

基于 32 位 ColdFire 构建嵌入式系统

王宜怀 陈建明 蒋银珍 著

电子工业出版社

内容简介

本书以飞思卡尔半导体公司（原摩托罗拉半导体部）的 32 位 ColdFire 系列微控制器中带以太网接口的 MCF52233、带 CAN 总线接口 MCF52235、带 USB2.0 接口 MCF52223 三个型号为蓝本阐述嵌入式系统的软件与硬件设计。全书共 16 章，其中第 1 章为概述，阐述嵌入式系统的知识体系、学习误区、学习建议。2-3 章概述 ColdFire 系列微处理器特点，给出 MCF52233 的引脚功能与硬件最小系统电路，给出第一个样例程序及 ColdFire 工程组织，完成第一个 ColdFire 工程的入门。4-10 章分别给出 UART、键盘、LED 与 LCD、AD、定时器、QSPI、I2C 与 Flash 存储器在线编程等。11-15 分别给出 MCF52235 的 CAN 总线、基于 MCF52233 嵌入式以太网、MCF52233 的其他模块、MCF52223 的 USB 2.0 编程、 $\mu\text{C}/\text{OS-II}$ 在 ColdFire 上的移植与应用，第 16 章给出基于硬件构件的嵌入式系统开发方法。

附录含有有关实践资料索引。本书配有光盘，包含了书中提供的所有实例源程序、辅助资料、相关芯片资料及常用软件工具。

本书可供大学有关专业的高年级学生和研究生用作教材或参考读物，也可供嵌入式系统开发与研究人员用作参考和进修资料。

前 言

嵌入式计算机系统简称嵌入式系统，其概念最初源于传统测控系统对计算机的需求。随着以微处理器（MPU）为内核的微控制器（MCU）制造技术的不断进步，计算机领域在通用计算机系统与嵌入式计算机系统这两大分支分别得以发展。通用计算机已经在科学计算、事物管理、通信、日常生活等各个领域产生重要影响。在后 PC 时代，嵌入式系统的广阔应用将是计算机发展的重要特征。一般来说，嵌入式系统的应用范围可以粗略分为两大类：一类是电子系统的智能化（工业控制、现代农业、家用电器、汽车电子、测控系统、数据采集等）；另一类是计算机应用的延伸（MP4、手机、通信、网络、计算机外围设备等）。不论如何分类，嵌入式系统的技术基础是不变的，即要完成一个以 MCU 为核心的嵌入式系统应用产品设计，需要有硬件、软件及行业领域相关知识。但是，随着嵌入式系统中软件规模日益增大，对嵌入式底层驱动软件的封装提出了更高的要求，可复用性与可移植性受到特别关注，嵌入式软件构件化开发方法逐步被业界所重视。本书以嵌入式硬件构件与底层软件构件设计为主线，按照嵌入式软件工程的要求，以飞思卡尔半导体公司的 32 位 ColdFire 系列微控制器中带以太网接口的 MCF52233、带 CAN 总线接口的 MCF52235、带 USB2.0 接口的 MCF52223 三个型号为蓝本，按照“通用知识—芯片编程结构概要—基本编程方法—底层驱动构件封装—应用方法与举例”的线条，逐步阐述电子系统智能化嵌入式应用的软件与硬件设计。

关于飞思卡尔 ColdFire 系列微控制器:

飞思卡尔半导体是全球最大半导体公司之一，在微控制器领域长期居全球市场领先地位，以高可靠性获得业界一致赞誉。该公司的微控制器产品系列齐全，由不同位数（8 位、16 位、32 位等），不同封装形式（DIP、SOIC、QFP 等），不同温度范围，所含模块不同等构成了庞大的飞思卡尔微控制器产品系列，广泛应用于汽车电子、消费电子、工业控制、网络和无线市场等嵌入式系统各个领域。该公司每年的研发投入超过 12 亿美元。飞思卡尔公司的 ColdFire 系列单芯片微控制器解决方案主要面向工业/家庭自动化、远程数据采集与控制、医疗仪表、照明控制节点、以太网应用、USB 应用、ZigBee™ 控制节点等领域。该公司自 2006 年开始提供 32 微控制器 MCF5223x、MCF5222x 部分样片以来，已经逐步形成了不同的功能、价格、集成度和调试能力的 32 位单芯片嵌入式应用的解决方案，使得用户可以各取所需。也就是说，面向电子系统智能化嵌入式应用的设计，无论需要怎样的系统功能和集成度，总能从 ColdFire 庞大产品系列中选取一款合适的芯片进行应用开发。这正是嵌入式系统产品设计者所期望的。本书是第一本以完整实例方式阐述飞思卡尔 MCF5223x 与 MCF5222x 应用设计的著作。

本书特点:

- （1）把握通用知识与芯片相关知识之间的平衡。书中对于嵌入式“通用知识”的基

本原理，以应用为立足点，进行语言简洁、逻辑清晰的阐述，同时注意与芯片相关知识之间的衔接，使读者在更好地理解基本原理的基础上，理解芯片应用的设计，同时反过来，加深对通用知识的理解。

(2) 把握硬件与软件的关系。嵌入式系统是软件与硬件的综合体，嵌入式系统设计是一个软件、硬件协同设计的工程，不能像通用计算机那样，软件、硬件完全分开来看。特别是对电子系统智能化嵌入式应用来说，没有对硬件的理解就不可能写好嵌入式软件，同样没有对软件的理解也不可能设计好嵌入式硬件。因此，本书注重把握硬件知识与软件知识之间的关系。

(3) 对底层驱动进行构件化封装。书中对每个模块均给出根据嵌入式软件工程基本原则并按照构件化封装要求编制底层驱动程序，同时给出详细规范的注释及对外接口，为实际应用提供底层构件，方便移植与复用，可以为读者进行实际项目开发节省大量时间。

(4) 设计合理的测试用例。书中所有源程序均经测试通过，并保留测试用例在随书所附的网上光盘中，避免了因例程的书写或固有错误给读者带来烦恼。这些测试用例，也为读者验证与理解带来方便。

(5) 网上光盘提供了所有模块完整的底层驱动构件化封装程序、文档与测试用例，同时网上光盘中还包含芯片参考手册、写入器安装与使用方法、工具软件（开发环境、程序写入与读出软件、串口调试工具、USB 工具、以太网工具）、有关硬件原理图及其他技术资料。

(6) 提供硬件评估板、写入调试器，并给出单独进行程序写入与读出的软件工具，方便读者进行实践与应用。

主要内容:

全书共 16 章，其中第 1 章为概述，阐述嵌入式系统的知识体系、学习误区、学习建议。2-3 章概述 ColdFire 系列微处理器特点，给出 MCF52233 的引脚功能与硬件最小系统电路，给出第一个样例程序及 ColdFire 工程组织，完成第一个 ColdFire 工程的入门。4-10 章分别给出 UART、键盘、LED 与 LCD、AD、定时器、QSPI、I2C 与 Flash 存储器在线编程等。11-15 分别给出 MCF52235 的 CAN 总线、基于 MCF52233 嵌入式以太网、MCF52233 的其他模块、MCF52223 的 USB 2.0 编程、 μ C/OS-II 在 ColdFire 上的移植与应用，第 16 章给出基于硬件构件的嵌入式系统开发方法。附录 A 给出了本书使用的 ColdFire 系列 MCU 芯片封装；附录给出了写入器、评估板及购买方法；附录 C 给出了常见实践问题解答。

本书网上光盘下载地址:

为方便读者，本书网上光盘可从下列地址之一进行下载，给出多个地址的目的是防止从某一地址下载不成功。作者可能每半年对下载内容进行更新，敬请读者关注。

电子工业出版社：<http://www.phei.com.cn>；苏州大学飞思卡尔嵌入式系统研发中心：<http://sumcu.suda.edu.cn>；苏州苏大万佳技术有限公司：<http://sueee.com.cn>；昆山鑫盛盟创

科技有限公司：<http://www.xsmc.net>；上海卓嘉电子有限公司：<http://www.can-lin.com>。

致谢：

本书除封面署名作者外，还有苏州大学计算机科学与技术学院嵌入式应用方向研究生王玉丽、屯娜、姚键东、王凤林、钱仇圣、舒胜强、钟海林等协助书稿整理及程序调试工作，他们卓有成效的工作，使本书更加实用。飞思卡尔半导体有限公司的 Andy Mastronardi 先生、马莉女士一直关心支持苏州大学飞思卡尔嵌入式系统研发中心的建设，为本书的撰写提供了硬件及软件资料，并提出了许多宝贵建议。飞思卡尔半导体有限公司的许多技术人员提供了技术支持。电子工业出版社的田宏峰先生为本书的出版付出了大量细致工作。在此一并表示诚挚的谢意。

鉴于作者水平有限，书中难免存在不足和错误之处，恳望读者提出宝贵意见和建议，以便再版时改进。

作 者

2009 年 7 月于苏州大学

目 录

第1章 概述	1
1.1 嵌入式系统定义、由来及特点	1
1.1.1 嵌入式系统的定义	1
1.1.2 嵌入式系统的由来及其与微控制器的关系	1
1.1.3 嵌入式系统的特点	3
1.2 嵌入式系统的知识体系、学习误区及学习建议	4
1.2.1 嵌入式系统的知识体系	4
1.2.2 嵌入式系统的学习误区	5
1.2.3 基础阶段的学习建议	8
1.3 嵌入式系统常用术语	9
1.3.1 与硬件相关的术语	9
1.3.2 与通信相关的术语	10
1.3.3 与功能模块及软件相关的术语	11
1.4 嵌入式系统常用的标准 C 语言基本语法概要	12
1.5 本书学习与实践资料索引	22
第2章 COLDFIRE概述与MCF52233硬件最小系统	24
2.1 ColdFire 系列微处理器概述	24
2.2 MCF5223x 系列微控制器存储器映像与编程结构	25
2.2.1 MCF5223x 性能概述与内部结构简图	26
2.2.2 MCF5223x 存储器映像	27
2.2.3 编程模式与寻址方式	28
2.2.4 ColdFire 异常和中断控制	32
2.3 MCF52233 的引脚功能与硬件最小系统	33
2.3.1 MCF52233 的引脚功能	33
2.3.2 MCF52233 的硬件最小系统	35
2.3.3 硬件最小系统测试方法	39
第3章 第一个样例程序及COLDFIRE工程组织	41
3.1 通用 I/O 接口基本概念及连接方法	41
3.2 MCF52233 的 GPIO	42
3.2.1 MCF52233 GPIO 编程的基本原理	42
3.2.2 GPIO 模块寄存器与 GPIO 的编程基本方法	43
3.3 开发套件 CodeWarrior 开发环境与 ColdFire 写入器	46

3.3.1 CodeWarrior 开发环境简介与基本使用方法.....	46
3.3.2 WA- I 型 32 位 ColdFire 写入器.....	47
3.3.3 MCF52233EVB 硬件评估板.....	48
3.4 CW 工程文件组织.....	48
3.4.1 工程文件的组织.....	48
3.4.2 初始化相关文件.....	51
3.4.3 主程序、中断程序及其他文件.....	54
3.4.4 链接文件及机器码文件.....	56
3.5 第一个应用实例：控制小灯闪烁.....	60
3.5.1 GPIO 构件.....	60
3.5.2 Light 构件.....	64
3.5.3 Light 测试工程主程序.....	66
3.6 理解第一个 C 工程的执行过程.....	67
第4章 异步串行通信与直接存储器访问DMA.....	70
4.1 异步串行通信的基础知识.....	70
4.1.1 基本概念.....	70
4.1.2 RS-232C 总线标准.....	72
4.1.3 电平转换电路原理.....	73
4.2 MCF52233 的 UART 模块的物理结构.....	74
4.3 MCF52233 的 UART 模块的编程结构.....	78
4.4 基于构件方法的 UART 编程.....	81
4.4.1 UART 构件的函数原型设计.....	81
4.4.2 UART 构件的头文件.....	81
4.4.3 UART 构件的源程序文件.....	84
4.4.4 UART 构件的测试工程.....	88
4.5 ColdFire 第一个带有中断功能的实例.....	90
4.6 MCF52233 的四通道 DMA 控制器.....	92
4.6.1 DMA 概述.....	92
4.6.2 MCF52233 四通道 DMA 特点.....	93
4.6.3 MCF52233 的 DMA 控制器编程结构.....	93
4.7 MCF52233 的 DMA 编程要点.....	96
4.7.1 DMA 初始化.....	97
4.7.2 DMA 启动序列与重启序列.....	97
4.8 基于构件的带 DMA 的 UART 编程实例.....	98
4.8.1 DMA 构件的函数原型设计.....	98
4.8.2 DMA 构件的头文件.....	98
4.8.3 DMA 构件的源程序文件.....	101

4.8.4 DMA 方式收发测试实例	105
4.9 进一步讨论	106
4.9.1 流控制与 Break 信号	106
4.9.2 延长串口通信的距离	107
4.9.3 串口的扩展	107
第5章 GPIO的应用实例--键盘、LED与LCD.....	109
5.1 键盘	109
5.1.1 键盘模型及接口	109
5.1.2 键盘编程的基本问题	110
5.1.3 键盘编程实例	110
5.2 数码管 LED	116
5.2.1 扫描法编程原理	117
5.2.2 数码管编程实例	118
5.3 液晶 LCD	121
5.3.1 HD44780 的编程结构.....	121
5.3.2 点阵字符型 LCD 编程实例	122
第6章 MCF52233的AD转换模块	129
6.1 AD 转换的基本问题.....	129
6.2 MCF52233 的 AD 转换结构与特性	130
6.3 MCF52233 的 AD 模块编程寄存器	131
6.4 基于构件的 AD 编程.....	140
6.4.1 AD 构件的函数原型设计.....	140
6.4.2 AD 构件的头文件.....	140
6.4.3 AD 构件的源程序文件	141
6.5 MCF52233 AD 模块编程实例——物理量采集回归系统.....	144
6.5.1 低端软件设计	144
6.5.2 高端软件设计	146
6.6 常用传感器简介及一个典型信号采样电路设计.....	147
6.6.1 常用传感器简介	147
6.6.2 电阻型传感器采样电路设计	149
第7章 定时器相关模块及其应用	151
7.1 可编程中断定时器 PIT	151
7.1.1 PIT 模块的编程寄存器	151
7.1.2 PIT 模块的运行模式与编程方法	152
7.1.3 PIT 模块的编程实例	153

