

- ◆ 兼顾基础，重在提高与技巧
- ◆ 解惑答疑，提升系统开发水平



嵌入式 系统开发

陈 卓 王 田 梁新元 等编著

代码的力量——

嵌入式系统开发

陈卓王田梁新元等编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书介绍了嵌入式开发多个方面的内容,涵盖面较为广泛。整本书分为3个部分:第1部分介绍嵌入式系统开发必须具备的软硬件基础。这一部分虽然是相对基础的内容,但在介绍重要内容的时候着重从应用的角度加以描述,在简单说明原理之后,回答了这些基础内容到底是怎么用的问题。由于涉及了不少实际系统,所以第1部分的内容是有一定深度的。第2部分讨论基于 $\mu\text{C}/\text{OSII}$ 的嵌入式系统开发,由于 $\mu\text{C}/\text{OSII}$ 系统的内核相对简单,所以这部分重点是说明 $\mu\text{C}/\text{OSII}$ 的程序设计框架及它的移植应用。在第3部分中,全面讨论了嵌入式Linux开发的各个重要方面,其中包括嵌入式Linux的驱动程序设计,这部分介绍了多种外设的驱动设计实例。另外,本书还讨论了Qt的程序设计和核心机制,以及Linux的网络实现的实例。最后还讨论了对一个嵌入式Linux系统移植所需要做的工作,分别介绍了BootLoader、内核及驱动程序的移植方法。

本书的读者对象包括:刚接触嵌入式开发的人员,以及有一定嵌入式开发基础,希望参考各种开发项目的工程人员。本书在编写的时候尽量做到了按知识需求组织,以方便不同的读者按需阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

嵌入式系统开发 / 陈卓等编著. —北京: 电子工业出版社, 2009.4
(代码的力量)
ISBN 978-7-121-08576-5

I. 嵌… II. 陈… III. 微型计算机—系统开发 IV. TP360.21

中国版本图书馆CIP数据核字(2009)第044869号

策划编辑: 祁玉芹

责任编辑: 段春荣

印 刷: 北京市天竺颖华印刷厂

装 订: 三河市鑫金马印装有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

开 本: 787×1092 1/16 印张: 25 字数: 640千字

印 次: 2009年4月第1次印刷

定 价: 38.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

前言

本书是讲解嵌入式系统开发的一本著作，完整地讨论了嵌入式开发必备的软、硬件基础知识，并且重点讲解了嵌入式开发的两个主要应用：驱动程序开发和嵌入式 GUI 程序开发。由于嵌入式系统开发有别于“一台 PC 就可以搞定”的纯软件开发，因此需要开发者有良好的软件开发技能，如：C/C++、汇编语言、操作系统等，同时还需要具备一定的硬件方面的知识，如：数字电路、计算机组成原理、接口电路等。而往往进入嵌入式开发领域的开发者要么以前是做软件开发的，对电路图、硬件驱动的原理等知识知之甚少，要么以前是一名硬件工程师，不太了解软件开发方面的基本要领，这些原因导致目前合格的嵌入式开发人员比较少。虽然已有不少关于嵌入式开发方面的优秀著作，但要么是针对水平较高的开发者编写，要么就是对嵌入式开发的某一个方面做深入的分析。这对刚涉及嵌入式开发的读者来说，会总觉得自己欠缺点什么，无法透彻理解其中的知识。

本书为了解决这个问题，特地提供给希望涉足嵌入式开发的朋友们以必要的、系统的知识，这也是本书的一个特色。在本书的第 1 部分中，比较详细地讨论了软件方面和硬件方面的必需了解和掌握的基本内容。第 1 章讨论嵌入式开发的最基本最重要的开发语言——C 语言的基本技能，特别是在嵌入式环境下的一些使用方法。第 2 章讨论嵌入式开发的硬件基础，包括对必要的数字电路方面知识的介绍及以 ARM 为核心的 S3C2410/S3C2440 处理器的介绍。阅读完本章后，读者首先应该能够分析一个电路的基本作用，并且能够对以 ARM 为核心的系列处理器有个比较深刻的认识。对于只有一定程序设计经历的读者，建议深入阅读整个第 1 部分；对硬件方面已经比较熟悉的读者则可以只阅读第 1 章。本书的第 2 部分讨论如何构建一个比较简单的嵌入式软件，并以实时嵌入式操作系统 μ C/OSII 为背景，首先分析其基本原理，然后结合具体的实例讨论其移植和相关应用。第 3 部分讨论比较复杂的嵌入式 Linux 系统，第 7 章着重讨论做嵌入式 Linux 的基础知识，包括重要的命令、开发环境的建立、交叉编译工具的建立与使用等。第 8 章在具备第 1 部分的软硬件基础的情况下讲解 Linux 驱动编写的基本方法，并且以几个典型的示例从基本的实现分析到驱动编写完整的向读者展示嵌

