreeRTOS 移植

李志明 檀永 徐石明 丁孝华 桑林 编著



机械工业出版社
China Machine Press

图书在版编目(CIP)数据

STM32嵌入式系统开发实战指南: FreeRTOS与LwIP联合移植 / 李志明等编著. —北京: 机械工业出版社, 2013.4

ISBN 978-7-111-41716-3

I. S… II. 李… III. 实时操作系统 IV. TP316.2

中国版本图书馆CIP数据核字(2013)第042854号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书以一款轻量级嵌入式实时操作系统为样本, 阐述了嵌入式实时操作系统任务管理、时间管理、资源共享、内存管理等机制, 介绍了内核及TCP/IP的移植和具体使用方法。为了避免枯燥的理论阐述, 本书辅以适量的例程帮助大家学习。此外, 本书还简要阐述了硬件平台设计、项目开展的一般步骤和注意事项。

本书适合已熟悉STM32的操作、掌握基于STM32官方驱动库的前后台模式应用软件开发读者或初级嵌入式软件开发工程师阅读。

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 秦 健

北京市荣盛彩色印刷有限公司印刷

2013年5月第1版第1次印刷

186mm×240mm·21印张

标准书号: ISBN 978-7-111-41716-3

ISBN 978-7-89433-833-4(光盘)

定 价: 69.00元(附光盘)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

购书热线: (010) 68326294 88379649 68995259

投稿热线: (010) 88379604

读者信箱: hzsj@hzbook.com

序 言

嵌入式系统起源于微型机时代，伴随着网络、通信、多媒体技术的发展，嵌入式系统的应用已经深入社会生活的各个领域，业已成为继个人计算机和互联网之后，信息技术领域技术的新热点。

测量与控制的需求一直伴随着人类社会的进步，在人类进入现代化工业社会后，人们对这种需求有了更高的要求 and 期待。从以蒸汽机的发明为标志的工业革命开始，人们对控制有了全新的认识。最初，人们运用机械原理实现复杂程度不一的机械结构来满足对工程控制的需求，随着电子技术的发展及其展现的优点，人们又逐渐采用模拟、数字或混合式的电子控制方式来完成既定的测量控制需求。直到 20 世纪 70 年代，新型的微型计算机凭借其体积小、功耗低、结构简单、可靠性高、使用方便、性能价格比高等一系列优点，及其表现出的智能化水平迅速获得了控制专业人士的青睐。例如，将微型计算机经适当的机械加固和电气改造，并配置相应外围接口电路，即可实现各种复杂的工程应用，如工况监测、实时控制等。在微型计算机诞生后相当长的时期，这种将微型计算机嵌入一个对象体系中，实现对象体系的智能化控制得到了广泛的工程应用。

但微型计算机的设计是以商业应用为初衷的，微型计算机的体积、价位和可靠性都无法满足广大对象系统的嵌入式应用要求。半导体厂商逐渐意识到嵌入式系统的潜在市场需求，并陆续推出了各具特色的微型化控制芯片。1976 年，Intel 公司推出了 MCS-48 单片机，开创了将微处理机系统的各种 CPU 以外的资源集成到 CPU 硅片上的时代。1980 年，Intel 公司对 MCS-48 单片机进行了全面完善，推出了 8 位 MCS-51 单片机，并获得巨大成功，奠定了嵌入式系统的单片机应用模式。1984 年，Intel 公司又推出了 16 位 8096 系列并将其称为嵌入式微控制器，“嵌入式”一词第一次在微处理机领域出现，这标志嵌入式应用的兴起与快速发展的时代来临。

随着半导体制造工艺的进步，为了满足高速、实时信号处理的市场需求，数字信号处理芯片（DSP）及可编程逻辑器件（PLD、CPLD、FPGA）等高速嵌入式处理器件应运而生。DSP 是将模拟信号转换成数字信号以后进行高速实时处理的专业处理器，在其诞生的最初只能完成既定的逻辑算术运算，但其处理速度已远远超越当时的微控制器。随着集成电路技术的发展，DSP 处理能

力得到了不断提升,当前基于 DSP 的工程应用主要着眼于算法设计和实现,已广泛应用于数字通信、测量控制、图像处理等领域。可编程逻辑器件的发展始于 20 世纪 70 年代,在经历了 40 多年的发展后,已形成了以现场可编程门阵列器件为代表的特色各异的信号处理器件。FPGA 通常比 DSP 拥有更快的运行速度,可以实现复杂的高速逻辑运算,其具有设计灵活、高集成度、高速、高可靠性、开发周期短、前期投资风险小等优点,在芯片内部可实现板级电路的功能,能有效提高设计的效率和可靠性。此外,随着 FPGA 芯片供应商和第三方公司对应用开发支持的进一步完善,芯片的设计周期和成本得以进一步压缩。

自嵌入式处理器诞生后的 40 多年历史中,面向各种不同应用领域、特色各异的嵌入式处理器件不断出现、升级和换代,广泛应用于人类的社会生活,也极大地改善了人类的生活。特别在 20 世纪 90 年代后,在网络、通信、多媒体技术应用需求的驱动下,嵌入式处理器经历了高度的发展和工程应用。随着 32 位微处理器和 32 位微控制器成本大幅下降,它们已经逐渐取代了传统 8 位、16 位微控制器。与此同时,由于面向嵌入式系统的高端应用的工作速度快,外部设备资源丰富,加上应用本身的复杂性、可靠性要求等,软件开发的复杂性逐渐提升,如网络通信设备、电力二次设备、手持终端、多媒体设备、机顶盒等。这促使嵌入式软件开发工程师渴望摆脱繁重的底层软件开发,将更多的精力集中于应用层面的开发。嵌入式软件开发工程师希望能够提供一种类似于微型计算机操作系统的平台,只需根据实际采用的硬件平台做少量的移植工作即可完成嵌入式系统资源的管理和任务调度,在此基础上将主要精力集中于任务级代码的实现。

嵌入式系统最初的应用是基于单片机的,大多数以可编程控制器的形式出现,具有监测、控制、工况指示等功能,在工业和军事应用领域最为普遍。由于受限于系统资源,通常没有获得操作系统的支持,只能通过汇编语言对系统进行直接控制,所以只能使用 8 位的单片机芯片来运行一些单线程的程序。20 世纪 80 年代,随着 I/O 接口、串行接口以及 RAM、ROM 等外设部件集成在芯片微处理器中,芯片制造商推出了面向 I/O 设计的微控制器,一经推出便迅速得到了市场和专业人士的认可。与此同时,业界出现了功能较为简单的“操作系统”雏形,使得开发周期和效率得到了一定的提升。直到 20 世纪 90 年代,在分布控制、柔性制造、网络通信和信息家电等巨大需求的驱动下,嵌入式系统进一步飞速发展,而面向实时信号处理的嵌入式处理器也向高速度、高精度、低功耗的方向发展。随着硬件实时性要求的提高,嵌入式系统的软件规模不断扩大,复杂程度也逐渐提高,实时多任务操作系统(Real-Time Operating System, RTOS)诞生,并开始成为嵌入式系统的主流,例如 WindRiver 公司的 VxWorks、QNX 系统软件公司的 QNX、嵌入式 Linux、微软公司的 Windows CE、美国系统集成公司的 pSOS、Micrium 公司的 uC/OS、FreeRTOS、RT-thread 等。

实时操作系统是一种能够在既定时间内完成系统功能,对外部和内部事件在同步或者异步时间内做出及时响应,并控制所有实时任务协调一致运行的系统。在实时操作系统中,操作的正确

性不仅依赖于逻辑设计的正确程度，而且与系统任务级时序密切相关。因而，实时系统具有及时响应和高可靠性等主要特点。通常，实时操作系统有硬实时和软实时之分，硬实时要求在规定的时间内必须完成既定的操作，否则有可能造成灾难性后果；软实时则只要按照任务的优先级，尽可能快地完成操作即可，广泛应用于消费类电子产品，这类操作系统对事件响应没有严格要求，较为关注事件响应的正确性和用户体验的舒适度，如产品的按键操作应不致使用户产生“反应迟钝”的感受。当然，上述这种划分仅具有概念上的意义，实时性对不同应用有着不同的界定，而且大多数嵌入式系统允许软硬两种实时性同时存在，对系统产生关键影响的事件一般具有严格的时限要求，是硬实时的，另外一些事件的时限要求则是软实时的，诸如人机交互。也有业内人士根据系统对任务的响应时间将实时操作系统分为弱实时系统、一般实时系统和强实时系统。无论如何划分，其本质是对系统事件响应时间的一种要求和评判。嵌入式软件开发工程师在项目规划时需根据工程实际需求，在硬件配置、实时操作系统选择和项目成本之间进行一定的统筹和折中，结合应用层代码开发策略和技巧，完成系统的实时性要求。

大多数嵌入式实时操作系统采用了微内核结构，即内核只提供诸如资源管理、内存管理和任务调度等基本功能，而网络通信功能、文件系统、GUI 系统等应用组件均驻留于用户进程（任务）中，或以函数调用的方式工作。用户可以根据实际工程的需求适当裁剪，选用相应的组件。嵌入式操作系统与通用操作系统类似，具有管理系统资源、物理层抽象的功能，能够提供库函数、驱动程序、开发工具集等。但与通用操作系统相比，嵌入式操作系统在系统实时性、硬件依赖性、软件固化性以及应用专用性等方面，具有更加鲜明的特点。目前，嵌入式系统的主流趋势是 32 位嵌入式微处理器或微控制器与实时多任务操作系统的结合，嵌入式系统往往指的是包含这种资源的系统。

需要指出的是，本书中涉及的“微处理器”和“微控制器”虽无本质上的区别，但从其组成要素及设计特色来讲，可以总结为如下几点：

□ 硬件结构

微处理器是一个单芯片的处理器，而微控制器则在一块集成电路芯片中集成了处理器和其他片内外设，功能相对健全，基本构成了一个完整的微型计算机系统。片内外设通常包括 RAM、ROM、串行接口、并行接口、计时器和中断管理等。微控制器具有体积小、功耗低、结构简单、可靠性高、使用方便等特点，应用领域遍及国防、工业、汽车、医疗设备和消费电子等几乎所有的行业和领域。此外，作为面向控制的设备，为了保证系统的实时性，微控制器必须拥有强大的中断功能，能够及时响应外界的刺激，快速执行上下文切换，因此微控制器在片上集成了所有处理中断必需的电路。

□ 应用领域

微处理器以微型计算机系统应用为设计初衷，适用于在计算机系统中处理信息。这也是微处

理器的优势所在。微控制器通常用于面向控制的应用，其系统设计追求小型化、低功耗，系统设计时尽可能减少元器件数量，适用于那些以极少的元件实现对输入/输出设备进行控制的场合。

□ 指令集特征

由于应用场合不同，微控制器和微处理器的指令集也有所不同。微处理器的指令集增强了处理功能，使其拥有强大的寻址模式和适于操作大规模数据的指令。另外，微处理器还具有其他特点，如用户程序中无法使用特权指令等。微控制器的指令集适用于输入/输出控制。许多输入/输出的接口是以位为单位寻址的。而微处理器通常不具备这些强大的位操作能力，因为设计者在设计微处理器时，仅考虑以字节或更大的单位来操作数据。

□ 处理能力

就处理能力而言，微处理器要远高于微控制器的处理水平，微控制器以牺牲处理能力来实现其他片内外设和功能。微控制器受限于片内资源（ROM、RAM），它的指令必须非常精简，大部分指令的长度都短于1个字节。微控制器指令集的基本特点就是具有精简的编码方案。而微处理器庞大的指令集使得其编码要远比微控制器复杂。

本书基于意法半导体公司的32位嵌入式微控制器，以FreeRTOS为嵌入式实时操作系统，讲述了FreeRTOS的移植及工程应用开发的全过程。本书定位于已熟悉STM32寄存器的操作，掌握了基于STM32官方驱动库的前后台模式应用软件开发的读者或初级嵌入式软件开发工程师，并希望在此基础上向读者展示嵌入式操作系统和TCP/IP协议的移植方法和要点。本书第四篇以一个完整工程项目为导引，向读者展示一个基于STM32F107硬件平台的嵌入式实时操作系统移植开发过程。读者在学习完本书内容后，将能够独立进行基于嵌入式实时操作系统和TCP/IP协议的移植和应用系统开发，并能够完成从初级软件开发者到高级开发者的转变。

前言

自 20 世纪 90 年代, 鉴于多任务支持、开发便捷、便于维护等特性, 同时能够提高系统的稳定性和可靠性, 嵌入式实时操作系统 (RTOS) 逐渐为广大嵌入式从业人员所接受和认可, 越来越多的工程师加入使用 RTOS 的队伍。

与此同时, 半导体技术的快速发展及市场需求的多样化对 RTOS 提出了更高的要求。一方面, 新型处理器的大量涌现要求 RTOS 自身结构的设计应易于移植, 以适应不同硬件架构平台的应用。另一方面, 人们在使用 RTOS 进行系统设计的同时, 不仅希望得到供应商的技术支持, 而且希望获得 RTOS 的源代码, 以便对 RTOS 做出符合工程实际需求的裁剪, 并降低硬件平台的构建成本。如通常裁剪后的内核对 ROM、RAM 的容量占用量更小, 用户可以选择更小容量的存储器以降低成本。为了适应这种市场需求, 许多 RTOS 提供商在出售 RTOS 时附加了源程序的代码, 在众多的 RTOS 供应商中也不乏免费开放源代码的 RTOS。本书以一款轻量级开源 RTOS 为样本, 通过适当的例程阐述了嵌入式实时操作系统任务管理、时间管理、资源共享、内存管理等机制, 介绍了 RTOS 内核及 TCP/IP 协议栈的移植和具体使用方法。

本书内容及读者对象

全书分为四篇, 共 13 章。第一篇 (第 1~5 章) 讲述了 ARM 处理器的发展沿革及技术特点、基于 STM32 的硬件平台、开发环境的搭建及在工程应用中的选型要点。同时, 结合笔者的工程开发经验, 简要介绍了成为一名合格嵌入式软件开发工程师应该具备的工程素养。对于已熟悉 ARM 硬件知识的读者, 可跳过第 1~3 章的内容。第二篇 (第 6~8 章) 简要介绍了嵌入式实时操作系统的基本概念, 重点讲述了一个轻量级嵌入式实时操作系统的使用方法, 并给出了基于 STM32 微控制器的移植代码实现。第三篇 (第 9~11 章) 结合 TCP/IP 原理介绍 LwIP 开源轻量级协议栈的基本原理及常见应用模式, 第 11 章实现了前后台模式下的 TCP/IP 协议栈移植, 并给出了源代码。第四篇 (第 12~13 章) 首先介绍了在 STM32F107 微控制器上移植 FreeRTOS 和 LwIP 的全过程, 随后介绍了工业通信网关的一般实现方式, 作为示例, 简要实现了以太网实现通信报文的转发和板载资源的控制。

本书以已熟悉 STM32 寄存器的操作、掌握基于 STM32 官方驱动库的前后台模式应用软件开发读者或初级嵌入式软件开发工程师为对象，向读者讲述了嵌入式实时操作系统和 TCP/IP 协议栈的移植方法和要点，并从系统工程的角度简要介绍了嵌入式工程开发的一般步骤和方法。

致谢

本书的顺利完稿与出版离不开本书的编辑张国强先生的鼓励与支持，他对书稿提出的专业而宝贵的建议使得成书的质量更进一步。笔者对他在书稿审阅和校对过程中付出的辛勤劳动表示衷心的感谢。本书在编写过程中参考了大量书籍和资料，参考文献中未能将其一一列出，在此对书中参考资料的作者一并表示感谢。最后，感谢广大的读者朋友，感谢您花费时间和精力阅读本书，由于水平有限书中难免存在疏漏与错误，诚恳地希望您批评指正。

目 录

序 前 言

第一篇 平台篇

第 1 章 ARM 处理器简介 2

- 1.1 ARM 内核处理器沿革 2
 - 1.1.1 传统 ARM 处理器 3
 - 1.1.2 Cortex 内核处理器 4
- 1.2 Cortex 内核系列处理器技术特点 8
 - 1.2.1 ARM Cortex-M 系列处理器 8
 - 1.2.2 ARM Cortex-R 系列处理器 10
 - 1.2.3 ARM Cortex-A 系列处理器 11
- 1.3 STM32 互联型嵌入式控制器 12
- 1.4 微控制器选型 14
 - 1.4.1 选型因素 14
 - 1.4.2 选型示例 17

第 2 章 基于 STM32F107 的 开发板 18

- 2.1 STM32F107 开发板 18
- 2.2 主要板载资源 19
 - 2.2.1 10/100M 以太网接口 19
 - 2.2.2 CAN 总线接口 27
 - 2.2.3 RS485 总线接口 35
 - 2.2.4 其他总线接口 37

- 2.3 硬件设计要点 44
 - 2.3.1 电磁兼容问题 44
 - 2.3.2 信号完整性 45
 - 2.3.3 电源完整性 47

第 3 章 开发环境 50

- 3.1 开发环境及搭建 50
 - 3.1.1 常见开发环境 50
 - 3.1.2 IAR EWARM 安装 52
 - 3.1.3 RealView MDK 安装 54
- 3.2 相关开发工具 57
- 3.3 创建工程 58

第 4 章 编程规范 72

- 4.1 ST 固件库编程规范 72
 - 4.1.1 缩写 72
 - 4.1.2 命名规则 73
 - 4.1.3 编码规则 73
- 4.2 基于 C 语言的嵌入式编程规范 77
 - 4.2.1 源代码的排版 77
 - 4.2.2 源代码的注释 80
 - 4.2.3 标识符命名 86
 - 4.2.4 代码可读性 88
 - 4.2.5 变量、结构 90
 - 4.2.6 函数、过程 94

4.2.7 可测性	102
4.2.8 程序效率	106
4.2.9 质量保证	109
4.2.10 代码编辑、编译、审查	115
4.2.11 测试与维护	116
4.2.12 宏定义	116

第 5 章 项目规划.....118

5.1 概述	118
5.2 系统分析	118
5.3 系统设计	119
5.4 系统制造	119
5.5 系统运用及反馈	120
5.6 开发团队	120
5.6.1 团队负责人	120
5.6.2 调研人员	121
5.6.3 开发人员	122

第二篇 RTOS 篇

第 6 章 操作系统原理基础知识.....126

6.1 前后台模式应用程序	126
6.2 嵌入式操作系统	126
6.2.1 相关基本概念	126
6.2.2 系统调用	127
6.2.3 操作系统结构	127
6.2.4 进程与任务	129
6.2.5 进程间的通信	129
6.2.6 进程调度	129
6.2.7 存储管理	130

第 7 章 FreeRTOS 嵌入式操作系统.....131

7.1 FreeRTOS 特色	131
7.2 任务管理	131
7.2.1 任务函数	131
7.2.2 基本任务状态	132
7.2.3 任务创建	133
7.2.4 任务的优先级	138
7.2.5 非运行状态	141
7.2.6 空闲任务及回调函数	147
7.2.7 改变任务优先级	148
7.2.8 删除任务	151
7.2.9 调度算法概述	153
7.3 队列管理	155
7.3.1 概述	155
7.3.2 使用队列	157
7.3.3 大型数据单元传输	168
7.4 中断管理	169
7.4.1 延迟中断处理	169
7.4.2 计数信号量	175
7.4.3 在中断服务例程中使用队列	179
7.4.4 中断嵌套	184
7.5 资源管理	185
7.5.1 基本概念	185
7.5.2 临界区与挂起调度器	187
7.5.3 互斥量	189
7.5.4 互斥的另一种实现	195
7.6 内存管理	199
7.6.1 概述	199
7.6.2 内存分配方案范例	199
7.7 常见错误	202

7.7.1 概述	202
7.7.2 栈溢出	202
7.7.3 其他常见错误	204

第 8 章 基于 STM32F107 的 FreeRTOS 移植

8.1 概述	206
8.2 FreeRTOS 移植	206
8.2.1 portmacro.h 头文件	206
8.2.2 port.c 源文件	209
8.2.3 portasm.s 汇编源文件	211
8.2.4 其他问题	215
8.3 创建测试任务	215

第三篇 LwIP 篇

第 9 章 TCP/IP 协议栈介绍 ...

9.1 引言	220
9.2 网络分层	220
9.2.1 OSI 七层参考模型	220
9.2.2 TCP/IP 分层	222
9.2.3 TCP/IP 协议簇的协议	223
9.3 IP 协议	224
9.4 ARP 协议与 RARP 协议	225
9.5 ICMP	226
9.6 TCP 协议	227
9.7 UDP 协议	229
9.8 FTP 协议	230

第 10 章 LwIP 轻量级 TCP/IP 协议栈

10.1 LwIP 进程模型	232
10.2 LwIP 缓冲与内存管理	233

10.2.1 LwIP 动态内存管理机制	233
10.2.2 LwIP 的缓冲管理机制	236
10.3 LwIP 网络接口	240
10.4 LwIP 的 ARP 处理	245
10.5 LwIP 的 IP 处理	248
10.6 LwIP 的 ICMP 处理	251
10.7 LwIP 的 UDP 处理	252
10.8 LwIP 的 TCP 处理	254
10.8.1 TCP 处理流程概述	254
10.8.2 TCP 控制块	255
10.8.3 LwIP 的 TCP 滑动窗口	259
10.8.4 LwIP 的 TCP 超时与重传	260
10.8.5 LwIP 的 TCP 拥塞控制	262
10.8.6 LwIP 的 TCP 定时器	263
10.9 LwIP 的应用程序接口简介	264
10.9.1 RAW API 接口	264
10.9.2 Sequential API 接口	267

第 11 章 基于 STM32F107 的 LwIP 移植

11.1 ethernetif.c 文件的移植	274
11.1.1 ethernetif_init 函数	274
11.1.2 low_level_init 函数	274
11.1.3 ethernetif_input 函数	276
11.1.4 low_level_input 函数	276
11.1.5 low_level_output 函数	277
11.2 网络驱动移植	278
11.2.1 以太网控制器概述	278
11.2.2 以太网控制器硬件配置	279
11.2.3 以太网控制器硬件的引脚 配置	282
11.2.4 以太网驱动之接收	283
11.2.5 以太网驱动之发送	284

11.2.6 其他注意事项	285
11.3 基于 RAW API 接口的 HelloWorld 例程	287

第四篇 移植篇

第 12 章 基于 FreeRTOS 的 LwIP 协议栈移植	294
12.1 概述	294
12.2 FreeRTOS 下以太网驱动 程序的移植	295
12.3 LwIP 程序移植	296
12.3.1 以太网接口文件 ethernetif.c	

的移植	296
12.3.2 操作系统模拟层文件 sys_arch.c 的移植	298

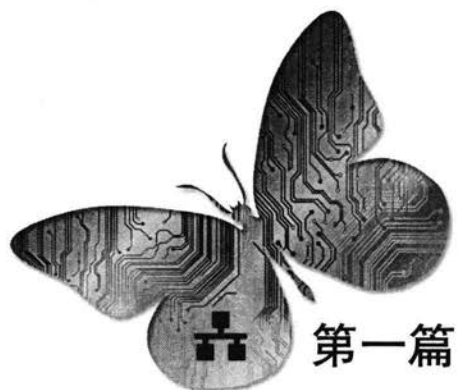
第 13 章 工业通信网关解析 ... 306

13.1 概述	306
13.2 编码实现	306
13.3 通信测试	310

附录 A 开发板原理图 313

附录 B 专业术语 319

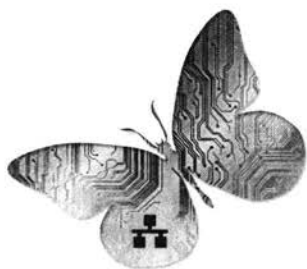
参考文献 321



平 台 篇

本篇主要介绍基于 STM32 的硬件平台、开发环境的搭建及在工程应用中的选型等要点。

- ☐ 第1章 ARM处理器简介
- ☐ 第2章 基于STM32F107的开发板
- ☐ 第3章 开发环境
- ☐ 第4章 编程规范
- ☐ 第5章 项目规划



第 1 章

ARM 处理器简介

本章主要讲述 ARM 系列处理器的发展沿革，从处理器技术特点的角度对各系列处理器进行介绍，着重讲述了 Cortex 内核处理器的技术特点。最后，简要介绍了意法半导体公司的以 Cortex-M3 为内核的互联型控制器。

1.1 ARM 内核处理器沿革

ARM (Advanced RISC Machines) 是微处理器行业的一家知名企业，1991 成立于英国剑桥，该公司主要出售芯片设计技术的授权。人们将采用 ARM 技术知识产权 (IP) 核的微处理器称为 ARM 微处理器。ARM 公司利用独特的商业模式在全球范围内拥有极其广泛的合作伙伴。ARM 公司将其技术授权给世界上许多著名的半导体、软件和 OEM 厂商，每个厂商得到的都是 ARM 公司提供的一套独一无二的 ARM 相关技术及服务，这些合作伙伴又保证了大量的开发工具和丰富的第三方资源。利用这种合作关系，ARM 公司很快成为许多全球性 RISC 标准的缔造者。

以 ARM 为处理器的应用领域已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75 % 以上的市场份额。世界上的主流半导体厂商已先后发布了基于 ARMv4、ARMv5、ARMv6、ARMv7 四个架构的 10 多个家族的主流嵌入式微控制器和微处理器产品，其中包括以经典的 ARM7、ARM9、ARM11 为内核的和以最新发布的 Cortex 为内核的种类繁多的微控制器和微处理器，如图 1.1 所示。目前，以 ARM 为内核的处理器大概有上千种，接下来简要介绍 ARM 处理器中的几个重要内核版本。

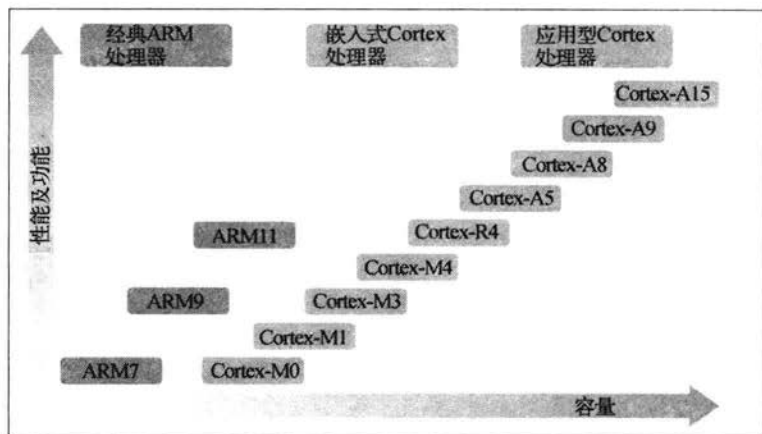


图 1.1 ARM 处理器及内核

提示 为方便起见,若无特别说明,后面内容对处理器和控制器不做详细区分,读者可根据上下文含义理解。对于二者的区别前文有简要的论述,读者可参阅。

1.1.1 传统 ARM 处理器

1. ARM7 处理器

ARM7 处理器采用了 ARMv4T (冯·诺依曼) 体系结构。这种体系结构将程序指令存储器和数据存储器合并在一起。主要特点就是程序和数据共用一个存储空间,程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置,采用单一的地址及数据总线,程序指令和数据的宽度相同。这样,处理器在执行指令时,必须先从存储器中取出指令进行译码,再取出操作数执行运算。总体来说,ARM7 体系结构的特点是:三级流水处理、空间统一的程序指令与数据 Cache、平均功耗为 0.6mW/MHz、时钟频率为 66MHz、每条指令平均执行 1.9 个时钟周期等。其中 ARM710、ARM720 和 ARM740 均为内置 Cache 的 ARM 核。ARM7 指令集与 Thumb 指令集扩展组合在一起,可以减少内存容量和系统成本。同时,ARM7 还利用嵌入式 ICE 调试技术简化系统设计,并通过一个 DSP 增强扩展来改进性能。ARM7 体系结构是小型、快速、低功耗、集成式的 RISC 内核结构。该产品主要用于早期的数字蜂窝电话和硬盘驱动器等,主流的 ARM7 内核是 ARM7TDMI、ARM7TDMI-S、ARM7EJ-S、ARM720T。市场上用得最多的 ARM7 处理器有 Samsung 公司的 S3C44B0X 与 S3C4510、Atmel 公司的 AT91FR40162 系列、Cirrus 公司的 EP73xx 系列等。目前,市场上以 ARM7 为处理器的产品大多已升级为 Cortex 内核的处理器。

2. ARM9、ARM9E 处理器

ARM9 处理器采用 ARMv4T (哈佛) 体系结构。这种体系结构将程序指令存储和数据存储分开,是一种并行体系结构。主要特点是程序和数据存储在不同的存储空间中,即程序存储器和数据存储器。它们是两个相互独立的存储器,每个存储器独立编址,独立访问。与两个存储器相对应的是体系结构中的 4 套总线:程序的数据总线和地址总线,数据的数据总线和地址总线。这种分离的程序总线和数据总线允许在一个机器周期内同时获取指令字和操作数,从而提高了执行速度,使数据的吞吐量提高了一倍。又由于程序存储器和数据存储器在两个分开的物理空间中,因而取指和执行能完全重叠。ARM9 体系结构的特点是:五级流水处理、分离的 Cache 结构、平均功耗为 0.7mW/MHz、时钟频率为 120~200MHz、每条指令平均执行 1.5 个时钟周期。与 ARM7 处理器系列相似,ARM9 中的 ARM920、ARM940 和 ARM9E 处理器均为内置 Cache 的 CPU 核,性能为 132MIPS (120MHz 时钟频率, 3.3V 供电) 或 220MIPS (200MHz 时钟频率)。ARM9 处理器同时也配备 Thumb 指令集扩展、调试和 Harvard 总线。在生产工艺相同的情况下,性能是 ARM7TDMI 处理器的两倍之多。ARM9 常用于无线设备、仪器仪表、联网设备、机顶盒设备、高端打印机及数码相机应用中。ARM9E 内核是在 ARM9 内核的基础上增加了紧密耦合存储器 TCM 及 DSP 部分。目前主流的 ARM9 内核是 ARM920T、ARM922T、ARM940。市场上相关的处理器有 Samsung 公司的 S3C2510、Cirrus 公司的 EP93xx 系列等。主流的 ARM9E 内核是

ARM926EJ-S、ARM946E-S、ARM966E-S 等。市场上早期的 PDA 多采用此类处理器。

3. ARM10E 处理器

ARM10E 处理器采用 ARMv5T 体系结构，特点是：六级流水处理、采用指令与数据分离的 Cache 结构、平均功耗为 1000mW/MHz、时钟频率为 300MHz、每条指令平均执行 1.2 个时钟周期等。ARM10TDMI 与所有 ARM 核在二进制级代码中兼容，内带高速 32×16 MAC，预留 DSP 协处理器接口。其中的 VFP10（向量浮点单元）为七级流水处理体系结构。ARM10E 中的 ARM1020T 处理器由 ARM10TDMI、32KB 指令、数据 Cache 及 MMU 部分构成。ARM10E 时钟频率高达 300MHz，指令 Cache 和数据 Cache 分别为 32KB，数据宽度为 64 位，支持多种商用操作系统，主要用于无线设备、数字消费品、成像设备、工业控制、通信和信息系统等领域。主流的 ARM10 内核是 ARM1020E、ARM1022E、ARM1026EJ-S 等。

4. SecurCore 处理器

SecurCore 系列处理器提供了基于高性能的 32 位 RISC 技术的安全解决方案。主要特点是：体积小、功耗低、代码密度大和性能高等。另外，最为特别的就是 SecurCore 系列处理器提供了安全解决方案的支持。SecurCore 系列处理器采用软内核技术，以提供最大限度的灵活性，以及防止外部对其进行扫描探测，提供面向智能卡的和低成本的存储保护单元 MPU，可以灵活地集成用户自己的安全特性和其他协处理器。

5. StrongARM 处理器

StrongARM 处理器采用 ARMv4T 的五级流水处理体系结构，主要有 SA110、SA1100、SA1110 这 3 个版本。另外 Intel 公司基于 ARMv5TE 体系结构的 Xscale PXA27x 系列处理器，与 StrongARM 相比增加了 I/D Cache，并且加入了部分 DSP 功能，更适合于移动多媒体应用。早期市场上的大部分智能手机的核心处理器就是 Xscale 系列处理器。

6. ARM11 处理器

ARM11 处理器采用 ARMv6 体系结构，可以在使用 130nm 代工厂技术、小至 2.2mm^2 芯片面积和低至 0.24mW/MHz 的前提下达到高达 500MHz 的性能表现。ARM11 处理器系列以众多消费产品市场为目标，推出了许多新技术，包括针对媒体处理的 SIMD、用以提高安全性能的 TrustZone 技术、智能能源管理（IEM），以及需要非常高的、可升级的超过 2600 Dhrystone 2.1 MIPS 性能的系统多处理技术。主要的 ARM11 处理器有 ARM1136JF-S、ARM1156T2F-S、ARM1176JZF-S、ARM11 MCORE 等多种。

1.1.2 Cortex 内核处理器

ARM 公司将经典处理器 ARM11 以后的产品命名为 Cortex，分成 A、R 和 M 三个系列，旨在为各种不同的市场提供服务。Cortex 系列属于 ARMv7 体系结构，这是 ARM 公司最新的指令集体系结构。ARMv7 体系结构定义了三大分工明确的系列：A 系列面向尖端的基于虚拟内存的操作系统和用户应用；R 系列针对实时系统；M 系列针对微控制器。由于应用领域不同，基于 ARMv7 体系结构的 Cortex 处理器系列所采用的技术也不相同，基于 ARMv7A 的称为 Cortex-A 系列，基

于 ARMv7R 的称为 Cortex-R 系列，基于 ARMv7M 的称为 Cortex-M 系列。

Cortex-M 系列处理器主要是针对微控制器领域开发的，该类处理器具有低功耗、良好的中断行为、卓越性能以及与现有平台的高兼容性等特点。Cortex-M 系列处理器作为微控制器主要应用于混合信号设备、智能传感器、汽车电子等场合。而 Cortex-R 系列处理器的开发则面向深层嵌入式实时应用，并对这些需求进行了平衡考虑。Cortex-R 系列处理器主要适用于汽车制动系统、动力传动、高档轿车的组件、大型发电机控制器、机器手臂控制器、联网等对实时性和可靠性要求较高的应用场合。ARM Cortex-A 系列应用型处理器可向托管丰富的操作系统平台的设备和用户应用提供全方位的解决方案，包括智能手机、移动计算平台、数字电视、机顶盒、企业网络、打印机和服务器解决方案等。

需要注意的是，ARM 体系结构版本号和处理器命名中的数字含义并不相同。比如，ARM7TDMI 是基于 ARMv4T 体系结构的（T 表示支持“Thumb 指令集”）；ARMv5TE 体系结构率先应用于 ARM9E 处理器家族。ARM9E 家族成员包括 ARM926E-S 和 ARM946E-S，该体系结构添加了“服务于多媒体应用增强的 DSP 指令”。

后来又出现了 ARM11，ARM11 是基于 ARMv6 体系结构建成的。基于 ARMv6 体系结构的处理器包括 ARM1136J (F) -S、ARM1156T2 (F) -S，以及 ARM1176JZ (F) -S。ARMv6 是 ARM 进化史上的一个重要里程碑：从那时起，引进许多突破性的新技术，存储器系统加入了很多崭新的特性，从 v6 开始首次引入单指令流多数据流（SIMD）指令。而最前卫的新技术就是经过优化的 Thumb-2 指令集，它专门针对低成本的单片机及汽车组件市场。

ARM 体系结构从最初开发到现在有了很大改进，并仍在完善和发展中。为了清楚地表达每个 ARM 应用实例所使用的指令集，ARM 公司定义了 7 种主要的 ARM 指令集体系结构版本，以版本号 v1 ~ v7 表示。

（1）ARM 版本 I：v1 体系结构

该版体系结构只在原型机 ARM1 中出现过，其只有 26 位寻址空间，没有用于商业产品。基本性能有：

- ☐ 基本的数据处理指令（无乘法）；
- ☐ 基于字节、半字和字的 Load/Store 指令；
- ☐ 转移指令，包括子程序调用及链接指令；
- ☐ 供操作系统使用的软件中断指令 SWI；
- ☐ 寻址空间：64MB。

（2）ARM 版本 II：v2 体系结构

该版体系结构对 v1 版进行了扩展，例如 ARM2 和 ARM3（v2a）体系结构。包含了对 32 位乘法指令和协处理器指令的支持。v2a 是 v2 的变种，ARM3 芯片采用了 v2a，是第一片采用片上 Cache 的 ARM 处理器。v2 同样为 26 位寻址空间，现在已经废弃不再使用。v2 体系结构与 v1 相比，增加了以下功能：

- ☐ 乘法和乘法累加指令；

- ❑ 支持协处理器操作指令；
- ❑ 快速中断模式；
- ❑ SWP/SWPB 的最基本存储器与寄存器交换指令；
- ❑ 寻址空间：64MB。

(3) ARM 版本Ⅲ：v3 体系结构

ARM 作为独立的公司，在 1990 年设计的第一个微处理器采用的是 v3 的 ARM6。ARM6 是 IP 核、独立的处理器、具有片上高速缓存、MMU 和写缓冲的集成 CPU。变种版本有 v3G 和 v3M。v3G 是不与 v2a 向前兼容的 v3，v3M 引入了有符号和无符号数乘法和乘法累加指令，这些指令产生全部 64 位结果。v3 体系结构（目前已废弃）对 ARM 体系结构做了较大的改动：

- ❑ 寻址空间增至 32 位（4GB）；
- ❑ 当前程序状态信息从原来的 R15 寄存器移到当前程序状态寄存器（Current Program Status Register, CPSR）中；
- ❑ 增加了程序状态保存寄存器（Saved Program Status Register, SPSR）；
- ❑ 增加了中止和未定义指令异常的两种处理器模式，使操作系统代码可方便地使用数据访问中止异常、指令预取中止异常和未定义指令异常；
- ❑ 增加了 MRS/MSR 指令，以访问新增的 CPSR/SPSR 寄存器；
- ❑ 增加了从异常处理返回的指令功能。

(4) ARM 版本Ⅳ：v4 体系结构

v4 体系结构在 v3 上做了进一步扩充。v4 体系结构是目前应用最广的 ARM 体系结构，ARM7、ARM8、ARM9 和 StrongARM 都采用该体系结构。v4 不再强制要求与 26 位地址空间兼容，而且明确了哪些指令会引起未定义指令异常。指令集中增加了以下功能：

- ❑ 符号化和非符号化半字及符号化字节的存/取指令；
- ❑ 增加了 T 变种，处理器可工作在 Thumb 状态，增加了 16 位 Thumb 指令集；
- ❑ 完善了软件中断 SWI 指令的功能；
- ❑ 处理器系统模式引进特权方式时使用用户寄存器操作；
- ❑ 把一些未使用的指令空间捕获为未定义指令。

(5) ARM 版本Ⅴ：v5 体系结构

v5 体系结构是在 v4 基础上增加了一些新的指令，ARM10 和 Xscale 都采用 v5 体系结构。这些新增指令有：

- ❑ 带有链接和交换的转移 BLX 指令；
- ❑ 计数前导零 CLZ 指令；
- ❑ BRK 中断指令；
- ❑ 增加了数字信号处理指令（v5TE 版）；
- ❑ 为协处理器增加更多可选择的指令；
- ❑ 改进了 ARM/Thumb 状态之间的切换效率；

- ❑ 增强型 DSP 指令集，包括全部算法操作和 16 位乘法操作；
- ❑ 支持新的 Java，提供字节代码执行的硬件和优化软件加速功能。

(6) ARM 版本 VI：v6 体系结构

v6 体系结构是 2001 年发布的，首先应用于 2002 年春季发布的 ARM11 处理器中。在降低耗电量的同时，还强化了图形处理性能。通过追加有效进行多媒体处理的 SIMD（Single Instruction Multiple Data，单指令多数据）功能，将语音及图像的处理功能提高到原型机的 4 倍。此体系结构在 v5 基础上增加了以下功能：

- ❑ THUMB2M：35% 代码压缩；
- ❑ DSP 扩充：高性能定点 DSP 功能；
- ❑ JazelleTM：Java 性能优化，可提高 8 倍；
- ❑ Media 扩充：音频/视频性能优化，可提高 4 倍。

(7) ARM 版本 VII：v7 体系结构

ARMv7 体系结构是在 ARMv6 体系结构的基础上诞生的。该体系结构采用了 Thumb-2 指令集技术——在 ARM 的 Thumb 代码压缩技术的基础上发展起来，并且保持了对现存 ARM 解决方案的完整的代码兼容性。主要新增功能如下：

- ❑ Thumb-2 指令集技术；
- ❑ NEON 技术；
- ❑ 向量浮点运算 V3；
- ❑ 支持动态编译器。

ARMv7 体系结构在设计时充分考虑了与早期 ARM 处理器软件的兼容性。ARM Cortex-M 系列支持 Thumb-2 指令集（Thumb 指令集的扩展集），对基于早期处理器编写的代码具有良好的兼容性。通过一个前向的转换方式，为 ARM Cortex-M 系列处理器所写的用户代码可以与 ARM Cortex-R 系列微处理器完全兼容。ARM Cortex-M 系列系统代码（如实时操作系统）可以很容易地移植到基于 ARM Cortex-R 系列的系统上。ARM Cortex-A 和 Cortex-R 系列处理器还支持 ARM 32 位指令集，向后完全兼容早期的 ARM 处理器。

需要注意的是，在 v7 体系结构中，Thumb-2 指令集技术比纯 32 位代码少使用 31% 的内存，减小了系统开销，同时能够提供比已有的基于 Thumb 指令集技术的解决方案高出 38% 的性能。ARMv7 体系结构还采用了 NEON 技术，将 DSP 和媒体处理能力提高了近 4 倍。ARMv7 体系结构支持改良的浮点运算，满足下一代 3D 图形、游戏物理应用以及传统嵌入式控制应用的需求。此外，ARMv7 还支持改良的运行环境，以迎合不断增加的 JIT（Just In Time）和 DAC（Dynamic Adaptive Compilation）技术。图 1.2 为 v5 ~ v7 体系结构的处理器技术比较。由于应用领域不同，基于 v7 体系结构的 Cortex 处理器系列所采用的技术也不相同。

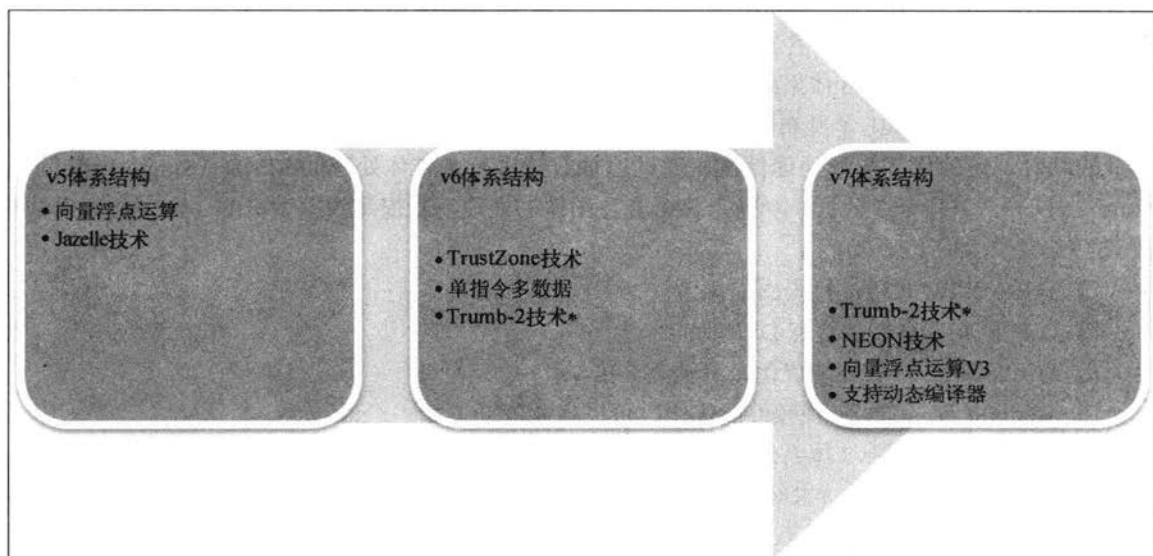


图 1.2 ARM 处理器内核技术沿革

1.2 Cortex 内核系列处理器技术特点

1.2.1 ARM Cortex-M 系列处理器

ARM Cortex-M3 是一种基于 ARMv7 体系结构的最新 ARM 嵌入式内核，它采用哈佛结构，使用分离的指令和数据总线（与冯·诺依曼结构的数据和指令共用一条总线相比，双总线架构使吞吐量得到有效提升）。除了使用哈佛结构，Cortex-M3 还具有其他显著的优点：具有更小的基础内核、价格更低、速度更快。与内核集成在一起的是一些系统外设，如中断控制器、总线矩阵、调试功能模块，而这些外设通常都是由芯片制造商增加的。Cortex-M3 还集成了睡眠模式和可选的完整八区域存储器保护单元。

Cortex-M3 旨在向专业嵌入式市场提供低成本、低功耗的芯片，主要应用于汽车和无线通信领域。与之前其他 ARM 内核一样，ARM 公司将内该设计授权给各个制造商来开发具体的芯片。迄今为止，已经有多家芯片制造商开始生产基于 Cortex-M3 内核的微控制器，如意法半导体、流明诺瑞（现已被德州仪器收购）、恩智浦等。

Cortex-M3 处理器是使用最少门数的 ARM CPU，相对于过去的产品大大减小了芯片面积，可减小装置的体积或采用更低成本的工艺进行生产，仅 33 000 门的内核性能可达 1.2 DMIPS/MHz。此外，基本系统外设还具备高度集成化特点，集成了许多紧耦合系统外设，合理利用了芯片空间，使系统满足下一代产品的控制需求。总体来讲，ARM Cortex-M3 处理器整合了多种技术，减少对内存的依赖，RISC 内核在降低功耗的基础上提供出色的性能，可实现由以往的代码向 32 位微控

制器的快速移植，是专为存储器和处理器的尺寸对产品成本影响极大的各种应用开发设计的。

在指令集上，Cortex-M3 只支持最新的 Thumb-2 指令集。这种设计的优势在于：

- ❑ 避免状态切换带来的额外开销；
- ❑ Thumb-2 指令集专门面向 C 语言设计，对硬件除法以及本地位域操作更为方便和高效；
- ❑ 允许用户在 C 代码层面维护和修改应用程序，易于重用；
- ❑ 具有调用汇编代码的功能。

综合以上这些优势，将更易于开发新产品，缩短上市时间。

Cortex-M3 处理器结合了执行 Thumb-2 指令集的 32 位哈佛体系结构和系统外设，包括 Nested Vectored Interrupt Controller 和 Arbiter 总线。该技术方案在测试和实例应用中表现出较高的性能：在台机电 180 nm 工艺下，芯片性能达 1.2 DMIPS/MHz，时钟频率高达 100 MHz。Cortex-M3 处理器还实现了末尾连锁 (Tail-Chaining) 中断技术。该技术是一项完全基于硬件的中断处理技术，最多可减少 12 个时钟周期，在实际应用中可减少 70% 中断。并且推出了新的单线调试技术，避免使用多引脚进行 JTAG 调试，并全面支持 RealView 编译器和 RealView 调试产品。RealView 工具向设计者提供模拟、创建虚拟模型、编译软件、调试、验证和测试基于 ARMv7 体系结构的系统等功，目前，以 Cortex-M4 为内核的微控制器也已面市，性能较 Cortex-M3 有较大幅度的提升，图 1.3 为 Cortex-M 系列处理器的发展沿革及性能对比。

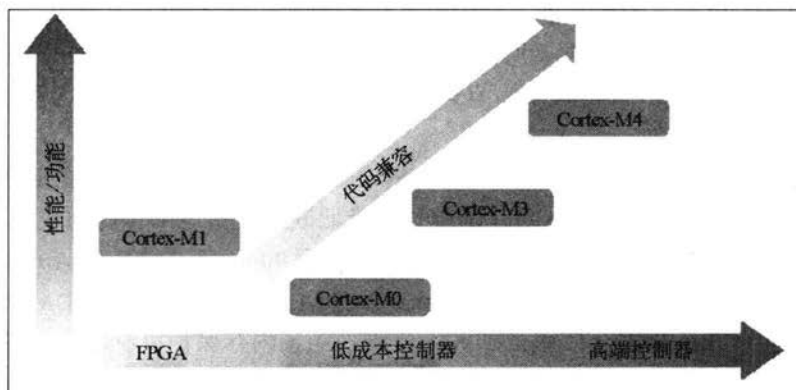


图 1.3 Cortex-M 系列处理器发展沿革

为微控制器应用而开发的 Cortex-M3 拥有以下性能：

- ❑ 实现单周期 Flash 应用最优化；
- ❑ 准确快速地中断处理，永不超过 12 个时钟周期，仅 6 个时钟周期的末尾连锁中断技术；
- ❑ 有低功耗时钟门控 (Clock Gating) 的 3 种睡眠模式；
- ❑ 单周期乘法和乘法累加指令；
- ❑ ARM Thumb-2 指令集混合的 16/32 位固有指令集，无模式转换；
- ❑ 包括数据观察点和 Flash 补丁在内的高级调试功能；
- ❑ 原子位操作，在一个单一指令中读取 / 修改 / 编写；

□ 1.25 DMIPS/MHz（与 0.9 DMIPS/MHz 的 ARM7 和 1.1 DMIPS/MHz 的 ARM9 相比）。

1.2.2 ARM Cortex-R 系列处理器

ARM Cortex-R 系列处理器目前包括 Cortex-R4 和 Cortex-R4F 两个型号，主要适用于实时系统的嵌入式处理器。

1. Cortex-R4 处理器

Cortex-R4 处理器支持手机、硬盘、打印机及汽车电子设计，能协助新一代嵌入式产品快速执行各种复杂的控制算法与实时工作的运算；可通过内存保护单元（Memory Protection Unit, MPU）、高速缓存以及紧密耦合内存（Tightly Coupled Memory, TCM）让处理器针对各种不同的嵌入式应用进行最佳化调整，且不影响基本的 ARM 指令集兼容性。这种设计能够在沿用原有程序代码的情况下，降低系统的成本与复杂度，同时其紧密耦合内存的功能也能提供更小的规格及更高效率的整合，并带来更短的响应时间。

Cortex-R4 处理器采用 ARMv7 体系结构，能够与现有的程序维持完全的回溯兼容性，支持现今全球各地的数十亿系统；并已针对 Thumb-2 指令集进行最佳化设计。此项特性带来很多的利益，其中包括：更低的时钟频率所带来的省电效益；更高的性能使各种多功能特色融入移动电话与汽车产品的设计中；更复杂的算法支持更高性能的数码影像与内置硬盘的系统。运用 Thumb-2 指令集，以及 RealView 开发套件，使芯片内部存储器的容量最多可降低 30%，大幅降低系统成本，而其速度比在 ARM9tt6E-S 处理器所使用的 Thumb 指令集高出 40%。由于存储器在芯片中的占用空间愈来愈多，因此这项设计将大幅节省芯片容量，让芯片制造商运用这款处理器开发各种 SoC（System on a Chip）器件。

相比于前几代处理器，Cortex-R4 处理器高效率的设计方案使其能以更低的时钟达到更高的性能。经过最佳化设计的 Artisan Mctro 内存则进一步降低嵌入式系统的体积与成本。Cortex-R4 处理器搭载一个具备双指令发送功能的先进的微架构，采用 90nm 工艺并搭配 Artisan Advantage 程序库的组件，底面积不到 1mm²，耗电最低 0.27mW/MHz，并能提供超过 600 DMIPS 的性能。

Cortex-R4 处理器在各种安全应用上加入了容错功能和内存保护机制，支持最新版 OSEK 实时操作系统；支持 RealView Develop 系列软件开发工具、RealView Create 系列 ESL 工具与模块，以及 Core Sight 除错与追踪技术，协助设计者迅速开发各种嵌入式系统应用。

2. Cortex-R4F 处理器

Cortex-R4F 处理器拥有针对汽车市场而开发的各项先进功能，包括自动除错功能、可相互连接的错误侦测机制，以及可选择优化的浮点运算单元（Floating-Point Unit, FPU）。ECC（Error Correcting Code）技术能监控内存存取作业，侦测并校正各种错误。当发生内存错误时，ECC 逻辑除通报错误并停止系统运作外，还会加以校正。Cortex-R4F 还拥有 Cortex-R4 处理器的各项先进功能，能够透过高效能内存保护单元、高速缓存以及紧密耦合内存，使处理器针对各种不同的应用进行最佳化调整；同时将传统处理器中的错误侦测功能延伸至整个 SoC 中，系统会不断地扫描先前侦错的资料，以提升系统的可靠度。基于对安全性能的重视，Cortex-R4F 处理器特别搭载

了高分辨率内存保护机制，能严密控制独立的软件作业。

Cortex-R4F 处理器中执行浮点运算的 FPU 提供胜过固定小数点操作数的动态范围及精准度。该 FPU 与 ARM 的其他处理器核心之间的 FPU 均维持同溯兼容性，并针对各种汽车应用常见的单精度处理作业进行优化。使用单倍精度格式，而非双倍精度的数值资料，不仅能将数据处理速度提升至 2 倍，更能维持必要的精度以提高 SoC 设计的效率。

Cortex-R4F 处理器采用一套具备双指令发送功能的先进微架构，透过 Artisan Advantage 程序库中针对 90nm 工艺的优化，达到超过 800 DMIPS 的性能水准。Level 1 内存松散的时序设计，使组件能使用高密度、低功耗的 RAM，使得在总成本中占有高比重的内存能像处理器逻辑一样拥有节省空间的优势。在 90nm 工艺下，占用空间不到 1mm^2 ，且耗电量不到 0.27 mW/MHz，可以有效地协助系统开发者降低成本与功耗。该处理器采用 ARMv7ISA，功能特点与 Cortex-R4 类似。

1.2.3 ARM Cortex-A 系列处理器

Cortex-A8 处理器是一款适用于复杂操作系统及用户应用的应用处理器。该处理器支持智能能源管理 (Intelligent Energy Manager, IEM) 技术的 Artisan Advantage 程序库以及先进的泄漏控制技术，使得 Cortex-A8 处理器实现了非凡的速度和功耗效率。在 65nm 工艺下，Cortex-A8 处理器的功耗不到 300mW，能够提供高性能和低功耗。它第一次为低费用、高容量的产品带来了台式机级别的性能。

Cortex-A8 处理器是第一款基于 ARMv7 架构的应用处理器，使用了能够带来更高性能、更低功耗和更高代码密度的 Thumb-2 指令集技术。它首次采用了强大的 NEON 信号处理扩展集，可为 H.264 和 MP3 等媒体编解码提供加速。Cortex-A8 的解决方案还包括 Jazelle-RCTJava 加速技术，对实时 (JIT) 和动态调整编译 (DAC) 提供最优化，同时减少内存占用空间高达 3 倍。该处理器配置了先进的超标量体系结构流水线，能够同时执行多条指令，并且提供超过 2.0 DMIPS/MHz 的性能。该处理器集成了一个可调尺寸的二级高速缓冲存储器，能够同高速缓存的 16KB 或者 32KB 一级高速缓冲存储器一起工作，从而达到最快的读取速度和最大的吞吐量。

Cortex-A8 处理器使用了先进的分支预测技术，并且具有专用的 NEON 整型和浮点型流水线进行媒体和信号处理。在使用小于 4mm^2 的硅片及低功耗的 65nm 工艺的情况下，Cortex-A8 处理器的运行频率将高于 600MHz (不包括 NEON 追踪技术和二级高速缓冲存储器)。在高性能的 90nm 和 65nm 工艺下，Cortex-A8 处理器运行频率最高可达 1GHz，能够满足高性能消费产品设计的需要。

这些新的 Cortex 处理器都是基于 ARMv7 架构的产品，从尺寸和性能方面来看，既有少于 33 000 个门电路的 Cortex-M 系列，也有高性能的 Cortex-A 系列。其中，ARMCortex-A 系列是针对日益增长的，能够运行包括 Linux、Windows CE 和 Symbian 操作系统在内的消费者娱乐和无线产品设计的；Cortex-R 系列针对的是需要运行实时操作系统进行控制应用的系统，包括汽车电子、网络和影像系统；Cortex-M 系列则是为那些对开发费用非常敏感同时对性能要求不断增加的嵌入式应用（如微控制器、汽车车身控制系统和各种大型家电）所设计的。随着在各种不同领域应用需求的增加，微处理器市场也在趋于多样化。为了适应市场的发展变化，基于 ARMv7 架

构的 ARM 处理器系列将不断拓展自己的应用领域。事实上, Cortex-A 系列处理器家族目前已有 Cortex-A7、Cortex-A8、Cortex-A9 和 Cortex-A15 等内核, 并且家族仍在继续扩大。

1.3 STM32 互联型嵌入式控制器

半导体制造厂商意法半导体 (ST) 是率先推出基于 Cortex-M3 的 32 位微控制器系列产品的厂家之一。STM32 系列产品所用的微处理器是 ARM 公司为要求高性能 (1.25 DMIPS/MHz)、低成本、低功耗的嵌入式应用专门设计的 Cortex-M3 内核。STM32 系列产品得益于 Cortex-M3 在体系结构上进行的改进, 所有新功能都同时具有优良的功耗水平。本节着重介绍意法半导体 STM32 互联型嵌入式控制器的特色及主要的工程应用。

STM32 互联型嵌入式控制器让设计人员可以在同时需要以太网、USB、CAN 和音频级 I2S 接口的产品设计中发挥工业标准的 32 位微处理器的优异性能。目前 STM32 下设两个产品系列: STM32FXX5 和 STM32FXX7。STM32FXX5 系列大多集成一个全速 USB 2.0 Host/Device/OTG 接口和两个具有先进过滤功能的 CAN2.0B 控制器; STM32FXX7 系列则在 STM32FXX5 系列基础上增加一个 10/100M 以太网模块 (MAC), 以完整的硬件支持 IEEE1588 精确时间协议, 使设计人员能够为实时应用开发以太网连接功能。内置专用缓存让 USB OTG、两个 CAN 控制器和以太网接口同时工作, 以满足通信应用的需求, 以及各种需要灵活的工业标准连接功能的挑战性需求。

STM32 系列产品按性能可分为两个不同的系列: “增强型” 系列和 “基本型” 系列。增强型系列时钟频率达到 72MHz, 是同类产品中性能最高的产品; 基本型系列时钟频率为 36MHz, 以 16 位产品的价格得到比 16 位产品大幅提升的性能, 是 16 位产品用户的最佳选择。两个系列都内置 32KB~128KB 的闪存, 不同的是 SRAM 的最大容量和外设接口的组合。时钟频率 72MHz 时, 从闪存执行代码, STM32 功耗仅有 36mA, 是 32 位市场上功耗最低的产品, 相当于 0.5mA/MHz。

Cortex-M3 内核主要应用于存储器和处理器的尺寸对产品成本影响极大的各种应用市场, 是针对这些市场的低成本需求而专门开发设计的微处理器内核。Cortex-M3 内核增强了芯片上集成的各种功能, 包括把中断之间延迟降到 6 个 CPU 时钟周期的嵌套向量中断控制器、允许在每一个写操作中修改单个数据位的独立位操作、分支指令预测、单周期乘法、硬件除法和高效的 Thumb-2 指令集, 这些改良技术使 Cortex-M3 内核具有优异的性能、代码密度、实时性和低功耗, 图 1.4 和图 1.5 为 STM32F10x 控制器内核及片内外设结构图和 STM32 系列内核控制器系列性能对比。

STM32 采用 2.0~3.6V 电源, 当复位电路工作时, 在待机模式下最低功耗 2 μ A, 因此非常适合由电池供电的应用设备。其他省电功能包括一个集成的实时时钟、一个专用的 32kHz 振荡器和四种功率模式, 其中实时时钟包括一个电池操作专用引脚。

在性能方面, STM32 系列产品的处理速度比同级别的基于 ARM7TDMI 的产品快 30%, 换句话说, 如果处理性能相同, STM32 系列产品功耗比同级别产品低 75%。同样, 通过使用新内核的 Thumb-2 指令集, 设计人员可以把代码容量降低 45%, 这几乎把应用软件所需内存容量减少了一半。此外, 根据 Dhrystones 和其他性能测试结果可知, STM32 的性能比最优秀的 16 位架构至少高出一倍。

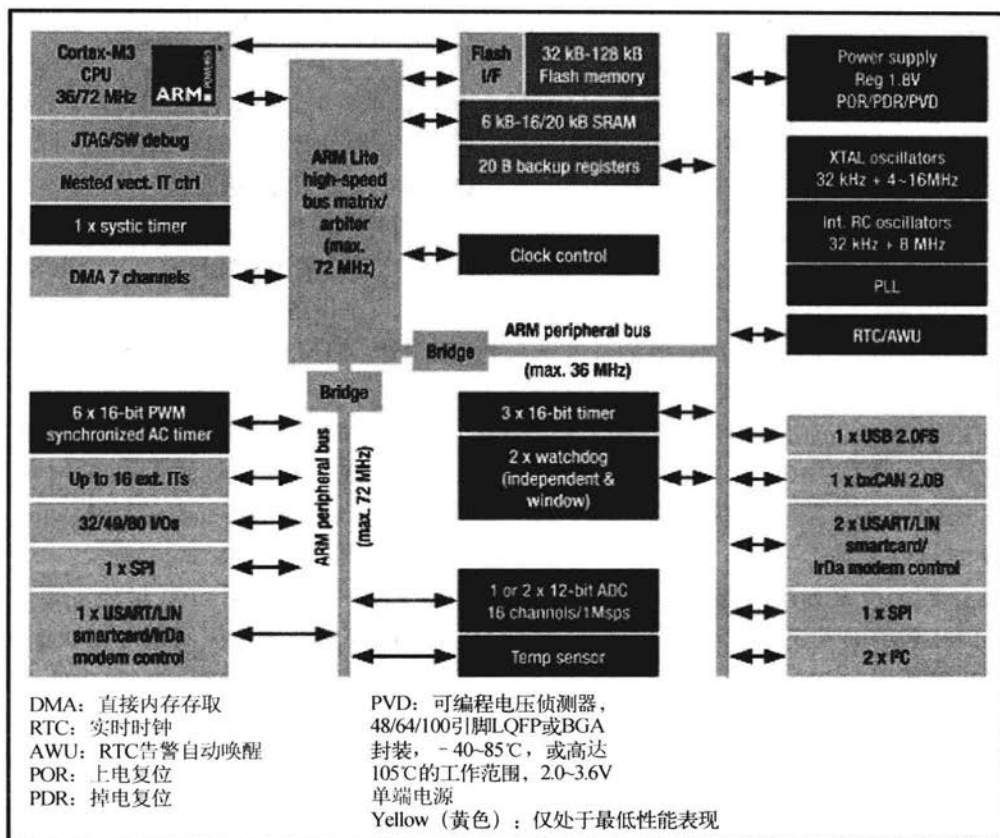


图 1.4 STM32F10x 控制器内核及片内外设结构图

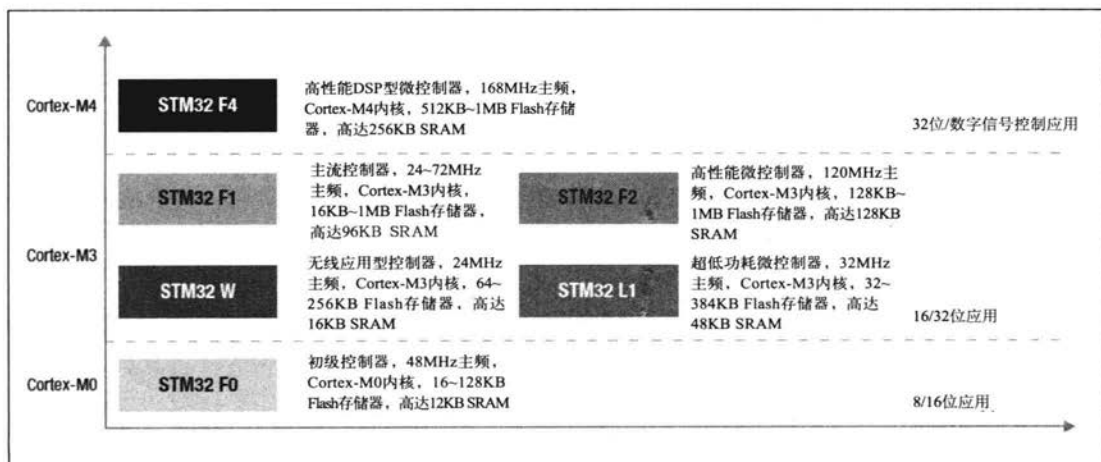


图 1.5 STM32 系列内核控制器系列性能对比

STM32 系列的新产品提供多达 128KB 的嵌入式闪存、20KB 的 RAM 和丰富的外设接口, 包括两个 12 位模数转换器 (1 μ s 的转换时间)、3 个 USART、2 个 SPI (18MHz 主/从控制器)、2 个 I2C、3 个 16 位定时器 (每个定时器有 4 个输入捕获模块、4 个输出比较器、4 个 PWM 控制器), 以及一个专门为电机控制向量驱动应用设计的内嵌死区时间控制器的 6-PWM 定时器、USB、CAN 和 7 个 DMA 通道。内置复位电路包括上电复位、掉电复位和电压监控器, 以及一个可用作主时钟的高精度工厂校准的 8MHz 阻容振荡器、一个使用外部晶振的 4~16MHz 振荡器和两个看门狗。由于集成度非常高, 除一个电源外, LQFP100 封装产品的最小系统只需要 7 个电容器。

除工业可编程逻辑控制器 (PLC)、家电、工业及家用安全设备、消防和暖气通风空调系统等传统应用, 以及智能卡和生物测定等消费电子与 PC 应用外, STM32 系列的新产品还特别适合侧重低功耗的设备, 如血糖和血脂监测设备。

STM32 系列产品配有成套的 ST 和第三方开发工具。ST 提供一个评估板、USB 开发工具包和一个免费的软件库。Hitex、IAR、Keil 和 Raisonance 不久将在经过验证的基于 ARM 内核的工具解决方案的基础上推出入门级开发工具。目前 Hitex、IAR、Keil、Raisonance 和 Rowley 的工具链均支持 STM32 系列。

STM32 的两个系列产品都采用 LQFP48、LQFP64、LQFP100 和 BGA100 封装, 均有 32KB、64KB 和 128KB 的嵌入式闪存可供选择。LQFP 封装产品的经销商定价区间是 (10 000 件) 1.80 美元 (基本型系列, 32KB 闪存, 48 引脚) ~3.60 美元 (增强型系列, 128KB 闪存, 100 引脚)。

1.4 微控制器选型

在项目的最初阶段, 首先需要解决的问题是选择适合工程需要的微控制器。通常, 选择一款适合工程需求的微控制器, 不仅需要考虑成本、主频、硬件接口, 还需要考虑是否运行操作系统、配套的开发工具、仿真器, 以及工程师对微控制器的经验和软件支持情况等。微控制器选型是否得当对项目开发的进度至关重要, 甚至关系项目的成败。

1.4.1 选型因素

(1) 价格及供货保证

芯片的价格和供货是必须考虑的因素。由于许多芯片目前处于试用阶段, 其价格和供货处于不稳定状态, 所以选型时尽量选择有量产的芯片。

(2) 主频 (核心频率)

芯片的主频很大程度上决定了系统的实时性能, 选择合适的主频既能保证产品的实时性能, 又能将系统硬件成本控制在合理的范围。

(3) 硬件接口

硬件接口通常包括片内外设和可扩展外设接口。芯片的片内外设越接近产品的需求, 产品开发相对就越简单。在片内资源无法满足工程需求时, 通常需要通过可扩展外设接口来满足某种工

程需求。常见的片内外设有以太网 MAC、I/O 接口、在线仿真接口、外部存储接口等。若硬件平台要支持功能较为齐备的操作系统，往往对 RAM 和 ROM 资源的要求就比较高。虽然芯片一般都内置 RAM 和 ROM，但其容量都很小，这就要求芯片可扩展存储器。

此外，若工程中存在浮点运算的需求，应尽量采用带有浮点运算系列的产品来减少工程开发的工作量。处理器的算法分为两大类：浮点运算和定点运算。通常定点运算处理器运行速度更快，对功率和成本都更加敏感，而浮点运算处理器则可以在较宽的动态范围内提供高精度运算。动态范围指的是数字格式所能表示的最大数与最小数的比值。而精确度指的是确定分度时的精细程度。

如今某些定点运算处理器能够在极高的时钟频率下运行，如果架构选取恰当的话，它有可能模拟出浮点运算处理器的操作，甚至某些芯片制造商提供的定点运算处理器或控制器的外围支持库能够模拟浮点运算。这种方法以牺牲浮点运算效率来保证低成本和低能耗工作。当然，对于需要进行密集浮点运算的应用来说这并不是最佳方案，但是在某种情况下也不失为一种低成本的好解决方案。

(4) 操作系统

总体而言，对于需要运行操作系统的工程应用，往往需要更高的主频及容量更大的存储空间，以满足操作系统实时性和编译生成代码空间的需求。通常，对于无需 MMU（存储器管理单元）支持的操作系统，其本身需要消耗一定的自身资源，因此工程师应根据操作系统编译后生成代码的大小选择合适存储空间的控制单元即可满足要求。而对于运行功能齐备的操作系统（如需要 MMU 和 GUI 组件），通常片内存储空间难以满足其空间需求，必须选择具有存储扩展接口的控制器，并根据实际空间需求扩展其存储空间。

操作系统的选择应考虑如下因素：

1) 支持的开发工具。某些实时操作系统（RTOS）的开发工具相对封闭，往往由该操作系统供应商提供，开发者往往需要向供应商获取编译器、调试器等，因而带来一些额外的费用；而某些操作系统的第三方工具相对广泛，可供开发者根据个人偏好及项目开发费用进行筛选。

2) 操作系统移植。操作系统的移植是一个重要的问题，是关系到整个系统能否按期完工的一个关键因素。因此，要选择那些可移植性程度高的操作系统，最好操作系统供应商能够提供一些工程实例，帮助开发者加速系统的开发进度。

3) 内存需求。一般而言，在充分考虑工程需求的基础上，操作系统需要均衡考虑不同操作系统的内存需求与扩展内存带来的额外开销。

4) 开发人员对此操作系统及其 API 的熟悉程度。

5) 硬件驱动及第三方工具或协议栈的支持，如网卡驱动、TCP/IP 协议栈、SSL 协议栈等。

6) 可剪裁性。可剪裁的操作系统往往可根据工程的实际需求进行裁剪以最大程度地减小系统资源的额外开销，常见的可裁剪操作系统有嵌入式 VxWorks、uCOS、FreeRTOS 等。

7) 操作系统的实时性能。

(5) 应用领域

常见的应用领域有航空航天、通信、计算机、工业控制、医疗系统、消费电子、汽车电子等。一般来讲，产品的功能和性能对应用领域具有约束作用，产品的功能和性能的确定将缩小选型范围。同样，应用领域的确定也将缩小选型的范围。如工业控制领域和航空航天领域的产品通常要求工作温度范围较宽，一般应选择工业级或军工级的芯片。

(6) 功耗

手持设备、PDA、平板电脑、智能手机、GPS 导航器、智能家电等消费类电子产品市场的快速增长使得对嵌入式微处理器的需求也不断提高。这些产品通常采用电池供电，系统的电池供电带来了一整套全新的应用需求。它们需要使用一个外形紧凑、节能的处理器解决方案。这种限制使得设计者必须在处理性能和功率或效率之间权衡利弊，做出折中。

低功耗的产品能减少对能源的消耗，从而减少环境污染。对个体而言，它能够减少人们的经济开销、提高用户体验的舒适度，因此低功耗也成为芯片选型时的一个重要指标。

(7) 封装

常见的微处理器芯片封装主要有 DIP、QFN、QFP、BGA 几大类型。BGA 类型的封装焊接比较麻烦，焊接成本相对较高。但 BGA 封装的芯片体积相对较小，能有效减小产品的体积。开发人员应根据需求进行折中选择。如果对产品体积无特殊要求，选型时尽量选择 DIP、QFN 或 QFP 封装。

(8) 芯片的可延续性及技术的可继承性

产品更新换代给产品的维护带来一定的额外风险。因此，开发人员在选型时要考虑芯片在未来一段时间内供货的可靠性。在芯片选型时，需要考察制造商对该系列内核芯片开发的延续性和继承性。一般而言，首选知名半导体公司，并据上述原则对其产品进行考察，做出最终选择。

(9) 仿真器

仿真器是硬件和底层软件调试时要用到的工具，是开发工作赖以进行的必要工具，开发人员需要利用仿真器安装固件，调试、验证开发程序的正确性。选择适合的仿真器，将为开发带来诸多便利。对于已有仿真器，开发人员需要留意其对所选芯片的支持情况。

(10) 技术支持

若开发人员对待选的芯片或操作系统较为陌生，则应优选技术支持较为全面的制造商所提供的芯片。优质的技术支持能帮助开发人员快速开展开发工作，避免开发工作中的探索过程和意外的风险。所以选芯片时最好选择知名的半导体公司。此外，选择成熟度较高的芯片可以从网络获得较多的共享资源，在满足工程需求的情况下，也是规避开发风险的一种方法。

(11) 开发工具

好的软件开发工具是提高开发效率的必要条件，优秀的开发工具能有效提高开发人员的工作效率。不仅如此，选择合适的软件开发工具对系统的实现会起到很好的作用。

(12) 开发周期

如果工程的开发周期较为紧迫，开发人员应尽量选择相对熟悉或技术支持相对完善的制造商所提供的芯片。

1.4.2 选型示例

1. 需求

- 1) 工程需求：工业通信网关，实现网口和 RS232 通信接口报文的互发互收，报文长度不超过 128B。
- 2) 操作系统：FreeRTOS V7.1。
- 3) 主频：60 MHz 以上。
- 4) 接口：带 DMA 控制的以太网控制器、2 个以上 RS232 串口、1 个 USB 2.0 接口、1 个 SPI 接口、2 个 CAN 总线接口。
- 5) 时钟：实时时钟或实时定时器。
- 6) 引脚封装：QFP。
- 7) 价格：不超过 80 元。

2. 选型需求分析

需求 1) 将“应用领域”定位于工业通信领域。目前市场上生产较适合于工业控制的微处理器的半导体公司有 NXP、Atmel、ST 公司。

需求 2) 所选操作系统是内存需求较小的 FreeRTOS V7.1 操作系统，其无需 MMU 的支持，所以把选型范围缩小到 ARM7 和 Cortex-M3 系列，Cortex-M3 系列采用最新架构，其性能优势较 ARM7 明显，也是未来发展的趋势，因此暂将芯片定位在采用 Cortex-M3 内核的芯片。

结合需求 3)、需求 4) 及工程应用需求，芯片必须带有 DMA 控制的以太网接口和 CAN 总线接口，而且芯片主频必须达到 60MHz 以上。在 NXP、Atmel、ST 公司生产的芯片中，具备此两种接口的有：NXP 公司的 LPC 系列、Atmel 公司的 SAM3 系列和 ST 公司的 STM32FX07 系列。

根据需求 4) ~7)，综合考虑选型因素，由于 ST 公司是业内率先推出 Cortex-M3 内核的厂家，其产品系列相对丰富，技术也相对成熟，因此在 ST 公司的互联型产品中优先选择符合需求 4) ~7) 的芯片。根据提供的选型表，可选择 STM32F107VCT6。



第 2 章

基于 STM32F107 的开发板

由于软件的开发调试难以脱离实际运行的硬件平台，因此本章以 STM32F107 开发板为硬件平台，简要介绍开发板的特点、主要板载资源和各部分外设电路原理，以及在实际设计开发过程中应注意的事项，如电磁兼容性和信号完整性设计要素。

2.1 STM32F107 开发板

本书中的所有例程均以内核为 STM32F107 的开发板为硬件平台进行介绍。STM32F107 VCT6 是意法半导体公司推出的全新 STM32 互联型（Connectivity）系列微控制器中的一款性能较强的产品。此芯片集成了各种高性能工业标准接口，且 STM32 不同型号的产品在引脚和软件上具有完美的兼容性，可以轻松适应更多的应用。STM32 新系列的标准外设包括 10 个定时器、两个 12 位 1Msample/sAD（快速交替模式下达到 2M sample/s）、两个 12 位 DA（数模转换器）、两个 I2C 接口、5 个 USART 接口和 3 个 SPI 端口和高质量数字音频接口 IIS，另外 STM32F107 拥有全速 USB（OTG）接口、两路 CAN2.0B 接口，以及以太网 10/100M MAC 模块。此芯片可以满足工业、医疗、楼宇自动化、家庭音响和家电市场等领域多种产品的需求。STM32F107 开发板的外观如图 2.1 所示。

STM32F107 开发板板载资源包括：

- ☐ 一个 10/100M 以太网接口
- ☐ 一个 USB 主从接口
- ☐ 一个 USB 驱动接口
- ☐ 外扩 RS232 接口
- ☐ 预留红外收发接口
- ☐ 预留 2.4G 射频模块接口
- ☐ 外扩 RS485 接口
- ☐ 两路 CAN 总线接口
- ☐ 一路温度传感器接口
- ☐ JTAG 调试接口
- ☐ SD 卡接口

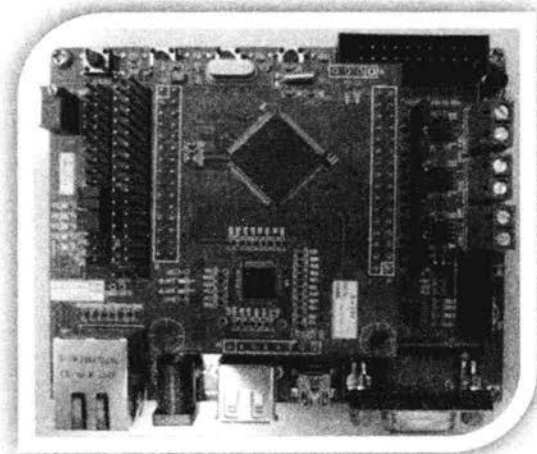


图 2.1 STM32F107 开发板外观

- ☐ 3 个自定义功能键、一个复位键
- ☐ 外扩 TFT 液晶模块接口
- ☐ 预留 IO 接口
- ☐ 电位器模拟数据采集接口
- ☐ 3 个 LED 指示灯

2.2 主要板载资源

本节介绍 STM32F107 微控制器的主要片内外设，着重介绍 10/100M 以太网接口、CAN 总线接口和 RS485 总线接口。

2.2.1 10/100M 以太网接口

1. 功能介绍

STM32F107 的以太网模块支持通过以太网收发数据，符合 IEEE 802.3-2002 标准。STM32F107 以太网模块灵活可调，能适应各种不同客户的需求。该模块支持两种标准接口连接到外接的物理层（PHY）模块：IEEE 802.3 协议定义的独立于介质的接口（MII）和简化的独立于介质的接口（RMII）。该模块适用于各类应用，如交换机、网络接口卡等。STM32F107 以太网模块符合以下标准：

- ☐ IEEE 802.3-2002 标准的以太网 MAC 协议。
- ☐ IEEE 1588-2002 的网络精确时钟同步标准。
- ☐ AMBA2.0 标准的 AHB 主/从端口。
- ☐ RMII 协会定义的 RMII 标准。

2. 主要功能

(1) MAC 控制器功能

- ☐ 通过外接的 PHY 接口，支持 10/100Mbps 的数据传输速率。
- ☐ 通过兼容 IEEE 802.3 标准的 MII 接口，外接高速以太网 PHY。
- ☐ 支持全双工和半双工操作：
 - ☐ 支持符合 CSMA/CD 协议的半双工操作。
 - ☐ 支持符合 IEEE 802.3 流控的全双工操作。
 - ☐ 在全双工模式下，可以选择性地转发接收的 PAUSE 控制帧到用户的应用程序。
 - ☐ 支持背压流控的半双工操作。
 - ☐ 在全双工模式下，当输入流控信号失效时，会自动发送 PAUSE 控制帧。
- ☐ 在发送时插入前导符和帧开始数据（SFD），在接收时去掉这些域。
- ☐ 以帧为单位，自动计算 CRC 和产生可控制的填充位。
- ☐ 在接收帧时，自动去除填充位 /CRC 为可选项。

- ☐ 可对帧长度进行编程，支持最长为 16KB 的标准帧。
- ☐ 可对帧间隙进行编程（40~96 位，以 8 位为单位改变）。
- ☐ 支持多种灵活的地址过滤模式：
 - ☐ 多达 4 个 48 位完美的目的地址（DA）过滤器，可在比较时屏蔽任意字节。
 - ☐ 多达 3 个 48 位源地址（SA）比较器，可在比较时屏蔽任意字节。
 - ☐ 64 位 Hash 过滤器（可选的），用于多播和单播（目的）地址。
 - ☐ 可选的令所有的多播地址帧通过。
 - ☐ 混杂模式，支持在进行网络监测时不过滤，允许所有的帧直接通过。
 - ☐ 允许所有接收的数据包通过，并附带其通过每个过滤器的结果报告。
- ☐ 对于发送和接收的数据包，返回独立的 32 位状态信息。
- ☐ 支持检测接收到帧的 IEEE 802.1Q VLAN 标签。
- ☐ 应用程序有独立的发送、接收和控制接口。
- ☐ 支持使用 RMON/MIB 计数器（RFC2819/RFC2665）进行强制性的网络统计。
- ☐ 使用 MDIO 接口对 PHY 进行配置和管理。
- ☐ 检测 LAN 唤醒帧和 AMD 的 Magic Packet™ 帧。
- ☐ 对 IPv4 和由以太网帧封装的 TCP 数据包的接收校验和卸载分流功能。
- ☐ 对 IPv4 报头校验以及对 IPv4 或 IPv6 数据格式封装的 TCP、UDP 或 ICMP 的校验和进行检查的高级接收功能。
- ☐ 支持由 IEEE 1588-2002 标准定义的以太网帧时间戳，在每个帧的接收或发送状态中加上 64 位的时间戳。
- ☐ 两套 FIFO：一个 2KB 的传输 FIFO，带可编程的发送阈值，以及一个 2KB 的接收 FIFO，带可编程的接收阈值（默认值是 64B）。
- ☐ 在接收 FIFO 的 EOF 后插入接收状态信息，使得多个帧可以存储在同一个接收 FIFO 中，而不需要另开辟一个 FIFO 来存储这些帧的接收状态信息。
- ☐ 可以过滤接收的错误帧，并在存储-转发模式下，不向应用程序转发错误的帧。
- ☐ 可以转发“好”的短帧给应用程序。
- ☐ 支持通过产生脉冲来统计在接收 FIFO 中丢失和破坏（由于溢出）的帧数目。
- ☐ 对于 MAC 控制器的数据传输，支持存储-转发机制。
- ☐ 根据接收 FIFO 的填充程度（阈值可编程），自动向 MAC 控制器产生 PAUSE 帧或背压信号。
- ☐ 在发送时，如遇到冲突可以自动重发。
- ☐ 在迟到冲突、冲突过多、顺延过多和欠载（underrun）情况下丢弃帧。
- ☐ 软件控制清空发送 FIFO。
- ☐ 在存储-转发模式下，在要发送的帧内，计算并插入 IPv4 的报头校验以及 TCP、UDP 或 ICMP 的校验和。
- ☐ 支持 MII 接口的内循环，可用于调试。

(2) DMA 功能

- ☐ 在 AHB 从接口下, 支持所有类型的 AHB 突发传输。
- ☐ 在 AHB 主接口下, 软件可以选择 AHB 突发传输的类型 (固定的或者不固定长度的突发)。
- ☐ 可以选择来自 AHB 主接口的地址对齐的突发传输。
- ☐ 优化的 DMA 传输, 传输以帧分隔符为界的数据帧。
- ☐ 支持以字节对齐的方式对数据缓存区寻址。
- ☐ 双缓存区 (环) 或链表形式的描述符列表。
- ☐ 优先描述符的架构, 使得大量的数据传输仅需要介入最小量的 CPU 时间。
- ☐ 每个描述符可以传输高达 8KB 的数据。
- ☐ 无论正常传输还是错误传输都有完整的状态信息报告。
- ☐ 可配置的发送与接收 DMA 突发传输长度, 优化总线使用。
- ☐ 可以设置以不同的操作条件产生对应的中断。
- ☐ 每个帧发送 / 接收完成时产生中断。
- ☐ 通过轮换或固定优先级方式仲裁 DMA 发送 / 接收控制器的优先级。
- ☐ 开始 / 停止模式。
- ☐ 状态寄存器指向当前发送 / 接收缓存区。
- ☐ 状态寄存器指向当前发送 / 接收描述符。

(3) PTP 功能

- ☐ 设置接收 / 发送帧的时间戳。
- ☐ 粗调和细调的校正方法。
- ☐ 当系统时间比目标时间大时触发中断。
- ☐ (通过 MCU 的复用功能 I/O) 输出秒脉冲。

3. 功能模块描述

以太网模块包括一个符合 IEEE 802.3 协议的 MAC 和专用的 DMA 控制器。以太网模块结构框图如图 2.2 所示。该模块支持默认的独立于介质的接口 (MII) 和简化的独立于介质的接口 (RMII), 可通过 AFIO_MAPR 寄存器的选择位来选择使用哪个接口。

DMA 控制器通过 AHB 主 / 从接口, 分别访问 MAC 控制器和存储器。AHB 主接口用于控制数据传输, AHB 从接口则用于访问控制和状态寄存器 (CSR) 区域。在 MAC 控制器发送数据前, DMA 会从系统存储区读出数据并存储到发送 FIFO 中。同样, 从总线上收到的以太网帧会存储在接收 FIFO 中, 并由 DMA 传送到系统存储区。

以太网模块还包括一个站点管理接口 (SMI), 用于与外接的 PHY 通信。一组配置寄存器则允许用户配置 MAC 和 DMA 控制器, 以实现所需要的功能。

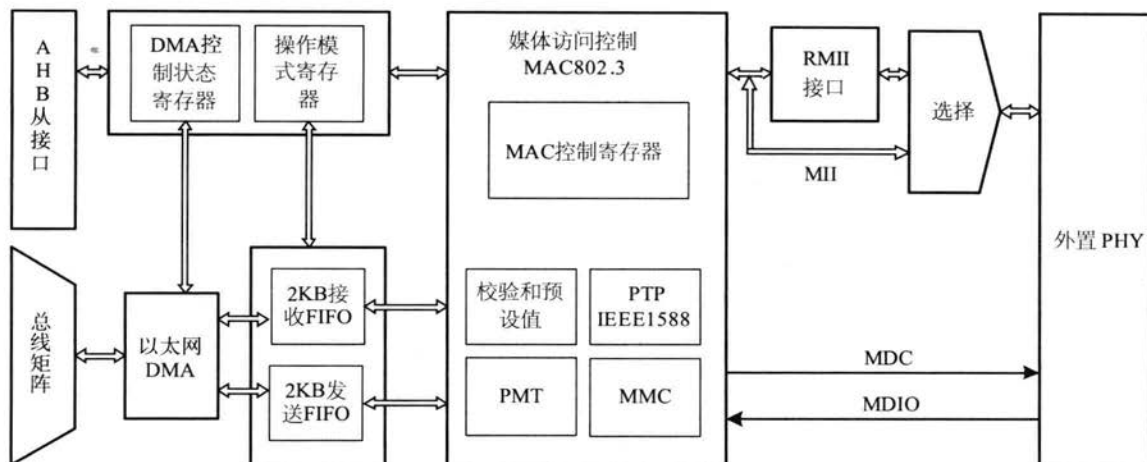


图 2.2 以太网模块结构框图

(1) 站点管理接口 (SMI)

站点管理接口 (SMI) 允许应用程序通过时钟和数据两根线来访问任何的 PHY 寄存器。这个接口可以支持多达 32 个 PHY。

应用程序可以选择 32 个 PHY 中的任意一个，并访问 PHY 的 32 个寄存器中的任意一个。但在任意时刻，只能访问一个 PHY 的一个寄存器。图 2.3 为 SMI 接口信号示意图，在控制器内部，MDC 时钟线和 MDIO 数据线都是作为复用 (AF) 功能的 I/O 端口：

- ❑ MDC：一个周期性的时钟信号，为数据的传输提供时钟，最高频率为 2.5MHz。MDC 信号的高电平和低电平的最小维持时间为 160ns，MDC 信号的最小周期为 400ns。在空闲状态下，SMI 接口将驱动 MDC 时钟信号保持在低电平状态。
- ❑ MDIO：数据的输入/输出线，在 MDC 时钟信号的驱动下，向 PHY 设备传递状态信息。

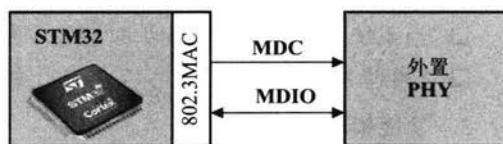


图 2.3 SMI 接口信号示意图

(2) 独立于介质的接口 (MII)

独立于介质的接口 (MII) 用于 MAC 子层和 PHY 之间的互联，允许 10Mbps 和 100Mbps 数据传输，图 2.4 为独立于介质的接口 (MII) 信号线接口示意图，图 2.5 为 MII 时钟源与外置 PHY 接口示意图。

- ❑ MII_TX_CLK：为传输发送数据而提供连续的时钟信号，对于 10Mbps 的数据传输，此时钟为 2.5MHz，对于 100Mbps 的数据传输，此时钟为 25MHz。
- ❑ MII_RX_CLK：为传输接收数据而提供连续的时钟信号，对于 10Mbps 的数据传输，此时钟为 2.5MHz，对于 100Mbps 的数据传输，此时钟为 25MHz。

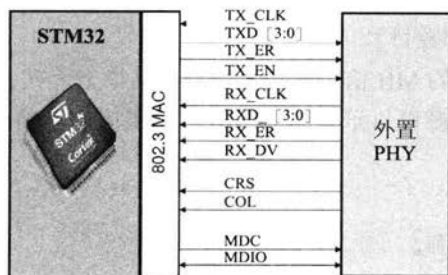


图 2.4 独立于介质的接口 (MII) 信号线接口示意图

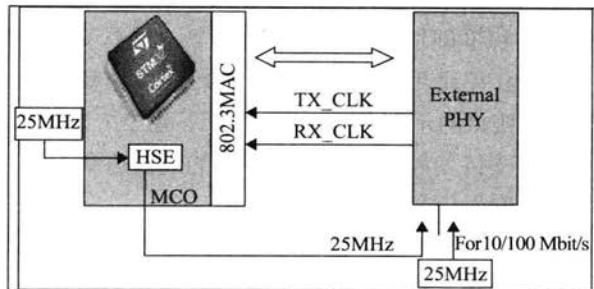


图 2.5 MII 时钟源与外置 PHY 接口示意图

- ❑ MII_TX_EN：传输使能信号，表示 MAC 正在输出要求 MII 接口传输的数据。此使能信号必需与数据前导符的起始位同步 (MII_TX_CLK) 出现，并且必需一直保持到所有需要传输的位都传输完毕为止。
- ❑ MII_TXD[3:0]：由 MAC 子层控制，每次同步传输 4 位数据，数据在 MII_TX_EN 信号有效时有效。MII_TXD[0] 是数据的最低位，MII_TXD[3] 是数据的最高位。当 MII_TX_EN 信号无效时，传输的数据对 PHY 无效。
- ❑ MII_CRD：载波侦听信号，由 PHY 控制，当发送或接收的介质非空闲时，使能此信号。当传送和接收的介质都空闲时，PHY 会撤消此信号。PHY 必需保证 MII_CS 信号在发生冲突的整个时间段内都保持有效。不需要此信号与发送 / 接收的时钟同步。在全双工模式下，此信号的状态对于 MAC 子层无意义。
- ❑ MII_COL：冲突检测信号，由 PHY 控制，当检测到介质发生冲突时，使能此信号，并且在整个冲突的持续时间内，保持此信号有效。此信号不需要与发送 / 接收的时钟同步。在全双工模式下，此信号的状态对于 MAC 子层无意义。
- ❑ MII_RXD[3:0]：由 PHY 控制，每次同步传输 4 位数据，数据在 MII_RX_DV 信号有效时有效。MII_RXD[0] 是数据的最低位，MII_RXD[3] 是数据的最高位。当 MII_RX_EN 无效，而 MII_RX_ER 有效时，PHY 会通过传送一组特殊的 MII_RXD[3:0] 数据来告知一些特殊的信息。
- ❑ MII_RX_DV：接收数据使能信号，由 PHY 控制，当 PHY 准备好卸载和解码数据供 MII 接收时，使能该信号。此信号必需和卸载好的帧数据的首位同步 (MII_RX_CLK) 出现，并在数据完全传输完毕之前，都保持有效。在传送最后 4 位数据后的第一个时钟之前，此信号必需变为无效状态。为了正确接收一个帧，MII_RX_DV 信号必需在整个帧传输期间内都保持有效，有效电平不能晚于数据线上的 SFD 位。
- ❑ MII_RX_ER：接收出错信号，保持一个或多个时钟周期 (MII_RX_CLK) 的有效状态，指示 MAC 子层在帧内检测到错误。错误情况必需配合 MII_RX_DV 的状态。

(3) 简化的独立于介质的接口 (RMII)

简化的独立于介质的接口 (RMII) 规范减少了与 10/100Mbps 通信时，STM32F107 以太网模

块和外部以太网之间的引脚数。根据 IEEE 802.3u 标准，MII 接口需要 16 个数据和控制信号引脚，而 RMII 标准则将引脚数减少到 7 个（减少了 56% 的引脚数目）。

RMII 模块用于连接 MAC 和 PHY，该模块将 MAC 的 MII 信号转换到 RMII 接口上，其与外置 PHY 接口的示意图如图 2.6 所示，图 2.7 为 RMII 时钟源接口示意图。RMII 模块具有以下特性：

- ❑ 支持 10Mbps 和 100Mbps 的通信速率。
- ❑ 时钟信号提高到 50MHz。
- ❑ MAC 和外部的以太网 PHY 需要使用同样的时钟源。
- ❑ 使用 2 位宽度的数据收发。