7.2 通用定时器 GPT 模块.....	155
7.2.1 计数器/定时器的基本工作原理	155
7.2.2 定时器模块的定时计数功能	157
7.2.3 定时器模块的输入捕捉功能	160
7.2.4 定时器模块的输出比较功能	164
7.2.5 定时器模块的脉冲累加功能	168
7.3 脉宽调制模块 PWM 与 D/A 转换.....	171
7.3.1 脉宽调制模块 PWM 基本概念	171
7.3.2 脉宽调制模块 PWM 的编程寄存器.....	172
7.3.3 脉宽调制模块 PWM 的编程实例	175
7.3.4 脉宽调制模块 PWM 的进一步讨论.....	182
7.3.5 用 PWM 实现 D/A 转换	182
第8章 队列式串行外设接口 QSPI.....	184
8.1 SPI 的基本工作原理.....	184
8.1.1 SPI 基本概念.....	184
8.1.2 SPI 的数据传输.....	185
8.1.3 SPI 模块的时序.....	185
8.1.4 模拟 SPI	187
8.2 MCF52233 队列串行外设接口 QSPI 模块编程结构	188
8.2.1 QSPI 概述.....	188
8.2.2 QSPI 工作原理.....	188
8.2.3 QSPI 功能描述.....	190
8.2.4 QSPI 模块寄存器.....	190
8.2.5 发送延时	194
8.3 MCF52233 的串行外设接口 QSPI 模块编程实例	195
第9章 I2C模块及应用实例.....	201
9.1 I2C 模块	201
9.1.1 I2C 总线特点	201
9.1.2 I2C 总线标准的发展历史	202
9.1.3 I2C 总线的相关术语	202
9.2 I2C 总线工作原理	202
9.2.1 总线上数据的有效性	203
9.2.2 总线上的信号	203
9.2.3 总线上数据传输格式	204
9.2.4 I2C 总线寻址约定	205
9.2.5 主机向从机读写 1 个字节数据的过程.....	205

9.3 I2C 模块的编程基础	206
9.3.1 MCF52233 的 I2C 模块	206
9.3.2 MCF52233 的 I2C 模块寄存器介绍	207
9.3.3 MCF52233 的 I2C 模块编程	210
9.4 MCF52233 的 I2C 模块的进一步讨论	215
9.4.1 仲裁程序	215
9.4.2 实现数据传输同步交换	216
第10章 FLASH在线编程	217
10.1 Flash 存储器概述	217
10.2 ColdFire Flash 存储器编程方法	218
10.2.1 ColdFire Flash 模块寄存器	219
10.2.2 ColdFire Flash 存储器擦除与写入的步骤	221
10.3 ColdFire Flash 擦写函数的测试工程	226
10.4 ColdFire Flash 存储器的保护特性和安全性	227
10.4.1 ColdFire Flash 存储器的保护特性	227
10.4.2 ColdFire Flash 存储器的安全性	227
第11章 MCF52235 FLEXCAN模块编程方法	232
11.1 CAN 总线通用知识	232
11.1.1 CAN 总线协议的历史概况	232
11.1.2 CAN 硬件系统的典型电路	232
11.1.3 CAN 总线的有关基本概念	234
11.1.4 帧结构	238
11.1.5 位时间	242
11.2 MCF52235 FlexCAN 模块概述与编程结构	243
11.2.1 FlexCAN 特性	243
11.2.2 操作模式	244
11.2.3 FlexCAN 模块的内存映像以及寄存器定义	245
11.2.4 FlexCAN 报文缓冲区 (Message Buffer, MB)	251
11.3 MCF52235 FlexCAN 模块报文发送与接收函数设计	254
11.3.1 数据帧发送/接收	254
11.3.2 远程帧发送/接收	258
11.3.3 仲裁处理、匹配处理及报文缓冲区管理	261
11.4 MCF52235 FlexCAN 模块编程实例	262
11.4.1 初始化函数设计	262
11.4.2 MCF52235 FlexCAN 模块测试工程说明	264
11.5 进一步讨论	266

第12章 基于MCF52233的嵌入式以太网	270
12.1 嵌入式以太网相关基础知识	270
12.1.1 以太网的由来与协议模型	270
12.1.2 以太网中主要物理设备	273
12.1.3 相关名词解释	275
12.2 MCF52233 以太网模块外部连接及编程基础	278
12.2.1 以太网模块引脚及硬件连接	278
12.2.2 以太网物理层收发器 EPHY 的编程基础	280
12.2.3 第一个测试实例：网络连接	285
12.2.4 快速以太网控制器 FEC 的编程基础	289
12.3 链路层编程	298
12.3.1 FEC 初始化函数 FEC_Init	298
12.3.2 以太帧发送	302
12.3.3 以太帧接收	304
12.3.4 第二个测试实例：以太帧的发送和接收	305
12.3.5 第三个测试实例：链路层 ARP 协议编程	308
12.4 网络层的 IP 协议与 ICMP 协议编程	310
12.4.1 IP 协议简介	311
12.4.2 第四个测试实例：使用 ICMP 协议响应 Ping 请求	313
12.5 运输层的 UDP 与 TCP 协议编程	314
12.5.1 UDP 协议概述	314
12.5.2 第五个测试实例：UDP 报文的发送和接收	315
12.5.3 TCP 协议概述	316
12.5.4 第六个测试实例：TCP 报文的发送和接收	317
12.6 应用层的 HTTP 协议编程	320
12.6.1 HTTP 协议概述	320
12.6.2 第七个测试实例：HTTP 协议静态页面的实现	322
第13章 其它模块	324
13.1 时钟模块	324
13.2 芯片配置模块	325
13.2.1 芯片配置模块简介	326
13.2.2 芯片配置模块寄存器定义	326
13.3 系统控制模块	327
13.3.1 系统控制模块功能概述	327
13.3.2 系统控制模块的可编程寄存器	327
13.3.3 对系统初始化过程的理解	329
13.4 实时时钟模块	331

13.5 电源管理模块	333
13.5.1 低功耗模式	333
13.5.2 低功耗模式下的外设行为	333
13.5.3 电源管理模块寄存器定义	334
13.6 复位控制模块	334
第14章 基于MCF52223的USB 2.0编程	336
14.1 USB 基本概念及硬件特性.....	336
14.1.1 USB 概述	336
14.1.2 USB 相关基本概念.....	337
14.1.3 USB 的物理特性.....	342
14.2 USB 的通信协议	344
14.2.1 USB 基本通信单元：包.....	344
14.2.2 USB 通信中的事务处理.....	346
14.2.3 从设备的枚举看 USB 数据传输.....	347
14.3 MCF52223 硬件最小系统及 USB 模块功能简介	352
14.3.1 MCF52223 硬件最小系统	352
14.3.2 MCF52223 的 USB 模块功能简介	353
14.3.4. MCF52223 的 USB 模块主要寄存器介绍	354
14.4 MCF52223 作为 USB 从机的开发方法	357
14.4.1 PC 方 USB 设备驱动程序的选择及基本原理	357
14.4.2 PC 机作为 USB 主机的程序设计	361
14.4.3 MCF52223 作为 USB 从机的程序设计	364
14.5 MCF52223 作为 USB 主机的开发方法	367
14.5.1 MCF52223 作为 USB 主机的基本功能	367
14.5.2 USB 主机与 CDC 类 USB 设备通信.....	369
14.5.3 USB 主机与 MassStorage 类 USB 设备通信	369
14.6 采用 OTG 技术的 USB 系统开发方法	370
第15章 μC/OS-II 在COLDFIRE上的移植与应用	373
15.1 实时操作系统概述	373
15.1.1 实时操作系统基本含义和要求	373
15.1.2 任务和多任务	374
15.1.3 RTOS 的常用术语	374
15.1.4 RTOS 的现状	375
15.2 嵌入式实时操作系统 μ C/OS-II	377
15.2.1 μ C/OS-II 简介	377
15.2.2 μ C/OS-II 与几种 RTOS 的比较	377

15.2.3 μ C/OS-II 中的任务	378
15.2.4 μ C/OS-II 用户应用程序的一般结构	379
15.3 μ C/OS-II 的任务划分	380
15.3.1 任务划分的目标	380
15.3.2 任务划分的方法	380
15.4 μ C/OS-II 在 MCF52233 上的应用实例	382
15.4.1 基于 μ C/OS-II 的应用实例	382
15.4.2 与无 RTOS 实例的比较	384
15.5 μ C/OS-II 在 MCF52233 上的移植	384
15.5.1 定义内核的大小和功能	385
15.5.2 与硬件相关的代码	386
第16章 基于硬件构件的嵌入式系统开发方法	390
16.1 嵌入式系统开发所遇到的若干问题	390
16.2 嵌入式硬件构件的基本思想与应用方法	390
16.3 基于硬件构件的嵌入式系统硬件电路设计原则	391
16.3.1 设计时需要考虑的基本问题	391
16.3.2 硬件构件化电路原理图绘制规则	392
16.3.3 PCB 设计基本原则	394
16.4 基于硬件构件的嵌入式底层软件构件的编程方法	395
16.4.1 嵌入式硬件构件和软件构件的层次模型	395
16.4.2 底层构件的实现方法与编程思想	396
16.4.3 硬件构件及底层软件构件的重用与移植方法	397
16.5 硬件构件化嵌入式开发技术的应用实例	399
16.5.1 系统功能	399
16.5.2 硬件设计	399
16.5.2 软件设计	404
附录 A 本书使用的 ColdFire 系列 MCU 芯片封装	407
附录 B 写入器、评估板及购买方法	408
附录 C 常见实践问题解答	410
主要参考书目	415

第1章 概述

作为全书导引，本章内容与具体芯片无关。主要内容有：简要给出嵌入式系统定义、由来及特点；简要阐述嵌入式系统的知识体系，分析如何避免进入嵌入式系统的学习误区，根据嵌入式系统的特点，就如何学习嵌入式系统提出几点建议；归纳嵌入式系统的常用术语；给出嵌入式系统常用的标准 C 语言基本语法概要；对本书提供的学习与实践资料给出概要说明与索引。

1.1 嵌入式系统定义、由来及特点

1.1.1 嵌入式系统的定义

嵌入式系统（Embedded system）有多种多样的定义，但本质是相同的。本书关于嵌入式系统的定义取自美国 CMP Books 出版的 Jack Ganssle 和 Michael Barr 著作《Embedded System Dictionary》^①。

嵌入式系统的定义：一种计算机硬件和软件的组合，也许还有机械装置，用于实现一个特定功能。在某些特定情况下，嵌入式系统是一个大系统或产品的一部分。世界上第一个嵌入式系统是 1971 年 Busicom 公司用 Intel 单芯片 4004 微处理器完成的商用计算器系列。该词典还给出了嵌入式系统的一些示例：微波炉、手持电话、计算器、数字手表、录像机、巡航导弹、GPS 接收机、数码相机、传真机、跑步机、遥控器和谷物分析仪等，难以尽数。通过与通用计算机的对比可以更形象地理解嵌入式系统的定义。该词典给出的通用计算机定义是：计算机硬件和软件的组合，用做通用计算平台。PC、MAC 和 Unix 工作站是最流行的现代计算机。

我国《国家标准 GB/T 5271 信息技术词汇—嵌入式系统与单片机》部分，给出的嵌入式系统定义是：置入应用对象内部起操作控制作用的专用计算机系统。

国内对嵌入式系统定义曾进行过广泛讨论，有许多不同说法。其中嵌入式系统定义涵盖面问题是主要争论焦点之一。例如，有的学者认为不能把手持电话叫嵌入式系统，而只能把其中起控制作用的部分叫嵌入式系统，而手持电话可以称为嵌入式系统的应用产品。其实，这些并不妨碍人们对嵌入式系统的理解，所以不必对定义感到困惑。有些国内学者特别指出，在理解嵌入式系统定义时，不要把嵌入式系统与嵌入式系统产品相混淆。实际上，从口语或书面语言角度，不区分“嵌入式系统”与“嵌入式系统产品”，只要不妨碍对嵌入式系统的理解就没有关系。

为了更清楚阐述嵌入式系统特点，首先介绍大多数嵌入式系统的核心部件—单片机

^① 中译本：Jack Ganssle 等著，马广云等译，《英汉双解嵌入式系统词典》，北京航空航天大学出版社，2006 年。

（微控制器）的基本概念。

1.1.2 嵌入式系统的由来及其与微控制器的关系

1. 单片机（微控制器）的基本含义

单片机是单片微型计算机的简称，早期的英文名是 Single-chip Microcomputer，后来大多数称之为微控制器或嵌入式计算机（Embedded computer）。现在 Microcontroller 已经是计算机中一个常用术语，但在 1990 年代之前，大部分英文词典并没有这个词。我国学者一般使用中文“单片机”一词，而缩写使用“MCU”^①。所以本书后面的简写一律以 MCU 为准。MCU 的基本含义是：在一块芯片上集成了中央处理单元（CPU）、存储器（RAM/ROM 等）、定时器/计数器及多种输入输出（I/O）接口的比较完整的数字处理系统。