入式 Linux 的驱动软件开发的基本流程及实现方法。第 9 章讨论采用 Qt 为嵌入式 GUI 的嵌入式软件开发，由于 Qt 的开发涉及的内容很广不可能全面去叙述，所以本书先讨论 Qt 程序设计的基本方法，然后重点讨论了 Qt 实现的一些核心机制，这些机制的学习对于深入理解 Qt 开发是很有必要的。由于通信和网络领域为嵌入式系统应用最广泛和成熟的领域之一，所以本书在第 8 章详细地分析了 Linux 平台下的部分重要的协议及网络控制机制，如 TCP、UDP、IP、QoS、Congestion Control 等，这也算本书的另一个特色。

总的说来，本书有几个特点：一是“有浅有深”，既有为进一步深入实践必须具备的基础知识，也有对一些相对比较深入话题的探讨。二是“新”，本书讨论的技术都是目前该领域最新的内容。三是“全”，本书在给出基础知识以后，就话题展开，在第 3 部分讨论了嵌入式开发的诸多方面，包括：驱动开发、嵌入式 GUI 程序设计、嵌入式系统的网络协议等，其目的是适应更多读者的需要，并且能给读者一些提示。

读者朋友如有任何疑问和建议，可以到如下网站的本书讨论区同笔者一起探讨：

<http://www.pubeta.com>。

编著者

2009 年 3 月

目 录

C O N T E N T S

第 1 部分 嵌入式开发必备基础

第 1 章 软件开发基础	1
1.1 嵌入式环境下的 C 语言使用技巧	1
1.1.1 重要的位 (bit) 操作	1
1.1.2 正确使用数据指针	2
1.1.3 函数等价于指令的集合	3
1.1.4 操作有限的存储空间	4
1.1.5 理解栈空间 (Stack) 和堆空间 (Heap)	5
1.1.6 关键词 const 的使用	6
1.1.7 关键词 volatile	7
1.1.8 处理器字长与内存位宽不一致处理	7
1.1.9 struct {} 结构体的使用	8
1.2 ARM 汇编语言	10
1.2.1 学习方法介绍	10
1.2.2 ARM 微处理器的指令的分类与格式	11
1.2.3 指令的条件域	12
1.2.4 指令的寻址方式	12
1.2.5 ARM 汇编的指令分类讲解及示例	14
1.2.6 GNU ARM 汇编的格式	29
1.3 ARM 汇编和 C 语言的混合编程的实例	30
1.3.1 在 C 语言程序中内嵌汇编实例	30
1.3.2 在汇编中使用 C 语言程序定义的全局变量实例	31
1.3.3 在 C 语言程序中调用汇编的函数实例	32
1.3.4 在汇编中调用 C 语言的函数实例	33

1.4 本章小结	33
第2章 嵌入式开发的软件结构	35
2.1 轮询方式的嵌入式软件结构及实例	35
2.2 带中断处理的软件结构及实例	37
2.2.1 中断	37
2.2.2 中断处理程序及中断向量	38
2.2.3 软件结构及实例	40
2.3 本章小结	42
第3章 嵌入式操作系统	43
3.1 嵌入式操作系统概述	43
3.1.1 嵌入式操作系统的发展	43
3.1.2 嵌入式操作系统选型	44
3.1.3 几种典型的嵌入式操作系统	45
3.2 嵌入式操作系统的重要概念	47
3.2.1 代码的临界区	47
3.2.2 进程及进程结构体	48
3.2.3 进程的状态	51
3.2.4 可剥夺的内核	51
3.3 进程调度程序实例解析	52
3.3.1 基于映射表(Mapping Table)的 μ C/OS II 进程调度程序实例解析	52
3.3.2 Linux 2.6.X 的 0 (1) 进程调度程序实例解析	54
3.4 嵌入式文件系统实例	57
3.4.1 yaffs 文件系统数据在 NAND 上的存储方式	58
3.4.2 SuperBlock 结构	59
3.4.3 文件在内存中的组织方式	59
3.4.4 yaffs2 文件系统实例解析	60
3.5 板级支持包 (BSP)	72
3.6 本章小结	74
第4章 嵌入式开发的硬件基础	75
4.1 常用的电子元器件	75