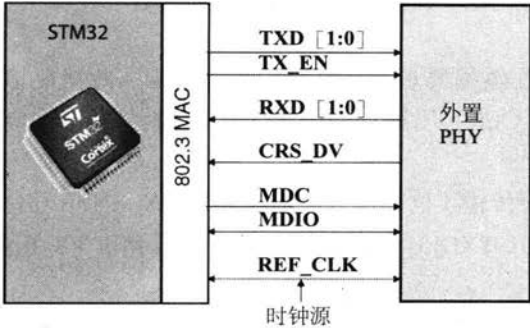


图 2.6 简化的独立于介质的接口 (RMII) 与外置 PHY 接口的示意图

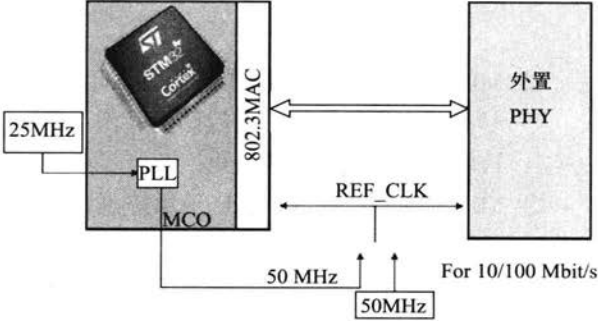


图 2.7 RMII 时钟源接口示意图

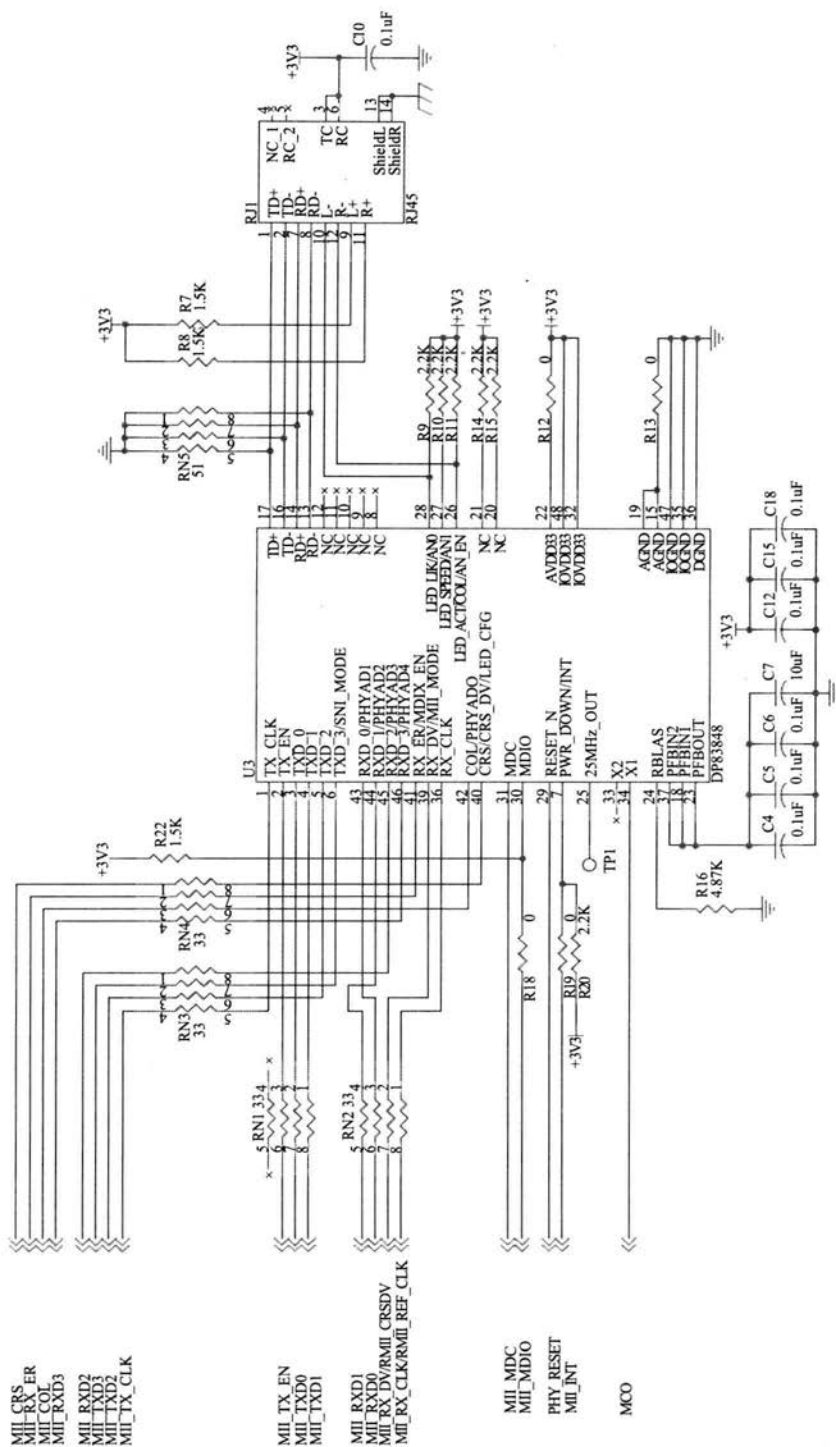
注意 在使用以太网模块时，AHB 的频率应至少为 25MHz。

4. 电路设计

图 2.8 为以太网部分的电路原理图。除了按照微控制器数据手册的要求进行原理图设计外，在设计时还需要注意的是，以太网接口部分 MII_TXD[3:0] 和 MII_RXD[0] 为高速收发信号，需要在线路中串接终端匹配电阻以实现阻抗匹配，保障传输线路上信号的完整性。

注意 阻抗匹配是指负载阻抗与激励源内部阻抗互相适配，得到最大功率输出的一种工作状态。对于不同特性的电路，匹配条件有所差异。在纯电阻电路中，当负载电阻等于激励源内阻时，则输出功率为最大，这种工作状态称为匹配，否则称为失配。当激励源内阻抗和负载阻抗含有电抗成分时，为使负载得到最大功率，负载阻抗与内阻必须满足共扼关系，即电阻成分相等，电抗成分只数值相等而符号相反，这种匹配条件称为共扼匹配。

在 MII_TXD[3:0] 和 MII_RXD[0] 数据收发总线上串接的匹配电阻应与传输线的特征阻抗近似或相等。特征阻抗并非普通意义上的电阻值，在高频电子线路中，特征阻抗有着特定的含义，读者可参阅相关文献进一步了解。此外，在电路布线时，应注意以下事项：



- ❑ 变压器与 RJ45 之间以及 PHY 层芯片与变压器之间的距离应控制在 1 英寸内为宜。当布局条件限制时，应优先保证变压器与 RJ45 之间的距离在 1 英寸内。
- ❑ 器件布局按照信号流向放置，避免路径迂回。
- ❑ 对于没有集成网络变压器的 RJ45 接口，变压器下方的地平面对要分割，分割线宽度不小于 100MIL，网口变压器放置在 GND 和 PGND 的分隔线上。
- ❑ 每对差分走线都要进行等长布线，同时注意阻抗匹配。
- ❑ 注意 PHY 层芯片的数字地和模拟地统一，数字电源和模拟电源使用磁珠进行隔离。同时要与变压器配合。注意 PHY 芯片的电源滤波，应参照芯片数据手册要求进行设计。
- ❑ 网口指示灯的电源线 3.3V 或者 2.5V 来自于电源平面，要对它们使用磁珠和电容进行退耦；指示灯驱动线要靠近 PHY 串联电阻，并在进入 I/O 区域之前进行电容滤波。这样防止噪声通过指示灯电源线耦合到差分线对区域。
- ❑ 指示灯电源线和驱动信号线要靠近走线，尽量减小环路面积。
- ❑ 对指示灯线和差分线进行必要的隔离，两者要保证距离足够远，如果有必要，可以使用 GND 平面进行隔离。
- ❑ 注意网口变压器芯片侧中心抽头对地的滤波电容要尽量靠近变压器引脚，保证引线最短，分布电感最小。
- ❑ 用于连接 GND 和 PGND 的 0 Ω 电阻或者电容要放置在地分割线上。
- ❑ PHY 芯片的模拟电源不宜大面积覆铜，应采取局部走线并串接磁珠接到变压器芯片侧的中心抽头上。
- ❑ PHY 芯片与变压器之间已经没有 VDD，将 PHY 芯片与变压器之间的平面层区域定义为 GND，这样可以切断来自 VDD 平面的噪声途径。
- ❑ 沿单板 PCB 的边缘每隔 250mil 打一个接地过孔，这些过孔排可以切断单板噪声向外辐射的途径，减小对 PGND 静地的影响。
- ❑ 单板的 PGND、GND 通过螺孔和结构相连接，保证系统地电位的统一。
- ❑ 保证电源平面和地平面之间的良好退耦（低阻），电源平面最好和地平面相邻。
- ❑ 和电源平面相邻的信号线不要超出电源平面的投影区域。
- ❑ 要保证和电源平面相邻的信号线的回流路径的完整性，否则需要改变平面的形状，使得信号线处在平面层内，回流路径的不完整会带来严重的电磁兼容性问题。
- ❑ 推荐把所有的高速信号线、I/O 线、差分线对优先靠近地平面走线，如果无法实现才以电源平面作为参考平面。
- ❑ 差分线要远离其他信号线，数字信号线或电源要远离模拟信号线或电源。
- ❑ 电源的去耦和旁路是十分重要的，它们可以为信号提供一个低阻抗通路，减小电源和地平面间的谐振。电容可以起到去耦和旁路的作用，但要保证由电容、走线、过孔、焊盘组成的环路的面积尽量小，以保证引线电感尽量小。

以上是在以太网接口原理图设计及布线时应注意的问题，关于软件设计的部分将在本书后续章节中详细叙述。

2.2.2 CAN 总线接口

控制器局域网 (Controller Area Network, CAN) 是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发, 并最终成为国际标准 (ISO11898), 是国际上应用最广泛的现场总线之一。在北美和西欧, CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线, 并且产生了以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。近年来, CAN 所具有的高可靠性和良好的错误检测能力受到重视, 广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强和振动大的工业环境。

CAN 总线属于现场总线的范畴, 它是一种有效支持分布式控制或实时控制的串行通信网络。较之目前许多 RS485 基于 R 线构建的分布式控制系统而言, 基于 CAN 总线的分布式控制系统在以下几个方面具有明显的优越性:

□ 通信实时性强

首先, CAN 控制器可以工作于多种方式下, 网络中的各节点都可根据总线访问优先权 (取决于报文标识符) 采用无损结构的逐位仲裁的方式竞争向总线发送数据, 且 CAN 协议废除了站地址编码, 而代之以对通信数据进行编码, 这可使不同的节点同时接收到相同的数据, 这些特点使得 CAN 总线构成的网络各节点之间的数据通信实时性强, 并且容易构成冗余结构, 提高系统的可靠性和系统的灵活性。而利用 RS485 只能构成主从式结构系统, 通信方式也只能以主站轮询的方式进行, 系统的实时性、可靠性较差。

□ 开发周期短

CAN 总线通过 CAN 收发器接口芯片 82C250 的两个输出端 CANH 和 CANL 与物理总线相连, 而 CANH 端的状态只能是高电平或悬浮状态, CANL 端只能是低电平或悬浮状态。这就保证不会出现 RS485 网络中的现象, 即当系统有错误, 出现多节点同时向总线发送数据时, 导致总线呈现短路, 从而损坏某些节点的现象。而且 CAN 节点在错误严重的情况下具有自动关闭输出功能, 以使总线上其他节点的操作不受影响, 从而保证不会出现在网络中, 因个别节点出现问题, 而使得总线处于“死锁”的状态。而且, CAN 完善的通信协议可由 CAN 控制器芯片及其接口芯片来实现, 从而大大降低系统开发难度, 缩短开发周期, 这些是仅有电气协议的 RS485 所无法比拟的。

□ 国际标准化

另外, 与其他现场总线相比较而言, CAN 总线是集通信速率高、容易实现且性价比高等诸多特点于一体的一种已形成国际标准的现场总线。这些也是目前 CAN 总线应用于众多领域, 具有强劲的市场竞争力的重要原因。

□ 应用广泛

CAN 属于工业现场总线的范畴。与一般的通信总线相比, CAN 总线的数据通信具有突出的可靠性、实时性和灵活性。由于其良好的性能及独特的设计, CAN 总线越来越受到人们的重视。它在汽车领域的应用是最广泛的, 世界上一些著名的汽车制造厂商, 如 BENZ (奔驰)、BMW (宝马)、PORSCHE (保时捷)、ROLLS-ROYCE (劳斯莱斯) 和 JAGUAR (美洲豹) 等都采用了 CAN 总线来实现汽车内部控制系统与各检测和执行机构间的数据通信。同时, 由于 CAN 总线本

身的特点，其应用范围目前已不再局限于汽车行业，而向自动控制、航空航天、航海、过程工业、机械工业、纺织机械、农用机械、机器人、数控机床、医疗器械及传感器等领域发展。CAN 已经形成国际标准，并已被公认为几种最有前途的现场总线之一。其典型的应用协议有：SAE J1939/ISO11783、CANOpen、CANaerospace、DeviceNet、NMEA2000 等。接下来简要介绍关于 CAN 总线的功能及工作模式。

1. 功能介绍

STM32 支持 CAN 协议 2.0A 和 2.0B。它的设计目标是，以最小的 CPU 负荷来高效处理收到的大量报文。它也支持报文发送的优先级要求（优先级特性可软件配置）。对于注重安全的应用，bxCAN 提供所有支持时间触发通信模式所需的硬件功能。

STM32 主要特点如下：

- ☐ 支持 CAN 协议 2.0A 和 2.0B 主动模式。

- ☐ 波特率最高可达 1Mbps。

- ☐ 支持时间触发通信功能。

(1) 发送

- ☐ 3 个发送邮箱。

- ☐ 发送报文的优先级特性可软件配置。

- ☐ 记录发送 SOF 时刻的时间戳。

(2) 接收

- ☐ 2 个 3 级深度的接收 FIFO。

- ☐ 可变的过滤器组：

- ☐ 在互联型产品中，CAN1 和 CAN2 分享 28 个过滤器组。

- ☐ 其他 STM32F103 系列产品中有 14 个过滤器组。

- ☐ 标识符列表。

- ☐ FIFO 溢出处理方式可配置。

- ☐ 记录接收 SOF 时刻的时间戳。

(3) 时间触发通信模式

- ☐ 禁止自动重传模式。

- ☐ 16 位自由运行定时器。

- ☐ 可在最后 2 个数据字节发送时间戳。

(4) 管理

- ☐ 中断可屏蔽。

- ☐ 邮箱占用单独 1 块地址空间，便于提升软件执行效率。

(5) 双 CAN

- ☐ CAN1：是主 bxCAN，负责管理 bxCAN 和 512B 的 SRAM 存储器之间的通信。

- ☐ CAN2：是从 bxCAN，不能直接访问 SRAM 存储器。

- ☐ 两个 bxCAN 模块共享 512B 的 SRAM 存储器。

注意 在中容量和大容量产品中, USB 和 CAN 共用一个专用的 512B 的 SRAM 存储器用于数据的发送和接收, 因此不能同时使用 USB 和 CAN (USB 和 CAN 模块互斥地访问共享的 SRAM 存储器)。USB 和 CAN 可以同时用于一个应用中但不能在同一个时间使用。

2. 主要模式

bxCAN 有 3 个主要的工作模式: 初始化模式、正常模式和睡眠模式。在硬件复位后, bxCAN 工作在睡眠模式以节省电能, 同时激活 CANTX 引脚的内部上拉电阻。软件通过对 CAN_MCR 寄存器的 INRQ 或 SLEEP 位置 1, 可以请求 bxCAN 进入初始化或睡眠模式。一旦进入了初始化或睡眠模式, bxCAN 就对 CAN_MSR 寄存器的 INAK 或 SLAK 位置 1 来进行确认, 同时禁用内部上拉电阻。当 INAK 和 SLAK 位都为 0 时, bxCAN 就处于正常模式。在进入正常模式前, bxCAN 必须跟 CAN 总线取得同步; 为取得同步, bxCAN 要等待 CAN 总线达到空闲状态, 即在 CANRX 引脚上监测到 11 个连续的隐性位。

除此之外, bxCAN 还有测试模式, 其中又包括静默模式、环回模式和环回静默模式。

在静默模式下, bxCAN 可以正常地接收数据帧和远程帧, 但只能发出隐性位, 而不能真正发送报文。编程人员可通过对 CAN_BTR 寄存器的 SILM 位置 1 来选择静默模式。静默模式通常用于分析 CAN 总线的活动, 而不会对总线造成影响——显性位 (确认位、错误帧) 不会真正发送到总线上。

在环回模式下, bxCAN 把发送的报文当做接收的报文并保存 (如果可以通过接收过滤) 在接收邮箱里。通过对 CAN_BTR 寄存器的 LBKM 位置 1 来选择环回模式。为了避免外部的影响, 在环回模式下 CAN 内核忽略确认错误。环回模式通常用于自测试目的。

环回静默模式可用于“热自测试”, 既可以像环回模式那样测试 bxCAN, 又不会影响 CANTX 和 CANRX 所连接的整个 CAN 系统。在环回静默模式下, CANRX 引脚与 CAN 总线断开, 同时 CANTX 引脚被驱动到隐性位状态。通过对 CAN_BTR 寄存器的 LBKM 和 SILM 位同时置 1, 可以选择环回静默模式。

3. 功能描述

(1) 发送处理

发送报文的流程为 (见图 2.9):

1) 应用程序选择 1 个空置的发送邮箱; 设置标识符、数据长度和待发送数据; 然后对 CAN_TxR 寄存器的 TXRQ 位置 1 来请求发送。

2) TXRQ 位置 1 后, 邮箱不再是空邮箱; 而一旦邮箱不再为空, 软件不再对邮箱寄存器有写权限。TXRQ 位置 1 后, 邮箱马上进入挂号状态, 并等待成为最高优先级的邮箱。

3) 一旦邮箱成为最高优先级的邮箱, 其状态变为预定发送状态。一旦 CAN 总线进入空闲状态, 预定发送邮箱中的报文就马上被发送 (进入发送状态)。

4) 一旦邮箱中的报文成功发送后, 邮箱马上变为空置邮箱; 硬件相应地对 CAN_TSR 寄存器的 RQCP 和 TXOK 位置 1, 来表明一次成功发送。

5) 如果发送失败, 由仲裁引起的就对 CAN_TSR 寄存器的 ALST 位置 1, 由发送错误引起的就对 TERR 位置 1。

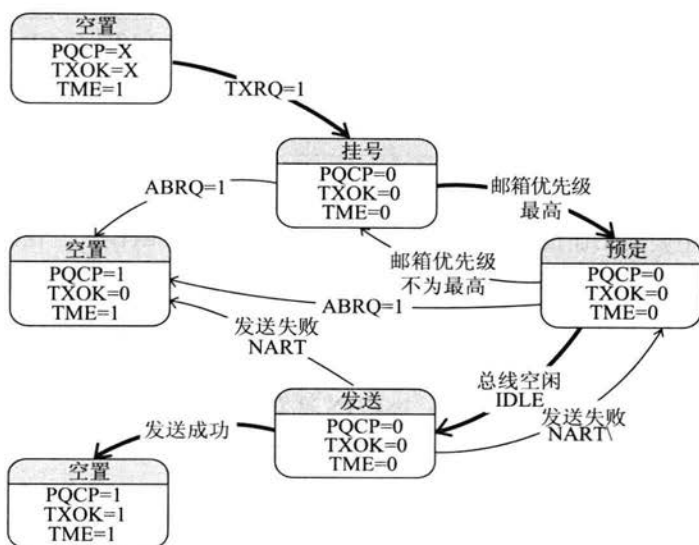


图 2.9 发送报文的流程示意图

注意事项 1：报文能否立即发送取决于发送的优先级与发送模式（见图 2.10）。

- ❑ 当有超过 1 个发送邮箱在挂号时，发送顺序由邮箱中报文的标识符决定。根据 CAN 协议，标识符数值最低的报文具有最高的优先级。如果标识符的值相等，那么邮箱号小的报文先被发送。
- ❑ 通过对 CAN_MCR 寄存器的 TXFP 位置 1，可以把发送邮箱配置为发送 FIFO。在该模式下，发送的优先级由发送请求次序决定。

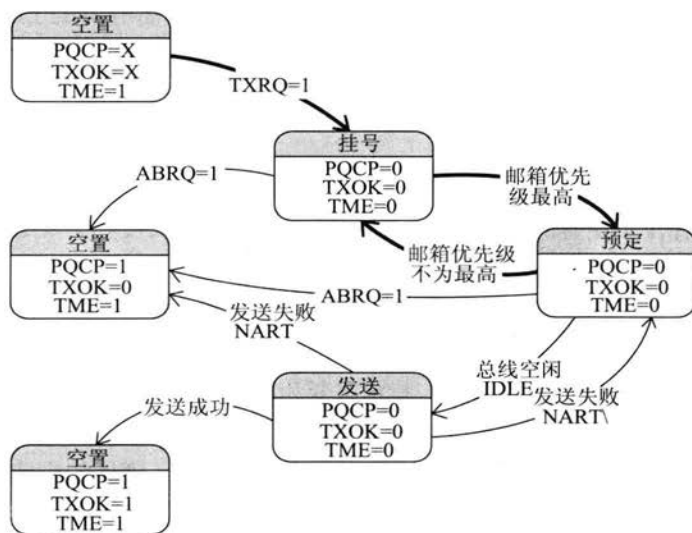


图 2.10 邮箱优先级决定是否优先发送报文

注意事项 2：发送中止（见图 2.11）及禁止自动重传（见图 2.12 和图 2.13）。

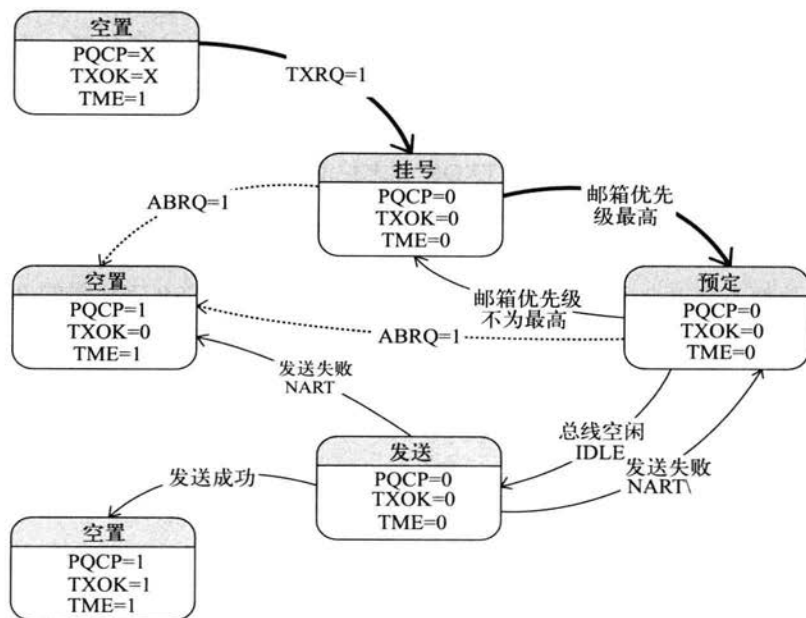


图 2.11 报文发送在“挂号”或“预定”状态下中止

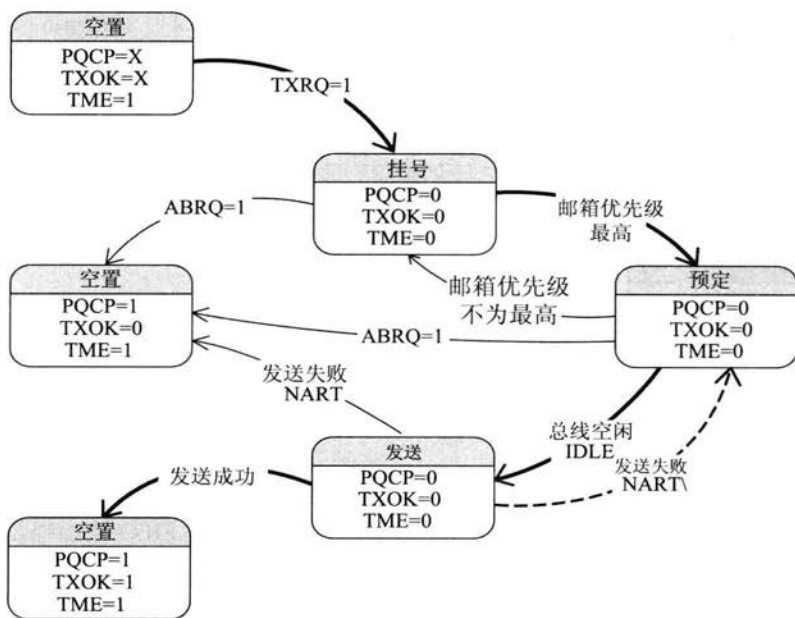


图 2.12 自动重传模式下的报文发送流程

- 发送中止：通过对 CAN_TSR 寄存器的 ABRQ 位置 1，可以中止发送请求。如果邮箱处于挂号或预定状态，发送请求立即被中止。如果邮箱处于发送状态，那么中止请求可能导致两种结果。如果邮箱中的报文已成功发送，那么邮箱变为空置邮箱，并且 CAN_TSR 寄存器的 TXOK 位被硬件置 1；如果邮箱中的报文发送失败，那么邮箱变为预定状态，然后发送请求被中止，邮箱变为空置邮箱且 TXOK 位被硬件清零。二者最终都将邮箱清空，区别在于对 CAN_TSR 寄存器的 TXOK 标志位的影响。
- 禁止自动重传：通过对 CAN_MCR 寄存器的 NART 位置 1，来让硬件工作在禁止自动重传模式。在该模式下，发送操作只会执行一次。无论是由于仲裁丢失或出错导致的发送失败，硬件都不会再自动发送该报文。并在完成后将 CAN_TSR 寄存器的 RQCP 位置 1，同时发送的结果反映在 TXOK、ALST 和 TERR 位上。该模式主要用于满足 CAN 标准中时间触发通信选项的需求。

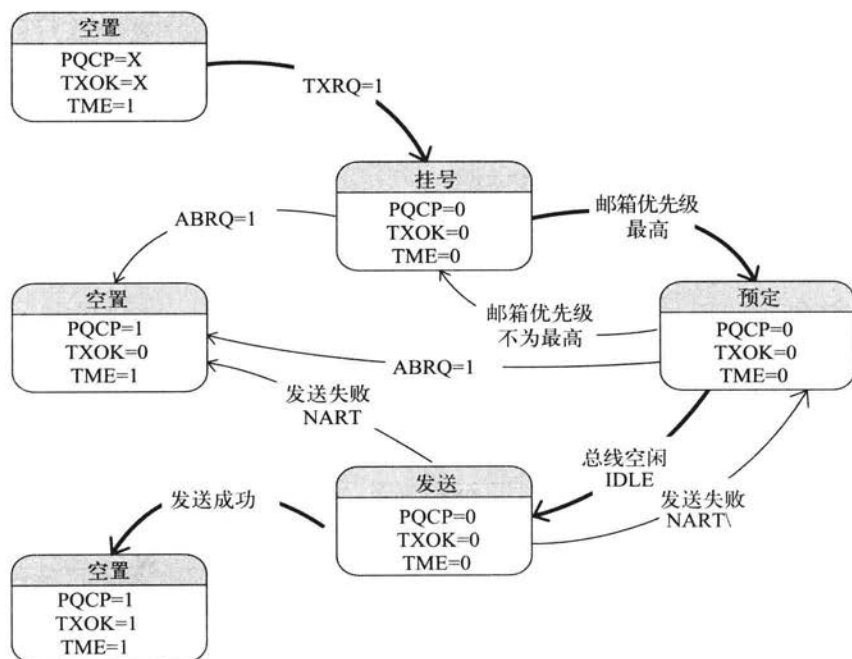


图 2.13 禁止自动重传模式下的报文发送流程

(2) 接收管理

接收的报文存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证数据的一致性。应用程序只能通过读取 FIFO 输出邮箱来读取 FIFO 中最先收到的报文，如图 2.14 所示。

根据 CAN 协议，当报文被正确接收（直到 EOF 域的最后一位都没有错误），且通过了标识符过滤，那么认为该报文是有效报文。

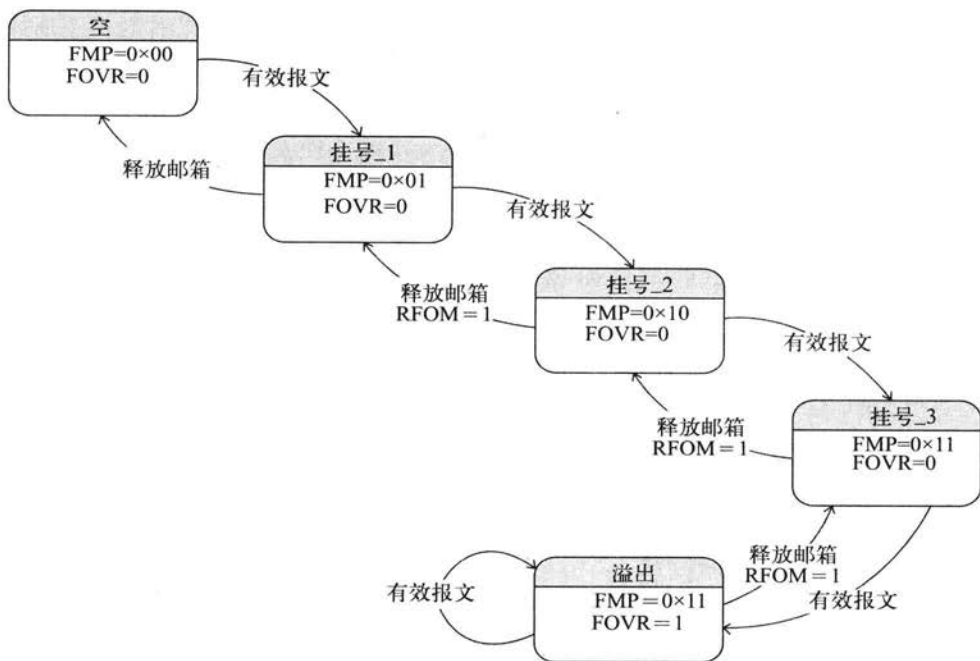


图 2.14 接收 FIFO 状态

1) FIFO 从空状态开始, 在接收到第一个有效的报文后, FIFO 状态变为挂号_1 (pending_1), 硬件相应地把 CAN_RFR 寄存器的 FMP[1:0] 设置为 01 (二进制 01b)。软件可以读取 FIFO 输出邮箱来读出邮箱中的报文, 然后通过对 CAN_RFR 寄存器的 RFOM 位设置 1 来释放邮箱, 将 FIFO 清空。如果在释放邮箱的同时又收到了一个有效的报文, 那么 FIFO 仍然保留在挂号_1 状态, 软件可以读取 FIFO 输出邮箱来读取新收到的报文。

2) 如果应用程序不释放邮箱, 在接收到第二个有效的报文后, FIFO 状态变为挂号_2 (pending_2), 硬件相应地把 FMP[1:0] 设置为 10 (二进制 10b)。

3) 重复上面的过程, 第三个有效的报文把 FIFO 变为挂号_3 状态 (pending_3, 此时 FMP[1:0] = 11b)。此时, 软件必须对 RFOM 位设置 1 来释放邮箱, 以便 FIFO 可以有空间存储下一个有效的报文; 否则, 当下一个有效的报文到来时就会导致一个报文的丢失。

4) 当 FIFO 处于挂号_3 状态 (即 FIFO 的 3 个邮箱都是满的), 下一个有效的报文就会导致溢出并且丢失。此时, 硬件对 CAN_RFR 寄存器的 FOVR 位进行置 1 来表明溢出情况。至于会丢弃哪个报文, 取决于对 FIFO 的设置:

- ❑ 如果禁用了 FIFO 锁定功能 (CAN_MCR 寄存器的 RFLM 位被清零), 那么 FIFO 中最后收到的报文就被新报文所覆盖。这样, 最新收到的报文不会丢失。
- ❑ 如果启用了 FIFO 锁定功能 (CAN_MCR 寄存器的 RFLM 位被置 1), 那么新收到的报文就被丢弃, 软件可以读到 FIFO 中最早收到的 3 个报文。

有关 CAN 模块的标识符、出错管理、位时间特性,读者可自行参考芯片数据手册或编程手册,这里不再赘述。

(3) CAN 中断

bxCAN 占用 4 个专用的中断向量。通过设置 CAN 中断允许寄存器 (CAN_IER), 每个中断源都可以单独允许和禁用。

□ 发送中断可由下列事件产生:

- 发送邮箱 0 变为空, CAN_TSR 寄存器的 RQCP0 位被置 1。
- 发送邮箱 1 变为空, CAN_TSR 寄存器的 RQCP1 位被置 1。
- 发送邮箱 2 变为空, CAN_TSR 寄存器的 RQCP2 位被置 1。

□ FIFO0 中断可由下列事件产生:

- FIFO0 接收到一个新报文, CAN_RF0R 寄存器的 FMP0 位不再是 00。
- FIFO0 变为满时, CAN_RF0R 寄存器的 FULL0 位被置 1。
- FIFO0 发生溢出时, CAN_RF0R 寄存器的 FOVR0 位被置 1。

□ FIFO1 中断可由下列事件产生:

- FIFO1 接收到一个新报文, CAN_RF1R 寄存器的 FMP1 位不再是 1。
- FIFO1 变为满时, CAN_RF1R 寄存器的 FULL1 位被置 1。
- FIFO1 发生溢出时, CAN_RF1R 寄存器的 FOVR1 位被置 1。

□ 错误和状态变化中断可由下列事件产生:

- 出错情况, 关于出错情况的详细信息请参考 CAN 错误状态寄存器 (CAN_ESR)。
- 唤醒情况, 在 CAN 接收引脚上监视到帧起始位 (SOF)。
- CAN 进入睡眠模式。

4. 电路设计

在 CAN 总线中, CAN_H 和 CAN_L 是一对差分信号 (Differential Signal)。所谓差分信号, 就是指驱动端发送两个等值、反相的信号, 接收端通过比较这两个电压的差值来判断逻辑状态“0”或“1”, 而承载差分信号的那一对走线就称为差分对, 与之相对的是单端信号。

差分信号和普通的单端信号走线相比, 具有抗干扰能力强、电磁辐射小、时序定位精确等特点。当差分走线布线良好, 差分对的耦合能够有效抑制外界的共模噪声干扰。同样的道理, 由于两根信号的极性相反, 它们对外辐射的电磁场可以相互抵消, 需要注意的是, 对噪声的抑制程度及对外辐射的程度也依赖于线路板布线的优良程度。

由于差分信号的开关变化位于两个信号的交点, 与普通单端信号依靠高低两个阈值电压判断不同, 因而受工艺、温度的影响小, 能降低时序上的误差, 同时也更适合于低幅度信号的电路。因此, 目前高速板级总线大多采用小振幅差分信号技术。

为保证 CAN 总线的稳定性和可靠性, 在线路板布线时应遵循差分布线的原则, 与上一节介绍的网络接口的布线原则类似, CAN 总线的电路原理图如图 2.15 所示。

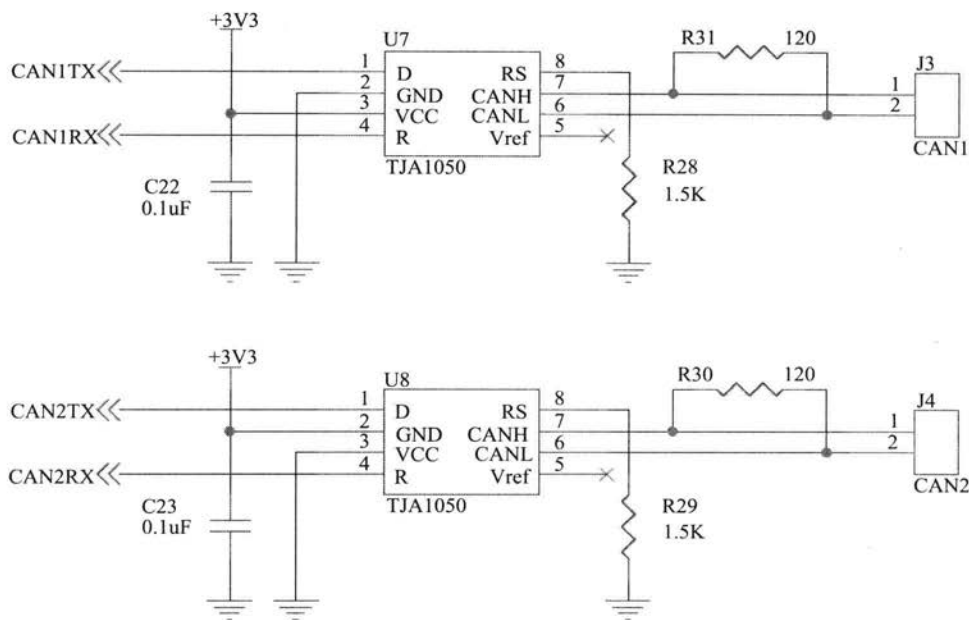


图 2.15 CAN 总线原理图

2.2.3 RS485 总线接口

RS485 标准是由两个行业协会共同制定和发布的,即美国电子工业协会(EIA)和美国通讯工业协会(TIA)。EIA 曾经在其所有的标准前面加上 RS (Recommended Standard),因此许多工程师一直沿用这种名称。

RS485 总线作为一种多点差分数据传输的电气规范,已成为业界应用最为广泛的标准通信接口之一。这种通信接口允许在简单的一对双绞线上进行多点双向通信,它所具有的噪声抑制能力、数据传输速率、电缆长度及可靠性是其他标准无法比拟的。正因为如此,许多不同领域都采用 RS485 作为数据传输线路。汽车电子、电信设备局域网、智能楼宇等中都经常可以见到具有 RS485 接口电路的设备。这项标准得到广泛接受的另外一个原因是它的通用性。RS485 标准只对接口的电气特性做出规定,而不涉及接插件、电缆及协议,在此基础上用户可以建立自己的高层通信协议,如 MODBUS 协议。

RS485 总线虽然得到了广泛应用,但也存在诸多缺点,诸如接纳设备容量不高、通信速率低、功耗较大、仅支持串行布线、稳定性较差、故障定位难度大、户外易击穿损坏等。

1. 电路设计

事实上,大多数 RS485 总线的应用都是以微控制器外扩 RS485 协议芯片的方式实现的。从设计本质上讲,RS485 总线接口的原理设计并不困难,难点在于如何保障总线的电磁兼容性、稳定

性和低功耗。图 2.16 是以 STM32 的 UART 串行接口外扩的 RS485 总线接口，RS485 总线接口芯片为常用的 SP3485。这里列举的是以低成本方式实现 RS485 总线接口，在实际应用中，应充分考虑隔离、保护等措施，保障在工程应用中的稳定性。

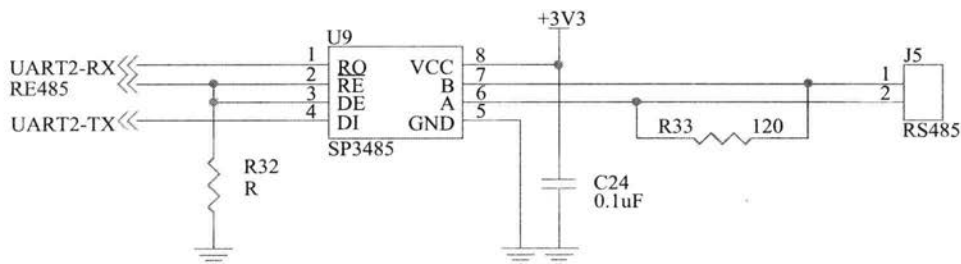


图 2.16 RS485 总线接口原理图

在实际应用中，要注意后面讨论的几个方面。

2. 抗干扰及保护措施

(1) 抗扰措施

- ❑ 共模干扰问题：RS485 收发器共模电压范围为 $-7 \sim +12\text{V}$ 。当网络线路中共模电压超出此范围时就会影响通信的稳定性与可靠性，甚至可能损坏接口。
- ❑ EMI 电磁干扰问题：当发送驱动器输出信号中的共模部分没有低阻的返回通道（信号地）时，共模信号会以辐射的形式返回源端，总线形成一个巨型天线向外辐射电磁波。因此必须保障整个 RS485 网络的各个接口有一条低阻的信号返回通道，避免形成天线效应。

(2) 保护措施

- ❑ 隔离保护方法：采用高频变压器、光耦等元件实现接口的电气隔离。将瞬态高压转移到隔离接口中的电隔离层上，从而不会产生损害性的浪涌电流，起到保护接口的作用。
- ❑ 旁路保护方法：利用瞬态抑制元件（如 TVS、MOV、气体放电管等）将危害性的瞬态能量旁路到大地。
- ❑ RS485 总线上每个通信节点上采取保护措施，如在每个节点的 A、B 线上串联一个 10Ω 的隔离电阻，可以防止某个节点损坏后影响整条线路的通信功能。

3. 安装注意事项

- ❑ 采用一条双绞线电缆作总线，将各个节点串接起来，从总线到每个节点的引出线长度应尽量短，以便使引出线中的反射信号对总线信号的影响最低。
- ❑ 注意总线特性阻抗的连续性，在阻抗不连续点就会发生信号的反射。易产生这种不连续性的情况有：总线的不同区段采用了不同电缆、某一段总线上有过多收发器紧靠在一起安装，或者是过长的分支线引出到总线。
- ❑ 当 RS485 总线空闲或开路时，可能导致接收器误触发。因此接收器一端应加偏置电阻，

将总线设定在一个确定的状态。

- 当采用 RS485 总线进行长距离通信时，由于阻抗不匹配会引起信号反射，因此必须在电缆的末端跨接一个与电缆的特性阻抗同样大小的终端电阻（通常为 120Ω ），使电缆的阻抗连续。

4. 总线节能

- 减小每帧数据发送量。
- 低功耗设计：收发器处于空闲模式时关闭发送驱动器，以减小功率消耗。
- 选择具有失效保护功能的低功耗器件（不需加偏置电阻）。
- 通信距离短、通信速率不高的场合不需加终端电阻。
- 网络终端采用 RC 阻容匹配或肖特基二极管方式代替终端电阻可有效减小电流消耗。

2.2.4 其他总线接口

由于除上述几种接口以外的其他接口电路较为简单，且大多数读者较为熟悉，因此不做太多介绍，接下来仅简单介绍电路原理图中涉及的要点。

1. USB 主从接口

USB 模块为 PC 主机和微控制器所实现的功能之间提供了符合 USB 规范的通信连接。PC 主机和微控制器之间的数据传输是通过共享一个专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512B 缓冲区，最多可用于 16 个单向或 8 个双向端点。当 USB 模块同 PC 主机通信时，根据 USB 规范可实现令牌分组的检测、数据发送/接收的处理以及握手分组的处理。整个传输的格式由硬件完成，其中包括 CRC 的生成和校验。USB 接口的电路原理图如图 2.17 所示。

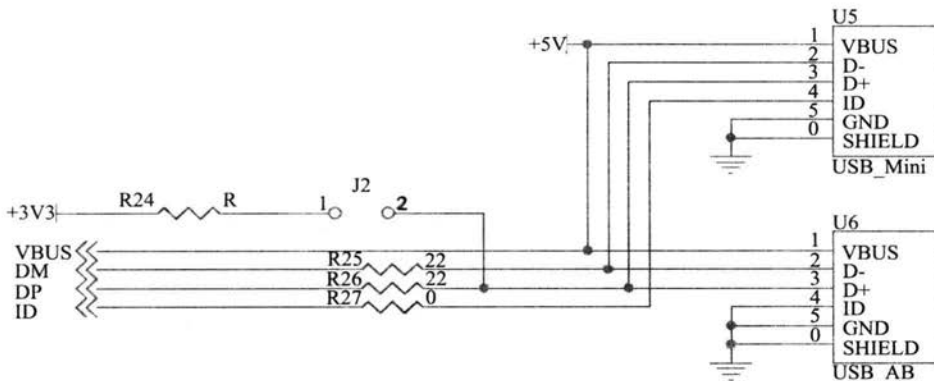


图 2.17 USB 接口电路

每个端点都有一个缓冲区描述块，用于描述该端点使用的缓冲区地址、大小和需要传输的字节数。

当 USB 模块识别出一个有效的功能 / 端点的令牌分组时，（如果需要传输数据并且端点已配置）随之发生相关的数据传输。USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后，如果需要，则根据传输的方向，发送或接收适当的握手分组。

在数据传输结束时，USB 模块将触发与端点相关的中断，通过读状态寄存器和 / 或者利用不同的中断处理程序，微控制器可以确定：

- ❑ 哪个端点需要得到服务。
- ❑ 产生如位填充、格式、CRC、协议、缺失 ACK、缓冲区溢出 / 缓冲区未满足等错误时，正在进行的是哪种类型的传输。

USB 模块对同步传输和高吞吐量的批量传输提供了特殊的双缓冲区机制，在微控制器使用一个缓冲区的时候，该机制保证了 USB 外设总是可以使用另一个缓冲区。

在任何不需要使用 USB 模块的时候，通过写控制寄存器可以使 USB 模块置于低功耗模式（SUSPEND 模式）。在这种模式下，不产生任何静态电流消耗，同时 USB 时钟也会减慢或停止。通过对 USB 线上数据传输的检测，可以在低功耗模式下唤醒 USB 模块。也可以将一特定的中断输入源直接连接到唤醒引脚上，以使系统能立即恢复正常的时钟系统，并支持直接启动或停止时钟系统。

2. 电源

开发板中的器件供电电平有两种 5V 和 3.3V，因此 3.3V 电源部分采用 AMS1117-3.3 为芯片的供电系统，5V 电源则直接由外接 5V 电源提供能源补给。板级电源的电路示意图如图 2.18 所示。

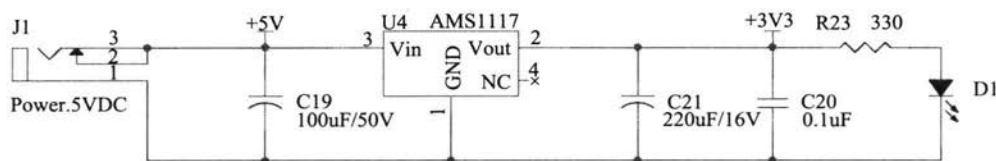


图 2.18 板级电源

3. RS232 接口

RS232 接口是 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通信的标准。RS232 的全名是“数据终端设备（DTE）和数据通信设备（DCE）之间串行二进制数据交换接口技术标准”。该标准规定采用一个 25 引脚的 DB25 连接器，对连接器的每个引脚的信号内容加以规定，还对各种信号的电平加以规定。随着设备的不断改进，出现了代替 DB25 的 DB9 接口，目前市面上的产品大多以 DB9 的形式出现，仅有少数设备保留了 DB25 的接口形式。与 RS485 类似，RS232 接口是通过在 UART 接口外部扩展的一

种物理总线，UART 提供了符合 RS232 的数据终端接口。RS232 的电路原理图如图 2.19 所示，接口芯片采用 ST3232。

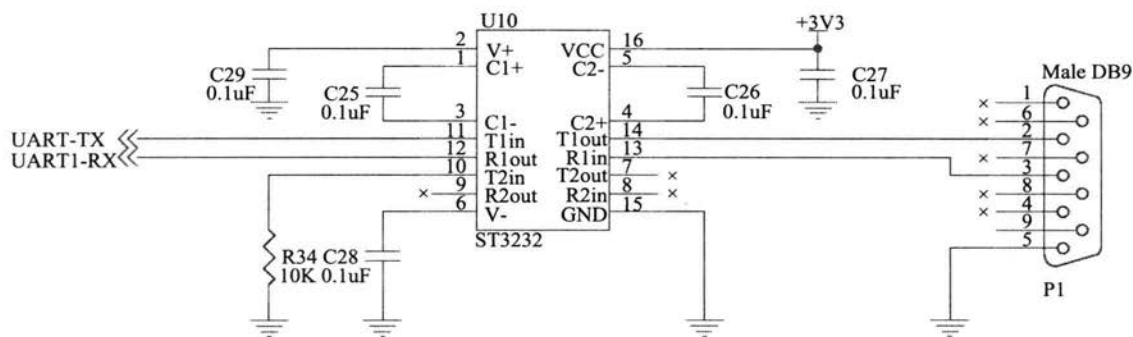


图 2.19 RS232 接口电路

4. SD 卡接口

SD 卡共支持三种传输模式：SPI 模式（独立序列输入和序列输出）、1 位 SD 模式（独立指令和数据通道，这是独有的传输格式）和 4 位 SD 模式（使用额外的引脚以及某些重新设置的引脚。支持 4 位宽的并行传输）。SD 卡接口电路设计如图 2.20 所示，需要注意的是，接口形式必须与传输模式相匹配。比如，本例采用 SPI 模式时，仅需要连接 MISO、SCK、MOSI、SDCS 四根物理信号线。若采用 4 位 SD 模式时，接口的电气连接方式必须做相应的调整，读者可参考 SD 2.0 规范。

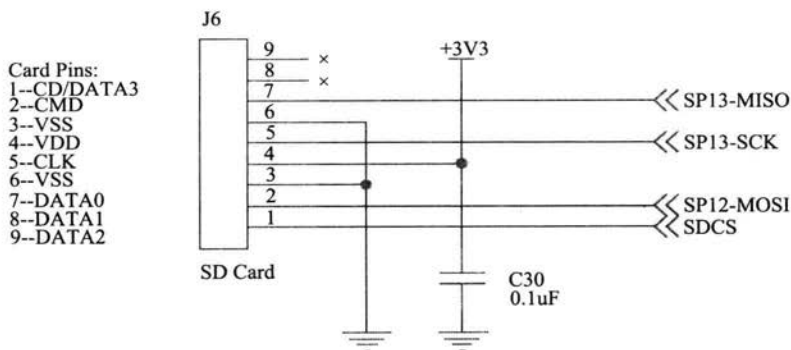


图 2.20 SD 接口电路

5. 红外收发接口、射频模块接口等

此部分电路较为简单,读者参考芯片数据手册进行了解,在此不再赘述。部分电路示意图如图 2.21~图 2.24 所示。

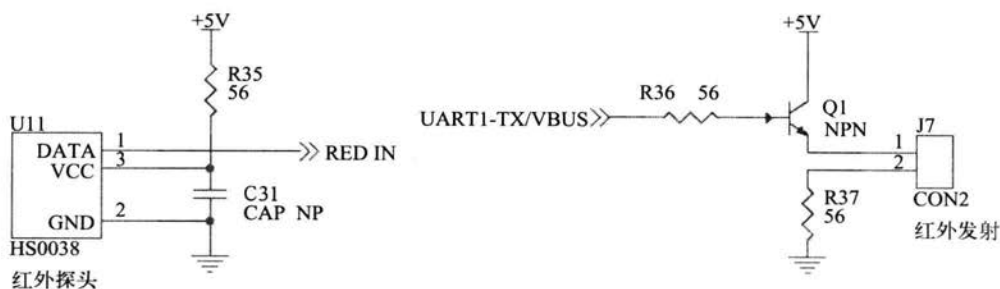


图 2.21 红外接口电路

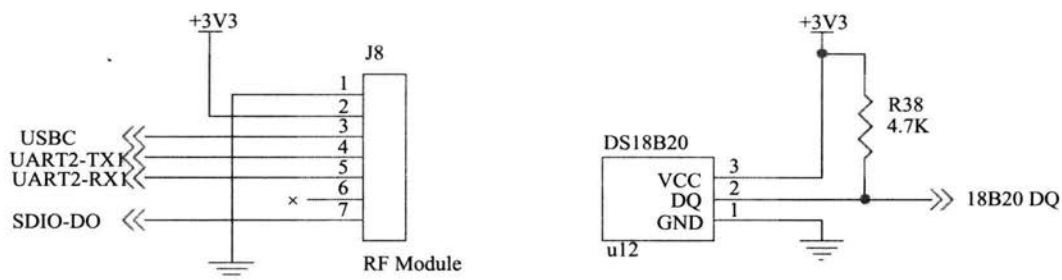


图 2.22 射频模块接口及温度传感器电路

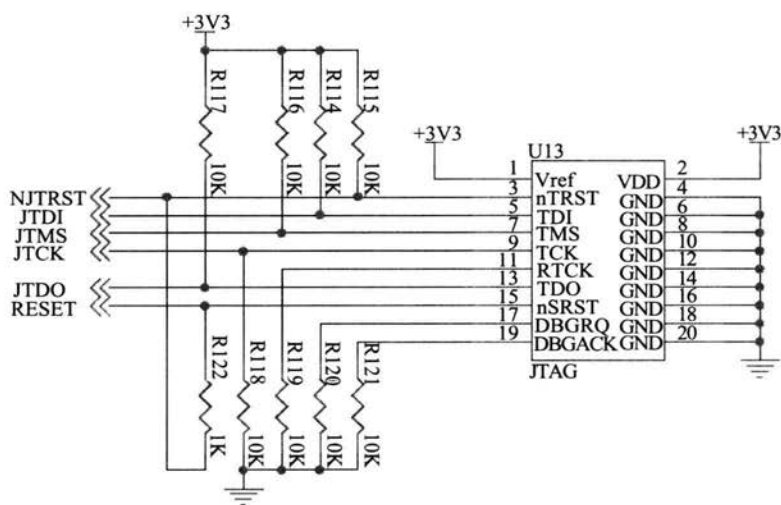


图 2.23 JTAG 调试接口

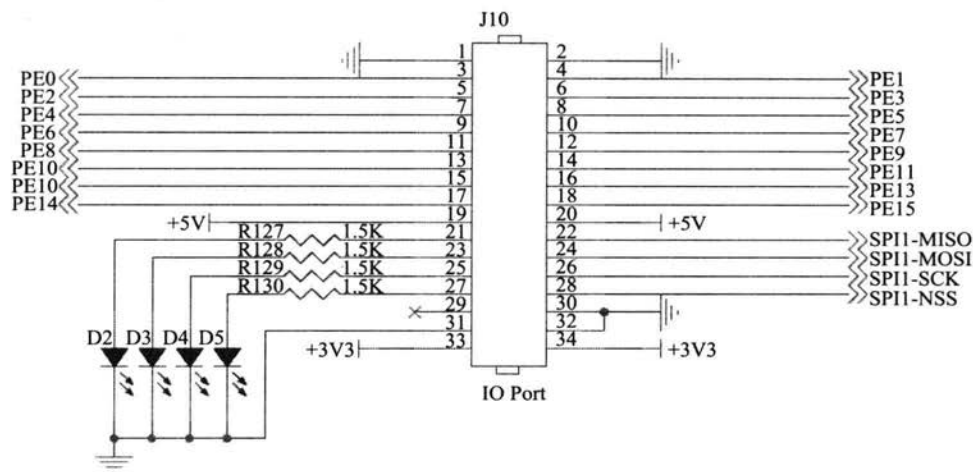


图 2.24 IO 及 LED 指示灯

6. 自定义功能键与复位键

按键开关是各种电子设备不可或缺的人机接口。在实际应用中，很大一部分的按键是机械按键。在机械按键的触点闭合和断开时，都会产生抖动，为了保证系统能正确识别按键的开关，就必须对按键的抖动进行处理。

在系统设计中，有各种各样的消除按键抖动的处理方法，通常有硬件消抖和软件延时消抖两种方法。虽然这两种方法都比较成熟，但在设计中应予以重视，保证人机交互时的友好和电路的稳定性。按键接口电路如图 2.25 所示

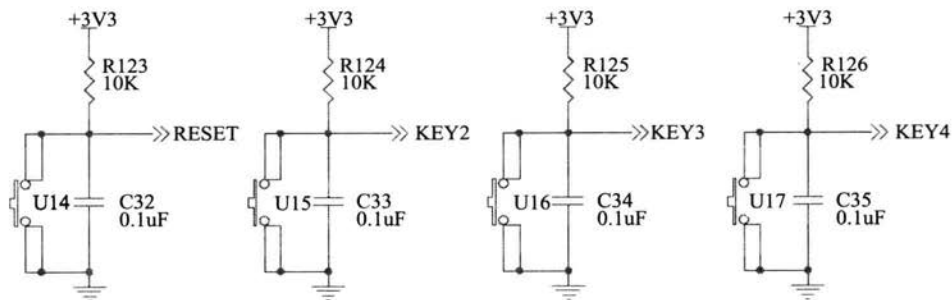


图 2.25 按键接口电路

7. 外扩 TFT 液晶模块接口

TFT (Thin Film Transistor) 一般代指薄膜液晶显示器。显示屏由许多可以发出任意颜色的光线的像素组成，通过控制各个像素显示相应的颜色来达到既定的视觉效果，传达特定的信息。在嵌入式应用领域，TFT 液晶显示屏通常以“裸屏”或模组形式出现。所谓“裸屏”，即未集成

任何驱动, 需要工程师进行综合软硬件开发才能实现既定的显示功能。而模组则具备了完整显示功能要素, 具有严格、完整的接口规范, 工程师仅需要按照规范对接口进行读写即可实现人机交互。下面以 ILI9320 为驱动芯片的液晶模组作为显示单元, 该模组采用 16 位并行接口驱动, 带有触摸功能, 接口较为简单, TFT 液晶模块接口电路图如图 2.26 所示。

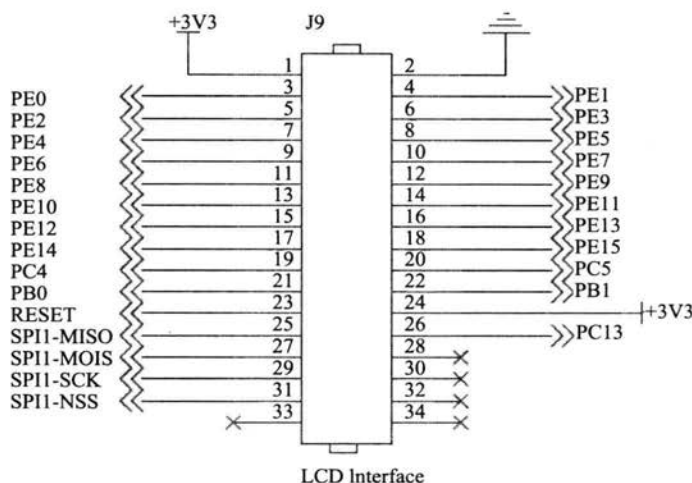


图 2.26 TFT 液晶模块接口电路

8. 模拟转换、蜂鸣器、E2PROM

E2PROM 采用 I2C 接口与微控制器相连, 读写命令格式可参考 24C16 数据手册。模拟电路为简单的数据采集试验提供便利, 实际工程应用应综合考虑应用背景与实际需求。蜂鸣器可采用 PWM 方式驱动, 实现音调和音量的调节。模拟转换、蜂鸣器、E2PROM 电路图如图 2.27 所示。

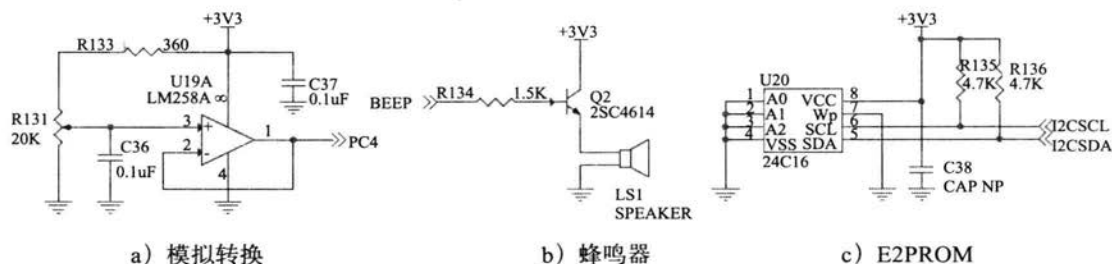


图 2.27 模拟转换、蜂鸣器、E2PROM

9. 微控制器

图 2.28 为微控制器部分的电气连接原理图, 本节中的电路采用平坦式电路设计模式, 因此各模块的外接端口均采用 offpage connector 引出。

2.3 硬件设计要点

本节以电路的电磁兼容设计为出发点，简要介绍在硬件开发工程中的注意事项和要点，并讲述了板级电路的信号完整性设计、电源完整性设计一般原则。

2.3.1 电磁兼容问题

电磁兼容性是电子设备或系统的主要性能之一，电磁兼容设计是实现设备或系统规定功能、使系统效能得以充分发挥的重要保证。设计人员必须在设备或系统功能设计的同时进行电磁兼容设计，充分考虑系统、分系统与周围环境之间的相互骚扰。

电磁兼容设计的目的是使所设计的电子设备或系统在预期的电磁环境中实现电磁兼容。最终的设备或系统应能在预期的电磁环境中正常工作，无性能损失或故障。同时，该设备或系统应不会对其他设备或系统的正常运行产生不利的影响。

本小节仅向读者介绍在电路设计或产品设计时，应遵循的基本原则和设计要领，读者可通过阅读相关书籍进行深入学习。

1. 电磁兼容设计的基本内容

在电子系统的开发中，每部分设计都应意识到电磁骚扰（EMI）问题。采取正确的防护措施能有效减小电子系统本身的EMI发射。大部分的骚扰问题可以在设计与开发过程中解决。若在设计之初未进行电磁兼容的考量与设计，工程师们将不得不在事后投入更多的精力去解决系统的骚扰问题，事后解决这些问题的成本也将成倍增加。而抗扰度是电磁兼容设计考量的另一个方面，是对设备或系统承受外接干扰能力的一个度量。

电磁兼容性设计又可分为系统内和系统间两部分。主要是对系统之间及系统内部的电磁兼容性进行分析、预测、控制和评估，实现电磁兼容和最佳效费比。

（1）系统间电磁骚扰的预测和控制

- ❑ 系统间电磁骚扰的预测。系统间电磁骚扰的预测往往涉及处在同一电磁环境中的一个或多个潜在的电磁骚扰源与一个或多个敏感设备之间的干扰预测。通过归纳出包含许多参数的全面方案，可推导出电磁干扰预测的基本方程。
- ❑ 对有用信号的控制：频谱管理和规定发射功率、信号类型（调制和带宽）、线的空间覆盖范围、方向性和极化、使用时间和地点等。在设计阶段还应尽量减小镜像频率响应、谐波频谱电平，以及乱真发射和乱真响应。
- ❑ 对人为骚扰的控制。系统间人为骚扰源主要是其他系统的发射机谐波和乱真发射、高压输电线、工科医设备等的骚扰发射，可参照有关的EMC标准来考量和控制。
- ❑ 自然骚扰源通常无法控制，只有在系统性能设计时加以考虑，并采取适当的防护措施等。

（2）系统内电磁兼容的预测和设计

- ❑ 系统内电磁兼容的预测，可通过理论分析、软件仿真等手段进行评估。
- ❑ 通常将系统内电磁兼容设计分为五个部分：印制电路板设计和元器件的选用、滤波、屏

蔽、布线以及接地。

2. 电磁兼容设计的效费比

在设备或系统设计的初始阶段, 同时进行电磁兼容设计, 把电磁兼容的大部分问题解决在设计定型之前, 可得到最高的效费比。如果等到生产阶段再去解决, 非但在技术上带来很大的难度, 而且会造成人力、财力和时间的极大浪费。

3. 电磁兼容设计的目标

电磁兼容设计的目标是实现 EMC 指标要求并通过 EMC 试验和认证。EMC 试验的项目通常依据产品应用的领域, 并参考国家标准或行业标准而定。

4. 电磁兼容设计的方法

电磁兼容设计的基本方法是指标分配和功能分块设计。也就是首先要根据有关的标准(国际、国家、企业、特殊标准等)把整体电磁兼容指标逐级分配到各功能块上, 细化成系统级的、设备级的、电路级的和元件级的指标。然后, 按照要实现的功能和电磁兼容指标进行电磁兼容设计, 如按电路或设备要实现的功能、骚扰源的类型、骚扰传播的渠道等。具体有时钟电路设计、防静电设计、防雷设计等。

在电磁兼容设计中有许多应用课题要解决, 如电磁波的散射、透射、传输、孔缝耦合、绕射理论等在实际问题中的求解问题, 各种骚扰源的机理和特性, 各种骚扰参数的计算和测试, 各种结构的屏蔽效果, 各种防护方法、测试方法、选用标准等。

在进行电磁兼容设计时, 可根据防护措施在实现电磁兼容时的重要性, 分层依次进行设计, 例如, 第一层为有源器件的选择和印制版设计; 第二层为接地设计; 第三层为屏蔽设计; 第四层为滤波设计, 然后进行综合设计。这称为分层与综合设计法。

随着电子产品性能的不断提高, 电磁兼容性问题会愈来愈突出。CPU 主频的提升、总线速度的提高、开关电源的广泛使用及小型化、板卡频繁升级使得开关频率不断增加等因素使电子产品的电磁兼容设计越来越迫切, 也越来越复杂。这也促使人们在进行产品开发之初就必须认真考虑电磁兼容设计。

2.3.2 信号完整性

在低速电路中, 电平跳变时信号上升时间较长, 器件间的互联线对电路的功能的影响可以忽略不计, 没有高速电路设计经验的设计人员通常意识不到信号完整性问题。随着器件输出开关速度的提高, 上升时间大多处于皮秒级, 按照低速电路设计方式设计的电路几乎无法工作, 无一例外地产生了信号完整性问题。另外, 对低功耗和速度追求使得内核电压越来越低, 而较低的内核电压极易受环境干扰, 若不加以控制有可能湮没在噪声中。因此系统能容忍的噪声余量越来越小, 这也使得信号完整性问题更加突出。

信号完整性(Signal Integrity, SI)是2000年后发展起来的新技术。SI解决的是在高速电路中信号传输过程中的质量问题。信号完整性可以泛指由互联线引起的所有电信号异常现象, 包括: 噪声、干扰和时序等。狭义的信号完整性是指信号电压(或电流)波形的形状及质量, 包括

反射和串扰。由于物理互联造成的干扰和噪声,使得连线上信号的波形外观变差,出现了异常形状的变形,这称为信号完整性被破坏。信号完整性问题是物理互联在高速情况下的直接结果。

提示 物理互联:物理互联(Interconnect)包括芯片内连线、芯片封装、PCB板及电子系统连接等,它们极大地影响高速时的信号和电源分配网络质量。真实的互联线包括芯片内连线、压焊点、封装引线、芯片引脚,以及芯片外的PCB板线接头、线条、接插件、连接电缆等。此外还有各种无源元件,包括介质、基板、屏蔽盒、机壳、机架等。

过去人们简单地将100MHz时钟作为高速电路与低速电路的分界线。事实上,高频不一定高速,低频也不见得低速。信号完整性问题的根源在于信号上升时间的减小,即使布线拓扑结构没有变化,当系统中的数字信号的上升沿小于1纳秒时,互联不再透明,也可能对电路和系统造成颠覆性后果。随着现代数字电子系统核心频率的不断提高,ASIC/PCB的设计必然面临日益突出的信号完整性问题。

信号不完整问题是物理互联在高速下必须认真对待的问题,否则将直接导致严重后果。信号完整性以传输线、电磁学等为理论基础,结合复杂的算法和模型,解决高速信号传输中的以下几个问题:反射、轨道塌陷、串扰、过冲、振铃、地弹、多次跨越逻辑电平错误、阻抗控制和匹配、EMC、热稳定性、时序分析等。接下来简述几种常见的信号完整性问题。

1. 反射

反射(reflection)是指传输线上有回波。信号功率(电压和电流)的一部分经传输线上传输到负载端,但是有一部分被反射回来形成振铃。

当信号沿着传输线向前传播时,无论什么原因使瞬时阻抗发生了改变,部分信号都将沿着与原传播方向相反的方向反射,而另一部分将继续传播,但幅度有所改变。瞬时阻抗发生改变的地方称为阻抗突变,或简称突变。引起反射的原因可能是信号传输中途遇到电阻、电容、电感、过孔、接插件等。图2.29显示了信号反射引起的波形畸变。电路设计时在时钟输出信号上串接一个小电阻,采用阻抗匹配的方法解决信号反射问题。阻抗在信号完整性问题中占据极其重要的地位。

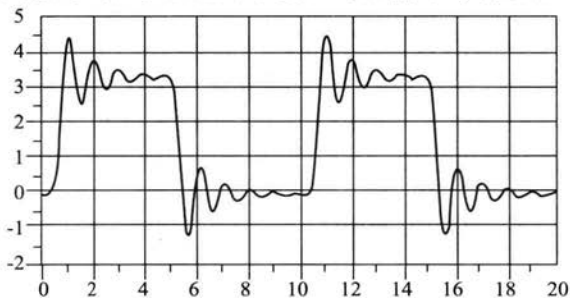


图 2.29 信号反射引起的波形畸变

2. 串扰

串扰(Crosstalk)是指在两个不同的电性能网络之间的互作用。产生串扰的称为 Aggressor (侵略者),而被干扰的称为 Victim (受害者)。通常,每一个网络既是 Aggressor,又是 Victim。

当两根信号线靠得很近时,一根信号线上的信号会通过电磁场耦合到另一根信号线上,这就是串扰。虽然被串扰影响的信号线上的波形不一定和邻近信号波形相似,也不一定有明显的规律,更多的是表现为噪声形式。随着集成电路和PCB设计的微型化、信号传输空间的聚敛,信号传输

通道必然靠得很近,因此信号的串扰问题无法避免,同时也是必须认真对待的问题。串扰问题在当今的高密度电路板中一直困扰着嵌入式开发人员。

串扰大小和电路板上的很多因素有关,PCB板层的参数、信号线间距、驱动端和接收端的电气特性及线端连接方式对串扰都有一定的影响。当然,信号传输通道的物理间距最容易控制,也是最常用的解决串扰的方法。

3. 轨道塌陷

噪声不仅存在于信号网络中,也存在于电源分配系统中。在电源分配系统中,由于存在能源传输通道阻抗,当传输通道中的电流发生变化时,电压会发生一定程度的衰减,甚至大幅跌落,通常将这一过程形象地称为“轨道塌陷”。轨道塌陷有时会产生致命的问题,比如电压“塌陷”致使主控电路复位,对于安全性要求较高的应用场合,这种情况是不允许发生的。

此外,高性能处理器集成的门数越来越多,开关速度也越来越快,在更短的时间内消耗更多的开关电流,可以容忍的噪声变得越来越小。但同时控制噪声越来越难,因为高性能处理器对电源系统有苛刻的要求,构建更低阻抗的电源分配系统变得越来越困难,阻抗控制成为高速电路设计中不可回避的问题。

通常,信号完整性采取经验法则、解析近似、数值仿真、实际测量等方法解决。工程师通过加强理论学习,在工程实践中不断积累实践经验,并结合运用分析仿真工具,可以全面提升自己解决信号完整性问题的分析和解决能力。值得一提的是,现在的SI仿真引擎完全可以仿真高速数字PCB,自动屏蔽SI问题并生成精确的“引脚到引脚”延迟参数。这使得器件模型和电路板制造参数的精确性成为决定仿真结果的关键因素。很多设计工程师首先仿真“最小”和“最大”的设计角落,然后采用相关的信息来解决问题并调整生产率。后制造阶段采取上述措施可以确保电路板的SI设计品质。在电路板装配完成之后,仍然有必要将电路板放在测试平台上,利用示波器或者时域反射计测量,将真实电路板和仿真预期结果进行比较。这些测量数据可以帮助工程师改进模型和制造参数,以便在下一轮预设计调研工作中做出更佳的(更少的约束条件)决策。常见的仿真工具有SPICE、Mentor公司的Hyperlynx、Cadence公司的SigXP、Ansoft公司的HFSS和Agilent公司的ADS等。

上述内容仅简单介绍几种常见的信号完整性问题,目的在于让刚涉足高速电路设计的工程师有初步的认识。信号完整性问题涉及面比较广,只有通过深入学习理论和不断积累工程实践才能逐步提升对信号完整性问题的认识和把握。

2.3.3 电源完整性

在以往的电路设计中,人们为了简化问题,通常把电源和地当成理想的情况来处理,将主要精力集中研究信号线,但在高速电路设计中,这种处理方法将给产品带来潜在的隐患。电源完整性和信号完整性是紧密联系在一起,从广义上说,信号电源完整性属于信号完整性研究范畴,电源完整性直接影响最终PCB板的信号完整性。电源完整性和信号完整性二者是密切关联的,而且很多情况下,影响信号畸变的主要原因是电源系统。例如,地反弹噪声太大、去耦电容的设计

不合适、回路影响很严重、多电源/地平面的分割不好、地层设计不合理、电流不均匀等。

过去人们更关注对集成电路功耗的控制，很少注意电源完整性，尽管电源完整性在决定电源和能耗方面具有举足轻重的作用。随着现代数字电子系统核心频率的不断提高，电源完整性已成为主导的设计约束条件之一。

1. 电源完整性概念

简单地讲，电源完整性是指特定电源与理想状态的接近程度，具体取决于电源的自然特性。对于家用设备的电源来说，主要关心的是电压幅度和频率，即不管附近的负荷和用电限制如何变化，电压幅度和频率能够保持多大的稳定性。对于优秀的电源完整性而言，重要的是供电电压差值中发生的降压和过冲或瞬时（和静态）变化保持在很小的范围内，比如标称值的5%，从而使集成电路保持可预测的性能。

造成电源不稳定的根源主要有两方面：一是器件在高速开关状态下瞬态的交变电流过大；二是电流回路上存在的电感。从表现形式上来看，又可以分为三类：同步开关噪声（Simultaneous Switch Noise, SSN，也称为 Δi 噪声）、非理想电源阻抗影响、谐振及边缘效应。有关这些主题的内容，这里仅作简要介绍，读者可参阅相关的书籍深入学习。

2. 电源完整性的主要问题

（1）电源阻抗控制

电源噪声的产生在很大程度上归结于非理想的电源分配系统。电源分配系统的作用就是给系统内的所有器件提供足够的电源，这些器件不但需要足够的功率消耗，同时对电源的平稳性也有一定的要求。大部分数字电路器件对电源波动的要求在正常电压的 $\pm 5\%$ 范围内。由于电源平面阻抗的存在，在瞬间电流通过的时候，器件端的电压就会产生一定的压降和摆动。

为了保证每个器件始终都能得到正常的电源供应，就需要对电源的阻抗进行控制，也就是尽可能降低其交直流阻抗。为了降低电源的电阻和电感，电源层和地层通常采用电阻率低的材料，在允许的情况下，使电源线尽可能加粗，并尽可能减少长度。事实上，目前市面上基本上都采用了大面积的铜皮层作为低阻抗的电源分配系统。

基于电源阻抗的要求，以往的电源总线形式已经不再适用于高速电路。当然，电源层本身的低阻抗也不能满足设计的需要，需要考虑的问题还有很多，比如，芯片封装中的电源引脚、连接器的接口，以及高频下的谐振现象等，这些都可能会造成电源阻抗显著增加。解决这些问题的最常用的方法就是合理增加去耦电容。

（2）同步开关噪声分析

同步开关噪声是指当器件处于开关状态，产生瞬间变化的电流 (di/dt)，在经过回流途径上存在的电感时，形成交流压降，从而引起噪声，也称为 Δi 噪声。如果是由于封装电感而引起地平面的波动，造成芯片地和系统地不一致，这种现象称为地弹。同样，如果是由于封装电感引起的芯片和系统电源差异，就称为电源反弹。所以，严格来说，同步开关噪声并不完全是电源的问题，它对电源完整性产生的影响最主要表现为地/电源反弹现象。

同步开关噪声主要是伴随着器件的同步开关输出（Simultaneous Switch Output, SSO）而产

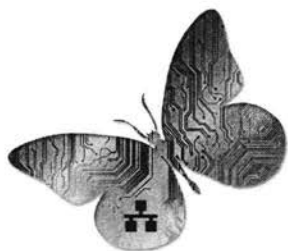
生, 开关速度越快, 瞬间电流变化越显著, 电流回路上的电感越大, 则产生的 SSN 越严重。基本公式为:

$$VSSN=N \cdot L_{Loop} \cdot (dI/dt)$$

其中 I 指单个开关输出的电流, N 是同时开关的驱动端数目, L_{Loop} 为整个回流路径上的电感, 而 VSSN 是同步开关噪声的大小。

(3) 谐振及边缘效应

电源平面可等效为由很多电感和电容构成的网络, 与通常 LC 网络一样, 在一定频率下, 由这些电容和电感构成的电源网络也会发生谐振现象, 从而影响电源层的阻抗。除了谐振效应外, 电源平面和地平面的边缘效应同样是电源设计中需要注意的问题, 这里说的边缘效应就是指边缘反射和辐射现象。如果抑制了电源平面上的高频噪声, 就能很好地减轻边缘的电磁辐射, 通常采用添加去耦电容的方法来解决这一问题。边缘效应是无法完全避免的, 在设计 PCB 时, 要尽量让信号走线远离铺铜区边缘, 以避免受到太大的干扰。



第 3 章

开发环境

通常，ARM 的开发环境以开发套件或集成开发环境的方式提供给终端用户使用，本章将介绍常用的开发套件或集成开发环境及 IAR EWARM 和 RealView MDK 的安装。

3.1 开发环境及搭建

3.1.1 常见开发环境

1. ADS1.2

ADS 是 ARM 公司的集成开发环境软件，拥有非常强大的功能。ADS 的前身是 SDT，SDT 是 ARM 公司几年前的开发环境软件，目前 SDT 早已经不再升级。ADS 包括四个模块，分别是：SIMULATOR、C 编译器、实时调试器和应用函数库。

ADS 的编译器与调试器较 SDT 都有了非常大的改观，ADS1.2 提供完整的 Windows 界面开发环境。ADS1.2 的 C 编译器效率极高，支持 C 以及 C++，使工程师可以很方便地使用 C 语言进行开发。ADS1.2 还提供软件模拟仿真功能，使没有仿真器的读者也能够熟悉 ARM 的指令系统。配合使用 FFT-ICE，ADS1.2 提供了强大的实时调试跟踪功能，使得片内运行情况尽在掌握之中。ADS1.2 需要硬件支持才能发挥强大功能。目前支持的硬件调试器有 Multi-ICE 以及兼容 Multi-ICE 的调试工具如 FFT-ICE，而简易下载电缆不支持 ADS1.2。目前，ADS 也已不再升级，大多数嵌入式开发工程师已转向使用其他开发环境。

2. ARM RealView Developer Suite

ARM RealView Developer Suite 工具是 ARM 公司是推出的新一代 ARM 集成开发工具。支持所有 ARM 系列核，并与众多第三方实时操作系统及工具商合作简化开发流程。该开发工具包含以下组件：

- ☐ 完全优化的 ISO C/C++ 编译器。
- ☐ C++ 标准模板库。
- ☐ 强大的宏编译器。
- ☐ 支持代码和数据复杂存储器布局的连接器。
- ☐ 可选 GUI 调试器。

- ☐ 基于命令行的符号调试器 (armsd)。
- ☐ 指令集仿真器。
- ☐ 生成无格式二进制工具、Intel 32 位和 Motorola 32 位 ROM 映像代码的指令集模拟工具。
- ☐ 库创建工具。
- ☐ 内容丰富的在线文档。

3. IAR EWARM

IAR Embedded Workbench for ARM 是 IAR Systems 公司为 ARM 微处理器开发的一个集成开发环境 (下面简称 IAR EWARM)。相较于其他的 ARM 开发环境, IAR EWARM 具有入门容易、使用方便和代码紧凑等特点。

IAR Systems 公司目前推出的最新版本是 IAR Embedded Workbench for ARM 6.40。IAR Systems 公司提供 32KB 代码限制但没有时间限制的 Kickstart 版供用户试用。

IAR EWARM 中包含一个全软件的模拟程序 (simulator)。用户不需要任何硬件支持就可以模拟各种 ARM 内核、外部设备甚至中断的软件运行环境。用户可以从中和评估 IAR EWARM 的功能和使用方法。

IAR EWARM 的主要特点如下:

- ☐ 高度优化的 IAR ARM C/C++ 编译器。
- ☐ IAR ARM 汇编器。
- ☐ 一个通用的 IAR XLINK 连接器。
- ☐ IAR XAR 和 XLIB 建库程序和 IAR DLIB C/C++ 运行库。
- ☐ 功能强大的编辑器。
- ☐ 项目管理器。
- ☐ 命令行实用程序。
- ☐ IAR C-SPY 调试器 (先进的高级语言调试器)。

4. KEIL ARM-MDK ARM

MDK 即 RealView MDK (Microcontroller Development Kit), 是 ARM 公司目前最新推出的针对各种嵌入式处理器的软件开发工具。RealView MDK 集成了业内最领先的技术, 包括 uVision4 集成开发环境与 RealView 编译器。支持 ARM7、ARM9 和最新的 Cortex-M3/M1/M0 核处理器, 具有自动配置启动代码、集成 Flash 烧写模块、强大的 Simulation 设备模拟以及性能分析等功能, 与 ARM 之前的工具包 ADS 等相比, RealView 编译器的最新版本改善性能超过 20%。

Keil 公司开发的 ARM 开发工具 MDK 是用来开发基于 ARM 核的系列微控制器的嵌入式应用程序。它适合不同层次的开发者使用, 包括应用程序专业开发工程师和嵌入式软件开发的初学者。MDK 包含了工业标准的 Keil C 编译器、宏汇编器、调试器、实时内核等组件, 支持所有基于 ARM 的设备, 能帮助工程师按照计划完成项目。

Keil uVision 调试器可以帮助用户准确地调试 ARM 器件的片内外设 (I2C、CAN、UART、SPI、中断、I/O 接口、A/D 转换器、D/A 转换器和 PWM 模块等)。ULINK USB-JTAG 转换器将 PC 的 USB 端口与用户的目标硬件相连 (通过 JTAG 或 OCD)，使用户可在目标硬件上调试代码。通过使用 Keil uVision IDE/ 调试器和 ULINK USB-JTAG 转换器，用户可以很方便地编辑、下载和在实际的目标硬件上测试嵌入的程序。

MDK 支持 Philips、Samsung、Atmel、Analog Devices、Sharp、ST 等众多厂商基于 ARM7 内核的 ARM 微控制器。

- ☐ 高效工程管理的 uVision3 集成开发环境。
 - ☐ Project/Target/Group/File 的重叠管理模式，并可逐级设置。
 - ☐ 高度智能彩色语法显示。
 - ☐ 支持编辑状态的断点设置，并在仿真状态下有效。
- ☐ 高速 ARM 指令 / 外设模拟器。
 - ☐ 高效模拟算法缩短大型软件的模拟时间。
 - ☐ 软件模拟进程中允许建立外部输入信号。
 - ☐ 独特的工具窗口，可快速查看寄存器和方便配置外设。
 - ☐ 支持 C 调试描述语言，可建立与实际硬件高度吻合的仿真平台。
 - ☐ 支持简单 / 条件 / 逻辑表达式 / 存储区读写 / 地址范围等断点。
- ☐ 多种流行编译工具选择。
 - ☐ Keil 高效率 C 编译器。
 - ☐ ARM 公司的 ADS/RealView 编译器。
 - ☐ GNU GCC 编译器。
 - ☐ 后续厂商的编译器。

5. WINARM (GCCARM)

WINARM 是一个免费的开发工具。WINARM 的下载网址是：http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/。WINARM 中除了包含 C/C++ 编译器 (GCC)，汇编、连接器 (Binutils)，调试器 (GDB) 等工具，也包括了通过 GDB 使用 Wiggler JTAG 的软件——OCDRemote。所以，WINARM 发行版本中包括了所需要的所有工具。

3.1.2 IAR EWARM 安装

本小节以 IAR EWARM 6.3 主流开发环境为例，向读者介绍一个开发环境的搭建过程。

- 1) 启动安装程序，出现如图 3.1 所示界面，单击其中的“Install IAR Embedded Workbench”。
- 2) 在弹出的界面中单击“Next”，如图 3.2 所示。
- 3) 在图 3.3 中选“ I accept the terms of the license agreement”，并单击“Next”继续安装过程。
- 4) 在图 3.4 中输入注册信息，包括 License 码，并单击“Next”继续安装。

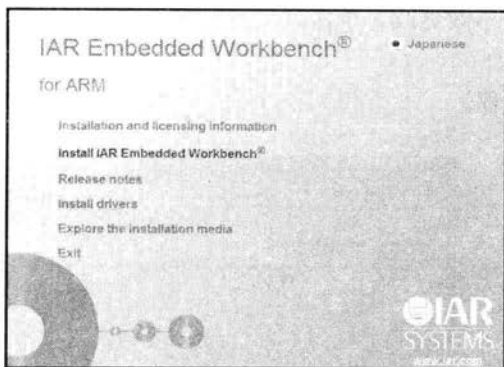


图 3.1 安装“IAR Embedded Workbench”

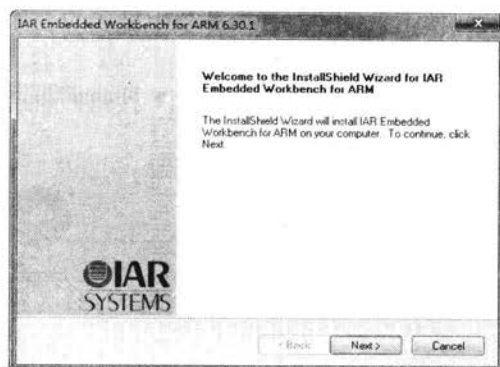


图 3.2 “IAR Embedded workbench for ARM 6.30.1” 欢迎界面

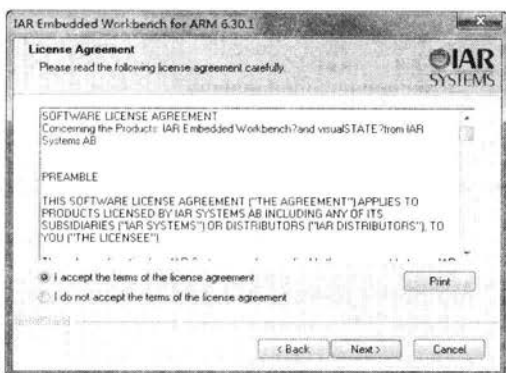


图 3.3 安装许可信息

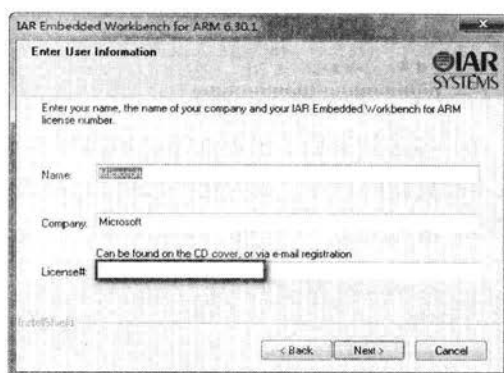


图 3.4 输入注册信息

5) 在图 3.5 中输入 License 码对应的 License Key, 并单击“Next”继续安装。

6) 在图 3.6 中设置 IAR EWARM 的安装路径, 并单击“Next”继续安装。

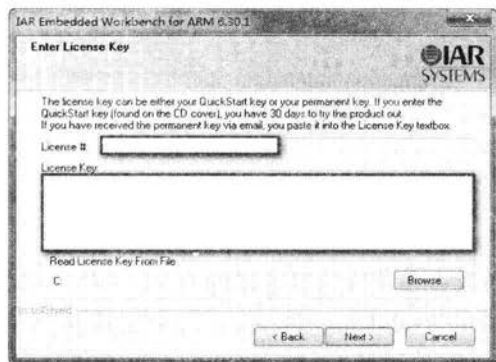


图 3.5 输入 License Key

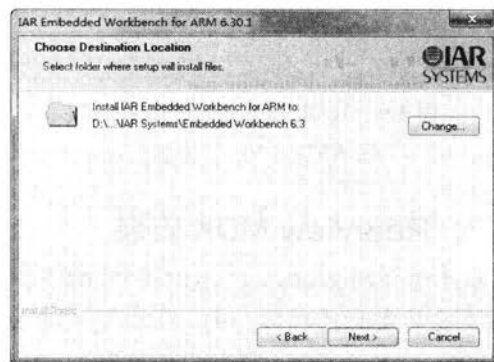


图 3.6 设置 IAR EWARM 的安装路径

7) 接下来设置在开始菜单中显示的名称, 一般不做更改, 单击“Next”继续安装, 如图 3.7 所示。

8) 单击图 3.8 中的“Install”按钮, 启动安装过程。

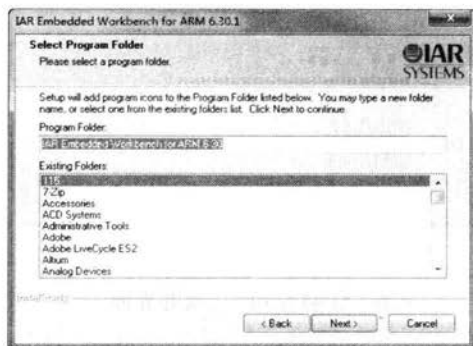


图 3.7 接受默认设置

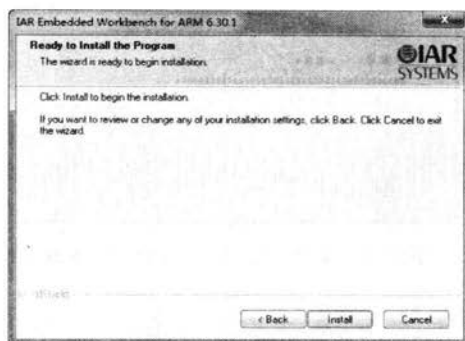


图 3.8 单击“Install”按钮, 开始安装

9) 安装过程如图 3.9 所示。

10) 安装结束后, 出现如图 3.10 所示界面, 用户可选择性勾选“View the release notes”浏览新版本的更新内容等, 或勾选“Launch IAR Embedded Workbench for ARM”启动 IAR EWARM 开发环境。最后单击“Finish”完成安装。

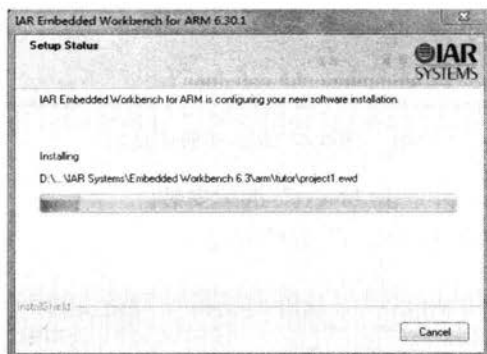


图 3.9 “IAR Embedded Workbench for ARM 6.30.1”安装中



图 3.10 安装结束

3.1.3 RealView MDK 安装

本小节介绍 RealView MDK V4.6 的安装过程, 若读者希望使用官方提供的 RL-ARM 实时库, 还需要安装实时库程序。RL-ARM 是 Keil MDK 自带的 Real-Time Library, 其中包括 RTX 内核 (Real-Time eXecutive, 实时操作系统)、RL-FlashFS (文件系统)、RL-TCPnet (TCP/IP 协议栈) 和 RL-CAN (CAN 总线函数库)。

MDK-ARM V4.6 的安装过程如下。

- 1) 启动 MDK-ARM V4.60 安装程序, 出现如图 3.11 所示界面。然后单击“Next”继续安装。
- 2) 弹出如图 3.12 所示界面, 勾选“I agree to all the terms of the preceding License Agreement”, 并单击“Next”继续安装。

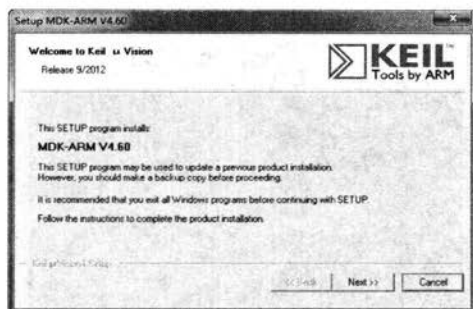


图 3.11 “MDK-ARM V4.60”欢迎界面

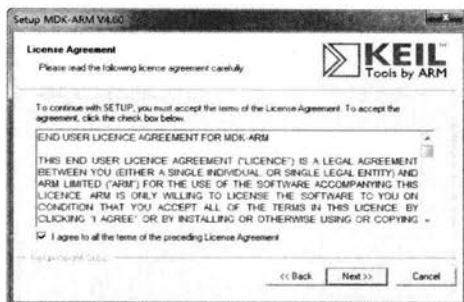


图 3.12 安装许可信息

- 3) 选择安装路径, 并单击“Next”继续安装, 如图 3.13 所示。
- 4) 输入注册信息, 单击“Next”启动安装过程, 如图 3.14 所示。

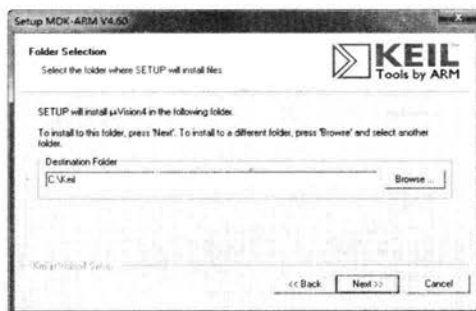


图 3.13 选择安装路径



图 3.14 输入注册信息

- 5) 进入安装过程如图 3.15 所示。
- 6) 选择默认设置, 选择“Next”继续安装, 如图 3.16 所示。

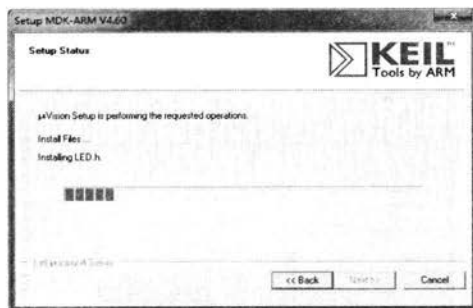


图 3.15 “MDK-ARM V4.60”安装中



图 3.16 接受默认设置

7) 若希望使用 ULINK 仿真器, 则勾选 “Launch Driver Installation : ‘ULINK Pro Driver V1.0’”, 并单击 “Finish” 继续安装, 如图 3.17 所示。

8) 若上步勾选了 “Launch Driver Installation : ‘ULINK Pro Driver V1.0’”, 则进入 ULINK 驱动安装界面, 安装完毕后, 将自动关闭安装界面, 如图 3.18 所示。

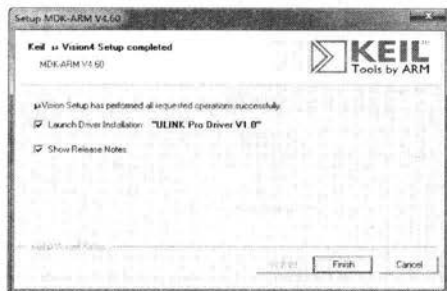


图 3.17 可选择性安装 “ULINK Pro Driver V1.0”

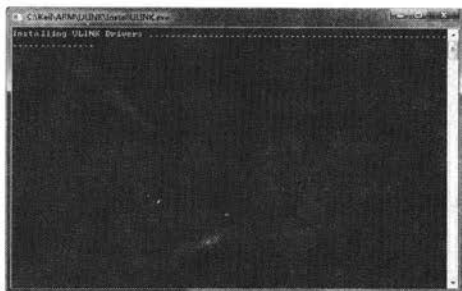


图 3.18 ULINK 驱动安装中

接下来将介绍 RealView Real-Time Library V4.13 的安装过程。

1) 启动安装程序, 如图 3.19, 并单击 “Next”。

2) 弹出如图 3.20 所示界面, 勾选 “I agree to all the terms of the preceding License Agreement”, 并单击 “Next” 继续安装。

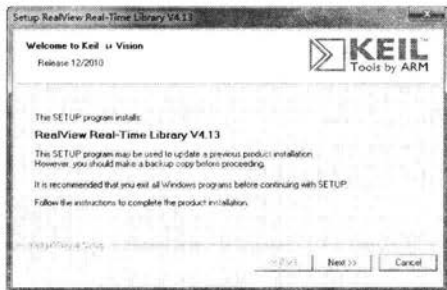


图 3.19 “RealView Real-Time Library V4.13” 欢迎界面

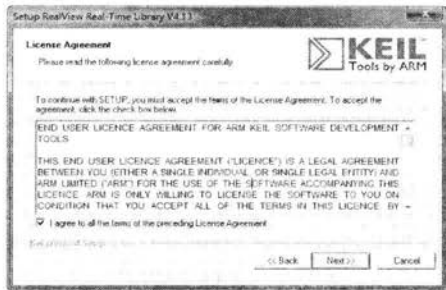


图 3.20 安装许可信息

3) 选择与 MDK-ARM V4.6 相同的安装目录, 并单击 “Next” 继续安装, 如图 3.21 所示。

4) 输入注册信息, 并单击 “Next” 继续安装, 如图 3.22 所示。

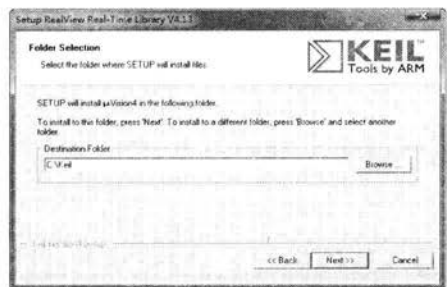


图 3.21 选择安装路径



图 3.22 输入注册信息

5) 单击“Finish”完成安装,可勾选“Show Release Notes”浏览软件发布更新记录,如图 3.23 所示。

至此,RealView MDK-ARM 和 RealView Real-Time Library 已安装完毕,启动 RealView MDK-ARM 并打开一个例程,界面如图 3.24 所示。