图 1-1 给出了典型的 MCU 组成框图。

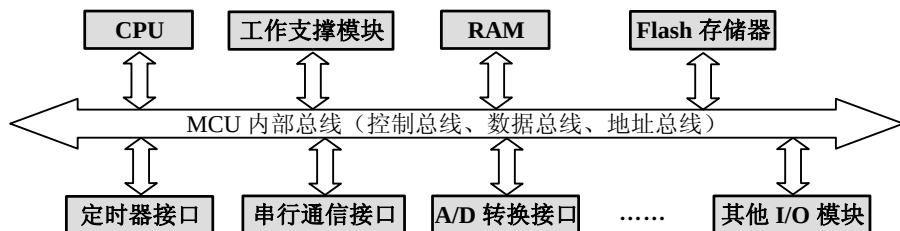


图1-1 一个典型的MCU内部框图

MCU 是在计算机制造技术发展一定阶段的背景下出现的，它使计算机技术从科学计算领域进入到智能化控制领域。从此，计算机技术在两个重要领域——通用计算机领域和嵌入式（Embedded）计算机领域都获得了极其重要的发展，为计算机的应用开辟了更广阔的空间。

就 MCU 组成而言，虽然它只是一块芯片，但包含了计算机的基本组成单元，仍由运算器、控制器、存储器、输入设备、输出设备五部分组成，只不过这些都集成在一块芯片上，这种结构使得 MCU 成为具有独特功能的计算机。

2. 嵌入式系统的由来

通俗地说，计算机是因科学家需要一个高速的计算工具而产生的。直到二十世纪七十年代，电子计算机在数字计算、逻辑推理及信息处理等方面表现出非凡的能力。在通信、测控与数据传输等领域，人们对计算机技术给予了更大的期待。这些领域的应用与单纯的高速计算要求不同，主要表现在：直接面向控制对象；嵌入到具体的应用体中，而非计算机的面貌出现；能在现场连续可靠地运行；体积小，应用灵活；突出控制功能，特别是对外部信息的捕捉与丰富的输入输出功能等。由此可以看出，满足这些要求的计算机与满足高速数值计算的计算机是不同的。因此，一种称之为单片机或微控制器的技术得以产生并发展。为了区分这两种计算机类型，通常把满足海量高速数值计算的计算机称为通用计算

^① MCU 的英文全称是 Microcontroller Unit。

机系统,而把嵌入到实际应用系统中,实现嵌入式应用的计算机称之为嵌入式计算机系统,简称嵌入式系统。

3. 嵌入式系统与MCU的关系

何立民先生说:“有些人搞了十多年的单片机应用,不知道单片机就是一个最典型的嵌入式系统”^①。实际上,MCU是在通用CPU基础上发展起来的,MCU具有体积小、价格低、稳定可靠等优点,它的出现和迅猛发展,是控制系统领域的一场技术革命。MCU以其较高的性能价格比、灵活性等特点,在现代控制系统中具有十分重要的地位。大部分嵌入式系统以MCU为核心进行设计。MCU从体系结构到指令系统都是按照嵌入式系统的应用特点专门设计的,它能很好地满足应用系统的嵌入、面向测控对象、现场可靠运行等方面的要求。因此以MCU为核心的系统是应用最广的嵌入式系统。在实际应用时,开发者是根据具体要求选用最佳型号的MCU嵌入到实际应用系统中。

在MCU出现之前,人们必须用模拟电路、数字电路实现大部分计算与控制功能,这样使得控制系统体积庞大,易出故障。MCU出现以后,情况发生了变化,系统中的大部分计算与控制功能由MCU的软件实现。其它电子线路成为MCU的外围接口电路,承担着输入、输出与执行动作等功能,而计算、比较与判断等原来必须用电路实现的功能,可以用软件取代,大大地提高了系统的性能与稳定性,这种控制技术称之为嵌入式控制技术。在嵌入式控制技术中,核心是MCU,其它部分依此而展开。

1.1.3 嵌入式系统的特点

要谈嵌入式系统特点,不同学者也许有不同说法。这里从与通用计算机对比角度谈嵌入式系统的特点。

1. 嵌入式系统是计算机系统,但不单独以通用计算机面目出现

嵌入式系统的本名叫嵌入式计算机系统(Embedded computer system),它不仅具有通用计算机的主要特点,又具有自身特点。嵌入式系统也必须要有软件才能运行,但其隐含在种类众多的具体产品中。同时,通用计算机种类屈指可数,而嵌入式系统不仅芯片种类繁多,而且由于应用对象大小各异,嵌入式系统作为控制核心,已经融入到各个行业的产品之中。

2. 嵌入式系统开发需要专用工具和特殊方法

嵌入式系统不像通用计算机那样有了计算机系统就可以进行应用开发。一般情况下,MCU芯片本身不具备开发功能,必须要有一套与相应芯片配套的开发工具和开发环境。这些工具和环境一般基于通用计算机上的软硬件设备以及各种逻辑分析仪、混合信号示波器等。开发时往往有主机和目标机的概念,主机用于程序的开发,目标机作为程序的执行机,开发时需要交替结合进行。

^① 《单片机与嵌入式系统应用》,2004年第1期。

3. 使用MCU设计嵌入式系统，数据与程序空间采用不同存储介质

在通用计算机系统中，程序存储在硬盘上。实际运行时，通过操作系统将要运行的程序从硬盘调入内存（RAM），运行中的程序、常数、变量均在 RAM 中。而以 MCU 为核心的嵌入式系统，其程序被固化到非易失性存储器中^①。变量及堆栈使用 RAM 存储器。

4. 开发嵌入式系统涉及软件、硬件及应用领域的知识

嵌入式系统与硬件紧密相关，嵌入式系统的开发需要硬件、软件协同设计、协同测试。同时，由于嵌入式系统专用性很强，通常是用在特定应用领域，如嵌入在手机、冰箱、空调、各种机械设备、智能仪器仪表中起核心控制作用，功能专用。因此，进行嵌入式系统的开发，还需要对领域知识有一定的理解。当然，一个团队协作开发一个嵌入式产品，其中各个成员可以扮演不同角色，但对系统的整体理解与把握并相互协作，有助于一个稳定可靠嵌入式产品的诞生。

5. 嵌入式系统的其他特点

除了以上特点之外，嵌入式系统还具有其他方面的特点。

在资源方面：嵌入式系统通常专用于某一特定应用领域，其硬件资源不会像通用计算机那样丰富；**在可靠性方面：**嵌入式系统一般要求更高可靠性和稳定性；**在实时性方面：**相当多嵌入式系统有实时性要求；**在成本方面：**嵌入式系统通常极其关注成本；**在功耗要求方面：**一些嵌入式系统要求低功耗；**在生命周期方面：**嵌入式系统通常比通用计算机系统生命周期长，升级换代比通用计算机慢。**在知识综合方面：**嵌入式系统是将先进的计算机技术、半导体技术及电子技术与各个行业的具体应用相结合的产物，是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。它的构成既有硬件又有软件，不仅包括应用软件，也可能包括系统软件。它有数字电路又有模拟电路。其产品技术含量高，涉及多种学科，不容易开发，因此也不容易形成技术垄断。

这些特点决定了嵌入式系统的开发方法、开发难度、开发手段等，均不同于通用计算机，也不同于常规的电子产品。

1.2 嵌入式系统的知识体系、学习误区及学习建议

1.2.1 嵌入式系统的知识体系

嵌入式系统的应用范围可以粗略分为两大类：（1）电子系统的智能化（工业控制，现代农业、家用电器、汽车电子、测控系统、数据采集等）；（2）计算机应用的延伸（MP3、手机、通信、网络、计算机外围设备等）。从这些应用可以看出，要完成一个以 MCU 为核心的嵌入式系统应用产品设计，需要有硬件、软件及行业领域相关知识。硬件主要有 MCU 的硬件最小系统、输入/输出外围电路、人机接口设计。软件设计有固化软件的设计，

^① 目前，非易失性存储器通常为 Flash 存储器，特点见“第10章 Flash 存储器在线编程”。

也可能含 PC 机软件的设计。行业知识需要通过协作、交流与总结获得。

概括地说,学习以 MCU 为核心的嵌入式系统,需要以下软件硬件基础知识与实践训练:

- (1) 硬件最小系统(包括电源、晶振、复位、写入调试接口);
- (2) 通用 I/O(开关量输入/输出,涉及各种二值量检测与控制);
- (3) 模数转换 A/D(各种传感器信号的采集与处理,如红外、温度、光敏、超声波、方向等);
- (4) 数模转换 D/A(对模拟量设备利用数字进行控制);
- (5) 通信(串行通信接口 SCI、串行外设接口 SPI、集成电路互联总线 I2C, CAN、USB、嵌入式以太网、ZigBee 技术等);
- (6) 显示(LED、LCD 等);
- (7) 控制(控制各种设备,包括 PWM 等控制技术);
- (8) 数据处理(图形、图像、语音、视频等处理或识别);
- (9) 各种具体应用。

事实上,万变不离其宗,任何应用都可以归入这几类。而应用中的硬件设计、软件设计、测试等都必须遵循嵌入式软件工程的方法、原理与基本原则。所以,嵌入式软件工程也是嵌入式系统知识体系的有机组成部分,只不过,它融于具体项目的开发过程之中。

以上实践训练涉及硬件基础、软件基础及相关领域知识。计算机语言、操作系统、开发环境等均是完成这些目的的工具。有些初学者,容易把工具的使用与所要达到的真正目的相混淆。例如,有的学习者,学了很长时间的嵌入式操作系统移植,而不进行实际嵌入式系统产品的开发,到了最后,做不好一个嵌入式系统小产品,偏离了学习目标,甚至放弃了嵌入式系统领域。这就是进入了嵌入式系统学习误区的情况,下面对此做一些分析。

1.2.2 嵌入式系统的学习误区

关于嵌入式系统的学习方法,因学习经历、学习环境、学习目的、已有的知识基础等不同,可能在学习顺序、内容选择、实践方式等方面有所不同。但是,应该明确哪些是必备的基础知识,哪些应该先学,哪些应该后学;哪些必须通过实践才能获得的;哪些是与具体芯片无关的通用知识,哪些是与具体芯片或开发环境相关的知识。

由于微处理器与微控制器种类繁多,也可能由于不同公司、不同机构出于自身的利益,给出一些误导性宣传,特别是我国嵌入式微控制器制造技术的落后及其他相关情况,使得人们对微控制器的发展,在认识与理解上存在差异,导致一些初学者,进入了嵌入式系统的学习误区,浪费了宝贵的学习时间。下面分析初学者可能存在的几个误区。

1. 嵌入式系统学习误区1—操作系统的困惑

如果说,学习嵌入式系统不是为了开发其应用产品,那就没有具体目标了,许多诸如学习方法问题也就不必谈了。实际上,这正是许多人想学,又不知从何开始学习的关键问题所在,不知道自己学习的具体目标。于是,看了一些培训广告,看了书店中书架上种类繁多的嵌入式系统的书籍,或上网以“嵌入式系统”为关键词进行查询,然后参加培训或

看书，开始“学习起来”。对于有计算机阅历的人，往往选择一个嵌入式操作系统就开始学习了。不好听的比喻，有点象“瞎子摸大象”，只了解其一个侧面。这样如何能对嵌入式产品的开发过程有个全面了解呢？针对许多初学者选择“xxx 嵌入式操作系统+xxx 处理器”的嵌入式系统入门学习模式，我认为是不合适的。我的建议是：首先把嵌入式系统软件与硬件基础打好了，再根据实际需要，选择一种实时操作系统（RTOS）进行学习实践。要记住：RTOS 是开发某些类嵌入式产品的辅助工具，是手段，不是目的。许多类嵌入式产品，并不需要 RTOS。所以，一开始就学习 RTOS，并不符合“由浅入深、循序渐进”的学习规律。

RTOS 本身由于种类繁多，实际使用何种 RTOS，一般需要工作单位确定。基础阶段主要学习 RTOS 的基本原理与在 RTOS 之上的软件开发方法，而不是学习如何设计 RTOS。以开发实际嵌入式产品为目标的学习者，不要把过多的精力花在设计或移植 RTOS 上面。正如很多人使用 Windows 操作系统，而设计 Windows 操作系统只有 Microsoft。许多人“研究”Linux，但不使用它，浪费时间了，人的精力是有限的，学习必须有所选择。

2. 嵌入式系统学习误区2－硬件与软件的困惑

以 MCU 为核心的嵌入式技术的知识体系必须通过具体的 MCU 来体现、实践与训练。但是，选择任何型号的 MCU，其芯片相关的知识只占知识体系的 20%，80%是通用知识。但是 80%的通用知识，必须通过具体实践才能进行，所以学习嵌入式技术要选择一个系列的 MCU。但不论如何，系统含有硬件与软件两大部分，它们之间的关系如何？

有些学者，仅从电子角度认识嵌入式系统。认为“嵌入式系统=MCU 硬件系统+小程序”。这些学者，大多具有良好的电子技术基础知识。实际情况是，早期 MCU 内部 RAM 小、程序存储器外接，需要外扩各种 I/O，没有象现在这样 USB、嵌入式以太网等较复杂的接口，因此，程序占总设计量小于 50%，使人们认为嵌入式系统（单片机）是“电子系统”，以硬件为主、程序为辅。但是，随着 MCU 制造技术的发展，不仅 MCU 内部 RAM 越来越大，Flash 进入 MCU 内部改变了传统的嵌入式系统开发与调试方式，固件程序可以被更方便地调试与在线升级，许多情况与开发 PC 机程序方便程度相差无几，只不过开发环境与运行环境不是同一载体而已。这些情况使得嵌入式系统的软件硬件设计方法发生了根本变化。