4.1.1	电阻	76
4.1.2	电容	77
4.1.3	二极管	77
4.1.4	电感	77
4.1.5	三极管	78
4.1.6	运算放大器	79
4.2	IC 与硬件框图分析	80
4.2.1	IC 及封装方式	80
4.2.2	电路框图及分析	81
4.3	嵌入式处理器	86
4.4	S3C2410/2440 处理器介绍	90
4.5	ARM 嵌入式微处理器的选型	91
4.6	本章小结	92

第 2 部分 $\mu\text{C}/\text{OSII}$ 嵌入式开发

第 5 章	$\mu\text{C}/\text{OSII}$ 开发基础	93
5.1	嵌入式实时操作系统	93
5.2	$\mu\text{C}/\text{OSII}$ 开发要点及程序框架	95
5.2.1	任务设计结构	95
5.2.2	重要的 $\mu\text{C}/\text{OSII}$ 函数	96
5.2.3	$\mu\text{C}/\text{OSII}$ 的多任务机制	98
5.3	$\mu\text{C}/\text{OSII}$ 的启动初始化过程实例解析	99
5.4	本章小结	100
第 6 章	$\mu\text{C}/\text{OSII}$ 的移植及应用实例	101
6.1	ARM 平台的 $\mu\text{C}/\text{OSII}$ 移植实例	101
6.1.1	移植需要的文件	101
6.1.2	移植文件代码分析	102
6.2	基于 $\mu\text{C}/\text{OSII}$ 的网络协议栈 Lwip 移植实例	109
6.2.1	Lwip 简介	110
6.2.2	Lwip 的进程模型 (process model)	110
6.2.3	移植 Lwip 到 $\mu\text{C}/\text{OSII}$ 实例	111
6.3	$\mu\text{C}/\text{OSII}$ 的系统优化	118

6.3.1	任务切换要保存的数据	119
6.3.2	C 语言编译器中变量在堆栈中的位置	119
6.3.3	μC/OSII 对任务栈的处理方法与缺陷	119
6.3.4	共用空间的堆栈处理方法	120
6.4	本章小结	122

第 3 部分 嵌入式 Linux 开发

第 7 章	嵌入式 Linux 开发基础	123
-------	----------------------	-----

7.1	使用开发套件提供的编译环境	123
7.1.1	Linux 的交叉编译器	123
7.1.2	使用开发套件提供的交叉编译环境	124
7.2	自建交叉编译环境	128
7.2.1	设置环境变量, 准备源码及相关补丁	128
7.2.2	准备源码包	128
7.2.3	准备补丁	128
7.2.4	编译 GNU binutils	129
7.2.5	准备内核头文件	129
7.2.6	译编 glibc 头文件	129
7.2.7	编译 gcc 第一阶段	130
7.2.8	编译完整的 gcc	131
7.2.9	GNU 交叉工具链的下载	131
7.3	GNU 交叉工具链的介绍与使用	131
7.4	编译和配置 BootLoader	132
7.4.1	什么是 BootLoader	132
7.4.2	BootLoader 启动方式	132
7.4.3	常见的 BootLoader	134
7.4.4	vivi 的编译与配置	134
7.4.5	U-Boot 的编译与配置	137
7.5	Makefile 文件及编写实例	139
7.5.1	什么是 Makefile	139
7.5.2	程序的编译及链接	140
7.5.3	Makefile 编写要点	141
7.5.4	Makefile 的实例	145
7.5.5	Makefile 的编写规则	147