图 3.23 安装结束



图 3.24 启动 RealView MDK-ARM

3.2 相关开发工具

1. JTAG 调试仿真器

- ☐ 支持 ARM7/ARM9,支持自动检测和手动指定内核。
- ☐ 使用 RDI 接口,支持 SDT、ADS、RealView 和 IAR。
- ☐ 支持 ADS1.2、SDT2.51 和 RealView。
- ☐ 支持单个硬件断点或者数量不限的软件断点。
- ☐ 支持 ARM/THUMB 模式。
- ☐ 支持 LITTLEENDIAN 和 BIGENDIAN。
- ☐ 支持 SEMIHOSTING。
- ☐ 支持 WIGGLER SDTJTAG 和自定义接口。

2. H-JTAG 调试代理

H-JTAG 是一个免费的 ARM 调试代理,这个程序没有任何限制,是为广大 ARM 爱好者提供的一个简单实用的学习工具。

- ☐ 支持 ARM7/ARM9,支持自动检测和手动指定内核。
- ☐ 使用 RDI 接口,支持 SDT、ADS、RealView 和 IAR。
- ☐ 支持 ADS1.2、SDT2.51 和 RealView。
- ☐ 支持单个硬件断点或者数量不限的软件断点。

- ☐ 支持 ARM/THUMB 模式。
- ☐ 支持 LITTLEENDIAN 和 BIGENDIAN。
- ☐ 支持 SEMIHOSTING。
- ☐ 支持 WIGGLER SDDTAG 和自定义接口。

3. J-LINK

IAR 公司的 J-LINK 是一款小巧的 ARM JTAG 硬件调试器，它通过 USB 接口与 PC 相连。IAR 的 J-LINK 与该公司的嵌入式开发平台紧密结合，且完全支持即插即用。

J-LINK 的主要特征如下：

- ☐ 支持 ARM7 和 ARM9。
- ☐ 下载速度高达 1MB/s。
- ☐ 无需额外电源供电，可直接通过 USB 供电。
- ☐ 自动辨识。
- ☐ 监控所有的 JTAG 引脚信号，并测量电压。
- ☐ 20 引脚标准 JTAG 连接器。
- ☐ 配有 USB 接口和 20 引脚插槽。
- ☐ 支持 Windows 2000 和 Windows XP。

相较于其他调试器，J-LINK 具有如下优势：

- ☐ 支持 ADS、KEIL、IAR、WINARM、RV 等几乎所有开发环境，并且可以和 IAR 无缝连接。
- ☐ 支持 FLASH 软件断点，可以设置 2 个以上断点（无限个断点），极大提高调试效率。
- ☐ 带 J-LINK TCP/IP server，允许通过 TCP/IP 网络使用 J-LINK。
- ☐ 支持几乎所有 ARM7 和 ARM9，暂时不支持 XSCALE。

官方网站：<http://www.segger.com/>

4. ULINK

ARM7 TDMI 结构的 Keil 开发套件采用最新设计的超豪华 uVision3 集成开发环境，内嵌 C 编译器、汇编器、工程管理器、调试器等功能模块，是一款稳定、可靠、高效的开发工具，适用于不同层次的用户，完全满足从专业应用开发工程师到嵌入式软件开发初学者的所有使用要求。类似于 8051 的智能平台将大幅度缩短工程师的开发周期，并且将逐渐支持各大半导体厂商的所有 ARM 型号。

3.3 创建工程

本小节以一个简单的 DEMO 为例，向读者展示如何创建一个工程实例。在介绍之前，应首先从 ST 公司官方网站获得 STM32FXXX 系列芯片的驱动库，其官方网址为：http://www.st.com/internet/com/SOFTWARE_RESOURCES/SW_COMPONENT/FIRMWARE/stm32f10x_stdperiph_lib.zip。接下来具体介绍操作步骤。

1) 启动 IAR EWARM, 如图 3.25 所示。

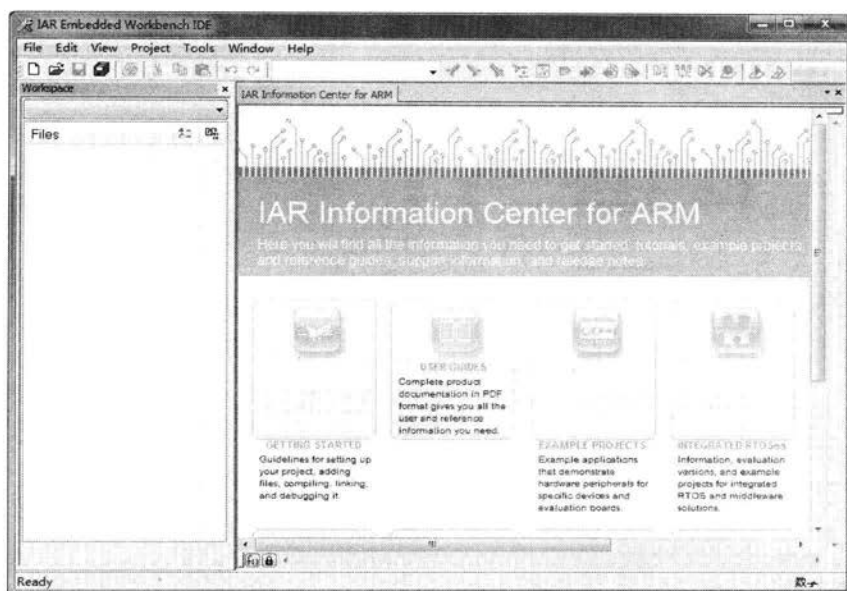


图 3.25 启动 IAR EWARM

2) 单击 “File” → “New” → “Workspace”, 新建工作空间, 如图 3.26 所示。

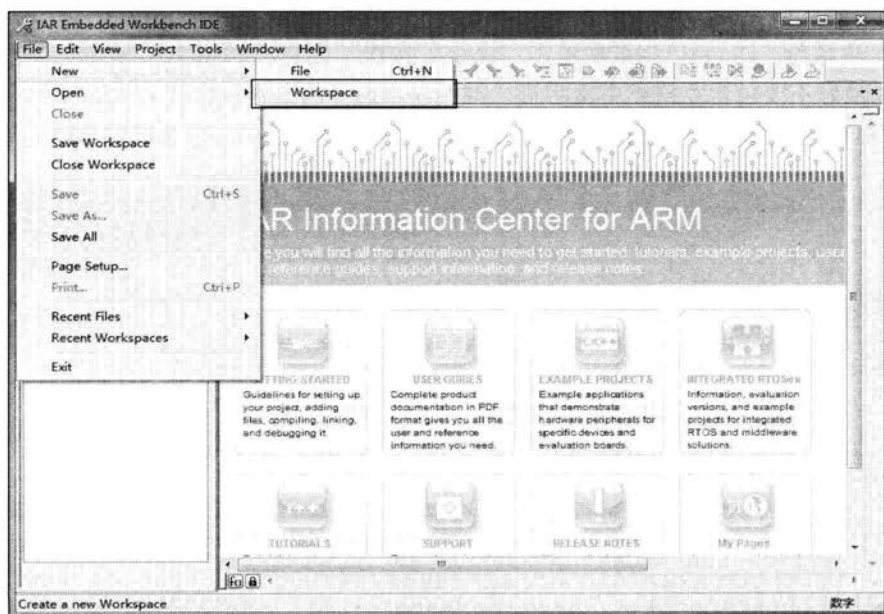


图 3.26 新建工作空间

3) 单击“Project”→“Create New Project”，如图 3.27 所示。

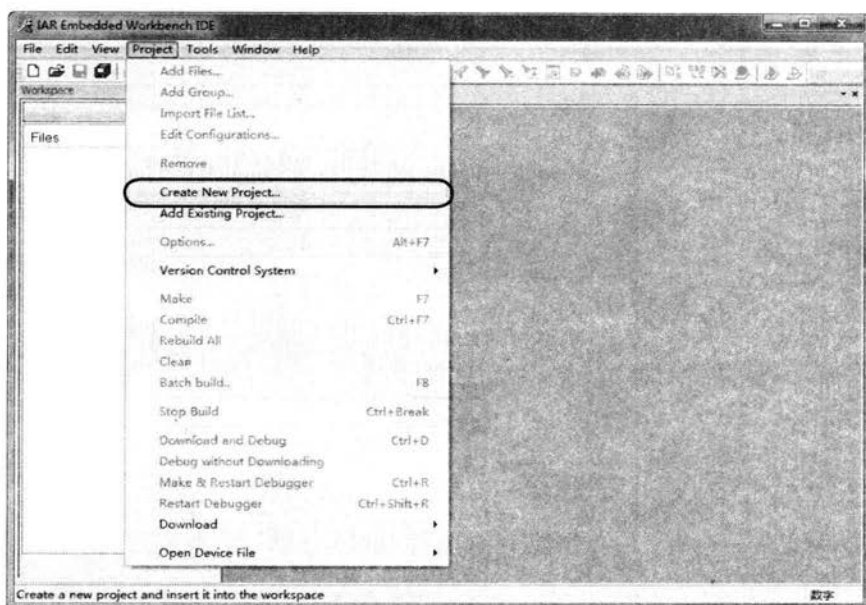


图 3.27 新建工程

4) 在完成步骤 3 后，弹出如图 3.28 所示对话框，选择“Empty project”，并单击“OK”按钮。

5) 完成步骤 4 后，弹出如图 3.29 所示对话框，读者选择或创建适当的工作路径，并给该工程命名，此处命名为“Demo1”，最后单击“保存”按钮，创建后如图 3.30 所示。需要注意的是，工程保存路径最好不要包含空格或中文字符。本例程中选择的工作路径为 G:\DemoProject\Demo1。

6) 给工程创建源文件分类目录，单击“Project”→“Add Group”，如图 3.31 所示。

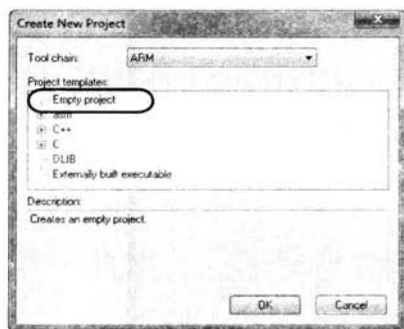


图 3.28 新建一个空工程



图 3.29 选择工作路径并设置工程名称

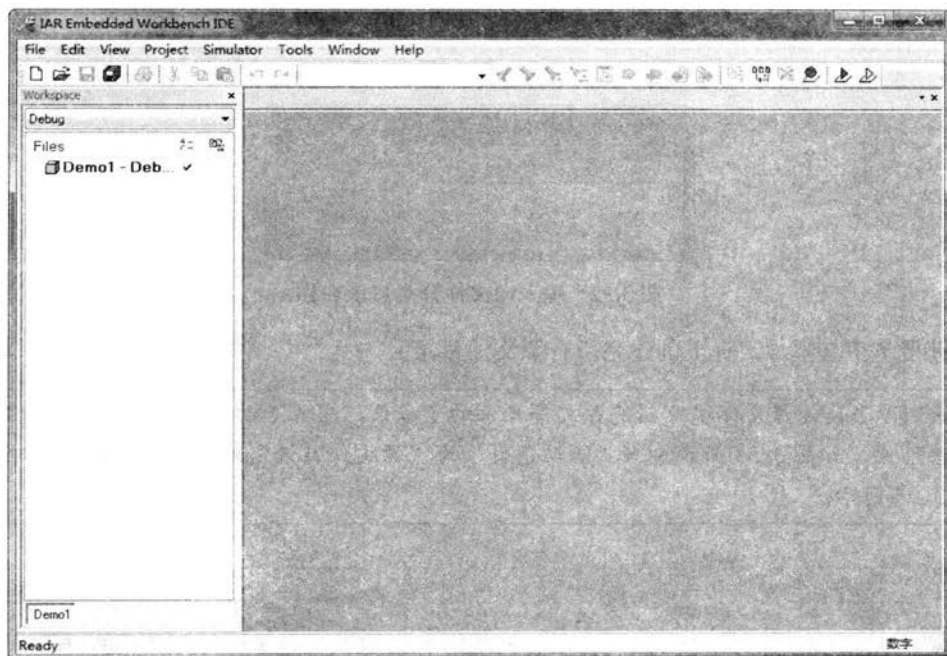


图 3.30 创建完成

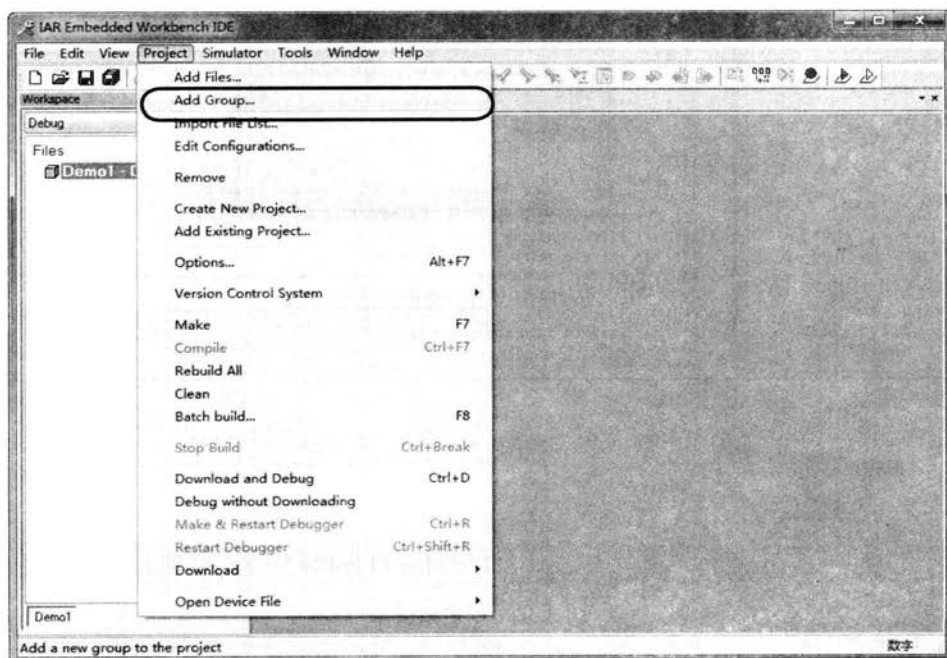


图 3.31 给工程创建源文件分类目录

7) 在完成步骤 6 后, 弹出如图 3.32 所示对话框, 输入源文件分类目录名称, 并单击“OK”按钮。

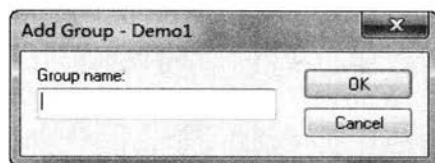


图 3.32 输入源文件分类目录名称

重复步骤 6 和步骤 7, 直至创建完所有源文件目录。

注意 IAR EWARM 开发环境可创建源文件目录的子目录, 当“活动条目”为“创建的工程”时, 创建的是根目录, 如图 3.33 所示; 当“活动条目”为“源文件目录”时, 创建的是与其对应的子目录, 如图 3.34 所示。

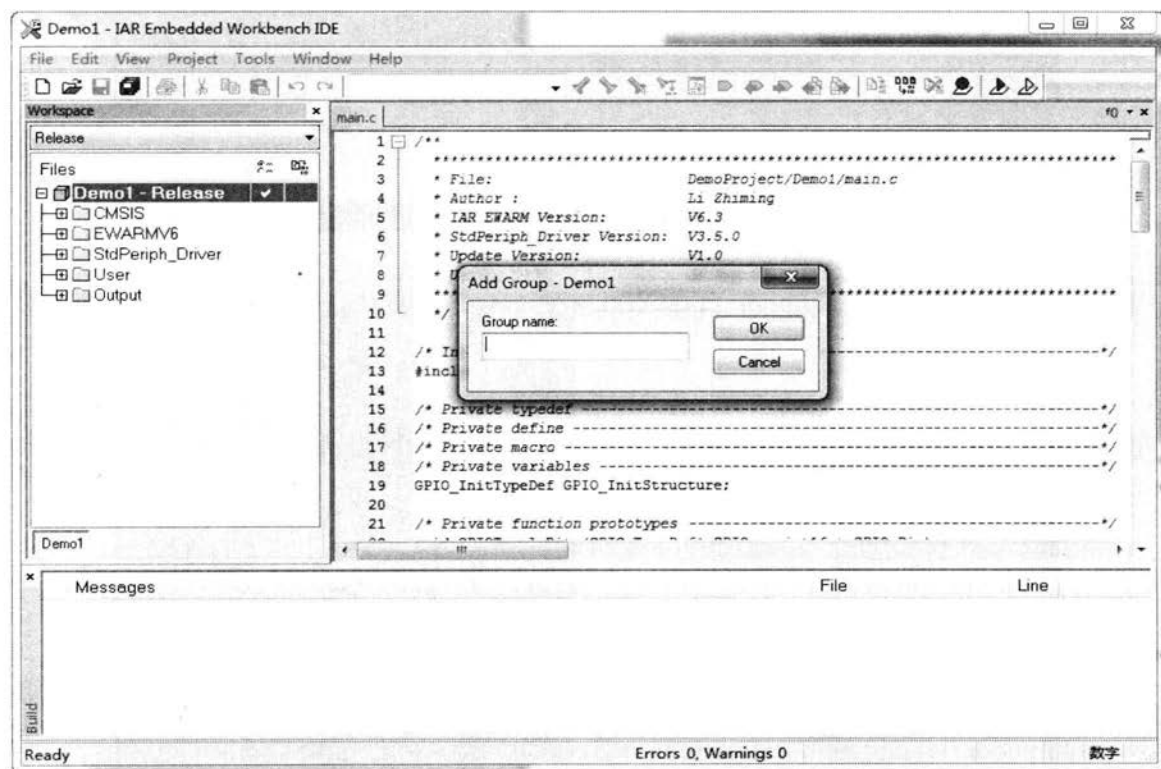


图 3.33 创建根目录

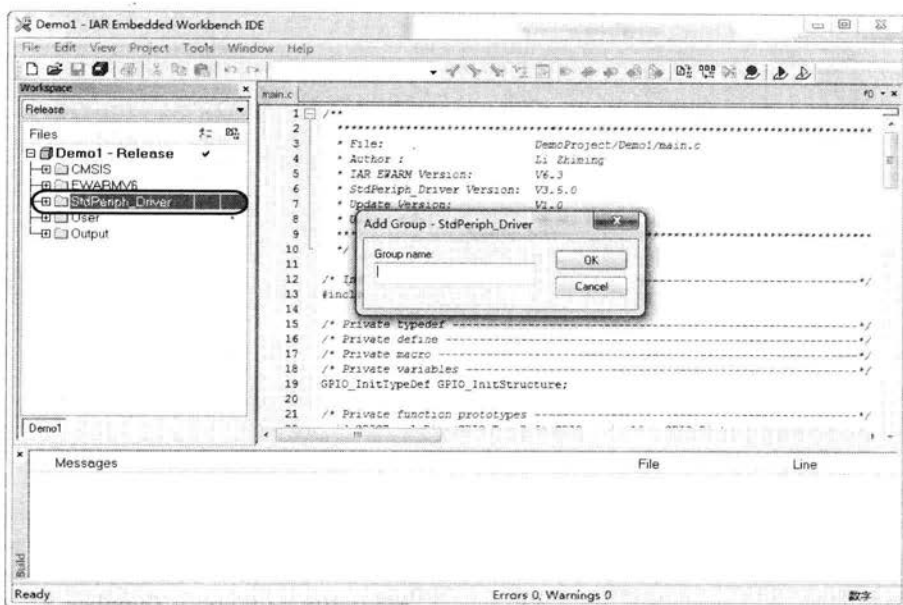


图 3.34 创建子目录

创建目录完毕，如图 3.35 所示。

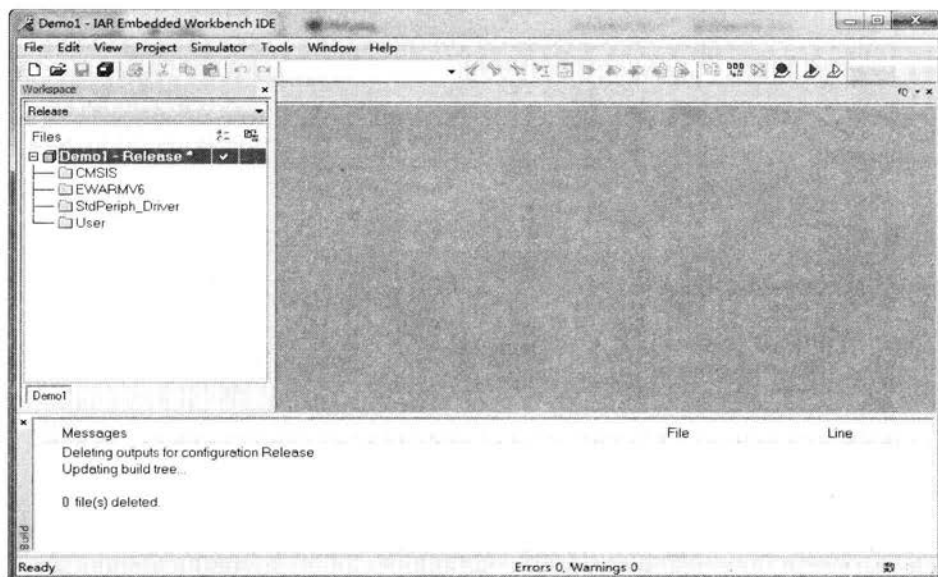


图 3.35 创建目录完成

8) 添加源文件。首先将 ST 公司提供的驱动解压缩，将其中的 Librarys 文件夹拷贝至 G:\DemoProject\Common 路径下，如图 3.36 所示。

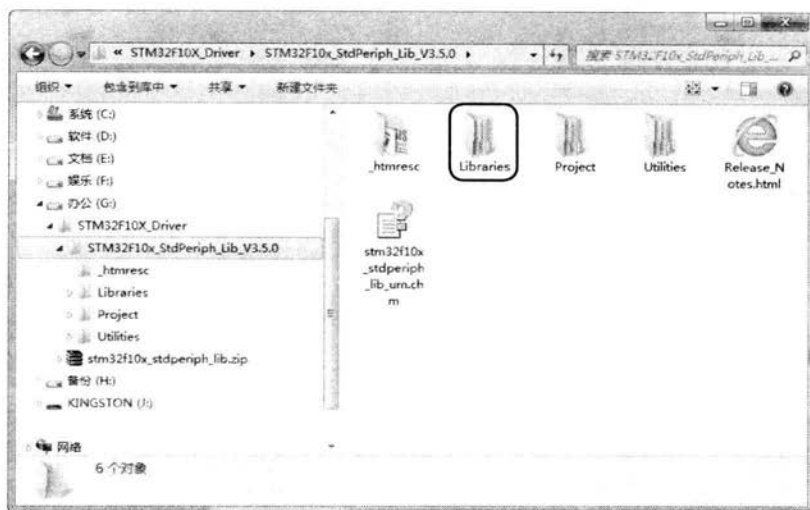


图 3.36 拷贝 Librarys 文件夹

将 \STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template\EWARM 文件夹下的 stm32f10x_flash.icf、stm32f10x_flash_extsram.icf、stm32f10x_nor.icf、stm32f10x_ram.icf 四个文件拷贝至 \DemoProject\Demo1 目录下，创建工程如图 3.37 所示。（实际上，仅需要其中一个即可，稍后介绍该文件。）



图 3.37 拷贝四个文件至 Demo1 目录下

添加源文件有两种方法：其一，在对应的源文件目录上单击右键，在弹出菜单中选择“Add Files”如图 3.38 所示；其二，选中源文件目录，然后单击“Project”→“Add Files”。在弹出的对话框中选择相应的源文件，如图 3.39 所示。

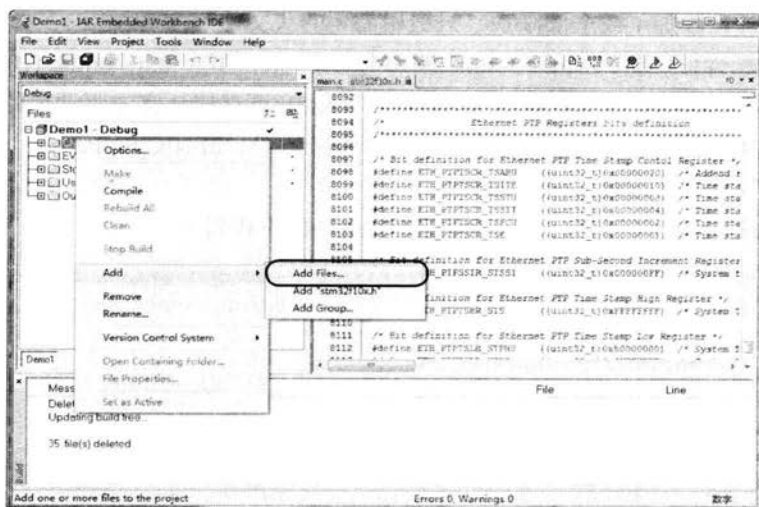


图 3.38 添加源文件方法一



图 3.39 添加源文件方法二

源文件目录中对应的源文件如下：

- ❑ CMSIS 源文件目录：system_stm32f10x.c，位于 \DemoProject\Common\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x 中。
- ❑ EWARMV6 源文件目录：startup_stm32f10x_cl.s，位于 \DemoProject\Common\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\iar 中。
- ❑ StdPeriph_Driver 源文件目录：stm32f10x_XXX.c (XXX 代表 ST 公司约定的外设英文缩写)，位于 \DemoProject\Common\Libraries\STM32F10x_StdPeriph_Driver\src 中，读者可根据实

际用到的外设添加；或全部添加，只引用需要的头文件。

- **User 源文件目录：**main.c、stm32f10x_it.c、stm32f10x_it.h、stm32f10x_conf.h，其中 stm32f10x_it.c、stm32f10x_it.h、stm32f10x_conf.h 可直接从官方驱动库的例程中拷贝，存放于 \STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template 文件夹中。用户仅需自己创建 main.c 文件。

9) 创建好 main.c 文件后，在 main.c 源文件中添加如下代码：

```

/*****
 * File name:                               DemoProject/Demo1/main.c
 * Author :                                  Li Zhiming
 * IAR EWARM Version:                       V6.3
 * StdPeriph_Driver Version:                V3.5.0
 * Update Version:                          V1.0
 * Update Date:                             08-Aug-2012
 *****/

/*Includes -----*/
#include "stm32f10x.h"

/*Private variables -----*/
GPIO_InitTypeDef GPIO_InitStructure;

/*Private function prototypes -----*/
void GPIOToggleBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);

/**
 * @brief Main program.
 * @param None
 * @retval None
 */
int main(void)
{
    /* GPIOD Periph clock enable */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

    /* Configure PD0 and PD2 in output pushpull mode */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    GPIO_ResetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_2);

    /* Infinite loop */
    while (1)
    {
        GPIOToggleBits(GPIOD, GPIO_Pin_0|GPIO_Pin_2);
        for(unsigned long i = 0; i<1000000; i++);
    }
}

```

```

    }
}

/**
 * @brief Toggles the selected GPIO Pin.
 * @param GPIOx: Specifies the GPIO Port.
 * This parameter can be of following parameters:
 * @arg GPIOx
 * @param GPIO_Pin: Specifies the Pin to be toggled.
 * This parameter can be one of following parameters:
 * @arg GPIO_Pin_x
 * @retval None
 */
void GPIONToggleBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Pin));

    GPIOx->ODR ^= GPIO_Pin;
}
/*****END OF FILE*****/

```

添加后的工程如图 3.40 所示，后续将介绍工程的设置。

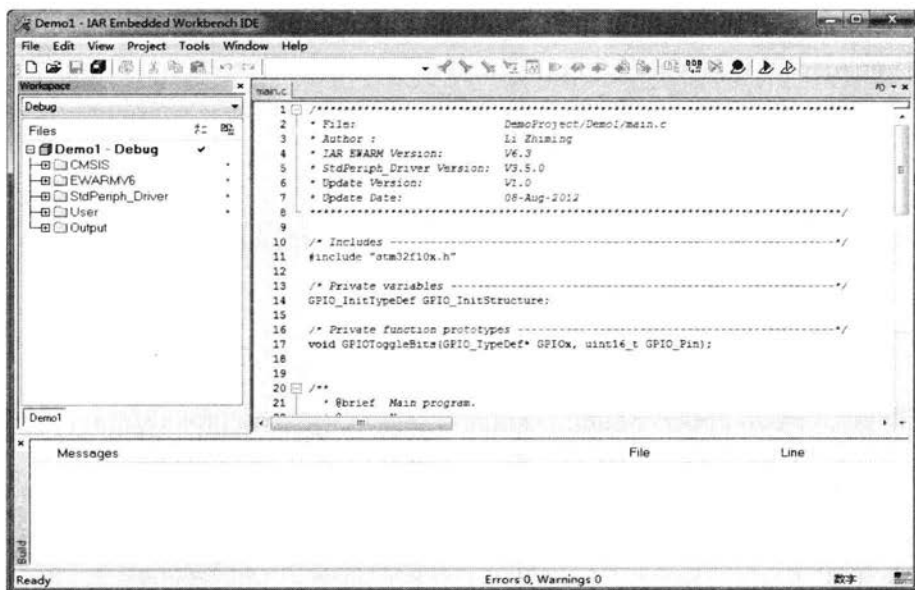


图 3.40 Demo1 工程界面

10) 工程设置。在“Workspace”区域的工程根目录上单击右键，在弹出菜单中单击“Options”，如图 3.41 所示。

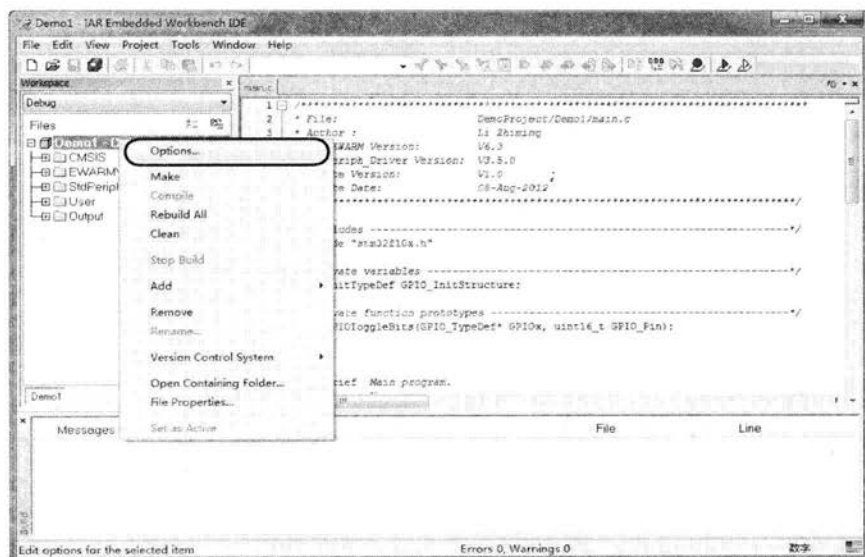


图 3.41 工程设置

11) 单击“Options”后弹出如图 3.42 所示对话框。

单击左侧列表框中的“General Option”，然后单击“Target”选项卡中“Device”后面的图标，设置处理器类别，根据实际选用的器件选择对应的型号，本工程采用的器件是 ST STM32F107VCT6，可按照图 3.43 进行选择。

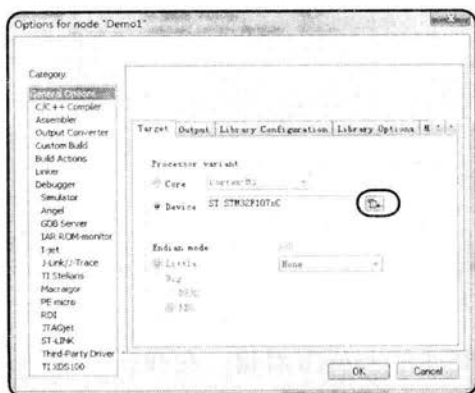


图 3.42 设置工程具体细节

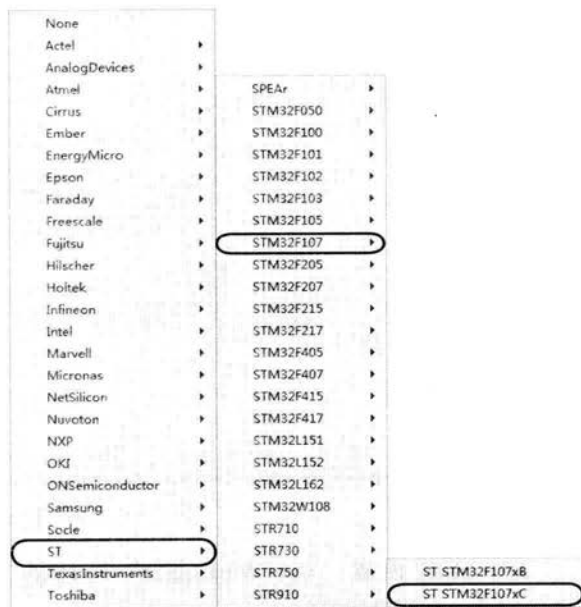


图 3.43 设置处理器类别

12) 配置头文件包含路径。图 3.42 中选择“C/C++ Compiler”，再选择“Preprocessor”选项卡，然后在“Additional include directories”的文本框中输入项目所需索引的头文件地址。该工程基于 ST 公司官方驱动库，按照官方库的要求，需要在“Defined symbols”中添加预编译条件，“USE_STDPERIPH_DRIVER”表示使用官方驱动库，等同于源程序文本中的“#define USE_STDPERIPH_DRIVER”；“STM32F10X_CL”表示所使用芯片是 STM32F10X 的互联型微控制器，等同于源程序文本中的“#define STM32F10X_CL”。

配置完成后如图 3.44 所示。

13) 在图 3.42 中选择“Linker”，再选择“Config”选项卡，勾选“Override Default”复选框，单击文本框右侧的文件选择按钮，在项目根目录中选择要连接配置的文件（例如，若准备将程序在 Flash 中运行，则选择 stm32f10x_flash.icf），配置完成如图 3.45 所示。

14) 选择“Linker”，再选择“List”选项卡，勾选“Generate linker map file”复选框，该选项用于指示生成链接映像文件，该文件主要包含了链接信息、程序起始地址信息、运行模式信息，以及目标文件占用代码、数据和常数空间的大小等，最后给出了在链接过程中出现的警告和错误信息。配置完成后如图 3.46 所示。

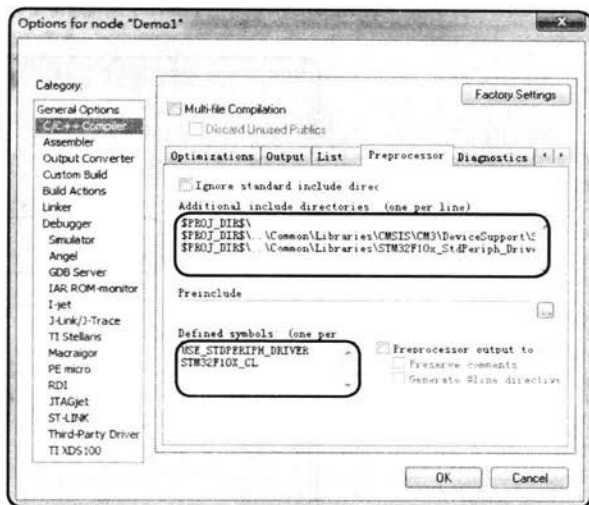


图 3.44 设置头文件包含路径

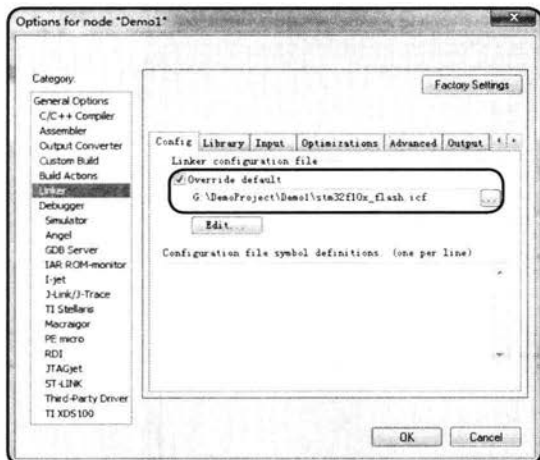


图 3.45 选择要连接配置的文件

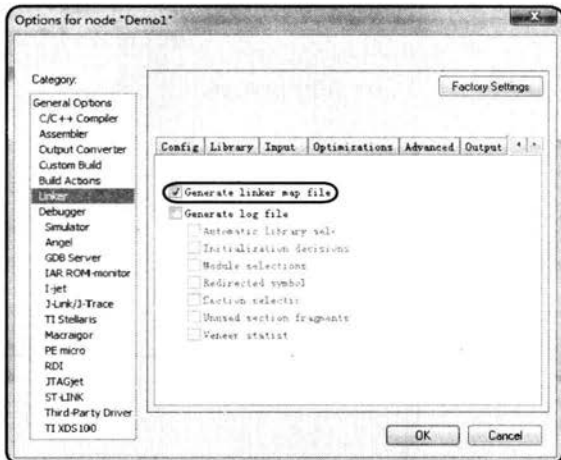


图 3.46 勾选“Generate linker map file”

15) 选择“Debugger”，在“Setup”选项卡的“Driver”下拉列表中选择仿真器的类型，本教

程中使用的是 J-Link，因此选中“J-Link/J-Trace”。若要想调试过程中程序从 main 函数处作为执行起点，则勾选“Run to”，并在下面的文本框中输入 main。配置完成后如图 3.47 所示。

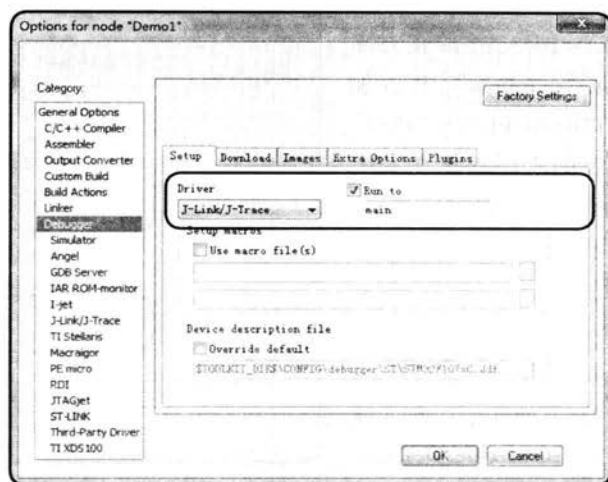


图 3.47 选择仿真器的类型

至此，整个工程已配置完毕，选择“Project”菜单中的“Make”，工程进行编译，如图 3.48 所示。

编译完成后，选择“Project”菜单中的“Download and Debug”，即可开始代码的调试，如图 3.49 所示。

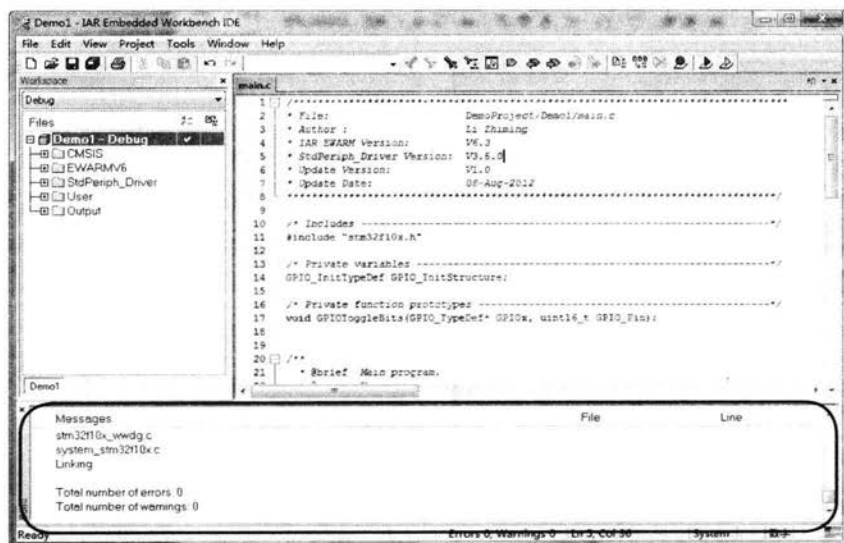


图 3.48 编译工程

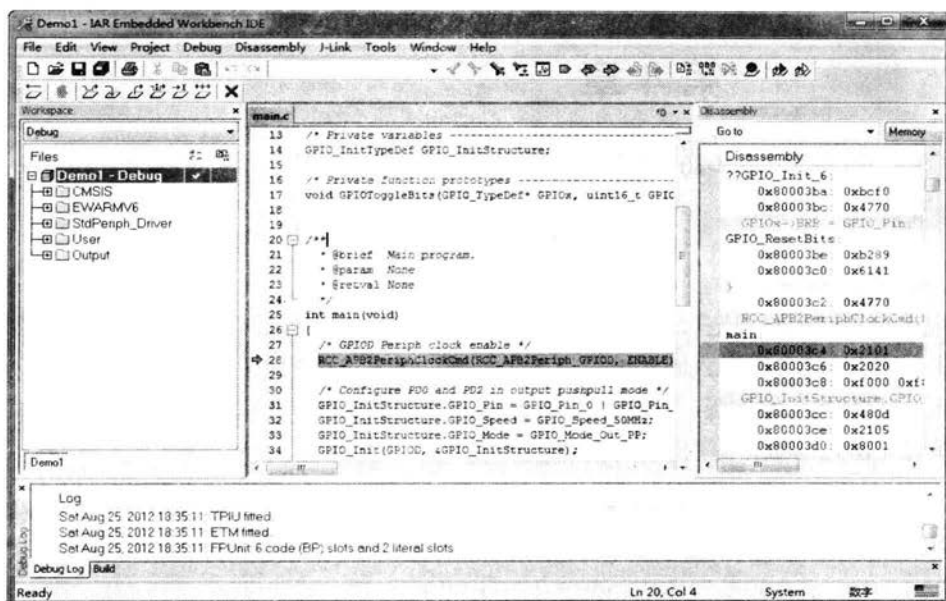
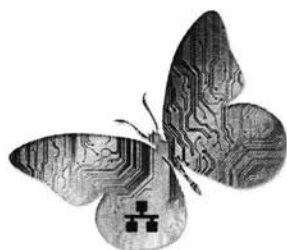


图 3.49 调试代码

至此，在 IAR EWARM 下的工程创建和配置已讲述完毕，关于 IAR EWARM 开发环境其他选项的设置方法可参考开发环境的帮助手册。本小节仅对工程实现的必要配置进行初步设置。



第 4 章

编程规范

编程规范也称“编码规范”，是为了提高代码质量和可维护性而对源代码的编写风格做出统一的规范约束。本节以 ST 嵌入式固件编程规范为基础，结合笔者的工程实践经验，简要介绍基于 C 语言的嵌入式编程规范的排版、注释、标识符命名、变量使用、代码可测性、程序效率、质量保证、代码编译、测试与维护等内容。

4.1 ST 固件库编程规范

4.1.1 缩写

ST 固件库中约定的外设 / 单元缩写见表 4.1。

表 4.1 固件库中约定的外设/单元缩写

缩 写	外设/单元
ADC	模数转换器
BKP	备份寄存器
CAN	控制器局域网模块
DMA	直接内存存取控制器
EXTI	外部中断事件控制器
FLASH	闪存存储器
GPIO	通用输入输出
I2C	内部集成电路
IWDG	独立看门狗
NVIC	嵌套中断向量列表控制器
PWR	电源 / 功耗控制
RCC	复位与时钟控制器
RTC	实时时钟
SPI	串行外设接口
SysTick	系统嘀嗒定时器

(续)

缩 写	外设/单元
TIM	通用定时器
TIM1	高级控制定时器
USART	通用同步 / 异步接收 / 发射端
WWDG	窗口看门狗

4.1.2 命名规则

- ❑ 系统、源程序文件和头文件命名都以“stm32f10x_”作为开头，例如：stm32f10x_conf.h。
- ❑ 常量仅被应用于一个文件的，定义于该文件中；被应用于多个文件的，在对应头文件中定义。所有常量的命名都采用英文字母大写形式。
- ❑ 寄存器作为常量处理。它们的命名都采用英文字母大写形式。
- ❑ 外设函数的命名以该外设的缩写加下划线为开头。每个单词的第一个字母均为英文字母大写形式，例如：SPI_SendData。在函数名中，只允许存在一个下划线，用以分隔外设缩写和函数名的其他部分。
- ❑ PPP_Init 函数的功能是根据 PPP_InitTypeDef 中指定的参数初始化外设 PPP，例如 TIM_Init。
- ❑ PPP_DeInit 函数的功能是复位外设 PPP 的所有寄存器至默认值，例如 TIM_DeInit。
- ❑ PPP_StructInit 函数的功能是通过设置 PPP_InitTypeDef 结构中的各种参数来定义外设的功能，例如 USART_StructInit。
- ❑ PPP_Cmd 函数的功能为使能或者失能外设 PPP，例如 SPI_Cmd。
- ❑ PPP_ITConfig 函数的功能为使能或者失能来自外设 PPP 的某中断源，例如 RCC_ITConfig。
- ❑ PPP_DMAConfig 函数的功能为使能或者失能外设 PPP 的 DMA 接口，例如 TIM1_DMAConfig。
- ❑ 用以配置外设功能的函数总是以字符串“Config”结尾，例如 GPIO_PinRemapConfig。
- ❑ PPP_GetFlagStatus 函数的功能为检查是否设置外设 PPP 的某标志位，例如 I2C_GetFlagStatus。
- ❑ PPP_ClearFlag 函数的功能为清除外设 PPP 的标志位，例如 I2C_ClearFlag。
- ❑ PPP_GetITStatus 函数的功能为判断来自外设 PPP 的中断是否发生，例如 I2C_GetITStatus。
- ❑ PPP_ClearITPendingBit 函数的功能为清除外设 PPP 的中断待处理标志位，例如 I2C_ClearITPendingBit。

4.1.3 编码规则

1. 变量

固件库定义了 24 个变量类型，并且它们的类型和大小是固定的。文件 stm32f10x_type.h 定义了如下这些变量：

```

typedef signed long s32;
typedef signed short s16;
typedef signed char s8;
typedef signed long const sc32; /* Read Only */
typedef signed short const sc16; /* Read Only */
typedef signed char const sc8; /* Read Only */
typedef volatile signed long vs32;
typedef volatile signed short vs16;
typedef volatile signed char vs8;
typedef volatile signed long const vsc32; /* Read Only */
typedef volatile signed short const vsc16; /* Read Only */
typedef volatile signed char const vsc8; /* Read Only */
typedef unsigned long u32;
typedef unsigned short u16;
typedef unsigned char u8;
typedef unsigned long const uc32; /* Read Only */
typedef unsigned short const uc16; /* Read Only */
typedef unsigned char const uc8; /* Read Only */
typedef volatile unsigned long vu32;
typedef volatile unsigned short vu16;
typedef volatile unsigned char vu8;
typedef volatile unsigned long const vuc32; /* Read Only */
typedef volatile unsigned short const vuc16; /* Read Only */
typedef volatile unsigned char const vuc8; /* Read Only */

```

2. 布尔型

文件 stm32f10x_type.h 定义如下，布尔型变量：

```

typedef enum
{
    FALSE = 0,
    TRUE = !FALSE
} bool;

```

3. 标志位状态类型

文件 stm32f10x_type.h 定义了标志位类型 (FlagStatus type) 的 2 个可能值，分别为“设置”与“重置” (SET 与 RESET)。

```

typedef enum
{
    RESET = 0,
    SET = !RESET
} FlagStatus;

```

4. 功能状态类型

文件 stm32f10x_type.h 定义了功能状态类型 (FunctionalState type) 的 2 个可能值，分别为“使能”与“失能” (ENABLE 与 DISABLE)。

```

typedef enum
{

```

```

    DISABLE = 0,
    ENABLE = !DISABLE
} FunctionalState;

```

5. 错误状态类型

文件 `stm32f10x_type.h` 定义了错误状态类型 (`ErrorStatus type`) 的 2 个可能值, 分别为“成功”与“出错” (`SUCCESS` 与 `ERROR`)。

```

typedef enum
{
    ERROR = 0,
    SUCCESS = !ERROR
} ErrorStatus;

```

6. 外设

用户可以通过指向各个外设的指针访问各个外设的控制寄存器。这些指针所指向的数据结构与各个外设的控制寄存器布局一一对应。

外设控制寄存器结构文件 `stm32f10x_map.h` 包含了所有外设控制寄存器的结构, 如下所示为 SPI 寄存器结构的声明:

```

/*----- Serial Peripheral Interface -----*/
typedef struct
{
    vu16 CR1;
    u16 RESERVED0;
    vu16 CR2;
    u16 RESERVED1;
    vu16 SR;
    u16 RESERVED2;
    vu16 DR;
    u16 RESERVED3;
    vu16 CRCPR;
    u16 RESERVED4;
    vu16 RXCRCR;
    u16 RESERVED5;
    vu16 TXCRCR;
    u16 RESERVED6;
} SPI_TypeDef;

```

寄存器命名遵循 4.1.2 节的寄存器缩写命名规则。RESERVED_i (_i 为一个整数索引值) 表示被保留区域。

文件 `stm32f10x_map.h` 包含了所有外设的声明, 如下所示为 SPI 外设的声明:

```

#ifndef EXT
#define EXT extern
#endif
...
#define PERIPH_BASE ((u32)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE

```

```

#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000) ...

/* SPI2 Base Address definition*/
#define SPI2_BASE (APB1PERIPH_BASE + 0x3800) ...
/* SPI2 peripheral declaration*/
#ifndef DEBUG
...
#endif
#ifdef _SPI2
#define SPI2 ((SPI_TypeDef *) SPI2_BASE)
#endif /* _SPI2 */
...
#else /* DEBUG */
...
#endif
#ifdef _SPI2
EXT SPI_TypeDef *SPI2;
#endif /* _SPI2 */
...
#endif /* DEBUG */

```

如果读者希望使用外设 SPI，那么必须在文件 stm32f10x_conf.h 中定义 _SPI 标签。通过定义标签 _SPIn，用户可以访问外设 SPIn 的寄存器。例如，用户必须在文件 stm32f10x_conf.h 中定义标签 _SPI2，否则不能访问 SPI2 的寄存器。在文件 stm32f10x_conf.h 中，读者可以按照如下所示定义标签 _SPI 和 _SPIn。

```

#define _SPI
#define _SPI1
#define _SPI2

```

每个外设都专门为标志位分配了若干寄存器。我们可以按照相应的结构定义这些寄存器。标志位的命名同样遵循 4.1.2 节的外设缩写规范，以“PPP_FLAG_”开始。对于不同的外设，标志位都在相应的文件 stm32f10x_ppp.h 中定义。

若读者想要进入除错（DEBUG）模式，则必须在文件 stm32f10x_conf.h 中定义标签 DEBUG。这样会在 SRAM 的外设结构部分创建一个指针。因此我们可以简化除错过程，并且通过转储外设来获得所有寄存器的状态。在所有情况下，SPI2 都是一个指向外设 SPI2 首地址的指针。

变量 DEBUG 可以仿照如下所示进行定义：

```

#define DEBUG 1

```

初始化 DEBUG 模式与文件 stm32f10x_lib.c 的代码如下所示：

```

#ifdef DEBUG void debug(void)
{
...
#ifdef _SPI2
SPI2 = (SPI_TypeDef *) SPI2_BASE;
#endif /* _SPI2 */
...
}
#endif /* DEBUG*/

```

注意：

- ❑ 当用户选择 DEBUG 模式时，将扩展宏 `assert_param`，同时激活固件库中的运行时间检查功能。
- ❑ 进入 DEBUG 模式会增大代码的尺寸，降低代码的运行效率。因此，我们强烈建议仅仅在除错的时候使用相应代码，并在最终的应用程序中删除它们。

4.2 基于 C 语言的嵌入式编程规范

4.2.1 源代码的排版

- ❑ 程序块要采用缩进风格编写，建议缩进的空格数以 4 个为宜。
- ❑ 相对独立的程序块之间、变量说明之后必须加空行，并加注其功能。

示例：如下例子不符合规范。

```
if (!valid_ni(ni))
{
    ... // program code
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

应按如下格式书写：

```
if (!valid_ni(ni))
{
    ... // program code
}

/* Initialize the response value */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

- ❑ 较长的语句（大于 80 个字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要适当缩进，使排版整齐，语句可读。

示例：

```
perm_count_msg.head.len = NO7_TO_STAT_PERM_COUNT_LEN
    + STAT_SIZE_PER_FRAM * sizeof( _UL );

act_task_table[frame_id * STAT_TASK_CHECK_NUMBER + index].occupied
= stat_poi[index].occupied;

act_task_table[taskno].duration_true_or_false
    = SYS_get_sccp_statistic_state( stat_item );

report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
&& (n7stat_stat_item_valid (stat_item))
&& (act_task_table[taskno].result_data != 0));
```

- ❑ 循环、判断等语句中若有较长的表达式或语句，则要适应划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首。

示例：

```
if ((taskno < max_act_task_number)
    && (n7stat_stat_item_valid (stat_item)))
{
    ... // program code
}

for (i = 0, j = 0; (i < BufferKeyword[word_index].word_length)
    && (j < NewKeyword.word_length); i++, j++)
{
    ... // program code
}

for (i = 0, j = 0;
     (i < first_word_length) && (j < second_word_length);
     i++, j++)
{
    ... // program code
}
```

- ❑ 若函数或过程中的参数较长，则要适当划分。

示例：

```
n7stat_str_compare((BYTE *) & stat_object,
                  (BYTE *) & (act_task_table[taskno].stat_object),
                  sizeof (_STAT_OBJECT));

n7stat_flash_act_duration( stat_item,
frame_id STAT_TASK_CHECK_NUMBER+index,
stat_object );
```

- ❑ 不允许把多个短语句写在一行中，即一行只写一条语句。

示例：如下例子不符合规范。

```
rect.length = 0; rect.width = 0;
```

应按如下格式书写：

```
rect.length = 0;
rect.width  = 0;
```

- ❑ if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号 {}。

示例：如下例子不符合规范。

```
if (pUserCR == NULL) return;
```

应按如下格式书写：

```
if (pUserCR == NULL)
{
    return;
}
```

❑ 对齐时只使用空格键，不使用 TAB 键。

说明 以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐，不要使用 BC 作为编辑器合版本，因为 BC 会自动将 8 个空格变为一个 TAB 键，因此使用 BC 合入的版本大多会将缩进格式变乱。

- ❑ 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case 语句下的情况处理语句也要遵从语句缩进要求。
- ❑ 程序块的分界符（如 C/C++ 语言的大括号 { 和 }）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及 if、for、do、while、switch、case 语句中的程序都要采用如上的缩进方式。

示例：如下例子不符合规范。

```
for (...) {
    ... // program code
}

if (...)
{
    ... // program code
}

void example_fun( void )
{
    ... // program code
}
```

应按如下格式书写：

```
for (...)
{
    ... // program code
}

if (...)
{
    ... // program code
}

void example_fun( void )
{
    ... // program code
}
```

- 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如 $->$ ），其后不应加空格。

说明 采用这种松散方式编写代码的目的是使代码更加清晰易读。

由于留空格所产生的清晰性是相对的，因此，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧（左括号后面和右括号前面）不需要加空格，多重括号间不必加空格，因为在 C/C++ 语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例 1：逗号、分号只在后面加空格。

```
int a, b, c;
```

示例 2：比较操作符，赋值操作符“=”、“+=”，算术操作符“+”、“%”，逻辑操作符“&&”、“&”，位域操作符“<<”、“^”等双目操作符的前后加空格。

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

示例 3：“!”、“~”、“++”、“--”、“&”（地址运算符）等单目操作符前后不加空格。

```
*p = 'a';           // 内容操作“*”与内容之间
flag = !isEmpty;    // 非操作“!”与内容之间
p = &mem;           // 地址操作“&”与内容之间
i++;                // “++”、“--”与内容之间
```

示例 4：“->”、“.”前后不加空格。

```
p->id = pid;         // “->”指针前后不加空格
```

示例 5：if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显。

```
if (a >= b && c > d)
```

- 一程序以小于 80 字符为宜，不要写得过长。

4.2.2 源代码的注释

- 一般情况下，源程序应尽量简洁明了。

说明 注释的原则是有助于对程序的阅读理解，注释应准确、易懂、简洁，此外为兼容各种编译环境，注释语言宜采用英文注释（为了便于读者理解，本规范中的示例源代码采用中文注释）。

- 说明性文件（如头文件 .h 文件、.icf 文件、.txt 文件等）头部应进行注释，注释必须列出：

版权说明、版本号、生成日期、作者、内容、功能、与其他文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```

/*****
Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
File name:                               // 文件名
Author:      Version:      Date:      // 作者、版本及完成日期
Description:      // 用于详细说明此程序文件完成的主要功能，与其他模块
                  // 或函数的接口，输出值、取值范围、含义及参数间的控
                  // 制、顺序、独立或依赖等关系
Others:          // 其他内容的说明
Function List:   // 主要函数列表，每条记录应包括函数名及功能简要说明
    1. ....
History:         // 修改历史记录列表，每条修改记录应包括修改日期、修改
                  // 者以及修改内容简述
    1. Date:
        Author:
        Modification:
    2. ...
*****/

```

❑ 源文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、模块目的 / 功能、主要函数及其功能、修改日志等。

示例：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```

/*****
Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
FileName: test.cpp
Author:      Version :      Date:
Description:      // 模块描述
Version:         // 版本信息
Function List:    // 主要函数及其功能
    1. -----
History:         // 历史修改记录
    <author> <time> <version > <desc>
    David   96/10/12   1.0   build this moudle
*****/

```

说明 Description 一项描述本文件的内容、功能、内部各部分之间的关系及本文件与其他文件的关系等。History 是修改历史记录列表，每条修改记录应包括修改日期、修改者及修改内容简述。

❑ 函数头部应进行注释，注释必须列出：函数的目的 / 功能、输入参数、输出参数、返回值、调用关系（函数、表）等。

示例：下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```

/*****
Function:           // 函数名称
Description:        // 函数功能、性能等的描述
Calls:              // 被本函数调用的函数清单
Called By:          // 调用本函数的函数清单
Table Accessed:     // 被访问的表（此项仅对于牵扯到数据库操作的程序）
Table Updated:      // 被修改的表（此项仅对于牵扯到数据库操作的程序）
Input:              // 输入参数说明，包括每个参数的作
                   // 用、取值说明及参数间关系
Output:             // 对输出参数的说明
Return:            // 函数返回值的说明
Others:            // 其他说明
*****/

```

- ❑ 注释应与编写代码同步，修改代码同时修改相应的注释，以保证注释与代码的一致性。
- ❑ 注释的内容要清楚明了，含义准确，避免产生歧义。
- ❑ 避免在注释中使用缩写，特别是非常用缩写，命名的标识符除外。

说明 在使用缩写时或之前，应对缩写进行必要的说明。

- ❑ 注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

如下例子不符合规范。

示例 1：

```

/* get replicate sub system index and net indicator */

repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;

```

示例 2：

```

repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */

```

应按如下格式书写：

```

/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;

```

- ❑ 对于所有具有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方或右方相邻位置。

示例：

```

/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000

#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */

```

- 数据结构声明（包括数组、结构、枚举等），如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

示例：可按如下形式说明枚举、数据、联合结构。

```
/* sccp interface with sccp user primitive message name */
enum SCCP_USER_PRIMITIVE
{
    N_UNITDATA_IND,      /* sccp notify sccp user unit data come */
    N_NOTICE_IND,        /* sccp notify user the No.7 network can not
                        */
    /* transmission this message */
    N_UNITDATA_REQ,      /* sccp user's unit data transmission
                        request*/
};
```

- 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

示例：

```
/******
 * Variable function:
 *   The ErrorCode when SCCP translate Global Title failure, as
 * follows
 * Available value:
 *   0 - SUCCESS
 *   1 - GT Table error
 *   2 - GT error
 *   Others - no use
 * Call relationship:
 *   only function SCCPTranslate() in this modual can modify
 *   it, and other module can visit it through call the function
 *   GetGTTransErrorCode()
 *****/
BYTE g_GTTranErrorCode;
```

- 注释与所描述内容进行同样的缩排。

说明 可使程序排版整齐，并方便注释的阅读与理解。

示例：如下例子的排版格式不整齐，阅读稍感不方便。

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

    /* code two comments */
    CodeBlock Two
}
```

应改为如下布局：

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

    /* code two comments */
    CodeBlock Two
}
```

❑ 将注释与其上面的代码用空行隔开。

示例：如下例子中的代码显得过于紧凑。

```
/* code one comments */
program code one
/* code two comments */
program code two
```

应按如下格式书写：

```
/* code one comments */
program code one

/* code two comments */
program code two
```

❑ 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

说明 这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释可以帮助读者更好地理解程序，有时甚至优于阅读设计文档。

❑ 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

说明 这样比较清晰地说明程序编写者的意图，有效防止无故遗漏 break 语句。

示例：

```
case CMD_UP:
    ProcessUp();
    break;

case CMD_DOWN:
    ProcessDown();
    break;

case CMD_FWD:
    ProcessFwd();

    if (...)
```

```

    {
    ...
    break;
    }
    else
    {
    ProcessCFW_B();` // now jump into case CMD_A
    }

    case CMD_A:
        ProcessA();
        break;

    case CMD_B:
        ProcessB();
        break;

    case CMD_C:
        ProcessC();
        break;

    case CMD_D:
        ProcessD();
        break;

    ...

```

- ❑ 避免在一行代码或表达式的中间插入注释。

说明 除非有必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

- ❑ 通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。

说明 命名清晰准确的函数、变量等，可增加代码可读性，并减少不必要的注释。

- ❑ 在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明 注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

```

/* if receive_flag is TRUE */
if (receive_flag)

```

而如下的注释则给出了额外有用的信息。

```

/* if mtp receive a message from links */
if (receive_flag)

```

- ❑ 在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明 当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

示例：如下例子展示了良好的格式。

```
if (...)
{
    // program code

    while (index < MAX_INDEX)
    {
        // program code
    } /* end of while (index < MAX_INDEX) */    // 指明该条 while 语句结束
} /* end of if (...) */                        // 指明是哪条 if 语句结束
```

- ❑ 注释格式尽量统一，建议使用“/* */”。
- ❑ 注释应考虑程序易读及外观排版的因素，使用的语言若是中英兼有的，建议使用中文，除非能用非常流利准确的英文进行表达。

说明 注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文进行注释。

4.2.3 标识符命名

- ❑ 标识符的命名要清晰明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

说明 较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp。

flag 可缩写为 flg。

statistic 可缩写为 stat。

increment 可缩写为 inc。

message 可缩写为 msg。

- ❑ 命名中若使用特殊约定或缩写，则要有注释说明。

说明 应该在源文件的开始之处对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

- ❑ 自己特有的命名风格，要自始至终保持一致，不可来回变化。

说明 个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用。（即命名规则中没有规定的地方才可使用个人命名风格。）

- ❑ 对于变量命名，禁止取单个字符（如 i、j、k 等），建议除了要有具体含义外，还能表明其变量类型、数据类型等，但 i、j、k 作局部循环变量是允许的。

说明 变量，尤其是局部变量，如果用单个字符表示，很容易输入错误（如将 i 写成 j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

示例：如下所示的局部变量名的定义方法可以借鉴。

```
int liv_Width;
```

其变量名解释如下：

- l 局部变量 (Local) (其他：g 全局变量 (Global) 等)
- i 数据类型 (Integer)
- v 变量 (Variable) (其他：c 常量 (Const) 等)
- Width 变量含义

这样可以防止局部变量与全局变量重名。

- ❑ 命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用 UNIX 的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的 m_ 和 g_，其后加上大小写混排的方式是允许的。

示例：不允许使用 Add_User，但 add_user、AddUser、m_AddUser 是允许使用的。

- ❑ 除非有必要，否则不要用数字或较奇怪的字符来定义标识符。

示例：如下命名可能使人产生疑惑。

```
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
void set_sls00( BYTE sls );
```

应改为有意义的单词命名：

```
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void set_udt_msg_sls( BYTE sls );
```

- ❑ 在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明 对接口部分的标识符应该有更严格的限制，防止冲突。如可规定接口部分的变量与常量之前加上“模块”标识等。

- ❑ 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

说明 下面是一些在软件中常用的反义词组。

add/remove	begin/end	create/destroy
insert/delete	first/last	get/release
increment/decrement	put/get	add/delete
lock/unlock	open/close	min/max
old/new	start/stop	next/previous
source/target	show/hide	send/receive
source/destination	cut/paste	up/down

示例：

```
int min_sum;
int max_sum;
int add_user( BYTE *user_name );
int delete_user( BYTE *user_name );
```

- ❑ 除了编译开关、头文件等特殊应用，应避免使用 `_EXAMPLE_TEST_` 之类以下划线开始和结尾的定义。

4.2.4 代码可读性

- ❑ 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明 防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：在下列表达式中，

```
word = (high << 8) | low      (1)
if ((a | b) && (a & c))      (2)
if ((a | b) < (c & d))      (3)
```

如果书写为：

```
high << 8 | low
a | b && a & c
a | b < c & d
```

由于

```
high << 8 | low = ( high << 8) | low,
a | b && a & c = (a | b) && (a & c),
```

因此 (1)(2) 不会出错，但语句不易理解：

```
a | b < c & d = a | (b < c) & d.
```

语句 (3) 造成了判断条件出错。

- ❑ 避免使用不易理解的数字，应该用有意义的标识来替代。涉及物理状态或者含有物理意义

的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)
{
    Trunk[index].trunk_state = 1;
    ... // program code
}
```

应改为如下形式：

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    ... // program code
}
```

❑ 源程序中关系较为紧密的代码应尽可能相邻。

说明 便于程序阅读和查找。

示例：以下代码布局不太合理。

```
rect.length = 10;
char_poi = str;
rect.width = 5;
```

若按如下形式书写，可能更清晰一些。

```
rect.length = 10;
rect.width = 5; // 矩形的长与宽关系较密切，放在一起。
char_poi = str;
```

❑ 不要使用不易理解的技巧性很高的语句，除非很有必要时。

说明 高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

示例：在如下表达式中，若考虑不周全就可能出问题，也较难理解。

```
* stat_poi ++ += 1;

* ++ stat_poi += 1;
```

应分别改为如下形式：

```
*stat_poi += 1;
stat_poi++; // 此二语句功能相当于 “ * stat_poi ++ += 1; ”

++ stat_poi;
*stat_poi += 1; // 此二语句功能相当于 “ * ++ stat_poi += 1; ”
```

4.2.5 变量、结构

- 尽可能少使用公共变量。

说明 公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间的耦合度。

- 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

说明 在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其他变量的关系。

- 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

说明 明确过程操作变量的关系后，有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明：

RELATION	System_Init	Input_Rec	Print_Rec	Stat_Score	
Student		Create	Modify	Access	Access
Score		Create	Modify	Access	Access, Modify

注意 RELATION 为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score 为四个不同的函数；Student、Score 为两个全局变量；Create 表示创建，Modify 表示修改，Access 表示访问。

其中，函数 Input_Rec、Stat_Score 都可修改变量 Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

- 当向公共变量传递数据时，要十分小心，防止发生赋予不合理的值或越界等现象。

说明 对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

- 防止局部变量与公共变量同名。

说明 若使用了较好的命名规则，那么此问题可自动消除。

- 严禁使用未经初始化的变量作为右值。

说明 特别是在 C/C++ 中引用未经赋值的指针，经常会引起系统崩溃。

- 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只能访问的公共变量，防止出现多个不同模块或函数都可以修改、创建同一公共变量的现象。

说明 降低公共变量耦合度。

- 使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切

的变量。

说明 使用标准的数据类型，有利于程序的移植。

示例：如下例子（在 DOS 下 BC3.1 环境中）在移植时可能产生问题。

```
void main()
{
    register int index; // 寄存器变量

    _AX = 0x4000;        // _AX 是 BC3.1 提供的寄存器“伪变量”
    ... // program code
}
```

❑ 结构的功能要单一，只是针对一种事务的抽象。

说明 设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下所示结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
    unsigned char name[8]; /* student's name */
    unsigned char age;     /* student's age */
    unsigned char sex;     /* student's sex, as follows */
                          /* 0 - FEMALE; 1 - MALE */
    unsigned char
        teacher_name[8]; /* the student teacher's name */
    unsigned char
        teacher_sex;     /* his teacher sex */
} STUDENT;
```

若改为如下形式，可能更合理些。

```
typedef struct TEACHER_STRU
{
    unsigned char name[8]; /* teacher name */
    unsigned char sex;     /* teacher sex, as follows */
    /* 0 - FEMALE; 1 - MALE */
} TEACHER;

typedef struct STUDENT_STRU
{
    unsigned char name[8]; /* student's name */
    unsigned char age;     /* student's age */
    unsigned char sex;     /* student's sex, as follows */
    /* 0 - FEMALE; 1 - MALE */
    unsigned int teacher_ind; /* his teacher index */
} STUDENT;
```

❑ 不要设计面面俱到、非常灵活的数据结构。

说明 面面俱到、灵活的数据结构反而容易引起误解和操作困难。

❑ 不同结构间的关系不要过于复杂。

说明 若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：如下所示两个结构的构造不合理。

```
typedef struct PERSON_ONE_STRU
{
    unsigned char name[8];
    unsigned char addr[40];
    unsigned char sex;
    unsigned char city[15];
} PERSON_ONE;
```

```
typedef struct PERSON_TWO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char tel;
} PERSON_TWO;
```

由于两个结构都是描述同一事物的，那么不如合成一个结构。

```
typedef struct PERSON_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON;
```

❑ 结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。

说明 增加结构的可理解性、可操作性和可维护性。

示例：假如认为如上的 PERSON 结构元素过多，那么可按如下方式对之划分：

```
typedef struct PERSON_BASE_INFO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
} PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
{
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
```

```

        unsigned char addr[40];
        unsigned char city[15];
        unsigned char tel;
    } PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO person_base;
    PERSON_ADDRESS person_addr;
} PERSON;

```

- ❑ 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

说明 合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：如下所示结构中的位域排列将占用较大空间，可读性稍差。

```

typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    PERSON person;
    unsigned int set_flg: 1;
} EXAMPLE;

```

若改成如下形式，不仅可节省 1 字节空间，可读性也较好。

```

typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    unsigned int set_flg: 1;
    PERSON person ;
} EXAMPLE;

```

- ❑ 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。

说明 软件向前兼容的特性是软件产品是否成功的重要标志之一。如果要想使产品具有较好的前向兼容性，那么在产品设计之初就应为以后版本升级保留一定余地，并且在产品升级时必须考虑前一版本的各种特性。

- ❑ 留心具体语言及编译器处理不同数据类型的原则及有关细节。

说明 如在 C 语言中，static 局部变量将在内存“数据区”中生成，而非 static 局部变量将在“堆栈”中生成。这些细节对程序质量的保证非常重要。

- ❑ 编程时，要注意数据类型的强制转换。

说明 当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

□ 对编译系统默认的数据类型转换，也要有充分的认识。

示例：如下所示赋值语句，多数编译器不产生警告，但值的含义还是稍有变化。

```
char chr;
unsigned short int exam;

chr = -1;
exam = chr; // 编译器不产生警告，此时 exam 为 0xFFFF。
```

□ 尽量减少没有必要的数据类型默认转换与强制转换。

□ 合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

□ 对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

说明 使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

示例：可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁明了。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned int   DWORD;
```

下面的声明可使数据类型具有更丰富的含义。

```
typedef float DISTANCE;
typedef float SCORE;
```

□ 当声明用于分布式环境或不同 CPU 间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题。

说明 比如 Intel CPU 与 68360 CPU，在处理位域及整数时，其在内存存放的“顺序”正好相反。

4.2.6 函数、过程

□ 对所调用函数的错误返回码要仔细、全面处理。

□ 明确函数功能，精确（而不是近似）地实现函数设计。

□ 编写可重入函数时，应注意局部变量的使用（如编写 C/C++ 语言的可重入函数时，应使用 auto 即默认态局部变量或寄存器变量）。

说明 编写 C/C++ 语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

□ 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护。

说明 若对所使用的全局变量不加以保护, 则此函数就不具有可重入性, 即当多个进程调用此函数时, 很有可能使有关全局变量处于不可知状态。

示例: 假设 Exam 是 int 型全局变量, 函数 Square_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int example( int para )
{
    unsigned int temp;

    Exam = para; // (**)
    temp = Square_Exam( );

    return temp;
}
```

若上述函数被多个进程调用, 其结果可能是未知的, 因为当 (**) 语句刚执行完后, 另外一个使用本函数的进程可能正好被激活, 那么当新激活的进程执行到此函数时, 将使 Exam 赋予另一个不同的 para 值, 所以当控制重新回到 “temp = Square_Exam()” 后, 计算出的 temp 很可能不是预想中的结果。此函数应进行如下所示的改进:

```
unsigned int example( int para )
{
    unsigned int temp;

    [ 申请信号量操作 ]          /* 若申请不到“信号量”, 说明另外的进程正处
                                   于给 Exam 赋值并计算其平方过程中 (即正在使
                                   用此信号), 本进程必须等待其释放信号后,
                                   才可继续执行。若申请到信号, 则可继续执行,
                                   但其他进程必须等待本进程释放信号量后, 才
                                   能再用本信号。*/

    Exam = para;
    temp = Square_Exam( );
    [ 释放信号量操作 ]

    return temp;
}
```

- 在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责, 默认是由函数调用者负责。

说明 对于模块间接口函数的参数的合法性检查这一问题, 往往有两个极端现象, 要么是调用者和被调用者对参数均不进行合法性检查, 结果就遗漏了合法性检查这一必要的处理过程, 造成问题隐患; 要么就是调用者和被调用者均对参数进行合法性检查, 这种情况虽不会造成问题, 但产生了冗余代码, 降低了效率。

- 防止将函数的参数作为工作变量。

说明 将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对于必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：如下所示函数的实现过程不太好：

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count;

    *sum = 0;
    for (count = 0; count < num; count++)
    {
        *sum += data[count]; // sum成了工作变量，不太好。
    }
}
```

若改为如下形式，则更好些：

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count ;
    int sum_temp;

    sum_temp = 0;
    for (count = 0; count < num; count ++ )
    {
        sum_temp += data[count];
    }

    *sum = sum_temp;
}
```

□ 函数的规模尽量限制在 200 行以内。

说明 不包括注释和空格行。

□ 一个函数仅完成一项功能。

□ 为简单功能编写函数。

说明 虽然为仅用一两行就可完成的功能去编函数好像没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。

示例：如下语句的功能不是很明显：

```
value = ( a > b ) ? a : b ;
```

改为如下形式就很清晰了：

```
int max (int a, int b)
{
```

```
    return ((a > b) ? a : b);
}
```

```
value = max (a, b);
```

或改为如下形式：

```
#define MAX (a, b) (((a) > (b)) ? (a) : (b))
```

```
value = MAX (a, b);
```

❑ 不要设计多用途、面面俱到的函数。

说明 若将多功能集于一函数，很可能使函数的理解、测试、维护等变得困难。

❑ 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

说明 带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在 C/C++ 语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须将 STATIC 的局部变量的地址作为返回值，若为 auto 类，则返回为错误指针。

示例：如下所示函数的返回值（即功能）是不可预测的。

```
unsigned int integer_sum( unsigned int base )
{
    unsigned int index;
    static unsigned int sum = 0; // 注意，是 static 类型的。
                                // 若改为 auto 类型，则函数即变为可预测。
    for (index = 1; index <= base; index++)
    {
        sum += index;
    }

    return sum;
}
```

❑ 尽量不要编写依赖于其他函数内部实现的函数。

说明 此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，因此通过具体语言的语法要求与编译器功能，基本就可以防止发生这种情况。但在汇编语言中，由于其灵活性，很可能使函数出现这种情况。

示例：如下是在 DOS 下 TASM 的汇编程序例子。过程 Print_Msg 的实现依赖于 Input_Msg 的具体实现，这种程序是非结构化的，难以维护、修改。

```
...                // 程序代码
proc Print_Msg      // 过程（函数）Print_Msg
...                // 程序代码
    jmp LABEL
```

```

...                                // 程序代码
endp

proc Input_Msg                     // 过程（函数）Input_Msg
...                                // 程序代码
LABEL:
...                                // 程序代码
endp

```

❑ 避免设计多参数函数，不使用的参数从接口中删除。

说明 目的是减少函数间接口的复杂度。

❑ 非调度函数应减少或防止控制参数，尽量只使用数据参数。

说明 本建议的目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

示例：如下所示函数构造不太合理：

```

int add_sub( int a, int b, unsigned char add_sub_flg )
{
    if (add_sub_flg == INTEGER_ADD)
    {
        return (a + b);
    }
    else
    {
        return (a - b);
    }
}

```

不如分为如下两个函数清晰：

```

int add( int a, int b )
{
    return (a + b);
}

int sub( int a, int b )
{
    return (a - b);
}

```

❑ 检查函数所有参数输入的有效性。

❑ 检查函数所有非参数输入的有效性，如数据文件、公共变量等。

说明 函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

- 函数名应准确描述函数的功能。
- 使用动宾词组为执行某操作的函数命名。如果是 OOP 方法，可以只有动词（名词是对象本身）。

示例：参照如下方式命名函数：

```
void print_record( unsigned int rec_ind ) ;
int  input_record( void ) ;
unsigned char get_current_color( void ) ;
```

- 避免使用无意义或含义不清的动词为函数命名。

说明 避免使用含义不清的动词如 process、handle 等为函数命名，因为这些动词并没有说明要具体做什么。

- 函数的返回值要清楚明了，让使用者不易忽视错误情况。

说明 函数的每种出错返回值的意义要清晰明了和准确，防止使用者误用、理解错误或忽视错误返回码。

- 除非有必要，否则不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。
- 让函数在调用点显得易懂。
- 在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明 因为数据类型转换或多或少存在危险。

- 避免函数中不必要的语句，防止在程序中产生垃圾代码。

说明 程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

- 防止把没有关联的语句放到一个函数中。

说明 防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿意把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

示例：如下所示函数就是一种随机内聚。

```
void Init_Var( void )
```

```

{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */

    Point.x = 10;
    Point.y = 10; /* 初始化“点”的坐标 */
}

```

矩形的长、宽与点的坐标基本没有任何关系，因此以上函数是随机内聚。

上述函数应按如下形式分为两个函数：

```

void Init_Rect( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */
}

void Init_Point( void )
{
    Point.x = 10;
    Point.y = 10; /* 初始化“点”的坐标 */
}

```

□ 如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

说明 若此段代码各语句之间有实质性关联并且是完成同一项功能的，那么可考虑把此段代码构造成为一个新的函数。

□ 功能不明确、较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。

说明 模块中函数划分过多，一般会使函数间的接口变得复杂。所以功能过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

□ 设计高扇入、合理扇出（小于7）的函数。

说明 扇出是指一个函数直接调用（控制）其他函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构分析，并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是3~5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

- 减少函数本身或函数间的递归调用。

说明 递归调用，特别是函数间的递归调用（如 $A \rightarrow B \rightarrow C \rightarrow A$ ），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。因此，除非为某些算法或功能的实现方便，否则应减少没必要的递归调用。

- 仔细分析模块的功能及性能需求，并进一步细分，同时若有必要画出有关数据流图，据此来进行模块的函数划分与组织。

说明 函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或 / 和数据流图映射出函数结构是常用方法之一。

- 改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则：
 - 不能影响模块功能的实现。
 - 仔细考查模块或函数出错处理及模块的性能要求并进行完善。
 - 通过分解或合并函数来改进软件结构。
 - 考查函数的规模，过大的要进行分解。
 - 降低函数间接口的复杂度。
 - 不同层次的函数调用要有较合理的扇入、扇出。
 - 函数功能应可预测。
 - 提高函数内聚。（单一功能的函数内聚最高。）

说明 对初步划分后的函数结构应进行改进、优化，使之更合理。

- 在多任务操作系统的环境下编程，要注意函数可重入性的构造。

说明 可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户单任务操作系统。

- 避免使用 BOOL 参数。

说明 原因有二，其一是 BOOL 参数值无意义，TRUE/FALSE 的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其二是 BOOL 参数值不利于扩充。还有 NULL 也是一个无意义的单词。

- ❑ 对于提供了返回值的函数，在引用时最好使用其返回值。
- ❑ 当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。

说明 这样可以增加编程效率和程序的可读性。

示例：在某过程中较多引用 TheReceiveBuffer[FirstSocket].byDataPtr，则可以通过以下宏定义来代替：

```
# define pSOCKDATA TheReceiveBuffer[FirstSocket].byDataPtr
```

4.2.7 可测性

- ❑ 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应输出函数，并且要有详细说明。

说明 本规则是针对项目组或产品组的。

- ❑ 在同一项目组或产品组内，调测输出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。

说明 统一的调测信息格式便于集成测试。

- ❑ 编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过调测开关）。

说明 为单元测试而准备。

- ❑ 在进行集成测试 / 系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。

说明 好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

- ❑ 使用断言来发现软件问题，提高代码可测性。

说明 断言是指对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告），它可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其他手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。在实际应用时，可根据具体情况灵活地设计断言。

示例：下面是 C 语言中的一个断言，用宏来设计的。（其中 NULL 为 0L。）

```

#ifdef _EXAM_ASSERT_TEST_ // 若使用断言测试

void exam_assert( char * file_name, unsigned int line_no )
{
    printf( "\n[EXAM]Assert failed: %s, line %u\n",
file_name,
        line_no );
    abort( );
}

#define EXAM_ASSERT( condition )
    if (condition) // 若条件成立, 则无动作
        NULL;
    else // 否则报告
        exam_assert( __FILE__, __LINE__ )

#else // 若不使用断言测试

#define EXAM_ASSERT(condition) NULL

#endif /* end of ASSERT */

```

- ☐ 用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。
- ☐ 不能用断言来检查最终产品肯定会出现且必须处理的错误情况。

说明 断言是用来处理不应该发生的错误情况的, 对于可能会发生的且必须处理的情况要写防错程序, 而不是断言。如某模块收到其他模块或链路上的消息后, 要对消息的合理性进行检查, 此过程为正常的错误检查, 不能用断言来实现。

- ☐ 对较复杂的断言加上明确的注释。

说明 为复杂的断言加注释, 可澄清断言含义并减少不必要的误用。

- ☐ 用断言确认函数的参数。

示例: 假设某函数参数中有一个指针, 那么使用指针前可对它检查, 如下所示:

```

int exam_fun( unsigned char *str )
{
    EXAM_ASSERT( str != NULL ); // 用断言检查“假设指针不为空”这个条件

    ... //other program code
}

```

- ☐ 用断言保证没有定义的特性或功能不可用。

示例: 假设某通信模块在设计时, 准备提供“无连接”和“连接”这两种业务。但当前的版本中仅实现了“无连接”业务, 且在此版本的正式发行版中, 用户(上层模块)不应产生“连接”业务的请求, 那么在测试时可用断言检查用户是否使用“连接”业务。如下所示:

```

#define EXAM_CONNECTIONLESS 0           // 无连接业务

#define EXAM_CONNECTION      1           // 连接业务

int msg_process( EXAM_MESSAGE *msg )
{
    unsigned char service; /* message service class */

    EXAM_ASSERT( msg != NULL );

    service = get_msg_service_class( msg );

    EXAM_ASSERT( service != EXAM_CONNECTION ); // 假设不使用连接业务

    ... //other program code
}

```

❑ 用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查。

说明 针对程序运行时所需的软硬件环境及配置要求，不能用断言来检查，而必须由一段专门代码处理。断言仅用于对程序开发环境中的假设及所配置的某版本软硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了，应由程序中的正式代码来检查；而此网卡是否具有某设想的功能，则可由断言来检查。

对编译器提供的功能及特性假设可用断言检查，原因是软件最终产品（即运行代码或机器码）与编译器已没有任何直接关系，即软件运行过程中（注意不是编译过程中）不会也不应该对编译器的功能提出任何需求。

示例：用断言检查编译器的 int 型数据占用的内存空间是否为 2，如下所示：

```
EXAM_ASSERT( sizeof( int ) == 2 );
```

❑ 正式软件产品中应把断言及其他调测代码删除（即把有关的调测开关关闭）。

说明 加快软件运行速度。

❑ 在软件系统中设置与取消有关测试手段，不能对软件实现的功能等产生影响。

说明 即有测试代码的软件和关闭测试代码的软件在功能行为上应一致。

- ❑ 用调测开关来切换软件的 DEBUG 版和正式版，而不要同时存在正式版本和 DEBUG 版本的不同源文件，以减少维护的难度。
- ❑ 软件的 DEBUG 版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。
- ❑ 在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如输出函数等。

说明 程序的调试与测试是软件生存周期中很重要的一个阶段，如何对软件进行较全面、高率的测试并尽可能地找出软件中的错误就成为很关键的一个问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列代码测试手段，为单元测试、集成测试及系统联调提供方便。

□ 调测开关应分为不同级别和类型。

说明 调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于某种其他目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

□ 编写防错程序，然后在处理错误之后可用断言宣布发生错误。

示例：假如某模块收到通信链路上的消息，则应对消息的合法性进行检查，若消息类别不是通信协议中规定的，则应进行出错处理，之后可用断言报告，如下例所示：

```
#ifdef _EXAM_ASSERT_TEST_ // 若使用断言测试

/* Notice: this function does not call 'abort' to exit program */
void assert_report( char * file_name, unsigned int line_no )
{
    printf( "\n[EXAM]Error Report: %s, line %u\n",
file_name,
            line_no );
}

#define ASSERT_REPORT( condition )
    if ( condition ) // 若条件成立，则无动作
        NULL;
    else // 否则报告
        assert_report ( __FILE__, __LINE__ )

#else // 若不使用断言测试

#define ASSERT_REPORT( condition ) NULL

#endif /* end of ASSERT */

int msg_handle( unsigned char msg_name, unsigned char * msg )
{
    switch( msg_name )
    {
        case MSG_ONE:
            ... // 消息 MSG_ONE 处理
            return MSG_HANDLE_SUCCESS;

            ... // 其他合法消息处理

        default:
```

```

        ... // 消息出错处理
        ASSERT_REPORT( FALSE ); // “合法”消息不成立, 报告
        return MSG_HANDLE_ERROR;
    }
}

```

4.2.8 程序效率

□ 编程时要经常注意代码的效率。

说明 代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

□ 在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

说明 不能因一味地追求代码效率而对软件的正确性、稳定性、可读性及可测性造成影响。

□ 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

□ 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明 这种方式是解决软件空间效率的根本办法。

示例：如下记录学生学习成绩的结构不合理：

```

typedef unsigned char  BYTE;
typedef unsigned short WORD;

typedef struct STUDENT_SCORE_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
    BYTE subject;
    float score;
} STUDENT_SCORE;

```

因为每位学生都有多科学习成绩，所以如上结构将占用较大空间。应按如下方式进行改进（分为两个结构），总的存储空间将变小，操作也变得更方便。

```

typedef struct STUDENT_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
} STUDENT;

```

```
typedef struct STUDENT_SCORE_STRU
{
    WORD student_index;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

❑ 循环体内工作量最小化。

说明 应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的执行效率。

示例：如下代码效率较低：

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
    back_sum = sum; /* backup sum */
}
```

语句“back_sum = sum;”完全可以放在 for 语句之后，如下所示：

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
}
back_sum = sum; /* backup sum */
```

- ❑ 仔细分析有关算法，并进行优化。
- ❑ 仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。
- ❑ 对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序执行效率。

说明 软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

- ❑ 编程时，要随时留心代码效率；优化代码时，要考虑周全。
- ❑ 不应花过多的时间提高调用不频繁的函数的代码效率。

说明 对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

- ❑ 要仔细地构造或直接汇编编写调用频繁或性能要求极高的函数。

说明 只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

- ❑ 在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量来提高空间效率。

说明 这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

❑ 在多重循环中，应将执行次数最多的循环放在最内层。

说明 减少 CPU 切入循环层的次数。

示例：如下代码效率较低：

```
for (row = 0; row < 100; row++)
{
    for (col = 0; col < 5; col++)
    {
        sum += a[row][col];
    }
}
```

可以改为如下方式，以提高效率：

```
for (col = 0; col < 5; col++)
{
    for (row = 0; row < 100; row++)
    {
        sum += a[row][col];
    }
}
```

❑ 尽量减少循环嵌套层次。

❑ 避免在循环体内包含判断语句，应将循环语句置于判断语句的代码块中。

说明 目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

示例：如下代码效率稍低：

```
for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
{
    if (data_type == RECT_AREA)
    {
        area_sum += rect_area[ind];
    }
    else
    {
        rect_length_sum += rect[ind].length;
        rect_width_sum += rect[ind].width;
    }
}
```

因为判断语句与循环变量无关，可进行如下改进，以减少判断次数：

```
if (data_type == RECT_AREA)
{
    for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
```

```

        {
            area_sum += rect_area[ind];
        }
    }
    else
    {
        for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
        {
            rect_length_sum += rect[ind].length;
            rect_width_sum += rect[ind].width;
        }
    }
}

```

❑ 尽量用乘法或其他方法代替除法，特别是浮点运算中的除法。

说明 浮点运算除法要占用较多 CPU 资源。

示例：如下表达式运算可能要占较多 CPU 资源。

```

#define PAI 3.1416
radius = circle_length / (2 * PAI);

```

应把浮点除法改为浮点乘法，如下所示：

```

#define PAI_RECIPROCAL (1/3.1416) // 编译器编译时，将生成具体浮点数
radius = circle_length * PAI_RECIPROCAL / 2;

```

❑ 不要一味追求紧凑的代码。

说明 因为紧凑的代码并不代表高效的机器码。

4.2.9 质量保证

- ❑ 在软件设计过程中构建软件质量。
- ❑ 代码质量保证优先原则：
 - ❑ 正确性，指程序要实现设计要求的功能。
 - ❑ 稳定性、安全性，指程序稳定、可靠、安全。
 - ❑ 可测试性，指程序要具有良好的可测试性。
 - ❑ 规范/可读性，指程序书写风格、命名规则等要符合规范。
 - ❑ 全局效率，指软件系统的整体效率。
 - ❑ 局部效率，指某个模块、子模块、函数的本身效率。
 - ❑ 个人表达方式、个人方便性，指个人编程习惯。
- ❑ 只引用属于自己的存储空间。

说明 若模块封装较好，那么一般不会发生非法引用他人的空间。

- ❑ 防止引用已经释放的内存空间。

说明 在实际编程过程中，稍不留心就会出现一个模块中释放了某个内存块（如 C 语言指针），而另一模块在随后的某个时刻又使用了它。要防止发生这种情况。

- ❑ 在过程 / 函数中分配的内存，在过程 / 函数退出之前要释放。
- ❑ 过程 / 函数中申请的（为打开文件而使用的）文件句柄，在过程 / 函数退出之前要关闭。

说明 分配的内存不释放以及文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重的后果，且难以定位。

示例：如下所示函数在退出之前没有把分配的内存释放。

```
typedef unsigned char BYTE;

int example_fun( BYTE gt_len, BYTE *gt_code )
{
    BYTE *gt_buf;

    gt_buf = (BYTE *) malloc (MAX_GT_LENGTH);
    ... //program code, include check gt_buf if or not NULL.

    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
    {
        return GT_LENGTH_ERROR; // 忘记释放 gt_buf
    }

    ... // other program code
}
```

应改为如下所示形式：

```
int example_fun( BYTE gt_len, BYTE *gt_code )
{
    BYTE *gt_buf;

    gt_buf = (BYTE *) malloc ( MAX_GT_LENGTH );
    ... // program code, include check gt_buf if or not NULL.