有些学者，仅从软件开发角度认识嵌入式系统。甚至有的仅从嵌入式操作系统认识嵌入式系统。这些学者，大多具有良好的计算机软件开发基础知识，认为硬件是生产厂商的事，没有认识到，嵌入式系统产品的软件与硬件均是需要开发者设计的。我常常接到一些关于嵌入式产品稳定性的咨询电话，发现大多数是由于软件开发对底层硬件的基本原理不理解造成的。特别是，有些功能软件开发，过分依赖于底层硬件的驱动软件设计完美，自己对底层驱动原理知之甚少。实际上，一些功能软件开发，名义上在做嵌入式软件，但仅仅是使用嵌入式编辑、编译环境而已，本质与开发通用 PC 机软件没有两样。而底层硬件驱动软件的开发，若不全面考虑高层功能软件对底层硬件的可能调用，也会使得封装或参数设计得不合理或不完备，导致高层功能软件的调用困难。从这段描述可以看出，若把一个嵌入式系统的开发孤立地分为硬件设计、底层硬件驱动软件设计、高层功能软件设

计，一旦出现了问题，就可能难以定位。实际上，嵌入式系统设计是一个软件、硬件协同设计工程，不能象通用计算机那样，软件、硬件完全分开来看，要在一个大的框架内协调工作。

面对学习嵌入式系统以软件为主还是以硬件为主，或是如何选择切入点，如何在软件与硬件之间取得一些平衡。对于这个困惑的建议是：要想成为一名真正的嵌入式系统设计师，在初学阶段，必须重视打好嵌入式系统的硬件与软件基础。以下是从事嵌入式系统设计二十多年的一个美国学者 John Catsoulis 在《Designing Embedded Hardware》一书中关于这个问题的总结：嵌入式系统与硬件紧密相关，是软件与硬件的综合体，没有对硬件的理解就不可能写好嵌入式软件，同样没有对软件的理解也不可能设计好嵌入式硬件。

3. 嵌入式系统学习误区3－片面认识嵌入式系统

嵌入式系统产品种类繁多，应用领域各异。在 1.2.1 小节中，把嵌入式系统的应用范围粗略分为电子系统的智能化与计算机应用的延伸两大类，从初学者角度，可能有分别从这两个角度片面认识嵌入式系统的问题。因此，一些从电子系统智能化角度认识嵌入式系统的学习者，可能会忽视编程结构、编程规范、软件工程的要求、操作系统等知识的积累。另一些从计算机应用的延伸角度认识嵌入式系统的学习者，可能会把通用计算机学习过程中的概念与方法生搬硬套到嵌入式系统的实践中，忽视嵌入式系统与通用计算机的差异。

实际上，在嵌入式系统学习与实践的初始阶段，应该充分了解嵌入式系统的特点，根据自身的已有知识结构，制定适合自身情况的学习计划。目标应该是打好嵌入式系统的硬件与软件基础，通过实践，为成为良好的嵌入式系统设计师建立起基本知识结构。学习过程，可以通过具体应用系统为实践载体，但不能拘泥于具体系统，应该有一定的抽象与归纳。例如，有的初学者开发一个实际控制系统，没有使用实时操作系统，但不要认为实时操作系统不需要学习。又例如，有的初学者以一个带有实时操作系统的样例为蓝本进行学习，但不要认为，任何嵌入式系统都需要使用实时操作系统，甚至把一个十分简明的实际系统加上一个不必要的实时操作系统。因此，片面认识嵌入式系统，可能导致学习困惑。应该根据实际项目需要，锻炼自己分析实际问题、解决问题的能力。这是一个比较长期的学习与实践过程，不能期望通过短期培训完成整体知识体系的建立，应该重视自身实践，全面地理解嵌入式系统的知识体系。

4. 嵌入式系统学习误区4－入门芯片选择的困惑

嵌入式系统的大部分初学者需要选择一个微控制器（MCU）进行入门级学习，面对众多厂家生产的微控制器系列，不知如何是好。

首先是关于位数问题，目前主要有 8 位、16 位、32 位，面对嵌入式系统应用的多样性，不同位数的 MCU 各有应用领域，这一点与通用微机有很大不同。你做一个遥控器，不需要使用一个 32 位 MCU，可能一个 MCU 芯片价格已经超过遥控器价格需求。对于首次接触嵌入式系统的学习者，可以根据自己的知识基础选择入门芯片的位数。建议大多数初学者，可以选择一个 8 位 MCU 作为快速入门芯片，了解一些汇编与底层硬件知识，之后再选一个 16 位或 32 位芯片进行学习实践。

关于芯片选择的另一个误区，认为有“主流芯片”存在，嵌入式系统也可以形成芯片

垄断。这完全是一种误解，是套用通用计算机系统的思维模式，而忽视了嵌入式系统应用的多样性。

关于学习芯片选择还有一个误区，是系统的工作频率。误认为选择工作频率高的芯片进行入门学习，表示更先进。实际上，工作频率高可能给初学者带来学习过程中的不少困难。

实际嵌入式系统设计不是追求芯片位数、工作频率、操作系统等因素，而是追求稳定可靠、维护、升级、功耗、价格等指标。而初学者选择入门芯片，是通过某一 MCU 作为蓝本获得嵌入式系统知识体系的通用基础，其基本原则是：入门时间较快、硬件成本较少，知识要素较多，学习难度较低。

1.2.3 基础阶段的学习建议

基于以上讨论，下面对广大渴望学习嵌入式系统的学子提出几点基础阶段的学习建议：

(1) **嵌入式系统软件硬件密切相关，一定要打好软件硬件基础。**其实，只要找到正确的方法，加上努力，任何理工科学生，甚至非理工科学生，都能学好嵌入式系统。

(2) **选择一个芯片及硬件评估板**（入门芯片最好是简单一点，例如 8 位 MCU）、选择一本好书（最好有规范的例子）、找一位好老师（最好是有经验且热心的）。硬件评估板的价格一定要在 1000 元以下，不要太贵，最好能有自己动手的空间。不花一分硬件钱，要想学好嵌入式系统不实际。因为，这是实践性很强的学科。好书，可以使你少走弯路，不会被误导，要知道有的书是会使人进入学习误区的。好老师也可以是做过一些实际项目的学长（一定要找做过几个成功项目的学长或老师做指导，否则，经验不足也可能误导），有教师指导，学习进程会加快（人工智能学科里有个术语叫无教师指导学习模式与有教师指导学习模式，无教师指导学习模式比有教师指导学习模式复杂许多）。

(3) **许多人怕硬件，其实嵌入式系统硬件比电子线路好学多了。**只要深入理解 MCU 的硬件最小系统，对 I/O 口、串行通信、键盘、LED、LCD、SPI、I2C、PWM、A/D（包括一些传感器）、D/A 等逐个实验理解，逐步实践。再通过自己做一个实际的小系统，底层硬件基础就有了。各个硬件模块驱动程序的编写是嵌入式系统的必备基础。学习嵌入式系统的初期，这个过程是必须的。

(4) **至于嵌入式实时操作系统 RTOS，一定不要一开始就学，这样会走很多弯路，也会使你对嵌入式系统感到畏惧。**等你软件硬件基础打好了，再学习就感到容易理解。实际上，众多嵌入式应用，并不一定需要操作系统。也可以根据实际项目需要再学习特定的 RTOS。一定不要被一些嵌入式实时操作系统培训班宣传所误导，而忽视实际嵌入式系统软件硬件基础知识的学习。

(5) **要避免片面地单纯从“电子”或“计算机软件”角度认识嵌入式系统。**前面说过，嵌入式系统是软件与硬件的综合体。因此，要逐步从 MCU 的最小系统开始，一点一点理解硬件原理及底层硬件驱动编程方法。要通过规范的例子，理解软件工程封装、可复用等思想。通过规范编程，积累底层构件（Component），也就是一个一个模块，但是要封装得比较好，可复用。

(6) **注重实验与实践**。这里说的实验主要指通过重复或验证他人的工作，目的是学习基础知识，这个过程一定要经历。实践是自己设计，有具体的“产品”目标。如果你能花 500 元左右自己做一个具有一定功能的小产品，且能稳定运行 1 年以上，就可以说接近入门了。

(7) **关于入门芯片的选择**。不要选太复杂的微控制器作为入门芯片，不能超越学习过程。不要一下子学习几种芯片，可以通过一个芯片入门并具有一个实践经验后，根据实际需要选择芯片开发实际产品。注意，不要把微处理器（MPU）与微控制器（MCU）概念相混淆，微处理器只是微控制器的内核。

(8) **关于嵌入式操作系统的选择**。可以等到你具有一定实践后，选择一个简单容易理解原理的进行学习，不要一开始就学习几种操作系统，理解了基本原理，实践中确有实际需要再学习也不迟。人总是要不断学习的。

(9) **关于汇编与 C 语言的取舍**。随着 MCU 对 C 编译的优化支持，对于汇编可以只了解几个必须的语句，直接使用 C 语言编程。但必须通过第一个程序理解芯片初始化过程、中断机制、程序存储情况等区别于 PC 机程序的内容。另外，为了测试的需要，最好掌握一门 PC 机编程语言。

(10) **要明确自己的学习目的，并注意学习方法**。关于学习目的要明确是打基础，还是为了适应某些工作而进行的短训；而学习方法方面，要根据学习目的选择合适的学习途径，注意理论学习与实践、通用知识与芯片相关知识、硬件知识与软件知识的平衡，要在理解软件工程基本原理基础上理解硬件构件与软件构件等基本概念。

以上建议，仅供参考。当然，以上只是基础阶段的学习建议，要成为良好的嵌入式系统设计师，还需要在实际项目中锻炼，并不断学习与积累经验。

1.3 嵌入式系统常用术语

在学习嵌入式应用技术的过程中，经常会遇到一些名词术语。从学习规律角度，初步了解这些术语有利于随后的学习。因此，本节对嵌入式系统中所用的一些常用术语给出简要说明，以便有个初始印象。

1.3.1 与硬件相关的术语

1. 封装（Package）

集成电路的封装是指用塑料、金属或陶瓷材料等把集成电路封在其中。封装可以保护芯片，并使芯片与外部世界连接。常用的封装形式可分为通孔封装和贴片封装两大类。

通孔封装主要有：单列直插 SIP（Single-in-line Package）、双列直插 DIP（Dual-in-line Package）、Z 字型直插式封装 ZIP（Zigzag-in-line Package）等。

常见的贴片封装主要有：小外形封装 SOP（Small Outline Package）、紧缩小外形封装 SSOP（Shrink Small Outline Package）、四方扁平封装 QFP（Quad-Flat Package）、薄方封装 LQFP（Plastic-Low-profile Quad-Flat Package）、塑料扁平组件式封装 PFP（Plastic Flat

Package)、带载封装 TCP(Tape Carrier Package)、插针网格阵列封装 PGA(Ceramic Pin Grid Array Package)、球栅阵列封装 BGA(Ball Grid Array Package)等。

2. 印刷电路板(PCB, Printed Circuit Board)

印刷电路板 PCB 是组装电子元件用的基板,是在通用基材上按预定设计形成点间连接及印制元件的印制板,是电路原理图的实物化。PCB 的主要功能是提供集成电路等各种电子元器件固定、装配的机械支撑;实现集成电路等各种电子元器件之间的布线和电气连接(信号传输)或电绝缘;为自动装配提供阻焊图形,为元器件插装、检查、维修提供识别字符和图形等。

3. 动态可读写随机存储器(DRAM, Dynamic Random Access Memory)

动态可读写随机存储器 DRAM,由一个 MOS 管组成一个二进制存储位。MOS 管的放电导致表示“1”的电压会慢慢降低。一般每隔一段时间就要控制刷新信息,给其充电。DRAM 价格低,但控制烦琐,接口复杂。

4. 静态可读写随机存储器(SRAM, Static Random Access Memory)

静态可读写随机存储器 SRAM,一般由四个或者六个 MOS 管构成一个二进制位。当电源有电时,SRAM 不用刷新,可以保持原有的数据。

5. 只读存储器(ROM, Read Only Memory)

只读存储器 ROM,数据可以读出,但不可以修改,所以称之为只读存储器。通常存储一些固定不变的信息,如:常数、数据、换码表、程序等。它具有断电后数据不丢失的特点。ROM 有固定 ROM、可编程 ROM(即 PROM)和可擦除 ROM(即 EPROM)三种。

PROM 的编程原理是通过大电流将相应位的熔丝熔断,从而将该位改写成 0,熔丝熔断后不能再次改变,所以只改写一次。

EPROM(Erase PROM)是可以擦除和改写的 ROM,它用 MOS 管代替了熔丝,所以可以反复擦除、多次改写。擦除是用紫外线擦除器来完成的,很不方便。有一种用低电压信号即可擦除的 EPROM 称为电可擦除 EPROM,简写为 E^2 PROM 或 EEPROM。

6. 闪速存储器(Flash Memory)

闪速存储器简称闪存,是一种新型快速的 E^2 PROM。由于工艺和结构上的改进,闪存比普通的 E^2 PROM 的擦除速度更快,集成度更高。如:飞思卡尔目前生产的微控制器芯片中大多采用第三代闪存技术,可以达到 10 万次重复读/写能力,页整体擦除和写入只需要几十毫秒。

闪存相对于传统的 E^2 PROM 来说,其最大的优点是系统内编程,也就是说不需要另外的器件来修改内容。闪存的机构随着时代的发展而有些变动,尽管现代的快速闪存是系统内可编程的,但仍然没有 RAM 使用起来方便。擦写操作必须通过特定的程序算法来实现。

7. 模拟量

模拟量是指时间连续、数值也连续的物理量，如温度、压力、流量、速度、声音等。在工程技术上，为了便于分析，常用传感器、变换器将模拟量转换为电流、电压或电阻等电学量。

8. 开关量

开关量是指一种二值信号，用两个电平（高电平和低电平）分别来表示两个逻辑值（逻辑 1 和逻辑 0）。

1.3.2 与通信相关的术语

1. 并行通信

并行通信是指数据的各位同时在多根并行数据线上进行传输的通信方式，数据的各位同时由源到达目的地。适合近距离、高速通信。常用有 4 位、8 位、16 位、32 位等同时传输。