7.5.6	Makefile 中命令的编写实例	153
7.5.7	正确在 Makefile 中使用变量	156
7.5.8	使用条件判断	162
7.5.9	make 命令的执行	165
7.6	本章小结	168
第 8 章	嵌入式 Linux 系统的驱动程序开发	169
8.1	Linux 内核简介	169
8.2	Linux 重要的内核机制	171
8.2.1	Linux 的时钟机制	171
8.2.2	Linux 的软中断机制	177
8.3	Linux 的内核模块	181
8.3.1	什么是内核模块 (Kernel Model)	181
8.3.2	内核模块编写方法	181
8.3.3	内核模块实例	183
8.3.4	内核模块常用的资源	184
8.4	Linux 驱动程序开发概要及基本流程	186
8.5	字符设备驱动开发及实例	192
8.5.1	LED 的驱动程序实例	192
8.5.2	键盘驱动程序实例	194
8.5.3	串口驱动程序实例	199
8.6	音频设备驱动开发及实例	202
8.6.1	数字音频介绍	202
8.6.2	音频信号的硬件接口	203
8.6.3	Linux 音频的编程接口	204
8.6.4	Linux OSS 音频设备驱动实例说明	207
8.7	网络设备驱动开发及实例	215
8.7.1	Linux 核心数据结构 sk_buff{ }	215
8.7.2	Linux 核心数据结构 net_device{ }	221
8.7.3	Linux 网络驱动设计流程及开发实例	228
8.8	Nand Flash 驱动程序的编写实例	235
8.8.1	Nand Flash 原理介绍	235
8.8.2	Nand Flash 的驱动程序编写	237
8.9	本章小结	244

第 9 章 嵌入式 Linux 的 Qt 开发	245
9.1 Qt/Embedded 和 Qtopia 简介	245
9.2 建立 Qt/Embedded 的开发环境	246
9.2.1 安装与建立 Qt 桌面运行环境	246
9.2.2 对 Qt/Embedded 进行交叉编译	248
9.2.3 建立本机 Qtopia 虚拟平台	250
9.3 Qt 程序设计基础及简单实例	251
9.3.1 Qt 程序的简单例子	251
9.3.2 连接信号和响应函数	252
9.3.3 排列控件	253
9.3.4 建立对话框	254
9.3.5 主窗口的创建	257
9.4 Qt 的信号和槽机制及实例	260
9.4.1 信号和槽机制概述	260
9.4.2 信号 (signals)	260
9.4.3 槽 (slots)	261
9.4.4 连接信号和槽	261
9.4.5 元对象工具介绍	263
9.4.6 信号和槽实例解析	263
9.4.7 使用信号和槽之注意事项	265
9.5 Qt 的显示机制分析	267
9.6 本章小结	272
第 10 章 嵌入式 Linux 系统的网络协议栈	273
10.1 TCP/IP 协议及 Linux 协议栈概述	273
10.1.1 TCP/IP 协议概述	273
10.1.2 Linux 的网络协议栈	274
10.2 ARP 协议的实例	276
10.3 IP 协议及路由机制的实例	283
10.3.1 数据结构分类	283
10.3.2 邻接表 (Neighbor Table)	283
10.3.3 对路由缓存的分析	285
10.3.4 网络层的重要函数	285