    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
    {
        free( gt_buf ); // 退出之前释放 gt_buf
        return GT_LENGTH_ERROR;
    }

    ... // other program code
}
```

□ 防止内存操作越界。

说明 内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当进行这些操作时一定要仔细小心。

示例：假设某软件系统最多可由 10 个用户同时使用，用户号为 1~10，那么如下程序将存在问题：

```
#define MAX_USR_NUM 10
unsigned char usr_login_flg[MAX_USR_NUM]="";

void set_usr_login_flg( unsigned char usr_no )
{
    if (!usr_login_flg[usr_no])
    {
        usr_login_flg[usr_no]= TRUE;
    }
}
```

当 `usr_no` 为 10 时，将使 `usr_login_flg` 越界。可采用如下方式解决：

```
void set_usr_login_flg( unsigned char usr_no )
{
    if (!usr_login_flg[usr_no - 1])
    {
        usr_login_flg[usr_no - 1]= TRUE;
    }
}
```

- 认真处理程序所能遇到的各种出错情况。
- 系统运行之初，要初始化有关变量及运行环境，防止引用未经初始化的变量。
- 系统运行之初，要对加载到系统中的数据进行一致性检查。

说明 使用不一致的数据，容易使系统进入混乱状态和不可知状态。

- 严禁随意更改其他模块或系统的有关设置和配置。

说明 编程时，不能随心所欲地更改不属于自己模块的有关设置如常量、数组的大小等。

- 不能随意改变与其他模块的接口。
- 充分了解系统的接口之后，再使用系统提供的功能。

示例：在 B 型机的各模块与操作系统的接口函数中，有一个要由各模块负责编写的初始化过程，此过程在软件系统加载完成后，由操作系统发送的初始化消息来调度。因此就涉及初始化消息的类型与消息发送的顺序问题，特别是消息顺序，若没明白就开始编程，很容易引起严重后果。以下示例引自 B 型机曾出现过的实际代码，其中使用了 `FID_FETCH_DATA` 与 `FID_INITIAL` 初始化消息类型，注意 B 型机的系统是在 `FID_FETCH_DATA` 之前发送 `FID_INITIAL` 的。

```

MID alarm_module_list[MAX_ALARM_MID];

int FAR SYS_ALARM_proc( FID function_id, int handle )
{
    _UI i, j;

    switch ( function_id )
    {
        ... // program code

        case FID_INITAIL:
            for (i = 0; i < MAX_ALARM_MID; i++)
            {
                if (alarm_module_list[i]== BAM_MODULE // **)
                    || (alarm_module_list[i]== LOCAL_MODULE)
                {

                    for (j = 0; j < ALARM_CLASS_SUM; j++)
                    {
                        FAR_MALLOC( ... );
                    }
                }
            }

            ... // program code

            break;

        case FID_FETCH_DATA:

            ... // program code

            Get_Alarm_Module( ); // 初始化 alarm_module_list

            break;

            ... // program code
    }
}

```

由于 FID_INITIAL 是在 FID_FETCH_DATA 之前执行的，而初始化 alarm_module_list 是在 FID_FETCH_DATA 中进行的，因此在 FID_INITIAL 中 (**) 处引用 alarm_module_list 变量时，它还没有被初始化。这是个严重错误。

应进行如下改正：要么把 Get_Alarm_Module 函数放在 FID_INITIAL 中 (**) 之前；要么必须考虑 (**) 处的判断语句是否可用（不使用 alarm_module_list 变量的）其他方式替代，或者是否可以取消此判断语句。

□ 编程时，要防止差 1 错误。

说明 此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编写完程序后，应对这些操作符进行彻底检查。

- 要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

说明 形式相近的操作符最容易引起误用，如 C/C++ 中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错误，编译器不一定能够检查出。

示例：如把“&”写成“&&”，或反之。

```
ret_flg = (pmsg->ret_flg & RETURN_MASK);
```

被写为：

```
ret_flg = (pmsg->ret_flg && RETURN_MASK);
```

或者

```
rpt_flg = (VALID_TASK_NO( taskno ) && DATA_NOT_ZERO( stat_data ));
```

被写为：

```
rpt_flg = (VALID_TASK_NO( taskno ) & DATA_NOT_ZERO( stat_data ));
```

- 有可能的话，if 语句尽量加上 else 分支，对没有 else 分支的语句要小心对待；switch 语句必须有 default 分支。
- 在 UNIX 下，多线程中的子线程退出必需采用主动退出方式，即子线程应 return 出口。
- 不要滥用 goto 语句。

说明 goto 语句会破坏程序的结构性，所以除非确实需要，否则最好不要使用 goto 语句。

- 不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。
- 除非为了满足特殊需求，否则避免使用嵌入式汇编。

说明 若在程序中嵌入式汇编，一般都对可移植性有较大的影响。

- 精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性。

说明 对于不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。

- 精心构造算法，并对其性能、效率进行测试。

❑ 对较关键的算法最好使用其他算法来确认。

❑ 时刻注意表达式是否会上溢、下溢。

示例：如下程序将造成变量下溢：

```
unsigned char size ;
while (size-- >= 0) // 将出现下溢
{
    ... // program code
}
```

当 size 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。应进行如下修改：

```
char size; // 将 unsigned char 改为 char
while (size-- >= 0)
{
    ... // program code
}
```

❑ 使用变量时要注意其边界值的情况。

示例：如 C 语言中字符型变量的有效值范围为 -128~127。因此以下表达式的计算存在一定风险。

```
char chr = 127;
int sum = 200;

chr += 1; // 127 为 chr 的边界值，再加 1 将使 chr 上溢到 -128，而不是 128。

sum += chr; // 故 sum 的结果不是 328，而是 72。
```

若 chr 与 sum 为同一种类型，或表达式按如下方式书写，可能会好些。

```
sum = sum + chr + 1;
```

❑ 留心程序机器码大小（如指令空间大小、数据空间大小、堆栈空间大小等）是否超出系统有关限制。

❑ 为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。

❑ 系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。

❑ 对一些具有危险性的操作代码（如写硬盘、删数据等）要仔细考虑，防止对数据、硬件等的安全构成危害，以提高系统的安全性。

❑ 使用第三方提供的软件开发工具包或控件时，要注意以下几点：

❑ 充分了解应用接口、使用环境及使用时的注意事项。

❑ 不能过分相信其正确性。

❑ 除非有必要，否则不要使用不熟悉的第三方工具包与控件。

说明 使用工具包与控件，可加快程序开发速度，节省时间，但使用之前一定对它有较充分的了解，同时第三方工具包与控件也有可能存在问题。

❑ 资源文件（多语言版本支持），如果资源是对语言敏感的，应让该资源与源代码文件脱离，具体方法包括：使用单独的资源文件、DLL 文件或其他单独的描述文件（如数据库格式）等。

4.2.10 代码编辑、编译、审查

- ☐ 打开编译器的所有警告开关对程序进行编译。
- ☐ 在产品软件（项目组）中，要统一编译开关选项。
- ☐ 通过代码走读及审查方式对代码进行检查。

说明 代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查，可由开发人员自己或以开发人员交叉的方式进行；代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审，可通过自审、交叉审核或指定部门抽查等方式进行。

- ☐ 测试部门测试产品之前，应对代码进行抽查及评审。
- ☐ 编写代码时要注意随时保存，并定期备份，防止由于断电、硬盘损坏等原因造成代码丢失。
- ☐ 同产品软件（项目组）内，最好使用相同的编辑器，并使用相同的设置选项。

说明 同一项目组最好采用相同的智能语言编辑器，如 Mui Editor、Visual Editor 等，并设计、使用一套缩进宏及注释宏等，将缩进等问题交由编辑器处理。

- ☐ 要小心地使用编辑器提供的块拷贝功能编程。

说明 当某段代码与另一段代码的处理功能相似时，许多开发人员都用编辑器提供的块拷贝功能来完成这段代码的编写。由于程序功能相近，故所使用的变量、采用的表达式等在功能及命名上可能都很相近，所以使用块拷贝时要注意，除了修改相应的程序外，一定要把使用的每个变量仔细查看一遍，以改成正确的。不应指望编译器能查出所有这种错误，比如当使用的是全局变量时，就有可能隐藏某种错误。

- ☐ 合理地设计软件系统目录，方便开发人员使用。

说明 方便、合理的软件系统目录可提高工作效率。目录构造的原则是方便有关源程序的存储、查询、编译、链接等工作，同时目录中还应具有工作目录——所有的编译、链接等工作应在此目录中进行，工具目录——有关文件编辑器、文件查找等工具可存放在此目录中。

- ☐ 某些语句经编译后产生警告，但如果你认为它是正确的，那么应通过某种手段消除警告信息。

说明 在 Borland C/C++ 中，可用“#pragma warn”来关掉或打开某些警告消息。

示例：

```
#pragma warn -rvl // 关闭警告
int examples_fun( void )
{
    // 程序，但无 return 语句。
}
#pragma warn +rvl // 打开警告
```

编译函数 `examples_fun` 时本应产生“函数应有返回值”警告，但由于关闭了此警告信息显示，所以编译时将不会产生此警告提示。

- ❑ 使用代码检查工具（若使用 C 语言则采用 PC-Lint）对源程序检查。
- ❑ 使用软件工具（如 LogiSCOPE）进行代码审查。

4.2.11 测试与维护

- ❑ 单元测试要求至少达到语句覆盖。
- ❑ 单元测试开始要跟踪每一条语句，并观察数据流及变量的变化。
- ❑ 清理、整理或优化后的代码要经过审查及测试。
- ❑ 代码版本升级要经过严格测试。
- ❑ 使用工具软件对代码版本进行维护。
- ❑ 正式版本上软件的任何修改都应有详细的文档记录。
- ❑ 发现错误立即修改，并且记录下来。
- ❑ 关键的代码在汇编级跟踪。
- ❑ 仔细设计并分析测试用例，使测试用例覆盖尽可能多的情况，以提高测试用例的效率。
- ❑ 尽可能模拟出程序的各种出错情况，充分测试出错处理代码。
- ❑ 仔细测试代码处理数据、变量的边界情况。
- ❑ 保留测试信息，以便分析、总结经验和进行更充分的测试。
- ❑ 不应通过“试”来解决问题，应寻找问题的根本原因。
- ❑ 对自动消失的错误进行分析，搞清楚错误是如何消失的。
- ❑ 修改错误不仅要治表，更要治本。
- ❑ 测试时应设法使很少发生的事件经常发生。
- ❑ 明确模块或函数处理哪些事件，并使它们经常发生。
- ❑ 坚持在编码阶段就对代码进行彻底的单元测试，不要等以后在测试工作中发现问题。
- ❑ 去除代码运行的随机性（如去掉无用的数据、代码以及尽可能防止并注意函数中的“内部寄存器”等），让函数运行的结果可预测，并使出现的错误可再现。

4.2.12 宏定义

- ❑ 用宏定义表达式时，要使用完整的括号。

示例：如下定义的宏都存在一定的风险。

```
#define RECTANGLE_AREA( a, b ) a * b
#define RECTANGLE_AREA( a, b ) (a * b)
#define RECTANGLE_AREA( a, b ) (a) * (b)
```

正确的定义应为：

```
#define RECTANGLE_AREA( a, b ) ((a) * (b))
```

❑ 将宏所定义的多条表达式放在大括号中。

示例：下面的语句只执行宏的第一条表达式。为了说明问题，for 语句的书写稍不符规范。

```
#define INTI_RECT_VALUE( a, b )
    a = 0;
    b = 0;

for (index = 0; index < RECT_TOTAL_NUM; index++)
    INTI_RECT_VALUE( rect.a, rect.b );
```

正确的写法应为：

```
#define INTI_RECT_VALUE( a, b )
{
    a = 0;
    b = 0;
}

for (index = 0; index < RECT_TOTAL_NUM; index++)
{
    INTI_RECT_VALUE( rect[index].a, rect[index].b );
}
```

❑ 使用宏时，不允许参数发生变化。

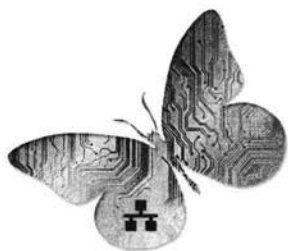
示例：如下用法可能导致错误。

```
#define SQUARE( a ) ((a) * (a))

int a = 5;
int b;
b = SQUARE( a++ );           // 结果：a = 7，即执行了两次加 1 操作。
```

正确的用法是：

```
b = SQUARE( a );
a++;                          // 结果：a = 6，即只执行了一次加 1 操作。
```



第5章

项目规划

5.1 概述

一般而言，一个功能相对独立的产品的开发工作是一个系统工程，这是为了合理进行开发、设计和运用系统而采用的思想、步骤、组织和方法等的总称。就其本质而言，系统工程是一个方法论。在开发工作中，除了必备的专业技术外，工程本身对项目的管理人员、规划人员和执行人员有着更高的要求，诸如团队沟通能力、协作能力等。

嵌入式系统开发项目通常包括系统分析、系统设计、系统制造、系统运用、系统评价和系统维护（性能、费用和时间等）六个阶段。从系统工程角度考虑，一般采用先决定整体框架，后进入详细设计的程序；先进行系统的逻辑思维过程总体设计，然后进行各子系统或具体问题的研究；通过对系统的综合、分析和构造系统模型（或原型）来调整改善系统的结构，使系统整体功能达到最优化。

系统工程的研究强调系统与环境的融合，近期利益与长远利益相结合，社会效益、生态效益与经济效益相结合。一个系统是为一个特定的目标而产生的，运行于某个特定的环境。因此，系统会与周围的环境有交接的边界，且在该环境中发挥有益社会效益的同时产生一定的经济效应。此外，系统应不致对社会生态产生破坏性的影响。否则，系统及其开发工作的本身就是贫价值或无价值的，在项目开发工作开展之初，就应当明确不违背上述初衷。

本章将从系统工程的角度出发，讲述嵌入式项目开发工作的系统分析、系统设计、系统制造、系统运用、系统评价、系统维护六个阶段，以及相应开发文档的建立的全过程及注意事项。

5.2 系统分析

对于一个从事具体技术开发工作的工程师而言，在系统分析阶段可能仅会聚焦于工程需求的细化和技术实现的细节，无法对工程整体从宏观上进行分析和掌控。事实上，这也是一个从事具体技术工作人员无法完成的，通常需要多个部门人员的协同工作来完成。因此，在一个项目策划之初，组织或单位就应组建协同工作的预研团队，对工程的应用背景、市场需求、社会效益、潜在风险、工程需求进行分析和评估，并通过可行性研究报告对工程预研活动做详细的文档记录和说明。

一个具体的工程实践是以特定的应用背景为基础的,离开具体的实施环境,工程项目不仅无法明确特定的市场需求和潜在价值,更无法将其付诸实践。在明确了工程的应用背景之后,项目组应通过调研或外委咨询的方式进行市场需求调研,为后续社会效益分析和风险评估提供数据论证和现实支撑。

所谓社会效益,是指人们的社会实践活动对社会发展所起的积极作用或产生的有益效果。社会效益有广义和狭义之分,广义的社会效益包括政治效益、经济效益、思想效益和文化效益等。狭义的社会效益是指企业经济活动给社会带来的收入,而社会成本则是其带来的消耗,两者之差就是社会收益,即企业所提供的社会贡献净额。对于企业而言,所涉及的社会效益通常是狭义上的社会效益,但在报告中应从广义、宏观的角度去分析潜在的效益。

5.3 系统设计

系统设计是新系统的物理设计阶段。根据系统分析阶段所确定的新系统的逻辑模型、功能要求,在用户提供的环境条件下,设计出一个能在既定环境上实施的方案,即建立新系统的物理模型。这个阶段的任务是设计系统的模块层次结构,其目的是明确系统如何实现。这个阶段又分概要设计和详细设计,概要设计解决系统架构设计。

以软件系统设计为例,概要设计解决系统的模块划分和模块的层次结构;详细设计解决每个模块的控制流程、内部算法和数据结构的设计。这个阶段结束,要交付概要设计说明书和设计说明,也可以合并在一起,称为设计方案。在系统分析的基础上,设计出能满足预定目标的系统的过程。系统设计内容主要包括:确定设计方针和方法,将系统分解为若干子系统,确定各子系统的目标、功能及其相互关系,决定对子系统的管理体制和控制方式,对各子系统进行技术设计和评价,对全系统进行技术设计和评价等。

系统设计通常应用两种方法:一种是归纳法,另一种是演绎法。应用归纳法进行系统设计的程序是:首先尽可能地收集现有的和过去的同类系统的系统设计资料;在对这些系统的设计、制造和运行状况进行分析研究的基础上,根据所设计的系统的功能要求进行多次选择,然后对少数几个同类系统做出相应修正,最后得出一个理想的系统。演绎法是一种公理化方法,即先从普遍的规则和原理出发,根据设计人员的知识和经验,从具有一定功能的元素集合中选择能符合系统功能要求的多种元素,然后将这些元素按照一定形式进行组合,从而创造出具有所需功能的新系统。在系统设计的实践中,这两种方法往往是并用的。

对于可行性研究报告而言,应当包括5.2节的全部内容和本节所述及的部分内容,后续几节简述工程实践中的后续几个环节。

5.4 系统制造

制造系统是制造过程及其所涉及的硬件、软件和人员所组成的一个将制造资源转变为产品或半成品的输入/输出系统,它涉及产品生命周期的全过程或部分环节,包括市场分析、产品设计、

工艺规划、加工过程、装配、运输、产品销售、售后服务及回收处理等。其中，硬件包括厂房、生产设备、工具、刀具、计算机及网络等；软件包括制造理论、制造技术（制造工艺和制造方法等）、管理方法、制造信息及其有关的软件系统等；制造资源包括狭义制造资源和广义制造资源；狭义制造资源主要指物能资源，包括原材料、坯件、半成品、能源等；广义制造资源还包括硬件、软件、人员等。

5.5 系统运用及反馈

对于富有责任感的企业来说，产品或系统的开发和维护应该是全生命周期的。从系统的规划到最终产品或系统的推广，企业应积极收集客户或市场的反馈和评价，并跟进系统维护与升级。只有企业充分重视客户或市场反馈的意见和建议，并用于改进产品的设计，才能赢得客户的尊重。

需要重视的一点是，任何对产品的升级和改进都应更新相应的设计输入文档，并对每次的更新活动的内容做详细的记录和说明。

5.6 开发团队

开发团队作为一个项目的执行实体，不仅在产品研发的过程中负责实现具体产品或系统功能的实现，还要在整个产品开发的初期对项目执行的可行性进行论证。因此，开发团队就其总体而言，必须兼备协调能力、管理能力、专业知识、商务能力等方面的素质。一个合格的开发团队的每个成员都必须有明确的工作目标和进度规划，并且能在权责范围内充分发挥功能。

5.6.1 团队负责人

作为团队的负责人，应该具备较为全面的综合素质。在项目执行的生命周期内，团队负责人除了制定切合实际并可行和具有一定弹性空间的详细执行计划，还应有能力取得所需的资源，协调运作团队内的各项工作，以期在进度内达到既定的目标。与此同时，项目负责人应能够控制项目的经费，避免中途追加预算，无节制地推升产品的成本和项目的执行成本。

团队负责人还应具备一定的所属行业或领域的专业知识，包括新产品开发涉及的专业知识，以及相关法规等。以便在团队遇到技术上的困难时，协调团队进行技术攻关。而商务能力是项目负责人必须具备的一种能力和素质，特别是在可行性论证阶段，项目负责人必须充分认识和了解新产品的未来客户、市场结构及可能的竞争对手，还应对整个项目及新产品所造成的财务影响有清晰的认识。总而言之，一个优秀的项目负责人，应具备协调能力、管理能力、专业知识、商务能力等方面的素质。

值得一提的是，人们至今对项目管理人员（在项目执行的生命周期中，项目负责人实际上是项目团队的最高管理者）是否应具有专业技能还存在着争议。所谓专业技能，是指利用所掌握的专业知识，并利用各种工具以有利于组织利益的某种方式解决问题的能力。

实际上，专业技能对于不同管理层次的管理者的相对重要性是不同的。专业技能的重要性依

据管理者所处的组织层次从低到高逐渐降低。对基层管理者来说,具备专业技能是最为重要的。当管理者在组织中的组织层次从基层往中层、高层发展时,随着他同下级直接接触的频率的减少,所涉及的专业技术问题也越来越少,专业技能的重要性也逐渐降低。对于高层管理者而言,管理能力和决策能力(商务能力的重要组成)特别重要,而对专业技能的要求相对来说则很低。在以项目管理为模式的组织或机构中,人事结构相对扁平,这种结构更强调执行的效率和速度。因此,对管理者的技能和知识结构要求较为全面。

因此,一个工程师在职业规划时,不论当前从事何种类型的分工,都应有意识地全面培养自己的能力,为自身发展开拓更为广阔的空间。

5.6.2 调研人员

调研人员是在开展项目的可行性研究时的主要参与者,负责对产品或系统的前景进行客观实际情况的调查研究,将调研获得的全部情况和材料进行“去粗取精、去伪存真、由此及彼、由表及里”的分析研究,揭示本质,寻找规律,总结经验,最后以书面形式进行陈述。这种以书面形式表达出来的内容成为可行性研究报告的重要组成部分。

调研的本质是实事求是地反映和分析客观事实。调研报告主要包括调查和研究两个部分。调查应以客观事实为依据,讲究论据充分、数据详实,以便准确地反映客观事实的本质。而研究是在掌握客观事实的基础上,通过分析、归纳和总结以揭示事物的本质和规律。只有在客观准确调研的基础上,才能对项目的前景做出准确判断。

可能大多数嵌入式开发工程师认为这些应该是从事市场工作人员的职责,事实上,一个公司做出的重大决策是在市场、管理和技术共同作用下,公司内部不同工作岗位工作人员共同参与决策的结果。大家都不会忘记2012年4月20日,美国伊士曼柯达公司正式宣布破产的案例。作为对世界影响最大的产品及相关服务的生产商和供应商,在20世纪70年代中叶,柯达垄断了美国90%的胶卷市场以及85%的相机市场份额。然而,柯达公司在迎来自己132岁生日的时候,却面临着真正的末日宣判。由于柯达公司的产品发展长期重点围绕传统胶卷、印像和冲洗业务,压制了数码相机的进步。即使在摄影技术从胶片化向数码化转型的趋势已十分明显的情况,柯达依然沉溺于传统胶片,无法扭转全球胶卷消费市场以每年10%的速度急速萎缩的颓势,直到不得不关闭生产了74年的胶卷工厂。或许有人会将这个“帝国”崩溃的原因归结于缺乏科技创新,事实上,仅从1900~1999年,柯达的工程师们共获得了19 576项专利,甚至在最后的时间靠出售专利挣扎求活。作为长期的行业领跑者,认为其缺乏科技创新显然不符合事实。从表面上看,柯达申请破产保护是因为该公司连年亏损,资不抵债。但更深层次原因在于其战略决策失误,未能适应市场变化,及时转型升级。

与柯达公司不同的是,美国苹果公司并不像柯达一样停留在技术创新阶段,而是把现代技术与现代艺术有机地结合在一起,向消费者呈现全新的互联网络工具。苹果公司的每一项技术都是可以复制的,但是,苹果公司生产的产品却是独一无二的。其中的原因就在于,苹果公司把文化作为一个重要的元素,从而使苹果公司的产品具有了文化的魅力。柯达影像帝国在市场大潮中的

轰然坍塌与苹果公司的崛起，表明了战略决策对于一个企业发展的重要意义，甚至决定着一个企业的成败。任何一种成熟的技术，都有被新技术完全替代的可能，成功的企业要能科学判断自己经营产品的技术发展方向及其市场前景，并及时做出相应的战略调整。

从这个角度讲，一个项目规划及决策比一个产品的开发和实现更为重要。而调研是一个项目规划的重要组成部分，也是做出决策的客观依据。众所周知，国外成熟企业在进行项目开发或做出重大决策时，往往会不惜重金委托专业咨询机构进行市场调研，以辅助其做出科学的决策。就调研的工作内容而言，通常包括计划、实施、收集、整理等一系列过程，其结果是调研人员劳动与智慧的结晶，也是反映真实市场需求的最重要的书面确认。调研的目的是将调查结果、战略性的建议以及其他结果传递给管理和决策部门，为其制定科学合理的决策提供理论论证和事实依据。因此，认真撰写可行性研究报告，准确分析调查结果，明确给出调查结论以便做出科学合理的决策依据，是调研人员的重要责任。

5.6.3 开发人员

中国嵌入式产业的发展面临着良好的发展环境与机遇，除了得益于政策的扶持、消费类电子产品的普及等因素外，信息产业与传统产业的融合也推动了嵌入式产业的蓬勃发展。然而，受限于开发人员的技能水准、嵌入式固有的软硬结合特性等诸多问题，使嵌入式高端人才一直处于相对匮乏的状态。遗憾的是，至今仍有众多嵌入式开发人员将其编码或设计能力作为评价其实力和水准的核心指标，而忽视了项目规划、架构设计及文档备案等能力的培养。本小节将简要探讨一个合格开发人员应具备的工程素养，以及进行高效嵌入式系统开发的注意事项和要点，同时也向读者引荐嵌入式实时系统开发中的几种常见工具。

1. 文档备案

随着嵌入式实时系统复杂度的提高，开发人员在定义和分析系统初始要求时必须认真考虑软硬件的协同关系。通常，开发人员还必须权衡系统的灵活性、速度、成本、计划和可用工具之间的关系。若不在项目开发过程中建立详细的文档备案而仅凭简单的梳理和记忆追溯，开发人员将很难做出合理的规划设计，也不利于后续系统或产品的维护。客观上来说，项目规划、架构设计及文档备案对项目的成败起着举足轻重的作用。很多初步涉足嵌入式开发的工程师往往更重视工具的应用和技能的培养，而把项目规划、架构设计和文档备案作为应付公司要求的例行任务，甚至在公司层面就不予以重视。这将导致系统或产品需求不明确、架构设计不合理、后续维护困难等诸多问题，并间接推升产品的开发和维护成本，甚至导致项目的最终失败。

实际上，一个系统或产品设计完成以后，设计的输出不应仅是设计图纸和源代码，还应该包括各种各样的规划和开发文档。这些文档的编制至少应与开发工作同步进行，甚至超前于具体开发工作，并且建立科学合理的评审机制，在建立文档的过程中对规划的合理性进行论证和评审。事实证明，编制规划设计文档并建立有效评审机制有助于规避项目设计中的不确定性风险，这些文档对系统的维护和升级都有重要的参考作用，这也是系统开发的规范化的重要组成部分。从另一个角度讲，系统开发的规范化不仅有利于开发人员的综合能力培养，也有利于企业的规范化发

展。规范化的设计让工程师工作更高效，这已经是不用争论的事实。目前，大型软件开发的规范化程度远胜于嵌入式系统的开发。在国内，一些公司的嵌入式研发人员通常将大部分精力都集中于工程的编码和后续的维护上，更有甚者，项目的规划开发文档根本无从稽考，这也将刚涉足该领域的工程师引入这样一种误区——几乎所有时间都在写代码、改代码。

2. 设计工具

通常，嵌入式系统是其行为与时间紧密相关的实时系统，传统的非实时系统设计方法已难以应付日益复杂系统的实现。为此，人们将广泛应用于大型软件系统设计与分析的统一建模语言（Unified Modeling Language, UML）进行了大量的扩展，并增加了实时建模支持，形成了基于实时系统的统一建模语言（RealTime-UML），为嵌入式实时系统的设计提供了良好的方法、工具和语言的支持。在2004年，25%的嵌入式系统项目使用了统一建模语言，而且该种设计方法也逐渐得到了业界的广泛肯定和认可。目前，常见的 RealTime-UML 的设计软件有 TNI 的 ControlBuild Embedded、IBM 的 Rational Rose RealTime、iLogix 的 Rhapsody、Artisansw 的 Artisan Studio 和 Telelogic 的 Tau 等。下文简要介绍几种基于实时系统的统一建模语言的设计工具软件。

ControlBuild 是由 TNI 软件公司设计的一款应用于工业自控软件领域的全流程设计软件，旨在帮助汽车、交通、制造、能源等的自控系统集成商、OEM 和设计人员按时、高质量并符合预算地完成自控工程。该软件提供了一个全流程的解决方案，涵盖了从用户需求、规格说明、设计、代码生成、仿真校验直到验收的完整嵌入式控制软件开发流程，如图 5.1 所示。ControlBuild 提供强大的仿真工具，逼真模拟实际列车或舰船等模型，并在完整的仿真环境中，对设计进行功能验证、代码校验以及自动生成设计文档，并确认设计完全符合客户需求，测试系统设计在各种异常情况或灾难下的响应，从而实现在现场安装调试之前校验测试整个设计，发现并校正绝大多数设计错误和问题，提高设计质量，并大大缩短现场安装调试的时间。使用 ControlBuild 可以减少设计的复杂性和成本，并显著地提高自控系统设计的质量。

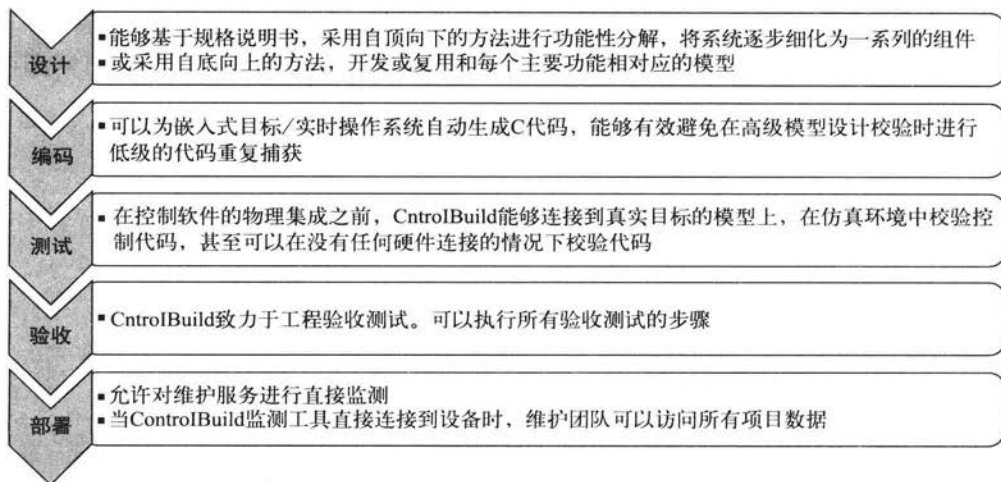


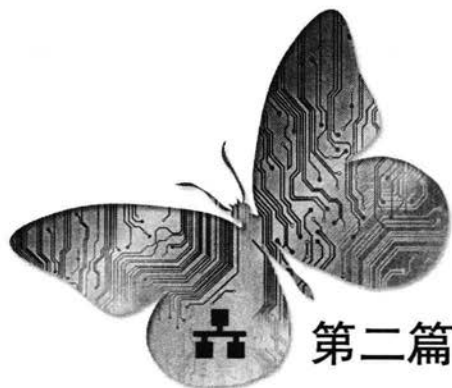
图 5.1 使用 ControlBuild 进行软件开发的流程

Rational Rose RealTime 是一种适用于整个生命周期的开发环境,使用 UML,专为迎接嵌入式实时系统的挑战而创建。它提供了 UML 兼容的解决方案,用于解决并发和分布式方面的独特问题。通过提供一组全面的集成工具,Rational Rose RealTime 使项目团队得以统一,从需求获取到生成高性能代码再到基于目标机平台的调试,满足整个团队的需求。Rational Rose RealTime 是以市场领先技术 Rational Rose (世界领先的可视化建模工具)为基础,扩展来自 ObjecTime Limited 公司的技术而形成的产品。ObjecTime Limited 公司是嵌入式实时系统开发工具领域的技术领先者,其 UML 模型代码生成和可视化技术已经得到业界认同。

Rhapsody 是由美国 iLogix 公司为嵌入式系统开发的一个模型驱动的开发环境。Rhapsody 能够最大限度地让系统、硬件、软件和测试工程师以更加灵活和优化的方式协同开发和交流。它以 UML 2.0 为基础,使大小不同的项目都能够通过可视化建模的方法分析、设计、实现和测试自己的工作,构建和配置实时嵌入式应用。Rhapsody 专为嵌入式市场的特殊需求设计和优化,包括实时系统的行为语义、实时操作系统的支持、无操作系统的实时应用支持、遗产代码的逆向工程、设计级的调试、高效的代码自动生成(C、C++、Ada、Java)和文档自动生成等。目前,Rhapsody 支持 UML 2.0 的增强功能,增强了实时建模在调度、性能、时间上的扩展。

Rhapsody 是一个开放的、可定制的、可扩展的设计平台。Rhapsody 提供与市场上提供的优秀的配置管理工具、需求管理跟踪工具和测试分析工具的接口,如 ClearCase、PVCS、CVS、Doors 等。另外,利用 Rhapsody 的 API 和 XMI 数据接口,Rhapsody 可以与开发工具链上的任何工具相连接。加上 Rhapsody 提供的可视化的模型区分与合并功能,真正使团队协作成为可能。同时,Rhapsody 提供对标准的 XML 格式的支持,方便开发者共享他们的设计成果。总之,开发人员对于合作设计和远程团队开发的选择不会因为 Rhapsody 而受到任何限制。目前,国外众多知名公司采用 Rhapsody 进行嵌入系统的建模和软件设计,如美国联合攻击战斗机(Joint Strike Fighter, JSF)、美军未来战斗系统(Future Combat System, FCS)和 F-22(美国洛克希德·马丁、波音和通用动力等公司联合为美国空军设计的重型隐身战斗机)。

至此,本章提纲挈领地向读者讲述了嵌入式项目系统规划和设计中的注意事项,最后简要介绍了几款基于统一建模语言的开发环境。由于涉及的内容庞杂,本章仅作为引荐性的阅读材料,读者可根据需要阅读相关的书籍和材料。

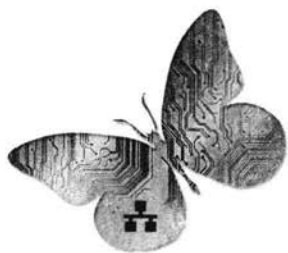


第二篇

RTOS篇

本篇主要结合操作系统原理简述 FreeRTOS 实时嵌入式操作系统的基本原理，以及 FreeRTOS 的任务管理、队列管理、中断管理等，最后介绍了 FreeRTOS 在 STM32 嵌入式处理器上的移植过程。

- 第6章 操作系统原理基础知识
- 第7章 FreeRTOS嵌入式操作系统
- 第8章 基于STM32F107的FreeRTOS移植



第 6 章

操作系统原理基础知识

本章简要介绍嵌入式软件开发中最常见的两种编程模式：前后台模式应用程序和嵌入式操作系统，并对二者的优缺点进行了比较。

6.1 前后台模式应用程序

对于嵌入式应用程序来说，应用程序通常是一个无限的循环，可称为前后台系统或前后台模式应用程序。在循环中，通过调用相应的函数完成相应的操作，这部分可以看成后台行为。由中断服务程序处理异步事件，这部分可以看成前台行为。异步事件通常是实时性要求较高的任务。在较为复杂的应用场合下，这种系统的实时性通常与设计预期有一定差距。

6.2 嵌入式操作系统

针对嵌入式操作系统，国内普遍认同的一个定义是：以应用为中心、以计算机技术为基础、软硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。嵌入式系统是与应用紧密结合的，具有极强的专用性，必须结合实际系统需求进行合理裁减利用。嵌入式实时操作系统能够支持多任务，使得程序开发更加容易，便于维护，同时能够提高系统的稳定性和可靠性。这已逐渐成为嵌入式系统开发的一个发展方向，其应用领域遍及科研、工业、电力、能源、军工等领域。本节简要叙述操作系统的一些基本概念和问题，为读者后续学习 FreeRTOS 提供入门知识，更深入的学习请读者阅读相关专业书籍。

6.2.1 相关基本概念

1. 进程

在操作系统中，“进程”是一个非常重要的概念，进程本质上是一个正在执行的程序，通常每个进程都有自己独立的访问地址空间。

第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈区域（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋

予程序生命时，它才能成为一个活动的实体，称为进程。进程是操作系统中最基本、最重要的概念，是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律而引进的一个概念，所有多道程序设计操作系统都建立在进程的基础上。

2. 存储管理

操作系统通常允许在内存中运行多个程序，为了避免多个程序之间的相互干扰，必须建立某种机制对运行程序的内存进行保护。

3. 输入 / 输出

操作系统最主要的功能之一就是控制所有的输入 / 输出设备，从输入获得执行的命令，捕获中断并进行处理。应当说输入 / 输出是实现系统既定功能必不可少的设备。

4. 文件系统

文件系统是支持操作系统的一个关键概念。它为程序员提供一个良好、清晰的独立于设备的抽象文件模型，而不必考虑磁盘和其他 I/O 设备的操作细节。

5. 死锁

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力介入，它们都将无法继续执行。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。由于资源占用是互斥的，当某个进程提出资源申请后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

6.2.2 系统调用

由操作系统实现的所有系统调用所构成的集合即程序接口或应用编程接口（Application Programming Interface, API），是应用程序同系统之间的接口。

操作系统的主要功能是为应用程序的运行创造良好的环境，为了达到这个目的，内核提供一系列具备预定功能的多内核函数，通过一组称为系统调用（system call）的接口呈现给用户。系统调用把应用程序的请求传给内核，调用相应的内核函数完成所需的处理，然后将处理结果返回给应用程序，这样就简化了应用程序的开发，使得编写大型应用程序成为可能。

6.2.3 操作系统结构

嵌入式应用开发工程师可能更多关注的是操作系统的接口，而对系统的内部结构了解不多。但要想深入理解操作系统的运行机理和实现，必须对操作系统的结构进行深入学习。通常，操作系统的设计思想有单体系统、分层系统、虚拟机、外核、客户机—服务器模型。FreeRTOS 是采用单体式设计思想实现的微型嵌入式操作系统。

1. 单体系统

单体系统以过程集合的方式实现。通过编译链接，生成一个大型可执行二进制程序，完全运行于内核态中。系统中的每个过程都可以调用其他过程。这种系统的特点在于，每个过程都有一个通过参数和返回结果给出良好定义的接口。必要时，每个过程也可以自由调用其他过程。当然，

在单体系统中也存在结构的概念，单体内核仅是将 OS 的全部功能在内核中实现，包括调度、文件系统、网络、设备驱动器、存储管理。单体内核通常在一大块代码中实际包含了所有操作系统功能，并作为一个单一进程运行，具有唯一地址空间。大部分 UNIX（包括 Linux）系统都采用单体内核。FreeRTOS 本质上属于微内核，与单体内核不同，微内核只是将 OS 中最核心的功能加入内核，包括 IPC 通信、地址空间分配和基本的调度，这些东西处在内核态运行。而其他功能如设备驱动、文件系统、存储管理、网络等作为一个个处于用户态的进程而向外提供某种服务来实现，而且这些处于用户态的进程可以针对某些特定的应用和环境需求进行定制。有时，也称这些进程为服务器。

微内核外部的操作系统部件被当做服务器进程来实现，它们可以借助与微内核传递消息来实现相互之间的交互。因此微内核一般具有如下优势：接口一致、可扩展、灵活、可移植性好、可靠性，小的微内核可以被严格测试。微内核的不足之处在于，由于微内核中只有最基本的功能，微内核外部操作系统部件被当做服务器进程实现，进程间通信通过微内核构造和发送消息、接受应答并解码所花费的时间比进行一次系统调用的时间要多。

2. 分层系统

顾名思义，分层系统由多层组成，上一层是在下一层的基础上构建的。在 Andrew S. Tanenbaum 的《现代操作系统》中，提到了荷兰 E. W. Dijkstra 和他的学生采用分层设计的方法实现的 THE 操作系统。该系统由 6 层组成，如表 6.1 所示。

表 6.1 THE 操作系统的分层及功能

层 号	功 能
0	处理器分配和多道程序设计
1	存储器和磁鼓管理
2	操作员-进程通信
3	输入/输出管理
4	用户程序
5	系统操作员

第 0 层实现处理器资源的分配，当中断发生或定时器计时结束时，由该层进行进程切换；第 1 层实现存储管理，为进程分配主存空间；第 2 层处理进程与操作员控制台的通信；第 3 层则管理输入输出设备和缓存相关的信息流；第 4 层是用户程序层，用户无需考虑进程、内存和输入输出设备的细节；第 5 层是系统操作员。

实际上，THE 分层方案只是为设计提供了一些方便，因为该系统的各个部分最终仍然被连接成了完整的单个目标程序。

3. 虚拟机

虚拟机（Virtual Machine）指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。虚拟机起源于麻省剑桥的一个研究中心开发的 VM/370 系统，该系统

的核心称为虚拟机，它在裸机上运行并具备了多通道程序设计功能，能为上层提供若干台虚拟机，每台虚拟机运行不同的任务。

通过虚拟机软件，你可以在一台物理计算机上模拟出两台或多台虚拟的计算机，这些虚拟机完全就像真正的计算机那样进行工作，例如，可以安装操作系统、安装应用程序、访问网络资源等。对于你而言，它只是运行在物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就是一台真正的计算机。

4. 外核

外核操作系统减少了传统概念，即操作系统必须提供构建应用程序的抽象内容。该方法实现了应用级资源管理，即由应用程序而不是操作系统管理硬件资源。这时，进程间通信、虚拟内存管理等抽象概念都是单个应用实现的。因此，外核将资源保护及其管理分割开。因为外核只提供有限的原语，所以外核操作系统效率很高。因为进程间通信、虚拟内存管理等传统概念都是在应用层实现的，所以可以很容易对它们进行扩展、专业化和替换。

5. 客户机-服务器模型

现在的操作系统越来越倾向于从操作系统中抽掉尽可能多的东西，仅保留最小的内核，将大多数操作系统的功能由用户进程来实现。当用户进程（即客户进程）想要获得某项服务时，便向服务器进程发送一个请求，服务器根据请求的内容完成相应操作，并将操作的结果反馈给客户进程。客户机-服务器模型的特点是将操作系统分割成许多部分，每部分完成某一特定的功能，诸如文件服务、进程服务、内存服务等。这种分割方式的优点在于系统某一部分的瘫痪不致影响整个系统的运行。

6.2.4 进程与任务

前面讲述了进程的基本概念，那么进程和任务之间又是怎样的关系呢？在嵌入式操作系统中，人们通常对进程或任务不加区分。但就概念本身而言，还是有着细微的区别。进程是一个程序在其自身的虚拟地址空间中的一次执行活动。之所以要创建进程，是因为要使多个程序可以并发执行，从而提高系统的资源利用率和吞吐量。任务是一个很宽泛的概念，通常由很多个进程相互作用，包括用户对系统操作时的各个动作及所对应的响应事件。进程是任务的执行实体，一个任务往往要由若干进程共同完成。进程是若干指令在一定环境下对数据集合的动态执行过程。

6.2.5 进程间的通信

进程间通信就是不同进程之间传播或交换信息，进程通过与内核及其他进程之间的互相通信来协调它们的行为，包括数据传输、共享数据、通知事件、资源共享和进程控制。一般来说，进程的用户空间是互相独立的，资源的访问是互斥的，唯一的例外是共享内存区。

6.2.6 进程调度

对单核嵌入式处理器而言，当多个进程同时抢占处理器资源时，操作系统必须决定进程运行

的先后次序和方式。操作系统中实现这种进程共享资源的次序及方式的部分称为调度器，运用的算法称为调度算法。

对进程的调度方式决定了进程所处的状态。通常进程有三种状态，即阻塞态、就绪态、运行态，其中阻塞态和就绪态同属于非运行态。阻塞态是指进程等待某个事件的到来，或等待其他进程占用的互斥性资源时所处的状态；就绪态是指进程等待系统分配处理器以便运行时所处的状态；当进程占有处理器并正在运行时，称该进程处于运行态。

此外，挂起态通常也作为进程所处的非运行状态的一种，与阻塞态和就绪态不同的是，被“挂起”的进程通常是被操作系统清理出内存时所处的一种状态。由于处理器的资源是有限的，在资源不足的情况下，操作系统对在内存中的程序进行合理安排，其中有的进程被暂时调离出内存，当条件允许的时候，会被操作系统再次调回内存，重新进入就绪态。需要提及的一点就是，在 FreeRTOS 中，调用 `vTaskSuspend()` API 函数可以使一个任务进入挂起态，调用 `vTaskResume()` 或 `vTaskResumeFromISR()` API 函数可唤醒一个挂起态的任务。

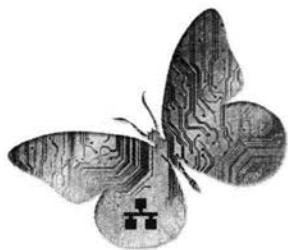
进程调度的方式通常又分为非剥夺方式和剥夺方式。非剥夺方式是指当某个进程占用处理器后便让它一直占用，直到进程完成或发生进程调度某事件而阻塞时，才将处理器让渡给另一个进程。而剥夺方式是指不管当前处理器资源是否空闲，当一个进程请求占用处理器资源时，系统可以基于某种原则，剥夺正在占有处理器资源的进程，使其退出运行态。剥夺原则有：优先权原则、短进程优先原则、时间片原则。

实时操作系统与其他操作系统的不同在于对外部事件的响应速度要求较为严格，通常要求在规定的严格时间内完成对该事件的处理。实时操作系统又分为硬实时操作系统和软实时操作系统，其中硬实时操作系统是指对事件响应的时限有明确要求的系统，任何一个实时任务都必须在时限之前完成，而软实时操作系统的要求则相对宽松。

6.2.7 存储管理

存储管理子系统是操作系统中最重要的组成部分之一。其主要功能包括：第一，记录存储器的使用情况，包括对使用部分和空闲部分的记录；第二，内存空间的分配，当有进程请求分配内存空间，且条件满足时予以分配；第三，当多个进程请求内存空间，而现有内存空间又无法同时满足所有进程要求时，依据某种策略将某个或部分进程存入磁盘，另一部分装入内存，完成内存空间的交换。

存储管理涉及交换技术、虚拟存储管理等方面的内容。读者有兴趣可参考相关书籍。FreeRTOS 提供了三种较为简单的常见内存管理策略供人们学习参考（在第 7 章有详细描述），由于工程需求的差异性，通常人们对内存管理也有着不同的要求。因此，很多工程不使用官方提供的内存管理策略，把内存管理的实现也作为 FreeRTOS 移植或系统开发的一部分。



第 7 章

FreeRTOS 嵌入式操作系统

FreeRTOS 作为一个轻量级嵌入式操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种嵌入式控制器上实现满足用户需求的应用。此外，无论商业应用还是个人学习，都无需商业授权，FreeRTOS 是完全免费的操作系统。

7.1 FreeRTOS 特色

FreeRTOS 作为一个轻量级嵌入式操作系统，提供一个高层次的可信任代码。源代码以 C 开发，系统实现的任务没有数量的限制，FreeRTOS 内核支持优先级调度算法，每个任务可根据重要程度的不同赋予一定的优先级，CPU 总是让处于就绪态的、优先级最高的任务先运行。FreeRTOS 内核同时支持轮转调度算法，系统允许不同的任务使用相同的优先级，在没有更高优先级任务就绪的情况下，同一优先级的任务共享 CPU 的使用时间。

此外，FreeRTOS 还具有强大的执行跟踪功能、堆栈溢出检测、互斥信号量、优先继承权等特点，在嵌入式操作系统中是为数不多的同时具有实时性、开源性、可靠性、易用性、多平台支持等特点的嵌入式操作系统。

FreeRTOS 提供的功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能等，可基本满足较小系统的需要。

7.2 任务管理

在嵌入式操作系统任务级应用程序开发中，如何创建任务、分配任务的处理时间和优先级以及掌握任务优先级对系统行为的影响是每个嵌入式软件工程师必须掌握的基本技能。本节将讲述如何实现一个或多个具体的任务、任务创建函数的参数对系统行为的影响、如何动态改变一个任务的优先级、任务的删除及周期性事务的处理等。

7.2.1 任务函数

在 FreeRTOS 中，任务是由 C 语言函数实现的。唯一特别的只是任务的函数原型，其必须返回 void，而且带有一个 void 指针参数。其函数原型参见程序清单 7.1。

程序清单 7.1 任务函数原型

```
void ATaskUser( void *pvParameters );
```

每个任务都是一个功能相对独立的程序。任务具有程序入口，通常运行在一个死循环中，永远不会退出，也不返回任何结果。任务不应执行到实现任务代码的末尾，必要时，应在末尾添加报错功能的代码，即当程序运行到此处时，以某种方式通知用户出错。此外，通常在此处添加一个删除任务的函数，以确保出错时该任务不致影响其他任务的运行，如程序清单 7.2 所示。

程序清单 7.2 典型的任务函数结构

```
void ATaskUser( void *pvParameters )
{
    /* 可以像普通函数一样定义变量。用这个函数创建的每个任务实例都有一个属于
       自己的 iVariableExample 变量。但如果 iVariableExample 被定义为 static，
       这一点则不成立——这种情况下只存在一个变量，所有的任务实例将会共享这个变量 */
    int iVariableExample = 0;

    /* 任务通常实现在一个死循环中 */
    for( ;; )
    {
        /* 完成任务功能的代码将放在这里 */
    }
    /* 如果任务的具体实现会跳出上面的死循环，则此任务必须在函数运行完之前删除。传入 NULL 参数表示删除
       的是当前任务 */
    vTaskDelete( NULL );
}
```

在 FreeRTOS 中，任务的创建方式比较灵活，任务函数既可以在系统处理器初始化后的主函数部分创建，也可以在具体的任务中创建。无论以何种方式创建的任务都拥有独自の栈空间，以及属于自己的自动变量，即任务函数本身定义的变量。需要注意的是，在编程时，应最大程度地避免任务之间的耦合，如不使用全局变量，这样可使代码更具可读性和可维护性。

7.2.2 基本任务状态

对于单核处理器来说，运行于内核的多个任务不可能同时获得处理器的服务，在任意给定时间，仅会执行一个任务。这就意味着当一个任务占有处理器资源时，其他任务只能等待内核下次为其分配处理时间。那么，当前占用处理器资源的任务就处于运行态，而其他任务则处于非运行态。实际上，处理器的非运行态根据其自身特点，又可划分为多种状态，这里仅就基于简化问题将其进行简单划分，稍后章节将对非运行态进行详细分析和阐述。如图 7.1 所示。

当某个任务处于运行态时，处理器就正在执行它的代码。当一个任务处于非运行态时，该任务进行休眠，它的所有状态都被妥善保存，以便在下次调试器决定让它进入运行态时可以恢复执行。当任务恢复执行时，其将精确地从离开运行态时正准备执行的那一条指令开始执行。任务从非运行态转移到运行态被称为切入或换入。相反，任务从运行态转移到非运行态被称为切出或换出。FreeRTOS 的调度器是能让任务切入切出的唯一实体。

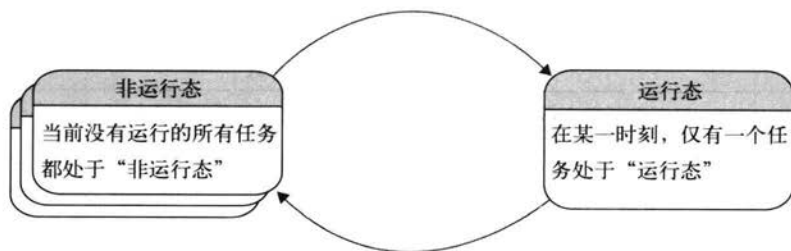


图 7.1 基本任务状态及状态转移

7.2.3 任务创建

xTaskCreate() API 函数

创建任务是所有嵌入式操作系统中最重要和复杂的内容之一，初学者看到众多的传参往往望而生畏。但对于大多数的学生朋友或工程师而言，深究这些函数的实现细节往往并不重要，只要熟悉这些参数对任务行为的影响就可以满足平时开发工作所需了。本书也是基于这样的目的，结合代码实例浅显地向读者介绍 FreeRTOS 的 API 函数使用方法。

FreeRTOS 采用 xTaskCreate() 函数创建任务，函数原型如程序清单 7.3 所示。该 API 函数共有五个传参，这可能是所有 API 函数中最复杂的函数，也是我们必须首先掌握的函数，因为它是多任务系统中最基本的组件。本书中所有示例程序都会用到 xTaskCreate()，参数具体功能如表 7.1 所示。

程序清单 7.3 xTaskCreate() API 函数原型

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask );
```

表 7.1 xTaskCreate() 参数描述

参 数 名	描 述
pvTaskCode	任务只是永不退出的 C 函数，以死循环实现。参数 pvTaskCode 为指向任务的实现函数的指针，与函数名相同
pcName	<input type="checkbox"/> 任务名描述。该参数对任务行为无任何影响，仅仅作为任务的标识 <input type="checkbox"/> FreeRTOSConfig.h 文件中的 config_MAX_TASK_NAME_LEN 宏定义限定了任务名的最大长度（包括 '\0' 结束符）。读者可根据需要修改
usStackDepth	<input type="checkbox"/> 内核在创建任务时为其分配唯一的堆栈空间。usStackDepth 指示内核为其分配栈空间的容量 <input type="checkbox"/> 该参数的单位为字（4 字节），若此处为 10，则实际分配的堆栈空间为 40 字节 <input type="checkbox"/> FreeRTOSConfig.h 文件中的 configMINIMAL_STACK_SIZE 宏定义限制了空闲任务的堆栈容量。FreeRTOS 提供的 Demo 通常是对所有任务的最小建议值 <input type="checkbox"/> 建议堆栈空间的大小应根据实际需求的预测并结合实际运行测试的结果来设置

(续)

参 数 名	描 述
pvParameters	void 类型指针 (void*)，pvParameters 的值将传递到任务函数的传参
uxPriority	<ul style="list-style-type: none"> ❑ 任务执行的优先级，取值范围：0 ~ (configMAX_PRIORITIES-1) ❑ configMAX_PRIORITIES 通过 FreeRTOSConfig.h 文件中的宏定义设置，是一个由用户定义的常量。建议设置为满足实际需要的最小数值，以避免内存浪费若 uxPriority 的值超过了 (configMAX_PRIORITIES-1)，任务的优先级被自动设置为 (configMAX_PRIORITIES-1)
pxCreatedTask	<ul style="list-style-type: none"> ❑ pxCreatedTask 为任务的句柄。其他 API 可通过该句柄对该任务进行引用，如改变任务优先级或者删除任务 ❑ 如果不使用该句柄，可将此参数设置为 NULL
返回值	返回值为 pdTRUE 或 errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY。前者代表创建成功；后者表示由于堆栈空间不足，任务创建失败

例 1 创建任务

本例以打印字符串为例，演示如何创建任务。任务延时采用原始的空循环方式来实现。两者在创建时指定了相同的优先级，堆栈深度简单地设置为 1000。程序清单 7.4 和程序清单 7.5 是这两个任务对应的实现代码。

程序清单 7.4 例 1 中的第一个任务实现代码

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "How are you ?\r\n";
    volatile unsigned long ul;
    /* 与大多数任务一样，该任务处于一个死循环中 */
    for( ;; )
    {
        /* 打印任务的名字 */
        vPrintString( pcTaskName );

        /* 延迟，以产生一个周期 */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* 这个空循环是最原始的延迟实现方式。在循环中不做任何事情 */
        }
    }
}
```

程序清单 7.5 例 1 中的第二个任务实现代码

```
void vTask2( void *pvParameters )
{
    const char *pcTaskName = "Fine, Thank you \r\n";
    volatile unsigned long ul;

    /* 与大多数任务一样，该任务处于一个死循环中 */
}
```

```

for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* 延迟，以产生一个周期 */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* 这个空循环是最原始的延迟实现方式。在循环中不做任何事情 */
    }
}
}

```

main() 函数代码参见程序清单 7.6，main() 函数首先创建这两个任务，然后启动调度器。

程序清单 7.6 启动例 1 中的任务

```

int main( void )
{
    /* 创建第一个任务。需要说明的是一个实用的应用程序中应当检测函数
    xTaskCreate() 的返回值，以确保任务创建成功 */
    xTaskCreate( vTask1,      /* 指向任务函数的指针 */
                "Task 1",    /* 任务的文本名字，只会在调试中用到 */
                1000,        /* 栈深度——大多数小型微控制器会使用的 */
                NULL,        /* 值会比此值小得多 */
                NULL,        /* 没有任务参数 */
                1,           /* 此任务运行在优先级 1 上 */
                NULL );      /* 不会用到任务句柄 */

    /* 以相同的方式创建另一个任务，并具有相同的优先级 */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* 启动调度器，任务开始执行 */
    vTaskStartScheduler();

    /* 正常情况下，main() 函数不应执行到此处 */
    for( ;; );
}

```

在开发环境中，我们可以看到代码的实际运行效果。运行的结果看似两个任务同时执行，但实际上是调度器快速地将两个任务不间断地在运行态和非运行态间进行切换，因此，两个任务看似同时都得到了执行，这也是操作系统的基本功能之一。任务的执行流程如图 7.2 所示。

图 7.2 中左侧向下的箭头表示从 t_1 起始的运行时刻。灰色方块表示每个运行的时间段，如 t_1 与 t_2 之间运行的是任务 1。在任何时刻只可能有一个任务处于运行态。所以一个任务进入运行态后（切入），另一个任务就会进入非运行态（切出）。

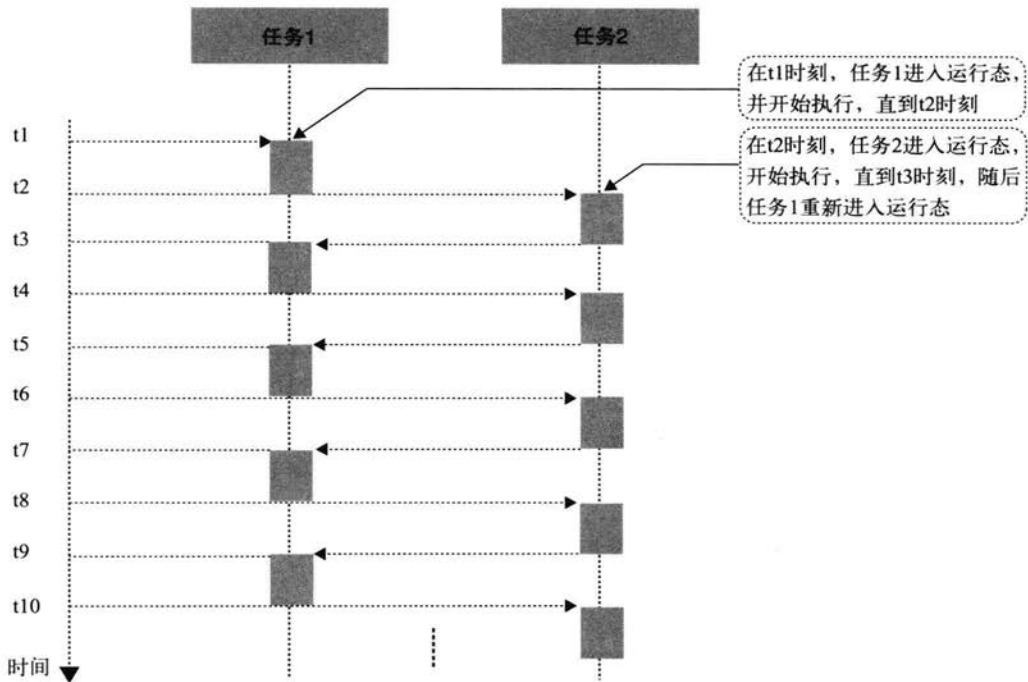


图 7.2 例 1 的实际执行流程

本例介绍了在 `main()` 函数主体中创建两个具体的任务，启动调度器后，两个任务得以执行。现在，我们以另一种方式创建任务，即在一个任务中创建另一个任务。首先在 `main()` 中创建任务 1，然后在任务 1 中创建任务 2。任务 1 的代码如程序清单 7.7 所示。这种方式与前面任务创建方式的区别在于，任务 2 在调度器启动之间还没有创建，当调度器启动后由任务 1 创建。但是整个程序运行的效果相同，这种方式也用在工程中动态地创建或删除任务。

程序清单 7.7 在任务中创建另一个任务

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile unsigned long ul;

    /* 如果已经执行到本任务的代码，表明调度器已经启动。在进入死循环之前创建
       另一个任务 */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );
    for( ;; )
    {
        /* 打印任务的名字 */
        vPrintString( pcTaskName );

        /* 延时一定时间 */
    }
}
```

```

for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
{
    /* 正常情况下, main() 函数不应执行到此处 */
}
}

```

例2 任务参数

细心的读者可能会发现, 例1中创建的两个任务除了打印的字符串外, 其余代码几乎完全相同。那么, 有没有其他方式可以避免这种重复性的工作呢? 答案是肯定的, 我们可以通过任务参数的传递实现字符串的打印。

程序清单 7.8 是与例1类似的一个任务函数代码实现。该任务函数代替了例1中的两个任务函数 (vTask1 与 vTask2)。需要注意的是, 这个函数的任务参数被强制转化为 char* 以得到任务需要打印输出的字符串。事实上 void 类型的指针可以强制转换为任何类型的指针, 因此也被称为万能指针。

程序清单 7.8 创建两个任务实例的任务函数

```

void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile unsigned long ul;

    /* 需要打印输出的字符串从入口参数传入。强制转换为字符指针 */
    pcTaskName = ( char * ) pvParameters;

    /* 与大多数的任务相同, 该任务实现为一个无限循环 */
    for( ;; )
    {
        /* 打印任务的名字 */
        vPrintString( pcTaskName );

        /* 延时一定时间 */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* 该循环并没有做具体的事务, 仅仅是一个原始的延时实现 */
        }
    }
}

```

尽管现在只有一个任务函数实现 (vTaskFunction), 但是可以通过该函数创建多个任务实例。每个任务实例都可以在 FreeRTOS 调度器的控制下独立运行。xTaskCreate() 的参数 pvParameters 将传递给任务函数, 用于传入字符串文本。具体代码如程序清单 7.9 所示。

程序清单 7.9 例2中的 main() 函数实现代码

```

/* 定义将要通过任务参数传递的字符串 */
static const char *pcTextForTask1= "Task 1 is running\r\n";

```

```

static const char *pcTextForTask2 ="Task 2 is running\t\n";
int main( void )
{
    /* 创建两个任务中的一个 */
    xTaskCreate( vTaskFunction,          /* 指向任务函数的指针 */
                "Task 1",                /* 任务名 */
                1000,                    /* 栈深度 */
                (void*)pcTextForTask1,   /* 通过任务参数传入需要打 */
                1,                        /* 印输出的文本 */
                NULL );                  /* 任务优先级为 1 */
    /* 创建另一个任务。与任务 1 相比，仅是传入的参数不同 */
    xTaskCreate( vTaskFunction,
                "Task 2",
                1000,
                (void*)pcTextForTask2,
                1,
                NULL );

    /* 启动调度器，开始执行任务 */
    vTaskStartScheduler();
    /* 正常情况下，main() 函数不应执行到此处 */
    for( ;; );
}

```

7.2.4 任务的优先级

xTaskCreate() API 函数的参数 uxPriority 为创建的任务赋予了一个初始优先级。这个优先级可以在调度器启动后调用 vTaskPrioritySet() API 函数进行修改。应用程序在文件 FreeRTOSConfig.h 中设定的编译时配置常量 configMAX_PRIORITIES 的值，即最多可具有的优先级数目。FreeRTOS 本身并没有限定这个常量的最大值，但这个值越大，则内核花销的内存空间就越大。所以建议将此常量设为满足实际需求的最小值。

对于任务的优先级分配，FreeRTOS 支持不同任务共享同一个优先级，调度器将采用时间片轮转的方式保障同优先级的任务都得以执行，这也保证设计弹性的最大化。当然，如果需要的话，每个任务可指定唯一的优先级。优先级的取值范围为 0~(configMAX_PRIORITIES-1)，数值越大，优先级越高。

调度器保障具有最高优先级的任务优先得到执行，使其进入运行态。如果当前最高优先级上具有多个任务，调度器会让这些任务轮流执行。每个任务都执行一个“时间片”，任务在时间片起始时刻进入运行态，在时间片结束时刻又退出运行态。图 7.3 中 t1 与 t2 之间的时段就等于一个时间片，也即一个心跳周期。

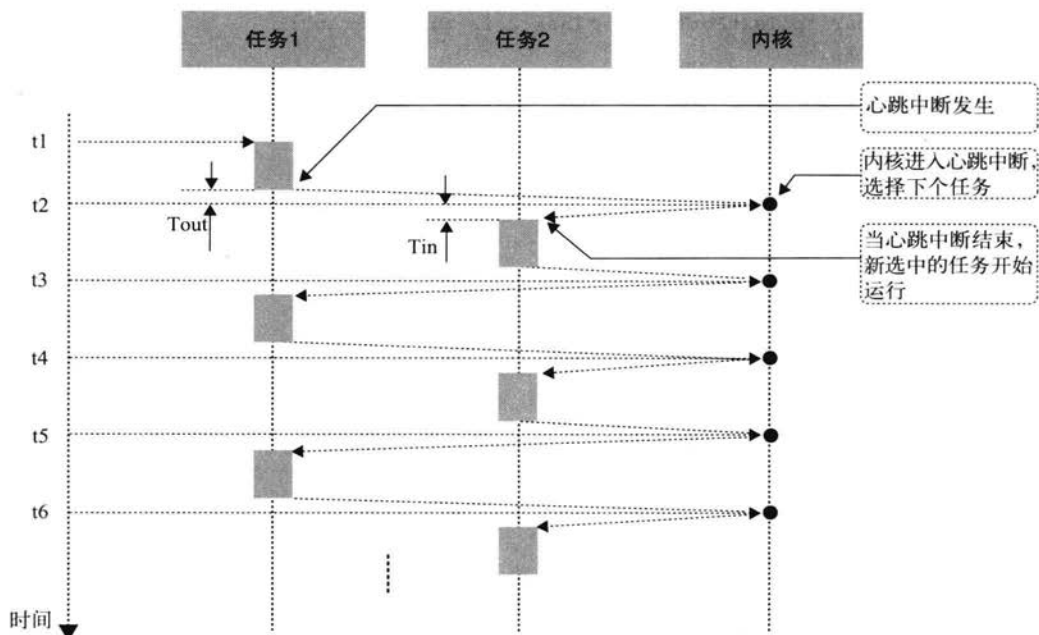


图 7.3 对执行流程进行扩展以显示心跳中断的执行

调度器在每个心跳周期结束时运行自身（实质上是进入空闲任务），选择下一个将要运行的任务，可以将图 7.3 进行扩展，将调度器本身的执行时间在整个执行流程中体现出来，请参见图 7.4。通常心跳时钟采用微处理器的滴答计数器实现，滴答计数器中断用于产生心跳时钟。时间片的长度通过心跳中断频率进行设定，心跳中断频率由 FreeRTOSConfig.h 中的编译时配置常量 configTICK_RATE_HZ 进行配置。若 configTICK_RATE_HZ 设为 100 Hz，则时间片长度为 10ms。需要说明的是，FreeRTOS 的 API 函数调用中指定的时间总是以心跳周期为单位。常量 portTICK_RATE_MS 用于将以心跳为单位的时间值转化为以毫秒为单位的时间值。有效精度依赖于系统心跳频率。心跳计数（tick count）值表示的是从调度器启动开始，心跳中断发生的次数。用户程序在指定延迟周期时不必考虑心跳计数溢出问题，因为时间连贯性可在内核中进行管理。

例 3 优先级实验

调度器总是在具备运行条件的任务中，选择具有最高优先级的任务使其进入运行态。在前面的示例程序中，因为两个任务都具有相同的优先级，所以，两个任务在运行态和非运行态不断地切入和切出。本例将创建两个不同优先级的任务，测试优先级对任务行为的影响。我们将任务 1 的优先级设置为 1，而另一个任务的优先级设置为 2。两个任务的代码见程序清单 7.10。两个任务的实现函数与前面章节相同，通过空循环产生延迟来周期性打印输出字符串。

程序清单 7.10 两个任务创建在不同的优先级上

```
/* 定义将要通过任务参数传递的字符串 */
static const char *pcTextForTask1="Task 1 is running\r\n";
```

```

static const char *pcTextForTask2 = "Task 2 is running\t\n";
int main( void )
{
    /* 任务1 创建在优先级1上 */
    xTaskCreate( vTaskFunction,
                "Task 1",
                1000,
                (void*)pcTextForTask1,
                1,
                NULL );

    /* 任务2 创建在优先级2上 */
    xTaskCreate( vTaskFunction,
                "Task 2",
                1000,
                (void*)pcTextForTask2,
                2,
                NULL );

    /* 启动调度器, 开始执行 */
    vTaskStartScheduler();
    return 0;
}

```

调度器总是选择具有最高优先级的可运行任务来执行。由于任务2的优先级比任务1高，并且总是可运行，因此任务2一直处于运行态，而任务1始终处于非运行态。这种情况称为任务1的执行时间被任务2“饿死”（starved）。任务2之所以总是可运行，是因为其不需要获取和其他任务共享的互斥性资源。图7.4展现了例3的执行流程。

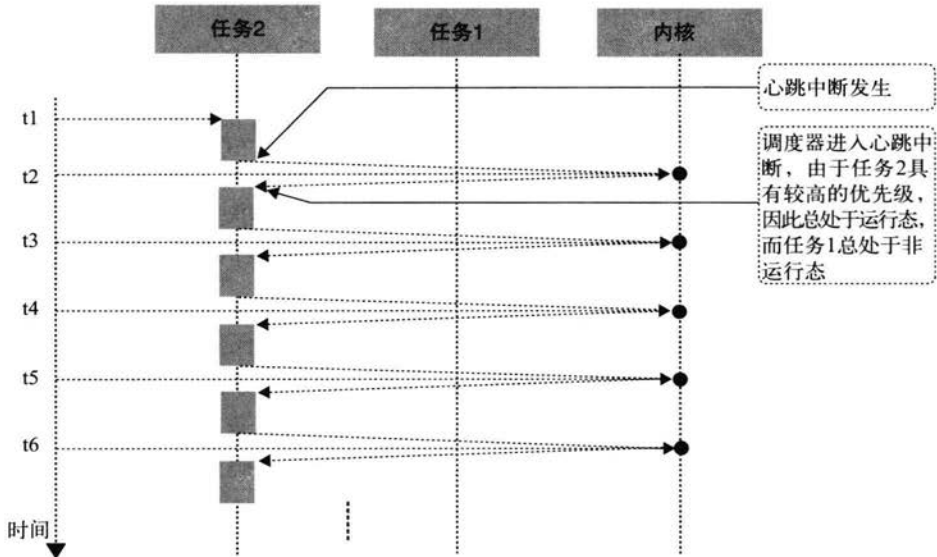


图 7.4 当一个任务优先级比另一个高时的执行流程

7.2.5 非运行状态

在前面的章节中，我们简单地将任务所处的状态分为运行态和非运行态。事实上，前面创建的任务都只是闭门造车，既不获取外界资源，也不需要等待某种信息。在现实中，这样的任务是缺乏实际应用价值的。即便使用，通常也被创建在最低优先级上，以防具有优先级更低的任务得不到执行。

为了使任务更具实用性，我们让任务以事件驱动的方式运行，以提高任务执行的效率。一个事件驱动任务只会在事件发生后触发，开始处理自己的事务，而在事件没有发生时不能进入运行态。调度器总是选择所有能够进入运行态的任务中具有最高优先级的任务。一个高优先级但不能运行的任务意味着不会被调度器选中，取而代之以另一个优先级虽然更低但能够运行的任务。因此，采用事件驱动任务的意义就在于任务可以被创建在许多不同的优先级上，并且最高优先级任务不会把所有的低优先级任务饿死。

1. 阻塞态

如果一个任务正在等待某个事件，则称这个任务处于“阻塞态”(blocked)。阻塞态是非运行态集的一个子状态。

任务可以进入阻塞态以等待以下两种不同类型的事件：

- ❑ 定时事件——这类事件可以是延迟到期或是绝对时间到点。例如，某个任务延迟 10ms 进入阻塞态。
- ❑ 同步事件——等待其他任务或中断的事件。比如，某个任务可以进入阻塞态以等待队列中有数据到来。

消息队列、二值信号量、计数信号量、互斥信号量、递归信号量和互斥信号量都可以用来实现同步事件，后续章节将详细叙述。

任务可以在进入阻塞态以等待同步事件的同时指定一个等待时限，这样可以有效地实现阻塞态下同时等待两种类型的事件。比如说，某个任务可以等待队列中有数据到来，时限为 10ms，若 10ms 内有数据到来或 10ms 后没有数据到来，任务都将退出阻塞态。

2. 挂起态

“挂起态”(suspended)也是非运行态集的子状态。调度器对处于挂起态的任务不做任何处理，除非其他任务或中断将其唤醒。让一个任务进入挂起态的唯一办法就是调用 `vTaskSuspend()` API 函数；而把一个挂起态的任务唤醒的唯一途径就是调用 `vTaskResume()` 或 `vTaskResumeFromISR()` API 函数。大多数应用程序中都不会用到挂起态。

3. 就绪态

如果任务处于非运行态，但既没有阻塞也没有挂起，则这个任务处于“就绪态”(ready)。处于就绪态的任务具备运行条件，等待调度器调度。

4. 状态转移图

图 7.5 包含了本节描述的非运行态的子状态，是一个较为完备的任务状态转移图。前面章节的示例程序仅没有涉及阻塞态和挂起态，图 7.5 描述了这些状态的转移条件，其中用粗线标示了

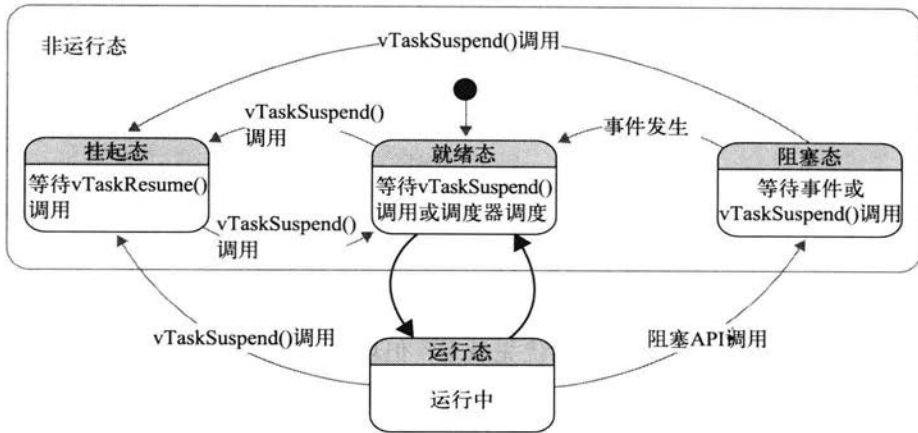


图 7.7 粗线条表示例 4 中的状态转移过程

vTaskDelayUntil() API 函数

vTaskDelayUntil() 与 vTaskDelay() 类似。函数 vTaskDelay() 的参数用来指定任务在调用 vTaskDelay() 到切出阻塞态整个过程包含多少个心跳周期。任务保持在阻塞态的时间量由 vTaskDelay() 的入口参数指定，但任务离开阻塞态的时刻实际上是相对于 vTaskDelay() 被调用那一刻的。vTaskDelayUntil() 的参数就是用来指定任务离开阻塞态进入就绪态那一刻的精确心跳计数值。vTaskDelayUntil() API 函数可以用于实现一个固定执行周期的需求，如定时采样等应用。由于调用此函数的任务解除阻塞的时间是绝对时刻，比起相对于调用时刻的相对时间更精确，即比调用 vTaskDelay() 可以实现更精确的周期性。vTaskDelayUntil() 的函数原型如程序清单 7.13 所示，各参数及描述见表 7.3。

程序清单 7.13 vTaskDelayUntil() API 函数原型

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime,
                    portTickType xTimeIncrement );
```

表 7.3 vTaskDelayUntil() 参数

参 数	描 述
pxPreviousWakeTime	<ul style="list-style-type: none">❑ 此参数命名时假定 vTaskDelayUntil() 用于实现某个任务以固定频率周期性执行。pxPreviousWakeTime 保存了任务上一次离开阻塞态的时刻，用作一个参考点来计算该任务下一次离开阻塞态的时刻❑ pxPreviousWakeTime 指向的变量值会在 API 函数 vTaskDelayUntil() 调用过程中自动更新
xTimeIncrement	<ul style="list-style-type: none">❑ 此参数命名时同样假定 vTaskDelayUntil() 用于实现某个任务以固定频率周期性执行——该频率由 xTimeIncrement 指定❑ xTimeIncrement 的单位是心跳周期，可以使用常量 portTICK_RATE_MS 将毫秒转换为心跳周期

例5 使用 vTaskDelayUntil() 转换示例任务

例4中的两个任务都是周期性任务，但是使用 vTaskDelay() 无法保证它们具有固定的执行频率，因为这两个任务退出阻塞态的时刻相对于调用 vTaskDelay() 的时刻。通过调用 vTaskDelayUntil() 来代替 vTaskDelay()，可以实现精确地周期执行。具体代码见程序清单 7.14。

程序清单 7.14 使用 vTaskDelayUntil() 实现示例任务

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    portTickType xLastWakeTime;

    /* 将要打印的字符串以传递参数形式传递给任务，参数为字符串指针类型 */
    pcTaskName = ( char * ) pvParameters;

    /* 变量 xLastWakeTime 需要被初始化为当前心跳计数值。注意，这是该变量唯一一次被显式赋值。之后，
    xLastWakeTime 将在函数 vTaskDelayUntil() 中自动更新 */
    xLastWakeTime = xTaskGetTickCount();

    /* 与大多数的任务相同，该任务实现为一个无限循环 */
    for( ;; )
    {
        /* 打印任务的名字 */
        vPrintString( pcTaskName );

        /* 本任务将精确地以 250ms 为周期执行。同 vTaskDelay() 函数一样，时间值是以心跳周期为单位的，
        可以使用常量 portTICK_RATE_MS 将毫秒转换为心跳周期 */
        vTaskDelayUntil( &xLastWakeTime,
                        (250/portTICK_RATE_MS ) );
    }
}
```

例6 合并阻塞与非阻塞任务

之前的范例分别测试了任务以查询方式和阻塞方式工作的系统行为。本例通过合并这两种方案的执行流程，再次实现具有既定预期的系统行为。

本例在优先级 1 上创建 2 个任务。这两个任务没有调用任何可能导致它们进入阻塞态的 API 函数，只是不停地打印输出字符串。因此这两个任务要么处于就绪态，要么处于运行态。具有这种性质的任务称为“持续处理”任务，这在本例中没什么意义，仅为了说明该种系统行为。持续处理任务的代码参见程序清单 7.15。第三个任务创建在优先级 2 上，高于另外两个任务的优先级。这个任务调用 vTaskDelayUntil() 周期性地打印输出字符串，在每两次打印之间让自己处于阻塞态。

周期性任务的实现代码参见程序清单 7.16。图 7.8 是对看到的行为方式对应的执行流程的解释。

程序清单 7.15 例6中持续处理任务的实现代码

```
void vContinuousProcessingTask( void *pvParameters )
{
    char *pcTaskName;
```

```

/* 打印输出的字符串由任务参数传入，强制转换为 char* */
pcTaskName = ( char * ) pvParameters;

/* 与大多数的任务相同，该任务实现为一个无限循环 */
for( ;; )
{
    /* 打印输出任务名，无阻塞，也无延迟 */
    vPrintString( pcTaskName );
}
}

```

程序清单 7.16 例 6 中周期任务的实现代码

```

void vPeriodicTask( void *pvParameters )
{
    portTickType xLastWakeTime;

    /* 初始化 xLastWakeTime，之后会在 vTaskDelayUntil() 中自动更新 */
    xLastWakeTime = xTaskGetTickCount();

    /* 与大多数的任务相同，该任务实现为一个无限循环 */
    for( ;; )
    {
        /* 打印输出任务名 */
        vPrintString( "Periodic task is running\r\n" );

        /* 该任务每 10ms 执行 1 次 */
        vTaskDelayUntil( &xLastWakeTime, (10/portTICK_RATE_MS) );
    }
}

```

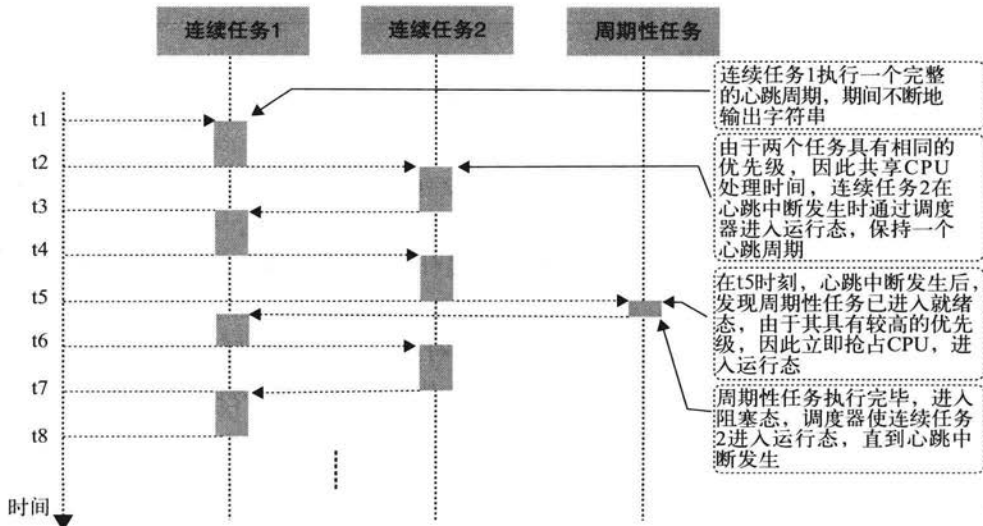


图 7.8 例 6 的执行流程

7.2.6 空闲任务及回调函数

例4中创建的任务大部分时间都处于阻塞态。在这种情况下，所有的任务都处于非运行状态，调度器会自动创建一个空闲任务。空闲任务是一个非常短小的循环，总是处于就绪态。空闲任务拥有最低优先级，以保证具有更高优先级的应用任务能够进入运行态。空闲任务运行在最低优先级，这可以保证一旦有更高优先级的任务进入就绪态，空闲任务可以立即切出运行态。

1. 空闲任务回调函数

通过空闲任务回调函数（直译为钩子函数，下文统一称为回调函数），可以直接在空闲任务中添加应用程序相关的功能。空闲任务回调函数会在空闲任务执行时调用。

通常空闲任务回调函数被用于：

- ❑ 执行低优先级，后台或需要不停处理的功能代码。
- ❑ 系统性能统计，通过测量空闲任务占用的处理时间计算系统的处理时间裕量。
- ❑ 降低功耗，使得在没有任何应用功能需要处理的时候，系统自动进入低功耗模式。

2. 空闲任务回调函数的实现限制

空闲任务回调函数必须遵从以下规则：

- ❑ 绝不能阻塞或挂起。空闲任务只会在其他任务都不运行时才会被执行。以任何方式阻塞空闲任务都可能导致没有任务能够进入运行态。
- ❑ 如果应用程序用到了 `vTaskDelete()` API 函数，则空闲回调函数必须能够尽快返回。因为在任务被删除后，空闲任务负责回收内核资源。如果空闲任务一直运行在回调函数中，则无法进行回收工作。

空闲任务回调函数必须具有程序清单 7.17 所示的函数名和函数原型。

程序清单 7.17 空闲任务回调函数原型

```
void vApplicationIdleHook( void );
```

例7 定义一个空闲任务回调函数

例4调用了带阻塞性质的 `vTaskDelay()` API 函数，会产生大量的空闲时间——在这期间空闲任务会得到执行，因为两个应用任务均处于阻塞态。本例通过空闲回调函数来使用这些空闲时间。具体代码参见程序清单 7.18。

程序清单 7.18 一个非常简单的空闲回调函数

```
/* 声明一个在回调函数中使用的变量 */
unsigned long ulIdleCycleCount = 0UL;

/* 空闲回调函数必须命名为 vApplicationIdleHook(), 无参数也无返回值 */
void vApplicationIdleHook( void )
{
    /* 该回调函数仅递增一个计数器 */
    ulIdleCycleCount++;
}
```

FreeRTOSConfig.h 中的配置常量 configUSE_IDLE_HOOK 必须定义为 1，这样空闲任务回调函数才会被调用。本例对应用任务实现函数进行了少量修改，用以打印输出变量 ulIdleCycleCount 的值，如程序清单 7.19 所示。

程序清单 7.19 示例任务现在用于打印输出 ulIdleCycleCount 的值

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* 将要打印的字符串以传递参数形式传递给任务，参数为字符串指针类型 */
    pcTaskName = ( char * ) pvParameters;

    /* 与大多数的任务相同，该任务实现为一个无限循环 */
    for( ;; )
    {
        /* 打印输出任务名，以及调用计数器 ulIdleCycleCount 的值 */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* 延时周期：250ms */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

7.2.7 改变任务优先级

vTaskPrioritySet() API 函数

vTaskPrioritySet() API 函数可以用于在调度器启动后改变任何任务的优先级。函数原型见程序清单 7.20，各参数及描述见表 7.4。

程序清单 7.20 vTaskPrioritySet() API 函数原型

```
void vTaskPrioritySet( xTaskHandle pxTask,
                      unsigned portBASE_TYPE uxNewPriority );
```

表 7.4 vTaskPrioritySet() 参数

参 数	描 述
pxTask	<ul style="list-style-type: none">❑ 被修改优先级的任务句柄（即目标任务）——参考 xTaskCreate() API 函数的参数 pxCreatedTask，以了解如何得到任务句柄方面的信息❑ 任务可以通过传入 NULL 值来修改自己的优先级
uxNewPriority	目标任务将被设置到哪个优先级上。如果设置的值超过了最大可用优先级 (configMAX_PRIORITIES-1)，则会被自动封顶为最大值。常量 configMAX_PRIORITIES 是在 FreeRTOSConfig.h 头文件中设置的一个编译时选项

uxTaskPriorityGet() API 函数

uxTaskPriorityGet() API 函数用于查询一个任务的优先级。函数原型见程序清单 7.21，各参数及描述见表 7.5。

程序清单 7.21 uxTaskPriorityGet() API 函数原型

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

表 7.5 uxTaskPriorityGet() 参数及返回值

参数	描 述
pxTask	<ul style="list-style-type: none"> ❑ 被查询任务的句柄（目标任务）——参考 xTaskCreate() API 函数的参数 pxCreatedTask，以了解如何得到任务句柄方面的信息 ❑ 任务可以通过传入 NULL 值来查询自己的优先级
返回值	被查询任务的当前优先级

例 8 改变任务优先级

调度器总是在所有就绪态任务中选择具有最高优先级的任务，并使其进入运行态。本例即是通过调用 vTaskPrioritySet() API 函数来改变两个任务的相对优先级，以达到对调度器这一行为的演示。

在不同的优先级上创建两个任务。这两个任务都没有调用任何会令其进入阻塞态的 API 函数，所以这两个任务要么处于就绪态，要么处于运行态——在这种情形下，调度器选择具有最高优先级的任务来执行。

例 8 具有以下行为方式：

- ❑ 任务 1（程序清单 7.22）创建在最高优先级，以保证其可以最先运行。任务 1 首先打印输出两个字符串，然后将任务 2（程序清单 7.23）的优先级提升到自己之上。
- ❑ 任务 2 一旦拥有最高优先级便进入运行态。由于任何时候只可能有一个任务处于运行态，所以当任务 2 运行时，任务 1 处于就绪态。
- ❑ 任务 2 打印输出一个信息，然后把自己的优先级设回低于任务 1 的初始值。
- ❑ 任务 2 降低自己的优先级意味着任务 1 又成为具有最高优先级的任务，所以任务 1 重新进入运行态，任务 2 被强制切入就绪态。

程序清单 7.22 例 8 中任务 1 的实现代码

```
void vTask1( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;

    /* 任务 1 将会比任务 2 更先运行，因为任务 1 创建在更高的优先级上。任务 1 和任务 2 都不会阻塞，所以两者
    要么处于就绪态，要么处于运行态。查询任务 1 当前运行的优先级——传递一个 NULL 值，表示说“返回我自己的
    优先级” */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
```

```

{
    /* 打印输出任务的名字 */
    vPrintString( "Task1 is running\r\n" );

    /* 把任务 2 的优先级设置高于任务 1 的优先级, 这会使得任务 2 立即得到执行 (因为任务 2 现在是在所有任务中具有最高优先级的任务)。注意调用 TaskPrioritySet() 时用到的任务 2 的句柄。程序清单 7.24 将展示如何得到这个句柄 */
    vPrintString( "About to raise the Task2 priority\r\n" );
    vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

    /* 本任务只会在其优先级高于任务 2 时才会得到执行。因此, 当此任务运行到这里时, 任务 2 必然已经执行过了, 并且将其自身的优先级设置回比任务 1 更低的优先级 */
}
}

```

程序清单 7.23 例 8 中的任务 2 实现代码

```

void vTask2( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;

    /* 任务 1 比任务 2 更早启动, 因为任务 1 创建在更高的优先级。任务 1 和任务 2 都不会阻塞, 所以两者要么处于就绪态, 要么处于运行态。查询任务 2 当前运行的优先级——传递一个 NULL 值, 表示说“返回我自己的优先级” */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* 当任务运行到这里, 任务 1 必然已经运行过了, 并将其自身的优先级设置到高于任务 1 本身 */
        vPrintString( "Task2 is running\r\n" );

        /* 将自己的优先级设置回原来的值。传递 NULL 句柄值意味“改变我自己的优先级”。把优先级设置到低于任务 1, 使得任务 1 立即得到执行——任务 1 抢占本任务 */
        vPrintString( "About to lower the Task2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}

```

任务在查询和修改自己的优先级时, 并没有使用一个有效的句柄——使用 NULL 代替。只有在某个任务需要引用其他任务的时候才会用到任务句柄。好比任务 1 想要改变任务 2 的优先级, 为了让任务 1 能够使用任务 2 的句柄, 在创建任务 2 时其句柄就被获得并保存下来, 就像程序清单 7.24 注释中重点提示的一样。

程序清单 7.24 例 8 中 main() 函数实现代码

```

/* 声明变量用于保存任务 2 的句柄 */
xTaskHandle xTask2Handle;
int main( void )
{
    /* 任务 1 创建在优先级 2 上。任务参数没有用到, 设为 NULL。任务句柄也不会用到, 也设为 NULL */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
}

```


程序清单 7.25 vTaskDelete() API 函数原型

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

表 7.6 vTaskDelete() 参数

参 数	描 述
pxTaskToDelete	<ul style="list-style-type: none"> □ 被删除任务的句柄（目标任务）—— 参考 xTaskCreate() API 函数的参数 pxCreatedTask □ 参数为 NULL 时，删除当前任务

例 9 删除任务

这是一个非常简单的范例，其行为如下：

- 任务 1 创建在优先级 1 上。任务 1 运行时，以优先级 2 创建任务 2。因此现在任务 2 具有最高优先级，所以会立即得到执行。main() 函数的代码参见程序清单 7.26，任务 1 的实现代码参见程序清单 7.27。
- 任务 2 仅是通过自己的任务句柄删除自身。当然，也可以通过传递 NULL 值以 vTaskDelete() 来删除自身。任务 2 的实现代码见程序清单 7.28。
- 当任务 2 删除自己后，任务 1 成为最高优先级的任务，所以继续执行，调用 vTaskDelay() 阻塞一段时间。
- 当任务 1 进入阻塞态后，空闲任务得到执行的机会。空闲任务会释放内核为已删除的任务 2 分配的内存。
- 任务 1 离开阻塞态后，再一次成为就绪态中具有最高优先级的任务，因此会抢占空闲任务。再一次创建任务 2，如此往复。

程序清单 7.26 例 9 中的 main() 函数实现

```
int main( void )
{
    /* 任务 1 创建在优先级 1 上 */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* 启动调度器，开始执行任务 */
    vTaskStartScheduler();

    /* main() 不应执行到此处，除非发生错误 */
    for( ;; );
}
```

程序清单 7.27 例 9 中任务 1 的实现代码

```
void vTask1( void *pvParameters )
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;
    for( ;; )
    {
```

```

/* 打印输出任务的名字 */
vPrintString( "Task1 is running\r\n" );

/* 创建任务2为最高优先级 */
xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

/* 因为任务2具有最高优先级, 所以任务1运行到这里时, 任务2已经完成执行, 删除了自己。任务1得以执行, 延迟100ms */
vTaskDelay( xDelay100ms );
}
}

```

程序清单 7.28 例9中的任务2实现代码

```

void vTask2( void *pvParameters )
{
    /* 任务2什么也没做, 只是删除自己。删除自己可以传入 NULL 值, 这里为了演示, 传入其自己的句柄 */
    vPrintString( "Task2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}

```

图 7.10 展示了例 9 的执行流程。

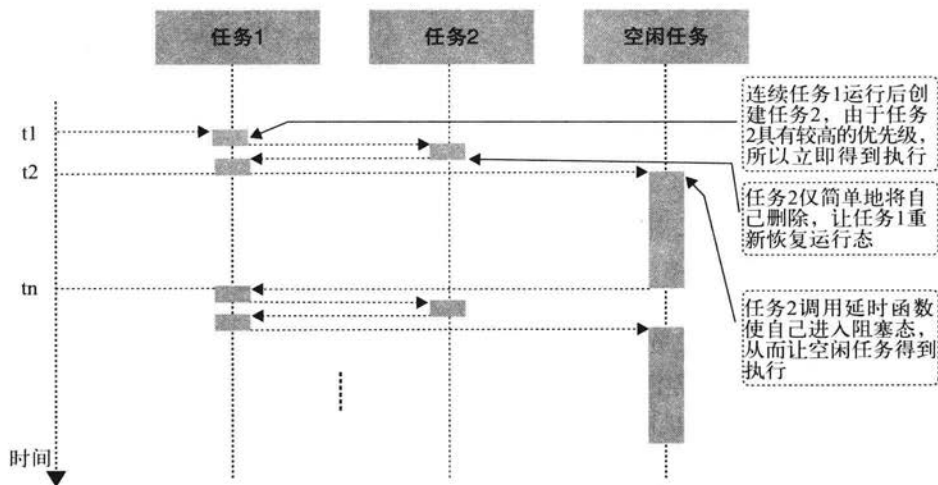


图 7.10 例9 执行流程

7.2.9 调度算法概述

1. 优先级抢占式调度

FreeRTOS 的调度方式被称为“固定优先级抢占式调度”。所谓“固定优先级”是指每个任务都被赋予了一个优先级, 这个优先级不能被内核本身改变。“抢占式”是指当任务进入就绪态或是

优先级被改变时，如果处于运行态的任务优先级更低，则该任务总是抢占当前运行的任务。

任务可以在阻塞态等待一个事件，当事件发生时其将自动回到就绪态。时间事件发生在某个特定的时刻，比如阻塞超时。时间事件通常用于周期性或超时行为。任务或中断服务例程向队列发送消息或发送任何一种信号量，都将触发同步事件。同步事件通常用于触发同步行为，比如某个外围的数据到达了。

图 7.11 通过图示某个应用程序的执行流程展现了抢占式调度的行为方式。

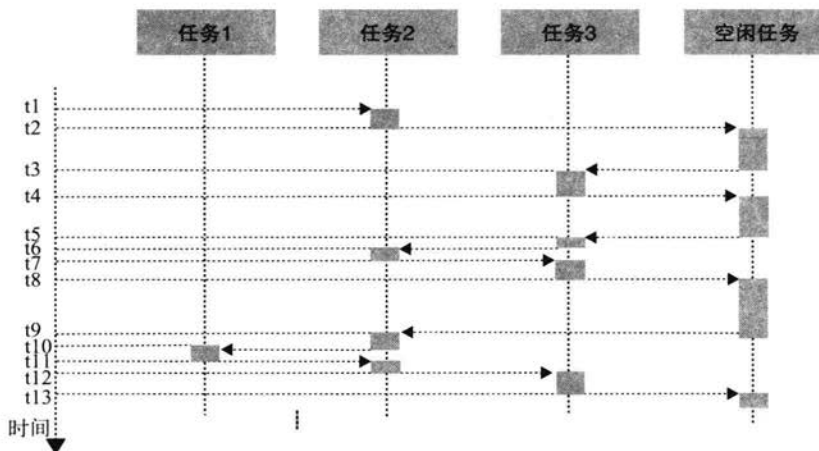


图 7.11 执行流程中的主要抢占点

图 7.11 中各任务如下所示。

□ 空闲任务

空闲任务具有最低优先级，所以每当有更高优先级任务处于就绪态时，空闲任务就会被抢占，如图中 t3、t5 和 t9 时刻。

□ 任务 3

任务 3 是一个事件驱动任务。其工作在一个相对较低的优先级，但优先级高于空闲任务。其大部分时间都在阻塞态等待其关心的事件。每当事件发生时其就从阻塞态转移到就绪态。FreeRTOS 中所有的任务间通信机制（队列、信号量等）都可以通过这种方式用于发送事件以及让任务解除阻塞。

事件在 t3、t5 以及 t9~t12 之间的某个时刻发生。发生在 t3 和 t5 时刻的事件可以立即被处理，因为这些时刻任务 3 在所有可运行任务中优先级最高。发生在 t9~t12 之间某个时刻的事件不会得到立即处理，需要一直等到 t12 时刻。

因为具有更高优先级的任务 1 和任务 2 尚在运行中，只有到了 t12 时刻，这两个任务进入阻塞态，使得任务 3 成为具有最高优先级的就绪态任务。

□ 任务 2

任务 2 是一个周期性任务，其优先级高于任务 3 的并低于任务 1 的。根据周期间隔，任务 2 期望在 t1、t6 和 t9 时刻执行。

在 t6 时刻任务 3 处于运行态，但是任务 2 相对具有更高的优先级，所以会抢占任务 3，并立即得到执行。任务 2 完成处理后，在 t7 时刻返回阻塞态。同时，任务 3 得以重新进入运行态，继续完成处理。任务 3 在 t8 时刻进入阻塞状态。

□ 任务 1

任务 1 也是一个事件驱动任务。任务 1 在所有任务中具有最高优先级，因此可以抢占系统中的任何其他任务。从图中可以看出，任务 1 的事件只是发生在在 t10 时刻，此时任务 1 抢占了任务 2。只有当任务 1 在 t11 时刻再次进入阻塞态之后，任务 2 才得以有机会继续完成处理。

2. 选择任务优先级

从图 7.11 中可以看到优先级分配是如何从根本上影响应用程序行为的。作为一种通用规则，完成硬实时功能的任务优先级会高于完成软件时功能任务的优先级。但其他一些因素，比如执行时间和处理器利用率，都必须纳入考虑范围，以保证应用程序不会超过硬实时的需求限制。

单调速率调度 (Rate Monotonic Scheduling, RMS) 是一种常用的优先级分配技术。其根据任务周期性执行的速率来分配一个唯一的优先级。具有最高周期执行频率的任务赋予最高优先级；具有最低周期执行频率的任务赋予最低优先级。这种优先级分配方式被证明可以最大化整个应用程序的可调度性 (schedulability)，但是运行时间不定以及并非所有任务都具有周期性，会使得对这种方式的全面计算变得相当复杂。

3. 协作式调度

本书主要介绍抢占式调度。FreeRTOS 也可以选择采用协作式调度。若采用一个纯粹的协作式调度器，只可能在运行态任务进入阻塞态或是运行态任务显式调用 taskYIELD() 时，才会进行上下文切换。任务永远不会被抢占，而具有相同优先级的任务也不会自动共享处理器时间。协作式调度的这种工作方式虽然比较简单，但可能会导致系统响应不够迅速。

实现混合调度方案也是可行的，这需要在中断服务例程中显式地进行上下文切换，从而允许同步事件产生抢占行为，但时间事件却不行。这样做的结果是得到了一个没有时间片机制的抢占式系统。或许这正是所期望的，因为获得了效率，并且这也是一种常用的调度器配置。

7.3 队列管理

基于 FreeRTOS 的应用程序是由一系列独立或交互的任务构成的，每个任务都拥有自己相对独立的资源。任务之间通过相互操作系统提供的某种通信机制实现信息的共享或交互。FreeRTOS 中所有的通信与同步机制都是基于队列实现的。本节将着重介绍 FreeRTOS 的“队列”这一通信机制，包括队列的创建、队列数据发送、队列数据接收、队列阻塞以及对系统行为的影响。