2. 串行通信

串行通信是指数据在单线（电平高低表征信号）或双线（差分信号）上，按时间先后一位一位地传送，其优点是节省传输线，但相对于并行通信来说，速度较慢。在嵌入式系统中，串行通信一词一般特指用串行通信接口 SCI（Serial Communication Interface）与 RS232 芯片连接的通信方式。SPI、I²C、USB 等也是串行通信，但它们使用专用名词。

3. 串行外设接口 SPI

串行外设接口（SPI，Serial Peripheral Interface）：也是一种串行通信方式，主要用于 MCU 扩展外围芯片使用。这些芯片可以是具有 SPI 接口的 A/D 转换、时钟芯片等。

4. 集成电路互连总线 I²C

I²C（Inter Integrated Circuit），是一种由 PHILIPS 公司开发的两线式串行总线，有的书籍也记为 IIC 或 I2C，主要用于用户电路板内 MCU 与其外围电路的连接。

5. 通用串行总线 USB

通用串行总线（USB，Universal Serial Bus），是 MCU 与外界进行数据通信的一种新的方式，其速度快，抗干扰能力强，在嵌入式系统中得到了广泛的应用。

6. 控制局域网 CAN

控制器局域网（CAN，Control Area Network），是一种全数字、全开放的现场总线控制网络，目前在汽车电子中应用最广。

7. 背景调试模式 BDM

背景调试模式（BDM，Background Debug Mode），是 Freescale 半导体公司提出的一

种调试接口，主要用于嵌入式 MCU 的程序下载与程序调试。

8. 边界扫描测试协议JTAG

边界扫描测试协议(JTAG, Joint Test Action Group),是由国际联合测试行动组开发,对芯片进行测试的一种方式,可将其用于对 MCU 的程序进行载入与调试。JTAG 能获取芯片寄存器等内容,或者测试遵守 IEEE 规范的器件之间引脚连接情况。

关于通信相关的术语还有嵌入式以太网、无线传感器网络、Zigbee、射频通信等,本章不再进一步介绍。

1.3.3 与功能模块及软件相关的术语

1. 通用输入/输出GPIO

通用输入/输出(GPIO, General Purpose I/O),即基本的输入/输出,有时也称并行 I/O。作为通用输入引脚时,MCU 内部程序可以读取该引脚,知道该引脚是“1”(高电平)或“0”(低电平),即开关量输入。作为通用输出引脚时,MCU 内部程序向该引脚输出“1”(高电平)或“0”(低电平),即开关量输出。

2. A/D与D/A

A/D 转换模块的功能是将电压信号(模拟量)转换为对应的数字量。实际应用中,这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过 A/D 转换,MCU 就可以处理这些物理量。而与之相反,D/A 转换则是将数字量转换为电压信号(模拟量)。

3. 脉冲宽度调制器PWM

脉冲宽度调制器(PWM, Pulse Width Modulator),是一个 D/A 转换器,可以产生一个高电平和低电平之间重复交替的输出信号,这个信号就是 PWM 信号。

4. 看门狗

看门狗(Watch Dog),是一个为了防止程序跑飞而设计的一种自动定时器。当程序跑飞时,由于无法正常执行清看门狗定时器,看门狗定时器会自动溢出,使系统程序复位。

5. 液晶显示LCD

液晶显示(LCD, Liquid Crystal Display),是电子信息产品的一种显示器件,可分为字段型、点阵字符型、点阵图形型三类。

6. 发光二极管LED

发光二极管(LED, Light Emitting Diode),是一种将电流顺向通到半导体 PN 结处而发光的器件。常用于家电指示灯、汽车灯和交通警示灯。

7. 键盘

键盘是嵌入式系统中最常见的输入设备。识别键盘是否有效被按下的方法有查询法、

定时扫描法与中断法等。

8. 实时操作系统RTOS

实时操作系统（Real Time Operating System, RTOS），是指一种运行于嵌入式系统上的操作环境，可以提供建立多任务的能力。RTOS 为每个任务建立一个可执行的环境，可以很方便地在任务之间传递消息，在一个中断处理程序和任务之间传递事件，区分任务执行的优先级，并协调多个任务对同一个 I/O 设备的调用。通常一个规模大、结构复杂的嵌入式系统可以分解为一系列较小、较简单的并行任务来实现，各个任务之间互不干扰，使用 RTOS 排除并行任务中的人为因素，降低复杂度，增强模块化，使工程由更简易和标准化的模块组成，处理起来更加轻松、快捷。

与软件相关术语很多，本章不再进一步介绍。

1.4 嵌入式系统常用的标准C语言基本语法概要

C 语言是在 70 年代初问世的。1978 年美国电话电报公司（AT&T）贝尔实验室正式发表了 C 语言。由 B.W.Kernighan 和 D.M.Ritchie 合著的《THE C PROGRAMMING LANGUAGE》一书，被简称为《K&R》，也有人称之为 K&R 标准。但是，在《K&R》中并没有定义一个完整的标准 C 语言，后来由美国国家标准学会在此基础上制定了一个 C 语言标准，于 1983 年发表，通常称之为 ANSI C 或标准 C。

本节简要介绍 C 语言的基本知识，特别是和嵌入式系统编程密切相关的基本知识，未学过标准 C 语言的读者可以通过本节了解 C 语言，以后通过实例逐步积累相关编程知识。对 C 语言很熟悉的读者，可以跳过本节。

1. 数据类型

C 语言的数据类型有基本类型和构造类型两大类。基本类型如表 1-1 所示。

表 1-1 C 语言基本数据类型

数据类型		简明含义	位数	字节数	值域
字节型	signed char	有符号字节型	8	1	-128~+127
	unsigned char	无符号字节型	8	1	0~255
整型	signed short	有符号短整型	16	2	-32768~+32767
	unsigned short	无符号短整型	16	2	0~65535
	signed int	有符号短整型	16	2	-32768~+32767
	unsigned int	无符号短整型	16	2	0~65535
	signed long	有符号长整型	32	4	-2147483648~+2147483647
	unsigned long	无符号长整型	32	4	0~4294967295
实型	float	浮点型	32	4	约±3.4×(10 ⁻³⁸ ~10 ⁺³⁸)
	double	双精度型	64	8	约±1.7×(10 ⁻³⁰⁸ ~10 ⁺³⁰⁸)

注：常用的嵌入式 C 语言中的 double 类型长度为 4 字节。

构造类型有数组、结构、联合、枚举、指针和空类型。结构和联合是基本数据类型的组合。枚举是一个被命名为整型常量的集合。空类型字节长度为 0，主要有两个用途：一

是明确地表示一个函数不返回任何值；二是产生一个同一类型指针（可根据需要动态地分配给其内存）。

2. 运算符

C 语言的运算符分为算术、逻辑、关系和位运算及一些特殊的操作符。表 1-2 列出了 C 语言的运算符及使用方法举例。

表 1-2 C 语言的运算符

运算类型	运算符	简明含义	举 例
算术运算	+ - * /	加、减、乘、除	N=1, N=N+5 等同于 N+=5, N=6
	^	幂	A=2, B=A^3, B=8
	%	取模运算	N=5, Y=N%3, Y=2
逻辑运算		逻辑或	A=TRUE, B=FALSE, C=A B, C=TRUE
	&&	逻辑与	A=TRUE, B=FALSE, C=A&&B, C=FALSE
	!	逻辑非	A=TRUE, B=!A, B=FALSE
关系运算	>	大于	A=1, B=2, C=A>B, C=FALSE
	<	小于	A=1, B=2, C=A<B, C=TRUE
	>=	大于等于	A=2, B=2, C=A>=B, C=TRUE
	<=	小于等于	A=2, B=2, C=A<=B, C=TRUE
	==	等于	A=1, B=2, C=(A==B), C=FALSE
	!=	不等于	A=1, B=2, C=(A!=B), C=TRUE
位运算	~	按位取反	A=0b00001111, B=~A, B=0b11110000
	<<	左移	A=0b00001111, A<<2=0b00111100
	>>	右移	A=0b11110000, A>>2=0b00111100
	&	按位与	A=0b1010, B=0b1000, A&B=0b1000
	^	按位异或	A=0b1010, B=0b1000, A^B=0b0010
		按位或	A=0b1010, B=0b1000, A B=0b1010
增量和减量运算	++	增量运算符	A=3, A++, A=4
	--	减量运算符	A=3, A--, A=2
复合赋值运算	+=	加法赋值	A=1, A+=2, A=3
	-=	减法赋值	A=4, A-=4, A=0
	>>=	右移位赋值	A=0b11110000, A>>=2, A=0b00111100
	<<=	左移赋值	A=0b00001111, A<<=2, A=0b00111100
	=	乘法赋值	A=2, A=3, A=6
	=	按位或赋值	A=0b1010, A =0b1000, A=0b1010
	&=	按位与赋值	A=0b1010, A&=0b1000, A=0b1000
	^=	按位异或赋值	A=0b1010, A^=0b1000, A=0b0010
	%=	取模赋值	A=5, A%=2, A=1
	/=	除法赋值	A=4, A/=2, A=2
指针和地址运算	*	取内容	
	&	取地址	
输出格式转换	0x	无符号十六进制数	0xa=0d10
	0o	无符号八进制数	0o10=0d8
	0b	无符号二进制数	0b10=0d2
	0d	带符号十进制数	0d10000001=-127
	0u	无符号十进制数	0u10000001=129

注：增量运算符和减量运算符存在运算和取数先后次序，例如，`A++`是先取变量 `A` 的值再对 `A` 加 1，而 `++A` 是先对变量 `A` 加 1 再取 `A` 的值。

3. 流程控制

在程序设计中主要有三种基本控制结构：顺序结构、选择结构和循环结构。

1) 顺序结构

顺序结构就是从前向后依次执行语句。从整体上看，所有程序的基本结构都是顺序结构，中间的某个过程可以是选择结构或循环结构。

2) 选择结构

在大多数程序中都会包含选择结构。其作用是，根据所指定的条件是否满足，决定执行哪些语句。在 C 语言中主要有 `if` 和 `switch` 两种选择结构。

(1) if 结构

```
if(表达式) 语句项;    或    if(表达式) 语句项;
                               else 语句项;
```

如果表达式取值真（除 0 以外的任何值），则执行 `if` 的语句项；否则，如果 `else` 存在的话，就执行 `else` 的语句项。每次只会执行 `if` 或 `else` 中的某一个分支。语句项可以是单独的一条语句、也可以是多条语句组成的语句块（要用一对大括号“{}”括起来）。

`if` 语句可以嵌套，有多个 `if` 语句时 `else` 与最近的一个配对。对于多分支语句，可以使用 `if ... else if ... else if ... else ...` 的多重判断结构，也可以使用下面讲到的 `switch` 开关语句。

(2) switch 结构

`switch` 是 C 语言内部多分支选择语句，它根据某些整型和字符常量对一个表达式进行连续测试，当一常量值与其匹配时，它就执行与该变量有关的一个或多个语句。`switch` 语句的一般形式如下：

格式：

```
switch(表达式)
{
    case 常数 1:
        语句项 1
        break;
    case 常数 2:
        语句项 2
        break;
    .....
    default:
        语句项
}
```

根据 `case` 语句中所给出的常量值，按顺序对表达式的值进行测试，当常量与表达式

值相等时，就执行这个常量所在的 `case` 后的语句块，直到碰到 `break` 语句，或者 `switch` 的末尾为止。若没有一个常量与表达式值相符，则执行 `default` 后的语句块。`Default` 是可选的，如果它不存在，并且所有的常量与表达式值都不相符，那就不做任何处理。

`switch` 语句与 `if` 语句的不同之处在于 `switch` 只能对等式进行测试，而 `if` 可以计算关系表达式或逻辑表达式。

`break` 语句在 `switch` 语句中是可选的，如果不用 `break`，就继续寻找下一个条件满足的 `case` 语句执行，一直到碰到 `break` 或 `switch` 的末尾为止，这样的程序效率比较低。

3) 循环结构

C 语言中的循环结构常用 `for` 循环，`while` 循环与 `do...while` 循环。

(1) `for` 循环

格式为：

```
for (初始化表达式; 条件表达式; 修正表达式)
{循环体}
```

执行过程为：先求解初始化表达式；再判断条件表达式，若为假（0），则结束循环，转到循环下面的语句；如果其值为真（非 0），则执行“循环体”中语句。然后求解修正表达式；再转到判断条件表达式处根据情况决定是否继续执行“循环体”。

(2) `while` 循环

格式为：

```
while(条件表达式)
{循环体}
```

当表达式的值为真（非 0）时执行循环体。其特点是：先判断后执行。

(3) `do...while` 循环

格式为：

```
do
{循环体}
while(条件表达式);
```

其特点是：先执行后判断。即当流程到达 `do` 后，立即执行循环体一次，然后才对条件表达式进行计算、判断。若条件表达式的值为真（非 0），则重复执行一次循环体。

4) `break`和`continue`语句在循环的应用

在循环中常常使用 `break` 语句和 `continue` 语句，这两个语句都会改变循环的执行情况。`break` 语句用来从循环体中强行跳出循环，终止整个循环的执行；`continue` 语句使其后语句不再被执行，进行新的一次循环（可以形象地理解为返回循环开始处执行）。

4. 函数

所谓函数，即子程序，也就是“语句的集合”，就是说把经常使用的语句群定义成函数，供其他程序调用，函数的编写与使用要遵循软件工程的基本规范。

使用函数要注意：函数定义时要同时声明其类型；调用函数前要先声明该函数；传给

函数的参数值，其类型要与函数原定义一致；接收函数返回值的变量，其类型也要与函数类型一致等。

函数的返回值：**return** 表达式；

return 语句用来立即结束函数，并返回一确定值给调用程序。如果函数的类型和 **return** 语句中表达式的值不一致，则以函数类型为准。对数值型数据，可以自动进行类型转换。即函数类型决定返回值的类型。