103	10.3.5 IP 层的辅助函数讨论.....	307
508	10.4 Linux 网络的 QoS 的支持.....	308
573	10.4.1 Linux 对 QoS 支持概述.....	308
578	10.4.2 QoS 重要数据结构.....	309
579	10.4.3 QoS 队列策略 FIFO 的分析.....	310
579	10.4.4 复杂的 QoS 队列策略 Token-Bucket Filter 的分析.....	313
777	10.4.5 注册一个 QoS_ops.....	319
588	10.4.6 创建一个 Qdisc 结构.....	319
588	10.5 ICMP 协议的实例分析.....	323
288	10.5.1 重要的数据结构.....	323
888	10.5.2 发送一个 ICMP 报文.....	324
	10.5.3 接收一个 ICMP 报文.....	327
	10.5.4 回应一个 ICMP 报文.....	329
	10.5.5 处理 ICMP 重定向数据包.....	330
	10.5.6 处理 ICMP 请求回应数据包.....	330
	10.5.7 处理 ICMP 时间请求数据包.....	331
	10.5.8 处理 ICMP 不可达数据包.....	332
	10.6 TCP 协议的实例分析.....	335
	10.6.1 Linux 中 TCP 的初始化.....	335
	10.6.2 TCP 路径 MTU 的发现机制.....	336
	10.6.3 Linux 的重传机制实现.....	340
	10.6.4 TCP 的滑动窗口协议实现.....	344
	10.6.5 接受一个 TCP 连收.....	346
	10.6.6 TCP 定时器机制实现.....	348
	10.7 本章小结.....	350
	第 11 章 嵌入式 Linux 系统的移植.....	351
	11.1 引导系统 BootLoader 的移植实例.....	351
	11.1.1 U-Boot 的实现.....	351
	11.1.2 移植 U-Boot.....	356
	11.2 uCLinux 的移植实例.....	360
	11.2.1 BootLoader 及内核解压.....	360
	11.2.2 几种内核启动方式介绍.....	360
	11.2.3 内核启动地址的确定.....	360

308	11.2.4 系统入口分析	361
308	11.2.5 内核引导过程分析	362
308	11.3 Linux 2.6 内核的移植实例	373
309	11.3.1 准备 Linux2.6.X 内核	373
310	11.3.2 修改 Makefile 文件	374
313	11.3.3 设置 Flash 分区结构	374
319	11.3.4 配置嵌入式 Linux2.6 内核	377
319	11.4 Linux 中网络驱动程序的移植实例	382
323	11.4.1 移植步骤	382
323	11.4.2 移植过程中的问题及解决方法	385
324	11.5 本章小结	388
327	10.2.3 接收一个 ICMP 报文	327
329	10.2.4 回应一个 ICMP 报文	329
330	10.2.5 处理 ICMP 重定向数据	330
330	10.2.6 处理 ICMP 超时数据	330
331	10.2.7 处理 ICMP 时间戳数据	331
332	10.2.8 处理 ICMP 不可达数据	332
332	10.3 TCP 协议的实现	332
332	10.3.1 Linux 中 TCP 的初始化	332
336	10.3.2 TCP 路由 MTU 的调整机制	336
340	10.3.3 Linux 的重传机制实现	340
344	10.3.4 TCP 的滑动窗口协议实现	344
346	10.3.5 接受一个 TCP 连接	346
348	10.3.6 TCP 定时器机制	348
350	10.4 本章小结	350
351	第 11 章 嵌入式 Linux 系统的移植	351
351	11.1 引导系统 Bootloader 的移植实例	351
351	11.1.1 U-Boot 的实现	351
356	11.1.2 移植 U-Boot	356
360	11.2 Linux 的移植实例	360
360	11.2.1 Bootloader 及内核移植	360
360	11.2.2 内核移植驱动方式介绍	360
360	11.2.3 内核启动地址的确定	360

第 1 部 分

嵌入式开发必备基础

嵌入式系统开发是一个“门槛”比较高的技术领域，因为该领域不但需要开发者能编写应用程序，还具备对硬件的基本分析和调试的能力。单纯只了解软件的开发由于对系统底层硬件很多原理缺乏基本认识，可能发现自己始终浮于表面，深入不下去。因此，做嵌入式开发需要“软硬兼施”。

第 1 章 软件开发基础

要成为一名合格的嵌入式开发工程师，对 C 语言的熟练掌握是必须的，可以说 C 语言是嵌入式开发最重要和最通用的语言。目前很多成熟的嵌入式软件产品自身就是采用 C 语言为主开发的，如： μ C/OSII、嵌入式 Linux、Qt、MiniGUI 等。所以，熟练掌握 C 语言不仅可以开发很多底层系统程序，并且能更加深刻的了解软件是如何操作硬件资源的。