7.3.1 概述

1. 数据存储

队列可以保存有限个具有确定长度的数据单元。队列可以保存的最大单元数目称为队列的

“深度”。在队列创建时需要设定其深度和每个单元的大小。

通常情况下，队列被作为 FIFO（先进先出）使用，即数据由队列尾写入，从队列首读出。当然，由队列首写入也是有可能的。

向队列写入数据是通过字节拷贝把数据存储在队列中；从队列读出数据使得把队列中的数据拷贝删除。图 7.12 展现了队列的写入与读出过程，以及读写操作对队列中数据的影响。

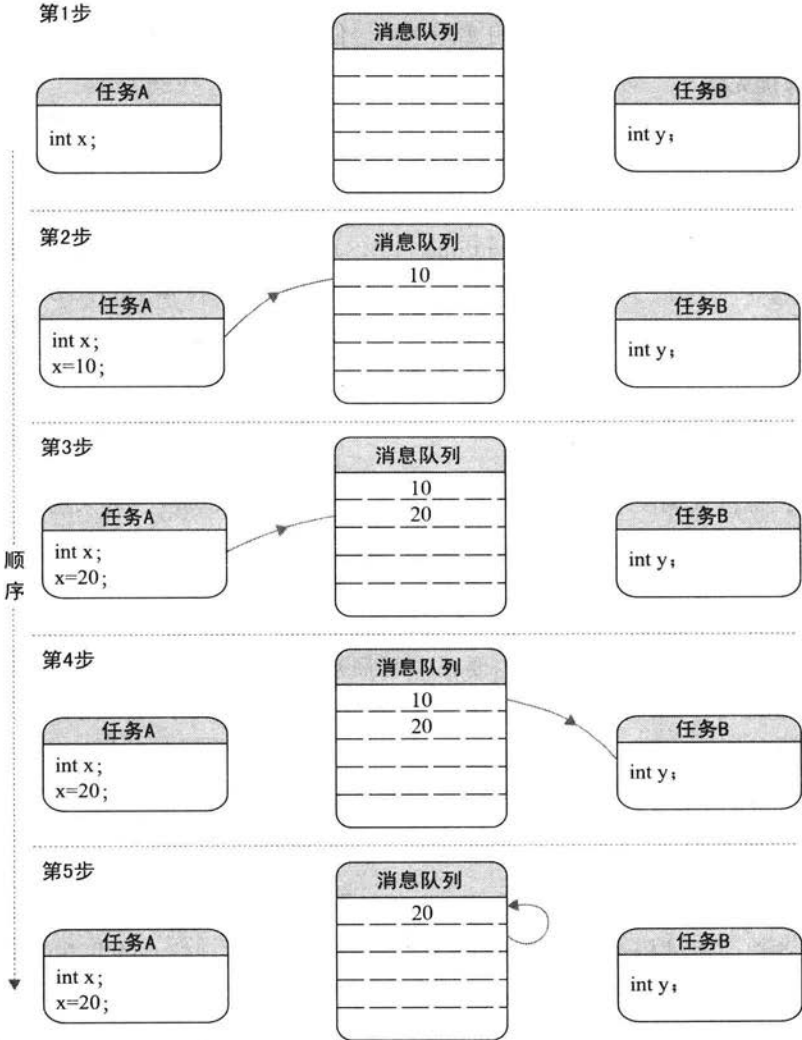


图 7.12 队列写入与读出过程示例

2. 可被多任务存取

队列是具有自己独立权限的内核对象，并不属于或赋予任何任务。所有任务都可以向同一队列写入和从同一列读出。通常一个队列由多方写入，但很少由多方读取。

3. 读队列时阻塞

当某个任务试图读取一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞态以等待队列数据有效。当其他任务或中断服务例程向其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转移为就绪态。

由于队列可以被多个任务读取，所以对单个队列而言，也可能有多个任务处于阻塞态以等待队列数据有效。在这种情况下，一旦队列数据有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待时间最长的任务。

4. 写队列时阻塞

同读队列一样，任务也可以在写队列时指定一个阻塞超时时间。这个时间是指当被写队列已满时，任务进入阻塞态以等待队列空间有效的最长时间。

由于队列可以被多个任务写入，所以对单个队列而言，也可能有多个任务处于阻塞态以等待队列空间有效。在这种情况下，一旦队列空间有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待时间最长的任务。

7.3.2 使用队列

xQueueCreate() API 函数

队列在使用前必须先被创建。队列由声明为 xQueueHandle 的变量进行引用。xQueueCreate() 用于创建一个队列，并返回一个 xQueueHandle 句柄以便于对其创建的队列进行引用。

当创建队列时，FreeRTOS 从堆空间中分配内存空间。分配的空间用于存储队列数据结构本身以及队列中包含的数据单元。如果内存堆中没有足够的空间来创建队列，xQueueCreate() 将返回 NULL。函数原型见程序清单 7.29，各参数及描述见表 7.7。

程序清单 7.29 xQueueCreate() API 函数原型

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize );
```

表 7.7 xQueueCreate() 参数与返回值

参 数	描 述
uxQueueLength	队列能够存储的最大单元数目，即队列深度
uxItemSize	队列中数据单元的长度，以字节为单位
返回值	<input type="checkbox"/> NULL 表示没有足够的堆空间分配给队列而导致创建失败 <input type="checkbox"/> 非 NULL 值表示队列创建成功。此返回值应当保存下来，以作为操作此队列的句柄

xQueueSendToBack() 与 xQueueSendToFront() API 函数

如同函数名字面意思所表示的一样，xQueueSendToBack() 用于将数据发送到队列尾；而 xQueueSendToFront() 用于将数据发送到队列首。函数原型见程序清单 7.30 和程序清单 7.31，各参数及描述见表 7.8。

xQueueSend() 完全等同于 xQueueSendToBack()。

但切记不要在中断服务例程中调用 xQueueSendToFront() 或 xQueueSendToBack()。系统提供中断安全版本的 xQueueSendToFrontFromISR() 与 xQueueSendToBackFromISR() 用于在中断服务中实现相同的功能。

程序清单 7.30 The xQueueSendToFront() API 函数原型

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait );
```

程序清单 7.31 The xQueueSendToBack() API 函数原型

```
portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,
                                  const void * pvItemToQueue,
                                  portTickType xTicksToWait );
```

表 7.8 xQueueSendToFront() 与 xQueueSendToBack() 函数参数及返回值

参 数	描 述
xQueue	目标队列的句柄。这个句柄即调用 xQueueCreate() 创建该队列时的返回值
pvItemToQueue	<ul style="list-style-type: none"> ❑ 发送数据的指针。其指向将要拷贝到目标队列中的数据单元 ❑ 因为在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间拷贝对应长度的数据到队列的存储区域
xTicksToWait	<ul style="list-style-type: none"> ❑ 阻塞超时时间。如果在发送时队列已满，这个时间即任务处于阻塞态等待队列空间有效的最长等待时间 ❑ 如果 xTicksToWait 设为 0，并且队列已满，则 xQueueSendToFront() 与 xQueueSendToBack() 均会立即返回 ❑ 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位 ❑ 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConfig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制
返回值	<p>有两个可能的返回值：</p> <ul style="list-style-type: none"> ❑ pdPASS。返回 pdPASS 只会有一种情况，那就是数据被成功发送到队列中 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列空间有效——在超时到来前能够将数据成功写入队列，则函数会返回 pdPASS ❑ errQUEUE_FULL。如果由于队列已满而无法将数据写入，则将返回 errQUEUE_FULL 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列空间有效。但直到超时也没有其他任务或中断服务例程读取队列而腾出空间，则函数会返回 errQUEUE_FULL

xQueueReceive() 与 xQueuePeek() API 函数

xQueueReceive() 用于从队列中接收（读取）数据单元。接收到的单元同时会从队列中删

除。xQueuePeek() 也是从队列中接收数据单元，不同的是，并不从队列中删出接收到的单元。xQueuePeek() 从队列首接收到数据后，不会修改队列中的数据，也不会改变数据在队列中的存储顺序。函数原型见程序清单 7.32 和程序清单 7.33，各参数及描述见表 7.9。

切记不要在中断服务例程中调用 xQueueReceive() 和 xQueuePeek()。中断安全版本的替代 API 函数 xQueueReceiveFromISR() 将会在后续章节中讲述。

程序清单 7.32 xQueueReceive() API 函数原型

```
portBASE_TYPE xQueueReceive( xQueueHandle xQueue,
                             const void * pvBuffer,
                             portTickType xTicksToWait );
```

程序清单 7.33 xQueuePeek() API 函数原型

```
portBASE_TYPE xQueuePeek( xQueueHandle xQueue,
                          const void * pvBuffer,
                          portTickType xTicksToWait );
```

表 7.9 xQueueReceive() 与 xQueuePeek() 函数参数与返回值

参 数	描 述
xQueue	被读队列的句柄。这个句柄即调用 xQueueCreate() 创建该队列时的返回值
pvBuffer	<ul style="list-style-type: none"> ❑ 接收缓存的指针。其指向一段内存区域，用于接收从队列中拷贝来的数据 ❑ 数据单元的长度在创建队列时就已经设定，所以该指针指向的内存区域大小应当足够保存一个数据单元
xTicksToWait	<ul style="list-style-type: none"> ❑ 阻塞超时时间。如果在接收时队列为空，则这个时间是任务处于阻塞状态以等待队列数据有效的最长等待时间 ❑ 如果 xTicksToWait 设为 0，并且队列为空，则 xQueueReceive() 与 xQueuePeek() 均会立即返回 ❑ 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位 ❑ 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConfig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制
返回值	<p>有两个可能的返回值：</p> <ul style="list-style-type: none"> ❑ pdPASS。只有一种情况会返回 pdPASS，那就是成功地从队列中读到数据 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效——在超时到来前能够从队列中成功读取数据，则函数会返回 pdPASS ❑ errQUEUE_FULL。如果在读取时由于队列已空而没有读到任何数据，则将返回 errQUEUE_FULL 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效。但直到超时也没有其他任务或中断服务例程向队列中写入数据，则函数会返回 errQUEUE_FULL

uxQueueMessagesWaiting() API 函数

uxQueueMessagesWaiting() 用于查询队列中当前有效数据单元个数。函数原型见程序清单 7.34，各参数及描述见表 7.10。

切记不要在中断服务例程中调用 uxQueueMessagesWaiting()。应当在中断服务中使用其中断

安全版本 uxQueueMessagesWaitingFromISR()。

程序清单 7.34 uxQueueMessagesWaiting() API 函数原型

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle
                                                xQueue );
```

表 7.10 uxQueueMessagesWaiting() 函数参数及返回值

参 数	描 述
xQueue	被查询队列的句柄。这个句柄即调用 xQueueCreate() 创建该队列时的返回值
返回值	当前队列中保存的数据单元个数。返回 0 表明队列为空

例 10 读队列时阻塞

本例示范创建一个队列，由多个任务向队列写数据，以及从队列读出数据。这个创建队列用于保存 long 型数据单元。向队列中写数据的任务没有设定阻塞超时时间，而读取队列的任务设定了超时时间。

向队列中写数据的任务的优先级低于读取队列任务的优先级。这意味着队列中永远不会保持超过一个的数据单元。因为一旦有数据被写入队列，读队列任务立即解除阻塞，抢占写队列任务，并从队列中接收数据，同时从队列中删除数据——队列再一次变为空队列。

程序清单 7.35 展现了写队列任务的代码实现。这个任务创建了两个实例，一个不停地向队列中写数值 100，而另一个实例不停地向队列中写数值 200。任务的入口参数用来为每个实例传递各自的写入值。

程序清单 7.35 例 10 中的写队列任务实现代码

```
static void vSenderTask( void *pvParameters )
{
    long lValueToSend;
    portBASE_TYPE xStatus;

    /* 该任务会创建两个实例，所以写入队列的值通过任务入口参数传递——这种方式使得每个实例使用不同的值。
    队列创建时指定其数据单元为 long 型，所以把入口参数强制转换为数据单元要求的类型 */
    lValueToSend = ( long ) pvParameters;

    /* 和大多数任务一样，本任务也处于一个死循环中 */
    for( ;; )
    {
        /* 向队列发送数据，第一个参数是要写入的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。
        第二个参数是被发送数据的地址，本例中即变量 lValueToSend 的地址。第三个参数是阻塞超时时间——当队列满时，
        任务转入阻塞状态以等待队列空间有效。本例中没有设定超时时间，因为此队列绝不会保持有超过一个数据单元的机会，
        所以也绝不会满 */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS )
        {
            /* 发送操作由于队列满而无法完成——这必然存在错误，因为本例中的队列不可能满 */
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

```

/* 允许执行其他发送任务。 taskYIELD() 通知调度器现在就切换到其他任务，而不必等到本任务的时间片耗尽 */
taskYIELD();
}
}

```

程序清单 7.36 展现了读队列任务的代码实现。读队列任务设定了 100ms 的阻塞超时时间，所以会进入阻塞态以等待队列数据有效。一旦队列中数据单元有效，或者即使队列数据无效但等待时间超过 100ms，此任务将会解除阻塞。在本例中，将永远不会出现 100ms 超时，因为有两个任务在不停地向队列中写数据。

程序清单 7.36 例 10 中的读队列任务实现代码

```

static void vReceiverTask( void *pvParameters )
{
    /* 声明变量，用于保存从队列中接收的数据 */
    long lReceivedValue;
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* 本任务依然处于死循环中 */
    for( ;; )
    {
        /* 此调用会发现队列一直为空，因为本任务将立即删除刚写入队列的数据单元 */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }
        /* 从队列中接收数据，第一个参数是被读取的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。第二个参数是保存接收的数据的缓冲区地址，本例中即变量 lReceivedValue 的地址。此变量类型与队列数据单元类型相同，所以有足够的大小来存储接收的数据。第三个参数是阻塞超时时间——当队列空时，任务转入阻塞状态以等待队列数据有效。本例中常量 portTICK_RATE_MS 用来将 100ms 绝对时间转换为以系统心跳为单位的时间值 */
        xStatus = xQueueReceive( xQueue,
                                &lReceivedValue,
                                xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* 成功读出数据，打印出来 */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* 等待 100ms 也没有收到任何数据。必然存在错误，因为发送任务在不停地向队列中写入数据 */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}

```

程序清单 7.37 包含了 main() 函数的实现。其在启动调度器之前创建了一个队列和三个任务。

尽管对任务的优先级的设计使得队列实际上在任何时候都不可能多于一个数据单元，但是本例代码还是创建了一个可以保存最多 5 个 long 型值的队列。

程序清单 7.37 例 10 中的 main() 函数实现代码

```

/* 声明一个类型为 xQueueHandle 的变量，其用于保存队列句柄，以便三个任务都可以引用此队列 */
xQueueHandle xQueue;
int main( void )
{
    /* 创建的队列用于保存最多 5 个值，每个数据单元都有足够的空间来存储一个 long 型变量 */
    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        /* 创建两个写队列任务实例，任务入口参数用于传递发送到队列的值。所以一个实例不停地向队列发送
        100，而另一个任务实例不停地向队列发送 200。两个任务的优先级都设为 1*/
        xTaskCreate( vSenderTask,
                    "Sender1",
                    1000,
                    (void *)
                    100,
                    1,
                    NULL );
        xTaskCreate( vSenderTask,
                    "Sender2",
                    1000,
                    (void *)
                    200,
                    1,
                    NULL );
        /* 创建一个读队列任务实例。其优先级设为 2，高于写任务优先级 */
        xTaskCreate( vReceiverTask,
                    "Receiver",
                    1000,
                    NULL,
                    2,
                    NULL );
        /* 启动调度器，任务开始执行 */
        vTaskStartScheduler();
    }
    else
    {
        /* 队列创建失败 */
    }
    /* 如果一切正常，main() 函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致无法
    创建空闲任务 */
    for( ;; );
}

```

写队列任务在每次循环中都调用 taskYIELD()。taskYIELD() 通知调度器立即进行任务切换，而不必等到当前任务的时间片耗尽。某个任务调用 taskYIELD() 等效于其自愿放弃运行态。由于本例中两个写队列任务具有相同的任务优先级，所以一旦其中一个任务调用了 taskYIELD()，另

一个任务将会得到执行——调用 `taskYIELD()` 的任务转移到就绪态，同时另一个任务进入运行态。这样就可以使得这两个任务轮流地向队列发送数据。

图 7.13 展示了例 10 的执行流程。

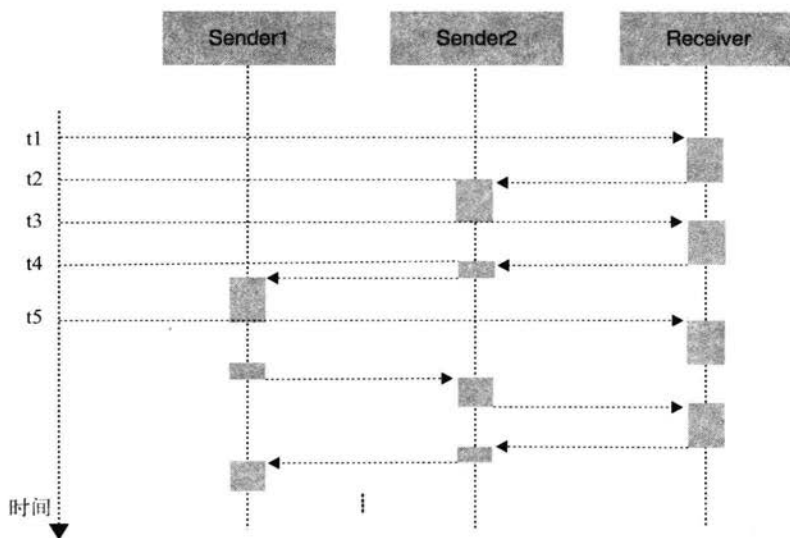


图 7.13 例 10 中代码的执行流程

使用队列传递复合数据类型

通常一个任务可以从单个队列中接收来自多个发送源的数据。接收方收到数据后，需要知道数据的来源，并根据数据的来源决定下一步如何处理。一个简单的方式就是利用队列传递结构体，结构体成员中包含了数据信息和来源信息。图 7.14 对这一方案进行了展现。

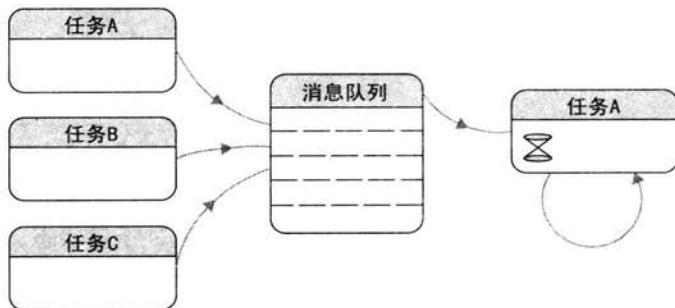


图 7.14 队列用于传递结构体的一种情形

从图 7.14 中可以看出：

- ❑ 创建一个队列用于保存类型为 `xData` 的结构体数据单元。结构体成员包括了一个数据值 and 表示数据含义的编码，两者合为一个消息可以一次性发送到队列。

- ❑ 中央控制任务用于完成主要的系统功能。其必须对队列中传来的输入和其他系统状态的改变做出响应。
- ❑ CAN 总线任务用于封装 CAN 总线的接口功能。当 CAN 总线任务收到并解码一个消息后，其将把解码后的消息放到 xData 结构体中并发往控制任务。
- ❑ 人机接口（HMI）任务用于对所有的人机接口功能进行封装。设备操作员可能通过各种方式进行命令输入和参数查询，人机接口任务需要对这些操作进行检测并解析。当接收到一个新的命令后，人机接口任务通过 xData 结构将命令发送到中央控制任务。结构体的 iMeaning 成员用于让中央控制任务知道这个数据是用来干什么的——从图中的描述可以看出，这个数据表示一个新的参数设置。结构体的 iValue 成员可以让中央控制任务知道具体的设置值。

例 11 写队列时阻塞 / 向队列发送结构体

例 11 与例 10 类似，只是写队列任务与读队列任务的优先级交换了，即读队列任务的优先级低于写队列任务的优先级。并且本例中的队列用于在任务间传递结构体数据，而非简单的长整型数据。

程序清单 7.38 展示了例 11 中要用到的结构体定义。

程序清单 7.38 定义队列传递的数据结构，并声明此类型的两个变量在本例中使用

```

/* 定义队列传递的结构类型 */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;

/* 声明两个 xData 类型的变量，通过队列进行传递 */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Used by Sender1. */
    { 200, mainSENDER_2 } /* Used by Sender2. */
};

```

在例 10 中读队列任务具有最高优先级，所以队列不会拥有一个以上的数据单元。这是因为一旦数据被写队列任务写进队列，读队列任务立即抢占写队列任务，读取刚写入的数据单元。在例 11 中，写队列任务具有最高优先级，所以队列正常情况下一直处于满状态。这是因为一旦读队列任务从队列中读取一个数据单元，某个写队列任务就会立即抢占读队列任务，把刚刚读取的位置重新写入，之后便又转入阻塞态以等待队列空间有效。

程序清单 7.39 是写队列任务的实现代码。写队列任务指定了 100 ms 的阻塞超时时间，以便在队列满时转入阻塞态以等待队列空间有效。进入阻塞态后，一旦队列空间有效，或是等待超过了 100ms 队列空间尚无效，其将解除阻塞。在本例中，将永远不会出现 100 ms 超时的情况，因为读队列任务在不停地从队列中读出数据从而腾出队列数据空间。

程序清单 7.39 例 11 中写队列任务的实现代码

```

static void vSenderTask( void *pvParameters )
{
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* 与大多数任务相同，该任务为一个死循环 */
    for( ;; )
    {
        /* Send to the queue. 第二个参数是要发送的数据结构地址。这个地址是从任务入口参数中传入，
        所以直接使用 pvParameters。第三个参数是阻塞超时时间——当队列满时，任务转入阻塞态等待队列空间
        有效的最长时间。指定超时时间是因为写队列任务的优先级高于读任务的优先级。所以队列如预期一样很快
        写满，写队列任务就会转入阻塞态，此时读队列任务才会得以执行，从队列中把数据读走 */
        xStatus = xQueueSendToBack( xQueue,
                                    pvParameters,
                                    xTicksToWait );

        if( xStatus != pdPASS )
        {
            /* 写队列任务无法将数据写入队列，直至 100ms 超时。这必然存在错误，因为只要写队列任务进入
            阻塞态，读队列任务就会得到执行，从而读走数据，腾出空间 */
            vPrintString( "Could not send to the queue.\r\n" );
        }
        /* 让其他写队列任务得到执行 */
        taskYIELD();
    }
}

```

读队列任务的优先级最低，所以只有在所有写队列任务都进入阻塞态后才有机会得到执行。而写队列任务只会在队列满时才会进入阻塞态，所以读队列任务得到执行时队列已满。因此读队列任务只管不停地读取数据，不必设定超时时间。

读队列任务的实现代码见程序列表 7.40。

程序清单 7.40 例 11 中读队列任务的实现代码

```

static void vReceiverTask( void *pvParameters )
{
    /* 声明结构体变量以保存从队列中读出的数据单元 */
    xData xReceivedStructure;
    portBASE_TYPE xStatus;

    /* 同样任务以死循环实现 */
    for( ;; )
    {
        /* 读队列任务的优先级最低，所以其只可能在写队列任务阻塞时得到执行。而写队列任务只会在队列写满
        时才会进入阻塞态，所以读队列任务执行时队列肯定已满。所以队列中数据单元的个数应当等于队列的深
        度——本例中队列深度为 3 */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }
    }
}

```

```

        &( xStructsToSend[1]),
        2,
        NULL );
/* 创建读队列任务。读队列任务优先级设为 1，低于写队列任务的优先级 */
xTaskCreate( vReceiverTask,
            "Receiver",
            1000,
            NULL,
            1,
            NULL );
/* 启动调度器，创建的任务得到执行 */
vTaskStartScheduler();
}
else
{
    /* 创建队列失败 */
}
/* 如果一切正常，main() 函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致空闲
任务无法创建 */
for( ;; );
}

```

和例 10 类似，写队列任务在每次循环中都主动进行任务切换，所以两个数据会被轮翻地写入队列。图 7.15 展示了例 11 的执行流程。表 7.11 对图 7.15 中的各个要点进行解释。

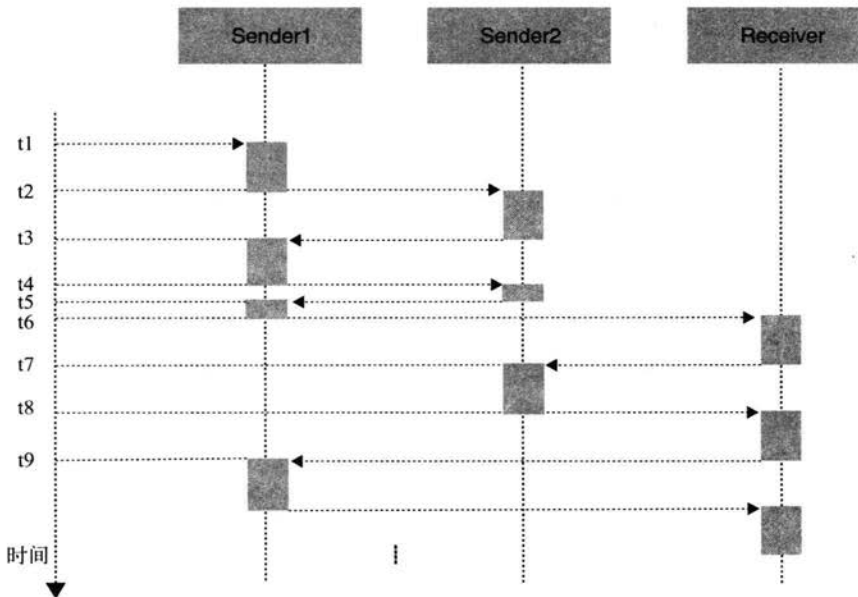


图 7.15 例 11 的执行流程

表 7.11 图 7.15 的要点解释

时 刻	描 述
t1	写队列任务 1 得到执行, 并向队列中发送数据
t2	写队列任务 1 切换到写队列任务 2。写队列任务 2 向队列中发送数据
t3	写队列任务 2 又切回写队列任务 1。写队列任务 1 再次将数据写入队列, 导致队列满
t4	写队列任务 1 切换到写队列任务 2
t5	写队列任务 2 试图向队列中写入数据。但由于队列已满, 所以写队列任务 2 转入阻塞态以等待队列空间有效。这使得写队列任务 1 再次得到执行
t6	写队列任务 1 试图向队列中写入数据。但由于队列已满, 所以写队列任务 1 也转入阻塞态以等待队列空间有效。此时写队列任务均处于阻塞态, 这才使得被赋予最低优先级的读队列任务得以执行
t7	读队列任务从队列读取数据, 并把读出的数据单元从队列中移出。一旦队列空间有效, 写队列任务 2 立即解除阻塞, 并且因为其具有更高优先级, 所以抢占读队列任务。写队列任务 2 又向队列中写入数据, 填充到刚刚被读队列任务腾出的存储空间, 使得队列再一次变满。写队列发送完数据后便调用 taskYIELD(), 但写队列任务 1 尚处于阻塞态, 所以写队列任务 2 并未被切换出去, 继续执行
t8	写队列任务 2 试图向队列中写入数据。但队列已满, 所以写队列任务 2 转入阻塞态。两个写队列任务再一次同时处于阻塞态, 所以读队列任务得以执行
t9	读队列任务从队列读取数据, 并把读出的数据单元从队列中移出。一旦队列空间有效, 写队列任务 1 立即解除阻塞, 并且因为其具有更高优先级, 所以抢占读队列任务。写队列任务 1 又向队列中写入数据, 填充到刚刚被读队列任务腾出的存储空间, 使得队列再一次变满。写队列发送完数据后便调用 taskYIELD(), 但写队列任务 2 尚处于阻塞态, 所以写队列任务 1 并未被切换出去, 继续执行。写队列任务 1 试图向队列中写入数据。但队列已满, 所以写队列任务 1 转入阻塞态 两个写队列任务再一次同时处于阻塞态, 所以读队列任务得以执行

7.3.3 大型数据单元传输

如果队列存储的数据单元尺寸较大, 最好利用队列来传递数据的指针而不是对数据本身在队列上一字节一字节地拷贝进或拷贝出。传递指针无论是在处理速度上还是内存空间利用上都更有效。但是, 当你利用队列传递指针时, 一定要十分小心地注意以下两点:

❑ 指针指向的内存空间的所有权必须明确。

当任务间通过指针共享内存时, 应该从根本上保证不会有任意两个任务同时修改共享内存中的数据, 或以其他行为方式使得共享内存数据无效或产生一致性问题。原则上, 共享内存存在其指针发送到队列之前, 其内容只允许被发送任务访问; 共享内存指针从队列中被读出之后, 其内容亦只允许被接收任务访问。

❑ 指针指向的内存空间必须有效

如果指针指向的内存空间是动态分配的, 只应该有一个任务负责对其进行内存释放。当这段内存空间被释放之后, 就不应该有任何一个任务再访问这段空间。

切忌用指针访问任务栈上分配的空间。因为当栈帧发生改变后，栈上的数据将不再有效。

7.4 中断管理

嵌入式实时操作系统本质上是对系统资源和事件的集约化管理，系统内核需要响应整个系统环境产生的事件，并在决策的同时调配响应的资源。系统通常需要处理来自不同外设产生的事件，这些事件对处理时间和响应时间都有不同的要求。因此，如何满足不同事件对响应时间和处理时间的要求，以达到最佳事件处理的实现策略，是嵌入式软件开发的重点。本节围绕“事件响应”这一基本概念讲述 FreeRTOS 的中断管理。

7.4.1 延迟中断处理

采用二值信号量同步

二值信号量可以在某个特殊的中断发生时，让任务解除阻塞，相当于让任务与中断同步。这样就可以让中断事件处理量大的工作在同步任务中完成，中断服务例程（ISR）中只是快速处理少部分工作。如此，中断处理可以说是被“推迟”（deferred）到一个“处理”（handler）任务。

如果某个中断处理要求特别紧急，其延迟处理任务的优先级可以设为最高，以保证延迟处理任务随时都可以抢占系统中的其他任务。这样，延迟处理任务就成为其对应的 ISR 退出后第一个执行的任务，在时间上紧接着 ISR 执行，相当于所有的处理都在 ISR 中完成一样。这种方案如图 7.16 所展现的。

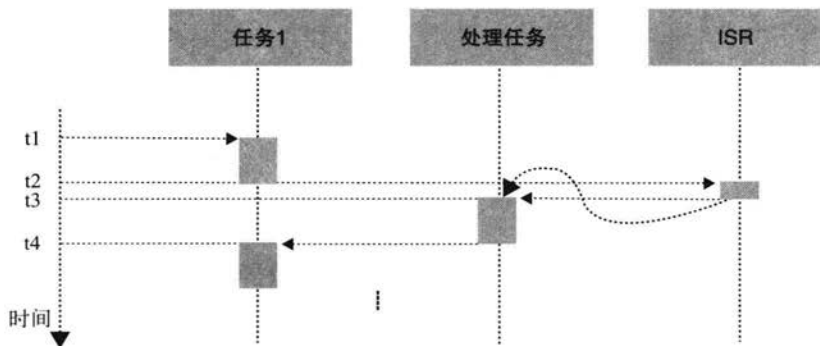


图 7.16 中断打断某个任务，但返回到另一个任务

延迟处理任务对一个信号量进行带阻塞性质的“take”调用，意思是进入阻塞态以等待事件发生。当事件发生后，ISR 对同一个信号量进行“give”操作，使得延迟处理任务解除阻塞，从而事件在延迟处理任务中得到相应的处理。

“获取”（Taking，带走，按通常的说法译为获取）和“给出”（Giving）信号量从概念上讲，在不同的应用场合有不同的含义。在经典的信号量术语中，获取信号量等同于一个 P() 操作，而给出信号量等同于一个 V() 操作。

在这种中断同步的情形下，信号量可以看做一个深度为 1 的队列。这个队列由于最多只能保存一个数据单元，所以其不为空则为满（所谓“二值”）。延迟处理任务调用 `xSemaphoreTake()` 时，等效于带阻塞时间地读取队列，如果队列为空的话任务则进入阻塞态。当事件发生后，ISR 简单地通过调用 `xSemaphoreGiveFromISR()` 放置一个令牌（信号量）到队列中，使得队列成为满状态。这也使得延迟处理任务切出阻塞态，并移除令牌，使得队列再次成为空。当任务完成处理后，再次读取队列，发现队列为空，又进入阻塞态，等待下一次事件发生。整个流程如图 7.17 所示。

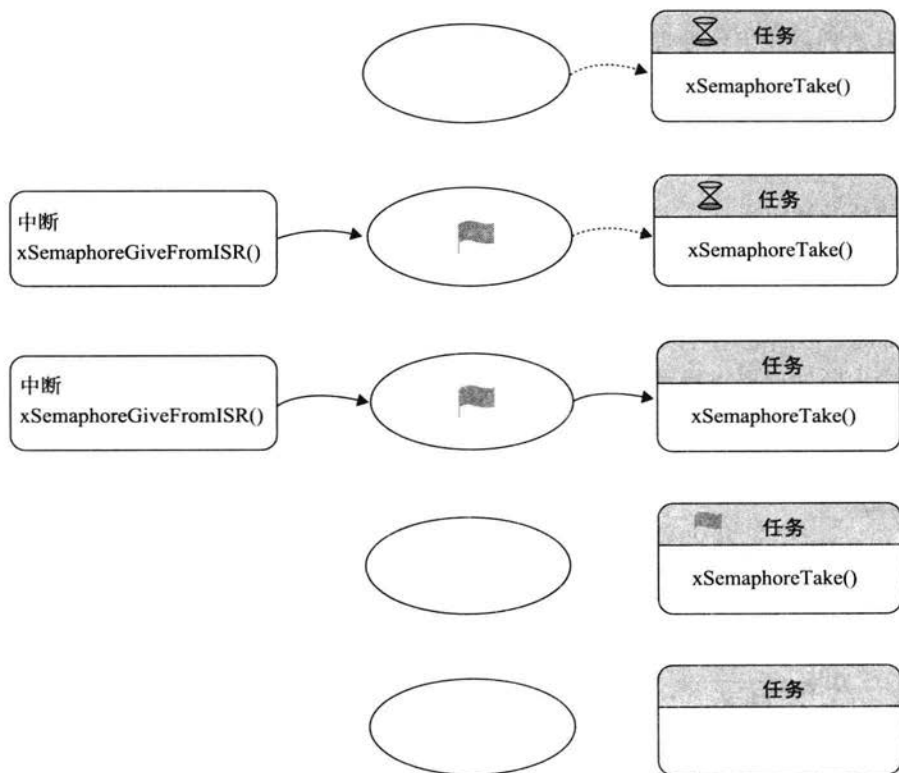


图 7.17 使用一个二值信号量实现任务与中断同步

如图 7.17 所示，中断给出信号量，甚至是在信号量第一次被获取之前就给出；而任务在获取信号量之后再也不给回来。这就是为什么说这种情况与读写队列相似。这也经常会给大家造成迷惑，因为这种情形和其他信号量的使用场合大不相同。在其他场合下，任务获取（Take）信号量之后，必须得给回来（Give）。

vSemaphoreCreateBinary() API 函数

FreeRTOS 中各种信号量的句柄都存储在 `xSemaphoreHandle` 类型的变量中。

在使用信号量之前，必须先创建它。创建二值信号量可以使用 `vSemaphoreCreateBinary()` API 函数。`vSemaphoreCreateBinary()` API 的函数原型见程序清单 7.42，各参数及描述见表 7.12。

程序清单 7.42 vSemaphoreCreateBinary() API 函数原型

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

表 7.12 vSemaphoreCreateBinary() 参数

参 数	描 述
xSemaphore	创建的信号量 需要说明的是，vSemaphoreCreateBinary() 在实现上是一个宏，所以信号量变量应当直接传入，而不是传递地址。本章中包含本函数调用的示例可用于参考

xSemaphoreTake() API 函数

“带走”(Taking) 一个信号量意为“获取”(Obtain) 或“接收”(Receive) 信号量。只有当信号量有效的时候才可以被获取。在经典的信号量术语中，xSemaphoreTake() 等同于一次 P() 操作。函数原型见程序清单 7.43，各参数及描述见表 7.13。

除互斥信号量 (Recursive Semaphore，直译为递归信号量，按通常的说法译为互斥信号量) 外，所有类型的信号量都可以通过调用函数 xSemaphoreTake() 来获取。

但 xSemaphoreTake() 不能在中断服务例程中调用。

程序清单 7.43 xSemaphoreTake() API 函数原型

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore,  
                               portTickType xTicksToWait );
```

表 7.13 xSemaphoreTake() 参数及返回值

参 数	描 述
xSemaphore	获取的信号量 信号量由定义为 xSemaphoreHandle 类型的变量引用。信号量在使用前必须先创建
xTicksToWait	<ul style="list-style-type: none"> ❑ 阻塞超时时间。任务进入阻塞态以等待信号量有效的最长时间 ❑ 如果 xTicksToWait 为 0，则 xSemaphoreTake() 在信号量无效时会立即返回 ❑ 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位 ❑ 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制
返回值	<p>有两个可能的返回值：</p> <ul style="list-style-type: none"> ❑ pdPASS。只有一种情况会返回 pdPASS，那就是成功获得信号量 如果设定了阻塞超时时间 (xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待信号量有效。如果在超时到来前信号量变为有效，亦可被成功获取，则函数返回 pdPASS ❑ pdFALSE。未能获得信号量 如果设定了阻塞超时时间 (xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待信号量有效。但直到超时信号量也没有变为有效，所以不会获得信号量，则函数返回 pdFALSE

xSemaphoreGiveFromISR() API 函数

除互斥信号量外，FreeRTOS 支持的其他类型的信号量都可以通过调用 xSemaphore GiveFromISR() 给出。

xSemaphoreGiveFromISR() 是 xSemaphoreGive() 的特殊形式，专门用于中断服务例程中。函数原型见程序清单 7.44，各参数及描述见表 7.14。

程序清单 7.44 xSemaphoreGiveFromISR() API 函数原型

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle Semaphore,  
                                       portBASE_TYPE *pxHigherPriorityTaskWoken );
```

表 7.14 xSemaphoreGiveFromISR() 参数与返回值

参 数	描 述
xSemaphore	给出的信号量 信号量由定义为 xSemaphoreHandle 类型的变量引用。信号量在使用前必须先创建
pxHigherPriorityTaskWoken	对某个信号量而言，可能有不止一个任务处于阻塞态在等待其有效。调用 xSemaphoreGiveFromISR() 会让信号量变为有效，所以会让其中一个等待任务切出阻塞态。如果调用 xSemaphoreGiveFromISR() 使得一个任务解除阻塞，并且这个任务的优先级高于当前任务（也就是被中断的任务），那么 xSemaphoreGiveFromISR() 会在函数内部将 *pxHigherPriorityTaskWoken 设为 pdTRUE 如果 xSemaphoreGiveFromISR() 将此值设为 pdTRUE，则在中断退出前应当进行一次上下文切换。这样才能保证中断直接返回到就绪态任务中优先级最高的任务
返回值	有两个可能的返回值： <input type="checkbox"/> pdPASS。xSemaphoreGiveFromISR() 调用成功 <input type="checkbox"/> pdFAIL。如果信号量已经有效，无法给出，则返回 pdFAIL

例 12 利用二值信号量对任务和中断进行同步

本例在中断服务例程中使用一个二值信号量让任务从阻塞态中切出——从效果上等同于让任务与中断进行同步。

一个简单的周期性任务用于每隔 500 ms 产生一个软件中断。之所以采用软件中断，是因为在模拟的 DOS 环境中，很难挂接一个真正的 IRQ 中断。相比之下，使用软件中断要方便得多。程序清单 7.45 即这个周期任务的实现代码。需要说明的是，此任务在产生中断之前和之后都会打印输出一个字符串。这样就可以在最终的执行结果中直观地看出整个程序的执行流程。

程序清单 7.45 例 12 中用于周期性产生软件中断的周期任务实现代码

```
static void vPeriodicTask( void *pvParameters )  
{  
    for( ;; )  
    {  
        /* 此任务通过每 500ms 产生一个软件中断来“模拟”中断事件 */  
        vTaskDelay( 500 / portTICK_RATE_MS );
```

```

/* 产生中断，并在产生之前和之后输出信息，以便在执行结果中直观地看出执行流程 */
vPrintString( "Periodic task - About to generate an
              interrupt.\r\n" );

__asm{ int 0x82 } /* 这条语句产生中断 */

vPrintString( "Periodic task - Interrupt
              generated.\r\n\r\n\r\n" );
}
}

```

程序清单 7.46 展现的是延迟处理任务的具体实现——此任务通过使用二值信号量与软件中断进行同步。这个任务也在每次循环中打印输出一个信息，这样做的目的同样是在程序的执行输出结果中直观地看出任务与中断的执行流程。

程序清单 7.46 例 12 中延迟处理任务的实现代码（此任务与中断同步）

```

static void vHandlerTask( void *pvParameters )
{
    /* 与大多数任务相同，任务实现为一个无限循环 */
    for( ;; )
    {
        /* 使用信号量等待一个事件。信号量在调度器启动之前，也即此任务执行之前就已被创建。任务被无超时阻塞，所以此函数调用也只会成功获取信号量之后返回。此处也没有必要检测返回值 */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* 程序运行到这里时，事件必然已经发生。本例的事件处理只是简单地打印输出一个信息 */
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}

```

程序清单 7.47 展现的是中断服务例程，这才是真正的中断处理程序。这段代码做的事情非常少，仅仅是给出一个信号量，以让延迟处理任务解除阻塞。注意这里是如何使用参数 `pxHigherPriorityTaskWoken` 的。这个参数在调用 `xSemaphoreGiveFromISR()` 前被设置为 `pdFALSE`，如果在调用完成后被置为 `pdTRUE`，则需要进行一次上下文切换。

本例中断服务例程的语法，以及用于上下文切换调用的宏，都是基于 EWARM 平台的移植，与其他平台的移植可能会有所不同。对于实际使用的平台，请参考对应移植的 Demo 应用示例，以找到正确的语法要求，此外字符串打印函数并没有采用 C 语言标准库中的函数，而是采用了独立编写的 `vPrintString` 函数实现字符串的打印输出。

程序清单 7.47 例 12 中软件中断的中断服务例程

```

static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    /* 发送信号量以激活任务 */
}

```

```

xSemaphoreGiveFromISR( xBinarySemaphore,
                        xHigherPriorityTaskWoken );
if( xHigherPriorityTaskWoken == pdTRUE )
{
    /* 给出信号量以使得等待此信号量的任务解除阻塞。如果解除阻塞的任务的优先级高于当前任务的优先级——强制进行一次任务切换，以确保中断直接返回到解除阻塞的任务（优先级更高）。说明：在实际使用中，ISR 中强制上下文切换的宏依赖于具体移植。对于实际使用的平台，请参考对应移植自带的示例程序，以决定正确的语法和符号 */
    portSWITCH_CONTEXT();
}
}
}

```

main() 函数很简单，创建二值信号量及任务，安装中断服务例程，然后启动调度器。具体实现代码参见程序清单 7.48。

程序清单 7.48 例 12 中的 main() 函数实现代码

```

int main( void )
{
    /* 信号量在使用前都必须先创建。本例中创建了一个二值信号量 */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* 安装中断服务例程 */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* 检查信号量是否成功创建 */
    if( xBinarySemaphore != NULL )
    {
        /* 创建延迟处理任务。此任务将与中断同步。延迟处理任务在创建时使用了一个较高的优先级，以保证中断退出后会被立即执行。在本例中，为延迟处理任务赋予优先级 3 */
        xTaskCreate( vHandlerTask,
                    "Handler",
                    1000,
                    NULL,
                    3,
                    NULL );

        /* 创建一个任务用于周期性产生软件中断。此任务的优先级低于延迟处理任务。每当延迟处理任务切出阻塞态，就会抢占周期任务 */
        xTaskCreate( vPeriodicTask,
                    "Periodic",
                    1000,
                    NULL,
                    1,
                    NULL );

        /* 启动调度器，创建的任务开始执行 */
        vTaskStartScheduler();
    }

    /* 如果一切正常，main() 函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到这里，很可能是由于系统内存不足而无法创建空闲任务 */
    for( ;; );
}

```

图 7.18 对执行流程做出了解释。

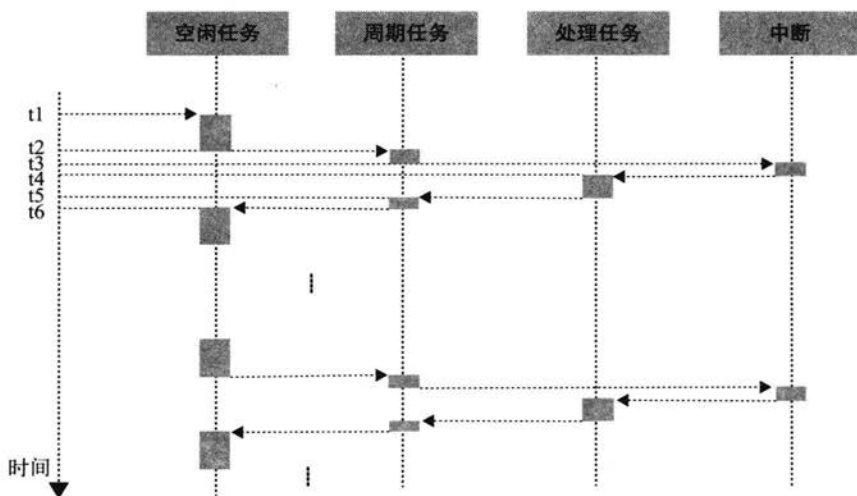


图 7.18 例 12 中代码的执行流程

7.4.2 计数信号量

例 12 演示了一个二值信号量用于让任务和中断进行同步。整个执行流程可以描述为：

- 1) 中断产生。
- 2) 中断服务例程启动，给出信号量以使延迟处理任务解除阻塞。
- 3) 当中断服务例程退出时，延迟处理任务得到执行。延迟处理任务做的第一件事便是获取信号量。
- 4) 延迟处理任务完成中断事件处理后，试图再次获取信号量——如果此时信号量无效，任务将切入阻塞等待事件发生。

在中断以相对较低的频率发生的情况下，上面描述的流程是足够而完美的。如果在延迟处理任务完成上一个中断事件的处理之前，新的中断事件又发生了，等效于将新的事件锁存在二值信号量中，使得延迟处理任务在处理完上一个事件之后，立即就可以处理新的事件。也就是说，延迟处理任务在两次事件处理之间，不会有进入阻塞态的机会，因为信号量中锁存有一个事件，所以当 `xSemaphoreTake()` 调用时，信号量立即有效。这种情形如图 7.19 所示。

从图 7.19 中可以看到，一个二值信号量最多只可以锁存一个中断事件。在锁存的事件还未被处理之前，如果还有中断事件发生，那么后续发生的中断事件将会丢失。如果用计数信号量代替二值信号量，那么可以避免这种丢中断的情形。

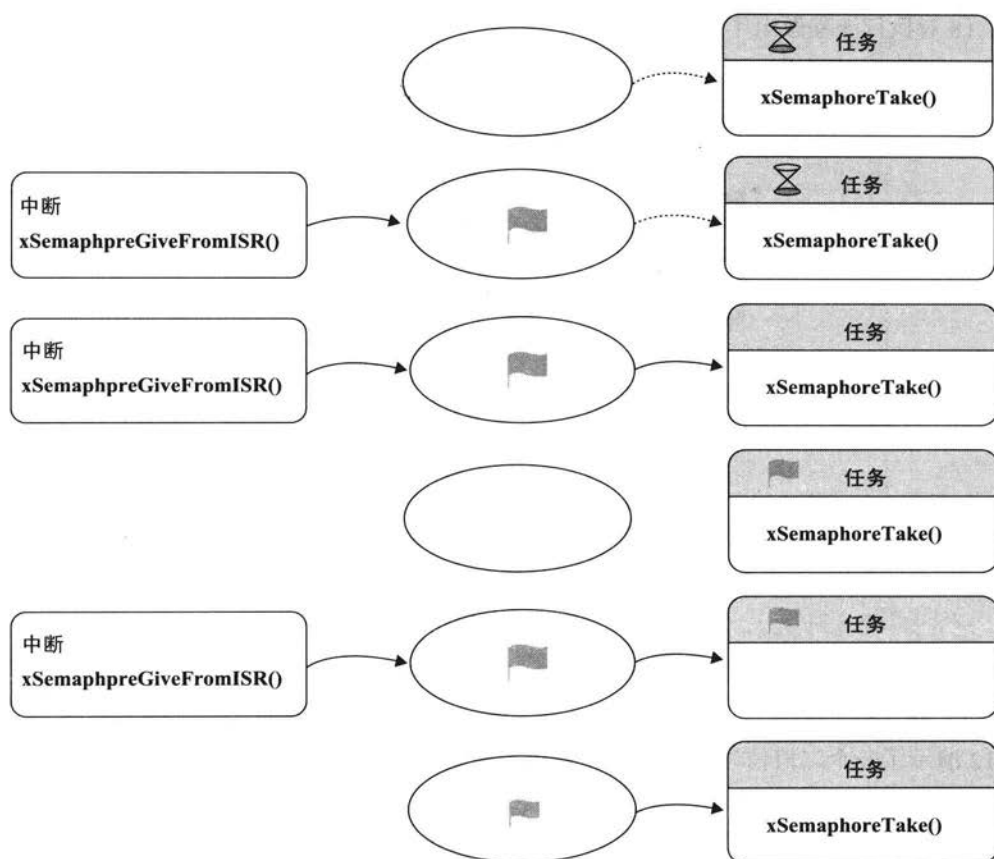


图 7.19 一个二值信号量最多只能锁存一个中断事件

就如同我们可以把二值信号量看做只有一个数据单元的队列一样，计数信号量可以看做深度大于 1 的队列。任务其实对队列中存储的具体数据并不感兴趣——其只关心队列是空还是非空。

计数信号量每次被给出（Given），其队列中的另一个空间将会被使用。队列中的有效数据单元个数就是信号量的“计数”（Count）值。

计数信号量有以下两种典型用法：

1. 事件计数

在这种用法中，每次事件发生时，中断服务例程都会“给出”（Give）信号量——信号量在每次被给出时其计数值加 1。延迟处理任务每处理一个任务都会“获取”（Take）一次信号量——信号量在每次被获取时其计数值减 1。信号量的计数值其实就是已发生事件的数目与已处理事件的数目之间的差值。这种机制可以参考图 7.20。

用于事件计数的计数信号量，在被创建时其计数值被初始化为 0。

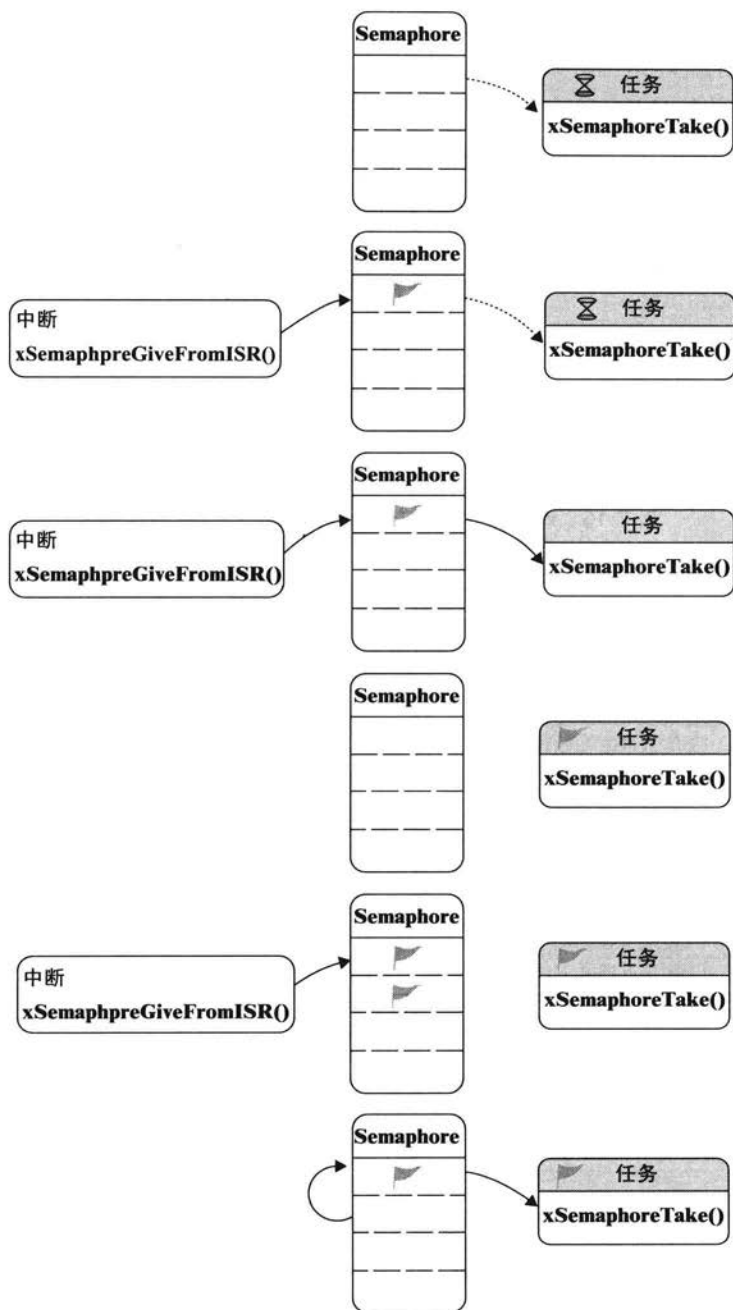


图 7.20 使用计数信号量对事件“计数”

2. 资源管理

在这种用法中，信号量的计数值用于表示可用资源的数目。一个任务要获取资源的控制权，

其必须先获得信号量——使信号量的计数值减 1。当计数值减至 0，则表示没有可用资源。当任务利用资源完成工作后，将给出（归还）信号量——使信号量的计数值加 1。

用于资源管理的信号量，在创建时其计数值被初始化为可用资源总数。

xSemaphoreCreateCounting() API 函数

FreeRTOS 中所有种类的信号量句柄都由声明为 xSemaphoreHandle 类型的变量保存。

信号量在使用前必须先被创建。可以使用 xSemaphoreCreateCounting() API 函数来创建一个计数信号量。函数原型见程序清单 7.49，各参数及描述见表 7.15。

程序清单 7.49 xSemaphoreCreateCounting() API 函数原型

```
xSemaphoreHandle xSemaphoreCreateCounting(  
    unsigned portBASE_TYPE uxMaxCount,  
    unsigned portBASE_TYPE uxInitialCount );
```

表 7.15 xSemaphoreCreateCounting() 参数与返回值

参 数	描 述
uxMaxCount	<ul style="list-style-type: none">❑ 最大计数值。如果把计数信号量类比为队列的话，uxMaxCount 值就是队列的最大深度❑ 当此信号量用于对事件计数或锁存事件的话，uxMaxCount 就是可锁存事件的最大数目❑ 当此信号量用于对一组资源的访问进行管理的话，uxMaxCount 应当设置为所有可用资源的总数
uxInitialCount	<ul style="list-style-type: none">❑ 信号量的初始计数值❑ 当此信号量用于事件计数的话，uxInitialCount 应当设置为 0——因为当信号量被创建时，还没有事件发生❑ 当此信号量用于资源管理的话，uxInitialCount 应当等于 uxMaxCount——因为当信号量被创建时，所有的资源都是可用的
返回值	<ul style="list-style-type: none">❑ 如果返回 NULL 值，表示堆上内存空间不足，所以 FreeRTOS 无法为信号量结构分配内存导致信号量创建失败❑ 如果返回非 NULL 值，则表示信号量创建成功。此值应当被保存起来作为这个信号量的句柄

例 13 利用计数信号量对任务和中断进行同步

例 13 用计数信号量代替二值信号量对例 12 的实现进行了改进。修改 main() 函数调用 xSemaphoreCreateCounting()，以代替对 xSemaphoreCreateBinary() 的调用。新的 API 调用如程序清单 7.50 所示。

程序清单 7.50 使用 xSemaphoreCreateCounting() 创建一个计数信号量

```
/* 在信号量使用之前必须先创建。本例中创建了一个计数信号量。此信号量的最大计数值为 10，初始计数值为 0 */  
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

为了模拟多个事件以高频率发生，需要修改中断服务例程，在每次中断多次“给出”（Give）信号量。每个事件都锁存到信号量的计数值中。修改后的中断服务例程如程序清单 7.51 所示。

程序清单 7.51 例 13 中的中断服务例程实现代码

```

static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    /* 多次给出信号量。第一次给出时使得延迟处理任务解除阻塞。后续给出用于演示利用该信号量锁存事件，以
    便处理任务轮流处理这些中断事件，而不致使某一中断事件得不到响应。此处仅用这种方式来模拟处理器产生多
    个中断，尽管这些事件只是在单次中断中模拟出来的 */
    xSemaphoreGiveFromISR( xCountingSemaphore,
                           &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore,
                           &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore,
                           &xHigherPriorityTaskWoken );
    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* 给出信号量以使得等待此信号量的任务解除阻塞。如果解除阻塞的任务的优先级高于当前任务的优先
        级——强制进行一次任务切换，以确保中断直接返回到解除阻塞的任务（优先级更高）。说明：在实际使用
        中，ISR 中强制上下文切换的宏依赖于具体移植。对于实际使用的平台，请参考对应移植自带的示例程序，
        以决定正确的语法和符号 */
        portSWITCH_CONTEXT();
    }
}

```

其他函数都复用例 12 中的代码，保持不变。

7.4.3 在中断服务例程中使用队列

`xQueueSendToFrontFromISR()`、`xQueueSendToBackFromISR()` 与 `xQueueReceiveFromISR()` 分别是 `xQueueSendToFront()`、`xQueueSendToBack()` 与 `xQueueReceive()` 的中断安全版本，专门用于中断服务例程。

信号量用于事件通信。而队列不仅可以用于事件通信，还可以用来传递数据。

`xQueueSendToFrontFromISR()` 与 `xQueueSendToBackFromISR()` API 函数

注意，`xQueueSendFromISR()` 完全等同于 `xQueueSendToBackFromISR()`。函数原型见程序清单 7.52 和程序清单 7.53，各参数及描述见表 7.16。

程序清单 7.52 `xQueueSendToFrontFromISR()` API 函数原型

```

portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,
    void *pvItemToQueue,
    portBASE_TYPE *pxHigherPriorityTaskWoken );

```

程序清单 7.53 `xQueueSendToBackFromISR()` API 函数原型

```

portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,
    void *pvItemToQueue,
    portBASE_TYPE *pxHigherPriorityTaskWoken );

```

表 7.16 xQueueSendToFrontFromISR 与 xQueueSendToBackFromISR() 参数与返回值

参 数	描 述
xQueue	目标队列的句柄。这个句柄即调用 xQueueCreate() 创建该队列时的返回值
pvItemToQueue	<ul style="list-style-type: none">❑ 发送数据的指针。其指向将要拷贝到目标队列中的数据单元❑ 因为在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间拷贝对应长度的数据到队列的存储区域
pxHigherPriorityTaskWoken	<ul style="list-style-type: none">❑ 对某个队列而言，可能不止一个任务处于阻塞态并在等待其数据有效。调用 xQueueSendToFrontFromISR() 或 xQueueSendToBackFromISR() 会使得队列数据变为有效，所以会让其中一个等待任务切出阻塞态。如果调用这两个 API 函数使得一个任务解除阻塞，并且这个任务的优先级高于当前任务（也就是被中断的任务），那么 API 会在函数内部将 *pxHigherPriorityTaskWoken 设为 pdTRUE❑ 如果这两个 API 函数将此值设为 pdTRUE，则在中断退出前应当进行一次上下文切换。这样才能保证中断直接返回到就绪态任务中优先级最高的任务
返回值	<p>有两个可能的返回值：</p> <ul style="list-style-type: none">❑ pdPASS。返回 pdPASS 只会有一种情况，那就是数据被成功发送到队列中❑ errQUEUE_FULL。如果因队列已满而无法写入数据，则返回 errQUEUE_FULL

有效使用队列

FreeRTOS 的大多数 Demo 应用程序中都包含一个简单的 UART 驱动，其通过队列将字符传递到发送中断例程，也使用队列将字符从接收中断例程中传递出来。发送或接收的每个字符都通过队列单独传递。这些 UART 驱动的这种实现方式只是单纯为了演示如何在中断中使用队列。实际上，利用队列传递单个字符是极其低效的，特别是在波特率较高的时候，所以这种方式并不建议用在产品代码中。实际应用中可以采用下述更有效的方式：

- ❑ 将接收到的字符先缓存到内存中。当接收到一个传输完成消息，或是检测到传输中断后，使用信号量让某个任务解除阻塞，这个任务将对字符缓存进行处理。
- ❑ 在中断服务中直接解析接收到的字符，然后通过队列将解析后经解码得到的命令发送到处理任务。这种技术仅适用于数据流能够快速解析的场合，这样整个数据解析工作才可以放在中断服务中完成。

例 14 利用队列在中断服务中发送或接收数据

本例演示在同一个中断服务中使用 xQueueSendToBackFromISR() 和 xQueueReceiveFromISR()。和之前一样，采用软件中断以方便实现。

创建一个周期任务用于每 200 ms 向队列中发送 5 个数值，5 个数值都发送完后便产生一个软件中断。周期任务的实现代码参见程序清单 7.54。

程序清单 7.54 例 14 中的写队列任务实现代码

```
static void vIntegerGenerator( void *pvParameters )
{
    portTickType xLastExecutionTime;
```

```

unsigned portLONG ulValueToSend = 0;
int i;

/* 初始化变量，用于调用 vTaskDelayUntil() */
xLastExecutionTime = xTaskGetTickCount();
for( ;; )
{
    /* 这是个周期性任务。此任务每 200ms 执行一次。执行到此处，任务进入阻塞态，200ms 延时结束，任务重新进入就绪态，若处理器处于空闲态，则立即进入运行态 */
    vTaskDelayUntil(&xLastExecutionTime,
                    200/portTICK_RATE_MS );

    /* 连续 5 次发送递增值到队列。这些数值将在中断服务例程中读出。中断服务例程会将队列读空，所以此任务可以确保将所有的数值都发送到队列。因此不需要指定阻塞超时时间 */
    for( i = 0; i < 5; i++ )
    {
        xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
        ulValueToSend++;
    }
    /* 产生中断，以让中断服务例程读取队列 */
    vPrintString( "Generator task - About to generate an interrupt.\r\n" );

    __asm( int 0x82 ) /* This line generates the interrupt. */

    vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
}
}

```

中断服务例程重复调用 `xQueueReceiveFromISR()`，直到被周期任务写到队列的数值都被读出，以将队列读空。每个接收到的数值的低两位用于一个字符串数组的索引，被索引到的字符串的指针将通过调用 `xQueueSendFromISR()` 发送到另一个队列中。中断服务例程的实现代码参见程序清单 7.55。

程序清单 7.55 例 14 的中断服务例程实现代码

```

static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    static unsigned long ulReceivedNumber;

    /* 这些字符串被声明为 static const，以保证它们不会被定位到 ISR 的栈空间中，即使 ISR 没有运行，它们也是存在的 */
    static const char *pcStrings[] =
    {
        "String 0\r\n",
        "String 1\r\n",
        "String 2\r\n",
        "String 3\r\n"
    }
}

```

```

};
xHigherPriorityTaskWoken = pdFALSE;
/* 重复执行, 直到队列为空 */
while( xQueueReceiveFromISR( xIntegerQueue,
                             &ulReceivedNumber,
                             &xHigherPriorityTaskWoken ) !=
                             errQUEUE_EMPTY )
{
    /* 截断收到的数据, 保留低两位 ( 数值范围 0~3)。然后将索引到的字符串指针发送到另一个队列 */
    ulReceivedNumber &= 0x03;
    xQueueSendToBackFromISR( xStringQueue,
                             &pcStrings[ ulReceivedNumber ],
                             &xHigherPriorityTaskWoken );
}
/* 被队列读写操作解除阻塞的任务, 其优先级是否高于当前任务? 如果是, 则进行任务上下文切换 */
if( xHigherPriorityTaskWoken == pdTRUE )
{
    /* 说明: 在实际使用中, ISR 中强制上下文切换的宏依赖于具体移植。其他平台下的移植可能有不同的语法要求。对于实际使用的平台, 请参考对应移植自带的示例程序, 以决定正确的语法和符号 */
    portSWITCH_CONTEXT();
}
}
}

```

另一个任务将接收从中断服务例程发出的字符串指针。此任务在读队列时被阻塞, 直到队列中有消息到来, 并将接收到的字符串打印输出。其实现代码参见程序清单 7.56。

程序清单 7.56 例 14 中的字符串接收任务实现, 其接收来自中断服务例程的字符串, 并打印输出

```

static void vStringPrinter( void *pvParameters )
{
    char *pcString;
    for( ;; )
    {
        /* 阻塞队列等待数据到达 */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );
        /* 打印接收的字符串 */
        vPrintString( pcString );
    }
}

```

和前面介绍的一样, main() 函数创建队列和任务, 然后启动调度器。具体代码见程序清单 7.57。

程序清单 7.57 例 14 中的 main() 函数实现

```

int main( void )
{
    /* 队列使用前必须先创建。本例中创建了两个队列。一个队列用于保存类型为 unsigned long 的变量, 另一个队列用于保存类型为 char* 的变量。两个队列的深度都为 10。在实际应用中应当检测返回值以确保队列创建成功 */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
}

```

```

xStringQueue = xQueueCreate( 10, sizeof( char * ) );

/* 安装中断服务例程 */
_dos_setvect( 0x82, vExampleInterruptHandler );

/* 创建任务用于向中断服务例程中发送数值。此任务优先级为 1 */
xTaskCreate( vIntegerGenerator,
            "IntGen",
            1000,
            NULL,
            1,
            NULL );

/* 创建任务用于从中断服务例程中接收字符串，并打印输出。优先级为 2 */
xTaskCreate( vStringPrinter,
            "String",
            1000,
            NULL,
            2,
            NULL );

/* 启动调度器，并执行已创建的任务 */
vTaskStartScheduler();

/* 如果一切正常，main() 函数不会执行到这里，因为调度器已经开始运行任务。
   但如果程序运行到了这里，很可能是由于系统内存不足而无法创建空闲任务 */
for( ;; );
}

```

例 14 的执行流程参考图 7.21。

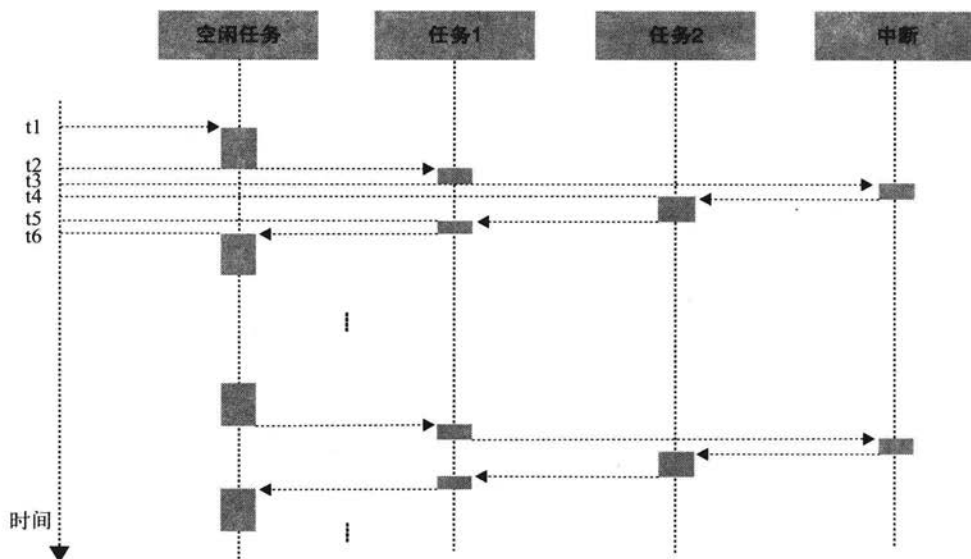


图 7.21 例 14 的执行流程

7.4.4 中断嵌套

最新的 FreeRTOS 移植中允许中断嵌套。中断嵌套需要在 FreeRTOSConfig.h 中定义表 7.17 详细列出的一个或两个常量。

表 7.17 控制中断嵌套的常量

常 量	描 述
configKERNEL_INTERRUPT_PRIORITY	<ul style="list-style-type: none"> □ 设置系统心跳时钟的中断优先级 □ 如果在移植中没有使用常量 configMAX_SYSCALL_INTERRUPT_PRIORITY，那么需要调用中断安全版本 FreeRTOS API 的中断都必须运行在此优先级上
configMAX_SYSCALL_INTERRUPT_PRIORITY	设置中断安全版本 FreeRTOS API 可以运行的最高中断优先级

建立一个全面的中断嵌套模型需要设置 configMAX_SYSCALL_INTERRUPT_PRIORITY 为比 configKERNEL_INTERRUPT_PRIORITY 更高的优先级。这种模型如图 7.22 所示。图 7.22 所示的情形假定常量 configMAX_SYSCALL_INTERRUPT_PRIORITY 设置为 3，configKERNEL_INTERRUPT_PRIORITY 设置为 1。同时也假定这种情形基于一个具有 7 个不同中断优先级的微控制器。这里的 7 个优先级仅仅是本例的一种假定，并非对应于任何一种特定的微控制器架构。

在任务优先级和中断优先级之间常常会产生一些混淆。图 7.22 所示的中断优先级是由微控制器架构所定义的。中断优先级是硬件控制的优先级，中断服务例程的执行会与之关联。因为任务并非运行在中断服务中，所以赋予任务的软件优先级与赋予中断源的硬件优先级之间没有任何关系。

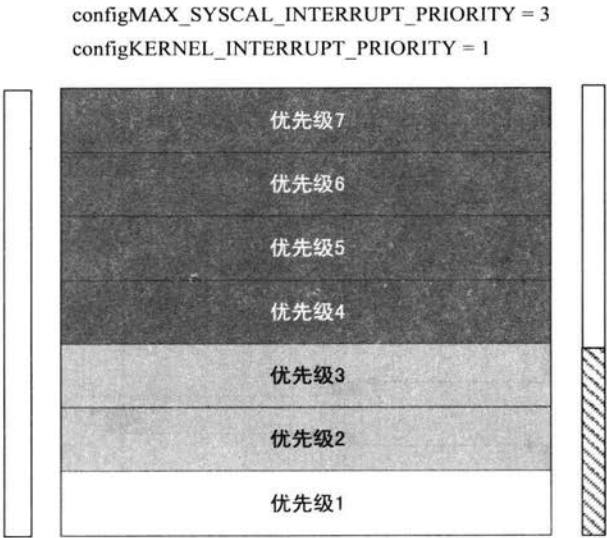


图 7.22 中断控制常量影响中断嵌套行为

从图 7.22 中可以看出：

- ❑ 处于中断优先级 1~3（含）的中断会被内核或处于临界区的应用程序阻塞执行，但是它们可以调用中断安全版本的 FreeRTOS API 函数。
- ❑ 处于中断优先级 4 及以上的中断不受临界区影响，所以其不会被内核的任何行为阻塞，可以立即得到执行——这是由微控制器本身对中断优先级的限定所决定的。通常需要严格时间精度的功能（如电机控制）会使用高于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的优先级，以保证调度器不会对其中断响应时间造成抖动。
- ❑ 不需要调用任何 FreeRTOS API 函数的中断，可以自由地使用任意优先级。

对 ARM Cortex M3 用户的一点提示

Cortex M3 使用低优先级号数值表示逻辑上的高优先级中断。这显得不是那么直观，所以很容易忘记。如果你想对某个中断赋予低优先级，则必须使用一个高优先级号数值。千万不要给它指定优先级号 0（或是其他低优先级号数值），因为这将会使得这个中断在系统中拥有最高优先级——如果这个优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY，将很可能导致系统崩溃。

Cortex M3 内核的最低优先级为 255，但是不同的 Cortex M3 处理器厂商实现的优先级位数不尽相同，而各自的配套库函数也使用了不同的方式来支持中断优先级。比如 STM32，ST 的驱动库中将最低优先级指定为 15，而最高优先级指定为 0。

7.5 资源管理

在多任务系统中，为了避免多个任务同时访问同一资源造成的数据一致性异常、数据损坏或其他类似的错误，通常采用互斥信号量这一机制来协调任务级的行为。比如，多个任务同时访问打印机资源，并向其发送打印输出的文本信息。若没有对打印机资源的协调管理机制，则打印出来的文本将变得难以理解，甚至无法正常阅读。本节主要从资源管理的角度介绍常用的资源管理机制——“互斥量”，涉及的主要概念有临界区、互斥、调度器挂起、优先级反转及互斥量的使用等。

7.5.1 基本概念

变量的非原子访问

更新结构体的多个成员变量，或是更新的变量长度超过了架构的自然长度（比如，更新一个 16 位机上的 32 位变量）均是非原子操作的例子。如果中断这样的操作，将可能导致数据损坏或丢失。

函数重入

如果一个函数可以安全地被多个任务调用，或在任务与中断中均可调用，则这个函数是可重入的。

每个任务都单独维护自己的栈空间及其自身所在的内存寄存器组中的值。如果一个函数除了访问自己栈空间上分配的数据或内核寄存器中的数据外，不会访问其他任何数据，则这个函数就是可重入的。程序清单 7.58 展示了一个可重入函数，而程序清单 7.59 展示了一个不可重入函数的例子。

程序清单 7.58 可重入函数示例

```
/* 将参数传入该函数。该参数将保存在堆栈或 CPU 寄存器中，这两种方式都能够保障存储数据的安全性，因为每个任务都有自己独立的堆栈空间和寄存器组（备份）*/
long lAddOneHundered( long lVar1)
{
    /* 根据编译器或优化方式不同，函数变量可能分配在任务的堆栈空间或寄存器中。每个调用该函数的任务或中断将保留 lVar2 的备份 */
    long lVar2;
    lVar2 = lVar1+ 100;

    /* 通常，返回值置于 CPU 寄存器中，尽管也有可能存放于堆栈中 */
    return lVar2;
}
```

程序清单 7.59 不可重入函数示例

```
/* 本例的 lVar1 是一个全局变量，因此调用该函数的任务将获取该变量的一个副本 */
long lVar1;
long lNonsenseFunction( void )
{
    /* 由于该变量为静态变量，因此为其分配的存储区域并不位于堆栈中，调用该函数的每个任务将获取该变量的副本 */
    static long lState = 0;
    long lReturn;
    switch( lState )
    {
        case 0 : lReturn = lVar1+ 10;
                lState = 1;
                break;
        case 1: lReturn = lVar1+ 20;
                lState = 0;
                break;
    }
}
```

互斥

访问一个被多任务共享，或是被任务与中断共享的资源时，需要采用互斥技术以保证数据在任何时候都保持一致性。这样做的目的是要确保任务从开始访问资源就具有排他性，直至这个资源又恢复到完整状态。

FreeRTOS 提供了多种特性用以实现互斥，但是最好的互斥方法（如果可能的话，任何时

候都应当如此) 还是通过精心设计应用程序, 尽量不要共享资源, 或者每个资源都通过单任务访问。

7.5.2 临界区与挂起调度器

基本临界区

基本临界区 (Critical Section) 是指宏 `taskENTER_CRITICAL()` 与 `taskEXIT_CRITICAL()` 之间的代码区间, 程序清单 7.60 是一段范例代码。Critical Section 也被称作 Critical Region。

程序清单 7.60 使用临界区对寄存器的访问进行保护

```

/* 为了保证对 PORTA 寄存器的访问不被中断, 将访问操作放入临界区。
进入临界区 */
taskENTER_CRITICAL();

/* 在 taskENTER_CRITICAL() 与 taskEXIT_CRITICAL() 之间不会切换到其他任务。中断可以执行, 允许嵌套, 但是只针对优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断, 而且这些中断不允许访问 FreeRTOS API 函数 */
PORTA |= 0x01;

/* 已经完成了对 PORTA 的访问, 因此可以安全地离开临界区了 */
taskEXIT_CRITICAL();

```

本书采用的范例工程使用了一个名为 `vPrintString()` 的函数, 用于向标准输出设备写入字符串。这个标准输出即 Open Watcom DOS 可执行程序的终端窗口。`vPrintString()` 被多个不同的任务调用, 所以理论上其函数实现中应当使用一个临界区对标准输出进行保护。如程序清单 7.61 所示。

程序清单 7.61 `vPrintString()` 的一种可能的实现方法

```

void vPrintString( const portCHAR *pcString )
{
    /* 向 stdout 中写入字符串, 使用临界区这种原始的方法实现互斥 */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();

    /* 允许按任意键停止应用程序运行。如果实际的应用程序使用了键值, 那么还需要对键盘输入进行保护 */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}

```

临界区是提供互斥功能的一种非常原始的实现方法。临界区的工作仅仅是简单地全部关闭中断，或是关掉优先级在 configMAX_SYSCAL_INTERRUPT_PRIORITY 及以下的中断——依赖于具体使用的 FreeRTOS 移植。抢占式上下文切换只可能在某个中断中完成，所以调用 taskENTER_CRITICAL() 的任务可以在中断关闭的时段一直保持运行态，直到退出临界区。

临界区必须只具有很短的时间，否则会反过来影响中断响应时间。在每次调用 taskENTER_CRITICAL() 之后，必须尽快配套调用一个 taskEXIT_CRITICAL()。从这个角度来看，对标准输出的保护不应当采用临界区（如程序清单 7.62 所示），因为写终端在时间上会是一个相对长的操作。DOS 模拟器和 Open Watcom 处理终端输出时没有采用这种互斥方式，其库函数中是没有关中断的。本章中的示例代码会继续探索其他解决方案。

临界区嵌套是安全的，因为内核维护了一个嵌套深度计数。临界区只会在嵌套深度为 0 时才会真正退出，即在为每个之前调用的 taskENTER_CRITICAL() 都配套调用了 taskEXIT_CRITICAL() 之后。

挂起（锁定）调度器

也可以通过挂起调度器来创建临界区。挂起调度器有些时候也称为锁定调度器。

基本临界区保护一段代码区间不被其他任务或中断打断。由挂起调度器实现的临界区只可以保护一段代码区间不被其他任务打断，因为在这种方式下，中断是使能的。

如果一个临界区太长而不适合简单地关中断来实现，那么可以考虑采用挂起调度器的方式。但是唤醒（resuming 或 un-suspending）调度器却是一个时间相对长的操作。所以评估哪种是最佳方式需要结合实际情况。

vTaskSuspendAll() API 函数

可以通过调用 vTaskSuspendAll() 来挂起调度器。函数原型见程序清单 7.62。挂起调度器可以停止上下文切换而不用关中断。如果某个中断在调度器挂起过程中要求进行上下文切换，则这个请求也会被挂起，直到调度器被唤醒后才会执行。

程序清单 7.62 vTaskSuspendAll() API 函数原型

```
void vTaskSuspendAll( void );
```

在调度器处于挂起状态时，不能调用 FreeRTOS API 函数。

xTaskResumeAll() API 函数

嵌套调用 vTaskSuspendAll() 和 xTaskResumeAll() 是安全的，因为内核维护了一个嵌套深度计数。调度器只会在嵌套深度计数为 0 时才会被唤醒，即在为每个之前调用的 vTaskSuspendAll() 都配套调用了 xTaskResumeAll() 之后。xTaskResumeAll() 函数原型见程序清单 7.63，参数及描述见表 7.18。

程序清单 7.63 xTaskResumeAll() API 函数原型

```
portBASE_TYPE xTaskResumeAll( void );
```

表 7.18 xTaskResumeAll() 返回值

参 数	描 述
返回值	在调度器挂起过程中，上下文切换请求也会被挂起，直到调度器被唤醒后才会执行。如果一个挂起的上下文切换请求在 xTaskResumeAll() 返回前执行，则函数返回 pdTRUE。在其他情况下，xTaskResumeAll() 返回 pdFALSE

程序清单 7.64 展示了实际使用的 vPrintString() 实现代码。这种实现方式是通过挂起调度器的方式来保护终端输出的。

程序清单 7.64 vPrintString() 实现代码

```
void vPrintString( const portCHAR *pcString )
{
    /* 将字符串写入终端，挂起调度器，作为实现互斥的一种方法 */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();
    /* 允许任意键中止程序运行 */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

7.5.3 互斥量

互斥量是一种特殊的二值信号量，用于控制在两个或多个任务间访问共享资源。单词 MUTEX（互斥量）源于“MUTual EXclusion”。

在用于互斥的场合，互斥量从概念上可看做与共享资源关联的令牌。一个任务想要合法地访问资源，其必须先成功地“得到”（Take）该资源对应的令牌（成为令牌持有者）。当令牌持有者完成资源使用，其必须马上“归还”（Give）令牌。只有归还了令牌，其他任务才可能成功持有，也才可能安全地访问该共享资源。一个任务除非持有了令牌，否则不允许访问共享资源。

虽然互斥量与二值信号量之间具有很多相同的特性，但图 7.23 展示的情形（互斥量用于互斥功能）完全不同于图 7.19 展示的情形（二值信号量用于同步）。两者间最大的区别在于信号量在被获得之后所发生的事情：

- ❑ 用于互斥的信号量必须归还。
- ❑ 用于同步的信号量通常是完成同步之后便丢弃，不再归还。

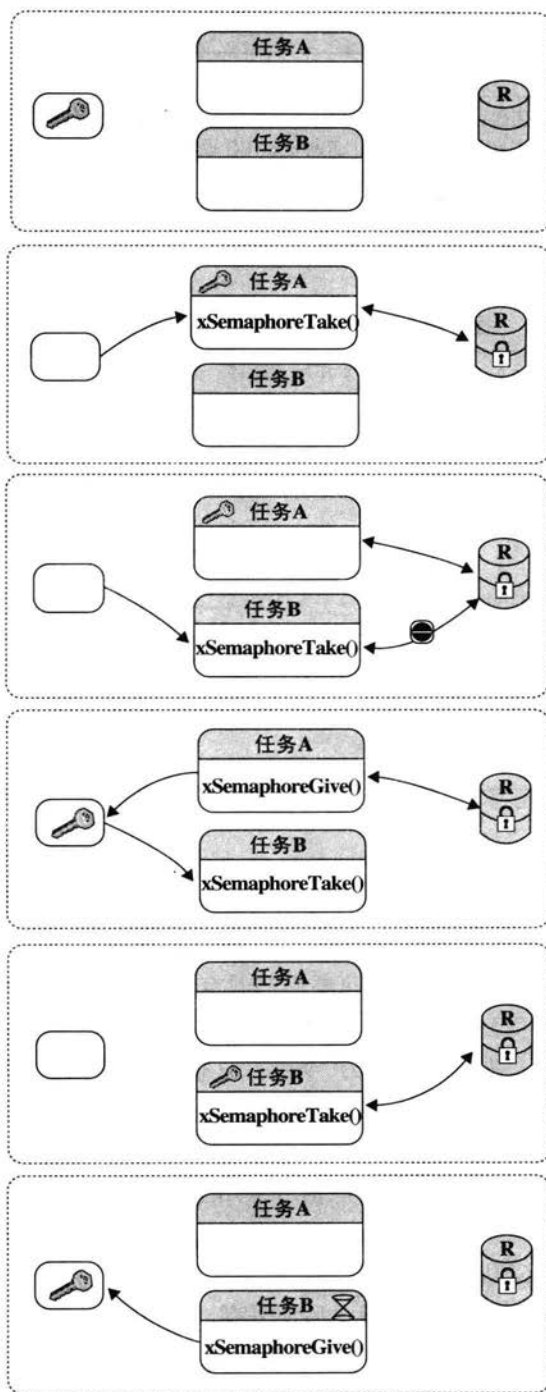


图 7.23 互斥量用于互斥功能

这种机制纯粹是工作于应用程序作者制定的规则之下。任务不是在任何时候都可以访问资源是不需要理由的,因为这是所有任务达成的一致,除非它们能成为互斥量的持有者。

xSemaphoreCreateMutex() API 函数

互斥量是一种信号量。FreeRTOS 中所有种类的信号量句柄都保存在类型为 xSemaphoreHandle 的变量中。

互斥量在使用前必须先创建。创建一个互斥量类型的信号量需要使用 xSemaphoreCreateMutex() API 函数。函数原型见程序清单 7.65, 返回值见表 7.19。

程序清单 7.65 xSemaphoreCreateMutex() API 函数原型

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

表 7.19 xSemaphoreCreateMutex() 返回值

参 数	描 述
返回值	<p>□ 如果返回 NULL 表示互斥量创建失败。原因是内存堆空间不足导致 FreeRTOS 无法为互斥量分配结构数据空间</p> <p>□ 返回非 NULL 值表示互斥量创建成功。返回值应当保存起来并作为该互斥量的句柄</p>

例 15 使用信号量重写 vPrintString()

本例创建了一个新版本的 vPrintString(), 称为 prvNewPrintString(), 然后在多任务中调用这个新版函数。prvNewPrintString() 具有与 vPrintString() 完全相同的功能, 只是在实现上使用互斥量代替基本临界区来实现对标准输出的控制。prvNewPrintString() 的实现代码参见程序清单 7.66。

程序清单 7.66 prvNewPrintString() 实现代码

```
static void prvNewPrintString( const portCHAR *pcString )
{
    /* 互斥量在调度器启动之前就已创建, 所以在此任务运行时信号量就已经存在了。试图获得互斥量。如果互斥量无效, 则将阻塞, 进入无超时等待。xSemaphoreTake() 只可能在成功获得互斥量后返回, 所以无需检测返回值。如果指定了等待超时时间, 则代码必须检测到 xSemaphoreTake() 返回 pdTRUE 后, 才能访问共享资源 (此处是指标准输出) */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* 程序执行到这里表示已经成功持有互斥量。现在可以自由访问标准输出, 因为任意时刻只会会有一个任务能持有互斥量 */
        printf( "%s", pcString );
        fflush( stdout );
        /* 互斥量必须归还 */
    }
    xSemaphoreGive( xMutex );

    /* 允许任意键中止程序运行 */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

```

    }
}

```

prvNewPrintString() 被一个任务的两个实例重复调用。在每次调用之间采用了一个随机延迟时间。任务的入口参数用于向任务的每个实例传递各自的输出字符串。任务 prvPrintTask() 的实现代码参见程序清单 7.67。

程序清单 7.67 例 15 中任务 prvPrintTask() 的实现代码

```

static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;
    /* 该任务的两个实例都采用传参的方式将打印的字符串传递给打印任务，需要注意的是，获得该参数后，需要
    将其转换为所需要的类型 */
    pcStringToPrint = ( char * ) pvParameters;
    for( ;; )
    {
        /* 用新定义的函数打印字符串 */
        prvNewPrintString( pcStringToPrint );

        /* 等待一个伪随机时间。注意函数 rand() 不要求可重入，因为在本例中 rand() 的返回值并不重要。但
        在安全性要求更高的应用程序中，需要用用一个可重入版本的 rand() 函数或在临界区中调用 rand() 函数 */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}

```

与前面介绍的一样，main() 函数简单地创建互斥量，创建任务，然后启动调度器。具体实现代码参见程序清单 7.68。

任务 prvPrintTask() 的两个实例在创建时指定了不同的优先级。所以运行时，低优先级任务在有些时候会被高优先级的任务抢占。由于使用了互斥量来保证每个任务在访问终端时保持互斥，所以即使任务被抢占，字符串也会正确显示，而不会有其他导致破坏的可能。任务的抢占频率还可以再提高，只需要减少任务在阻塞态中花费的最长时间，本例中这个最长时间默认为 0x1ff 个系统心跳周期。

程序清单 7.68 例 15 的 main() 函数实现

```

int main( void )
{
    /* 信号量使用前必须先创建。本例创建了一个互斥量类型的信号量 */
    xMutex = xSemaphoreCreateMutex();
    /* 本例中的任务会使用一个随机延迟时间，这里给随机数发生器生成种子 */
    srand( 567 );
    /* 在创建任务前，检查信号量是否创建成功 */
    if( xMutex != NULL )
    {
        /* 创建打印字符串任务的两个实例，要打印的字符串通过任务参数传递。由于所创建的两个实例优先级
        不同，因此在运行过程中，将会发生抢占现象 */
        xTaskCreate( prvPrintTask,
                    "Print1",

```


这种最坏的情形如图 7.25 所示。

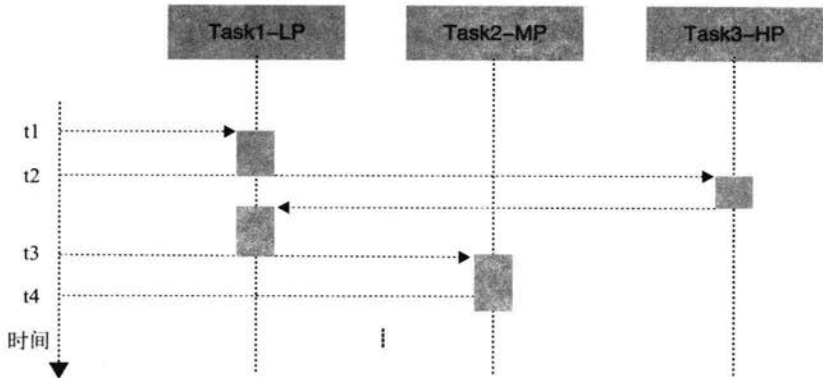


图 7.25 优先级反转的一种最坏情况

优先级反转可能会产生重大问题。但是在一个小型的嵌入式系统中，通常可以在设计阶段就通过规划好资源的访问方式避免出现这个问题。

优先级继承

FreeRTOS 中互斥量与二值信号量十分相似——唯一的区别就是互斥量自动提供了一个基本的“优先级继承”机制。优先级继承是最小化优先级反转负面影响的一种方案——其并不能修正优先级反转带来的问题，仅仅是减小优先级反转的影响。优先级继承使得系统行为的数学分析更为复杂，所以，如果可以避免的话，并不建议系统实现对优先级继承有所依赖。

优先级继承暂时地将互斥量持有者的优先级提升至所有等待此互斥量的任务所具有的最高优先级。持有互斥量的低优先级任务“继承”了等待互斥量的任务的优先级。这种机制在图 7.26 中进行展示。互斥量持有者在归还互斥量时，优先级会自动设置为其原来的优先级。

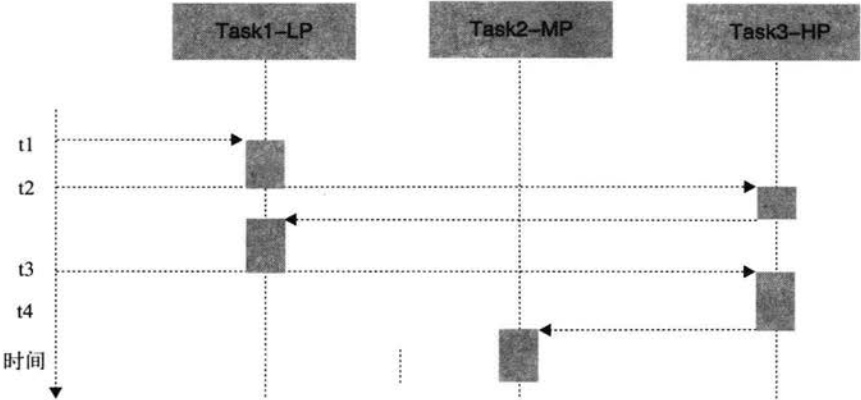


图 7.26 优先级继承最小化优先级反转的影响

由于最好优先考虑避免优先级反转，并且因为 FreeRTOS 本身是面向内存有限的微控制器，所以只实现了最基本的互斥量的优先级继承机制，这种实现假定一个任务在任意时刻只会持有一个互斥量。

死锁

死锁 (Deadlock) 是利用互斥量提供互斥功能的另一个潜在缺陷。死锁有时候会被更戏剧性地称为 “deadly embrace” (抱死)。

当两个任务都在等待被对方持有的资源时，两个任务都无法再继续执行，这种情况就称为死锁。考虑如下情形，任务 A 与任务 B 都需要获得互斥量 X 与互斥量 Y 以完成各自的工作：

- 1) 任务 A 执行，并成功获得了互斥量 X；
- 2) 任务 A 被任务 B 抢占；
- 3) 任务 B 成功获得了互斥量 Y，之后又试图获取互斥量 X，但互斥量 X 已经被任务 A 持有，所以对任务 B 无效。任务 B 选择进入阻塞态以等待互斥量 X 被释放。
- 4) 任务 A 得以继续执行。其试图获取互斥量 Y，但互斥量 Y 已经被任务 B 持有而对任务 A 无效。任务 A 也选择进入阻塞态以等待互斥量 Y 被释放。

这种情形的最终结局是，任务 A 在等待一个被任务 B 持有的互斥量，而任务 B 也在等待一个被任务 A 持有的互斥量。于是发生死锁，因为两个任务都不可能再执行下去了。

和优先级反转一样，避免死锁的最好方法就是在设计阶段就考虑到这种潜在风险，这样设计出来的系统就不应该出现死锁的情况。于实践经验而言，对于一个小型嵌入式系统，死锁并不是一个大问题，因为系统设计者对整个应用程序都非常清楚，所以能够找出发生死锁的代码区域，并消除死锁问题。

7.5.4 互斥的另一种实现

守护任务提供了一种干净利落的方法来实现互斥功能，而不用担心会发生优先级反转和死锁。

守护任务是对某个资源具有唯一所有权的任务。只有守护任务才可以直接访问其守护的资源——其他任务要访问该资源只能间接地通过守护任务提供的服务。

例 16 采用守护任务重写 vPrintString()

例 16 提供了 vPrintString() 的另一种实现方法，这里采用了一个守护任务来管理对标准输出的访问。当一个任务想要向终端写信息的时候，其不能直接调用打印函数，而是将消息发送到守护任务。

守护任务使用了一个 FreeRTOS 队列来对终端实现串行化访问。该任务内部实现不必考虑互斥，因为它是唯一能够直接访问终端的任务。

守护任务大部分时间都处在阻塞态，等待队列中有信息到来。当一个信息到达时，守护任务仅仅简单地将收到的信息写到标准输出上，然后又返回阻塞态，继续等待下一条信息到来。守护任务的具体实现参见程序清单 7.71。

中断中可以写队列，所以中断服务例程也可以安全地使用守护任务提供的服务，从而把信息

输出到终端。在本例中，一个心跳中断回调函数用于每 200 心跳周期就输出一个消息。

心跳回调函数由内核在每次心跳中断时调用。要挂接一个心跳回调函数，需要做以下配置：

- 设置 FreeRTOSConfig.h 中的常量 configUSE_TICK_HOOK 为 1。
- 提供回调函数的具体实现，要求使用程序清单 7.69 中的函数名和原型。

程序清单 7.69 心跳回调函数名及原型

```
void vApplicationTickHook( void );
```

因为心跳回调函数在系统心跳中断的上下文上执行，所以必须保证非常短小，适度占用栈空间，并且不要调用任何名字不带后缀“FromISR”的 FreeRTOS API 函数。守护任务的具体代码见程序清单 7.70。

程序清单 7.70 守护任务

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;

    /* 这是唯一允许直接访问终端输出的任务。任何其他任务想要输出字符串，都不能直接访问终端，而是要将输出的字符串发送到此任务。并且因为只有本任务才可以访问标准输出，所以本任务在实现上不需要考虑互斥和串行化等问题 */
    for( ;; )
    {
        /* 等待信息到达。指定了一个无限长阻塞超时时间，所以不需要检查返回值——此函数只会在成功收到消息时才会返回 */
        xQueueReceive( xPrintQueue,
                      &pcMessageToPrint,
                      portMAX_DELAY );

        /* 输出收到的字符串 */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );
        /* 返回循环执行的开始，等待下条消息到达 */
    }
}
```