5. 指针

指针是C语言中广泛使用的一种数据类型，运用指针是C语言最主要的风格之一。在嵌入式编程中，指针尤为重要。利用指针变量可以表示各种数据结构，很方便地使用**数组**和**字符串**，并能像汇编语言一样处理内存地址，从而编出精练而高效的程序。

指针是一种特殊的数据类型，在其它语言中一般没有。**指针是指向变量的地址，实质上指针就是存储单元的地址**。根据所指的变量类型不同，可以是整型指针（**int ***）、浮点型指针（**float ***）、字符型指针（**char ***）、结构指针（**struct ***）和联合指针（**union ***）。

1) 指针变量的定义

其一般形式为：类型说明符 * 变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示本指针变量所指向的变量的数据类型。

例如：

```
int *p1;    //表示 p1 是指向整型数的指针变量，p1 的值是整型变量的地址
```

2) 指针变量的赋值

指针变量同普通变量一样，使用之前不仅要进行声明，而且必须赋予具体的值。未经赋值的指针变量不能使用，否则将造成系统混乱，甚至死机。指针变量的赋值只能赋予地址。

例如：

```
int a;          //a 为整型数据变量
int *p1;        //声明 p1 是整型指针变量
p1 = &a;        //将 a 的地址作为 p1 初值
```

3) 指针的运算

(1) 取地址运算符&

取地址运算符**&**是单目运算符，其结合性为自右至左，其功能是取变量的地址。

(2) 取内容运算符*

取内容运算符*****是单目运算符，其结合性为自右至左，用来表示指针变量所指的变量。在*运算符之后跟的变量必须是指针变量。

例：

```
int a,b;        //a,b 为整型数据变量
```

```
int *p1;           //声明 p1 是整型指针变量
p1 = &a;           //将 a 的地址作为 p1 初值
a = 80;
b = *p1;           //运行结果:b=80, 即为 a 的值
```

注意：取内容运算符“*”和指针变量声明中的“*”虽然符号相同，但含义不同。在指针变量声明中，“*”是类型说明符，表示其后的变量是指针类型。而表达式中出现的“*”则是一个运算符用以表示指针变量所指的变量。

(3) 指针的加减算术运算

对于指向数组的指针变量，可以加/减一个整数 n （由于指针变量实质是地址，给地址加/减一个非整数就错了）。设 pa 是数组 a 的指针变量，则 $pa+n$ ， $pa-n$ ， $pa++$ ， $++pa$ ， $pa--$ ， $--pa$ 运算都是合法的。指针变量加/减一个整数 n 的意义是把指针指向的当前位置（指向某数组元素）向前或向后移动 n 个位置。

注意：数组指针变量前/后移动一个位置和地址加/减 1 在概念上是不同的。因为数组可以有不同的类型，各种类型的数组元素所占的字节长度是不同的。如指针变量加 1，即向后移动 1 个位置，表示指针变量指向下一个数据元素的首地址。而不是在原地址基础上加 1。

例如：

```
int a[5], *pa;      //声明 a 为整型数组（下标为 0~5），pa 为整型指针
pa = a;             //pa 指向数组 a，也是指向 a[0]。
pa = pa + 2;        //pa 指向 a[2]，即 pa 的值为 &a[2]
```

注意：指针变量的加/减运算只能对数组指针变量进行，对指向其它类型变量的指针变量作加/减运算是毫无意义的。

4) void 指针类型

顾名思义， $\text{void} *$ 为“无类型指针”，即用来定义指针变量，不指定它是指向哪种类型数据，但可以把它强制转化成任何类型的指针。

众所周知，如果指针 $p1$ 和 $p2$ 的类型相同，那么可以直接在 $p1$ 和 $p2$ 间互相赋值；如果 $p1$ 和 $p2$ 指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。

例如：

```
float *p1;          //声明 p1 为浮点型指针
int *p2;            //声明 p2 为整型指针
p1 = (float *)p2;    //强制转换整型指针 p2 为浮点型指针值给 p1 赋值
```

而 $\text{void} *$ 则不同，任何类型的指针都可以直接赋值给它，无需进行强制类型转换：

```
void *p1;           //声明 p1 无类型指针
int *p2;            //声明 p2 为整型指针
p1 = p2;            //用整型指针 p2 的值给 p1 直接赋值
```

但这并不意味着， $\text{void} *$ 也可以无需强制类型转换地赋给其它类型的指针，也就是说

p2=p1 这条语句编译就会出错，而必须将 p1 强制类型转换成“void*”类型。因为“无类型”可以包容“有类型”，而“有类型”则不能包容“无类型”。

6. 结构体

结构体是由基本数据类型构成的，并用一个标识符来命名的各种变量的组合。结构体中可以使用不同的数据类型。

1) 结构体的说明和结构体变量的定义

例 1：定义一个名为 student 的结构体变量类型

```
struct student      //定义一个名为student的结构体变量类型
{
    char name[8];    //成员变量“name”为字符型数组
    char class[10];  //成员变量“class”为字符型数组
    int age;         //成员变量“age”为整型
};
```

这样，若声明 s1 为一个“student”类型的结构体变量，则使用如下语句：

```
struct student s1;  //声明s1为“student”类型的结构体变量
```

例 2：定义一个名为 student 的结构体变量类型，同时声明 s1 为一个“student”类型的结构体变量

```
struct student      //定义一个名为student的结构体变量类型
{
    char name[8];    //成员变量“name”为字符型数组
    char class[10];  //成员变量“class”为字符型数组
    int age;         //成员变量“age”为整型
}s1;                //声明s1为“student”类型的结构体变量
```

2) 结构体变量的使用

结构体是一个新的数据类型，因此结构体变量也可以象其它类型的变量一样赋值运算，不同的是结构体变量以成员作为基本变量。

结构体成员的表示方式为：

结构体变量.成员名

如果将“结构体变量.成员名”看成一个整体，则这个整体的数据类型与结构体中该成员的数据类型相同，这样就像前面所讲的变量那样使用。例如：

```
s1.age=18;          //将数据 18 赋给 s1.age（理解为学生 s1 的年龄为 18）
```

3) 结构体指针

结构体指针是指向结构体的指针。它由一个加在结构体变量名前的“*”操作符来声明。例如用上面已说明的结构体声明一个结构体指针如下：

```
struct student *Pstudent; //声明 Pstudent 为一个“student”类型指针
```

使用结构体指针对结构体成员的访问，与结构体变量对结构体成员的访问在表达方式上有所不同。结构体指针对结构体成员的访问表示为：

结构体指针名->结构体成员

其中">"是两个符号"-"和">"的组合，好象一个箭头指向结构体成员。例如要给上面定义的结构体中 name 和 age 赋值，可以用下面语句：

```
strcpy(Pstudent->name,"LiuYuZhang");
```

```
Pstudent->age=18;
```

实际上，Pstudent->name 就是 (*Pstudent).name 的缩写形式。

需要指出的是结构体指针是指向结构体的一个指针，即结构体中第一个成员的首地址，因此在使用之前应该对结构体指针初始化，即分配整个结构体长度的字节空间。这可用下面函数完成，例如：

```
Pstudent=(struct student*)malloc(sizeof (struct student));
```

sizeof (struct student) 自动求取 student 结构体的字节长度，malloc () 函数定义了一个大小为结构体长度的内存区域，然后将其地址作为结构体指针返回。

7. 位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进制位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域 (bit field)”，也有人翻译为“位段”。所谓“位域”，实际上是字节中一些位的组合，可以认为它是“位信息组”。位域将一个字节中的二进制划分为几个不同的区域，并给每个区域起个域名，允许在程序中按域名进行操作。

1) 位域的定义

例：定义一个名为 bs 的位域变量类型，同时声明 b1 为 bs 类型的变量。

```
struct bs
{
    unsigned int  a:2;  //第 0~1 位
    unsigned int  b:6;  //第 2~7 位
}b1;
```

注意：定义位域必须使用“unsigned int”。上例声明 b1 为 bs 变量，共占 8 位 (1 字节)。其中位域 a 占 2 位 (a 的范围为 0~3)，位域 b 占 6 位 (b 的范围为 0~63)。对于位域的定义尚有以下几点说明：

(1) 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
    unsigned int a:4
    unsigned int :0  //空域(本字节剩余位不用)
    unsigned int b:4  //从下一单元开始存放
```

```
unsigned int c:4
```

```
};
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

(2) 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进制。

(3) 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
```

```
{
```

```
    unsigned int a:1    //第 0 位
```

```
    unsigned int :2    //无域名，2 位不能使用
```

```
    unsigned int b:3    //第 3~5 位
```

```
    unsigned int c:2    //第 6~7 位
```

```
    }b1;
```

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进制分配的。

2) 位域的使用

位域的使用和结构成员的使用相同，其一般形式为：

位域变量名 · 位域名

例如在上面定义的位域 b1 可以这样调用：

```
b1.a=1;    //将 b1 的第 0 位置 1（注意一个字节从最低位开始）
```

```
b1.b=7;    //将 b1 的第 3~5 位置 111
```

通过位域定义位变量，是实现单个位操作的重要途径和方法，采用位域定义位变量，产生的代码紧凑、高效。

8. 编译预处理

C 语言提供编译预处理的功能，“编译预处理”是 C 编译系统的一个重要组成部分。C 语言允许在程序中使用几种特殊的命令（它们不是一般的 C 语句）。在 C 编译系统对程序进行通常的编译（包括语法分析、代码生成、优化等）之前，先对程序中的这些特殊的命令进行“预处理”，然后将预处理的结果和源程序一起再进行常规的编译处理，以得到目标代码。C 提供的预处理功能主要有宏定义、条件编译和文件包含。

1) 宏定义

```
#define 宏名 表达式
```

表达式可以是数字、字符，也可以是若干条语句。在编译时，所有引用该宏的地方，都将自动被替换成宏所代表的表达式。例如：

```
#define PI 3.1415926    //以后程序中用到数字 3.1415926 就写 PI
```

```
#define S(r) PI*r*r //以后程序中用到 PI*r*r 就写 S(r)
```

2) 条件编译

```
#if 表达式
#else 表达式
#endif
```

如果表达式成立，则编译`#if`下的程序，否则编译`#else`下的程序，`#endif`为条件编译的结束标志。

```
#ifdef 宏名 //如果宏名称被定义过，则编译以下程序
#ifndef 宏名 //如果宏名称未被定义过，则编译以下程序
```

条件编译通常用来调试、保留程序（但不编译），或者在需要对两种状况做不同处理时使用。

3) “文件包含”处理

所谓“文件包含”是指一个源文件将另一个源文件的全部内容包含进来，其一般形式为：

```
#include “文件名”
```

9. 用typedef定义类型

除了可以直接使用 C 提供的标准类型名（如 `int`、`char`、`float`、`double`、`long` 等）和自己定义的结构体、指针、枚举等类型外，还可以用 `typedef` 定义新的类型名来代替已有的类型名。例如：

```
typedef unsigned char INT8U;
```

指定用 `INT8U` 代表 `unsigned char` 类型。这样下面的两个语句是等价的：

```
unsigned char i;          等价于          INT8U i;
```

用法说明：

- (1) 用 `typedef` 可以定义各种类型名，但不能用来定义变量。
- (2) 用 `typedef` 只是对已经存在的类型增加一个类型别名，而没有创造新的类型。
- (3) `typedef` 与 `#define` 有相似之处，如：