1.1 嵌入式环境下的 C 语言使用技巧

本节主要结合嵌入式开发的特点，讲解 C 语言的应用要点。随着进一步的学习和实践，读者将会发现本章总结的几个基本技巧是很有用的。

1.1.1 重要的位 (bit) 操作

位 (bit) 是程序设计中可以操作的最小数据单位，理论上可以用“位运算”来完成所有的运算和操作。在 PC 程序设计中涉及到对位操作的机会比较少。但如上面提到的，在嵌入式程序开发中常常操作的是以位为单位的数据，因此首先讨论 C 语言位操作的两个重要作用。

位操作的作用之一是可以减少除法和取模的运算。灵活的位操作可以有效地提高程序运行的效率，这对于处理器能力相对较弱的嵌入式开发来讲是非常有必要的。举例如下：

```
/* 方法1 */      /* 方法2 */
unsigned int i,j;  unsigned int i,j;
i = 156 / 16;     i = 156 >> 4;
```

```
j = 562 % 32;
```

```
j = 562 - (562 >> 5 << 5);
```

对于以 2 的指数次方为“*”、“/”或“%”因子的数学运算，右边的方法 2 采用移位运算“<<”及“>>”通常可以提高算法效率。这是因为乘除运算的处理器指令周期通常比移位运算大，在读一些嵌入式程序代码时经常会看到这样的移位运算，实际作用就是降低代码的运行时间，提高效率。

移位操作有两种，即左移位和右移位。左移位操作比较简单，左移几位就在右边空出的位置补上几个零，即对于一个数 NUM 的位表示 $[X_{n-1}, X_{n-2}, \dots, X_0]$ ，如果 $NUM \ll k$ 过后，NUM 的位表示应该为 $[X_{n-k-1}, X_{n-k-2}, \dots, X_0, 0, \dots, 0]$ ，一共在右边填充了 k 个零。而相比之下右移位操作要复杂一些，有两种情况即：算数右移和逻辑右移，逻辑右移在左边补 k 个零，对于 $NUM \gg k$ 后得到的结果为 $[0, \dots, 0, X_{n-1}, X_{n-2}, \dots, X_k]$ ，而算数右移 $NUM \gg k$ 后得到的结果为 $[X_{n-1}, \dots, X_{n-1}, X_{n-1}, X_{n-2}, \dots, X_k]$ ，结果是在左边补充 k 个最高位 X_{n-1} 。一般而言，编译程序对于有符号数的右移都是算数右移，而对于无符号数的右移都是逻辑右移，在做嵌入式程序设计中常定义一个无符号数，如上面的例子，使用 unsigned int 类型，而不直接使用 int 类型，其目的就是为使用逻辑右移。

C 语言位运算除了可以提高运算效率外，在嵌入式程序设计中，它的第二个重要作用是实现位间的与(&)、或(|)、非(~)操作。这跟嵌入式系统的程序设计特点有很大关系。因为对底层硬件（处理器，内存，外围器件）的操作实际上都是通过读写相关寄存器进行，而对寄存器的读写，实际上就是按照硬件的器件手册（Data Sheet）的描述对该寄存器的某个或某些位进行与、或、非操作后置位。这一点，会在第 2 章中进一步解释。例如，通过将 AM186ER 型 80186 处理器的中断屏蔽控制寄存器的低 6 位设置为 0（开中断 2），最通用的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp & INT_I2_MASK);
```

而将该位设置为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp | ~INT_I2_MASK);
```