信息输出任务与例 15 相似，不同的是，本例中字符串是通过队列发送到守护任务，而不是直接输出到终端。具体实现参见程序清单 7.71。与之前一样，为这个任务创建了两个独立的实例，每个实例输出各自从任务入口参数传入的字符串。

程序清单 7.71 例 16 中的打印输出任务实现代码

```
static void prvPrintTask( void *pvParameters )
{
    int iIndexToString;
    /* 创建该任务的两个实例，任务参数用来传递字符串数组的索引，需要注意的是应将其转换为适合的类型 */
    iIndexToString = ( int ) pvParameters;
    for( ;; )
```

```

{
    /* 打印输出字符串，不能直接输出，通过队列将字符串指针发送到守护任务。队列在调度器启动之前就创建了，所以任务执行时队列已经存在了，与此同时，指定队列的超时时限，此处设置为 0，表示任务将立即返回 */
    xQueueSendToBack( xPrintQueue,
                      &(amp; pcStringsToPrint[ iIndexToString ] ),
                      0 );

    /* 等待一个伪随机时间。注意函数 rand() 不要求可重入，因为在本例中 rand() 的返回值并不重要。但在安全性要求更高的应用程序中，需要用一个可重入版本的 rand() 函数或在临界区中调用 rand() 函数 */
    vTaskDelay( ( rand() & 0x1FF ) );
}
}

```

心跳回调函数仅仅简单地对其被调用次数进行计数，当计数至 200 时就向守护任务发送信息。为了具有更好的演示效果，心跳回调函数将信息发送到队列首，而打印输出任务将信息发送到队列尾。心跳回调函数的实现代码如程序清单 7.72 所示。

程序清单 7.72 心跳回调函数实现代码

```

void vApplicationTickHook( void )
{
    static int iCount = 0;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 每 200 个心跳周期打印一条消息，打印的消息不是直接写入终端，而是发送给守护任务，由守护任务实现字符串打印 */
    iCount++;
    if( iCount >= 200 )
    {
        /* 在本例中，最后一个参数 (xHigherPriorityTaskWoken) 并未使用，但仍是必须的 */
        xQueueSendToFrontFromISR( xPrintQueue,
                                  &(amp; pcStringsToPrint[ 2 ] ),
                                  &xHigherPriorityTaskWoken );

        /* 复位计数值，准备在下个 200 心跳周期结束时再次打印字符串 */
        iCount = 0;
    }
}

```

与前面介绍的一样，main() 函数创建队列和所有任务，然后启动调度器。main() 函数的具体实现参见程序清单 7.73。

程序清单 7.73 例 16 中的 main() 函数实现代码

```

/* 定义任务和中断将会通过守护任务输出的字符串 */
static char *pcStringsToPrint[] =
{
    "Task 1*****\r\n",
    "Task 2 ----- \r\n",
    "Message printed from the tick hook interrupt #####\r\n"
}

```

```

};

/* 声明 xQueueHandle 变量。这个变量将会用于打印任务和中断向守护任务发送消息。*/
xQueueHandle xPrintQueue;

int main( void )
{
    /* 创建队列，深度为 5，数据单元类型为字符指针 */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* 为伪随机数发生器产生种子 */
    srand( 567 );

    /* 检查队列是否创建成功 */
    if( xPrintQueue != NULL )
    {
        /* 创建任务的两个实例，用于向守护任务发送信息。任务入口参数传入需要输出的字符串索引号。这两个任务具有不同的优先级，所以高优先级任务有时会抢占低优先级任务 */
        xTaskCreate( prvPrintTask,
                    "Print1",
                    1000,
                    ( void * ) 0,
                    1,
                    NULL );
        xTaskCreate( prvPrintTask,
                    "Print2",
                    1000,
                    ( void * ) 1,
                    2,
                    NULL );

        /* 创建守护任务。这是唯一一个允许直接访问标准输出的任务 */
        xTaskCreate( prvStdioGatekeeperTask,
                    "Gatekeeper",
                    1000,
                    NULL,
                    0,
                    NULL );

        /* 启动调度器，并执行已创建的任务 */
        vTaskStartScheduler();
    }

    /* 如果一切正常，main() 函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到了这里，很可能是由于系统内存不足而无法创建空闲任务 */
    for( ;; );
}

```

因为守护任务的优先级低于打印任务，所以发送到守护任务的消息会一直保持在队列中，直到两个打印任务都进入阻塞态。在一些情况下，需要给守护任务赋予一个较高的优先级，消息就可以得到更快的处理，但这样做会由于守护任务的开销使得低优先级任务被推迟，直到守护任务完成对受其保护的资源的访问。

7.6 内存管理

嵌入式系统通常对内存配置和时间有着不同的要求。单一的内存分配算法不可能满足所有的应用需求。因此，FreeRTOS 仅提供了三种常见的内存管理策略，用户可能根据自身需求选择响应的内存管理策略，也可以将内存分配作为可移植工作的一部分，实现适合应用需求的内存管理策略。

7.6.1 概述

每当任务、队列或是信号量被创建时，内核需要进行动态内存分配。虽然可以调用标准的 `malloc()` 与 `free()` 库函数，但必须处理以下若干问题：

- ❑ 这两个函数在小型嵌入式系统中可能不可用。
- ❑ 这两个函数的具体实现可能会相对较大，会占用较多宝贵的代码空间。
- ❑ 这两个函数通常不具备线程安全特性。
- ❑ 这两个函数具有不确定性。每次调用时的时间开销都可能不同。
- ❑ 这两个函数会产生内存碎片。
- ❑ 这两个函数会使得链接器配置变得复杂。

因为不同的嵌入式系统具有不同的内存配置和时间要求，所以单一的内存分配算法只可能适合部分应用程序。因此，FreeRTOS 将内存分配作为可移植层面（相对于基本的内核代码部分而言）。这使得不同的应用程序可以提供适合自身的具体实现。

当内核请求内存时，其调用 `pvPortMalloc()` 而不是直接调用 `malloc()`；当释放内存时，调用 `vPortFree()` 而不是直接调用 `free()`。`pvPortMalloc()` 具有与 `malloc()` 相同的函数原型；`vPortFree()` 也具有与 `free()` 相同的函数原型。

FreeRTOS 自带 3 种 `pvPortMalloc()` 与 `vPortFree()` 实现范例，这 3 种方式都会在本章描述。FreeRTOS 的用户可以选用其中一种，也可以采用自己的内存管理方式。

这 3 个范例对应 3 个源文件：`heap_1.c`、`heap_2.c` 与 `heap_3.c`，这 3 个文件都放在目录 `FreeRTOS\Source\Portable\MemMang` 中。早期版本的 FreeRTOS 所采用的原始内存池和内存块分配方案已经被移除了，因为定义内存块和内存池的大小需要较深入的努力和理解。

在小型嵌入式系统中，通常是在启动调度器之前创建任务、队列和信号量。这种情况表明，动态分配内存只会出现在应用程序真正开始执行实时功能之前，而且内存一旦分配就不会再释放。这就意味着选择内存分配方案时不必考虑一些复杂的因素，比如确定性与内存碎片等，而只需要从性能上考虑，比如代码大小和简易性。

7.6.2 内存分配方案范例

Heap_1.c

`Heap_1.c` 实现了一个非常基本的 `pvPortMalloc()` 版本，而且没有实现 `vPortFree()`。如果应用

程序不需要删除任务、队列或者信号量，则具有使用 heap_1 的潜质。Heap_1 总是具有确定性。

这种分配方案是将 FreeRTOS 的内存堆空间看做一个简单的数组。当调用 pvPortMalloc() 时，则将数组又简单地细分为更小的内存块。

数组的总大小（字节为单位）在 FreeRTOSConfig.h 中由 configTOTAL_HEAP_SIZE 定义。以这种方式定义一个巨型数组会让整个应用程序看起来耗费了许多内存——即使是在数组没有进行任何实际分配之前。

需要为每个创建的任务在堆空间上分配一个任务控制块（TCB）和一个栈空间。图 7.27 展示了 heap_1 是如何在任务创建时细分这个简单数组的。

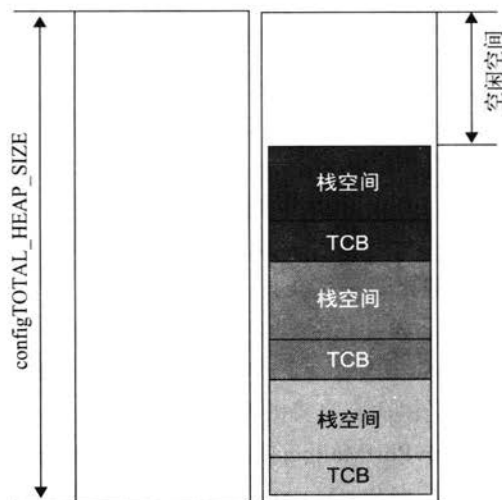


图 7.27 每次创建任务后的内存分配情况

Heap_2.c

Heap_2.c 也是使用了一个由 configTOTAL_HEAP_SIZE 定义大小的简单数组。不同于 heap_1 的是，heap_2 采用了一个最佳匹配算法来分配内存，并且支持内存释放。由于声明了一个静态数组，所以会让整个应用程序看起来耗费了许多内存——即使是在数组没有进行任何实际分配之前。

最佳匹配算法保证 pvPortMalloc() 会使用最接近请求大小的空闲内存块。比如，考虑以下情形：

- ❑ 堆空间中包含了三个空闲内存块，分别为 5 字节、25 字节和 100 字节。
- ❑ pvPortMalloc() 被调用以请求分配 20 字节的内存空间。

因为匹配请求字节数的最小空闲内存块是具有 25 字节的内存块，所以 pvPortMalloc() 会将这个 25 字节块再分为一个 20 字节块和一个 5 字节块，然后返回一个指向 20 字节块的指针。剩下的 5 字节块则保留下来，留待以后调用 pvPortMalloc() 时使用。

Heap_2.c 并不会把相邻的空闲块合并成一个更大的内存块，所以会产生内存碎片——如果分配和释放的总是相同大小的内存块，则内存碎片就不会成为一个问题。Heap_2.c 适用于那些重

复创建与删除具有相同栈空间任务的应用程序。

图 7.28 展示了当任务创建、删除以及再创建过程中，最佳匹配算法是如何工作的。从图 7.28 中可以看出：

- ❑ 图 7.28a 表示数组在创建了 3 个任务后的情形。数组的顶部还剩余一个较大空闲块。
- ❑ 图 7.28b 表示数组在删除了一个任务后的情形。顶部的较大空闲块保持不变，并多出了两个小的空闲块，分别是被删除任务的 TCB 和任务栈。
- ❑ 图 7.28c 表示数组在又创建了一个任务后的情形。创建一个任务会产生两次调用 `pvPortMalloc()`，一次是分配 TCB，一次是分配任务栈（调用 `pvPortMalloc()` 发生在 `xTaskCreate()` API 函数内部）。

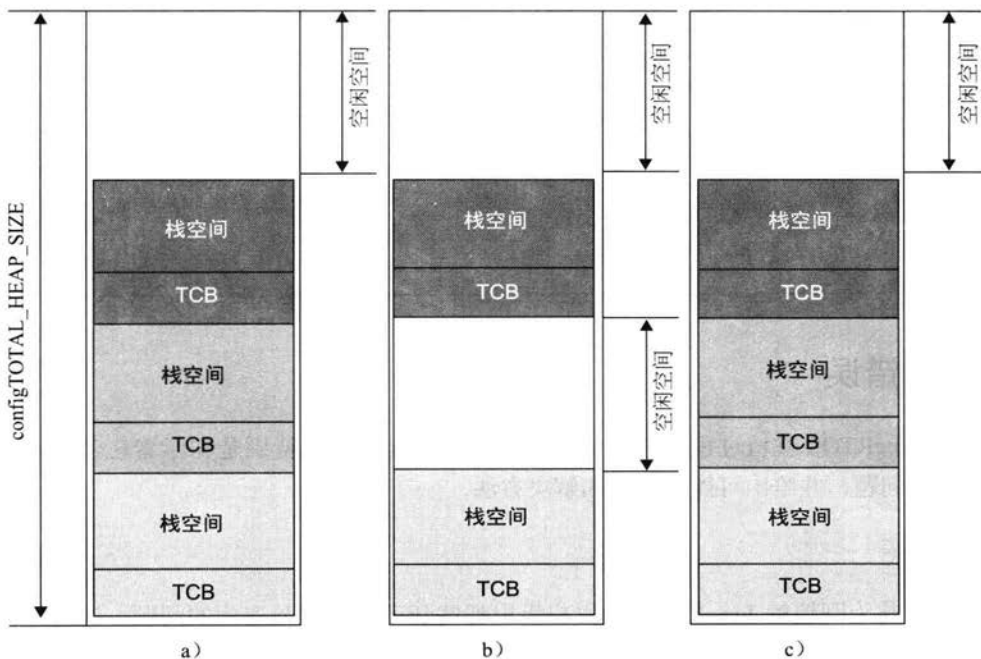


图 7.28 创建和删除任务后的内存分配情况

每个 TCB 都具有相同大小，所以最佳匹配算法可以确保之前被删除的任务占用的 TCB 空间被重新分配用作新任务的 TCB 空间。新建任务的栈空间与之前被删除任务的栈空间大小相同，所以最佳匹配算法会保证之前被删除任务占用的栈空间会被重新分配用作新任务的栈空间。数组顶部的较大空闲块依然保持不变。虽然 `Heap_2.c` 不具备确定性，但是比大多数标准库实现的 `malloc()` 与 `free()` 更有效率。

Heap_3.c

`Heap_3.c` 简单地调用了标准库函数 `malloc()` 和 `free()`，但是通过暂时挂起调度器使得函数调用具备线程安全特性。其实现代码参见程序清单 7.74。

此时的内存堆空间大小不受 `configTOTAL_HEAP_SIZE` 影响，而是由链接器配置决定。

程序清单 7.74 heap_3.c 实现代码

```
void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn;
    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();
    return pvReturn;
}

void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

7.7 常见错误

本节对 FreeRTOS 移植过程中常见的错误进行了简要罗列，特别是初学者在学习实践的过程中可能出现的问题，并给出可能的原因和解决方法。

7.7.1 概述

本章主要是为刚接触 FreeRTOS 的用户指出那些新手通常容易遇到的问题。这里把最主要的篇幅放在栈溢出以及栈溢出侦测上，因为栈相关的问题是过去几年遇到最多的问题。对其他一些比较常见的问题，本章简要地以 FAQ（问答）的形式给出可能的原因和解决方法。

printf-stdarg.c

当调用标准 C 库函数时，栈空间使用量可能会急剧上升，特别是 IO 与字符串处理函数，比如 sprintf()。在 FreeRTOS 下载包中有一个名为 printf-stdarg.c 的文件。这个文件实现了一个栈效率优化版的小型 sprintf()，可以用来代替标准 C 库函数版本。在大多数情况下，这样做可以使得调用 sprintf() 及相关函数的任务对栈空间的需求量小很多。

7.7.2 栈溢出

FreeRTOS 提供了多种特性来辅助跟踪调试栈相关的问题。

uxTaskGetStackHighWaterMark() API 函数

每个任务都独立维护自己的栈空间，栈空间总量在任务创建时进行设定。uxTaskGetStackHighWaterMark() 主要用来查询指定任务的运行历史中其栈空间还差多少就要溢出。这个值被称为栈空间的“高水线”（High Water Mark）。函数原型见程序清单 7.75，参数及描述见表 7.20。

程序清单 7.75 uxTaskGetStackHighWaterMark() API 函数原型

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

表 7.20 uxTaskGetStackHighWaterMark() 参数与返回值

参数	描 述
xTask	<input type="checkbox"/> 被查询任务的句柄——欲知如何获得任务句柄，详情请参见 API 函数 xTaskCreate() 的参数 pxCreatedTask <input type="checkbox"/> 如果传入 NULL 句柄，则任务查询的是自身栈空间的高水线
返回值	任务栈空间的实际使用量会随着任务执行和中断处理过程上下浮动。uxTaskGetStackHighWaterMark() 返回从任务启动执行开始的运行历史中栈空间具有的最小剩余量。这个值是栈空间使用达到最深时剩下的未使用的栈空间。这个值越接近 0，则这个任务就离栈溢出越近

运行时栈侦测——概述

FreeRTOS 包含两种运行时栈侦测机制，由 FreeRTOSConfig.h 中的配置常量 configCHECK_FOR_STACK_OVERFLOW 进行控制。这两种方式都会增加上下文切换开销。

栈溢出回调函数由内核在侦测到栈溢出时调用。要使用栈溢出回调函数，需要进行以下配置：

- ☐ 在 FreeRTOSConfig.h 中把 configCHECK_FOR_STACK_OVERFLOW 设为 1 或 2。
- ☐ 提供回调函数的具体实现，采用程序清单 7.76 所示的函数名和函数原型。

程序清单 7.76 栈溢出回调函数原型

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask,  
signed portCHAR pcTaskName );
```

栈溢出回调函数只是为了使跟踪调试栈空间错误更容易，而无法在栈溢出时对其进行恢复。函数的入口参数传入了任务句柄和任务名，但任务名很可能在溢出时已经遭到破坏。

栈溢出回调函数还可以在中断的上下文中进行调用。某些微控制器在检测到内存访问错误时会产生错误异常，很可能在内核调用栈溢出回调函数之前就触发了错误异常中断。

运行时栈侦测——方法 1

当 configCHECK_FOR_STACK_OVERFLOW 设置为 1 时选用方法 1。

任务被交换出去的时候，该任务的整个上下文被保存到其栈空间中。这时任务栈的使用应当达到了一个峰值。当 configCHECK_FOR_STACK_OVERFLOW 设为 1 时，内核会在任务上下文保存后检查栈指针是否还指向有效栈空间。一旦检测到栈指针的指向已经超出任务栈的有效范围，就会调用栈溢出回调函数。

方法 1 具有较快的执行速度，但栈溢出有可能发生在两次上下文保存之间，无法侦测到这种情况。

运行时栈侦测——方法 2

将 `configCHECK_FOR_STACK_OVERFLOW` 设为 2 就可以选用方法 2。方法 2 在方法 1 的基础上进行了一些补充。

当创建任务时，任务栈空间中就预置了一个标记。方法 2 会检查任务栈的最后 20 字节，查看预置在这里的标记数据是否被覆盖。如果最后 20 字节的标记数据与预设值不同，则调用栈溢出回调函数。

方法 2 没有方法 1 的执行速度快，但测试仅仅 20 字节相对来说也是很快的。这种方法应该可以侦测到任何时候发生的栈溢出，虽然理论上还是有可能漏掉一些情况，但这些情况几乎是可能发生的。

7.7.3 其他常见错误

问题现象：在一个 Demo 应用程序中增加了一个简单的任务，导致应用程序崩溃。

任务创建时需要在内存堆中分配空间。许多 Demo 应用程序定义的堆空间大小只够用于创建 Demo 任务——所以当任务创建完成后，就没有足够的剩余空间来增加其他的任务、队列或信号量。

空闲任务是在 `vTaskStartScheduler()` 调用中自动创建的。如果由于内存不足而无法创建空闲任务，则 `vTaskStartScheduler()` 会直接返回。在调用 `vTaskStartScheduler()` 后加上一条空循环 `for(;;)` 可以使这种错误更加容易调试。

如果要添加更多的任务，可以增加内存堆空间大小，或删除一些已存在的 Demo 任务。

问题现象：在中断中调用一个 API 函数，导致应用程序崩溃。

除了具有后缀为“FromISR”函数名的 API 函数外，千万不要在中断服务例程中调用其他 API 函数。

问题现象：有时候应用程序会在中断服务例程中崩溃。

需要做的第一件事是检查中断是否导致了栈溢出。

在不同的移植平台和不同的编译器上，中断的定义和使用方法是不尽相同的，所以，需要做的第二件事是检查在中断服务例程中使用的语法、宏和调用约定是否符合 Demo 程序的文档描述，以及是否和 Demo 程序中提供的中断服务例程范例相同。

如果应用程序工作在 Cortex-M3 上，需要确定给中断指派优先级时，使用低优先级号数值表示逻辑上的高优先级中断，因为这种方式不太直观，所以很容易忘记。一个比较常见的错误就是，在优先级高于 `configMAX_SYSCALL_INTERRUPT_PRIORITY` 的中断中调用了 FreeRTOS API 函数。

问题现象：在启动第一个任务时，调度器崩溃。

如果使用的是 ARM7，那么请确定调用 `vTaskStartScheduler()` 时处理器处于管理模式 (supervisor mode)。最简单的方式就是在 `main()` 之前的 C 启动代码中将处理器设置为管理模式。

ARM7 的 Demo 应用程序就是这么做的。

如果处理器不在管理模式下，调度器是无法启动的。

问题现象：临界区无法正确嵌套。

除了 `taskENTER_CRITICAL()` 和 `taskEXIT_CRITICAL()` 外，千万不要在其他地方修改控制器的中断使能位或优先级标志。这两个宏维护了一个嵌套深度计数，因此只有当所有的嵌套调用都退出后计数值才会为 0，也才会使能中断。

问题现象：在调度器启动前应用程序崩溃。

如果一个中断会产生上下文切换，则这个中断不能在调度器启动之前使能。这同样适用于那些需要读写队列或信号量的中断。在调度器启动之前，不能进行上下文切换。

还有一些 API 函数不能在调度器启动之前调用。在调用 `vTaskStartScheduler()` 之前，最好限定只使用创建任务、队列和信号量的 API 函数。

问题现象：在调度器挂起时调用 API 函数，导致应用程序崩溃。

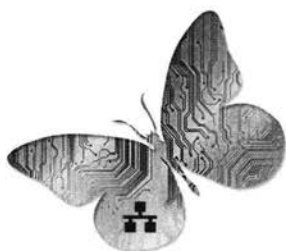
调用 `vTaskSuspendAll()` 使得调度器挂起，而唤醒调度器调用 `xTaskResumeAll()`。

千万不要在调度器挂起时调用其他 API 函数。

问题现象：函数原型 `pxPortInitialiseStack()` 导致编译失败。

每种移植都需要定义一个对应的宏，以把正确的内核头文件加入到工程中。如果编译函数原型 `pxPortInitialiseStack()` 时出错，这种现象基本上可以确定是因为没有正确定义相应的宏。

可以基于相应平台的 Demo 工程建立新的应用程序。这种方式就不用担心没有包含正确的文件，也不必担心没有正确配置编译器选项。



第 8 章

基于 STM32F107 的 FreeRTOS 移植

本章以 FreeRTOS 在 STM32F107 上的移植为主要内容，介绍移植的主要工作内容，以及移植过程中的注意事项。

8.1 概述

FreeRTOS 的实现主要由 list.c、queue.c、croutine.c 和 tasks.c 4 个文件来完成。list.c 是一个链表的实现，主要供给内核调度器使用；queue.c 是一个队列的实现，支持中断环境和信号量控制；croutine.c 和 task.c 是两种任务的组织实现。协程（croutine）是采用各任务共享同一个堆栈，使 RAM 的需求进一步缩小，但也正因为如此，它的使用受到相对严格的限制。而 task 则是传统的实现，任务使用自己的堆栈，支持完全的抢占式调度。

- ❑ FreeRTOS 的主要功能可以归结为以下几点：
- ❑ 优先级调度、相同优先级任务的轮转调度，同时可设成可剥夺内核或不可剥夺内核。
- ❑ 任务可选择是否共享堆栈（co-routines & tasks），并且没有任务数限制。
- ❑ 消息队列、二值信号量、计数信号量和递归互斥体。
- ❑ 时间管理。
- ❑ 内存管理。

与 uC/OS-II 一样，FreeRTOS 在 STM32 上的移植大致由 3 个文件实现，一个 .h 文件定义编译器相关的数据类型和中断处理的宏定义；一个 .c 文件实现任务的堆栈初始化、系统心跳的管理和任务切换的请求；一个 .s 文件实现具体的任务切换。在本次移植中，使用的编译软件为 IAR EWARM 6.3。

8.2 FreeRTOS 移植

8.2.1 portmacro.h 头文件

portmacro.h 头文件主要包括两部分内容。第一部分内容定义了一系列内核代码中用到的数据类型。FreeRTOS 与 uC/OS-II 一样，并不直接使用 char、int 等原生数据类型，而是将其重新定义为一系列以 port 开头的新类型。这样处理的好处在于，在不同架构的处理器移植时，仅需针对处理器的位宽对这些数据类型做相应的调整，而无需繁琐地从头至尾修改源代码。

第二部分内容包含了实现 FreeRTOS 移植所需要定义的函数。包括与架构相关的定义、内核调度、临界区管理、任务优化等。

1. 数据类型定义

定义编译器相关的各种数据类型。在 uC/OS-II 的移植代码中，通常采用 typedef 来定义新的类型，而 FreeRTOS 的作者似乎更喜欢用宏定义 #define。虽然 typedef 和 #define 能够实现相同的功能，但也有细微的区别，typedef 由编译器解释，而非预处理器执行，并且仅限于对数据类型进行定义。尽管如此，typedef 在其受限范围内比 #define 更为灵活。下面是相应的代码片段。

```
#define portCHAR    char
#define portFLOAT   float
#define portDOUBLE  double
#define portLONG    long
#define portSHORT   short
#define portSTACK_TYPE  unsigned portLONG
#define portBASE_TYPE  long
```

2. 架构相关的定义

定义与处理器或控制器架构相关的宏定义。

条件编译中的代码针对的处理器字长为 16 位，工程配置时应在 FreeRTOSConfig.h 中将宏定义改为“#define configUSE_16_BIT_TICKS 1”，若处理器字长为 32 位，则宏定义为“#define configUSE_16_BIT_TICKS 0”。

宏 portSTACK_GROWTH 定义了堆栈的生长方向。对于不同的嵌入式操作系统，堆栈的生长方向定义可能不同，在移植时应予以明确。portSTACK_GROWTH 定义为 1，表示堆栈是正向生长的，定义为 -1，则表示堆栈是逆向生长的。一般来说，堆栈都是逆向生长的。Cortex-M3 的堆栈增长方向为高地址向低地址增长，因此这里定义为 -1。

portTICK_RATE_MS 表示的是 Tick 间间隔多少毫秒，只在应用代码中可能会用到。如使用 vTaskDelay 延时函数可实现任务定时间隔地执行，调用方法如下：

```
vTaskDelay( 250 / portTICK_RATE_MS );
```

vTaskDelay 传递参数中的 250 表示系统使当前任务处于阻塞态，并维持 250ms。

portBYTE_ALIGNMENT 在 uC/OS-II 是不需要的，用在 FreeRTOS 代码中，表示分配任务堆栈空间时实现 SRAM 访问的字节对齐。

```
/* SYSTEM TICK 的长度。如果是 16 位以下的处理器置 1；如果是 32 位处理器，则置 0 */
#if( configUSE_16_BIT_TICKS == 1 )
    /* 16 位处理器及以下处理器 */
    typedef unsigned portSHORT portTickType;
    #define portMAX_DELAY ( portTickType ) 0xffff
#else
    /* 32 位处理器 */
    typedef unsigned portLONG portTickType;
    #define portMAX_DELAY ( portTickType ) 0xffffffff
#endif
```

```

/* Cortex-M3 的堆栈增长方向为高地址向低地址增长 */
#define portSTACK_GROWTH ( -1 )

/* 定义心跳时钟周期, 表示相邻 Tick 间间隔多少毫秒 */
#define portTICK_RATE_MS ((portTickType)1000/configTICK_RATE_HZ)

/* 访问 SRAM 的字节对齐 */
#define portBYTE_ALIGNMENT 8

```

3. 内核调度函数

portYIELD() 等同于 vPortYieldFromISR(), 实现的是任务切换。vPortYieldFromISR() 函数在 PORT.C 文件中实现, 相当于 uC/OS-II 中的 OS_TASK_SW()。

```

/* 声明该函数定义在其他文件中, 实现强制上下文切换, 在任务环境中调用 */
extern void vPortYieldFromISR( void );
#define portYIELD() vPortYieldFromISR()

/* 强制上下文切换, 在中断处理环境中调用 */
#define portEND_SWITCHING_ISR(xSwitchRequired) if(xSwitchRequired)\
vPortYieldFromISR()

```

4. 临界区管理函数

定义临界区的管理函数。vPortSetInterruptMask 和 vPortClearInterruptMask 在 portasm.s 中定义, 实现中断屏蔽位的清除或置位。vPortEnterCritical 和 vPortExitCritical 函数在 port.c 文件中定义, 实现临界区的进入与退出。下面代码中的最后两个宏定义则是用于中断环境的中断屏蔽和开启, 即是否允许中断嵌套。

```

/* 声明以下 4 个函数在其他文件中定义, 实现临界区的进入与退出、中断的允许和关闭 */
extern void vPortEnterCritical( void );
extern void vPortExitCritical( void );
extern void vPortSetInterruptMask( void );
extern void vPortClearInterruptMask( void );

/* 中断允许和关闭 */
#define portDISABLE_INTERRUPTS() vPortSetInterruptMask()
#define portENABLE_INTERRUPTS() vPortClearInterruptMask()

/* 临界区进入和退出 */
#define portENTER_CRITICAL() vPortEnterCritical()
#define portEXIT_CRITICAL() vPortExitCritical()

/* 用于中断环境的中断屏蔽和开启 */
#define portSET_INTERRUPT_MASK_FROM_ISR() 0;vPortSetInterruptMask()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x) \
vPortClearInterruptMask();(void)x

```

5. 任务优化函数

以下两个宏定义是为了利用某些 C 编译器的扩展功能而对任务函数进行更好的优化。

portNOP() 顾名思义就是对空操作定义了个宏。

```
/* 预留扩展函数，用于对任务函数的优化 */
#define portTASK_FUNCTION_PROTO( vFunction, pvParameters ) void \
vFunction( void *pvParameters )
#define portTASK_FUNCTION( vFunction, pvParameters ) void \
vFunction( void *pvParameters )

/* 消耗一个机器周期，用来延时或空操作 */
#define portNOP()
```

8.2.2 port.c 源文件

1. 堆栈初始化

在此文件中进行堆栈的初始化，使堆栈处于预知的确定状态。下面是堆栈初始化的代码实现，其模拟了一个由中断引起的上下文切换导致的堆栈操作序列。

```
portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,
                                         pdTASK_CODE pxCode,
                                         void *pvParameters )
{
    /* 计算存储程序状态寄存器 xPSR 的堆址，用于 MCU 在进入或退出中断时恢复现场 */
    pxTopOfStack--;
    /* 程序状态寄存器的值保存于堆栈中 */
    *pxTopOfStack = portINITIAL_XPSR;

    pxTopOfStack--;
    /* 任务的入口点 */
    *pxTopOfStack = ( portSTACK_TYPE ) pxCode;

    pxTopOfStack--;
    /* LR */
    *pxTopOfStack = 0;

    /* R12, R3, R2 and R1. */
    pxTopOfStack -= 5;

    /* 任务的参数 */
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters;

    /* R11, R10, R9, R8, R7, R6, R5 and R4. */
    pxTopOfStack -= 8;
    return pxTopOfStack;
}
```

2. 启动任务调度

```
portBASE_TYPE xPortStartScheduler( void )
{
    /* 让任务切换中断和心跳中断位于最低优先级，使更高优先级可以抢占 MCU */
```

```

*(portNVIC_SYSPRI2) |= portNVIC_PENDSV_PRI;
*(portNVIC_SYSPRI2) |= portNVIC_SYSTICK_PRI;

/* 启动定时器, 开始产生系统的心跳时钟, 此处中断已被关闭 */
prvSetupTimerInterrupt();

/* 初始化临界区嵌套的个数, 准备启动第一个任务 */
uxCriticalNesting = 0;

/* 启动第一个任务 */
vPortStartFirstTask();

/* 执行到 vPortStartFirstTask 函数, 内核已经开始正常调度 */
return 0;
}

```

3. 主动释放 MCU 使用权

通过设置中断控制及状态寄存器触发 PendSV 系统服务中断, 请求一次上下文切换, 中断到来时由汇编函数 xPortPendSVHandler() 处理。

```

void vPortYieldFromISR( void )
{
    /* 通过设置中断控制及状态寄存器, 请求一次上下文切换 */
    *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
}

```

4. 临界区进出

进入临界区时, 首先关闭中断, 当退出所有嵌套的临界区后再使能中断。下面这段代码中进出临界区是通过关中断和开中断操作来实现的, 相当于 uC/OS-II 中 OS_CRITICAL_METHOD == 1 的情况。不过, 通过全局变量 uxCriticalNesting 来记录临界区的嵌套层数以此来实现临界区的嵌套操作。虽然 uxCriticalNesting 是全局变量, 但是后面可以看到在任务切换时会将 uxCriticalNesting 的值保存到当期任务的堆栈中, 完成任务切换后再从新的任务的堆栈中取出 uxCriticalNesting 的值。每个任务通过这种方式维护各自的临界区嵌套次数 uxCriticalNesting。实际上, FreeRTOS 和 uC/OS-II 中临界区的概念是相同的, 因此在 uC/OS-II 中可用的临界区保护的方法都可以应用到 FreeRTOS 中。

```

/* 进入临界区 */
void vPortEnterCritical( void )
{
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;
}

/* 退出临界区 */
void vPortExitCritical( void )
{
    uxCriticalNesting--;
}

```

```

        if( uxCriticalNesting == 0 )
        {
            portENABLE_INTERRUPTS();
        }
    }
}

```

5. 心跳时钟处理函数

心跳时钟处理函数在系统心跳中断发生时开始执行，函数首先对调度方式进行判断。当调度方式配置为抢占式调度时，若存在需要调度的任务，则通过设置中断控制及状态寄存器进行强制上下文切换。当调度方式为非抢占式调度时，则屏蔽中断响应，继续执行原任务。

```

void xPortSysTickHandler( void )
{
    unsigned portLONG ulDummy;

    /* 在配置为抢占式调度时，若有需要调度的任务则通过设置中断控制及状态寄存器进行强制上下文切换 */
    #if configUSE_PREEMPTION == 1
        *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
    #endif

    /* 若配置为非抢占式调度，则屏蔽中断响应 */
    ulDummy = portSET_INTERRUPT_MASK_FROM_ISR();
    {
        /* 通过task.c的心跳时钟处理函数vTaskIncrementTick()进行时钟计数和延时任务的处理 */
        vTaskIncrementTick();
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( ulDummy );
}

```

函数prvSetupTimerInterrupt用于配置SysTick中断频率，以产生系统的心跳时钟。

```

void prvSetupTimerInterrupt( void )
{
    *(portNVIC_SYSTICK_LOAD) = ( configCPU_CLOCK_HZ /
                                   configTICK_RATE_HZ ) - 1UL;
    *(portNVIC_SYSTICK_CTRL) = portNVIC_SYSTICK_CLK |
                                portNVIC_SYSTICK_INT |
                                portNVIC_SYSTICK_ENABLE;
}

```

8.2.3 portasm.s 汇编源文件

1. 头文件及函数预定义

```

/* 包含对FreeRTOS内核配置的头文件 */
#include "FreeRTOSConfig.h"

/* 检查内核优先级，若内核优先级不为最低，将其置为最低 */
#ifndef configKERNEL_INTERRUPT_PRIORITY
    #define configKERNEL_INTERRUPT_PRIORITY 0

```

```
#endif
/* 表示开始一个新的代码段，并以 4 字节对齐 */
RSEG CODE:CODE(2)
/* 该指令表明以下代码为 Thumb 指令 */
thumb

/* 外部变量或函数声明 */
EXTERN vPortYieldFromISR
EXTERN pxCurrentTCB
EXTERN vTaskSwitchContext

/* 表明该源文件中实现的函数为公共函数，可供外部调用 */
PUBLIC vSetMSP
PUBLIC xPortPendSVHandler
PUBLIC vPortSetInterruptMask
PUBLIC vPortClearInterruptMask
PUBLIC vPortSVCHandler
PUBLIC vPortStartFirstTask
```

上述代码中有一条伪指令 RSEG，用于指示开始一个新的代码段。汇编器为每个代码段都保留一个单独的位置计数器，使得在任何时候，代码段和模式的切换成为可能，同时不用保存当前代码位置计数器。该伪指令的格式如下：

```
RSEG section [:type] [:flag] [{align}]
```

其中：

- section：段名称，无需明确标识时可省略。
- type：存储器类型，可以是 CODE、CONST 或 DATA。
- flag：可以是 ROOT 或 NOROOT。ROOT（默认值）表示该段不能被丢弃。NOROOT 表示如果在该段中没有任何标识被索引，那么链接器会将其丢弃。正常情况下，除了启动代码和中断向量外的所有的段都应设置该标识。也可以是 REORDER 或 NOREORDER。NOREORDER（默认值）表示在当前段中开始一个新的分段，并取一特定的名字，如果和以往的段不重名，则表示开始一个新的段。REORDER 表示以特定的名字开始一个新段。
- align：地址的对齐方式，假设指令中的值为 x，那么地址的对齐方式就是 2x 字节对齐，x 允许的范围是 0~8，除了代码段默认是为 1 外，其余默认值均为 0。

源文件中的“thumb”表示后续代码为 Thumb 指令，需要注意的是，不同的汇编器对指令的解释有所差异。如 IASMARM 汇编器将其解释为偶地址对齐，而 ARMASM 却将其解释为奇地址对齐。

此外，该代码还对用到的外部变量或函数进行了声明。并且将一些函数声明为公共函数，供外部调用。

2. 请求切换任务

xPortPendSVHandler 函数首先将 R4~R11 压入堆栈，并将新的栈顶地址保存到 pxCurrentTCB-

>pxTopOfStack, 用以保护当前任务的上下文。随后, 通过调用vTaskSwitchContext()函数更新pxCurrentTCB指针, 指向当前优先级最高的任务。最后恢复之前任务的上下文环境。

```
xPortPendSVHandler:

mrs r0, psp                ; 保存当前任务的上下文到其任务控制块
ldr r3, = pxCurrentTCB     ; 获取当前任务的任务控制块指针
ldr r2, [r3]

stmdb r0!, {r4-r11}        ; 保存 R4~R11 到该任务的堆栈
str r0, [r2]               ; 将最后的堆栈指针保存到任务控制块的 pxTopOfStack

stmdb sp!, {r3, r14}
mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY ; 关闭中断
msr basepri, r0
bl vTaskSwitchContext      ; 切换任务的上下文, pxCurrentTCB 已指向新的任务
mov r0, #0
msr basepri, r0
ldmia sp!, {r3, r14}

ldr r1, [r3]               ; 恢复新任务的上下文到各寄存器
ldr r0, [r1]               ; pxCurrentTCB 中的第一个条目就是栈顶对应的任务
ldmia r0!, {r4-r11}        ; 弹出寄存器, 恢复现场
msr psp, r0
bx r14
```

任务切换的示意图如图 8.1 所示。

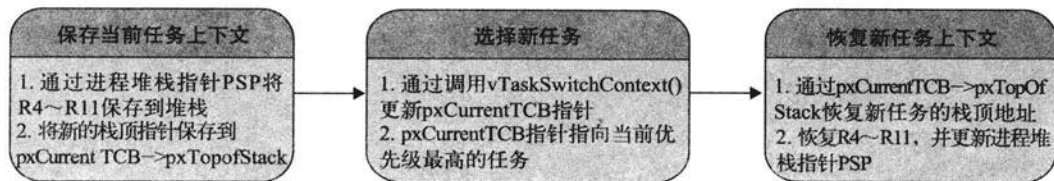


图 8.1 任务切换示意图

3. 中断允许和关闭的实现

通过 BASEPRI 屏蔽相应优先级的中断源。以下两个函数比较简单, 在此不再赘述。

```
vPortSetInterruptMask:
push { r0 }
mov R0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
msr BASEPRI, R0
pop { R0 }

bx r14
```

```
vPortClearInterruptMask:
PUSH { r0 }
MOV R0, #0
```

```
MSR BASEPRI, R0
POP { R0 }
bx r14
```

4. 直接切换任务

用于 vPortStartFirstTask 第一次启动任务时初始化堆栈和各寄存器，与堆栈的初始化类似，在此不再赘述。

```
vPortSVCHandler:
    ldr r3, =pxCurrentTCB
    ldr r1, [r3]
    ldr r0, [r1]
    ldmia r0!, {r4-r11}
    msp psp, r0
    mov r0, #0
    msp basepri, r0
    orr r14, r14, #13
    bx r14
```

5. 启动第一个任务的汇编实现

将堆栈地址保存到主堆栈指针 msp 中，触发 SVC 软中断，由 vPortSVCHandler() 完成第一个任务的具体切换工作。

```
vPortStartFirstTask:
    /* 通过中断向量表定位堆栈的地址 */
    ldr r0, =0xE000ED08 ; 中断向量表偏移量寄存器
    ldr r0, [r0]
    ldr r0, [r0]

    /* 将主堆栈指针 msp 置入堆栈起始地址 */
    msp msp, r0

    /* 调用 SVC 开始执行第一个任务 */
    cpsie i
    svc 0
```

FreeRTOS 内核调度器的启动流程如图 8.2 所示。

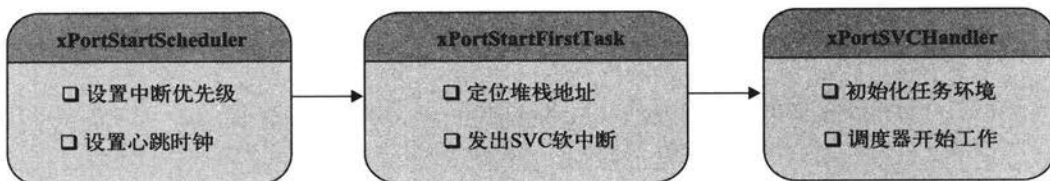


图 8.2 内核调度器的启动流程

以上 3 个文件实现了 FreeRTOS 内核调度所需的底层接口，相关代码十分精简。

8.2.4 其他问题

需要注意的是, ST 的官方驱动库为读者提供了一个组织规范合理的工程模板。但该模板中的 startup 文件与 FreeRTOS 的接口函数应保持一致, 以 STM32F107VCT6 为例, 读者需要更新官方提供的 startup 文件 (startup_stm32f10x_cl.s) 中的有关函数的定义及中断向量表, 需要更新的代码已用粗体标出。

```

MODULE ?cstartup

;; Forward declaration of sections.
SECTION CSTACK:DATA:NOROOT(3)

SECTION .intvec:CODE:NOROOT(2)

EXTERN __iar_program_start
EXTERN SystemInit
IMPORT vPortSVCHandler
IMPORT xPortSysTickHandler
IMPORT xPortPendSVHandler
PUBLIC __vector_table

DATA
__vector_table
DCD sfe(CSTACK)
DCD Reset_Handler ; Reset Handler
DCD NMI_Handler ; NMI Handler
DCD HardFault_Handler ; Hard Fault Handler
DCD MemManage_Handler ; MPU Fault Handler
DCD BusFault_Handler ; Bus Fault Handler
DCD UsageFault_Handler ; Usage Fault Handler
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD vPortSVCHandler ; SVCall Handler
DCD DebugMon_Handler ; Debug Monitor Handler
DCD 0 ; Reserved
DCD xPortPendSVHandler ; PendSV Handler
DCD xPortSysTickHandler ; SysTick Handler

```

8.3 创建测试任务

为了检验移植的情况, 本节将建立一个简单的实例程序, 测试 FreeRTOS 在 STM32F107 开发板上的运行状况。

首先, 在工程目录中创建一个 Group, 命名为 FreeRTOS, 在添加 FreeRTOS 的 task.c、queue.c、list.c 和 heap_2.c 源文件的同时, 将移植好的 3 个文件添加到 FreeRTOS 目录中。工程实例目录如图 8.3 所示。

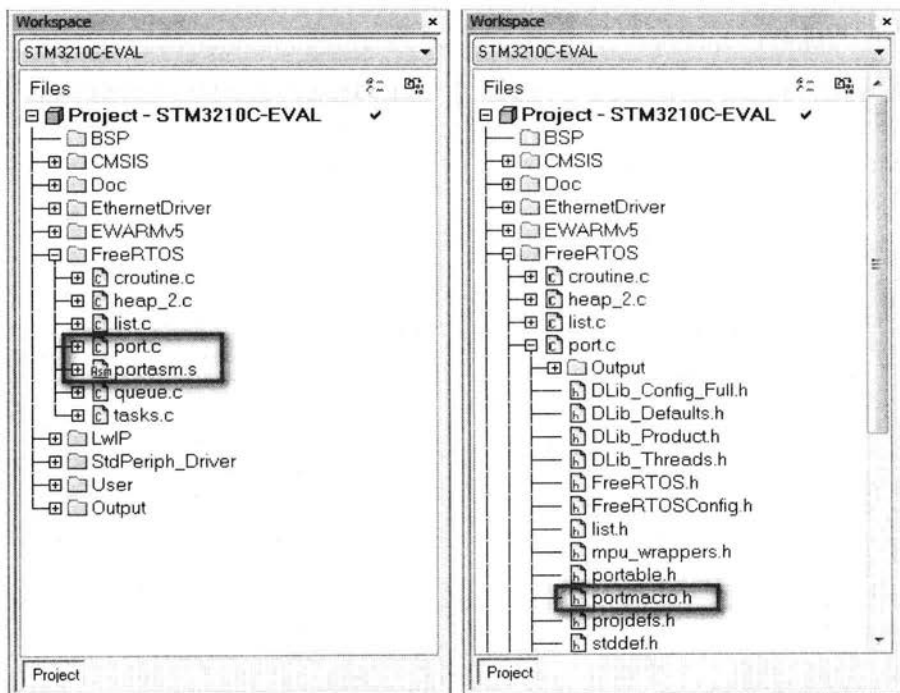


图 8.3 测试工程实例目录

随后，设置好源程序头文件的包含路径，设置方法可参考 3.3 节中的方法。最后，创建如下 main.c 源文件，并添加入 User 目录。

```

/* Standard includes -----*/
#include <stdio.h>

/* Drivers includes -----*/
#include "stm32f10x.h"

/* FreeRTOS includes -----*/
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

/* Variable define -----*/
const char *pcTextForTask1 = "Task 1 is running\r\n";
const char *pcTextForTask2 = "Task 2 is running\t\n";

/* main 函数 */
int main( void )
{
    /* 自此，控制器时钟已通过 SystemInit() 函数完成初始化，该函数在控制器启动时从 startup 文件中调用 */

```

```

/* 任务1的优先级为1 */
xTaskCreate( vTaskFunction,
            "Task 1",
            1000,
            (void*)pcTextForTask1,
            1,
            NULL );

/* 任务2的优先级为2 */
xTaskCreate( vTaskFunction,
            "Task 2",
            1000,
            (void*)pcTextForTask2,
            2,
            NULL );

/* 启动调度器, 开始执行 */
vTaskStartScheduler();

/* 无限循环 */
for( ;; );
}

```

以下是任务函数的代码实现：

```

void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* 要打印的字符通过传参传递, 并将其强制转换为字符指针 */
    pcTaskName = ( char * ) pvParameters;
    /* 每个任务都以一个无限循环实现 */
    for( ;; )
    {
        /* 打印任务的名字 */
        printf( "%s", pcTaskName );

        /* 延时 250ms, 使任务处于阻塞态 */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}

```

最后运行测试实例, 如图 8.4 所示。

大家或许会诧异, 创建一个任务竟是如此简单。事实上, 这也是嵌入式操作系统的优势所在, 它使得代码的架构更清晰, 维护也更方便。通过上述实例, 大家可以了解到 FreeRTOS 的任务创建与 uC/OS-II 差异不大, 主要涉及任务函数、堆栈大小和任务的优先级。

至此, FreeRTOS 在 STM32 上的移植与测试已完成。与 uC/OS-II 相比, FreeRTOS 精简的实现不仅适合用来学习实时操作系统的工作原理, 也方便读者对其进行深入剖析和工程应用。

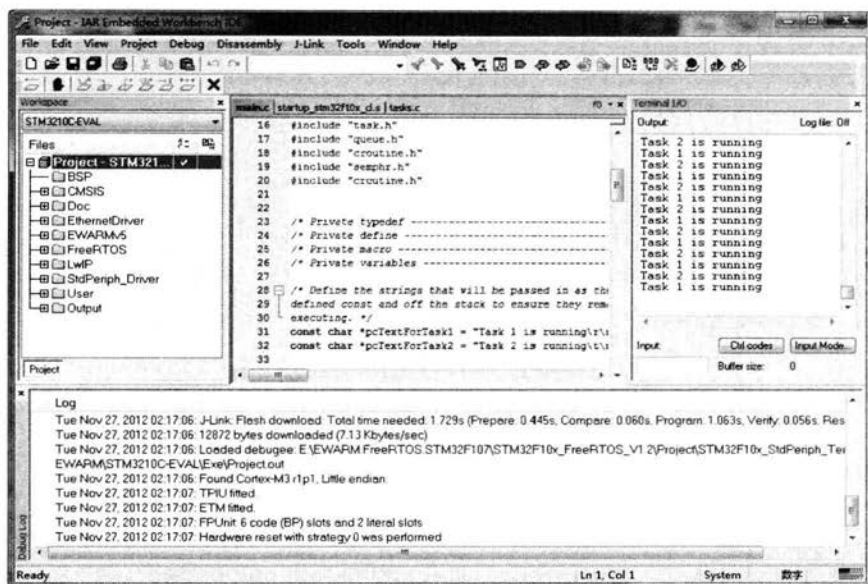
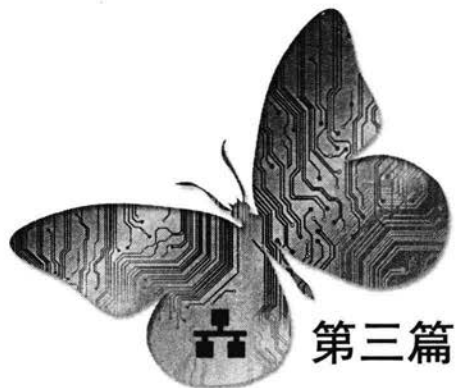


图 8.4 测试实例运行

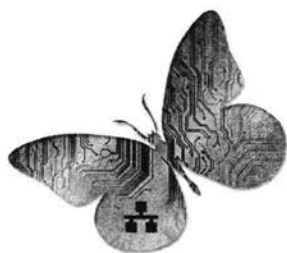


第三篇

LwIP篇

本篇结合 TCP/IP 原理介绍 LwIP 开源轻量级协议栈的基本原理及常见应用模式。

- 第9章 TCP/IP协议栈介绍
- 第10章 LwIP轻量级TCP/IP协议栈
- 第11章 基于STM32F107的LwIP移植



第 9 章

TCP/IP 协议栈介绍

本章以 TCP/IP 协议栈的四层协议系统为出发点，简要介绍 TCP/IP 协议栈的基本原理，帮助读者理解 LwIP 开源轻量级协议栈的基本原理，主要包括 TCP/IP 协议的分层、以太网报文的封装和分用过程、地址解析等内容。

9.1 引言

TCP/IP 协议簇是计算机网络通信协议的一种，它是 Internet（国际互连网络）的基础，也是 Internet 最基本的协议。该协议簇定义了各类计算机及网络互联设备如何接入 Internet，以及数据在它们之间传输的标准。TCP/IP 起源于 20 世纪 60 年代末美国政府资助的一个分组交换网络研究 ARPA，并在 20 世纪 90 年代中期获得蓬勃发展，成为计算机之间最常用的组网形式。

本章将介绍 TCP/IP 协议簇的基本原理，为读者理解 LwIP 轻量级 TCP/IP 协议栈提供充分的背景知识。

9.2 网络分层

计算机网络通信协议通常按不同层次进行开发，每一层负责不同的通信功能。例如，ISO 国际标准组织所定义的开放系统互连七层模型，即七层模型，是一组不同层次上的多个协议的组合。而 TCP/IP 协议与 OSI 七层模型并不完全匹配，一般认为是一个四层协议系统。

9.2.1 OSI 七层参考模型

OSI（Open System Interconnection，开放系统互连）七层网络模型称为开放式系统互联参考模型，它不是一个实际的物理模型，而是一个规范了网络协议的逻辑模型，它从逻辑上把网络分为 7 层，如图 9.1 所示。

该模型的第一层是物理层，它是 OSI 七层模型的基础，直接面向实际承担数据传输的物理媒体及互联设备。OSI 将物理层定义

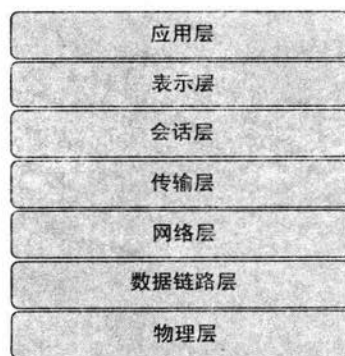


图 9.1 OSI 七层参考模型

为：物理层提供机械的、电气的、功能的和规程的特性，目的是启动、维护和关闭数据链路实体之间进行比特传输的物理连接。在物理层上，相关的物理媒体和互联设备包括：双绞线、电缆、光纤、无线信道和数据终端设备等。它主要完成如下功能：

- 为数据端设备提供传送数据的通路，数据通路可以是一个物理媒体，也可以由多个物理媒体连接而成。一次完整的数据传输包括激活物理连接、传送数据、终止物理连接等。
- 传输数据。物理层要形成适合数据传输需要的实体，实现数据的可靠传输。

在物理层上，传输数据的单位是比特。

数据链路层是在物理层提供比特流服务的基础上，建立相邻节点之间的数据链路，通过差错检测、差错控制、流量控制等方法，实现数据帧在信道上的无差错透明传输。数据链路层传输的基本单位是帧。它主要完成如下功能：

- 链路管理：进行数据链路的建立、维护和拆除。在链路两端的节点进行通信前，必须首先确认对方已处于就绪状态，并交换一些必要的信息以对帧序列进行初始化，然后建立链路连接。在传输过程中，还要能维持这种连接，传输完毕后要拆除该连接。
- 帧同步：为了使传输中发生差错后只将有错的有限数据进行重发，数据链路层将比特流封装成帧进行传送。每个帧除了要传送的数据外，还包括校验码以使接收方能发现传输中的差错。帧的组织结构必须设计成使接收方能够明确地从物理层收到的比特流中对其进行界定。
- 流量控制：为防止双方速度不匹配或接收方没有足够的接收缓存而导致数据拥塞或溢出，数据链路层必须采取一定的措施使通信网络中的链路或节点上的信息流量不超过某一阈值。
- 差错控制与恢复：为了在不可靠的物理媒体上实现数据的可靠传输，数据链路层必须具备差错控制的功能，使得差错被控制在所能允许的尽可能小的范围内。差错控制通常可以分为前向纠错法、反馈重传法和混合法三种类型。

数据链路层所规定的协议主要包括：SDLC、HDLC、PPP、STP、帧中继等。

网络层是第三层，位于数据链路层和传输层之间，根据 OSI 的定义，网络层为一个网络连接的两个传送实体间交换网络服务数据单元提供功能和规程方法，它使传送实体独立于路由选择和交换的方式。在网络层上，传输数据的单位是数据包，它位于通信子网的最高层，并处于处理端到端传输的最底层。网络层主要完成如下功能：

- 向传输层提供无连接的和面向连接的服务。
- 路由选择：路由算法是网络层软件的一个部分。它负责将一个分组报文通过一个最佳路径，传输到目的端。路由算法通常可以分为非自适应的算法和自适应算法。
- 拥塞控制：与数据链路层的流量控制不同，拥塞控制需要确保通信子网能够承载用户提交的通信量，这是一个全局性问题，涉及主机、路由器等很多因素。

传输层是整个协议层次的核心所在，是 OSI 七层参考模型中最重要的一层。它的设计目标是在源机器和目标机器之间提供可靠的、高效的数据传输功能，并完全独立于当前所使用的物理网

络。它主要完成如下功能：

- ❑ 连接管理：负责传输连接的建立、维护与释放。
- ❑ 端到端的流量控制：传输层在发送本层数据报文时，还要确保数据的完整性，流量控制是完成这项任务的方法之一。流量控制避免了接收主机缓冲溢出的问题，溢出会造成数据丢失。
- ❑ 端到端的差错检测与恢复。
- ❑ 提供用户要求的服务质量：用户在通信时会要求特定的网络服务质量，例如，高吞吐量、低延迟、低费用和高可靠性服务等。传输层可根据需要提供相应的网络服务。
- ❑ 提供端到端的可靠通信。

会话层对传输的报文提供同步管理服务。它不参与具体的传输，但提供包括访问验证和会话管理在内的通信机制。会话层用于建立通信链接，保持会话过程中通信链接的通畅，同步和管理两个节点之间的通话。

表示层提供多种功能用于应用层数据编码和转化，以确保以一个系统应用层发送的信息可以被另一个系统应用层识别。表示层的主要功能为：

- ❑ 语法转换：将抽象语法转换成传送语法，并在对方实现相反的转变。
- ❑ 语法协商：根据应用层的要求协商选用合适的上下文，即确定传送语法并传送。
- ❑ 连接管理：包括利用会话层服务建立表示连接，管理在这个连接之上的数据传输和同步控制，以及正常地或异常地终止这个连接。

应用层是 OSI 参考模型的最高层，是直接为应用进程提供服务的，是用户与网络之间的接口。它确定了进程间通信的性质以满足用户的需要，并且提供了网络与用户应用软件之间的接口服务。它的主要功能是为用户的应用程序提供接口服务，完成文件传输、文件管理以及电子邮件处理等功能。

9.2.2 TCP/IP 分层

TCP/IP 模型与 OSI 参考模型并不完全相同，它由 4 个层次组成，分别是链路层、网络层、传输层和应用层，如图 9.2 所示。

链路层，也称作网络接口层或数据链路层，它通过网络设备驱动程序，接收 IP 数据报，并把数据报通过选定的网络接口发送出去。同时，链路层还定义了传输的物理媒介的各种特性：机械特性、电子特性、功能特性、规程特性。

网络层，又称 IP 层，主要处理计算机之间的通信问题。它主要完成以下功能：

- ❑ 处理来自传输层的分组发送请求。它把分组封装到 IP 数据报中，填入数据报的首部，使用路由算法选择去往信宿机的最佳路径，然后将数据报发送至适当的网络接口。
- ❑ 处理接收到的数据报，检验其正确性，并使用路由选择算法决定是在本地进行处理，还是继续向前发送。如果数据报的目标机处于本机所在的网络，该层程序就把数据报的报

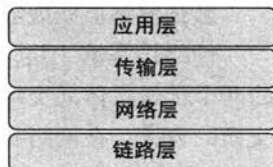


图 9.2 TCP/IP 参考模型

头剥去，再选择适当的传输层协议软件来处理这个分组报文。否则，继续向前发送该数据报。

□ 处理路由选择、差错检测与恢复、流量控制、网络拥塞等问题。

传输层，它的基本任务是提供应用程序之间的通信服务，这种通信又称为端到端的通信。传输层既要系统地管理数据信息的流动，还要提供可靠的传输服务，以确保数据准确而有序地到达目的地。为了达到这个目的，传输层的协议软件需要进行协商，让接收方回送确认信息，若未收到确认信息，发送方将重发丢失的报文。

应用层是 TCP/IP 模型的最高层，用户通过调用应用程序，如电子邮件、文件传输访问、远程登录等来访问 TCP/IP 互连网络，以享受网络提供的各种服务。应用程序负责发送和接收数据，每个应用程序可以选择所需要的传输服务类型，并把数据按照传输层的要求组织好，再向下层传送。

9.2.3 TCP/IP 协议簇的协议

TCP/IP 协议簇是由众多协议组成的，图 9.3 描述了 TCP/IP 协议簇中的主要协议，同时也描述了 TCP/IP 模型与 OSI 七层参考模型的基本对应关系。

应用层	网络管理 协议 SNMP	电子邮件 协议 POP 3、 SMTP	文件传输 FTP	网络文件服 务协议 NFS 协议	应用层
表示层					
会话层					
传输层	TCP 协议		UDP 协议		传输层
网络层	IP 协议	ICMP 协议	ARP 协议	RARP 协议	网络层
数据链路层	IEEE 802.3 协议	IEEE 802.5 协议	PPP /SLIP 协议	FDDI 协议	数据链路层
物理层					

图 9.3 TCP/IP 协议簇的协议

数据链路层是 TCP/IP 与各种物理通信网络的数据接口层，该层包含了如 IEEE802.3、点对点串行通信协议 PPP/SLIP 协议等。

网络层含有四个重要的协议：互联网协议 IP、互联网控制报文协议 ICMP、地址转换协议 ARP 和反地址转换协议 RARP。IP 协议是 TCP/IP 协议簇中最为核心的协议，TCP、UDP、ICMP、ARP 等都以 IP 数据报格式传输。IP 协议提供不可靠的和无连接的数据报传送服务。

ARP 和 RARP 协议用于互连网络地址（即 IP 地址）与物理地址之间的相互转换。ICMP 协议的主要任务是为了使互联网能够报告差错、网路拥塞和提供有关意外情况的信息。

TCP 协议和 UDP 协议是传输层的两个协议。其中，TCP 协议是在 IP 协议的基础上，提供端到端的面向连接的可靠的数据传输服务，UDP 协议则提供端到端的不可靠的、无连接的数据传输服务。

在 TCP/IP 的应用层，则存在着各种各样的应用层协议，如文件传输协议 FTP、电子邮件协议 POP3、网络管理协议等。

接下来就几个应用较为广泛的协议进行简单介绍。

9.3 IP 协议

IP 协议是 TCP/IP 协议簇中最为重要的协议。它的主要功能包括：将 TCP、UDP、ICMP、IGMP 等数据封装到 IP 数据报中；通过路由选择，选择合适的目的路径，将 IP 数据报传送到目的地。

IP 协议提供不可靠的和无连接的服务。所谓不可靠的服务是指它不能保证 IP 数据报一定能够正确无误地到达目的地。而无连接的服务是指发送数据方有可能会在接收方未做好接收数据准备时，就发送数据。换言之，就是 IP 协议并不维护任何关于后续 IP 数据报的状态协议。由于 IP 协议只提供不可靠和无连接的服务，所以差错检测和流量控制就需要由上层其他协议来完成，这样可以保证 TCP/IP 协议的高效率。不同的上层协议将自己决定是否面向连接的，或是可靠的。如 TCP 协议是可靠的、面向连接的协议，而 UDP 则不是。

IP 地址是 IP 协议中一个很重要的组成部分，根据 IPv4 协议，它是给每个连接在 Internet 上的主机分配的一个 32 位的逻辑地址。在 Internet 上，每个网络和每一台计算机都被唯一分配一个 IP 地址，这个 IP 地址在整个网络中是唯一的。IP 地址由两部分组成，一部分是网络地址，另一部分是主机地址。同一个物理网络上的所有主机都使用同一个网络地址，一个主机有一个主机地址与其对应。目前，IP 地址可以分为 A、B、C、D 和 E 五种类型，以适应不同容量的网络。图 9.4 描述了各类 IP 的构成规则。

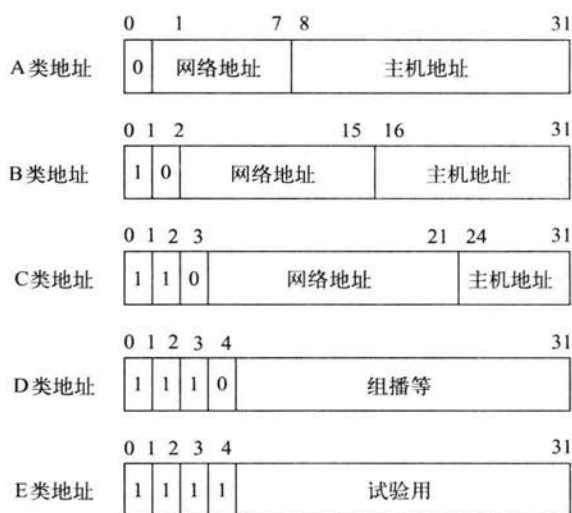


图 9.4 IP 地址构成规则

A 类 IP 地址中，网络地址占用了第一个字节，且该字节的最高位为“0”，余下的 7 位是真正的网络地址，因此 A 类地址共可以支持 $2^7 - 2$ 个网络。IP 地址的剩余 3 个字节是主机地址，因此共有 $(2^{24} - 2)$ 个主机地址。

B 类 IP 地址中，网络地址占用了前面 2 个字节，并使用最高两位“10”来标志是 B 类 IP 地址，余下的 14 位是真正的网络地址，因此 B 类地址共可以支持 $(2^{14} - 2)$ 个网络。剩下的两个字

节作为主机地址，因此共有 $(2^{16}-2)$ 个主机地址。

C 类 IP 地址是 Internet 使用最多的一类地址。它的网络地址占用了 3 个字节，且最高位是“110”，余下的 21 位是真正的网络地址，因此 C 类地址共可以支持 $(2^{21}-2)$ 个网络。C 类地址含有 254 (2^8-2) 个主机地址。

D 类 IP 地址通常用于组播传输数据，它没有网络地址与主机地址之分，其 IP 地址范围从 224.0.0.0 到 239.255.255.255。

E 类 IP 地址与 D 类 IP 地址相似，也不分网络地址与主机地址，它是为搜索、实验和开发而保留的。它的 IP 地址范围是 240.0.0.1 到 255.255.255.254。

子网掩码是 IP 协议中另一个较为重要的概念，它不能单独存在，必须结合 IP 地址一起使用。子网掩码用于指明一个 IP 地址的哪些位标识的是网络地址，哪些位是主机地址。子网掩码中，对应网络地址的位为“1”，对应主机地址的位为“0”。例如，C 类 IP 地址使用前 3 个字节作为网络地址，最后一个字节作为主机地址，因此 C 类网络的默认子网掩码是 255.255.255.0。

9.4 ARP 协议与 RARP 协议

ARP 协议与 RARP 协议是网络层的一个重要协议。ARP (Address Resolution Protocol) 协议的全称是地址解析协议，它是为了建立 IP 地址与物理地址之间的映射关系而设计的。ARP 协议用于将 IP 地址转化为物理地址，RARP 协议则是将物理地址转化为 IP 地址。

在网络中，一方面，每个物理通信设备都有唯一的 48 位物理地址，另一方面，为了屏蔽物理层协议及物理地址的差异，IP 协议使用 IP 地址进行数据传输，因此在数据传输过程中，必须进行 IP 地址到物理地址的相互转换。

ARP 协议进行 IP 地址到物理地址的转换过程为：当 A 网络设备想要与其他网络设备 B 进行通信时，首先 A 设备将查询自己的 ARP 高速缓存，如果缓存中存在 B 设备的 IP 地址，则使用该 IP 地址对应的物理 MAC 地址，直接将数据报发送给 B；若缓存中不存在 B 设备的 IP 地址，那么 A 将以广播方式发送一个 ARP 请求包，B 设备收到该广播包后，会发送一个 ARP 应答包，应答包中包含有 B 设备的物理地址，A 设备收到该应答包后，将把 B 设备的物理地址与 IP 地址的组合添加到 ARP 高速缓存中。

图 9.5 描述了一个 ARP 数据报的基本组成。



图 9.5 ARP 请求或应答数据报格式

在 ARP 数据报中，相关字段的解释如下：

硬件类型：表示硬件地址的类型，0x0001 表示以太网。

协议类型：表示要映射的协议地址类型，0x0800 表示 IP 地址。

硬件地址长度：表示物理 MAC 地址的长度（以字节个数表示），值为 6。

协议地址长度：表示协议地址的长度（以字节个数表示），值为 4。

操作类型：共有 4 种操作类型，ARP 请求（值为 0x01）、ARP 应答（值为 0x02）、RARP 请求（值为 0x03）和 RARP 应答（值为 0x04）。

补齐字节：有的时候，有些物理设备要求发送的报文长度一定要大于某个阈值，因此通常可以采用补 0 以满足此要求。

一个典型的 ARP 请求报文描述如表 9.1 所示。

表 9.1 ARP 请求报文

字段长度（字节）	字段说明	字段值
6	目的 MAC 地址	FF : FF : FF : FF : FF : FF
6	源 MAC 地址	00 : 21 : 86 : FF : FE : 00
2	帧类型	0x0806
2	硬件类型	0x0001
2	协议类型	0x0800
1	硬件地址长度	6
1	协议地址长度	4
2	操作类型	0x0001
6	发送端 MAC 地址	00 : 21 : 86 : FF : FE : 00
4	发送端 IP 地址	192 : 168 : 0 : 100
6	目的端 MAC 地址	00 : 00 : 00 : 00 : 00 : 00
4	目的端 IP 地址	192 : 168 : 0 : 101
18	补齐字节	0 x 00
4	CRC 检验	

9.5 ICMP

ICMP（Internet Control Message Protocol）协议称为网际控制报文协议，是网络层一个比较重要的协议，常用的 Ping 工具就是使用 ICMP 协议实现的。ICMP 协议是为了实现差错控制而实现的，这是因为 IP 协议是不可靠的传输协议，它不能实现差错控制，因此需要一种在发生如通信线路故障、IP 报文传输错序、重复等情况时能够报告差错的机制。

ICMP 协议使用 IP 报文传输差错及控制报文。当主机需要发送 ICMP 报文时，它会被封装到 IP 报文的数据区中。图 9.6 说明了 ICMP 报文的封装格式。

总的说来，ICMP 可以分为三大类，分别是差错报文、控制报文、请求 / 应答报文。其中，差错报文又可以分为超时报文、目的端不可达报文、参数出错报文。控制报文则可以分为源抑制报文和重定向报文两种类型。请求 / 应答报文有三种类型，分别是回应请求 / 应答报文、时间戳请求 / 应答报文、地址掩码请求 / 应答报文。尽管 ICMP 报文种类较多，但它们都以相同的 ICMP 报文首部开始。ICMP 报文首部是一个 32 位的数据段，其格式如图 9.7 所示。

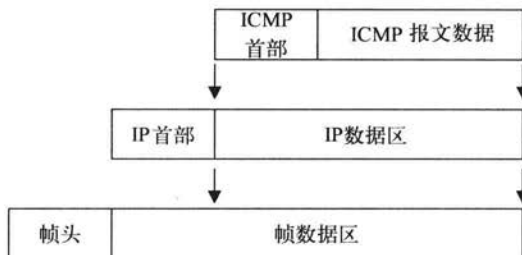


图 9.6 ICMP 报文的封装格式

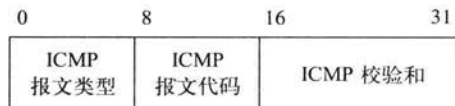


图 9.7 ICMP 报文首部格式

由图 9.7 可以看出，ICMP 报文首部以 8 位的 ICMP 报文类型字段开始，它标志了报文的类别。而第二个字节报文代码则进一步描述了报文的信息，如传输目的端不可达的原因、报文重定向的原因等。ICMP 校验和的校验机制与 IP 校验机制相同，但该校验和只对 ICMP 报文的数据区进行校验。

9.6 TCP 协议

TCP (Transmission Control Protocol) 传输控制协议，是位于传输层的重要协议之一。它在 IP 协议提供的不可靠的数据服务的基础上，为应用层的协议提供了一个可靠的、面向连接的数据传输服务。

传输层的 TCP 协议是如何做到保证传输的可靠性呢？它主要是通过确认 / 重传机制、流量控制、拥塞控制、差错检测与控制实现的。确认 / 重传机制是 TCP 协议能够做到保证可靠性的一个重要原因。它的基本原理是：发送节点发送一个数据报后，即启动了一个定时器，接收方需在规定时间内，向发送方发送一个确认报文，若发送方的定时器超时了，即没有收到接收方的确认报文，发送方就会重新发送该数据报。TCP 协议通过使用“滑动窗口”机制实现了拥塞控制和流量控制。

TCP 协议通过“三次握手”机制实现了面向连接的数据传输。TCP 连接的建立与关闭都需要通过“三次握手”机制来完成。图 9.8 描述了“三次握手”机制。

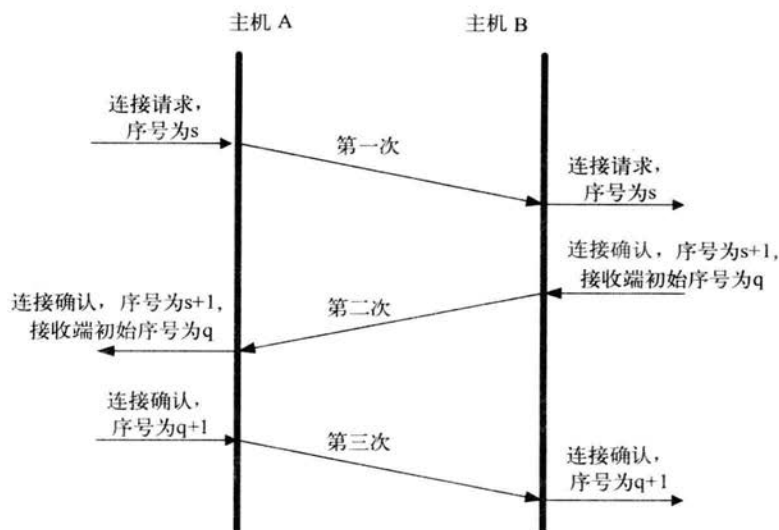


图 9.8 “三次握手”机制

- ❑ “第一次握手”：主机 A 发送一个同步（SYN）标志位 1 的 TCP 数据包，且该数据包指明了接收端的接收端口等信息，同时将该数据包的发送序号标志为 s。
- ❑ “第二次握手”：主机 B 收到主机 A 发送的序号为 s 的连接请求报文后，将发送一个序号为 s+1 的同步报文作为应答，同时告知主机 A，主机 B 的初始序号为 q。
- ❑ “第三次握手”：主机 A 收到主机 B 的连接确认报文后，将序号加 1，即变为 q+1，发送对主机 B 的接收确认报文，主机 B 收到该确认报文后，连接建立，“三次握手”结束。

TCP 报文同 ICMP 报文一样，将封装在 IP 报文中进行传输。它的基本传输单元称为“段”（segment），也称为 TCP 报文段。图 9.9 描述了 TCP 报文段的基本数据格式。

0		16		31	
源端口			目的端口		
32 位序号					
32 位确认序号					
4 位 首部长度	6 位保留	标志位	16 位窗口大小		
校验和			紧急指针		
选项					
数据区					

图 9.9 TCP 报文段数据格式

其中，源端口和目的端口各占 2 个字节，端口的含义是指发送和接收数据的进程。TCP/IP 协议规定了 0~1023 端口的用途，如端口 80 表示超文本传输协议，端口 69 为纯文本传输协议。一个 IP 地址和一个端口号又构成了通常所说的套接字（socket）。

序号字段占用了 4 个字节，表明了本段数据在发送端数据流中的位置。在一个 TCP 连接中，所传输的 TCP 报文可以被视为连续的数据流，TCP 为该数据流中的每个字节编了一个序号。

确认序号也占用了 4 个字节，它仅在标志位的确认位（ACK）为 1 时才有效。它表明了发送确认报文的一端所期望收到的下一个序号。因此确认序号通常是上次已成功接收的序号加 1。

TCP 数据段首部含有 6 个标志位，如表 9.2 所示。

表 9.2 TCP 数据段首部的标志位

名 称	含 义
URG	紧急指针是否有效
ACK	确认是否有效
PSH	是否是 PUSH 操作
RST	连接是否复位
SYN	同步标志
FIN	发送端完成发送

窗口大小字段表明了接收端窗口的大小，以字节表示。

校验和字段是对 TCP 首部和数据段进行校验的结果。

紧急指针字段仅在紧急指针标志（URG）为 1 时才有效，它表明了 TCP 报文段中紧急数据的最后一个字节的序号。紧急指针字段使得接收端知道紧急数据的长度。

9.7 UDP 协议

UDP（User Datagram Protocol）用户数据报协议与 TCP 协议同处传输层，它为应用层程序提供了一个不可靠的、无连接的服务，使得应用层的程序可以不必建立连接就发送 IP 数据报文。

与 TCP 协议相比，UDP 协议作为一种无连接的协议，它的错误检测与控制功能较弱，但它提高了数据传输的高速性。同时，由于 UDP 协议是不可靠的协议，因此它占用的网络资源较少，更节约网络资源。

UDP 协议的作用就是将 UDP 消息提供给应用层，它并不负责重新发送丢失的或出错的数据消息，不对接收到的无序 IP 数据报重新排序，不消除重复的 IP 数据报，不对已收到的数据报进行确认，也不负责建立和终止连接。所有这些问题都将由使用 UDP 协议的应用程序负责处理。

UDP 报文也封装在 IP 报文中进行传输, 图 9.10 描述了 UDP 数据报文的基本格式。



图 9.10 UDP 数据报文格式

UDP 的数据报文格式与 TCP 相比, 少了很多字段, 如帧序号等。它的源端口表明了源计算机发送该数据报文的端口号。目的端口则是目的主机的端口号。UDP 长度包含了 UDP 数据区的数据长度和首部数据长度。而数据校验和是对 UDP 数据区数据和 UDP 首部数据进行校验得到的。

9.8 FTP 协议

FTP (File Transfer Protocol) 文件传输协议位于 TCP/IP 参考模型的应用层, 以 TCP 协议为基础, 是用于文件传输的一个标准协议。FTP 协议基于客户端 / 服务器模式, 它由 FTP 客户端、FTP 服务器、FTP 通信协议三部分组成。FTP 客户端通常是指运行在用户计算机上的程序, FTP 服务器是远程的计算机。FTP 客户端通过运行程序, 使用 FTP 通信协议连接至 FTP 服务器, 来实现文件的上传与下载。

FTP 客户端与服务器之间通过建立两个 TCP 连接来实现文件的传输, 如图 9.11 所示。

控制端口用于建立 FTP 控制连接, 它的端口号是 21。数据端口则用于建立另一个 TCP 连接, 即数据连接, 用于实现传输文件数据, 端口号是 20。FTP 服务器在控制端口 21 上不断监听用户的连接请求, 当用户使用用户名和密码进行登录时, 就向 FTP 服务器发送了连接请求, 此时 FTP 控制连接就建立了, 用户的用户名和密码将通过控制连接传送至服务器, 服务器收到这个请求后进行用户识别, 然后向客户发送确认或拒绝的应答信息。当客户端看到登录成功的信息后, 就可以发出文件传输的命令。服务器从控制连接上收到文件的传输命令后, 就在数据端口 20 发起数据连接, 传输数据。



图 9.11

用户访问 FTP 服务器可以通过多种方式实现。常用的实现方式有：

1) 使用 FTP 工具：Windows 下, 通过一些常用的 FTP 工具如 CuteFtp、FileZilla 等, 可以较方便地实现 FTP 文件传输。

2) 使用浏览器登录：Windows 下, 可以在 Internet Explorer 的地址栏中直接输入 ftp://+ 主机名或 IP 地址, 并在弹出的对话框中输入正确的用户名和密码即可登录 FTP 服务器。

3) 通过使用 Windows 的命令行工具实现访问 FTP 服务器。在命令行上输入 : ftp 主机名或 IP 地址, 同样在输入正确的用户名和密码后, 即可登录 FTP 服务器。

表 9.3 列出了 FTP 协议一些常用的命令。

表 9.3 FTP 协议常用命令

FTP命令	功 能
ascii	将文件传输模式设为 ASCII 模式
bin	将文件传输设为二进制模式
dir	打印远程主机目录, 并显示其读写属性
ls	打印远程主机目录
delete/del	删除远程主机的某个文件
put	将文件上传到远程主机
get	将文件从远程主机下载
by	关闭本次 FTP 连接
quit	强制退出本次 FTP 连接

通过使用上述命令, 用户可以实现 FTP 的诸多功能, 如文件的删除、上传和下载等。



第 10 章

LwIP 轻量级 TCP/IP 协议栈

LwIP 是 TCP/IP 协议栈的一个开放源代码实现，它由瑞士计算机科学院（Swedish Institute of Computer Science）的 Adam Dunkels 等开发，目的是减少内存使用率和代码空间大小，因此 LwIP 适用于运行在资源受限的嵌入式系统环境中。对于不同配置的 LwIP 协议栈，LwIP 可以在几百字节或者几十 KB 的 RAM 空间中运行。LwIP 既可以移植到操作系统上运行，也可以在没有操作系统的环境下独立运行。

LwIP 具有如下特性：

- ☐ 支持多网络接口下的 IP 转发，且在新版本中支持 IP 数据分片传输功能。
- ☐ 支持 ICMP 协议。
- ☐ 支持 UDP（用户数据报协议）协议。
- ☐ 支持包含阻塞控制、RTT 估算和快速恢复和快速转发的 TCP（传输控制协议）协议。
- ☐ 支持 PPP（Point to Point Protocol）点对点通信协议。
- ☐ 支持 DHCP 协议，可以动态分配 IP 地址。
- ☐ 支持 IPv6 协议。
- ☐ 提供专门的内部回调函数接口，用于提高应用程序性能。
- ☐ 提供可选的 Berkeley 接口函数，即 Socket 套接字 API 函数。

本章结合 TCP/IP 协议栈的基本原理，讲解了包括 LwIP 的进程模型、缓冲和内存的管理方法、协议栈的运输层及与应用程序的接口等基本内容。

10.1 LwIP 进程模型

本节简要介绍常见的三种进程模型和 LwIP 所采用的进程模型，以及其优缺点。

这里所说的进程模型是指 TCP/IP 协议栈的各协议如 IP 协议、TCP 协议、ICMP 协议等是如何实现的。通常，进程模型可以分为如下三类：

- ☐ TCP/IP 协议栈的每一个协议都通过一个不同的进程实现。在该模型下，每个进程都严格地与一个协议相对应。这种进程模型的优点是网络协议的每一层都很清晰，代码的调试和理解都很容易，而且每一层都可以随时参与系统运行，或从系统运行中退出。然而该模型的缺点则是，进程间的上下文切换较为频繁，系统将为频繁的上下文切换付出较大的代价。
- ☐ TCP/IP 协议栈驻留在操作系统的内核中，应用程序通过系统调用与 TCP/IP 协议栈通信。该

模型下，各层协议并非严格地与一个进程相对应。Windows 通过这种进程模型实现了 TCP/IP 协议栈。

- TCP/IP 协议栈驻留在同一个进程中，独立于操作系统内核空间。LwIP、uIP 均采用了这种模型实现了 TCP/IP 协议栈。LwIP 作为一个独立的进程，运行在用户空间内，其优点是方便地移植到不同的操作系统中运行，对于无操作系统的嵌入式环境而言，通过封装相应的操作系统模拟接口函数，便可以使 LwIP 运行在无操作系统环境下。LwIP 提供了三种接口，供应用程序访问：底层的内核回调函数，也称为原始 API 接口；较高层次的有序 API 接口；Berkeley 接口 API。这里简单介绍一下有序 API。有序 API (Sequential API) 提供一种常规、序列化的编程方法来使用 LwIP 堆栈。它与基于 BSD 模式的套接字 API 调用非常类似。有序 API 的执行模型是基于阻塞的打开-读-写-关闭的范式。应用程序通过有序 API 接口访问 LwIP 协议栈时，必须与 LwIP 驻留在不同的进程中。

10.2 LwIP 缓冲与内存管理

通信系统里的核心问题之一就是：内存与缓冲管理如何适应不同大小的内存需求，如一个 TCP 段可能有几百个字节，而一个 ICMP 回显数据却仅有几个字节。为了避免内存复制带来系统额外的开销，应该尽可能让缓冲区中的数据内容驻留在不能被网络子系统管理的存储区中，比如应用程序存储区或者只读存储器。本节结合嵌入式开发中的应用需求，介绍 LwIP 协议栈的缓冲与内存管理策略。

根据前面对 TCP/IP 协议栈的分析可以知道，TCP/IP 各分层的数据有着自己特定的格式，当数据在各层之间传输时，将对数据缓冲区进行频繁操作，如在数据缓冲区首部和尾部添加数据，从数据缓冲区移除数据等。频繁地对数据缓冲区进行操作，将会影响程序的执行效率，也会影响程序的稳定性。通常嵌入式系统的内存资源相对紧张，因此需要一个简单高效的 TCP/IP 协议栈，以减少对内存和程序空间的需求，而这些都是内存与缓冲管理所需要解决的问题。

10.2.1 LwIP 动态内存管理机制

LwIP 的动态内存管理机制大体上可以分为三种：标准 C 运行库自带的内存分配策略、LwIP 的动态内存堆分配策略、LwIP 的动态内存池分配策略。具体使用哪种动态内存管理机制，可以通过修改 opt.h 文件中的相关宏定义来实现。以下是相关的源代码。

```
#ifndef
#define MEM_LIBC_MALLOC 0
#endif
#ifndef MEMP_MEM_MALLOC
#define MEMP_MEM_MALLOC 0
#endif
#ifndef MEM_USE_POOLS
#define MEM_USE_POOLS 0
#endif
```

在上述代码中，将 `MEM_LIBC_MALLOC` 修改为 1，则表明使用标准 C 运行库自带的内存分配策略；将 `MEMP_MEM_MALLOC` 修改为 1，则表明使用 LwIP 的动态内存堆分配策略；将 `MEM_USE_POOLS` 修改为 1，则表明使用 LwIP 的动态内存池分配策略。值得注意的是，LwIP 的动态内存堆分配策略可以与 C 运行库自带的内存分配策略共用，即将宏定义 `MEM_LIBC_MALLOC` 和 `MEMP_MEM_MALLOC` 同时定义为 1，但是若使用了动态内存池分配策略，则不能再使用 C 运行库自带的内存分配策略。

通常情况下，我们更多地会使用 LwIP 自带的动态内存堆分配策略和动态内存池分配策略，下面简单介绍这两种内存分配策略。

首先介绍动态内存堆分配策略。

动态内存堆分配策略的基本原理是对一个预先定义好大小的连续的内存块进行管理。申请内存时，如果在内存堆中找到一个比所请求的内存大的空闲块，就会从其中切割出合适的块，并把剩余的部分返回动态内存堆中。在内存堆管理程序中，宏定义 `MIN_SIZE` 规定了最小申请内存的大小，通常为 12 字节，该 12 字节会存放内存分配器管理的私有数据，该数据区不能被用户程序修改，否则将导致致命问题。释放内存时，内存堆管理程序会查看该节点前后相邻的内存块是否空闲，如果空闲则合并成一个大的内存空闲块。

采用这种分配策略的优点是内存管理比较简单，节省空间，适用于小内存的管理，但缺点就是，如果动态分配和释放次数过于频繁，就可能会造成较多的内存碎片，如果碎片情况严重的话，可能会导致内存分配不成功。接下来将对内存堆分配策略中使用的几个函数做个简单介绍。

- ❑ `void mem_init(void)`：这是内存堆的初始化函数，主要用于获取内存堆的起止地址，同时也将初始化空闲列表，该函数在 LwIP 初始化函数 `LwIP_Init()` 中调用，外部的应用程序不能够调用此函数。
- ❑ `void * mem_malloc(mem_size_t size)`：这是申请内存的函数。它的形参是将要申请的内存字节数，返回值是指向最新分配的内存的指针，若内存申请失败，则返回值是 `NULL`，分配的空间大小会受到内存对齐的影响，可能会比申请的略大。需要留意的是，使用该函数申请的内存空间是没有被初始化的，因此这块内存可能包含任何随机的数据。该函数不能在中断函数里面调用。另外，由于内存堆是全局变量，因此对内存的申请、释放操作做了线程安全保护，如果有多个线程在同时进行内存申请和释放，那么可能会因为信号量的等待而导致申请耗时较长。
- ❑ `void *mem_calloc(mem_size_t count, mem_size_t size)`：该函数也用于申请内存。它的形参有两个：元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小。该函数与 `mem_malloc` 功能相似，与之不同的是它会把动态分配的内存清零，因此程序可以不必在使用 `mem_set()` 函数时对申请的内存空间进行清零的初始化操作。
- ❑ `void mem_free(void *rmem)`：该函数是内存释放函数。它的形参是所申请的内存空间的指针。调用该函数后，之前申请的内存空间将返回内存堆中。

相比于动态内存堆分配策略，动态内存池分配策略可以有效防止内存碎片的产生，而且内存的分配、释放效率更高，不过，它会消耗更多的内存空间。当使用动态内存池作为 LwIP 的内存

管理机制时，还需要再定义一个头文件 lwippools.h，并增加如下宏定义（详情请见 mem.c 文件的注释）。

```
LWIP_MALLOC_MEMPOOL_START
LWIP_MALLOC_MEMPOOL(20, 256)
LWIP_MALLOC_MEMPOOL(10, 512)
LWIP_MALLOC_MEMPOOL(5, 1512)
LWIP_MALLOC_MEMPOOL_END
```

以上宏定义的意义是分配 20 个 256 字节长度的内存块、10 个 512 字节的内存块、5 个 1512 字节的内存块。内存池管理程序会根据以上的宏定义自动在内存中开辟一大块连续的空间用于内存池。当申请内存的时候，内存池管理程序将根据所请求的大小，选择到最适合长度的池里面去申请。而且，如果启用宏定义 MEM_USE_POOLS_TRY_BIGGER_POOL，而上述最适合长度的池中并没有空间可以用了，分配器将从更大长度的池中申请，这也是动态内存池分配策略会浪费更多内存的原因。

LwIP 内部存在很多种内存池，根据 opt.h 文件里的配置不同，将生成不同种类的内存池，如将宏 LWIP_UDP 定义为 1，则就会建立 UDP 类型的内存池。若将宏 LWIP_TCP 定义为 1，则就会建立与 TCP 关联的内存池。同时，LwIP 为内部的一些结构也设计了专用的内存池，如专门存放网络数据包信息的 PBUF_POOL 等，关于各内存池的定义详情可查阅 memp_std.h 文件。

动态内存池分配策略的底层函数在 memp.c、memp.h 文件中实现。仔细阅读 memp.c、memp.h、memp_std.h 中的代码，不得不让人惊叹于 LwIP 对于宏定义的理解，它把 C 语言里对宏定义的运用达到了极致，大量采用了 C 语言的宏特性，设计上面也非常精妙，代码看上去也很优雅，不过对于初学者来说，想要把代码完全看懂，真非易事。下面就让我们通过对部分代码的分析，探讨一下 LwIP 是如何实现一大块内存池的，也希望对读者阅读 LwIP 的代码有所帮助。

```
static u8_t memp_memory[MEM_ALIGNMENT - 1]
#define LWIP_MEMPOOL(name,num,size,desc) + ( (num) * (MEMP_SIZE
                                         + MEMP_ALIGN_SIZE(size) ) )
#include "lwip/memp_std.h"
];
```

数组 memp_memory 就是动态内存池管理所使用的内存池，它的大小是各个组件用量的叠加。各个组件包括与 LWIP_UDP、LWIP_TCP、PBUF_POOL 等相关联的内存池，如下面的代码定义所示，详情可参阅 memp_std.h 文件。

```
#if LWIP_UDP
LWIP_MEMPOOL(UDP_PCB,
             MEMP_NUM_UDP_PCB,
             sizeof(struct udp_pcb),
             "UDP_PCB")
#endif /* LWIP_UDP */

#if LWIP_TCP
LWIP_MEMPOOL(TCP_PCB,
             MEMP_NUM_TCP_PCB,
```

```

        sizeof(struct tcp_pcb),
        "TCP_PCB")
LWIP_MEMPOOL(TCP_PCB_LISTEN,
            MEMP_NUM_TCP_PCB_LISTEN,
            sizeof(struct tcp_pcb_listen),
            "TCP_PCB_LISTEN")
LWIP_MEMPOOL(TCP_SEG,
            MEMP_NUM_TCP_SEG,
            sizeof(struct tcp_seg),
            "TCP_SEG")
#endif /* LWIP_TCP */

#if LWIP_NETCONN
LWIP_MEMPOOL(NETBUF,
            MEMP_NUM_NETBUF,
            sizeof(struct netbuf),
            "NETBUF")
LWIP_MEMPOOL(NETCONN,
            MEMP_NUM_NETCONN,
            sizeof(struct netconn),
            "NETCONN")
#endif /* LWIP_NETCONN */

```

实际上, memp_std.h 头文件是由一条条 LWIP_MEMPOOL(name,num,size,desc) 语句组成的, 而 memp_memory 使用了宏定义 #define LWIP_MEMPOOL(name,num,size,desc) + ((num) * (MEMP_SIZE + MEMP_ALIGN_SIZE(size))), 因此 memp_std.h 最终演变为由一条条 “+ ()” 组成的代码, 换言之, memp_memory 等价于如下的定义:

```

static u8_t memp_memory[MEM_ALIGNMENT - 1
+ ( (num) * (MEMP_SIZE + MEMP_ALIGN_SIZE(size)) )
+ ( (num) * (MEMP_SIZE + MEMP_ALIGN_SIZE(size)) )
.....
];

```

因此, 内存池数组 memp_memory 是 memp_std.h 文件里定义的几个组件所用内存缓冲的叠加。

10.2.2 LwIP 的缓冲管理机制

LwIP 缓冲管理机制的功能是尽量避免内存拷贝, 尽量减少对内存和空间的需求, 提高程序的执行效率。它使用数据结构 pbuf 来描述 LwIP 内部的缓冲数据包。文件 pbuf.h 给出了该数据结构的源代码实现, 如下所示:

```

struct pbuf
{
    struct pbuf *next;
    void *payload;
    u16_t tot_len;
    u16_t len;
    u8_t type;

```

```

u8_t  flags;
u16_t refs;
};

```

在上述代码中，数据字段 `next` 是一个指针，指向下一个 `pbuf` 结构。由于实际发送或接收的数据包长度不一，而每个 `pbuf` 只能管理一部分数据，因此对于大容量的数据包，就必须使用多个 `pbuf` 才能完整地描述它。LwIP 使用链表的数据结构来管理多个数据包。

数据字段 `payload` 是数据指针，指向该 `pbuf` 管理的数据的起始地址，根据数据字段 `type` 的不同，`payload` 所指向的数据起始地址可能位于 RAM，也可能在 ROM 中，后面还将详细讨论这个问题。

数据字段 `tot_len` 表示本 `pbuf` 和其后所有 `pbuf` 有效数据的长度，数据字段 `len` 则表示本 `pbuf` 的有效数据长度。因此 `tot_len` 是当前 `pbuf` 的数据字段 `len` 和其后一个 `pbuf` 中 `tot_len` 的和。因此在 `pbuf` 链表中，第一个 `pbuf` 的 `tot_len` 表示整个数据包的长度，而最后一个 `pbuf` 的 `tot_len` 则与其字段 `len` 相等。

数据字段 `type` 表明了该 `pbuf` 的类型。目前 LwIP 定义了四种类型的 `pbuf`，分别是：`PBUF_RAM`、`PBUF_ROM`、`PBUF_REF` 和 `PBUF_POOL`，在文件 `pbuf.h` 中，这四种类型的 `pbuf` 的代码如下：

```

typedef enum
{
    PBUF_RAM,          /* pbuf data is stored in RAM */
    PBUF_ROM,          /* pbuf data is stored in ROM */
    PBUF_REF,          /* pbuf comes from the pbuf pool */
    PBUF_POOL          /* pbuf payload refers to RAM */
} pbuf_type;

```

数据字段 `flags` 也是 `pbuf` 类型的描述符，不过与 `type` 不同，它表明了协议栈如何处理该 `pbuf`。数据字段 `flags` 对应的宏定义如下所示：

```

#define PBUF_FLAG_PUSH          0x01U
#define PBUF_FLAG_IS_CUSTOM    0x02U
#define PBUF_FLAG_MCASTLOOP    0x04U

```

例如，若 `flags` 被置为 `PBUF_FLAG_IS_CUSTOM`，则在释放该 `pbuf` 时，将会有一些特殊的处理，详情可参考 `pbuf.c` 文件。

数据字段 `refs` 则表明了该 `pbuf` 被引用的次数，它的初始值为 1，当有其他类型的 `pbuf` 指向该 `pbuf` 时，其数据字段 `refs` 加 1。当要释放一个 `pbuf` 时，其 `refs` 字段值必须为 1。