```
typedef unsigned int INT16U;
#define INT16U unsigned int;
```

这两句的作用都是用 `INT16U` 代表 `unsigned int`。但事实上它们二者不同，`#define` 是在预编译时处理，它只能做简单的字符串替代，而 `typedef` 是在编译时处理。

(4) 当不同源文件中用到各种类型数据（尤其是像数组、指针、结构体、共用体等较复杂数据类型）时，常用 `typedef` 定义一些数据类型，并把它们单独存放在一个文件中，而后在需要用到它们的文件中用 `#include` 命令把它们包含进来。

- (5) 使用 `typedef` 有利于程序的通用与移植。

1.5 本书学习与实践资料索引

为了配合 ColdFire 的应用与设计,苏州大学飞思卡尔嵌入式系统实验室设计了相关硬件,本书配备了网上光盘,名称为“《基于 32 位 ColdFire 构建嵌入式系统》光盘.rar”,有关下载地址在前言中已经给出。本书随后所说的光盘,均特指该文件。图 1-2 给出了网上光盘简明目录结构。

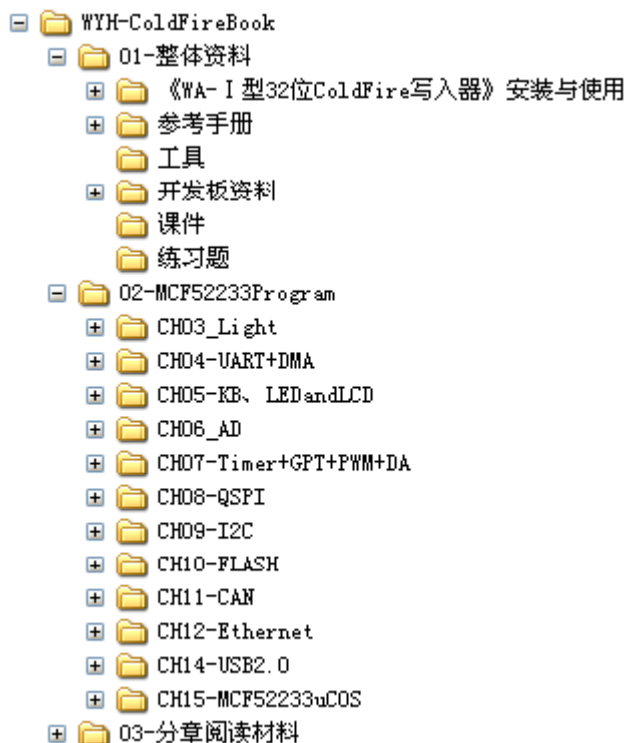


图1-2 网上光盘简明目录结构

1. 光盘主要内容

1) 芯片参考手册

(1) MCF52235RM (内含 MCF52230、MCF52231、MCF52232、MCF52233、MCF52234、MCF52235、MCF52236)

(2) MCF52223RM (MCF52221、MCF52223)

2) 《WA- I 型32位ColdFire写入器》安装与使用

(1) 《WA- I 型 32 位 ColdFire 写入器》调试动态连接库

(2) 《WA- I 型 32 位 ColdFire 写入器》驱动

(3) 《WA- I 型 32 位 ColdFire 写入器》使用说明

(4) 解除密码 (提供加密芯片解除密码方法)

3) 工具软件

开发环境、程序写入与读出软件、串口调试工具、USB 工具、以太网工具

4) 分章资料

各章源程序、课件、参考文章等

2. 硬件评估系统

(1) WA- I 型 32 位 ColdFire 写入器

(2) MCF52233 核心板 (型号 WA-MCF52233EVB): 使用 80 引脚的 MCF52233CAF60 芯片, 带以太网接口、RS232 串口、一个复位按钮、两个按键、引出所有 I/O 口。外接 5V 直流电源, 板内转为 3.3V 给芯片供电。带有与苏州大学 Freescale 嵌入式系统实验室的 MT-扩展板- I 型的对接口。

(3) MCF52223 核心板 (型号 WA-MCF52223EVB)

(4) MCF52235 核心板 (型号 WA-MCF52225EVB):

(5) 通用扩展板: WA-扩展板- I 型 (30cm*21cm), 可选配 46cm*29cm*11cm 仪器箱。

第2章 ColdFire概述与MCF52233硬件最小系统

本章简要概述 ColdFire 系列微处理器, 给出 MCF5223x 系列微控制器存储器映像与编程结构、寻址方式、异常向量表; 重点指出学习一个新 MCU 芯片比较快速的学习过程; 给出 MCF52233 的引脚功能与硬件最小系统电路。本章的重点是: 存储空间的地址分配、中断结构、硬件最小系统电路。

2.1 ColdFire系列微处理器概述

在 1996 年, Motorola 半导体部(现 Freescale 半导体公司)将具有划时代意义的 ColdFire 微处理器加入该公司的 32 位机大家族。四年以后, 全世界已经使用了超过一亿片的以 68K/ColdFire 微处理器为核心的微控制器。ColdFire 是著名的 68K 体系结构的延续, 沿袭了 68K 系列的精华, 改进了模块与工具。经过多年的发展, ColdFire 已经发展出大量的微控制器系列产品, 集成了各种外设和不同大小的存储器, 成为许多低成本、高可靠、高性能的嵌入式应用的理想解决方案。Freescale 公司提供了不同的功能、价格、集成度和调试能力的 32 位单芯片嵌入式应用的解决方案, ColdFire 系列微控制器建议零售价从 5 美元到 25 美元不等, 处理器的性能水平从 10 到 400MIPS^①不等, 使得用户可以各取所需。也就是说, 用户无论需要怎样的系统功能和集成度, 总能从 ColdFire 的庞大产品系列中选到一款合适的芯片进行应用开发。这正是嵌入式系统产品的设计者所期望的。

了解 ColdFire 系列微处理器的结构特点, 有助于产品设计时的选型。下面简要介绍 ColdFire 系列微处理器体系结构的主要特点。

ColdFire 系列具有变长精简指令集计算机 (Reduced Instruction Set Computer, RISC) 架构, 集成了高代码密度的高性能内核和大量外围接口。ColdFire 系列微处理器最先推出的版本是 V2, 以 ColdFire V2 微处理器为核心制造的微控制器产品种类达百种以上。后来版本发展到 V3、V4, 还在继续发展, 目标是覆盖各种不同应用。后来 Freescale 推出了单片机业界首个 8/32 位兼容架构中缺失的一环 ColdFire V1。

V1 内核是 ColdFire V2 内核的简化版本, 设计用于 32 位入门级应用。它改善了字节 (8 位) 和字 (16 位) 操作数的处理能力, 同时保持了与 ColdFire 架构相同的寻址方式和指令定义。通过将硬件乘法累加器 (Multiply-ACcumulate unit, MAC)^②、增强型乘法累加器 (Enhanced Multiply-ACcumulate unit, EMAC) 和除法 (DIV) 变为可选功能, 设计者能够将那些不需要增强型处理性能的应用的成本降至最低水平。

V2 内核基于存储器可配置分层架构, 是 100%可综合的, 并且专门设计用来重新使用

^① MIPS (millions of instructions per second, 百万条指令每秒)。许多工程师用该术语表示处理器的处理能力。它是一个技术上并不全面但已被广泛采用的基准度量。

^②乘法累加器是 Multiply-Accumulate Unit 的缩写, 有的文献使用 MAU, 但 Freescale 的 ColdFire 编程参考手册使用了 MAC 这一缩写, 本书沿用。

和简化集成定制设计。在 0.13 μm 工艺的基础上, 利用带有 32 位地址和数据总线与集成调试模块的单线程、基于标准单元的设计, 内核可以支持高达 166MHz 的性能。内核采用长度可变的 RISC 架构, 允许的指令长度有 16、32 和 48 位。结果就是能更有效地将代码存储到存储器中, 从而降低了存储器的要求, 并削减了系统总成本。**V2 主要特点:** 变长的 RISC 指令集、2 个独立的流水线 (两级取指令流水线和两级操作数执行流水线)、用 3 个长字的 FIFO 缓冲区将流水线彼此分离、32 位内部总线地址支持 4GB 的地址空间、32 位数据总线、16 个用户可用的 32 位通用寄存器、具有管理员/用户编程模式、地址空间可通过配置寄存器重新定位、对高级语言的优化支持等。

V3 内核包括优化的指令预取流水线, 分支预测能力, 和更高的操作频率。该内核性能比 V2 内核提高了 300%, 而系统操作频率仅增加不到 50%。另外还包括一个高速的乘法累加单元, 专用的硬件除法和增强的调试功能。

V4 内核的微体系结构和 V3 内核相比, 性能提高了 2.8 倍, 拥有更加广阔的应用。改良的微体系结构拥有更高的性能: 大于 200 Dhrystone 2.1 MIPS@150MHz^①。

ColdFire 内核中还可以集成硬件乘法累加器 MAC、增强型乘法累加器 EMAC 和密码加速单元 (Cryptographic Acceleration Unit, CAU), 提高了效率和性能。

乘法累加器 MAC 用于设计完成一系列的 DSP 操作, 当系统支持基于 ColdFire 体系结构的整数乘法指令时, MAC 可以提供嵌入式代码的性能。带有 MAC 的微处理器能以较少的周期快速执行信号处理算法。例如对于较小的数字滤波器, 该算法执行一些方差计算的时间还可以接受。MAC 可以用于以下 3 方面: 有符号和无符号的整数乘法、有符号的定点小数运算及多种寄存器操作数运算。ColdFire 系列支持 2 个 MAC, 由于其成本不同, 故具有不同的功能和性能。

增强型乘法累加器 EMAC 对乘法累加器 MAC 进行了改进, 采用 4 级流水线, 最优地实现了 32 位操作数运算, 具有全流水的 32 $\times$ 32 乘法数组, 以及 4 个 48 位累加器。

密码加速单元 CAU 是一个协处理器, 用来执行一些 ColdFire 协处理器命令。CAU 能加速一些常用的加密算法, 为网络设备 (如家庭网关或路由器) 的安全性保障提供技术基础。其他的 CAU 加密应用有自动售货机、自动收费机、雇员或个人识别系统。本书的范例芯片 MCF5223x 系列中 MCF52235 含有密码加速单元 CAU。

Freescale 以 ColdFire 微处理器不同版本为内核生产的微控制器 (MCU), 命名方法一般如下: ColdFire V1 内核的产品以 MCF51xx 命名, ColdFire V2 内核的产品以 MCF52xx 命名, ColdFire V3 内核的产品以 MCF53xx 命名, ColdFire V4 内核的产品以 MCF54xx 命名, 各自形成 MCU 子系列。这些不同型号的 MCU, 因内部模块数量、I/O 口数目、封装形式等不同形成具体型号的产品。较为详细的资料可参见光盘 “···\第 02 章 阅读材料” 目录中的资料。这些资料都在更新, 最新的资料可以在 Freescale 的网站 (英文网站 <http://www.freescale.com/ColdFire> 或中文网站 <http://www.freescale.com.cn/ColdFire>) 上下载。

^① Dhrystone, 一种常用的处理器基准测试程序, 主要用于市场目的的计算机性能测试。

2.2 MCF5223x系列微控制器存储器映像与编程结构

2006 年, Freescale 开始提供 MCF5223x 样片, 2007 年 MCF5223x 批量上市。MCF5223x 系列 MCU 是首个能在单片内提供快速以太网控制器及以太网物理层和闪存的 32 位 MCU。本书以该芯片为主要蓝本阐述嵌入式开发所必备的硬件设计、软件设计及相关技术。一般来说, 学习一个新的 MCU 芯片, 若用 C 语言进行编程, 比较快速的学习过程是:

- (1) 了解性能及内部主要功能模块与存储空间的地址分配。
- (2) 了解基本的编程结构、编程模式及寻址方式。
- (3) 了解中断结构。
- (4) 了解芯片的引脚的总体布局情况, 掌握硬件最小系统电路。
- (5) 理解第一个工程的结构, 理解工程中各个文件的基本功能。一般来说, 第一个工程为一个简单的小程序, 如利用通用 I/O 模块编程控制几个发光二极管, 主要目的是给出程序框架和工作过程。
- (6) 进行实际环境的编译 (compile)、链接 (link) 生成可以下载到芯片内部 Flash 存储器中的程序 (可以运行的机器码), 基本理解列表文件、机器码文件。
- (7) 一定要有硬件评估环境, 这是学习新 MCU 的必需品。这样就可将程序利用写入调试器下载到目标 MCU 中, 在目标板上, 观察运行情况。随后, 可进一步利用嵌入式软件的打桩调试技术, 即在被测程序代码中插入一些函数或语句, 利用这些函数或语句产生可在硬件板上显示物理现象, 供观察程序运行情况之用。
- (8) 从整个工程组成、各个文件、写入 Flash 存储器的机器码等角度, 透彻理解第一工程的执行过程。

至此, 就算“入门”了。以后逐步学习各个功能模块就方便了。

本章给出上述过程的 (1) — (4) 步, 下一章给出上述过程的 (5) — (8) 步, 通过两章的学习完成“入门”。

2.2.1 MCF5223x性能概述与内部结构简图

MCF5223x 是单芯片 32 位微控制器解决方案。重要特点是: 该系列芯片内部含有以太网模块, 即 ColdFire V2 内核上集成了 10/100M 快速以太网控制器 (Fast Ethernet Controller, FEC) 与以太网物理层 (Ethernet Physical Transceiver, EPHY) 模块。该系列芯片以合理的价格提供卓越的性能 (频率为 60MHz 时最高 56MIPS)。该系列芯片的典型应用为医疗仪表和监视器、远程数据采集、嵌入式以太网、ZigBee™ 控制节点、安全/访问控制面板、照明控制节点、家庭/工业自动化等。图 2-1 给出了 MCF5223x 的功能框图。

MCF5223x 的主要常规模块和特点总结如下:

- (1) 静态 RAM (SRAM): 32KB;
- (2) Flash 存储器: 128KB/256KB, 10 万次擦写周期, 10 年数据保持;
- (3) 支持后台调试模式 (Background Debug Mode, BDM);

- (4) 支持对集成电路提供串行的外部测试访问,符合 JTAG (Joint Test Action Group) 标准;
- (5) 用于常规标准串行通信的通用异步收发器 (Universal Asynchronous Receiver/Transmitter, UART);
- (6) 队列式串行外围设备接口 (Queued Serial Peripheral Interface, QSPI);
- (7) 集成电路互连总线 (Inter-Integrated Circuit, I2C)^①;