判断该位是否为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
if (wTemp & ~INT_I2_MASK)
{
    ... /* 该位为 1 */
}
```

1.1.2 正确使用数据指针

在嵌入式系统的程序设计中，常常要求在特定的存储单元读写内容或者直接访问嵌入式处理器的某个经过地址编址的寄存器（Register），通常汇编有对应的 MOV 指令，而除 C/C++

以外的其他程序设计语言基本没有直接访问绝对地址的能力。在嵌入式系统的实际调试中，多借助 C 语言指针所具有的对绝对地址单元内容的读写能力。以指针直接操作内存多发生在如下几种情况：①某 I/O 芯片被定位在 CPU 的存储空间而非 I/O 空间，而且寄存器对应于某特定地址；②两个 CPU 之间以双埠 RAM 通信，CPU 需要在双埠口 RAM 的特定单元（称为 mail box）书写内容以在对方 CPU 产生中断；③在利用字符迭加做自字符及简单图形显示时，需要读取在 ROM 或 Flash 的特定单元所刻录的汉字和英文字母。例如：

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0xF000+0xFF00（这里假设处理器为 16 位的）写入 11。在使用绝对地址指标时，要注意指标自增自减操作的结果取决于指针指向的数据类别。上例中 p++后的结果是 p=0xF000FF01，若 p 指向 int，即：int *p = (int *) 0xF000FF00; p++（或 ++p）的结果等同于：p = p+sizeof (int)，而 p-（或-p）的结果是 p = p-sizeof (int)。同理，若执行：long int *p = (long int *) 0xF000FF00; 则 p++（或 ++p）的结果等同于：p = p+sizeof (long int)，而 p-（或-p）的结果是 p = p-sizeof (long int)。

通过上面的解释，可以看到处理器是以字节为单位编址，而 C 语言指针以指向的数据类型长度做自增和自减。理解这一点对于以指标直接操作存储空间或硬件的地址空间是相当重要的。

1.1.3 函数等价于指令的集合

首先要理解以下 3 个问题：

① 语言中函数名直接对应于函数生成的指令代码在内存中的地址，因此函数名可以直接赋给指向函数的指针。

② 调用函数实际上等同于“调转指令+参数传递处理+回归位置入栈”，本质上最核心的操作是将函数生成的目标代码的首地址赋给 CPU 的 PC 寄存器。

③ 因为函数调用的本质是跳转到某一个地址单元的 code 去执行，所以可以“调用”一个根本就不存在的函数实体。大学里的《计算机组成原理》课程中曾经讲到，186 CPU 启动后跳转至绝对地址 0xFFFF0（对应 C 语言指针是 0xF000FFF0，0xF000 为段地址，0xFFFF0 为段内偏移）执行，请看下面的代码：

```
typedef void (*lpFunction) ( ); /* 定义一个无参数、无返回类型的 */
/* 函数指针类型 */
lpFunction lpReset = (lpFunction)0xF000FFF0; /* 定义一个函数指针，指向*/
/*CPU启动后所执行第一条指令的位置 */
lpReset(); /* 调用函数 */
```

在以上的程序中，根本没有看到任何一个函数实体，但是却执行了这样的函数调用 lpReset()，它实际上起到了“软重启”的作用，跳转到 CPU 启动后第一条要执行的指令的位置。所以应该理解函数的本质，实际上函数就是一个指令集合；你可以调用一个没有函数体的函数，本质上只是换一个地址开始执行指令。

1.1.4 操作有限的存储空间

在嵌入式系统中易失存储器申请比一般系统程序设计时有更严格的要求，这是因为嵌入式系统的存储空间往往十分有限，不经意的存储空间泄露可能会很快导致整个系统崩溃。所以如果要采用动态分配策略操作存储空间，一定要小心以下几种情况。

① 一定要保证 `malloc` 和 `free` 语句成对出现，谁申请的空间就要由谁来释放。如有下面这样的一段程序：

```
char * function(void)
{
    char *p;
    p = (char *)malloc(...);
    if(p==NULL)
        ...;
    ...
    /* 一系列针对p的操作 */
    return p;
}
```

在某处调用 `function()`，用完 `function` 中动态申请的内存后将其 `free`，如下：

```
char *q = function();
...
free(q);
```

上述代码实际上存在很大的隐患，不小心很容易导致存储空间泄露。存储空间由 `function` 来申请，却由另外一个函数来释放。如果在外函数中没有使用 `free (q)`，那么就会导致每使用一次 `function` 函数就会损失一块存储空间，不满足 `malloc` 和 `free` 成对出现的原则。另外，这样使用还会导致代码的耦合度增大，因为用户在调用 `function` 函数时需要知道其内部细节。

正确的操作方法是在调用子函数时申请存储空间，并把指针传入 `function` 函数，如下：

```
char *p=malloc(...);
if(p==NULL)
    ...;
function(p);
...
free(p);
p=NULL;          /*避免野指针*/

void function(char *p) /*函数接收指针p*/
{
    ...
    /* 一系列针对p的操作 */
}
```

② 不要用指针动态申请存储空间。先看如下一段代码，这种错误的使用方法比较常见。

```
void GetMem(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
```