接下来将详细讨论四种类型的 `pbuf`。首先介绍 `PBUF_RAM` 类型的 `pbuf`。

`PBUF_RAM` 类型的 `pbuf` 是通过内存堆分配得到的。LwIP 协议栈和应用程序要传递的数据一般都使用该类型的 `pbuf`。当申请该类型的 `pbuf` 时，LwIP 不仅从内存堆中为其分配申请的数据缓冲区的大小，还为 `pbuf` 数据结构描述部分分配了相应的空间。其代码如下：

```

p = (struct pbuf*)mem_malloc(
    LWIP_MEM_ALIGN_SIZE(sizeof(struct pbuf) + offset)+

```

```
LWIP_MEM_ALIGN_SIZE(length));
```

由上述代码可以看出，LwIP 使用内存堆分配函数 `mem_malloc` 为 `PBUF_RAM` 类型的 `pbuf` 分配内存空间，而申请的内存空间由三部分组成：`pbuf` 数据结构描述占用的空间、长度为 `length` 的有效数据空间和大小为 `offset` 的内存空间。

宏定义 `SIZEOF_STRUCT_PBUF` 是 `pbuf` 数据结构描述部分的长度，其定义如下：

```
#define SIZEOF_STRUCT_PBUF  
    LWIP_MEM_ALIGN_SIZE(sizeof(struct pbuf))
```

数据字段 `offset` 因 `pbuf` 所处层的不同而大小迥异。其代码如下：

```
switch (layer)  
{  
    case PBUF_TRANSPORT:  
        offset += PBUF_TRANSPORT_HLEN;  
    case PBUF_IP:  
        offset += PBUF_IP_HLEN;  
    case PBUF_LINK:  
        offset += PBUF_LINK_HLEN;  
        break;  
    case PBUF_RAW:  
        break;  
    default:  
        LWIP_ASSERT("pbuf_alloc: bad pbuf layer", 0);  
        return NULL;  
}
```

因此，当 `pbuf` 位于传输层时，`offset` 为传输层数据首部长度；当其位于 IP 层时，`offset` 为 IP 首部长度；若 `pbuf` 位于数据链路层，`offset` 为数据链路层数据结构首部长度。

当 `PBUF_RAM` 类型的 `pbuf` 申请成功后，其描述如图 10.1 所示。

`PBUF_POOL` 类型的 `pbuf` 是通过内存池分配得到的。由于分配此类型的 `pbuf` 可以快速完成，适合中断处理，因此它更多地应用在网络设备驱动层。`PBUF_POOL` 类型的 `pbuf` 分配代码如下：

```
p = (struct pbuf *)memp_malloc(MEMP_PBUF_POOL);
```

由上述代码可以看出，LwIP 是通过调用内存池分配函数为 `PBUF_POOL` 类型的 `pbuf` 申请内存空间的，然而与 `PBUF_RAM` 类型的 `pbuf` 不同，由如下代码可以看到，系统将为 `PBUF_POOL` 类型的 `pbuf` 生成一个链表。

```
while (rem_len > 0)  
{
```

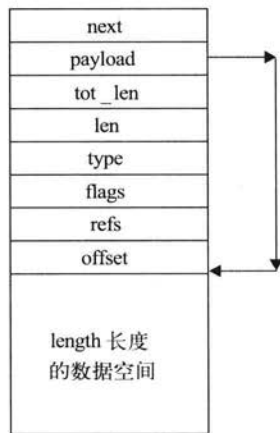


图 10.1 `PBUF_RAM` 类型的 `pbuf`

```

q = (struct pbuf *)memp_malloc(MEMP_PBUF_POOL);
if (q == NULL)
{
    PBUF_POOL_IS_EMPTY();

    /* free chain so far allocated */
    pbuf_free(p);

    /* bail out unsuccessfully */
    return NULL;
}

q->type = type;
q->flags = 0;
q->next = NULL;

/* make previous pbuf point to this pbuf */
r->next = q;

/* set total length of this pbuf and next in chain */
q->tot_len = (u16_t)rem_len;

/* this pbuf length is pool size, unless smaller sized tail */
q->len = LWIP_MIN((u16_t)rem_len,
PBUF_POOL_BUFSIZE_ALIGNED);
q->payload = (void *)((u8_t *)q + SIZEOF_STRUCT_PBUF);
q->ref = 1;

/* calculate remaining length to be allocated */
rem_len -= q->len;

/* remember this pbuf for linkage in next iteration */
r = q;
}

```

分配成功后的 pbuf 如图 10.2 所示。

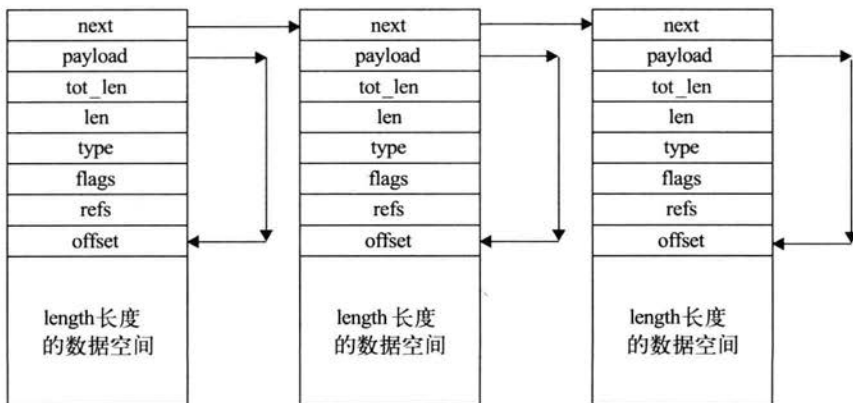


图 10.2 PBUF_POOL 类型的 pbuf

PBUF_REF 和 PBUF_ROM 类型的 pbuf 基本相同，它们都是从内存池中申请分配 pbuf 结构首部空间，而不申请数据区的空间。两者的区别在于，前者指向 RAM 空间内的某段数据，后者指向 ROM 空间内的某段数据。它们的 pbuf 分配代码如下：

```
p = (struct pbuf *)memp_malloc(MEMP_PBUF);
```

由上述代码可以看出，LwIP 也是通过调用 memp_malloc 函数为这两种类型的 pbuf 申请内存空间的，只是请求的内存池类型为 MEMP_PBUF，在文件 memp_std.h 里，其定义如下：

```
LWIP_PBUF_MEMPOOL(PBUF, MEMP_NUM_PBUF,  
0, "PBUF_REF/ROM")
```

通过该定义可以看到，数据空间大小为 0。宏 MEMP_NUM_PBUF 在文件 opt.h 里给出了定义。

这两种类型的 pbuf 主要用于应用程序要发送的数据放置在应用程序管理的存储区的情况，成功分配空间后的 pbuf 结构如图 10.3 所示。

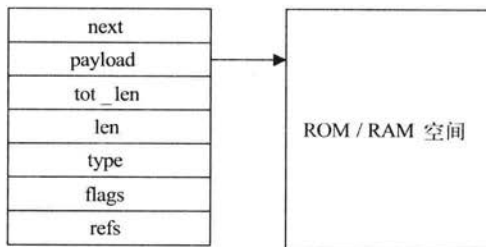


图 10.3 PBUF_REF/PBUF_ROM 类型的 pbuf

10.3 LwIP 网络接口

在 LwIP 中，物理网络硬件的设备驱动通过一个与 BSD 中相似的网络接口结构体 netif 来描述一个硬件网络接口的。网络接口保存在一个全局链表中，它们通过结构体中的 next 指针连接。在单网卡中，这个结构体只有一个；在多网卡中可有与网卡数目相同的 netif 结构体，它们构成一个数据链。

每一个网络接口都拥有一个名字，并保存在 name 字段中。两个字符的名字标识网络接口使用的设备驱动的种类并且只用于这个接口在运行时由人工操作进行配置的情况。名字由设备驱动来设置并且应该反映通过网络接口表示的硬件的种类。比如蓝牙设备的网络接口名字可以是 bt，而 IEEE 802.11b WLAN 设备的名字就可以是 wl。

LwIP 网络接口层主要完成了与底层网络硬件设备驱动相关的功能，它通过结构体 netif 来描述一个硬件接口。该结构体定义如下：

```
struct netif  
{  
    struct netif *next;  
    struct ip_addr ip_addr;  
    struct ip_addr netmask;  
    struct ip_addr gw;  
    err_t (* input)(struct pbuf *p, struct netif *inp);  
    err_t (* output)(struct netif *netif,  
                     struct pbuf *p,  
                     struct ip_addr *ipaddr);  
    err_t (* linkoutput)(struct netif *netif, struct pbuf *p);
```

```

#if LWIP_NETIF_STATUS_CALLBACK
    void (* status_callback)(struct netif *netif);
#endif /* LWIP_NETIF_STATUS_CALLBACK */

#if LWIP_NETIF_LINK_CALLBACK
    void (* link_callback)(struct netif *netif);
#endif

    void *state;

#if LWIP_DHCP
    struct dhcp *dhcp;
#endif /* LWIP_DHCP */

#if LWIP_AUTOIP
    struct autoip *autoip;
#endif

#if LWIP_NETIF_HOSTNAME
    char* hostname;
#endif

    u8_t hwaddr_len;
    u8_t hwaddr[NETIF_MAX_HWADDR_LEN];
    u16_t mtu;
    u8_t flags;
    char name[2];
    u8_t num;

#if LWIP_SNMP
    u8_t link_type;
    u32_t link_speed;
    u32_t ts;
    u32_t ifinoctets;
    u32_t ifinucastpkts;
    u32_t ifinnucastpkts;
    u32_t ifindiscards;
    u32_t ifoutoctets;
    u32_t ifoutucastpkts;
    u32_t ifoutnucastpkts;
    u32_t ifoutdiscards;
#endif

#if LWIP_IGMP
    err_t (*igmp_mac_filter)( struct netif *netif,
        struct ip_addr *group,
        u8_t action);
#endif

#if LWIP_NETIF_HWADDRHINT
    u8_t *addr_hint;
#endif

```

```

#if ENABLE_LOOPBACK
    struct pbuf *loop_first;
    struct pbuf *loop_last;
#if LWIP_LOOPBACK_MAX_PBUFS
    u16_t loop_cnt_current;
#endif /* LWIP_LOOPBACK_MAX_PBUFS */
#endif /* ENABLE_LOOPBACK */
};

```

通过该结构体的定义可以看到，netif 是一个链表结构，链表中的元素通过 next 指针连接。一个网络硬件接口对应一个 netif 结构的变量，当存在多个网络硬件接口的时候，这些 netif 结构的变量通过 next 指针相互连接，构成了一个链表。因此，用户程序应当建立一个全局变量，以维护该链表。

接下来将对 netif 结构的主要字段给出具体的说明，其他的字段定义详情可参考文件 netif.h。

字段 ip_addr、netmask、gw 分别表示了 IP 地址、子网掩码、网关。LwIP 中可以使用如下代码设定这三个字段的值：

```

struct ip_addr ipaddr;
struct ip_addr netmask;
struct ip_addr gw;
IP4_ADDR(&ipaddr, 192, 168, 0, 100);
IP4_ADDR(&netmask, 255, 255, 255, 0);
IP4_ADDR(&gw, 192, 168, 0, 1);

```

- ❑ 字段 input 是一个函数指针，它所指向的函数用于将网络硬件接口收到的数据包传递给上层 TCP/IP 协议栈，参数 p 是收到的数据包。
- ❑ 字段 output 是一个函数指针，它所指向的函数用于将 IP 层的数据包发送到网络硬件接口上。参数 p 是要发送的数据包，参数 ipaddr 是网卡需要将该数据报发送到的目的 IP 地址。
- ❑ 字段 linkoutput 同样也是一个函数指针，这个函数的功能与 output 函数功能基本相同，只是它在 ARP 模块中调用。而且，output 指向的函数也是通过调用 linkoutput 指向的函数实现数据报发送的。
- ❑ 字段 state 可以指向设备驱动的一些状态信息。
- ❑ 字段 hwaddr_len 用于表示硬件接口的地址长度，对于以太网而言，就是 MAC 地址的长度，为 6。
- ❑ 字段 hwaddr 则存放了硬件接口的地址，对于以太网而言，就是 MAC 地址。
- ❑ 字段 mtu 则表明了最大的网络传输个数，以字节为单位。
- ❑ 字段 flags 是硬件接口状态信息标志位，如是否建立链接状态，是否允许广播功能等，LwIP 定义了如下标志位：

```

#define NETIF_FLAG_UP          0x01U
#define NETIF_FLAG_BROADCAST  0x02U
#define NETIF_FLAG_POINTTOPOINT 0x04U
#define NETIF_FLAG_DHCP       0x08U

```

```
#define NETIF_FLAG_LINK_UP          0x10U
#define NETIF_FLAG_ETHARP          0x20U
#define NETIF_FLAG_IGMP            0x40U
```

❑ 字段 `name` 用来表示硬件接口使用的驱动类型。这个值可以随意设置，但也有一些约定俗成，比如蓝牙设备的网络接口 `name` 值为 `bl`，无线局域网 IEEE802.11b 的网络接口 `name` 值为 `wl`。

❑ 字段 `num` 用来表示硬件接口的编号。当两个硬件接口的 `name` 字段相同的时候，该子段可以用来区分是哪一個硬件接口。

了解了以上主要字段的定义后，接下来看一段如何增加一个网络接口的源代码，这段代码应当由用户自己实现。

```
struct ip_addr ipaddr;           ❶
struct ip_addr netmask;         ❷
struct ip_addr gw;              ❸
uint8_t MACAddress[6]={0x0F,01,0x0a,0,0,1};  ❹
IP4_ADDR(&ipaddr, 192, 168, 0, 8);  ❺
IP4_ADDR(&netmask, 255, 255, 255, 0);  ❻
IP4_ADDR(&gw, 192, 168, 0, 1);  ❼
Set_MAC_Address(MACAddress);      ❽
netif_add(&netif,
          &ipaddr,
          &netmask,
          &gw,
          NULL,
          &ethernetif_init,
          &ethernet_input);      ❾
netif_set_default(&netif);        ❿
netif_set_up(&netif);             ⓫
```

在上述代码中，第❶~❽行用于设置 IP 地址、子网掩码、网关和 MAC 地址，第❾行的代码很关键，正是通过 `netif_add` 这个函数向全局变量 `netif` 链表结构中增加了一个硬件网络接口。这个函数的最后两个参数很重要，就是函数指针 `ethernetif_init` 和 `ethernet_input`，前者是用户程序自己定义的底层接口初始化函数，后者在文件 `etharp.c` 中定义，用于处理接收到的数据报文，并根据报文的类型，将该报文传递给不同的协议层处理函数。

下面分别来看一下 `ethernetif_init` 和 `ethernet_input` 这两个函数的源代码。

`ethernetif_init` 由用户程序自己定义，其代码如下：

```
err_t ethernetif_init(struct netif *netif)
{
    struct ethernetif *ethernetif;
    LWIP_ASSERT("netif != NULL", (netif != NULL));
    ethernetif = mem_malloc(sizeof(struct ethernetif));
    if (ethernetif == NULL)
    {
        return ERR_MEM;
    }

    #if LWIP_NETIF_HOSTNAME
```

```

/* 初始化主机名 */
netif->hostname = "tylzm";
#endif
/* 初始化与 SNMP 相关的一些变量, 最后一个变量是连接速度值, 以 bit/s 为单
位 */
NETIF_INIT_SNMP(netif,
                 snmp_ifType_ethernet_csmACd,
                 100000000);
netif->state = ethernetif;
netif->name[0] = 't';
netif->name[1] = 'y';

/*output 指向的函数可以由用户程序重新编写 */
netif->output = etharp_output;

/*low_level_output 由用户程序定义, 是底层发送数据报的函数 */
netif->linkoutput = low_level_output;

ethernetif->ethaddr = (struct eth_addr *)&(netif->hwaddr[0]);

/* 初始化底层硬件接口 */
low_level_init(netif);

return ERR_OK;
}

```

ethernet_input 在文件 etharp.c 文件中定义, 其代码如下:

```

err_t ethernet_input(struct pbuf *p, struct netif *netif)
{
    struct eth_hdr* ethhdr;

    /* p 的 payload 指向了以太网数据报的报文首部 */
    ethhdr = p->payload;
    switch (htons(ethhdr->type))
    {
        /* 检查是否是 IP 数据报 */
        case ETHTYPE_IP:
            #if ETHARP_TRUST_IP_MAC
                /* 更新 ARP 表 */
                etharp_ip_input(netif, p);
            #endif

            /* 重新调整 p 的 payload 指针位置, 越过报文首部 */
            if(pbuf_header(p, -(s16_t)sizeof(struct eth_hdr)))
            {
                LWIP_ASSERT("Can't move over header in packet", 0);
                pbuf_free(p);
                p = NULL;
            }
            else
            {

```

```

ip_input(p, netif);          /* 传递给 IP 层的调用函数 */
}
break;

case ETHTYPE_ARP:
    /* 传递给 ARP 模块 */
    etharp_arp_input(netif,
                      (struct eth_addr*)(netif->hwaddr),
                      p);

    break;

    default:
        ETHARP_STATS_INC(etharp.proterr);
        ETHARP_STATS_INC(etharp.drop);
        pbuf_free(p);
        p = NULL;
        break;
    }
    return ERR_OK;
}

```

在函数 `ethernet_input` 中, `htons(ethhdr->type)` 是值得注意的一个函数。它的功能是将一个半字长的数据从网络数据字节顺序转换到处理器 CPU 所使用的字节顺序。处理器 CPU 使用的字节顺序有大端模式 (big-endian) 和小端模式 (little-endian) 两种, 本书使用的处理器是基于 Cortex-M3 内核的 STM32F107 系列处理器, 因此使用的是小端模式。而网络数据字节顺序则使用了大端模式, 所以需要使用函数 `htons(ethhdr->type)` 实现大端模式到小端模式的转换。

LwIP 的网络层接口还有很多与底层设备驱动相关的函数, 如 `low_level_init`、`low_level_input`、`low_level_output` 等, 本章节就不再一一详述了, 将在第 11 章中再做详细讨论。

10.4 LwIP 的 ARP 处理

如前所述, ARP 协议是 TCP/IP 协议的基础, 而它的本质是实现 IP 地址与底层物理地址的相互转换。当你在装有 Windows 操作系统的 PC 机的命令行里输入命令 “arp -a” 后, 就会看到如图 10.4 所示的 ARP 缓存信息。

每个 IP 地址都与一个 48 位长的物理地址 (MAC 地址) 相对应。

ARP 协议的核心是 ARP 缓存表, 而 ARP 协议的实质就是对缓存表的建立、更新、查询等操作。ARP 缓存表是由一个个缓存表项组成的, 在 LwIP 中, 描述缓存表项的数据结构叫 `etharp_entry`, 源代码如下:



图 10.4 ARP 缓存描述

```

struct etharp_entry
{
#ifdef ARP_QUEUEING
    // 数据队列指针
    struct etharp_q_entry *q;
#endif
    // IP 地址
    struct ip_addr ipaddr;
    // MAC 地址
    struct eth_addr ethaddr;
    // 缓存表项的状态
    enum etharp_state state;
    // 缓存表项的时间信息
    u8_t ctime;
    // 该缓存表项对应的物理网络接口信息
    struct netif *netif;
};

```

在上述结构体中，字段 `ipaddr` 存放 IP 地址，`ethaddr` 则用于存放与 `ipaddr` 相对应的物理地址，这两个字段是 ARP 缓存表的核心。字段 `state` 是个枚举类型的变量，其定义如下：

```

enum etharp_state
{
    ETHARP_STATE_EMPTY = 0,
    ETHARP_STATE_PENDING,
    ETHARP_STATE_STABLE
};

```

在上述代码中，`ETHARP_STATE_EMPTY` 状态表明缓存表项处于初始状态，没有记录任何信息；`ETHARP_STATE_PENDING` 状态表示该表项处于不稳定状态，例如，该表项只记录了 IP 地址，但是尚未记录与该 IP 地址相对应的 MAC 地址。在该状态下，LwIP 内核会向总线上发出一个广播 ARP 请求，以让对应 IP 地址的主机回应其 MAC 地址。`ETHARP_STATE_STABLE` 状态表明该表项已经完全记录了一对 IP 地址和 MAC 地址，此时该表项处于稳定状态。

字段 `ctime` 记录 ARP 缓存表项处于某个状态的时间，当某表项的 `ctime` 值大于规定的表项最大生存值时，LwIP 内核会删除该表项。因此使用 ARP 功能时，必须设置一个 ARP 超时事件，该超时事件的基本功能就是对每个表项的 `ctime` 字段值加 1，然后删除那些生存时间大于最大生存值的表项。

在本书的第 9 章中，表 9.1 描述了一个 ARP 报文的基本结构，在 LwIP 中，也使用了一个结构体来描述 ARP 报文的首部，其定义如下：

```

struct etharp_hdr
{
    PACK_STRUCT_FIELD(struct eth_hdr ethhdr);
    PACK_STRUCT_FIELD(u16_t hwtype);
    PACK_STRUCT_FIELD(u16_t proto);
    PACK_STRUCT_FIELD(u16_t _hwlen_protolen);
    PACK_STRUCT_FIELD(u16_t opcode);
};

```

```

PACK_STRUCT_FIELD(struct eth_addr shwaddr);
PACK_STRUCT_FIELD(struct ip_addr2 sipaddr);
PACK_STRUCT_FIELD(struct eth_addr dhwaddr);
PACK_STRUCT_FIELD(struct ip_addr2 dipaddr);
} PACK_STRUCT_STRUCT;

```

字段 ethhdr 长度为 14 个字节, 包含了目的物理地址、源物理地址和帧类型描述字。字段 hwtype 是硬件协议类型。字段 proto 是协议类型。字段 _hwlen_protolen 用于描述硬件地址长度和协议地址长度。字段 opcode 是操作类型描述。字段 shwaddr 是源主机物理地址。字段 sipaddr 是源主机 IP 地址。字段 dhwaddr 是目的主机物理地址。字段 dipaddr 是目的主机 IP 地址。

图 10.5 描述了 LwIP 中 ARP 的操作流程。

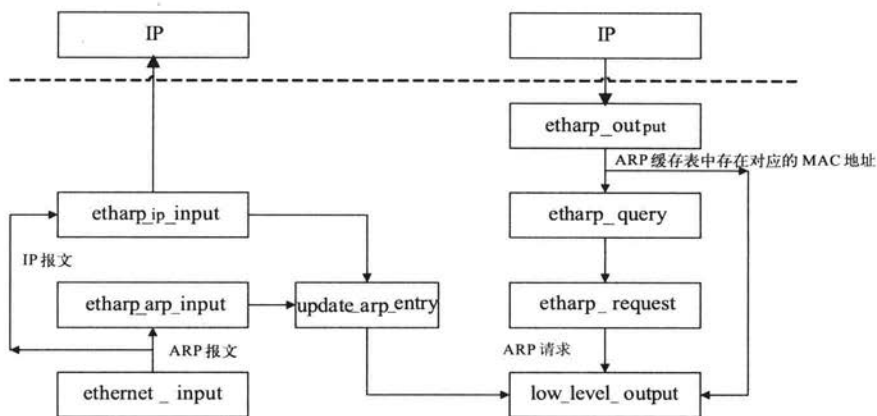


图 10.5 ARP 处理流程

从上述流程图可以看到, 当有数据包输入时, 首先要判断是否是 ARP 数据包, 如果是, 则针对不同的 ARP 包类型做相应的响应; 如果是 IP 数据包, 则继续向上递交给 IP 层处理。而 IP 层向下发送一个数据报的时候, 需要通过 ARP 实现 IP 到 MAC 地址的映射, 若在 ARP 缓存表中找不到对应的目的主机 MAC 地址, 则需要发送 ARP 数据报文, 获得目的主机的 MAC 地址。

在 ARP 处理流程中, 可以看到一些与 ARP 处理相关的函数, 下面就简单介绍一下这些函数。

```

void etharp_ip_input(struct netif *netif, struct pbuf *p);
void etharp_arph_input(struct netif *netif,
                      struct eth_addr *ethaddr,
                      struct pbuf *p);
static err_t update_arp_entry(struct netif *netif,
                             struct ip_addr *ipaddr,
                             struct eth_addr *ethaddr,
                             u8_t flags);
err_t etharp_query(struct netif *netif,
                  struct ip_addr *ipaddr,
                  struct pbuf *q);
err_t etharp_request(struct netif *netif, struct ip_addr *ipaddr);

```

- ❑ 函数 `ethernet_input` 根据报文首部的帧类型字段判断接收到的报文类型，如果是 IP 包，则将该包递交给 `etharp_ip_input`，如果是 ARP 包，则将该包递交给 `etharp_arp_input`。
- ❑ 函数 `etharp_ip_input` 调用函数 `update_arp_entry`，它是利用报文首部的 MAC 地址和 IP 地址更新 ARP 缓存的。在 LwIP 1.3.1 中，这个函数的实现很简单，其代码如下：

```
void etharp_ip_input(struct netif *netif, struct pbuf *p)
{
    struct ethip_hdr *hdr;
    LWIP_ERROR("netif != NULL", (netif != NULL), return);
    hdr = p->payload;
    if (!ip_addr_netcmp(&(hdr->ip.src),
                      &(netif->ip_addr),
                      &(netif->netmask)))
    {
        return;
    }
    update_arp_entry(netif, &(hdr->ip.src), &(hdr->eth.src), 0);
}
```

- ❑ 函数 `etharp_arp_input` 的实现则较为复杂，它首先判断接收到的 ARP 数据包的类型，如果是 ARP 请求包，那么首先判断这个包是不是给自己的，如果是给自己的，就在原有包的基础上重组一个 ARP 应答包发送出去；如果不是给自己的，则直接忽略。而如果接收到的数据包是 ARP 应答包，那么就调用函数 `update_arp_entry` 更新 ARP 缓存表。
- ❑ 函数 `update_arp_entry` 是 ARP 相关函数中较为重要的一个函数，它用于更新 ARP 缓存表中的表项或者在缓存表中插入一个新的表项，并且该函数会在收到一个 IP 数据包或 ARP 数据包后被调用。
- ❑ 函数 `etharp_query` 的功能是向指定的 IP 地址发送一个 ARP 请求，它将分为几种情况：如果给定的 IP 地址不在 ARP 缓存表中，则创建一个新的 ARP 表项，并且该表项处于 `ETHARP_STATE_PENDING` 状态，同时还将发送一个关于该 IP 地址的 ARP 请求包；如果 IP 地址在 ARP 缓存表中有相应的表项，但该表项处于 `ETHARP_STATE_PENDING` 状态，则也发送一个关于该 IP 地址的 ARP 请求包；如果 IP 地址在 ARP 表中有相应的表项，且表项处于 `ETHARP_STATE_STABLE` 状态，然后再判断给定的数据包是否为空，如果不为空则直接将该数据包发送出去，如果为空则发送一个关于该 IP 地址的 ARP 请求包。
- ❑ 函数 `etharp_request` 的作用就是发送一个由形参 `ipaddr` 指定 IP 地址的 ARP 请求包。

10.5 LwIP 的 IP 处理

第 9 章简单介绍了 IP 数据报文的基本格式及 IP 首部各字段的含义，IP 首部的定义如图 10.6 所示。

4位版本	4位首部长度	8位服务类型	16位总长度	
16位标志			3位标志	13位片偏移
8位生存时间	8位协议		16位首部校验和	
32位源地址				
32位目的地址				

图 10.6 IP 首部数据格式

在 LwIP 中，有着与之相对应的数据结构，其代码如下：

```
struct ip_hdr
{
    PACK_STRUCT_FIELD(u16_t _v_hl_tos);
    PACK_STRUCT_FIELD(u16_t _len);
    PACK_STRUCT_FIELD(u16_t _id);
    PACK_STRUCT_FIELD(u16_t _offset);
#define IP_RF 0x8000
#define IP_DF 0x4000
#define IP_MF 0x2000
#define IP_OFFMASK 0x1fff
    PACK_STRUCT_FIELD(u16_t _ttl_proto);
    PACK_STRUCT_FIELD(u16_t _chksum);
    PACK_STRUCT_FIELD(struct ip_addr src);
    PACK_STRUCT_FIELD(struct ip_addr dest);
} PACK_STRUCT_STRUCT;
```

- ❑ 字段 `_v_hl_tos` 包含了 4 位版本号、4 位首部长度和 8 位服务类型的描述。
- ❑ 字段 `_len` 与 16 位总长度相对应。
- ❑ 字段 `_id` 与 16 位标志相对应。
- ❑ 字段 `_offset` 包含了 3 位标志和 13 位片偏移的描述。
- ❑ 字段 `_ttl_proto` 包含了 8 位 TTL 字段和 8 位协议类型描述。
- ❑ 字段 `_chksum` 与 16 位首部数据校验和相对应。
- ❑ 字段 `src`、`dest` 分别是 32 位源 IP 地址、32 位目的 IP 地址。

在 LwIP 中，关于 IP 数据报文的处理有几个较为重要的函数，现列举如下：

```
err_t ip_input(struct pbuf *p, struct netif *inp);
err_t ip_output(struct pbuf *p,
                struct ip_addr *src,
                struct ip_addr *dest,
                u8_t ttl,
                u8_t tos,
                u8_t proto);
```

函数 `ip_input` 是 IP 报文的输入函数，该函数在 `ethernet_input` 中被调用。它的处理流程如图 10.7 所示。

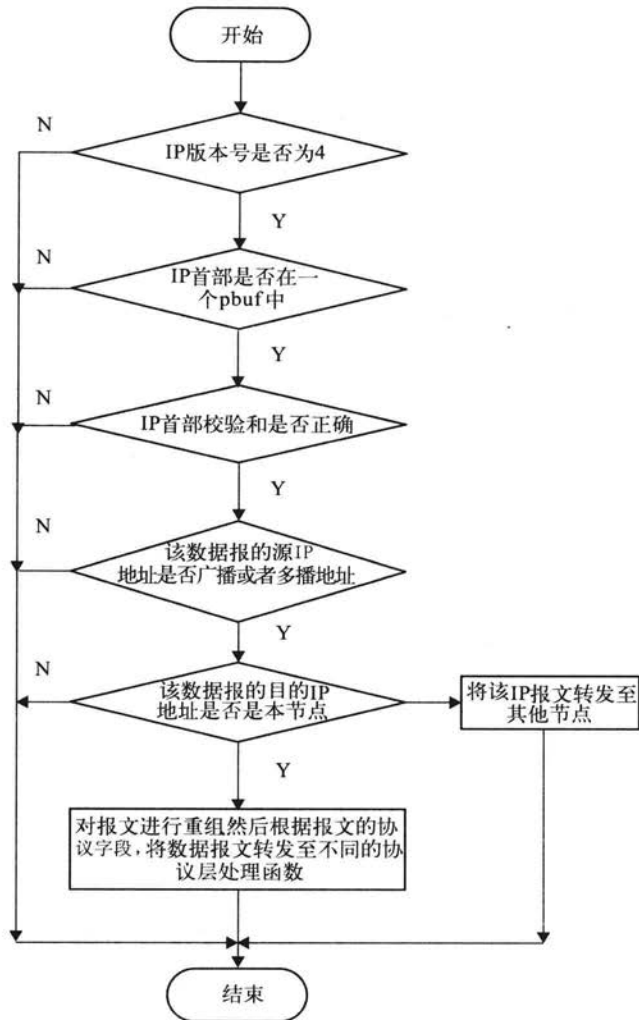


图 10.7 `ipinput` 函数的处理流程

首先，函数 `ip_input` 检查 IP 首部的 4 位版本号字段，这是因为低版本的 LwIP 协议栈（如 1.3.1 版本）并不支持 IPv6，因此若当前的版本值不为 4，则直接丢弃该数据包。

然后该函数检查 IP 首部是否只保存于一个 pbuf 中，如果不是，直接丢弃该 IP 包，这是因为 LwIP 不允许 IP 首部被封装在不同的 pbuf 里面。同时，函数还检查 IP 首部中的总长度字段是否大于数据报文总长度，如果是，则说明存在传输错误，直接丢弃数据包。

接下来，函数 `ip_input` 通过调用函数 `inet_chksum` 校验 IP 首部数据，将得到的校验和与报文

中的 16 位 IP 首部校验和字段相比较，若不相等，则丢弃该报文。

紧接着，函数 `ip_input` 还将判断发送该报文的源 IP 地址是否是广播 IP 地址或是组播 IP 地址，如果是，则丢弃该报文。

接下来，函数 `ip_input` 检测 IP 数据包中的目的 IP 地址是否与本节点的 IP 地址相符，如果是本节点的 IP 地址，则根据该 IP 数据包首部的协议字段判断该数据包应该被递交给哪个上层协议，并调用相应的函数。如果是 UDP 协议，则调用 `udp_input` 函数；如果是 TCP 协议，则调用 `tcp_input` 函数；如果是 ICMP 协议，则调用 `icmp_input` 函数；如果是 IGMP 协议，则调用 `igmp_input` 函数；如果都不是，则调用函数 `icmp_dest_unreach` 返回一个协议不可达 ICMP 数据包给源主机，同时将该数据包删除。如果不是本节点的 IP 地址，则通过调用函数 `ip_forward` 对数据包进行转发。这里值得注意的是，由于一个节点可能含有多个 IP 地址，因此 `ip_input` 函数会遍历网络接口链表 `netif_list` 上的 `netif` 结构变量，来查找与 IP 数据包中相匹配的 IP 地址。

至此，IP 报文处理中关于接收 IP 数据报文的部分就介绍完了。下面再来讨论下 LwIP 是如何处理发送的报文的。这通过调用函数 `ip_output` 来实现。

函数 `ip_output` 的源代码实现很简单，如下所示：

```
err_t ip_output(struct pbuf *p,
                struct ip_addr *src,
                struct ip_addr *dest,
                u8_t ttl,
                u8_t tos,
                u8_t proto)
{
    struct netif *netif;
    if ((netif = ip_route(dest)) == NULL)
    {
        IP_STATS_INC(ip.rterr);
        return ERR_RTE;
    }
    return ip_output_if(p, src, dest, ttl, tos, proto, netif);
}
```

总的说来，函数 `ip_output` 使用 `ip_route()` 函数查找目的网络接口 `netif` 来发送 IP 数据包。当网络接口 `netif` 确定后，IP 数据包通过函数 `ip_output_if()` 发送出去。若 `ip_route()` 没有找到合适的网络接口，则丢弃该报文，终止本次发送。函数 `ip_route()` 通过遍历网络接口链表 `netif_list`，查找与目的 IP 地址在同一个子网中的网络接口，并将该网络接口返回给变量 `netif`。如果没有找到在同一个子网中的网络接口，则 `ip_route()` 返回默认的网络接口变量。

10.6 LwIP 的 ICMP 处理

在 LwIP 协议栈中，文件 `icmp.c` 实现了 ICMP 的相关处理。LwIP 对 ICMP 的实现较为简单，并未实现 ICMP 协议的全部内容，例如 `icmp_input` 函数只解析 ICMP 回显请求报文等。

文件 `icmp.c` 包含四个函数，如下所示：

```

void icmp_input(struct pbuf *p, struct netif *inp);
void icmp_dest_unreach(struct pbuf *p, enum icmp_dur_type t);
void icmp_time_exceeded(struct pbuf *p, enum icmp_te_type t);
void icmp_send_response(struct pbuf *p, u8_t type, u8_t code);

```

函数 `icmp_input` 在 `ip_input` 中被调用，它处理接收到的 ICMP 数据包，并根据包类型做相应的处理。在 1.3.1 版本的 LwIP 协议栈中，它只处理 ICMP 回显请求包，对其他类型的 ICMP 包不做响应。`icmp_input` 在处理 ICMP 回显请求时，首先判断该数据包是否为广播或者组播包，如果是，则直接返回，不再继续处理；如果不是，则继续判断该数据包长度是否小于 ICMP 回显请求头部长度，如果是则丢弃数据包；如果不是则将该 ICMP 报文类型字段变为 0，重新计算校验和，并将 IP 报文首部的源 IP 地址和目的 IP 地址交换位置，并通过调用函数 `ip_output_if` 将数据包发送出去。

函数 `icmp_dest_unreach` 在 `ip_input`、`udp_input` 中被调用，它的功能是通过调用函数 `icmp_send_response` 发送一个“目的地不可到达”类型的 ICMP 报文。在函数 `ip_input` 中，当所接收的 IP 报文协议字段不可识别时，`icmp_dest_unreach` 将被调用。而在 UDP 处理中，若不能找到与接收的报文相对应的端口号，则 `icmp_dest_unreach` 也将被调用。

函数 `icmp_time_exceeded` 在 `ip_forward` 中被调用，它的功能是通过调用函数 `icmp_send_response` 发送一个“超时”类型的 ICMP 报文。在函数 `ip_forward` 中，当 TTL 值减小为 0 时，调用该函数。

10.7 LwIP 的 UDP 处理

UDP (User Datagram Protocol) 用户数据报协议为应用层程序提供了一个不可靠的、无连接的服务，使得应用层的程序可以不必建立连接就发送 IP 数据报文。在 LwIP 协议栈中，UDP 协议的实现流程如图 10.8 所示。

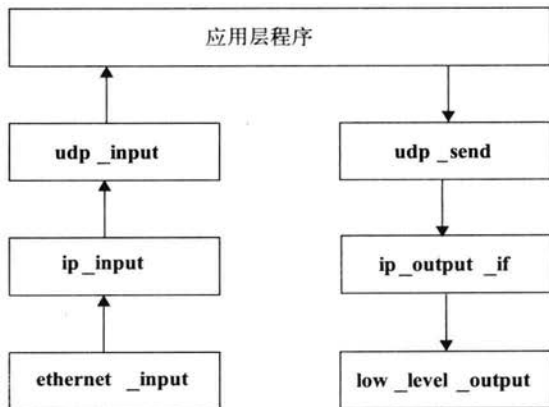


图 10.8 UDP 的处理流程

函数 `ip_input` 解析 IP 数据报文后, 若该报文是 UDP 报文, 便将该报文传递给函数 `udp_input`, 由其检查报文的 UDP 校验, 并最终调用函数 `recv`, 将收到的报文传递给应用层程序。

当应用层程序要通过 UDP 协议向外发送 IP 报文时, 将通过调用函数 `udp_send` 实现, 函数 `udp_send` 通过调用 IP 层的函数 `ip_output_if` 实现报文的发送。

LwIP 使用链表结构体 `udp_pcb` 来保存每一个 UDP 会话的状态, 其源代码如下:

```
struct udp_pcb
{
    struct udp_pcb *next;
    u8_t flags;
    u16_t local_port, remote_port;
#ifdef LWIP_IGMP
    struct ip_addr multicast_ip;
#endif
#ifdef LWIP_UDPLITE
    u16_t chksum_len_rx, chksum_len_tx;
#endif
    void (* recv)(void *arg,
                  struct udp_pcb *pcb,
                  struct pbuf *p,
                  struct ip_addr *addr,
                  u16_t port);

    void *recv_arg;
};
```

在这个结构体中, 字段 `next` 是指向下一个 `udp_pcb` 的指针变量。字段 `flags` 标识什么样的 UDP 校验和策略应该用于这个会话。字段 `local_port` 和 `remote_port` 分别表示本地的端口号和远端主机的端口号。字段 `recv` 是一个函数指针, 是在由 `udp_pcb` 指定的会话收到一个数据包时使用, 即在收到 UDP 数据包时调用 `recv` 指向的函数。宏定义 `LWIP_IGMP` 表明是否使用 IGMP 功能, 宏定义 `LWIP_UDPLITE` 用于表明是否支持基于 UDP 的流媒体协议功能。

接下来将对 LwIP 1.3.1 协议中实现的 `udp` 协议函数做一个简单说明。

```
void udp_input(struct pbuf *p, struct netif *inp);
err_t udp_send(struct udp_pcb *pcb, struct pbuf *p);
err_t udp_sendto(struct udp_pcb *pcb,
                  struct pbuf *p,
                  struct ip_addr *dst_ip,
                  u16_t dst_port);
err_t udp_sendto_if(struct udp_pcb *pcb,
                    struct pbuf *p,
                    struct ip_addr *dst_ip,
                    u16_t dst_port,
                    struct netif *netif);
err_t udp_bind(struct udp_pcb *pcb,
               struct ip_addr *ipaddr,
               u16_t port);
err_t udp_connect(struct udp_pcb *pcb,
                  struct ip_addr *ipaddr,
```

```

        u16_t port);
void udp_disconnect(struct udp_pcb *pcb);
void udp_recv(struct udp_pcb *pcb,
              void (* recv)(void *arg,
                           struct udp_pcb *upcb,
                           struct pbuf *p,
                           struct ip_addr *addr,
                           u16_t port),
              void *recv_arg);
void udp_remove(struct udp_pcb *pcb);

```

- ❑ 函数 `udp_input` 用于处理接收到的 UDP 数据报文，其函数内部通过调用函数 `recv` 将收到的报文数据递交给上层应用程序。
- ❑ 函数 `udp_send` 用于发送 UDP 数据包，它的源代码实现很简单，就是直接调用 `udp_sendto` 函数。
- ❑ 函数 `udp_sendto` 用于发送 UDP 数据包到远端主机的指定 IP 地址和端口上。这个函数将会调用 `ip_route` 以查找远端主机是否存在于网络接口链表 `netif_list` 中，当找到远端主机后，将调用 `udp_sendto_if` 发送数据报文。
- ❑ 函数 `udp_sendto_if` 按照指定的网络接口和 ip 地址发送 UDP 数据报文。
- ❑ 函数 `udp_bind` 用于将本地 IP 地址和端口号绑定于一个 UDP 协议控制块中。
- ❑ 函数 `udp_connect` 用于实现与给定的 IP 地址和端口号的远端主机相连接。
- ❑ 函数 `udp_disconnect` 用于断开与指定的 UDP 协议控制块的连接。
- ❑ 函数 `udp_recv` 用于设置接收到数据包时调用的回调函数及其参数。
- ❑ 函数 `udp_remove` 用于从协议控制链表中删除指定 UDP 协议控制块，并释放相应的内存资源。

10.8 LwIP 的 TCP 处理

10.8.1 TCP 处理流程概述

TCP（传输控制协议 / 网间协议）是一个工业标准的协议集。TCP 属于传输层协议，它为应用层提供了可靠的字节流服务。本节结合 LwIP 讲述 TCP/IP 协议栈中 TCP 的实现和收发机理。

在 LwIP 协议栈 1.3.1 版本中，TCP 的实现代码是最为复杂的一部分，它占据了整个协议栈 50% 左右的源代码，而且由 3 个文件组成，分别是 `tcp.c`、`tcp_in.c`、`tcp_out.c`。其中，`tcp.c` 包含了一些 TCP 处理的通用函数，如超时处理、建立连接等；`tcp_in.c` 主要是处理 TCP 的输入；`tcp_out.c` 则是与 TCP 输出相关的一些函数。

图 10.9 简要描述了 TCP 发送和接收数据报文的基本流程。这个流程也与 TCP/IP 的四层模型基本相对应。

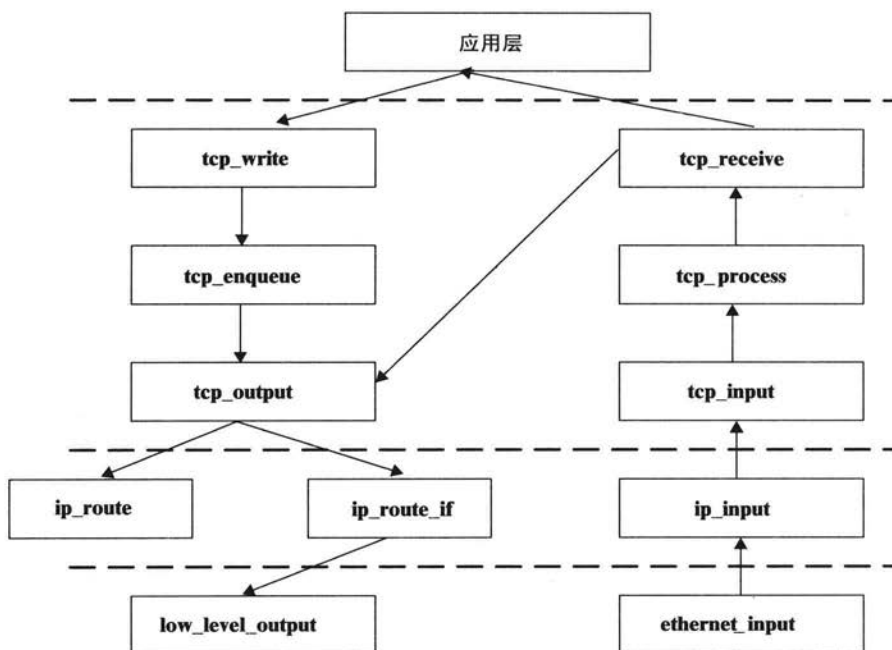


图 10.9 TCP 处理简要流程

首先来看看其发送数据流程。报文的发送是由应用层程序发起的。应用层程序首先调用 `tcp_write` 函数，接着 `tcp_write` 函数再将控制权交给 `tcp_enqueue` 函数，这个函数会在必要时将数据分割为适当大小的 TCP 段，然后再把这些 TCP 段放到所属连接的传输队列中。这时，`tcp_output` 函数会检查能否发送数据，如果可以发送数据，就调用 `ip_route` 及 `ip_output_if` 函数发送报文。

其次来看看报文的接收流程。网络接口层函数 `ethernet_input` 收到报文后将数据包传递给 `ip_input` 函数，该函数将属于 TCP 协议的报文传递给 `tcp_input` 函数。`tcp_input` 函数主要完成两个功能：一是对数据报文进行校验和判断与 TCP 选项解析；二是判定这个 TCP 报文属于哪个 TCP 连接。接着，这个 TCP 报文到达 `tcp_process` 函数，这个函数实现了 TCP 状态机，任何必要的状态转换将在这里实现。当该 TCP 报文所属的连接正处于接收网络数据的状态，`tcp_receive` 函数将被调用。最终，`tcp_receive` 函数将数据传给上层的应用程序，完成数据报文的接收过程。

10.8.2 TCP 控制块

在 LwIP 中，使用结构体 `tcp_pcb` 来描述一个完整的 TCP 连接。其源代码如下：

```

struct tcp_pcb
{
    IP_PCB;
    TCP_PCB_COMMON(struct tcp_pcb);
    u16_t remote_port;
    u8_t flags;

```

```

/* 以下宏定义是 flags 的值 */
#define TF_ACK_DELAY      ((u8_t)0x01U)
#define TF_ACK_NOW        ((u8_t)0x02U)
#define TF_INFR           ((u8_t)0x04U)
#define TF_TIMESTAMP      ((u8_t)0x08U)
#define TF_FIN            ((u8_t)0x20U)
#define TF_NODELAY        ((u8_t)0x40U)
#define TF_NAGLEMEMERR    ((u8_t)0x80U)

/* 用于数据接收, 接收窗口的相关变量 */
u32_t rcv_nxt;
u16_t rcv_wnd;
u16_t rcv_ann_wnd;
u32_t rcv_ann_right_edge;

/* 定时器相关的变量 */
u32_t tmr;
u8_t polltmr, pollinterval;
s16_t rtime;
u16_t mss;

/* rtt 相关的变量 */
u32_t rttest;
u32_t rtseq;
s16_t sa, sv;

/* 重发超时时间 */
s16_t rto;

/* 重发的次数 */
u8_t nrtx;

/* 快速重传 / 恢复相关的参数 */
u32_t lastack;
u8_t dupacks;

/* 拥塞避免 / 控制 */
u16_t cwnd;
u16_t ssthresh;

/* 发送的相关变量 */
u32_t snd_nxt;
u16_t snd_wnd;
u32_t snd_wl1, snd_wl2;
u32_t snd_lbb;
u16_t acked;
u16_t snd_buf;
#define TCP_SNDQUEUELEN_OVERFLOW (0xffff-3)
u16_t snd_queuelen;

/* tcp_seg 相关的变量 */
struct tcp_seg *unsent;

```

```

    struct tcp_seg *unacked;

#ifdef TCP_QUEUE_OOSEQ
    struct tcp_seg *ooseq;
#endif

    struct pbuf *refused_data;

#ifdef LWIP_CALLBACK_API
    err_t (* sent)(void *arg, struct tcp_pcb *pcb, u16_t space);
    err_t (* recv)(void *arg,
                   struct tcp_pcb *pcb,
                   struct pbuf *p,
                   err_t err);
    err_t (* connected)(void *arg,
                        struct tcp_pcb *pcb,
                        err_t err);
    err_t (* poll)(void *arg, struct tcp_pcb *pcb);
    void (* errf)(void *arg, err_t err);
#endif /* LWIP_CALLBACK_API */

#ifdef LWIP_TCP_TIMESTAMPS
    u32_t ts_lastacksent;
    u32_t ts_recent;
#endif

    /* 坚持 / 保活定时器变量 */
    u32_t keep_idle;

#ifdef LWIP_TCP_KEEPALIVE
    u32_t keep_intvl;
    u32_t keep_cnt;
#endif
    u32_t persist_cnt;
    u8_t persist_backoff;
    u8_t keep_cnt_sent;
};

```

在上述结构体中，IP_PCB 是一个宏，描述了连接的 IP 地址等相关信息，包括本地和目的节点的 IP 地址、TTL 信息等。

TCP_PCB_COMMON 也是一个宏，其实现如下：

```

#define TCP_PCB_COMMON(type) \
    type *next; \
    enum tcp_state state; \
    u8_t prio; \
    void *callback_arg; \
    u16_t local_port; \
    DEF_ACCEPT_CALLBACK

```

在上述定义中，next 是一个函数指针，它指向下一个 tcp_pcb 结构，所有的 tcp_pcb 结构构成

了一个 TCP 连接的链表。enum tcp_state 是一个枚举变量，定义了一个 TCP 连接的状态，共有 11 种状态，包括关闭连接、监听状态、超时等待等，其定义如下：

```
enum tcp_state
{
    CLOSED          = 0,
    LISTEN          = 1,
    SYN_SENT        = 2,
    SYN_RCVD        = 3,
    ESTABLISHED     = 4,
    FIN_WAIT_1      = 5,
    FIN_WAIT_2      = 6,
    CLOSE_WAIT      = 7,
    CLOSING         = 8,
    LAST_ACK        = 9,
    TIME_WAIT       = 10
};
```

- ❑ 字段 flags 定义了一些附属的状态信息，如连接是否快速恢复等，在 tcp_pcb 中，紧接着 flags 定义的宏就是 flags 的值。
- ❑ 字段 rcv_nxt、rcv_wnd、rcv_ann_wnd、rcv_ann_right_edge 用于接收数据。rcv_nxt 是指期望从远程主机得到的下一个包序号，因此该字段用于向远程主机发送 ACK 包。rcv_wnd 则表明了本地接收窗口的大小。rcv_ann_wnd、rcv_ann_right_edge 在高版本的 LwIP 协议栈中才有。rcv_ann_wnd 表示向远程主机声明的本地接收窗口大小，而 rcv_ann_right_edge 的含义则是向远程主机声明接收窗口时，本地节点经过计算得到的接收窗口的最右端。
- ❑ 字段 tmr、polltmr、pollinterval、rttime、mss 是与定时器相关的一些变量。tmr 表示该连接上次数据包到达的时间，polltmr、pollinterval 用于查询相关的定时器。rttime 是重传定时器。字段 mss 表明了最大数据段的大小，LwIP 中实际发送的数据包都是以 mss 大小进行切分的。
- ❑ 字段 rttest、rtseq、sa、sv 用于 RTT（往返时间估计）算法。其中，rttest 的含义是用于计算 RTT 的数据分片的发送时间，它是以 500ms 为单位的整数。rtseq 表示计算 RTT 的数据分片的序列号。sa、sv 则是 RTT 算法估计出的平均值及其时间差。
- ❑ 字段 lastack、dupacks 用于“快速重传 / 恢复”机制，分别表示最大的确认序号和这个序号被重传的次数。
- ❑ 字段 cwnd、ssthresh 用于拥塞避免 / 控制，其中 cwnd 的含义是连接的当前阻塞窗口，ssthresh 则表示了慢速启动阈值的大小。
- ❑ 字段 snd_nxt、snd_wnd、snd_wl1、snd_wl2、snd_lbb、acked、snd_buf、snd_queuelen 用于发送数据。snd_nxt 表明了下一次将要发送的序列号。snd_wnd 则指定了发送窗口的大小。snd_wl1、snd_wl2 分别表示上次更新发送窗口时（收到了一个数据包）的序号和确认序号。snd_lbb 保存了传输队列最后一个字节的顺序编号。acked 表示已经接收的最高 ACK 序号。snd_buf 则是按照发送窗口计算得到的可以发送的数据缓冲字节数。snd_queuelen 表示可用的发送缓冲空间。

- ❑ 字段 `unsent`、`unacked`、`ooseq` 则是与 `tcp_seg` 链表相关的一些变量。从应用层接收但还未被发送的数据被放置在 `unsent` 队列排队等待发送，已经发送但还未收到远程主机应答确认的数据保存在 `unacked` 队列。接收到序列以外的数据由 `ooseq` 保存。
- ❑ 字段 `refused_data` 表示已经收到但是还没有被应用程序取走的数据。
- ❑ 字段 `keep_idle` 表明了多久后进行保活检测。`keep_intvl` 和 `keep_cnt` 分别用于记录用户自定义的保活包时间间隔与保活包个数。`keep_cnt_sent` 记录了已经发送的保活数据包的个数。
- ❑ 字段 `persist_cnt`、`persist_backoff` 是与坚持定时器相关的两个变量。`persist_cnt` 用于坚持定时器计数，当该计数值超过某个值时，则发出窗口探查数据包。`persist_backoff` 用于表示是否启动坚持定时器，以及已经发出几个探查数据包。

10.8.3 LwIP 的 TCP 滑动窗口

TCP 的滑动窗口协议是用于实现流量控制的。在 LwIP 1.3.1 版本中，使用了较多的变量来实现该协议，如 `snd_nxt`、`snd_wnd`、`snd_wl1`、`snd_wl2`、`snd_lbb`。图 10.10 描述了发送滑动窗口的基本构成。

从总体上看，滑动窗口把发送缓冲区分成三个部分。最左边至 `lastack` 是已经被协议栈发送且收到对方确认的数据分组；`lastack` 至 `snd_wnd` 末端是窗口内部数据分组；从 `snd_wnd` 至右边则是尚未发送的分组数据。

再来看看窗口内部的构成。从 `lastack` 到 `snd_nxt` 之间的数据是已经发送出去但未被确认的，它们被挂接在控制块的 `unacked` 链表上；从 `snd_nxt` 到 `snd_lbb` 间的数据是已经被应用程序递交给协议栈，但并未被协议栈发送出去的数据，这些数据挂接在控制块的 `unsent` 链表上；从 `snd_lbb` 到窗口末端表示可用的发送缓冲，但应用层还未递交数据，下一次应用层递交的数据将从 `snd_lbb` 开始编号。

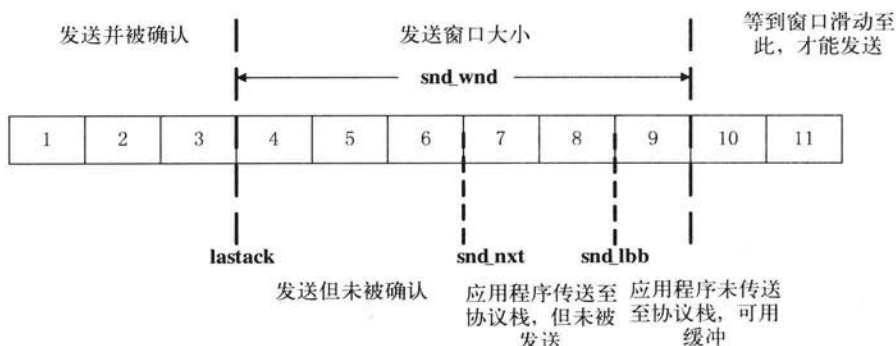


图 10.10 TCP 发送滑动窗口

发送窗口的更新与 `snd_wl1`、`snd_wl2` 密切相关。当收到一个数据段时，如果 `snd_wl1` 和 `snd_wl2` 字段的值与数据段中的 `seqno` 和 `ackno` 间的大小关系满足窗口更新条件，则更新发送窗口。这个更新条件在函数 `tcp_receive` 中被实现，其代码如下：

```

if (TCP_SEQ_LT(pcb->snd_wll, seqno) ||
    (pcb->snd_wll == seqno && TCP_SEQ_LT(pcb->snd_wl2, ackno)) ||
    (pcb->snd_wl2 == ackno && tcphdr->wnd > pcb->snd_wnd))
{
    pcb->snd_wnd = tcphdr->wnd;
    pcb->snd_wll = seqno;
    pcb->snd_wl2 = ackno;
    if (pcb->snd_wnd > 0 && pcb->persist_backoff > 0)
    {
        pcb->persist_backoff = 0;
    }
}

```

在上述代码中，宏定义 TCP_SEQ_LT 用于判断两个变量相减值是否小于 0，其实现如下：

```
#define TCP_SEQ_LT(a,b) ((s32_t)((a)-(b)) < 0)
```

从上述代码可以看到，发送窗口的更新满足下列条件之一即可：

- ❑ snd_wll 小于收到的包序号。
- ❑ snd_wll 与收到的包序号相等，并且 snd_wl2 小于收到的包的确认序号。
- ❑ snd_wl2 与收到的包确认序号相等，并且接收端的窗口大于本地窗口。

接收窗口的实现相对于发送窗口来说较为简单，共有 4 个字段，分别是 rcv_nxt、rcv_wnd、rcv_ann_wnd、rcv_ann_right_edge。

在图 10.11 中，rcv_nxt 表示下一个将要接收的数据段序号；rcv_wnd 表示接收窗口的大小；rcv_ann_wnd 表示向发送方通告的窗口大小；rcv_ann_right_edge 则是向远程主机声明接收窗口时，本地节点经过计算得到的接收窗口的最右端。在文件 tcp.c 中，函数 tcp_update_rcv_ann_wnd 实现了对接收窗口的更新。

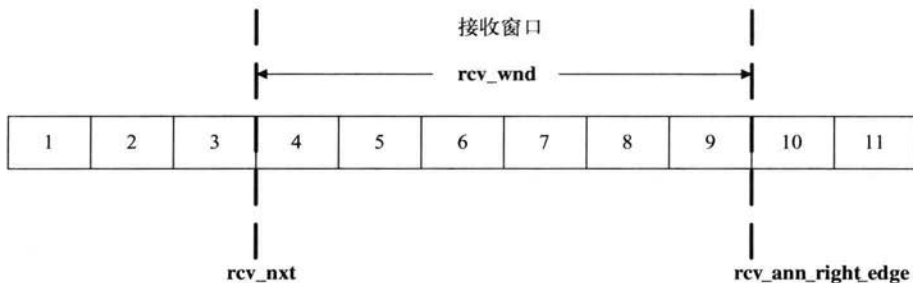


图 10.11 TCP 接收滑动窗口

10.8.4 LwIP 的 TCP 超时与重传

TCP 的超时与重传机制提高了数据传输的可靠性，下面就来看看在 LwIP 中是如何实现超时与重传的。

TCP 超时的判断主要使用了两个字段：rtime、rto。rtime 值为 -1 时表示超时计数器未被使能，当值为非 0 时表示超时计数器使能。rto 是设定的超时时间。当超时计数器被使能后，rtime 的值

在 TCP 的 500ms 中断中加 1，一旦 rtime 大于或者等于 rto，则所有 unacked 链表上的数据段都将被重新发送。在函数 tcp_slowtmr（这个函数每 500ms 被调用一次）中，如下一段代码显示了数据的超时与重传机制。

```
/* 超时计数器被使能后，rtime 每 500ms 加 1 */
if (pcb->rtime >= 0)
    ++pcb->rtime;

/* unacked 队列中有数据段未发送且已经超时 */
if (pcb->unacked != NULL && pcb->rtime >= pcb->rto)
{
    if (pcb->state != SYN_SENT)
    {
        pcb->rto = ((pcb->sa >> 3) + pcb->sv) <<
                    tcp_backoff[pcb->nrtx];
    }

    /* 复位超时计数器 */
    pcb->rtime = 0;

    /* 调整拥塞控制窗口和慢启动阈值 */
    eff_wnd = LWIP_MIN(pcb->cwnd, pcb->snd_wnd);
    pcb->ssthresh = eff_wnd >> 1;
    if (pcb->ssthresh < pcb->mss)
    {
        pcb->ssthresh = pcb->mss * 2;
    }
    pcb->cwnd = pcb->mss;

    /* 数据段重传 */
    tcp_rexmit_rto(pcb);
}
```

函数 tcp_rexmit_rto 实现重传的机制其实很简单，它将 unacked 链表上的所有数据段插入 unsent 队列的前端，并将控制块重传次数 nrtx 加 1，最后调用 tcp_output 重发数据段。其代码如下：

```
void tcp_rexmit_rto(struct tcp_pcb *pcb)
{
    struct tcp_seg *seg;
    if (pcb->unacked == NULL)
    {
        return;
    }
    for (seg = pcb->unacked; seg->next != NULL; seg = seg->next);
    seg->next = pcb->unsent;
    pcb->unsent = pcb->unacked;
    pcb->unacked = NULL;
    ++pcb->nrtx;
    pcb->rttest = 0;
}
```

```

    tcp_output(pcb);
}

```

10.8.5 LwIP 的 TCP 拥塞控制

在 LwIP 中，拥塞控制是通过慢启动算法和拥塞避免算法来实现的。这其中有两个很重要的字段，分别是 `cwnd`、`ssthresh`。

`cwnd` 定义了拥塞窗口的大小，它表明了发送方在收到 ACK 应答之前，允许向网络发送的数据量大小，发送数据后未收到 ACK 应答前，拥塞窗口变小，收到 ACK 后，拥塞窗口增加。`ssthresh` 则是慢启动的阈值，它也随着通信的过程不断变化。

使用慢启动算法时，`cwnd` 初始化时被设定为 1 个报文段大小，此后每收到一个 ACK 就增加 1 倍，因此 `cwnd` 呈指数级增长，可以说，慢启动算法一点也不慢，只是初始值比较小。`cwnd` 的指数级增长可以最大程度地利用网络带宽资源，但是 `cwnd` 不能一直这样无限增长下去，一定需要某个限制。慢启动门限变量 `ssthresh` 就用于实现限制功能，当 `cwnd` 超过该值后，慢启动过程结束，进入拥塞避免阶段。拥塞避免算法要求每次收到一个确认时将 `cwnd` 增加 $1/cwnd$ 。总的说来，当 `cwnd` 小于 `ssthresh` 时，使用慢启动算法；当 `cwnd` 大于等于 `ssthresh` 时，使用拥塞避免算法。

接下来，再看看以下几种情况发生时，`cwnd` 和 `ssthresh` 是如何更新的。当超时重传发生时，函数 `tcp_slowtmr` 中有如下代码：

```

/* 发送缓冲区只能取 cwnd 和 snd_wnd 中较小者 */
eff_wnd = LWIP_MIN(pcb->cwnd, pcb->snd_wnd);

/* ssthresh 设置为有效大小的一半 */
pcb->ssthresh = eff_wnd >> 1;

/* 修正 ssthresh 至少为 2 个报文段大小 */
if (pcb->ssthresh < pcb->mss)
{
    pcb->ssthresh = pcb->mss * 2;
}

/* 设置 cwnd 大小为 1 个报文段大小 */
pcb->cwnd = pcb->mss;

```

当拥塞是由重复确认引起的，则相应的更新代码在 `tcp_receive` 函数中实现：

```

if (pcb->cwnd > pcb->snd_wnd)
    pcb->ssthresh = pcb->snd_wnd / 2;
else
    pcb->ssthresh = pcb->cwnd / 2;

if (pcb->ssthresh < 2*pcb->mss)
{
    pcb->ssthresh = 2*pcb->mss;
}

pcb->cwnd = pcb->ssthresh + 3 * pcb->mss;

```

当发送数据 pcb 收到 ACK 应答时，也将更新 cwnd 和 ssthresh 的值。

```
if (pcb->state >= ESTABLISHED)
{
    /* 使用慢启动算法 */
    if (pcb->cwnd < pcb->ssthresh)
    {
        if ((u16_t)(pcb->cwnd + pcb->mss) > pcb->cwnd)
        {
            pcb->cwnd += pcb->mss;
        }
    }
}
else /* 使用拥塞算法 */
{
    u16_t new_cwnd = (pcb->cwnd + pcb->mss * pcb->mss /
                     pcb->cwnd);

    if (new_cwnd > pcb->cwnd)
    {
        pcb->cwnd = new_cwnd;
    }
}
}
```

10.8.6 LwIP 的 TCP 定时器

LwIP 中含有两个定时器函数：tcp_fasttmr 和 tcp_slowtmr。tcp_fasttmr 每 250 ms 调用一次，也称为快速定时器；tcp_slowtmr 每 500ms 调用一次，也称为慢速定时器。

快速定时器处理函数 tcp_fasttmr 主要做了两方面的工作，一是向上层递交上层一直未接收的数据，二是发送该连接上的延迟 ACK 请求数据段。函数 tcp_fasttmr 的实现代码较为简单，如下所示：

```
void tcp_fasttmr(void)
{
    struct tcp_pcb *pcb;

    /* 遍历 tcp_active_pcbs 链表 */
    for(pcb = tcp_active_pcbs; pcb != NULL; pcb = pcb->next)
    {
        /* 如果某个控制块还有数据未接收 */
        if (pcb->refused_data != NULL)
        {
            /* 调用上层函数接收数据 */
            err_t err;
            TCP_EVENT_RECV(pcb, pcb->refused_data, ERR_OK, err);
            if (err == ERR_OK)
            {
                /* 成功接收则复位指针 */
                pcb->refused_data = NULL;
            }
        }
    }
}
```

```

    }
    /* 若控制块开启了延迟 ACK 定时器 */
    if (pcb->flags & TF_ACK_DELAY)
    {
        /* 发送一个“立即确认”消息 */
        tcp_ack_now(pcb);
        pcb->flags &= ~(TF_ACK_DELAY | TF_ACK_NOW);
    }
}
}

```

慢速定时器函数 `tcp_slowtmr` 的实现较为复杂，这个函数中完成了较多的功能，如超时与重传、拥塞控制、坚持与保活定时器的处理等，这里就不再一一赘述了，有兴趣的读者可参考相关文献。

10.9 LwIP 的应用程序接口简介

使用 TCP/IP 协议栈提供的服务有两种方法：其一，直接调用 TCP 与 UDP 模块的函数；其二，使用后面将要介绍的 LwIP API 函数。TCP 与 UDP 模块提供网络服务的一个基本接口，该接口基于函数回调技术，因此使用该接口的应用程序可以不用进行连续操作。不过，这会增加应用程序编写难度且代码不易理解。为了接收数据，应用程序会向协议栈注册一个回调函数。该回调函数与特定的连接相关联，当该关联的连接到达一个信息包，该回调函数就会被协议栈调用。

由 BSD 提供的高级别的套接字应用程序接口不适合于资源受限系统的 TCP/IP 实现，特别是 BSD 套接字需要将要发送的数据从应用程序复制到 TCP/IP 协议栈的内部缓冲区。复制数据的原因是应用程序与 TCP/IP 协议栈通常驻留在不同的受保护空间，此外，数据的复制将消耗嵌入式系统的有限资源。

LwIP API 是以嵌入式应用程序为设计目标，它可以充分利用 LwIP 的内部结构以避免 BSD API 对系统资源的过度依赖。LwIP API 与 BSD API 类似，但操作相对低级。API 不需要在应用程序和协议栈之间复制数据，减少了数据复制给系统带来的额外开销，因为应用程序可以巧妙地直接处理内部缓冲区。因为 BSD 套接字应用程序接口易于理解，并且很多应用程序为它而写，所以 LwIP 保留了一个 BSD Socket 兼容层。

在 LwIP 1.3.1 中，LwIP 提供了三种应用程序接口，用于应用层程序访问 LwIP 协议栈内部提供的各种服务：一是直接调用协议栈各模块的函数，它是基于回调函数的 API 接口，也称为 RAW API 接口；二是使用 LwIP 提供的专用 API 函数，也称为 Sequential API 接口；三是与 BSD Socket 兼容的 Socket 函数接口。下面将着重对前两 API 接口做一个探讨，关于 Socket 函数接口，读者可以参考 `socket.c` 文件。

10.9.1 RAW API 接口

RAW API 接口的程序执行机制是以回调函数为基础的，回调函数直接被协议栈代码调用，因

此应用程序代码和 TCP/IP 协议栈运行在同一个进程里，无需使用操作系统，两者之间这种良好的结合可以使得程序的执行效率更高，而且在运行中它占用更少的内存资源。然而使用 RAW API 接口会使得应用程序的代码编写比较困难，而且代码通常都晦涩难懂，不易于提高源代码的可读性。

在 LwIP 源代码的 doc 文件夹下，rawapi.txt 文件对 RAW API 接口做了详细说明，本小节对 TCP 的部分函数做一个简单介绍。

首先对关于 TCP 实现的一些 RAW API 接口函数做一个简单介绍。

1. 应用连接状态设置函数

```
void tcp_arg(struct tcp_pcb *pcb, void *arg);
```

tcp_arg() 函数用于将连接的具体状态传递给应用程序，在函数 tcp_new() 调用之后才能被调用，形参 pcb 是当前 TCP 连接的控制块，arg 则是需要传递给回调函数的参数。

2. 建立 TCP 连接的函数

与建立 TCP 连接有关的函数有 3 个，分别是 tcp_new()、tcp_bind 和 tcp_listen。

```
struct tcp_pcb *tcp_new(void);
```

tcp_new() 函数在定义一个 tcp_pcb 控制块后应该首先被调用，以建立该控制块的连接标志。如果正常建立了连接标志，则返回建立的 pcb；如果不成功，则返回 NULL。

```
err_t tcp_bind(struct tcp_pcb *pcb,
               struct ip_addr *ipaddr,
               u16_t port);
```

tcp_bind() 函数用于绑定本地的 IP 地址和端口号，用户可以将其绑定在一个任意的本地 IP 地址上，它也只能在函数 tcp_new() 调用之后才能被调用。形参 pcb 表示准备绑定的连接控制块。形参 ipaddr 则是绑定的 IP 地址。如果 ipaddr 的值为 IP_ADDR_ANY，则将连接绑定到所有的本地 IP 地址上。形参 port 表示绑定的本地端口号。若成功绑定，则返回 ERR_OK。

```
struct tcp_pcb *tcp_listen(struct tcp_pcb *pcb);
```

tcp_listen() 函数使指定的连接开始进入监听状态。形参 pcb 指定将要进入监听状态的连接控制块。若监听成功，就返回一个新的连接控制块 pcb，它将作为一个参数传递给将要分派的函数。否则返回 NULL。

```
void tcp_accepted(struct tcp_pcb *pcb);
```

tcp_accepted() 函数用于通知 LwIP 一个新来的连接已经被接收。这个函数通常在由 tcp_accept 指定的回调函数中被调用。它允许 LwIP 去执行一些内务工作，例如将新来的连接放入监听队列中，以等待处理。

```
void tcp_accept(struct tcp_pcb *pcb,
               err_t (* accept)(void *arg,
                                struct tcp_pcb *newpcb,
                                err_t err));
```

`tcp_accept()` 函数将指定处于监听状态的连接成功建立连接后将要调用的回调函数。形参 `pcb` 是一个处于监听状态的连接，`accept` 则是回调函数。

```
err_t tcp_connect(struct tcp_pcb *pcb,
                  struct ip_addr *ipaddr,
                  u16_t port,
                  err_t (* connected)(void *arg,
                                      struct tcp_pcb *tpcb,
                                      err_t err));
```

`tcp_connect()` 函数用于请求连接指定的连接到远程主机。形参 `pcb` 是指定的连接控制块。`ipaddr` 是远程主机的 IP 地址。`port` 是远程主机的端口号。`connected` 是连接正确建立后调用的回调函数。

3. TCP 数据发送函数

```
err_t tcp_write(struct tcp_pcb *pcb,
                void *dataptr,
                u16_t len,
                u8_t copy);
```

`tcp_write()` 函数用于发送 TCP 数据。将要发送的数据会被放入发送队列中，由协议栈内核发送。形参 `pcb` 是指定所要发送的连接控制块。`dataptr` 是一个指针，它指向准备发送的数据。`len` 则是要发送数据的长度。`copy` 为 0 或者 1，它将决定是否为发送的数据分配新的内存空间，如果该参数为 0，则不会为发送的数据分配新的内存空间，若为 1 则分配新的内存空间。

```
void tcp_sent(struct tcp_pcb *pcb,
              err_t (* sent)(void *arg,
                              struct tcp_pcb *tpcb,
                              u16_t len));
```

`tcp_sent()` 函数用于指定当远程主机成功接收数据后，应用程序调用的回调函数。

4. TCP 数据接收函数

```
void tcp_recv(struct tcp_pcb *pcb,
              err_t (* recv)(void *arg,
                              struct tcp_pcb *tpcb,
                              struct pbuf *p,
                              err_t err));
```

`tcp_recv()` 函数用于指定接收数据时调用的回调函数。形参 `pcb` 是与远程主机相连接的控制块。`recv` 则是回调函数。

```
void tcp_recved(struct tcp_pcb *pcb, u16_t len);
```

`tcp_recved()` 函数用于获取接收到的数据的长度，它必须在由 `tcp_recv` 指定的回调函数中被调用。

5. TCP 关闭连接的函数

```
err_t tcp_close(struct tcp_pcb *pcb);
```

tcp_close() 函数用于关闭一个指定的 TCP 连接，调用该函数后，将会释放 pcb 控制块所占用的内存空间。

```
void tcp_abort(struct tcp_pcb *pcb);
```

tcp_abort() 函数用于终止一个指定的连接。形参 pcb 是当前的连接控制块，调用该函数后，pcb 控制块所占用的内存空间将被释放。

```
void tcp_err(struct tcp_pcb *pcb,
void (* err)(void *arg, err_t err));
```

tcp_err() 函数用于指定处理错误的回调函数。

10.9.2 Sequential API 接口

RAW API 函数接口是基于回调函数机制的，它无需操作系统支持，而 Sequential API 接口则与 BSD 标准的 socket API 非常相似，程序的执行过程基于 open-read-write-close 模型，而且需要操作系统的支持，另外在文件 lwipopts.h 中，需要把宏定义 NO_SYS 定义为 0。

Sequential API 被分成两部分实现。一部分驻留在应用程序进程中，另一部分在 TCP/IP 协议栈进程内实现。这两部分 API 之间采用由操作系统模拟层提供的进程间通信机制（也称为 IPC）进行通信。在 LwIP 中，操作系统模拟层是 LwIP 协议栈的一部分，它存在的目的是方便 LwIP 的移植，它在底层操作系统和 LwIP 协议栈之间提供了一个接口，当用户移植 LwIP 到一个新的目标系统的时候，只需要修改这个接口内的函数即可。关于操作系统模拟层的具体实现要求，读者可以参考 doc 文件夹下的 sys_ach.txt 文件。

驻留在应用程序进程中的 API 接口与 TCP/IP 协议栈进程中的 API 之间通过共享内存传递数据，对于该共享内存区的描述是采用和 pbuf 类似的结构来实现，LwIP 将其定义为 netbuf。netbuf 的定义如下所示：

```
struct netbuf
{
    struct pbuf *p, *ptr;
    struct ip_addr *addr;
    u16_t port;
};
```

由以上代码可以看到，netbuf 是基于 pbuf 来实现的。字段 p 是一个函数指针，指向了保存数据的 pbuf 链表。字段 ptr 也指向该保存数据的 pbuf 链表，但与 p 的区别在于，p 固定指向 pbuf 链表中的第一个 pbuf 结构，而 ptr 则可以指向该链表中的任意位置。与 netbuf 相关的几个较为重要的函数如下所示：

```
struct netbuf *netbuf_new (void);
```

netbuf_new() 函数用于分配一个新的 netbuf 结构。这里申请的 netbuf 结构并不包含实际存储数据的区域。

```
void *netbuf_alloc(struct netbuf *buf, u16_t size);
```

netbuf_alloc() 函数分配大小为 size 的内存空间，以存放数据。

```
void netbuf_delete (struct netbuf *buf);
void netbuf_free (struct netbuf *buf);
```

以上两个函数用于删除 buf 所指向的内存空间。其中函数 netbuf_free 在 netbuf_delete 中被调用。

Sequential API 的函数被主要分布在两个文件中，分别是 api_lib.c 和 api_msg.c。前者包括了应用程序可以直接调用的 API 接口函数，后者则包括了与 LwIP 协议栈进程通信的 API 函数，应用程序不可直接调用。这两部分 API 函数之间通过消息邮箱进行通信，邮箱中传递的数据结构为 api_msg。api_lib.c 中的 API 函数与应用程序之间则通过数据结构 netconn 来共享一个连接的各种属性信息。下面分别对这两个数据结构进行阐述。

api_msg 的定义如下：

```
struct api_msg
{
    void (* function)(struct api_msg_msg *msg);
    struct api_msg_msg msg;
};
```

字段 function 是一个函数指针，它指向 api_msg.c 中的某个与协议栈接口的 API 函数。字段 msg 中包含了这个函数执行时需要的所有参数，对于不同的 function 函数指针，msg 有着不同的内容。api_msg_msg 的定义如下：

```
struct api_msg_msg
{
    /* 与消息相关的连接结构 */
    struct netconn *conn;
    union
    {
        /* 函数 do_send 的参数 */
        struct netbuf *b;

        /* 函数 do_newconn 的参数 */
        struct
        {
            u8_t proto;
        } n;

        /* 函数 do_bind 和 do_connect 的参数 */
        struct
        {
            struct ip_addr *ipaddr;
```

```

        u16_t port;
    } bc;

    /* 函数 do_getaddr 的参数 */
    struct
    {
        struct ip_addr *ipaddr;
        u16_t *port;
        u8_t local;
    } ad;

    /* 函数 do_write 的参数 */
    struct
    {
        const void *dataptr;
        size_t len;
        u8_t apiflags;
    } w;

    /* 函数 do_recv 的参数 */
    struct
    {
        u16_t len;
    } r;
#endif LWIP_IGMP
    /* 函数 do_join_leave_group 的参数 */
    struct
    {
        struct ip_addr *multiaddr;
        struct ip_addr *interface;
        enum netconn_igmp join_or_leave;
    } jl;
#endif
    } msg;
};

```

这个结构体只包含了两个字段：conn 用于描述连接信息，它包含了与该连接相关的邮箱和信号量等信息，协议栈程序需要用这些信息来完成与应用程序间的同步与通信；msg 是个共同体类型的变量，用于传递函数指针 function 需要的所有参数。这里通过 app_lib.c 中的一个函数 netconn_getaddr 来看看 api_msg 结构的具体用法。

```

err_t netconn_getaddr(struct netconn *conn,
                     struct ip_addr *addr,
                     u16_t *port,
                     u8_t local)
{
    struct api_msg msg;
    msg.function = do_getaddr;
    msg.msg.conn = conn;
    msg.msg.ad.ipaddr = addr;
    msg.msg.ad.port = port;
}

```

```

    msg.msg.msg.ad.local = local;
    TCPIP_APIMSG(&msg);
    return conn->err;
}

```

由上述代码可以看到，`netconn_getaddr` 首先将字段 `function` 指向了 `api_msg.c` 文件中的函数 `do_getaddr`。然后依次将 `do_getaddr` 需要的各参数赋值给 `api_msg` 的各变量。这里需要说明的是，`app_lib.c` 中 `netconn_xxx` 类型的函数，其代码实现与上述代码基本一致。只是所传递的参数和 `function` 指向的函数不一样。

接下来看看结构体是如何定义的。`netconn` 的定义如下：

```

struct netconn
{
    /* 指明连接的类型：TCP、UDP 或 RAW */
    enum netconn_type type;

    /* 连接的当前状态 */
    enum netconn_state state;
    union
    {
        struct ip_pcb *ip;
        struct tcp_pcb *tcp;
        struct udp_pcb *udp;
        struct raw_pcb *raw;
    } pcb;

    /* 该连接最近一次发生错误的代码 */
    err_t err;

    /* 用于两部分 API 程序间同步的信号量 */
    sys_sem_t op_completed;

    /* 这个邮箱用来存储已经收到的数据，直到其被应用程序进程取走 */
    sys_mbox_t recvmbox;

    /* 用来接收外部连接的邮箱 */
    sys_mbox_t acceptmbox;

    /* 该字段只在 socket 实现中使用 */
    int socket;

#ifdef LWIP_SO_RCVTIMEO
    /* 等待接收新数据的定时器 */
    int rcv_timeout;
#endif

#ifdef LWIP_SO_RCVBUF
    /* recvmbox 邮箱能够容纳的最大数据量，以字节为单位 */
    int rcv_bufsize;
#endif /* LWIP_SO_RCVBUF */
}

```

```

    s16_t recv_avail;

#ifdef LWIP_TCP
    /* 数据不能正常处理时, 临时保存数据 */
    struct api_msg_msg *write_msg;

    /* 数据不能正常处理时, 指明临时保存的数据已经有多少被发送了 */
    size_t write_offset;
#endif /* LWIP_TCP */

    /* 与该连接相关的回调函数 */
    netconn_callback callback;
};

```

由上述代码可以看到, 数据结构 `netconn` 描述了应用程序要使用 API 函数建立一个连接的各种属性。包括了连接的类型、最近的故障代码、回调函数等。

接下来简单介绍一下 LwIP 为应用程序提供的 API 接口函数。

```

struct netconn *netconn_new_with_proto_and_callback(
    enum netconn_type t,
    u8_t proto,
    netconn_callback callback);

```

`netconn_new_with_proto_and_callback()` 函数用于建立一个新的 `netconn` 连接。形参 `t` 是一个连接类型的变量, 表明了该连接的类型。 `proto` 指定了该连接使用的协议类型。 `callback` 则是该连接调用的回调函数。该函数首先调用 `netconn_alloc` 函数分配并初始化一个 `netconn` 结构。初始化的过程包括设置 `netconn` 类型字段, 同时为该结构的 `op_completed` 创建一个信号量, 为 `recvmbx` 创建一个接收邮箱。接下来该函数会构建一个 `api_msg` 消息, 该消息要求内核执行函数 `do_newconn`, 最后调用函数 `tcpip_apimsg` 来将消息包装成 `tcpip_msg` 结构并发送出去。 `tcpip_thread` 函数解析该消息并调用函数 `do_newconn`, `do_newconn` 根据参数的类型调用函数 `tcp_new` 创建一个 TCP 控制块。函数 `tcpip_apimsg` 会阻塞在一个信号量上, 直至 `do_newconn` 释放该信号量。

```
err_t netconn_delete(struct netconn *conn);
```

`netconn_delete()` 函数用来删除 `netconn` 所指向的连接。与 `netconn_new_with_proto_and_callback()` 的流程相同, 它通过将 `function` 指向函数 `do_delconn`, 而 `do_delconn` 调用 `tcp_close` 函数来关闭 TCP 连接。然后 `netconn_delete` 调用 `netconn_free` 函数释放 `netconn` 结构所占用的内存空间。

```

err_t netconn_getaddr(struct netconn *conn,
    struct ip_addr *addr,
    u16_t *port,
    u8_t local);

```

`netconn_getaddr()` 函数用于获取 `conn` 连接的远程主机 IP 地址及端口号。

```
err_t netconn_bind(struct netconn *conn,
```

```
struct ip_addr *addr,
u16_t port);
```

`netconn_bind()` 函数用于将一个 IP 地址及端口号与 `conn` 指向的连接绑定。同样，该函数是通过 `function` 指向的函数 `do_bind` 调用 `tcp_bind` 完成相应 TCP 控制块的绑定工作的。

```
err_t netconn_connect(struct netconn *conn,
struct ip_addr *addr,
u16_t port);
```

`netconn_connect()` 函数用于将服务器端的 IP 地址和端口号与 `conn` 指向的连接绑定。同样，该函数是通过 `function` 指向的函数 `do_connect` 调用 `tcp_connect` 完成相应的工作的。

```
err_t netconn_disconnect(struct netconn *conn);
```

`netconn_disconnect()` 函数用于断开 `conn` 指向的连接。

```
err_t netconn_listen_with_backlog(struct netconn *conn, u8_t backlog);
```

`netconn_listen_with_backlog()` 函数用于将 `conn` 指向的连接设定为监听状态。

```
struct netconn *netconn_accept(struct netconn *conn);
```

`netconn_accept()` 函数用于接收客户端的连接，该函数主要阻塞在 `acceptmbox` 邮箱上，当接收到新的连接后，在该邮箱上取下连接的 `netconn` 结构并返回。

```
struct netbuf * netconn_recv(struct netconn *conn);
```

`netconn_recv()` 函数用于接收数据，接收到的数据被封装为 `netbuf` 结构。在这里会由协议栈调用内核函数 `tcp_recved`，以通知内核数据被正常接收。

```
err_t netconn_sendto(struct netconn *conn,
struct netbuf *buf,
struct ip_addr *addr,
u16_t port);
```

`netconn_sendto()` 函数用于向一个指定的 IP 地址和端口号发送数据。值得注意的是，这个函数只能用在 `conn` 类型为 UDP 或 RAW 的连接中。该函数内部是通过调用函数 `netconn_send` 来实现发送数据的功能的。

```
err_t netconn_write(struct netconn *conn,
const void *dataptr,
size_t size,
u8_t apiflags);
```

`netconn_write()` 函数用于向相应的 TCP 连接上发送数据，这个函数只用于发送 TCP 的报文，不发送 UDP 或者 RAW 连接类型的报文。

```
err_t netconn_close(struct netconn *conn);
```

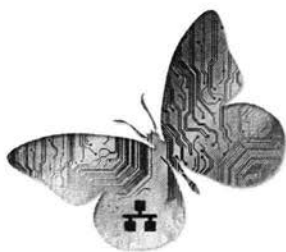
`netconn_close()` 函数用于关闭 `conn` 指向的连接。

在以上各 API 函数里，均有这样一句代码：

```
TCPIP_APIMSG(&msg);
```

这句代码的功能就是调用函数 `tcpip_apimsg` 或者 `tcpip_apimsg_lock`，用来将消息包装成 `tcpip_msg` 结构并发送给内核进程 `tcpip_thread`。如果宏 `LWIP_TCPIP_CORE_LOCKING` 被定义，则使用 `tcpip_apimsg_lock`，否则使用 `tcpip_apimsg`。这两个函数完成的功能相同，只是 `tcpip_apimsg_lock` 使用信号量机制来实现，而 `tcpip_apimsg` 则使用邮箱机制来实现。

最后需要说明的是，函数 `tcpip_thread` 是处理 TCP/IP 的内核协议栈进程，它只接收 `tcpip_msg` 结构封装的消息，并根据消息的类型来判定该消息来自物理网卡或应用层程序。如果接收到网卡的 IP 报文，则将该报文递交给 `ip_input` 函数，完成相应的处理。如果是应用层程序发送的消息，则通过调用消息指定的内核处理函数来完成相应的功能。



第 11 章

基于 STM32F107 的 LwIP 移植

本章将开始 LwIP 在 STM32F107 上的移植之旅。LwIP 的官方源代码中并没有提供对任何芯片的底层驱动移植程序，因此这部分程序需要由开发者自己完成。

LwIP 在 STM32F107 上的底层驱动移植工作主要包含两方面：一是修改文件 `ethernetif.c`，该文件是连接 LwIP 协议栈和 STM32F107 网络驱动程序的桥梁，LwIP 协议栈为开发者提供了 `ethernetif.c` 的程序模板；二是编写 STM32F107 的网络驱动程序，实现网络底层的初始化、收发报文功能。接下来将对以上两部分的工作做详细描述。

11.1 ethernetif.c 文件的移植

`ethernetif.c` 是 LwIP 协议栈和 STM32F107 网络驱动程序之间的接口，它主要包含 `ethernetif_init`、`ethernetif_input`、`low_level_init`、`low_level_input`、`low_level_output` 等函数，接下来将从 `ethernetif_init` 函数开始，逐一实现这些函数。

11.1.1 ethernetif_init 函数

这个函数的源代码在 10.3 节已经详细描述过，这里就不再给出该函数的源代码了。总的来说，它是 LwIP 底层网络接口的初始化函数，指定了网络接口 `netif` 对应的主机名及网卡描述，并指定了该网卡的 MAC 地址。同时，该函数还指定了 `netif` 的发送数据报文函数，并调用了网络底层驱动初始化函数 `low_level_init` 对网络底层进行初始化。

11.1.2 low_level_init 函数

这个函数是网卡的初始化函数。虽然该函数的代码量不大，但是它完成的工作量却不少。其源代码如下所示：

```
static void low_level_init(struct netif *netif)
{
    /* 设定 MAC 地址长度，默认是 6 字节 */
    netif->hwaddr_len = ETHARP_HWADDR_LEN;

    /* 设定网卡 MAC 地址 */
    netif->hwaddr[0] = MACAddr[0];
```

```

netif->hwaddr[1] = MACAddr[1];
netif->hwaddr[2] = MACAddr[2];
netif->hwaddr[3] = MACAddr[3];
netif->hwaddr[4] = MACAddr[4];
netif->hwaddr[5] = MACAddr[5];

/* 设定网卡每一包最大的发送数据字节数 */
netif->mtu = 1500;

/* 指定网卡具备广播、ARP 等功能 */
netif->flags = NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP |
              NETIF_FLAG_LINK_UP;

/* 初始化 DMA 发送描述符链表 */
ETH_DMATxDscChainInit(DMATxDscrTab,
                      &Tx_Buff[0][0],
                      ETH_TXBUFNB);
/* 初始化 DMA 接收描述符链表 */
ETH_DMARxDscChainInit(DMARxDscrTab,
                      &Rx_Buff[0][0],
                      ETH_RXBUFNB);

/* 使能接收中断 */
{
    int i;
    for(i=0; i<ETH_RXBUFNB; i++)
    {
        ETH_DMARxDscReceiveITConfig(&DMARxDscrTab[i],
        ENABLE);
    }
}

#ifdef CHECKSUM_BY_HARDWARE
/* 发送数据帧时, 硬件生成 CRC 校验字节 */
{
    int i;
    for(i=0; i<ETH_TXBUFNB; i++)
    {
        ETH_DMATxDscChecksumInsertionConfig(&DMATxDscrTab[i],
        ETH_DMATxDsc_ChecksumTCPUDPICMPFull);
    }
}
#endif

/* 启动 STM32F107 的以太网控制器相关功能 */
ETH_Start();
}

```

low_level_init 函数设定了网卡的物理地址和每帧最大传输数据字节数, 并对发送和接收使用的 DMA 描述符链表进行了初始化。而该函数最为重要的地方是调用了函数 ETH_Start(), 该函数使得 STM32F107 的以太网控制器处于工作状态, 并启动了以太网 DMA 的发送和接收功能。关于

函数 ETH_Start() 的详细描述将在后面内容中进一步展开。

11.1.3 ethernetif_input 函数

该函数用于从底层物理网卡读取报文，并将该报文向上传递给 LwIP 协议栈函数 ethernet_input 进行处理。ethernetif_input 的源代码如下所示：

```
err_t ethernetif_input(struct netif *netif)
{
    err_t err;
    struct pbuf *p;

    /* 将收到的报文拷贝至一个新的 pbuf 中 */
    p = low_level_input(netif);

    /* 若未读取到报文则返回 */
    if (p == NULL)
        return ERR_MEM;
    err = netif->input(p, netif);
    if (err != ERR_OK)
    {
        LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input
            error\n"));
        pbuf_free(p);
        p = NULL;
    }
    return err;
}
```

11.1.4 low_level_input 函数

该函数用于从内存中申请一个新的 pbuf，并把接收到的数据报文内容拷贝至该 pbuf 中。其源代码如下所示：

```
static struct pbuf * low_level_input(struct netif *netif)
{
    struct pbuf *p, *q;
    u16_t len;
    int l = 0;
    FrameTypeDef frame;
    u8 *buffer;

    p = NULL;

    /* 接收以太网控制器中接收到的报文 */
    frame = ETH_RxPkt_ChainMode();

    /* 获取报文长度 */
    len = frame.length;
```

```

/* 获取该报文的内存首地址 */
buffer = (u8 *)frame.buffer;

/* 从内存池中申请一个 pbuf */
p = pbuf_alloc(PBUF_RAW, len, PBUF_POOL);
if (p != NULL)
{
    for (q = p; q != NULL; q = q->next)
    {
        memcpy( (u8_t*)q->payload,
                (u8_t*)&buffer[l],
                q->len);
        l = l + q->len;
    }
}
/* 设置接收描述符 (Rx descriptor) 的状态 */
frame.descriptor->Status = ETH_DMARxDesc_OWN;

/* 当 DMA 的接收报文缓冲区标志 ETH_DMASR_RBUS 被置位的时候, 重新使
能接收 */
if ((ETH->DMASR & ETH_DMASR_RBUS) != (u32)RESET)
{
    ETH->DMASR = ETH_DMASR_RBUS;

/* 重新使能 DMA 接收报文 */
ETH->DMARPDR = 0;
}
return p;
}

```

在本函数中, FrameTypeDef 是一个结构体变量, 其定义如下:

```

typedef struct
{
    /* 接收到的数据报文的长度 */
    u32 length;

    /* 接收到的数据报文数据缓冲区地址 */
    u32 buffer;

    /* 以太网 DMA 描述符变量 */
    ETH_DMADESCTypeDef *descriptor;
}FrameTypeDef;

```

函数 ETH_RxPkt_ChainMode 则是 STM32F107 的以太网驱动程序的接收报文函数, 该函数将 DMA 接收控制器中的数据报文拷贝至变量 FrameTypeDef 所指向的缓冲区中, 最终放入 pbufs 指向的 payload 区域。关于函数 ETH_RxPkt_ChainMode 的详细描述将在后面内容中展开。

11.1.5 low_level_output 函数

该函数的功能是将 pbuf 中的数据帧通过底层发送函数 ETH_TxPkt_ChainMode 发送出去。由

于要发送的数据可能被分割在多个 pbuf 中，而这些 pbuf 通过 pbuf->next 指针连接起来，因此 low_level_output 函数需要使用 for 循环将这些 pbufs 中的数据拷贝至当前的发送缓冲区中。其函数源代码如下：

```
static err_t low_level_output(struct netif *netif,
struct pbuf *p)
{
    struct pbuf *q;
    int l = 0;
    u8 *buffer = (u8 *)ETH_GetCurrentTxBuffer();
    for(q = p; q != NULL; q = q->next)
    {
        /* 将 pbuf 中的数据拷贝至发送缓存 */
        memcpy((u8_t*)&buffer[l], q->payload, q->len);
        l = l + q->len;
    }
    ETH_TxPkt_ChainMode(l);
    return ERR_OK;
}
```

至此，ethernetif.c 文件的移植工作就完成了，由以上的移植过程可以看到，开发者的工作主要是修改 LwIP 协议栈提供的几个框架函数，如 ethernetif_init 等。

11.2 网络驱动移植

LwIP 在 STM32F107 上移植的另一个重要内容是编写 STM32F107 的网络驱动程序。在本书中，stm32_ethdrv.c、stm32_ethdrv.h 是 STM32F107 的网络驱动程序的源代码，下面将对网络驱动程序中的一些重要函数做一个详细阐述，具体的源代码可参见随书光盘。

11.2.1 以太网控制器概述

STM32F107 的以太网模块由 MAC 802.3（介质访问控制器）、独立于介质的接口（MII/RMII）管理模块和一个专用的 DMA 控制器组成。

MAC 模块实现了局域网 CSMA/CD 的子层，支持 10Mb/s 和 100Mb/s 数据传输率，而且支持全双工和半双工两种操作模式。

DMA 模块具有独立的发送和接收控制器，还有 1 组控制和状态寄存器。其发送控制器负责把数据从系统存储器转送至发送 FIFO，而接收控制器负责把数据从接收 FIFO 读出到系统存储器。DMA 控制器利用描述符来实现数据从源端到目的端之间的移动，从而减轻了 CPU 的负担，避免了 CPU 的负荷过重。DMA 和 CPU 之间的通信通过两种数据结构实现：控制和状态寄存器（ETH_MACCCR 和 ETH_MACCSR）、描述符列表和数据缓存。

关于以太网控制器硬件部分的描述可参考前面章节的相关内容。

11.2.2 以太网控制器硬件配置

函数 Ethernet_Configuration 是 STM32F107 的硬件配置函数，其源代码如下：

```
void Ethernet_Configuration(void)
{
    ETH_InitTypeDef  ETH_InitStructure;

    /* 选择 MII 或 RMII 接口方式 */
#ifdef MII_MODE
    GPIO_ETH_MediaInterfaceConfig(GPIO_ETH_MediaInterface_MII);

    /* 在 PA8 引脚输出 25MHz 频率 */
    RCC_MCOConfig(RCC_MCO_HSE);
#elif defined RMII_MODE
    GPIO_ETH_MediaInterfaceConfig(GPIO_ETH_MediaInterface_RMII);

    /* 使用锁相环技术在 PLL3 引脚输出 50MHz 频率 */
    RCC_PLL3Config(RCC_PLL3Mul_10);

    /* 使能 PLL3 */
    RCC_PLL3Cmd(ENABLE);

    /* Wait till PLL3 is ready */
    while (RCC_GetFlagStatus(RCC_FLAG_PLL3RDY) == RESET)
    {
    }

    /* Get PLL3 clock on PA8 pin (MCO) */
    RCC_MCOConfig(RCC_MCO_PLL3CLK);
#endif

    /* 在以太网总线上复位以太网控制器外设模块 */
    ETH_DeInit();

    /* MAC 的 DMA 控制器复位 MAC 所有子系统的内部寄存器和逻辑电路 */
    ETH_SoftwareReset();

    /* 等待复位完成 */
    while (ETH_GetSoftwareResetStatus() == SET);

    /* 初始化 ETH_InitStructure */
    ETH_StructInit(&ETH_InitStructure);

    /* 设置以太网控制器的具体参数 */
    /* 开启自适应功能 */
    ETH_InitStructure.ETH_AutoNegotiation =
        ETH_AutoNegotiation_Enable ;

    /* 关闭环路功能 */
    ETH_InitStructure.ETH_LoopbackMode =
        ETH_LoopbackMode_Disable;
```

```

/* 不适用重复传输功能 */
ETH_InitStructure.ETH_RetryTransmission =
    ETH_RetryTransmission_Disable;

/* 自动填充 CRC */
ETH_InitStructure.ETH_AutomaticPadCRCStrip =
    ETH_AutomaticPadCRCStrip_Disable;

/* 不接收所有报文 */
ETH_InitStructure.ETH_ReceiveAll = ETH_ReceiveAll_Disable;

/* 使能广播报文接收 */
ETH_InitStructure.ETH_BroadcastFramesReception =
    ETH_BroadcastFramesReception_Enable;

/* 当变量 ETH_PromiscuousMode 为 1 时，无论接收到帧的目的地址和源地址是
   什么，所有的帧都能通过地址过滤器，否则使用地址过滤器检查接收帧的物理地址
   */
ETH_InitStructure.ETH_PromiscuousMode =
    ETH_PromiscuousMode_Disable;

/* 对接收到的多播帧设定过滤模式 */
ETH_InitStructure.ETH_MulticastFramesFilter =
    ETH_MulticastFramesFilter_Perfect;

/* 对接收到的单播帧设定过滤模式 */
ETH_InitStructure.ETH_UnicastFramesFilter =
    ETH_UnicastFramesFilter_Perfect;

/* 硬件使能 CRC 校验功能 */
#ifdef CHECKSUM_BY_HARDWARE
    ETH_InitStructure.ETH_ChecksumOffload =
        ETH_ChecksumOffload_Enable;
#endif

/* 以下是与 DMA 相关的配置 */
/* 当收到的 TCP/IP 帧仅有校验和错误时，该变量用于表示是否丢弃该帧，1：
   不丢弃；0：丢弃 */
ETH_InitStructure.ETH_DropTCPChecksumErrorFrame =
    ETH_DropTCPChecksumErrorFrame_Enable;

/* 当一个数据帧被完全写入接收 FIFO 后，DMA 控制器才会把它转发给应用程序 */
ETH_InitStructure.ETH_ReceiveStoreForward =
    ETH_ReceiveStoreForward_Enable;

/* 当一个数据帧被完全写入发送 FIFO 后，DMA 控制器才会把它转发给应用程序 */
ETH_InitStructure.ETH_TransmitStoreForward =
    ETH_TransmitStoreForward_Enable;

/* 接收时丢弃所有错误帧 */
ETH_InitStructure.ETH_ForwardErrorFrames =
    ETH_ForwardErrorFrames_Disable;

/* 接收时丢弃所有长度小于 64 字节的帧 */

```

```

ETH_InitStructure.ETH_ForwardUndersizedGoodFrames =
    ETH_ForwardUndersizedGoodFrames_Disable;

/*DMA 发送控制器在接收到前一个帧的发送状态信息前, 就开始发送下一个帧的数据 */
ETH_InitStructure.ETH_SecondFrameOperate =
    ETH_SecondFrameOperate_Enable;

/*AHB 接口对齐所有突发传输至起始地址的最低位 */
ETH_InitStructure.ETH_AddressAlignedBeats =
    ETH_AddressAlignedBeats_Enable;

/*AHB 接口可以进行固定长度的突发传输 */
ETH_InitStructure.ETH_FixedBurst = ETH_FixedBurst_Enable;
ETH_InitStructure.ETH_RxDMABurstLength =
    ETH_RxDMABurstLength_32Beat;
ETH_InitStructure.ETH_TxDMABurstLength =
    ETH_TxDMABurstLength_32Beat;

/*DMA 控制器采用轮询的方式对 DMA 发送和接收通道同时访问 AHB 主接口时进行
仲裁, 且接收和发送优先级比例为 2:1*/
ETH_InitStructure.ETH_DMAArbitration =
    ETH_DMAArbitration_RoundRobin_RxTx_2_1;

/* 调用配置函数, 配置以上设置 */
ETH_Init(&ETH_InitStructure, PHY_ADDRESS);

/* 使能接收中断 */
ETH_DMAITConfig(ETH_DMA_IT_NIS | ETH_DMA_IT_R, ENABLE);
}

```

该函数完成了对 STM32F107 的基本硬件配置, 包括 MII/RMII 接口选择、自适应功能选择、硬件 CRC 功能、DMA 的相关配置等。在本开发板中, STM32F107 使用 MII 接口与 PHY 芯片 DP83848 通信, 因此宏 MII_MODE 被定义。在该函数中, 结构体 ETH_InitTypeDef 描述了与以太网控制器相关的各变量, 它与以太网控制器中的相关寄存器一一对应, 关于这个结构体的定义说明, 读者可参考头文件 stm32_ethdrv.h。对 ETH_InitTypeDef 中各变量的配置, 并不能直接配置 STM32F107 的以太网控制器的相关功能。若要使得对 ETH_InitTypeDef 中各变量的配置生效, 需要通过调用函数 ETH_Init 实现。该函数同样位于文件 stm32_ethdrv.c 中, 其函数原型如下:

```

uint32_t ETH_Init(ETH_InitTypeDef* ETH_InitStruct,
    uint16_t PHYAddress);

```

其中, 形参 ETH_InitStruct 是 ETH_InitTypeDef 类型的变量, 包含了对以太网控制模块的相关配置, PHYAddress 则是 PHY 芯片的 PHY 地址。在本开发板中, PHY 芯片是 DP83848, 其 PHY 地址是 1, 因此传入函数 ETH_Init 的宏定义 PHY_ADDRESS 值为 1。

11.2.3 以太网控制器硬件的引脚配置

STM32F107 的 GPIO 接口有很多引脚是功能复用的, 以 PA2 为例, 它既可以作为通用输入输出引脚, 也可以作为以太网控制器的 MDO 引脚。表 11.1 列出了使用 MII 接口时, 以太网控制信号与 GPIO 引脚的对应情况。

表 11.1 以太网控制信号的 GPIO 引脚分配

以太网控制器信号	GPIO引脚
ETH_MII_MDO	PA2
ETH_MII_MDC	PC1
ETH_MII_TX_EN	PB11
ETH_MII_TX_CLK	PC3
ETH_MII_TXD0	PB12
ETH_MII_TXD1	PB13
ETH_MII_TXD2	PC2
ETH_MII_TXD3	PB8
ETH_MII_RX_DV	PD8
ETH_MII_RX_CLK	PA1
ETH_MII_RX_ER	PB10
ETH_MII_RXD0	PD9
ETH_MII_RXD1	PD10
ETH_MII_RXD2	PD11
ETH_MII_RXD3	PD12

函数 EthIO_Config 实现了对以太网控制器信号引脚的配置, 源代码如下所示。其中, GPIO_InitStructure、GPIO_Init 是 ST 提供的外设驱动库里的变量和函数。读者可以在 ST 的官方网站下载最新的外设驱动库源代码。

```
void EthIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

```

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_8 |
                                GPIO_Pin_11 | GPIO_Pin_12 |
                                GPIO_Pin_13;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOB, &GPIO_InitStructure);

/* 将相关的 GPIO 作为以太网控制器信号 */
GPIO_PinRemapConfig(GPIO_Remap_ETH, ENABLE);
/* Configure PA0, PA1 and PA3 as input */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 |
                                GPIO_Pin_3;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOB, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOC, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 |
                                GPIO_Pin_10 | GPIO_Pin_11 |
                                GPIO_Pin_12;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/* 将 PA8 配置为 MCO 输出引脚, 在实际电路中, 该引脚被用作 ETH_MII_TX_CLK 和 ETH_MII_RX_CLK */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStructure);
}

```

11.2.4 以太网驱动之接收

在介绍函数 `low_level_input` 时, 曾简要介绍过函数 `ETH_RxPkt_ChainMode` 的基本功能: 将 DMA 接收控制器中的数据报文拷贝至变量 `FrameTypeDef` 所指向的缓冲区中。本节将通过解析其源代码的方式来详细看一下该函数是如何接收报文的。其函数源代码如下:

```

FrameTypeDef ETH_RxPkt_ChainMode(void)
{
    u32 framelength = 0;

```

```

FrameTypeDef frame = {0,0};

/* 检查接收到的数据帧是否依然被 DMA 占用 */
if ((DMARxDescToGet->Status & ETH_DMARxDesc_OWN) != (u32)RESET)
{
    frame.length = ETH_ERROR;

    /* 当 DMA 的接收报文缓冲区标志 ETH_DMASR_RBUS 被置位的时候, 表明当前接收缓冲区不可用, 需要重新使能接收 */
    if ((ETH->DMASR & ETH_DMASR_RBUS) != (u32)RESET)
    {
        ETH->DMASR = ETH_DMASR_RBUS;

        /* 重新使能 DMA 接收报文 */
        ETH->DMARPDR = 0;
    }
    return frame;
}

/* DMA 接收控制器接收报文完毕, CPU 可以读取 DMA 缓冲区里的报文 */
if (((DMARxDescToGet->Status & ETH_DMARxDesc_ES) == (u32)RESET)
&& ((DMARxDescToGet->Status & ETH_DMARxDesc_LS) != (u32)RESET)
&& ((DMARxDescToGet->Status & ETH_DMARxDesc_FS) != (u32)RESET))
{
    /* 获取报文长度, 该报文长度不含有 4 字节 CRC 校验字节 */
    framelength = ((DMARxDescToGet->Status & ETH_DMARxDesc_FL)
        >> ETH_DMARxDesc_FrameLengthShift) - 4;

    /* 获取报文缓冲区的地址 */
    frame.buffer = DMARxDescToGet->Buffer1Addr;
}
else
{
    /* 返回故障 */
    framelength = ETH_ERROR;
}
frame.length = framelength;
frame.descriptor = DMARxDescToGet;

/* 更新 DMA 接收缓冲区 */
DMARxDescToGet = (ETH_DMADESCTypeDef*) (DMARxDescToGet->
    Buffer2NextDescAddr);

/* 返回报文数据帧结构 */
return (frame);
}

```

11.2.5 以太网驱动之发送

函数 ETH_TxPkt_ChainMode 是 STM32F107 以太网驱动的发送函数。当 ETH_DMATxDesc_OWN 位被清零时, 说明 CPU 已经将要发送的数据帧存放在发送缓冲区中, DMA 发送控制器可以

发送该缓冲区中的数据。其源代码如下：

```
u32 ETH_TxPkt_ChainMode(u16 FrameLength)
{
    /* 检查该报文是否已传送到 DMA 的 RAM 区域 */
    if((DMATxDescToSet->Status & ETH_DMATxDesc_OWN) != (u32)RESET)
    {
        /* 如果 CPU 尚未释放控制权，则返回错误 */
        return ETH_ERROR;
    }

    /* 设定发送报文数据长度：位 [12:0] */
    DMATxDescToSet->ControlBufferSize = (FrameLength &
                                          ETH_DMATxDesc_TBS1);

    /* 在一个发送描述符里发送一帧数据 */
    DMATxDescToSet->Status |= ETH_DMATxDesc_LS |
                          ETH_DMATxDesc_FS;

    /* 触发发送 */
    DMATxDescToSet->Status |= ETH_DMATxDesc_OWN;

    /* 检查发送缓冲区是否可以发送数据 */
    if ((ETH->DMASR & ETH_DMASR_TBUS) != (u32)RESET)
    {
        ETH->DMASR = ETH_DMASR_TBUS;
    }

    /* 重新启动发送 */
    ETH->DMATPDR = 0;
    DMATxDescToSet = (ETH_DMADESCTypeDef*) (DMATxDescToSet->
                                              Buffer2NextDescAddr);

    /* 返回成功标志 */
    return ETH_SUCCESS;
}
```

11.2.6 其他注意事项

在 low_level_init 中，有 3 个函数的调用，值得引起读者的注意，分别是 ETH_Start、ETH_DMARxDescChainInit、ETH_DMATxDescChainInit。本节内容将对这几个函数做进一步阐述。

首先来看看 ETH_Start 函数，该函数完成了如下功能：使能 STM32F107 的 MAC 控制器在 MII 总线上的发送和接收功能；使能 DMA 控制器的发送和接收功能。其源代码如下：

```
void ETH_Start(void)
{
    /* 使能 STM32F107 的 MAC 控制器在 MII 总线上的发送功能 */
    ETH_MACTransmissionCmd(ENABLE);

    /* 清空发送 FIFO */
}
```

```

    ETH_FlushTransmitFIFO();

    /* 使能 STM32F107 的 MAC 控制器在 MII 总线上的接收功能 */
    ETH_MACReceptionCmd(ENABLE);

    /* 启动 DMA 发送控制器 */
    ETH_DMATransmissionCmd(ENABLE);

    /* 启动 DMA 接收控制器 */
    ETH_DMAReceptionCmd(ENABLE);
}

```

其次是函数 ETH_DMARxDescChainInit。本书在介绍移植 STM32F107 网络驱动程序的时候，DMA 描述符采用了“链结构”。因此函数 ETH_DMARxDescChainInit 的功能就是对链结构的 DMA 接收描述符进行初始化。其源代码如下：

```

void ETH_DMARxDescChainInit(ETH_DMADESCTypeDef *DMARxDescTab,
                             uint8_t *RxBuff, uint32_t RxBuffCount)
{
    uint32_t i = 0;
    ETH_DMADESCTypeDef *DMARxDesc;

    /* 将全局变量指针 DMARxDescToGet 指向 DMA 接收描述符列表 DMARxDescTab 的基地址 */
    DMARxDescToGet = DMARxDescTab;
    for(i=0; i < RxBuffCount; i++)
    {
        /* 将 DMA 接收描述符列表 DMARxDescTab 中每个元素的首地址赋值给 DMARxDesc */
        DMARxDesc = DMARxDescTab+i;

        /* 该描述符被 DMA 占用 */
        DMARxDesc->Status = ETH_DMARxDesc_OWN;
        DMARxDesc->ControlBufferSize = ETH_DMARxDesc_RCH |
            (uint32_t)ETH_MAX_PACKET_SIZE;

        /* 将接收描述符缓冲区 1 的地址指向接收缓存 RxBuff */
        DMARxDesc->Buffer1Addr = (uint32_t)(&RxBuff[i*ETH_MAX_PACKET_SIZE]);

        /* 将 DMA 接收描述符形成“链结构” */
        if(i < (RxBuffCount-1))
        {
            DMARxDesc->Buffer2NextDescAddr = (uint32_t)(DMARxDescTab+i+1);
        }
        else
        {
            DMARxDesc->Buffer2NextDescAddr = (uint32_t)(DMARxDescTab);
        }
    }

    /* 将 DMA 接收描述符列表的基地址写入寄存器 DMARDLAR */
    ETH->DMARDLAR = (uint32_t) DMARxDescTab;
}

```

最后是函数 ETH_DMATxDescChainInit，该函数的源代码与 ETH_DMARxDescChainInit 基本相似，这里就不再详细阐述了。总的说来，该函数对链结构的 DMA 发送描述符进行了初始化。

至此，LwIP 在 STM32F107 上的移植工作就已经完成了，关于源代码的一些详细信息，读者可以参阅本书附带的光盘文件。

11.3 基于 RAW API 接口的 HelloWorld 例程

在介绍 HelloWorld 例程之前，首先讨论如下内容：在基于 Raw API 接口方式下，如何实现对 LwIP 定时器的调用以及如何在应用程序中实现报文的接收。

首先介绍对 LwIP 定时器的调用方式。在 Cortex-M3 内核的 ARM 芯片中，有一个 24 位的系统定时器，简称为 SysTick。该定时器通常用于产生一个周期性的中断，供应用程序调用。在本节所阐述的 HelloWorld 例程中，也使用了该定时器，其主要作用是作为 LwIP 协议栈的定时器管理者，用于实现周期性调用 tcp_tmr 和 etharp_tmr。接下来看看对 SysTick 定时器的配置，其代码如下：

```
// 获取 STM32F107 中各系统时钟变量值
CC_GetClocksFreq(&RCC_Clocks);

// 实现对 SysTick 的配置，其中断周期为 10ms
SysTick_Config(RCC_Clocks.SYSCLK_Frequency / 100);
```

由以上代码可以看到，SysTick 的中断频率为 10ms。其中，再看看 SysTick 对应的中断程序内容，其代码如下：

```
void SysTick_Handler(void)
{
    sysLocalTime += 10;
}
```

该中断服务程序非常简单，只完成了一件事情：实现对系统计时器变量 sysLocalTime 每 10ms 累加。

通过接下来的代码，读者将会明白为何程序中需要对 sysLocalTime 进行维护。

```
void LwIP_Periodic_Handle(__IO uint32_t localtime)
{
    /* TCP 的调度周期是 250 ms */
    if (localtime - TCPTimer >= TCP_TMR_INTERVAL)
    {
        TCPTimer = localtime;
        tcp_tmr();
    }

    /* ARP 的调度周期是 5s */
    if (localtime - ARPTimer >= ARP_TMR_INTERVAL)
    {

```

```

    ARPTimer = localtime;
    etharp_tmr();
}
}

```

函数 `LwIP_Periodic_Handle` 在主程序的 `while(1)` 循环中被调用。

```

int main(void)
{
    ...
    while (1)
    {
        LwIP_Periodic_Handle(sysLocalTime);
    }
}

```

函数 `LwIP_Periodic_Handle` 实时检测 `sysLocalTime` 与 `TCPTimer` 的差值，一旦大于宏 `TCP_TMR_INTERVAL` 定义的数值，则启动 LwIP 协议栈中 TCP 的定时器管理程序 `tcp_tmr`。同理，当 `sysLocalTime` 与 `ARPTimer` 的差值大于宏 `ARP_TMR_INTERVAL` 定义的数值时，将调用 ARP 定时器管理程序 `etharp_tmr`。

其次是如何实现以太网报文接收程序的调用。在前面对 `ethernetif.c` 文件的移植过程中，介绍了函数 `ethernetif_input`，该函数用于实现接收报文。那么该函数又是如何被调用的呢？请看下面的代码：

```

void ETH_IRQHandler(void)
{
    /* 检查接收的报文长度是否为 0 */
    while(ETH_GetRxPktSize() != 0)
    {
        /* 接收报文 */
        ethernetif_input(&netif);
    }

    /* 清除 DMA 的 RX 中断 */
    ETH_DMAClearITPendingBit(ETH_DMA_IT_R);
    ETH_DMAClearITPendingBit(ETH_DMA_IT_NIS);
}

```

函数 `ETH_IRQHandler` 是 STM32F107 以太网控制器的接收中断函数。在该中断服务程序中，通过调用 `ethernetif_input` 实现接收报文。

接下来揭开 HelloWorld 例程的庐山真面目。HelloWorld 例程基于 Telnet 协议实现。用户在 PC 机的命令行运行窗口中输入如下命令，以便与 STM32F107 开发板建立 Telnet 连接。

```
C:>Telnet 192.168.0.10
```

当成功建立连接后，命令行窗口将弹出 STM32F107 发出的如下消息：

```
"Hello. How do you do?"
```

此时，无论用户敲入任何字符，命令行窗口将弹出“Hello”的消息。

由此可以看到，这是一个很简单的例程，然而通过对该例程的介绍，读者将会了解如何使用 Raw API 接口方式，实现 LwIP 与应用程序的通信。

文件 HelloWorld.c 是 HelloWorld 例程的源文件，其实现代码如下：

```
#include "helloworld.h"
#include "lwip/tcp.h"
#include <string.h>

#define GREETING "Hello. How do you do?\r\n"
#define HELLO "Hello"
#define MAX_NAME_SIZE 128

struct name
{
    int length;
    char bytes[MAX_NAME_SIZE];
};

/* 该函数在接收到 Telnet 连接上的数据报文时被调用 */
static err_t HelloWorld_recv(void *arg,
                              struct tcp_pcb *pcb,
                              struct pbuf *p,
                              err_t err)
{
    struct pbuf *q;
    struct name *name = (struct name *)arg;
    int done;
    char *c;
    int i;

    if (p != NULL)
    {
        tcp_recved(pcb, p->tot_len);
        if(!name)
        {
            pbuf_free(p);
            return ERR_ARG;
        }
        done = 0;
        for(q=p; q != NULL; q = q->next)
        {
            c = q->payload;
            for(i=0; i<q->len && !done; i++)
            {
                done = ((c[i] == '\r') || (c[i] == '\n'));
                if(name->length < MAX_NAME_SIZE)
                {
                    name->bytes[name->length++] = c[i];
                }
            }
        }
    }
}
```

```

        if(done)
        {
            if(name->bytes[name->length-2] != '\r' ||
               name->bytes[name->length-1] != '\n')
            {
                if((name->bytes[name->length-1] == '\r' ||
                   name->bytes[name->length-1] == '\n') &&
                   (name->length+1 <= MAX_NAME_SIZE))
                {
                    name->length += 1;
                }
                else if(name->length+2 <= MAX_NAME_SIZE)
                {
                    name->length += 2;
                }
                else
                {
                    name->length = MAX_NAME_SIZE;
                }

                name->bytes[name->length-2] = '\r';
                name->bytes[name->length-1] = '\n';
            }
            tcp_write(pcb, HELLO, strlen(HELLO), 1);
            name->length = 0;
        }
        pbuf_free(p);
    }
    else if (err == ERR_OK)
    {
        mem_free(name);
        return tcp_close(pcb);
    }
    return ERR_OK;
}

static err_t HelloWorld_accept(void *arg,
                               struct tcp_pcb *pcb,
                               err_t err)
{
    tcp_arg(pcb, mem_malloc(sizeof(struct name), 1));
    tcp_err(pcb, HelloWorld_conn_err);
    tcp_recv(pcb, HelloWorld_recv);
    tcp_write(pcb, GREETING, strlen(GREETING), 1);

    return ERR_OK;
}

void HelloWorld_init(void)
{
    struct tcp_pcb *pcb;
    pcb = tcp_new();

```

```
tcp_bind(pcb, IP_ADDR_ANY, 23);
pcb = tcp_listen(pcb);
tcp_accept(pcb, HelloWorld_accept);
}

static void HelloWorld_conn_err(void *arg, err_t err)
{
    struct name *name;
    name = (struct name *)arg;
    mem_free(name);
}
```

HelloWorld_init 函数是本例程的初始化函数。它首先通过调用 tcp_bind 函数将申请的 pcb 绑定至端口 23（该端口是 Telnet 协议的通信端口），接下来通过函数 tcp_listen 实现在端口 23 上的监听。函数 tcp_accept 指定 HelloWorld_accept 函数作为 Telnet 连接成功建立后将要调用的回调函数。函数 HelloWorld_recv 在收到数据报文时被调用，它是本例程的核心。其基本功能是当接收到用户在命令行输入的字符后，返回“Hello”消息至用户的命令行窗口。

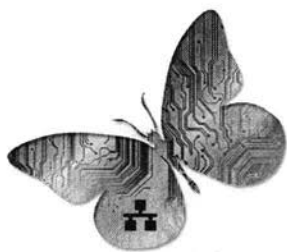
至此，LwIP 在 STM32F107 上的移植工作就已经完成了，关于源代码的一些详细信息，读者可以参阅本书附带的光盘文件。



第四篇 移植篇

本篇主要介绍 FreeRTOS 和 LwIP 的联合移植，以及基于此的一个具体项目。

- 第12章 基于FreeRTOS的LwIP协议栈移植
- 第13章 工业通信网关解析



第 12 章

基于 FreeRTOS 的 LwIP 协议栈移植

本章结合前面介绍的内容，介绍了在 STM32F107 嵌入式处理器上移植 FreeRTOS 和 LwIP 的全过程。

12.1 概述

如前所述，LwIP 协议栈与应用程序之间有三种接口方式：RAW API 接口、Sequential API 接口、BSD 兼容的 Socket 接口。RAW API 接口无需操作系统支持，在第 11 章中，本书已经介绍了如何将 LwIP 移植到无操作系统支持的 STM32F107 开发板上，并介绍了如何使用 RAW API 接口方式与应用程序通信。本节将讨论在使用 Sequential API 接口方式下，如何将 LwIP 移植到基于 FreeRTOS 操作系统的 STM32F107 开发板上。

图 12.1 描述了在 FreeRTOS 操作系统下的 LwIP 任务模型，同时该模型也是本书将 LwIP 移植到 FreeRTOS 操作系统上的一个基本程序框架结构。由该图可知：

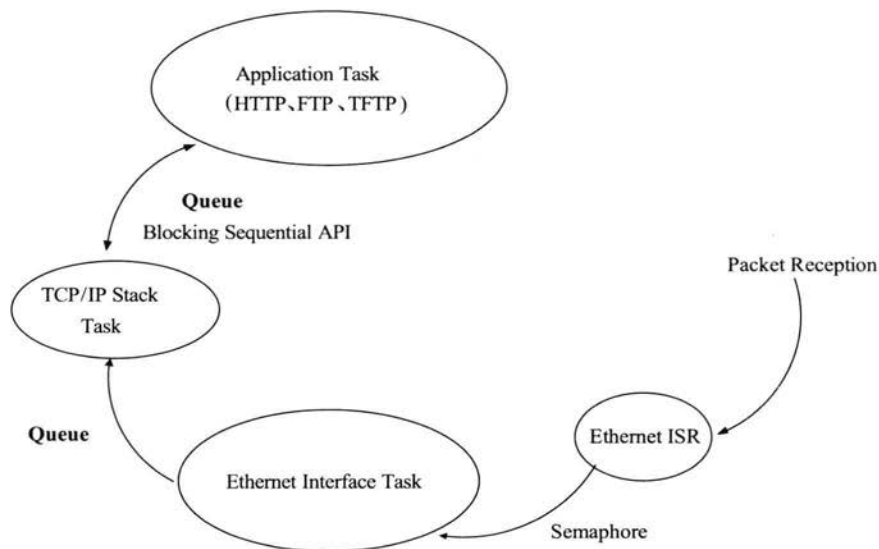


图 12.1 基于 FreeRTOS 的 LwIP 移植

- ❑ 以太网接口任务 (Ethernet Interface Task) 用于接收来自物理网卡的数据报文, 同时将收到的报文通过 FreeRTOS 提供的邮箱传递给 TCP/IP 协议栈任务。以太网接口任务平时处于挂起状态。当 STM32F107 收到报文时, 将产生接收报文中断 “Ethernet ISR”, 该中断以信号量的方式将以太网接口任务激活。
 - ❑ 应用程序使用 TCP/IP 协议栈提供的 Sequential API 接口访问 LwIP, 同时这两个独立的任务需要使用 FreeRTOS 提供的邮箱机制实现彼此之间信息的交互。值得注意的是, Sequential API 接口函数在 FreeRTOS 操作系统运行环境下是 “阻塞” 函数, 也就是说应用程序任务在调用 Sequential API 接口函数时, 将会被阻塞, 直到收到来自 TCP/IP 协议栈返回的消息应答。
 - ❑ 基于 LwIP 的 TCP/IP 协议栈与应用程序运行在两个独立的任务中。
- 接下来将根据图 12.1 提供的程序框架, 详细介绍如何将 LwIP 移植到 FreeRTOS 操作系统上。

12.2 FreeRTOS 下以太网驱动程序的移植

以太网驱动程序中关于物理网卡初始化、DMA 操作、报文接收和发送等函数在本书的第 11 章中已经做了详细阐述, 这部分驱动程序并不需要做改变, 需要改变的是以太网报文接收中断函数。其代码实现如下所示:

```
void ETH_IRQHandler(void)
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    /* 检查是否收到报文 */
    while(ETH_GetRxPktSize() != 0)
    {
        // 通知以太网接口任务接收报文
        xSemaphoreGiveFromISR( s_xSemaphore,
                               &xHigherPriorityTaskWoken );
    }
    /* 清除 DMA 的 RX 中断 */
    ETH_DMAClearITPendingBit(ETH_DMA_IT_R);
    ETH_DMAClearITPendingBit(ETH_DMA_IT_NIS);
    // 进行上下文切换, 确保最高优先级的阻塞任务可以执行
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
    }
}
```

该中断函数调用了 FreeRTOS 提供的中断信号量释放函数 xSemaphoreGiveFromISR, 因信号量 s_xSemaphore 阻塞的任务将被激活, 并得到执行。对于本移植程序而言, 以太网接口任务 ethernetif_input 将得到执行, 它将调用报文接收函数接收报文。

12.3 LwIP 程序移植

该部分的移植包含两个部分：以太网接口 ethernetif.c 的移植和操作系统模拟层 sys_arch.c 的移植。

与本书第 11 章介绍的不同，本章介绍的 ethernetif.c 文件将建立一个以太网接口任务，用于接收以太网数据报文。接下来将详细介绍 ethernetif.c 文件中各函数的移植过程。

12.3.1 以太网接口文件 ethernetif.c 的移植

1. low_level_init 函数

该函数的源代码如下：

```
static void low_level_init(struct netif *netif)
{
    /* 设定 MAC 地址长度，默认是 6 字节 */
    netif->hwaddr_len = ETHARP_HWADDR_LEN;
    /* 设定网卡 MAC 地址 */
    netif->hwaddr[0] = MACAddr[0];
    netif->hwaddr[1] = MACAddr[1];
    netif->hwaddr[2] = MACAddr[2];
    netif->hwaddr[3] = MACAddr[3];
    netif->hwaddr[4] = MACAddr[4];
    netif->hwaddr[5] = MACAddr[5];
    /* 设定网卡每一包最大的发送数据字节数 */
    netif->mtu = 1500;
    /* 指定网卡具备广播、ARP 等功能 */
    netif->flags = NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP |
        NETIF_FLAG_LINK_UP;

    s_pxNetIf = netif;

    /* 初始化 DMA 发送描述符链表 */
    ETH_DMATxDscrChainInit(DMATxDscrTab, &Tx_Buff[0][0],
        ETH_TXBUFNB);
    /* 初始化 DMA 接收描述符链表 */
    ETH_DMARxDscrChainInit(DMARxDscrTab, &Rx_Buff[0][0],
        ETH_RXBUFNB);

    /* 创建二进制信号量 s_xSemaphore，该信号量用于接收报文 */
    if (s_xSemaphore == NULL)
    {
        s_xSemaphore = xSemaphoreCreateCounting(20, 0);
    }
    /* 创建二进制信号量 xTxSemaphore，该信号量用于发送报文 */
    if (xTxSemaphore == NULL)
    {
        vSemaphoreCreateBinary(xTxSemaphore);
    }
}

/* 使能接收中断 */
```



```

    {
        p = low_level_input( s_pxNetIf );
        if (ERR_OK != s_pxNetIf->input( p, s_pxNetIf))
        {
            pbuf_free(p);
            p=NULL;
        }
    }
}

```

由以上代码可以看到，任务 ethernetif_input 平时处于阻塞状态，当信号量 s_xSemaphore 被以太网接收报文中断释放时，该任务得以执行。

3. low_level_input 函数

该函数无需做改动，读者可以参考本书第 11 章中关于该函数的移植代码。

4. low_level_output 函数

该函数的源代码如下所示。在本函数中，使用了二进制信号量 xTxSemaphore，这是因为该函数可能被多个任务调用，因此需要使用 xTxSemaphore 来保护 DMA 发送缓冲区的临界区资源。

```

static err_t low_level_output(struct netif *netif, struct pbuf
    *p)
{
    struct pbuf *q;
    uint32_t l = 0;
    u8 *buffer ;
    if (xSemaphoreTake(xTxSemaphore, 250))
    {
        buffer = (u8 *)ETH_GetCurrentTxBuffer();
        for(q = p; q != NULL; q = q->next)
        {
            /* 将 pbuf 中的数据拷贝至发送缓存 */
            memcpy((u8_t*)&buffer[l], q->payload, q->len);
            l = l + q->len;
        }
        ETH_TxPkt_ChainMode(l);
        xSemaphoreGive(xTxSemaphore);
    }
    return ERR_OK;
}

```

12.3.2 操作系统模拟层文件 sys_arch.c 的移植

接下来是操作系统模拟层 sys_arch.c 的移植。如前文所述，在 LwIP 中，操作系统模拟层是 LwIP 协议栈的一部分，它存在的目的是方便将 LwIP 移植到各种不同的操作系统上，它为操作系统和 LwIP 协议栈之间提供了一个接口桥梁，当用户移植 LwIP 到一个新的操作系统的时候，只需要修改操作系统模拟层内的各函数即可。对于如何实现操作系统模拟层，sys_ach.txt 文件给出了

详细说明。总的说来，操作系统模拟层主要完成了与信号量、消息邮箱机制、线程相关的功能，接下来将对 sys_arch.c、sys_arch.h 文件的代码实现进行更深层次的解释。

首先来看如下代码：

```
typedef xSemaphoreHandle sys_sem_t;
typedef xQueueHandle sys_mbox_t;
typedef xTaskHandle sys_thread_t;
```

在 LwIP 中，信号量使用变量 sys_sem_t 定义，队列消息使用变量 sys_mbox_t 定义，线程则使用变量 sys_thread_t 定义。因此为了将 LwIP 移植到 FreeRTOS 操作系统，在 sys_arch.h 文件中，需要使用 typedef 的方式对上述 3 个变量进行重定义。

其次，sys_arch.c 是信号量、消息队列机制、线程相关的功能的实现代码，本章接下来的内容将着重讨论 sys_arch.c 内的各函数实现。

1. sys_mbox_new 函数

该函数的功能是使用 FreeRTOS 提供的消息队列机制创建一个空的消息队列。在 FreeRTOS 中，消息队列创建函数是 xQueueCreate。创建的邮箱大小由 sys_arch.h 中的宏定义 archMESG_QUEUE_LENGTH 实现，在本书中，将其定义为 8。sys_mbox_new 的具体代码如下：

```
sys_mbox_t sys_mbox_new(int size)
{
    xQueueHandle mbox;

    ( void ) size;
    /* 创建队列 */
    mbox = xQueueCreate( archMESG_QUEUE_LENGTH, sizeof( void * ) );

    #if SYS_STATS
        ++lwip_stats.sys.mbox.used;
        if (lwip_stats.sys.mbox.max < lwip_stats.sys.mbox.used)
        {
            lwip_stats.sys.mbox.max = lwip_stats.sys.mbox.used;
        }
    #endif
    return mbox;
}
```

2. sys_mbox_free 函数

该函数的功能与 sys_mbox_new 相反，它用于删除一个队列。当该队列中还有未被取出的消息时，该函数应当报错，并通知应用程序。其代码如下：

```
void sys_mbox_free(sys_mbox_t mbox)
{
    if( uxQueueMessagesWaiting( mbox ) )
    {
        /* 报错 */
        portNOP();
        #if SYS_STATS
```

```

        lwip_stats.sys.mbox.err++;
    #endif
}
/* 删除该队列 */
vQueueDelete( mbox );
#if SYS_STATS
    --lwip_stats.sys.mbox.used;
#endif
}

```

3. sys_mbox_post 函数

该函数用于将消息发送至消息队列中。该函数是一个阻塞函数。当消息被发送至队列后，该函数才退出阻塞状态。其代码如下：

```

void sys_mbox_post(sys_mbox_t mbox, void *data)
{
    while ( xQueueSendToBack(mbox, &data, portMAX_DELAY ) !=
pdTRUE ){}
}

```

4. sys_mbox_trypost 函数

该函数用于尝试将某个消息发送至消息队列中，当消息被成功投递后，则返回成功，否则返回失败。

```

err_t sys_mbox_trypost(sys_mbox_t mbox, void *msg)
{
    err_t result;
    if ( xQueueSend( mbox, &msg, 0 ) == pdPASS )
    {
        result = ERR_OK;
    }
    else
    {
        // 不能成功投递，可能是队列满了
        result = ERR_MEM;
        #if SYS_STATS
            lwip_stats.sys.mbox.err++;
        #endif /* SYS_STATS */
    }
    return result;
}

```

5. sys_arch_mbox_fetch 函数

该函数用于从消息队列中取出一条消息。该函数是一个阻塞函数。调用该函数的线程若未取到消息，则在形参 timeout 所指定的时间内，该线程被阻塞。当超过 timeout 所指定的时间后，该线程恢复至就绪状态。若 timeout 为 0，则调用该函数的线程一直被阻塞，直到收到消息。

```

u32_t sys_arch_mbox_fetch(sys_mbox_t mbox, void **msg, u32_t
timeout)

```

```

{
    void *dummyptr;
    portTickType StartTime, EndTime, Elapsed;
    StartTime = xTaskGetTickCount();
    if ( msg == NULL )
    {
        msg = &dummyptr;
    }
    if ( timeout != 0 )
    {
        /* 在 timeout 时间内接收消息 */
        if ( pdTRUE == xQueueReceive( mbox, &(*msg), timeout /
                                      portTICK_RATE_MS ) )
        {
            EndTime = xTaskGetTickCount();
            Elapsed = (EndTime - StartTime) * portTICK_RATE_MS;
            return ( Elapsed );
        }
        else // 超时退出
        {
            *msg = NULL;
            return SYS_ARCH_TIMEOUT;
        }
    }
    else
    {
        /* 直到收到消息才退出阻塞状态 */
        while( pdTRUE != xQueueReceive( mbox, &(*msg),
                                         portMAX_DELAY ) )
        {
        }
        EndTime = xTaskGetTickCount();
        Elapsed = (EndTime - StartTime) * portTICK_RATE_MS;
        return ( Elapsed ); // return time blocked TODO test
    }
}

```

6. sys_arch_mbox_tryfetch 函数

该函数尝试从消息队列中取出消息。它是一个非阻塞函数。当取到消息时，则返回成功，否则立即退出，返回“队列空”。

```

u32_t sys_arch_mbox_tryfetch(sys_mbox_t mbox, void **msg)
{
    void *dummyptr;
    if ( msg == NULL )
    {
        msg = &dummyptr;
    }
    if ( pdTRUE == xQueueReceive( mbox, &(*msg), 0 ) )
    {
        return ERR_OK;
    }
}

```

```

    }
    else
    {
        return SYS_MBOX_EMPTY;
    }
}

```

7. sys_sem_new 函数

该函数用于创建一个信号量，其中形参 count 指明了当前信号量的状态。若 count 为 0，则该信号量在创建时就被“取走”(take)了。其代码如下：

```

sys_sem_t sys_sem_new(u8_t count)
{
    xSemaphoreHandle xSemaphore;
    /* 创建二进制信号量 */
    vSemaphoreCreateBinary( xSemaphore );
    if( xSemaphore == NULL )
    {
        #if SYS_STATS
            ++lwip_stats.sys.sem.err;
        #endif /* SYS_STATS */
        return SYS_SEM_NULL;
    }
    if(count == 0)
    {
        xSemaphoreTake(xSemaphore, 1);
    }
    #if SYS_STATS
        ++lwip_stats.sys.sem.used;
        if (lwip_stats.sys.sem.max < lwip_stats.sys.sem.used)
        {
            lwip_stats.sys.sem.max = lwip_stats.sys.sem.used;
        }
    #endif
    return xSemaphore;
}

```

8. sys_arch_sem_wait 函数

该函数是一个阻塞函数。调用该函数的线程在形参 timeout 指定的时间内被阻塞。若 timeout 为 0，则调用该函数的线程将一直被阻塞，直到等待的信号量被释放。当该函数取到信号量时，它将返回取得该信号量所占用的时间。其代码如下：

```

u32_t sys_arch_sem_wait(sys_sem_t sem, u32_t timeout)
{
    portTickType StartTime, EndTime, Elapsed;
    StartTime = xTaskGetTickCount();
    if(timeout != 0)
    {
        if( xSemaphoreTake( sem, timeout / portTICK_RATE_MS ) ==
            pdTRUE )

```

```

    {
        EndTime = xTaskGetTickCount();
        Elapsed = (EndTime - StartTime) * portTICK_RATE_MS;
        return (Elapsed);
    }
    else
    {
        return SYS_ARCH_TIMEOUT;
    }
}
else
{
    while( xSemaphoreTake( sem, portMAX_DELAY ) != pdTRUE ){}
    EndTime = xTaskGetTickCount();
    Elapsed = (EndTime - StartTime) * portTICK_RATE_MS;
    return ( Elapsed ); // return time blocked
}
}

```

9. sys_sem_signal 函数

该函数用于释放一个信号量。其代码如下：

```

void sys_sem_signal(sys_sem_t sem)
{
    xSemaphoreGive( sem );
}

```

10. sys_sem_free 函数

该函数用于删除一个信号量。其代码如下：

```

void sys_sem_free(sys_sem_t sem)
{
    #if SYS_STATS
        --lwip_stats.sys.sem.used;
    #endif
    vQueueDelete( sem );
}

```

11. sys_thread_new 函数

该函数用于创建一个新的线程。其中，形参 name 指定了该线程的名称，thread 是该线程对应的函数，arg 是该线程的形参，stacksize 指定了该线程对应的堆栈大小，prio 则指定了该线程的优先级。其代码如下：

```

sys_thread_t sys_thread_new(char *name, void (* thread)(void
*arg), void *arg, int stacksize, int prio)
{
    xTaskHandle CreatedTask;
    int result;
    if ( s_nextthread < SYS_THREAD_MAX )
    {

```

```

/* 创建线程 */
result = xTaskCreate( thread, ( signed portCHAR * ) name,
stacksize, arg, prio, &CreatedTask );
/* 在线程定时器管理数组中存储该线程 */
s_timeoutlist[s_nexttthread++].pid = CreatedTask;
if(result == pdPASS)
{
    return CreatedTask;
}
else
{
    return NULL;
}
}
else
{
    return NULL;
}
}

```

12. sys_init 函数

该函数是操作系统模拟层的初始化函数。它主要对定时器管理数组进行了初始化。其代码如下：

```

void sys_init(void)
{
    int i;
    for(i = 0; i < SYS_THREAD_MAX; i++)
    {
        s_timeoutlist[i].pid = 0;
        s_timeoutlist[i].timeouts.next = NULL;
    }
    s_nexttthread = 0;
}

```

13. sys_arch_timeouts 函数

该函数用于返回当前任务的定时器管理链表首地址。其代码如下：

```

struct sys_timeouts *sys_arch_timeouts(void)
{
    int i;
    xTaskHandle pid;
    struct timeoutlist *tl;
    pid = xTaskGetCurrentTaskHandle();
    for(i = 0; i < s_nexttthread; i++)
    {
        tl = &(s_timeoutlist[i]);
        if(tl->pid == pid)
        {
            return &(tl->timeouts);
        }
    }
}

```

```
}  
return NULL;  
}
```

14. sys_arch_protect 函数

该函数用于保护临界区资源，其代码如下：

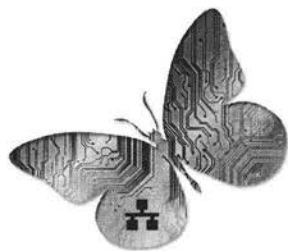
```
sys_prot_t sys_arch_protect(void)  
{  
    vPortEnterCritical();  
    return 1;  
}
```

15. sys_arch_unprotect 函数

该函数在访问临界区资源时使用，它必须与 sys_arch_protect 函数成对使用。其代码如下：

```
void sys_arch_unprotect(sys_prot_t pval)  
{  
    ( void ) pval;  
    vPortExitCritical();  
}
```

至此，将 LwIP 移植到 FreeRTOS 中的工作就完成了。总的说来，LwIP 具有良好的移植性，将 LwIP 移植到各操作系统中，只需要修改操作系统模拟层和以太网接口层的几个文件即可，也正因为其移植过程并不复杂，所以它才受到广大嵌入式开发者的喜爱。



第 13 章

工业通信网关解析

在前面介绍 STM32 平台上联合移植 FreeRTOS 和 LwIP 的基础上，本章将介绍通信网关的一般实现方式，通过以太网实现通信报文的转发和板载资源的控制。

13.1 概述

为了使初学者容易理解，本章实现的通信网关严格来说并不是真正意义上的通信网关，仅是对通信网关原理性的解析，因为网关的功能通常不仅是物理传输总线的适配和简单的报文转发，往往还要进行不同协议的转换。本章的通信网关仅仅是将以太网接收的报文转发到串行接口，对报文的内容不做任何解析和转换（通常也称为透明传输模式），这样便于读者观察和理解实际报文的传送状况。比如，人们通过客户端发送了一帧“hello world”的字符串，结果在串行接口接收到了“38 af de amb”的字符串，不明就里的人就会感到迷茫，究竟是编码错误，还是何种原因？事实上，网关只是在此基础上增加了对报文的解析和转化，就像翻译一样，将同样的信息以另一种语言来表达，这种转换本身没有太高深的难度，关键在于如何不失真地传达语言的原意。同样的道理，通信网关的难点在于如何在工程应用中保障通信的稳定性和可靠性，这些通常不是在各种工业通信协议的文本中所能述及和实现的。与 TCP/IP 协议一样，人们通常采用报文校验、超时重传、心跳报文等手段来保障通信链路的通畅和可靠性。

此外，读者应该将重点集中于操作系统的任务管理和任务间的通信等要点上，这也是本书想要读者掌握的重要内容。

13.2 编码实现

在第 12 章中，我们已实现了基于 FreeRTOS 的 LwIP 移植，我们将在此基础上，创建 3 个用户级任务，完成以太网和串口报文的互发互收，这 3 个任务分别是 tcpapp_thread、vComTxTask 和 vComRxTask。

首先通过函数 vAltStartComTestTasks 创建串口收发的两个函数 vComTxTask 和 vComRxTask，代码如下所示。函数 xSerialPortInitMinimal 将串口初始化（参数：波特率为 9600、无校验、8 位数据位和 1 位停止位）。

```

void vAltStartComTestTasks( unsigned portBASE_TYPE uxPriority,
                           unsigned long ulBaudRate)
{
    /* 初始化串口 */
    xSerialPortInitMinimal( ulBaudRate, comBUFFER_LEN );

    /* 创建消息队列 */
    xAppForTx = xQueueCreate( 5, sizeof(CommAppMsg) );

    /* 创建发送任务, 其优先级低于接收任务 */
    xTaskCreate( vComTxTask,
                ( signed char * )
                "COMTx",
                comSTACK_SIZE,
                NULL,
                uxPriority - 1,
                ( xTaskHandle * ) NULL );

    /* 创建接收任务 */
    xTaskCreate( vComRxTask,
                ( signed char * )
                "COMRx",
                comSTACK_SIZE,
                NULL,
                uxPriority,
                ( xTaskHandle * ) NULL );
}

```

当串口因收到数据而发生中断时, 中断服务程序通过信号量触发告知串口接收任务 vComRxTask 接收数据, 然后 vComRxTask 任务将数据存入自己的接收缓冲区, 当接收任务超时后, 将缓冲区中的数据通过以太网发送。

```

static portTASK_FUNCTION( vComRxTask, pvParameters )
{
    signed char cByteRxed;
    unsigned int len = 0;
    unsigned int i = 0;

    /* 避免编译器警告 */
    ( void ) pvParameters;

    for( ;; )
    {
        /* 阻塞消息队列, 直到串口接收缓冲区为空 */
        if( xSerialGetChar( xPort, &cByteRxed, comRX_BLOCK_TIME ) )
        {
            /* 使 LED 指示灯状态反转 */
            STM_EVAL_LEDToggle(LED2);
            /* 将接收的字节数据添加入队列 */
            UartPushQueueMsg(&gUartRecvQueue, cByteRxed);
        }
    }
}

```

```

/* 若串口接收缓冲区中无数据 */
else
{
    /* 获取接收队列中数据长度 */
    len = UartGetQueueLen(&gUartRecvQueue);
    for(i=0; i<len; i++)
    {
        /* 将数据存入接收缓存区, 等待发送 */
        UartGetQueueMSG(&gUartRecvQueue, &gUartRxbuf[i]);
    }
    /* 若有数据, 在超时后发送到网口 */
    if(len)
    {
        if(newtcpAppConn != NULL)
        {
            /* 发送数据 */
            netconn_write(newtcpAppConn,
                          gUartRxbuf,
                          len,
                          NETCONN_NOCOPY);
        }
    }
}
}
}

```

当以太网数据要通过串口发送时, 首先将数据存入响应的消息队列, 而串口发送任务则不断检查消息队列的状态。在数据就绪后, 串口发送任务从消息队列中将数据搬运到发送缓存区, 每搬运一个字节, 发送一次, 同时信号灯状态反转一次, 直到数据发送完毕, 最后串口发送一个回车字符, 并再次将信号灯反转。

```

static portTASK_FUNCTION( vComTxTask, pvParameters )
{
    CommAppMsg p;
    char cByteToSend = 0;
    unsigned int i = 0;

    /* 避免编译器警告 */
    ( void ) pvParameters;

    for( ;; )
    {
        /* 若消息队列不为空, 则接收数据 */
        if( xQueueReceive( xAppForTx, &p, 10 ) == pdTRUE )
        {
            for(i=0; i<p.len; i++)
            {
                /* 从队列搬运一个字节的数据 */
                cByteToSend = p.buf[i];
                /* 每搬运一个字节的数据, 从串口发送一次 */
            }
        }
    }
}

```

```

        if( xSerialPutChar( xPort, cByteToSend, comNO_BLOCK ) ==
            pdPASS )
        {
            /* 发送成功, 信号灯状态反转 */
            STM_EVAL_LEDToggle(LED1);
        }
    }
    /* 最后发送回车字符 */
    cByteToSend = '\r';
    if( xSerialPutChar( xPort, cByteToSend, comNO_BLOCK ) ==
        pdPASS )
    {
        /* 发送成功, 信号灯状态反转 */
        STM_EVAL_LEDToggle(LED1);
    }
}
}
}

```

以太网接收任务 `tcpapp_thread` 首先创建一个连接, 端口号为 2000, 随后进入监听模式。当以太网接收到数据时, 以太网接收任务 `tcpapp_thread` 将数据搬运到自己的数据缓存区, 并通过消息队列通知串口发送任务有数据等待发送。

```

static void tcpapp_thread(void *arg)
{
    err_t err;

    LWIP_UNUSED_ARG(arg);

    /* 创建一个连接标识 */
    tcpAppConn = netconn_new(NETCONN_TCP);

    if (tcpAppConn!=NULL)
    {
        /* 将连接绑定到端口 2000 */
        err = netconn_bind(tcpAppConn, NULL, 2000);

        if (err == ERR_OK)
        {
            /* 进入监听模式 */
            netconn_listen(tcpAppConn);

            while (1)
            {
                /* 建立连接 */
                newtcpAppConn = netconn_accept(tcpAppConn);

                /* 处理连接 */
                if (newtcpAppConn)
                {

```

```

        struct netbuf *buf;
        void *data;
        u16_t len;
        /* 当以太网接收到数据 */
        while ((buf = netconn_rcv(newtcpAppConn)) != NULL)
        {
            do
            {
                /* 将数据搬运到数据缓存区 data */
                netbuf_data(buf, &data, &len);
                /* 该函数将数据搬运到消息队列 */
                TCPApp_UartTx(data, len);
            } while (netbuf_next(buf) >= 0);
            /* 清空以太网数据 */
            netbuf_delete(buf);
        }

        /* 关闭连接, 丢弃连接标识 */
        netconn_close(newtcpAppConn);
        netconn_delete(newtcpAppConn);
    }
}
else
{
    printf(" can not bind TCP netconn");
}
}
else
{
    printf("can not create TCP netconn");
}
}

```

以太网发送的部分则直接通过 `netconn_write` 函数将数据发送, 与消息队列传递数据相比, 效率高, 但也存在代码维护不便等问题。当然, 凡事都有两面性, 读者可以将其视作一种功能实现策略的契机。

事实上, 某一功能实现的方法不止一种, 本章仅为读者展示了深入认识 FreeRTOS 特性和使用方法的一种简单实现。

13.3 通信测试

为了测试我们的例程, 本书的附带光盘中包含串口调试工具和以太网客户端工具。本节简要介绍使用这两种工具进行报文收发测试的方法。

1) 安装本书附带光盘的 ClientTool, 并启动。出现如图 13.1 所示的界面。在文本框中输入开发板网口的 IP 地址 192.168.0.10 (注意: 开发板作为服务器运行), 并单击“OK”按钮。

2) 在弹出的对话框中填入服务器的端口号, 这里为 2000, 并单击“OK”按钮, 如图 13.2 所示。至此, 客户端工具已启动, 如图 13.3 所示, 界面的下方显示了客户端和服务器的 IP 地址及对应的名称, 最上端则是报文发送串口, 在该窗口中输入报文并按下回车键即可向服务器发送报文。中间部分则是报文显示窗口, 用以显示接收到的以太网报文。

3) 启动串口调试工具, 界面如图 13.4 所示, 请按图中参数进行设置, 并单击左侧的串口打开控件, 使串口处于打开状态。

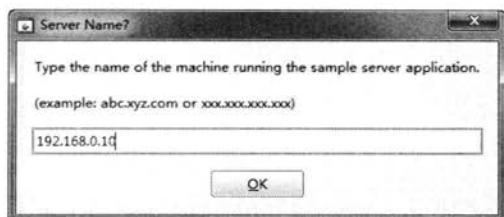


图 13.1 服务器 IP 地址配置

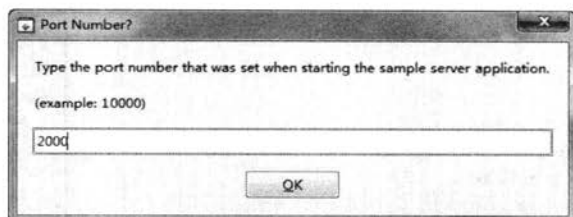


图 13.2 服务器端口号配置

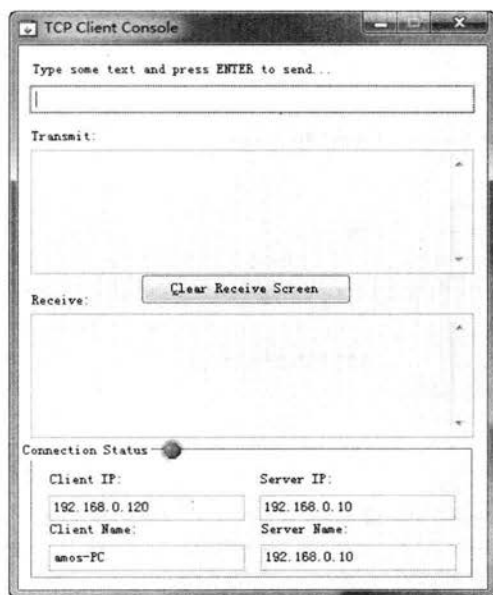


图 13.3 客户端界面



图 13.4 串口调试工具界面

4) 在以太网客户端的报文发送窗口输入想要发送的报文并按下回车键, 读者将会在串口调试工具的报文接收区看到刚才发送的报文, 同时在客户端的已发送报文串口将看到发送报文的记录, 如图 13.5 和图 13.6 所示。

5) 在串口调试工具的报文发送窗口输入想要发送的报文并单击发送, 将在客户端调试工具里看到串口调试工具发送的报文, 如图 13.7 和图 13.8 所示。

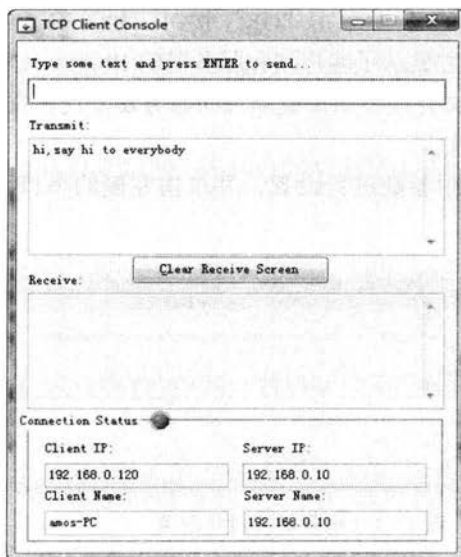


图 13.5 通过以太网向串口发送报文

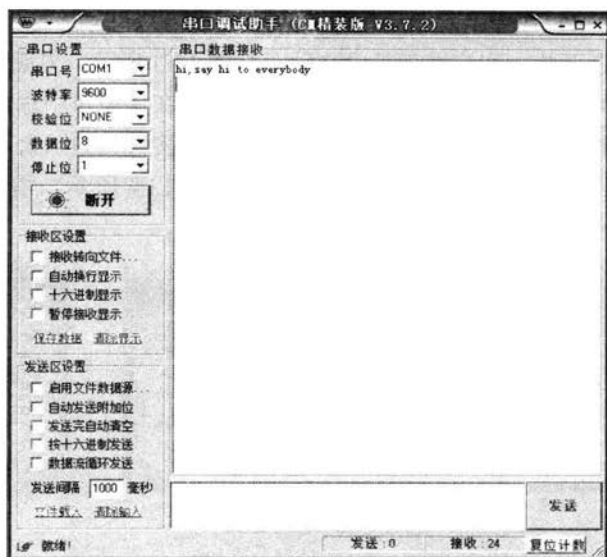


图 13.6 串口接收到的以太网报文



图 13.7 通过串口向以太网发送报文

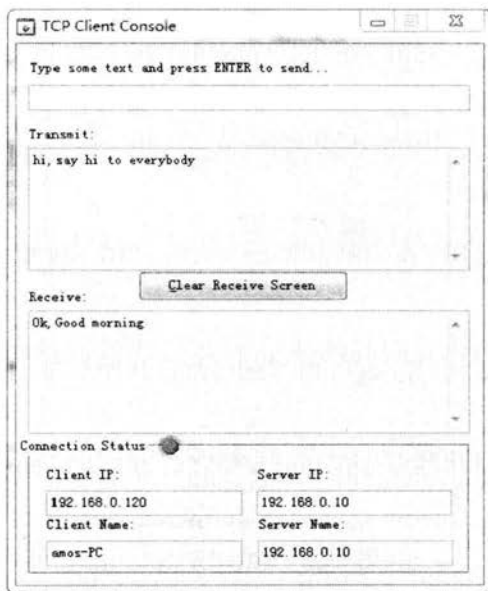


图 13.8 客户端工具接收到的串口报文

至此，我们已经完成了一个简单意义上的网关原型实现，并测试了报文的相互转发功能。与此同时，我们希望读者已经掌握了本书讲述的主要内容，即能够在 FreeRTOS 内核的基础上，灵活实现自己期望的某种功能。

附录A 开发板原理图

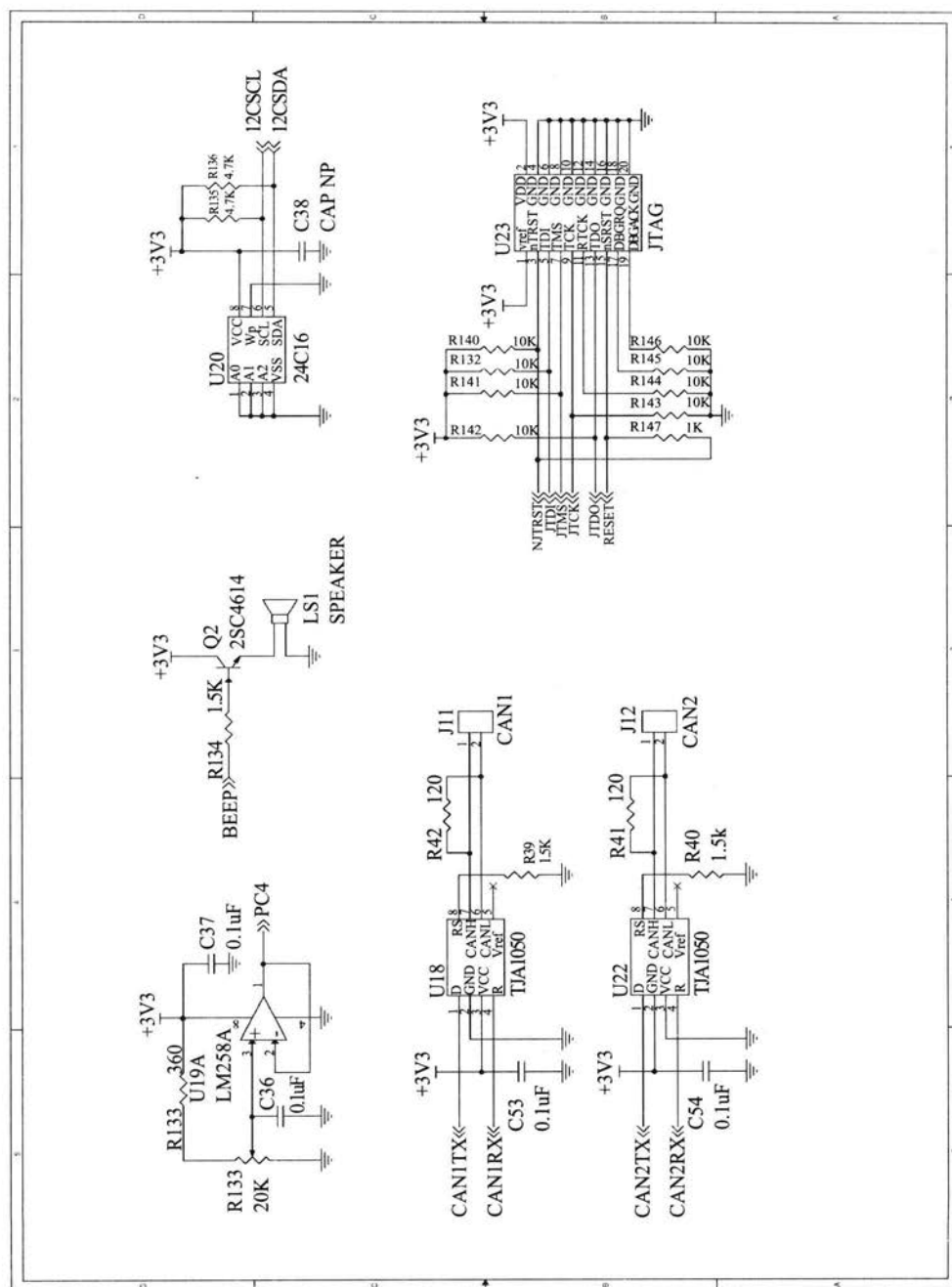


图 A.1 电位器模拟数据采集、CAN 总线接口、蜂鸣器、E2PPROM、JTAG 接口电路原理图

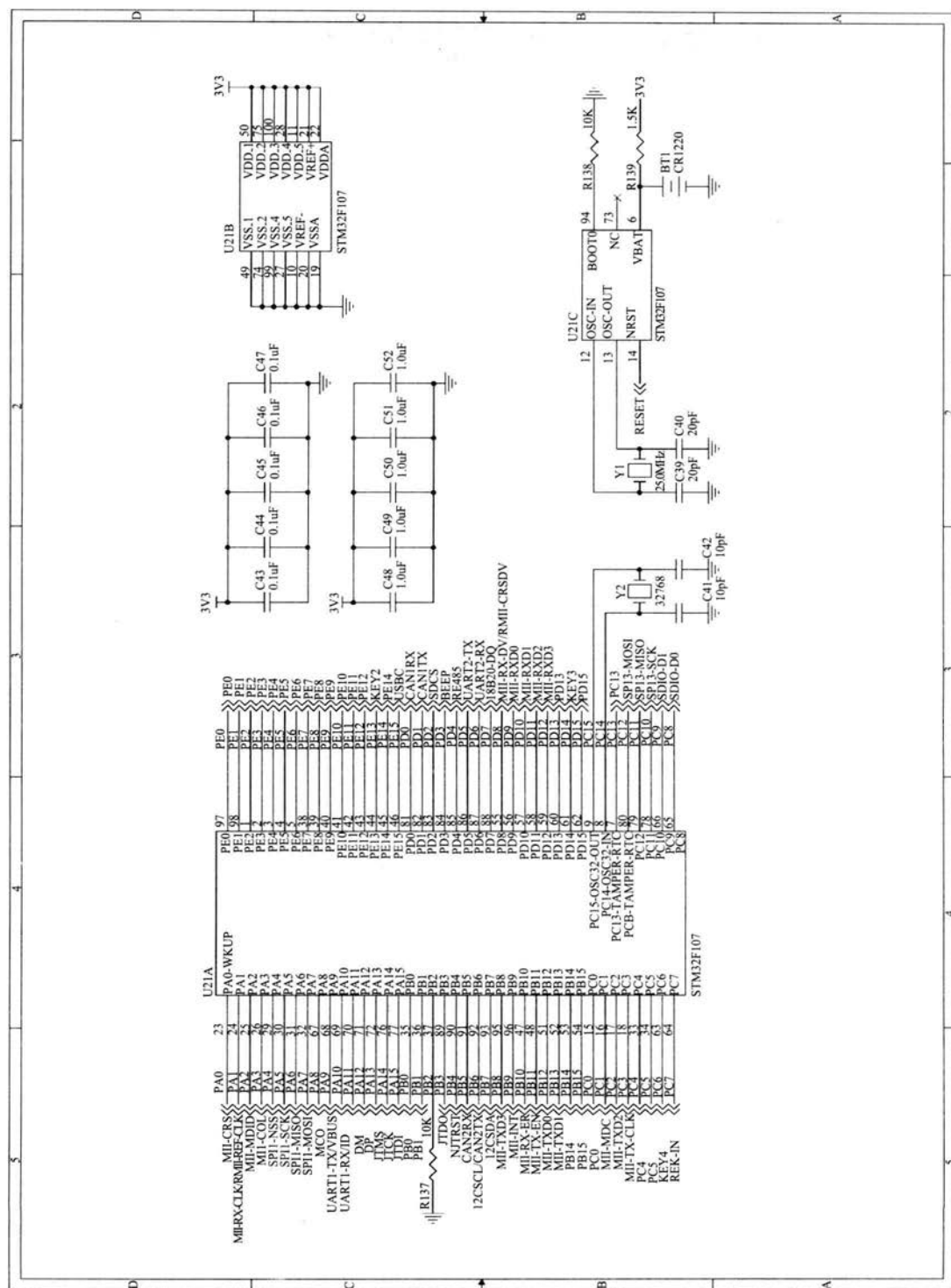


图 A.2 控制器部分电路原理图

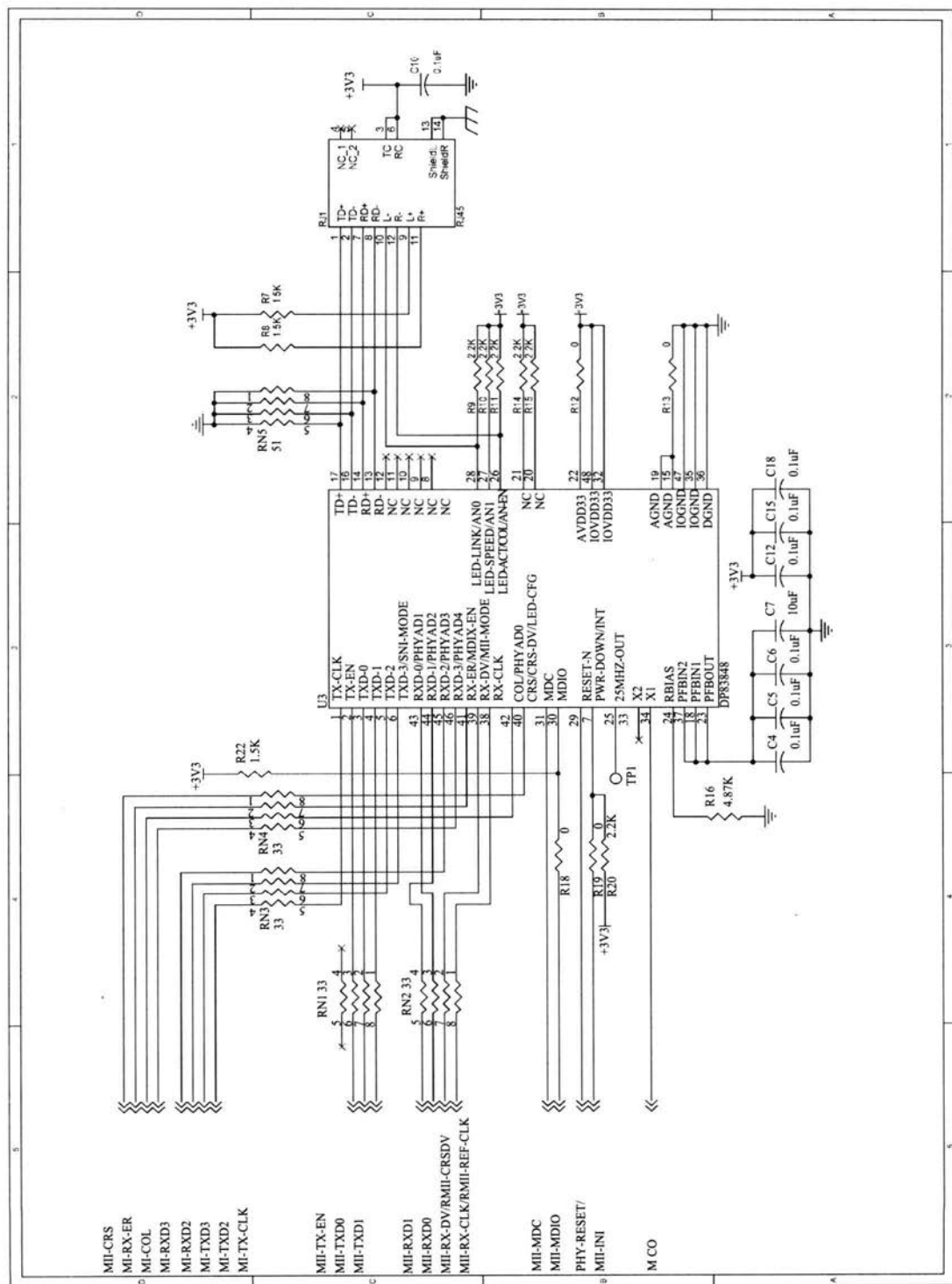


图 A.3 10/100M 自适应以太网接口电路原理图

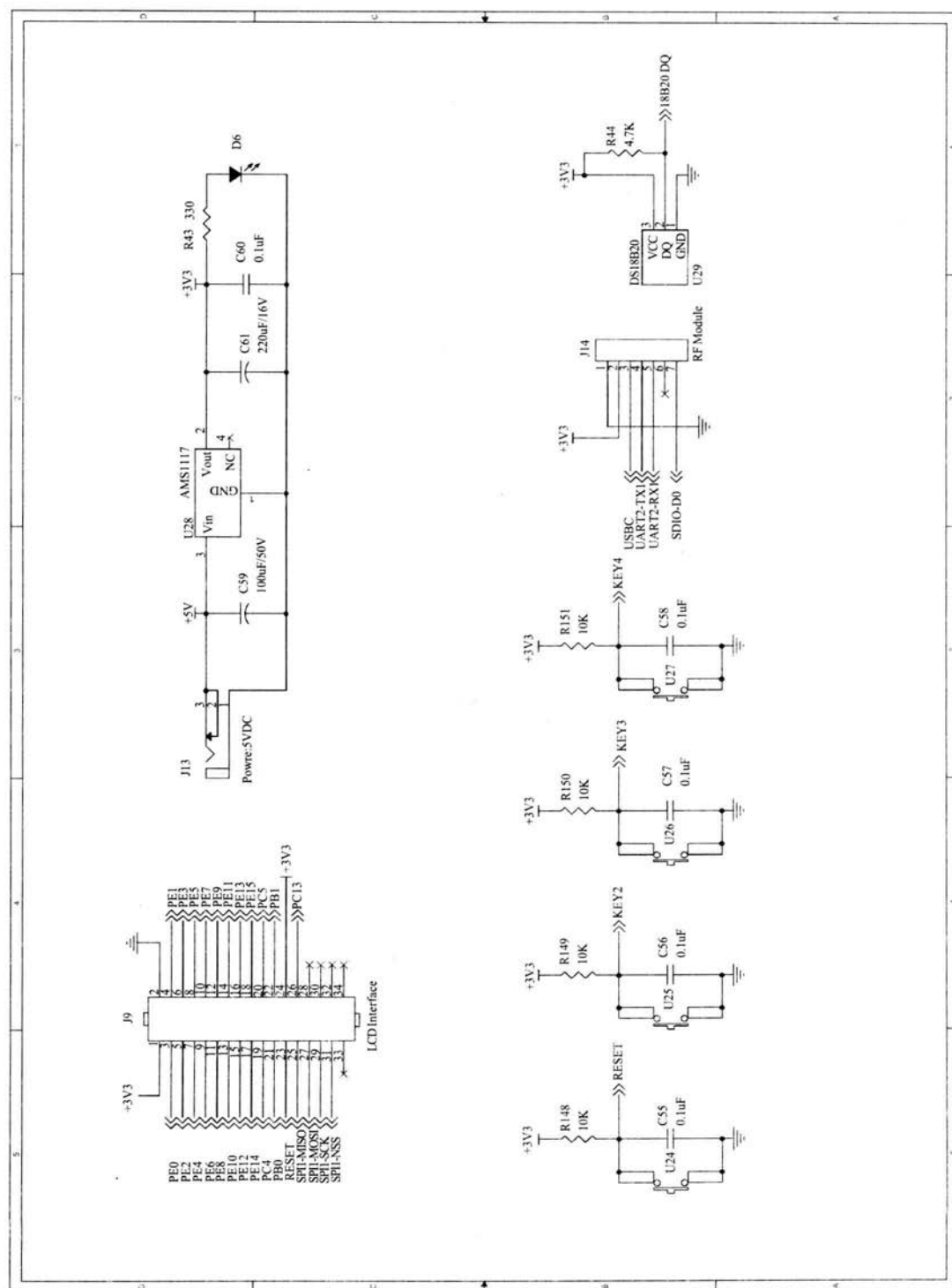


图 A.4 TFT 液晶模块接口、自定义功能键、复位键、电源、2.4G 射频模块接口、温度传感器原理图

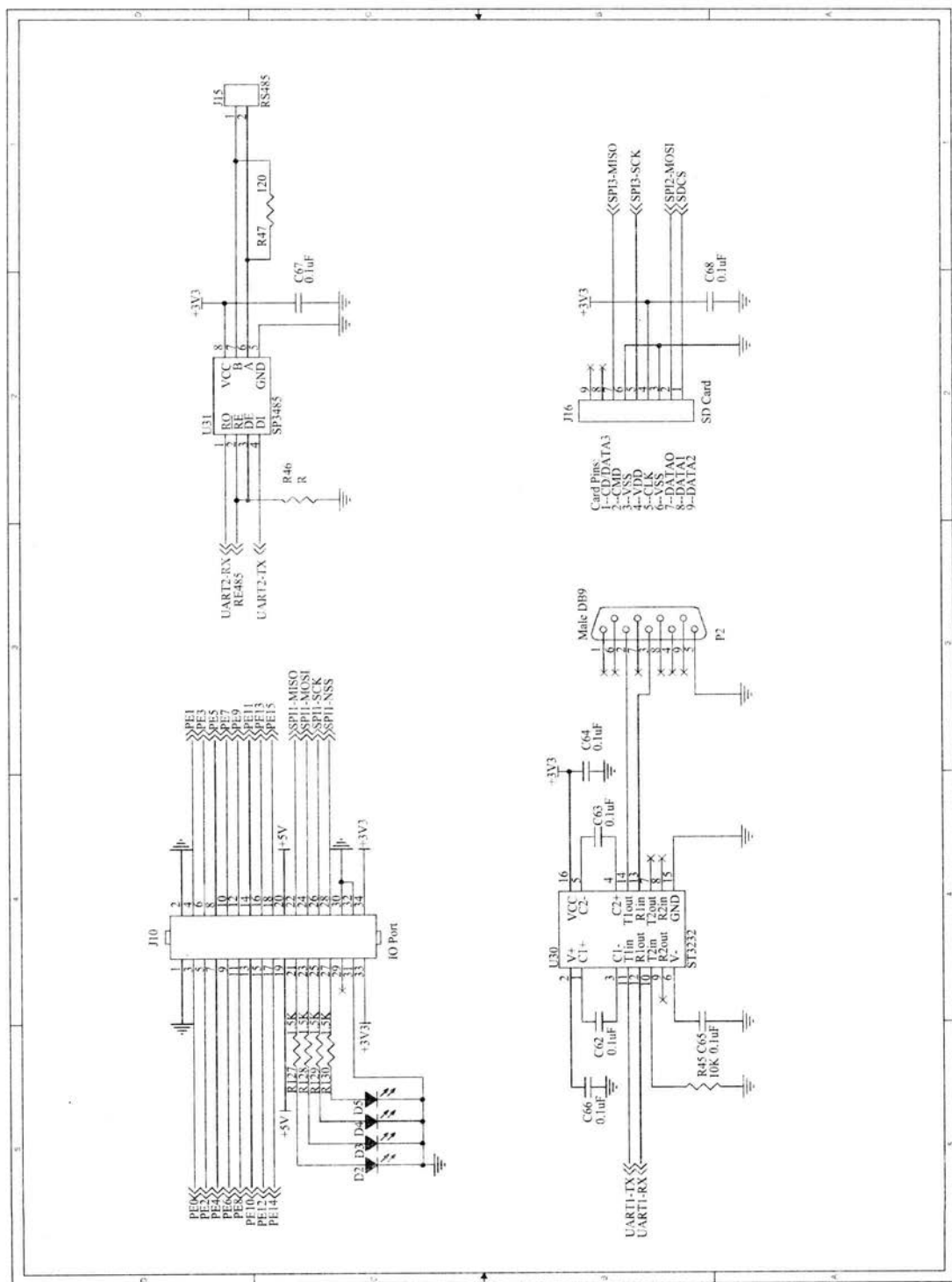


图 A.5 预留 IO 接口、RS232 接口、RS485 接口、SD 卡接口原理图

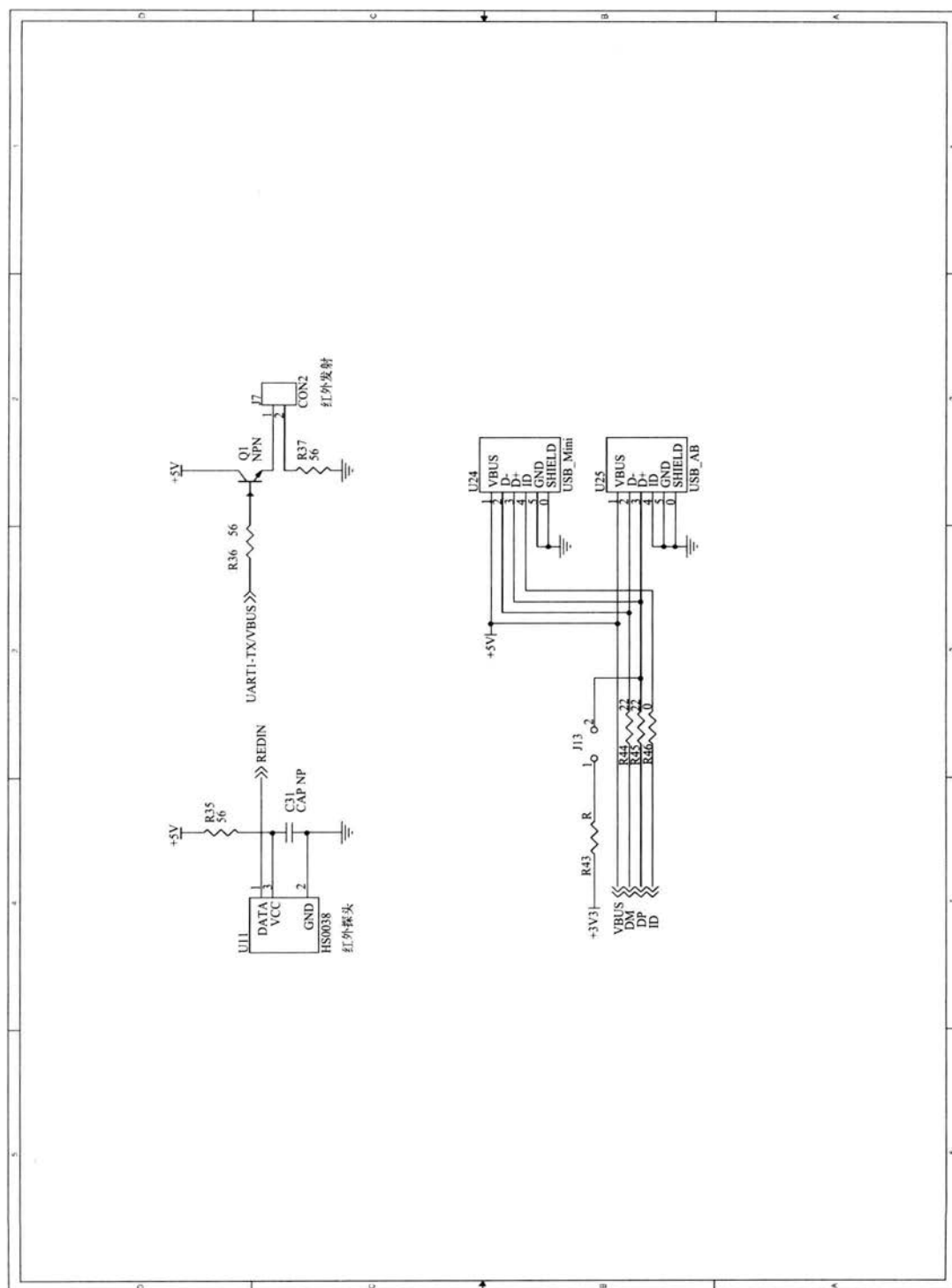


图 A.6 红外收发接口、USB Host 接口、USB Device 接口原理图

附录B 专业术语

application software (应用软件): 用来描述一个特定的嵌入式项目中的某一软件模块。应用软件不像可重用的交叉嵌入式平台, 这是因为每一个嵌入式系统都有不同的应用软件。

assembler (汇编编译器): 将汇编语言程序转换成处理器可理解和运行的机器指令序列的软件开发工具。

assembly language (汇编语言): 一种人可读的处理器指令集的形式。大多数处理器相关的功能必须用汇编语言编写。

binary semaphore (二进制信号量): 一种只有两种状态的信号。也叫互斥信号。

CPU (中央处理器): 处理器中执行指令的那一部分。

compiler (编译器): 把高级编程语言程序转换成目标处理器能够识别和执行的机器指令的一种软件开发工具。

context (上下文): 通常指处理器当前的寄存器状态和标志位。

context switch (上下文切换): 在多任务操作系统中由一个任务切换到另一个的过程。上下文切换包括保存正在运行的任务的上下文和恢复早先保存的另一个任务的上下文。

counting semaphore (计数信号量): 一种用来跟踪多个相同类型资源的信号灯。仅仅在所有可用的资源都被用完时才阻塞。相对二元信号而言。

critical section (临界区): 一段必须按次序执行的代码, 并且不能被中断, 否则不能保证软件正确操作。

cross-compiler (交叉编译器): 一个运行在不同平台上的编译器, 其中之一能产生目标代码。交叉编译器在主机上运行并且产生目标机的目标代码。

DMA (直接内存访问): 一种直接在两个外设 (通常是内存和 I/O 设备) 之间进行数据传输的技术, 它只要处理器最少的介入。DMA 传输由 DMA 控制器进行管理。

DRAM (动态随机访问存储器): 一种 RAM, 存储在其设备中的数据被定期刷新时才能保持它的内容。刷新周期一般由一个叫 DRAM 控制器的外设完成。

deadlock (死锁): 一种不希望出现的软件状态, 在这个状态下, 所有的任务因为等待一个只有在这些被阻塞任务之一才能产生的事件而被阻塞。如果死锁发生, 唯一解决的方法是重启硬件。但是, 通过可靠的软件设计实践活动通常可以避免死锁的发生。

device driver (设备驱动程序): 一个软件模块, 它隐藏特定外设的细节并提供高级的外设编程接口。

emulator (仿真器): 在线仿真器的简写。一个在目标板上放置模拟处理器行为的调试工具, 该工具允许在运行程序时观察处理器内部状态。

firmware (固件): 作为目标代码存储在 ROM 中的嵌入式软件。

flash memory (闪存): 一种能在软件的控制下被擦除和重写的存储器。

heap (堆): 一块被用作动态内存分配的内存区域。调用 malloc 和 free、C++ 的操作符 new、delete 在运行时进行堆操作。

I/O device (I/O 设备): 一种介于处理器和外界之间的硬件设备。一般的实例是开关、LED、串口和网络控制器等。

I/O space (I/O 空间): 一个由处理器提供的特殊内存区域。在 I/O 空间的内存位置和寄存器只能通过特殊的指令进行访问。

instruction pointer (指令指针): 包含下一条要执行指令地址的处理器中的寄存器。也叫程序计数器。

interrupt (中断): 处理器由于响应外部的事件而中止执行当前代码的行为。当一个中断发生, 当前的处理器状态被保存并且中断服务程序开始运行。当中断服务程序退出, 对处理器的控制权转到之前执行的代码处。

interrupt latency (中断延迟): 在中断发生和相关的中断服务程序运行之间的时间间隔。

interrupt service routine (中断服务程序): 响应特定中断而运行的一小段程序。

interrupt vector (中断向量): 中断服务程序所在的地址。

interrupt vector table (中断向量表): 中断类型决定的中断向量和索引序列, 包含了中断与中断服务程序之间的处理器的映射, 由程序员进行初始化。

intertask communication (进程间通信): 一种用于任务和中断服务程序(或任务)之间共享信息和同步对共享资源访问的机制。

multitasking (多任务): 伪并行运行的多个软件程序, 操作系统通过分配处理器时间来模拟并行方式。

mutex (互斥): 是指某一资源同时只允许一个任务对其进行访问的机制, 通常所访问的资源具有唯一性和排他性。

object file (目标文件): 包含目标代码的文件, 由编译器或汇编器产生。

peripheral (外设): 是指物理上连在处理器以外的硬件设备。对数据和信息起着传输、转送和存储的作用, 是嵌入式系统的重要组成部分。

polling (轮询): 一种通过不断读取外设寄存器, 以达到监测外设状态变化的一种策略。

reentrant (可重入): 可同时多次运行程序的说法。可再入的函数可以被安全地递归调用或由多任务多次调用。使代码可再入的关键在于确保在访问全局变量或共享寄存器时发生互斥现象。

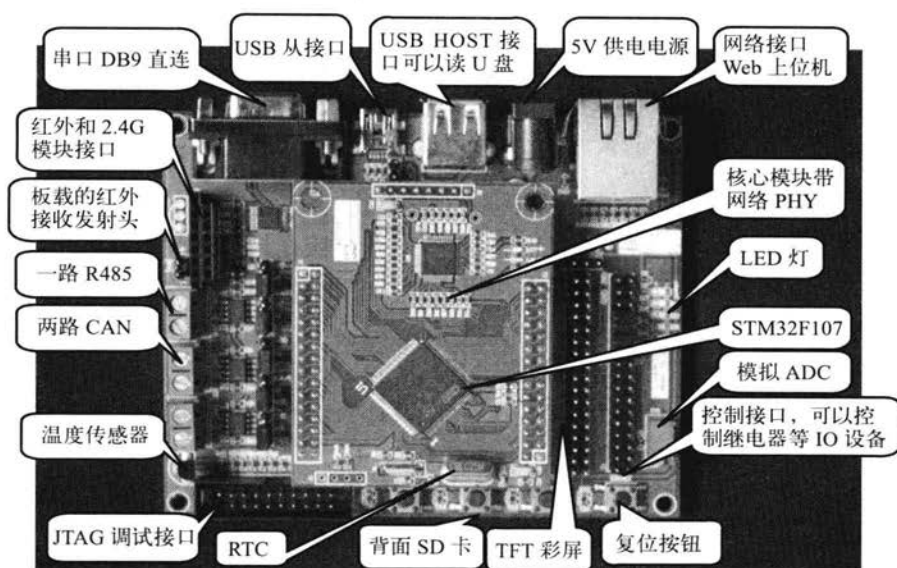
stack (堆栈): 一个包含后进先出队列的内存区域, 用来存储参数、自动变量、返回地址和其他一些必须在函数调用中进行维护的信息。对于嵌入式实时操作系统, 一般每一个任务都有自己独立的堆栈区。

startup code (启动代码): 所谓启动代码, 就是处理器在启动的时候执行的一段代码, 通常进行初始化处理器模式、设置堆栈、初始化变量等。与处理器体系结构和系统配置密切相关, 一般采用汇编语言编写。

参 考 文 献

- [1] 田泽. 嵌入式系统开发与应用 [M]. 北京: 北京航空航天大学出版社, 2005.
- [2] 杜春雷. ARM 体系结构与编程 [M]. 北京: 清华大学出版社, 2003.
- [3] 马忠梅, 等. 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.
- [4] 何加铭主编. 嵌入式 32 位微处理器系统设计与应用 [M]. 北京: 电子工业出版社, 2006.
- [5] 周立功主编. ARM 嵌入式系统基础教程 [M]. 北京: 北京航空航天大学出版社, 2005.
- [6] Brian W.Kernighan, Dennis M.Ritchie. C 程序设计语言 [M]. 徐宝文, 李志丛, 译. 北京: 机械工业出版社, 2004.
- [7] W.Richard Stevens. TCP/IP 详解 卷 1: 协议 [M]. 范建华, 等译. 北京: 机械工业出版社, 2009.
- [8] William Stallings. 操作系统: 精髓与设计原理 [M]. 陈向群, 陈渝, 译. 北京: 机械工业出版社, 2010.
- [9] Peter van der Linden. C 专家编程 [M]. 徐波译. 北京: 人民邮电出版社, 2008.
- [10] Jack Ganssle. 嵌入式系统设计的艺术 [M]. 李中华, 等译. 北京: 人民邮电出版社, 2011.
- [11] STMicroelectronics. PM0056, STM32F10xxx Cortex-M3 programming manual[P/OL]. www.st.com.
- [12] STMicroelectronics. PM0008, STM32F107xx advanced ARM-based 32-bit MCUsReference manual[P/OL]. www.st.com.
- [13] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack[M]. Swedish Institute of Computer Science, 2001.

配套STM32硬件平台



硬件配置：

核 心 板	底 板
<input type="checkbox"/> 主芯片STM32F107VCT6	一路串口
<input type="checkbox"/> PHY DP83848	一路RS485串口
<input type="checkbox"/> RTC时钟	两路CAN接口
	SPI接口 (SD卡)

该硬件平台基于 STM32F107，因此它不是基础的 STM32F103 系列，也不是互联的 STM32F105 系列，而是涵盖了两方并且有日趋使用广泛的网络功能的 STM32F107 系列 MCU。基于这个硬件平台，读者不仅可以学会简单的串口、SPI 等基础接口的一些知识，而且可以深入了解流行的 USB 和网络接口，也可以学习一门上位机语言。

讨论和购买该硬件平台请到 EETOP 论坛 STM32 子论坛或者扫描二维码：



对于现在的高校高年级学生而言,能够用C语言实现简单的嵌入式开发已不再是困难的事情。但普遍存在的问题是编码质量不高,缺乏系统性的规划,并且大多停留在简单实现特定功能的层面。本书为具有一定嵌入式开发基础的学生及初涉嵌入式领域的工程师系统性地介绍了嵌入式系统开发的一般过程和注意事项,并着重引荐了国外的一种开源嵌入式实时操作系统内核及TCP/IP协议栈,详述了移植、设计的一般步骤和方法,内容详实全面,特此向广大从事嵌入式开发工作的读者推荐。

——东南大学仪器科学与工程学院机器人传感与控制技术研究所 徐宝国老师

RTOS使从事嵌入式软件开发的广大工程师摆脱了繁重复杂的底层开发工作,而将精力集中于应用层软件的开发及系统的稳定性和实时性。或许,很多朋友对微内核的RTOS的稳定性会产生怀疑。事实表明,任何一个健壮的系统都源于严谨、规范化的设计和测试验证,如果没有对设计开发行为的良好规范和约束,最终的产品或系统实现将潜伏无数隐患。本书旨在引导广大读者朋友从系统性开发的角度出发着手具体的工作。本书讲述了嵌入式开发项目始末的一般过程和注意事项,重点介绍了RTOS的移植开发过程和方法,并在此基础上引入TCP/IP协议栈的移植和应用。在此,真诚愿广大读者朋友从中汲取各自成长所需的养分。

——中国电子科技集团公司第二十八研究所高级工程师, IEEE会员 崔桐

如今,某一行业的嵌入式应用水准也从某种意义上反映了相应行业的装备信息化水平。因此,提升嵌入式从业人员的专业技能和素养也是振兴我国各行各业的基础。本书不但为大家介绍了基于RTOS的嵌入式系统开发的一般方法和技巧,而且从嵌入式开发系统工程的角度开拓了读者视野,为有志于提升嵌入式开发水平,培养全面专业素养的读者朋友提供了一种值得借鉴或学习的方法。在此,也希望本书能引起广大嵌入式从业人员的有益思考。

——西安交通大学电气工程学院, IEEE会员, 袁凌老师

使用过RTOS的朋友应该都会有这样的感受:一旦熟悉了某个RTOS,并掌握了其使用方法,就更容易依赖上它。这是因为RTOS能够使程序构建得更具有逻辑性,而不必在一个轮询的死循环中精心构思设计方法和模式。RTOS可以将应用程序分解成多个具体的任务,每个任务完成相对独立的功能,任务间通过内核提供的某种机制进行通信或资源共享,这样就使得程序开发变得更加容易,提高了开发效率,缩短了开发周期,也便于维护。更为重要的是,RTOS能够对外部事件在确定的时限内予以响应,即RTOS的实时特性。这也是嵌入式实时系统最重要的特性之一。然而,RTOS的复杂性往往容易使初学者望而生畏。为此,本书通过介绍一款简洁易用的轻量级开源RTOS,展示了基于RTOS的嵌入式系统开发方法及要领,并在此基础上进行了TCP/IP协议栈的移植,着重讲述了移植的过程、方法和注意事项。此外,本书还讲述了在项目论证、编码实现环节中的注意事项。

——FreeRTOS中国站长 杨波



客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259
投稿邮箱: (010) 88379604

读者信箱: hzjsj@hzbook.com
华章网站: www.hzbook.com
网上购书: www.china-pub.com