- (8) 4 通道具有直接存储器访问 (Direct Memory Access, DMA) 功能的 32 位定时器;

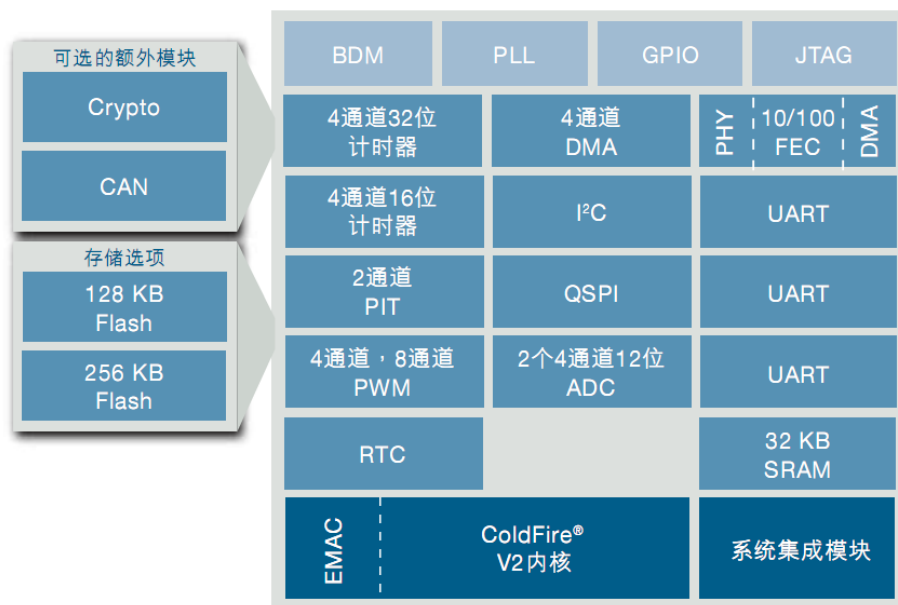


图2-1 MCF5223x方框图

- (9) 通道 16 位或 8 通道 8 位的脉宽调制器 (Pulse-Width Modulator, PWM) 生成器;
- (10) 可编程中断定时器 (Programmable Interrupt Timers, PIT)、4 通道 DMA 控制器;
- (11) 8 通道 12 位模数转换器 (Analog-Digital Converter, ADC);
- (12) 系统集成模块包含锁相环 (Phase-Locked Loop, PLL) 和看门狗 (Computer Operating Properly, COP) 等功能;
- (13) 工作电压为 3.3V。

MCF5223x 系列包括 MCF52230、MCF52231、MCF52232、MCF52233、MCF52234、MCF52235 和 MCF52236。这些器件或速度不同或存储器大小不同或封装不同,而且还有部分器件增加了特殊模块,比如 MCF52235 增加了密码加速协处理器,用户可以从中选

^① 关于 Inter-Integrated Circuit 的缩写,不同文献可能使用不同写法,有 I²C、IIC 及 I2C,本书统一使用 I2C,这样也方便程序注释。

择最适合其应用需求的器件。表 2-1 给出了 MCF5223x 系列 MCU 性能的简要描述，供选型参考之用。

表 2-1 MCF5223x 系列 MCU 性能简表

编号	Flash/SRAM	内部主要功能模块	封装	速度
MCF52230	128/32KB	UART、I2C、QSPI、A/D、FEC EPHY、DMA、16/32/PWM 定时器	80/112 LQFP	60MHz
MCF52231	128/32KB	与 MCF52230 相比增加 FlexCAN 模块	80/112 LQFP	60MHz
MCF52232	128/32KB	与 MCF52230 相同	80 LQFP	50MHz
MCF52233	256/32KB	与 MCF52230 相同	80/112 LQFP	60MHz
MCF52234	256/32KB	与 MCF52230 相比增加 FlexCAN 模块	112 LQFP 121MAPBGA	60MHz
MCF52235	256/32KB	与 MCF52230 相比增加 FlexCAN 模块及密码加速协处理器	112 LQFP 121MAPBGA	60MHz
MCF52236	256/32KB	与 MCF52230 相同	80 LQFP	50MHz

2.2.2 MCF5223x 存储器映像

ColdFire 可寻址 4G 的地址空间，地址范围从 0x0000_0000~0xFFFF_FFFF^①。以 MCF52233 为例，该芯片具有 256KB 的 Flash 和 32KB 的 SRAM。在 4G 的地址空间中，需要安排 Flash、SRAM 及必须通过地址进行操作的 MCU 内部外设系统。MCF52233 使用 Flash 基址寄存器（Flash Base Address Register，FLASHBAR）定位 Flash 存储器的起始地址，使用 RAM 基址寄存器（RAM base address register，RAMBAR）定位 SRAM 存储器的起始地址，使用内部外设系统基址寄存器（Internal Peripheral System Base Address Register，IPSBAR）定位 MCU 内部外设系统占用的 1GB 的存储空间的起始地址。

表 2-2 MCF52233 的存储器空间地址映像

地址范围	空间大小	实际的物理对象
0x8000_0000~0xFFFF_FFFF	2GB	未使用
0x4000_0000~0x7FFF_FFFF	1GB	内部外设系统
0x2000_8000~0x3FFF_FFFF	511.96875MB	未使用
0x2000_0000~0x2000_7FFF	32KB	RAM
0x0004_0000~0x1FFF_FFFF	511.75MB	未使用
0x0000_0000~0x0003_FFFF	256KB	Flash 存储器

一般情况下，设置 Flash 基址寄存器 FLASHBAR=0x0000_0000。这样，256KB 的 Flash 存储器的地址范围是：0x0000_0000~0x0003_FFFF。设置 RAM 基址寄存器 RAMBAR=0x2000_0000，这样，SRAM 的地址范围是：0x2000_0000~0x2000_7FFF。设置内部外设系统基址寄存器 IPSBAR=0x4000_0000，MCU 内部外设系统占用 1GB 的地址空间，这样，MCU 内部外设系统使用的地址范围是：0x4000_0000~0x7FFF_FFFF。表 2-2 给出了 MCF52233 的存储器空间地址映像。设置时，要注意各实际存储体的地址不要重叠，防止读写混乱。若 SRAM 与内部外设系统地址重叠，则 IPSBAR 优先级高于 RAMBAR。

^① 0x00000000 写成 0x0000_0000 仅是为了便于阅读。

ColdFire处理器可以一个周期内完成对SRAM的访问，SRAM主要用于程序变量、堆栈使用。DMA可以双向访问SRAM。RAM通常在掉电后数据会丢失，重新上电后里面是一些没用的数据。若要使RAM的数据不丢失，需要后备电源。ColdFire Flash模块（ColdFire Flash Module, CFM）是非易失存储器（Non-Volatile Memory，NVM）模块。所谓“非易失”，就是说即使掉电，里面的内容也不会丢失。Flash存储器主要用于存储程序、常数以及需要在电源关闭后还必须保存的数据。对Flash存储器的擦除与写入需要通过一定的编程方法才能完成，不像操作RAM那样容易。将在“第10章 Flash存储器在线编程”讨论对Flash存储器的操作。

2.2.3 编程模式与寻址方式

ColdFire处理器具有两种执行指令模式：用户模式和管理员模式。用户模式提供大多数应用程序的执行环境，管理员模式为操作系统和其他系统软件提供一些附加指令和特权指令，有些ColdFire处理器内部寄存器只能在管理员模式下被操作，用户模式下无法操作它们。本书的样例程序在执行时一直在管理员模式下进行操作。

1. 用户模式与管理模式均可操作的寄存器



图2-2 用户模式的编程模型

用户模式与管理模式均可操作的寄存器有16个32位通用寄存器、1个32位程序计数器（Program Counter, PC）和一个8位条件码寄存器（Condition Code Register, CCR）。16个32位通用寄存器分别是8个数据寄存器（D7~D0）和8个地址寄存器（A7~A0）。但需特别指出的是，A7被作为堆栈指针（Stack Pointer, SP），条件码寄存器 CCR 是状态寄存器（Status Register, SR）的低8位。有关寄存器的简明结构见图2-2所示。

1) 数据寄存器（D7~D0）

这些寄存器用于位、字节（8位）、字（16位）以及长字（32位）的操作。它们还可以被用作变址寄存器。字节操作数使用低8位，字操作数使用低16位，而长字操作数占用

寄存器的全部32位。

2) 地址寄存器 (A6~A0) 及堆栈指针SP

A6~A0被用作变址寄存器或基址寄存器。该基址寄存器可以被用来进行字或长字操作。A7被用作的堆栈指针SP。当地址寄存器以16位字访问时，它自动扩展为带符号的32位长字，这是因为这些寄存器用作地址寄存器的指针。A7虽然是用户堆栈指针，但同样可以用作地址寄存器，不过最好别这样使用。

当进行子程序调用时，堆栈可用来存储系统状态，也可将临时数据存储在用户堆栈空间中。堆栈区可以指向 MCU 寄存器映射的 RAM 地址空间的任意地方，大小可以是系统所允许的最大存储容量。ColdFire 的堆栈使用 RAM 的方向是从大地址向小地址的，即堆栈从 SP 所指向的地址指针开始向下（小地址方向）生长，每次一个字节数据被压入堆栈，堆栈指针 SP 自动减小；每次一个字节数据出栈，堆栈指针 SP 自动增加。堆栈指针 SP 总是指向最后的压栈处。复位时，SP 从异常向量表中自动加载（地址 0x0，或向量号 0x0），当做一个特殊向量。

当进行子程序调用时，CPU首先将当前指令的后一条指令地址压入堆栈（返回地址），然后程序转入子程序执行。子程序返回指令（RTS）在子程序的结尾处执行，此时将之前压栈的返回地址装载到程序计数器PC中，程序从返回地址处继续执行。

3) 程序计数器 (PC)

程序计数器PC中存放内核正在运行的指令的地址。当前一个指令执行完成时，程序计数器会自动增加，指向下一条指令地址。当子程序调用和调用返回时，PC值被载入。

复位时，异常向量表中自动加载 PC 值（地址 0x4，或向量号 0x1），当做一个特殊向量。加载到 PC 的值是复位或系统初始化的第一条指令的指针。

当内核出现异常或中断时，PC值将被自动压入管理员堆栈中，然后根据内核产生的异常或中断情况加载相应的异常向量。

4) 条件码寄存器 (CCR)

状态寄存器SR的低8位字节为条件码寄存器CCR，该字节的每一位都是内核结果的标志位。其含义见表2-3。

表 2-3 CCR 位描述

位	域	描述
7-5	—	V2 版本应为 0。
4	X	扩展。为算法操作设置进位的值，否则不被影响或设置成特定的值。
3	N	负数标志。如果结果的最高位置位就设置该位，否则就清零。
2	Z	零标志位。如果结果等于 0，则设置该位，否则清零。
1	V	溢出标志位。如果发生算术溢出，就说明结果不能用操作数的字长来表示，就设置该位，否则清零。
0	C	进位标志。如果在加法时发生了进位，或减法时发生了借位，则设置该位，否则清零。

2. 仅管理员模式操作的寄存器

系统复位后, 进入管理员模式。在管理员模式下, 内核可以无限制地访问所有资源。由内核状态寄存器SR的S位决定管理员模式还是选择用户模式。在ColdFire内核中, 有多个管理员模式的寄存器, 它们在用户模式下是不能访问的。只有在管理员模式下才能改变系统运行环境, 这可以保护系统免遭运行中的用户程序破坏。当用户任务强行访问管理员模式下的资源时, 会出现特权违规异常错误。

1) 状态寄存器SR

状态寄存器 (Status Register) 的高8位为系统字节, 低8位为条件码字节。图2-3给出了状态寄存器SR各位的标志。

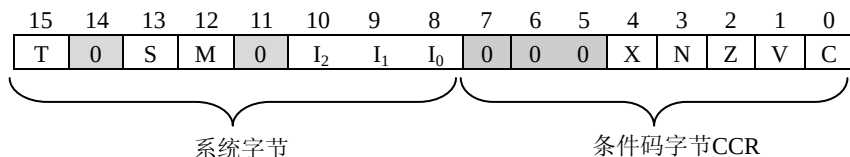


图2-3 状态寄存器SR

T——跟踪位。该位允许系统进行测试和调试。通过调试程序, 跟踪位可提供单步功能。若T位被置1, 则在每一条指令执行后, 内核开始跟踪异常。

S——管理员/用户状态位。该位指示内核处于管理员模式还是用户模式。当该位被置1时, 内核在管理员模式下执行; 反之, 内核处于用户模式下执行。在默认情况下, 当系统复位后, 该位被置1, 系统执行管理员模式下的启动代码。

M——主控/中断位。中断异常可以使该位清零, 也可以通过软件将该位置1。

I_[2:0]——中断优先屏蔽位。这3位用来定义当前的中断优先级。系统复位后, 三位 I_[2:0] 被设置为“111”, 可防止除7级中断以外的中断请求。7级中断是最高优先级, 且是非屏蔽中断。通常在系统初始化后, 将这3位设置为“000”, 以便允许所有CPU内核的中断。当出现中断时, 该位将自动设置为请求中断的级别, 所有优先级等于或低于当前中断级别的中断请求被屏蔽。但边沿触发的7级中断除外, 因为它是非屏蔽中断。正在执行的是7级中断时, 其他的7级中断仍然可被响应。

2) 管理员堆栈指针 (SSP)

管理员堆栈指针 (Supervisor Stack Pointer, SSP) 和用户堆栈指针 (User Stack Pointer, USP) 十分相似, 不同的是SSP只能在管理员模式下访问。系统复位后, SSP被自动加载异常向量表地址0x0; 在系统启动时, SSP立即指向管理员堆栈。

3) 向量基址寄存器 (VBR)

向量基址寄存器 (Vector Base Register, VBR) 存储ColdFire向量表的基地址。系统复位后, 该寄存器被加载为0, 指向0x0处的异常向量表。当异常或者中断确认后, ColdFire内核将确定引起异常或中断的事件向量号作为VBR所指异常向量基址的偏移量。

4) RAM基址寄存器和Flash基址寄存器 (RAMBAR&FLASHBAR)

存储器基址寄存器 (Memory Base Address Register) 用于存储片内SRAM或Flash模块的基址, 并指示这些模块的映射类型。每个寄存器都包括基地址、写保护位、地址空间屏蔽位和使能位。为了使片内RAM和Flash处于允许状态, 必须用理想的基址写这两个寄存器。

5) 内部外设系统基址寄存器 (IPSBAR)

内部外设系统基址寄存器 (Internal Peripheral System Base Address Register, IPSBAR) 定位了MCU内部外设系统占用的1GB的存储空间的起始地址。

另外还有Cache控制寄存器 (Cache Control Register, CACR)、两个访问控制寄存器 (Access Control Register) ACR0与ACR1, 与Cache操作相关, 这里不再介绍。

3. 寻址方式

ColdFire 指令集具有多种寻址方式, 提供了在寄存器和存储器中灵活而有效地存取操作数的方式。与 RISC 机制相比, ColdFire 微处理器能够直接使用处理器中的操作数, 而不需要借助内核寄存器, 就能完成读—修改—写操作。寻址方式包括寄存器直接寻址方式、立即寻址方式、地址寄存器间接寻址、后增地址寄存器间接寻址、预减地址寄存器间接寻址、16 位偏移量的地址寄存器间接寻址、变址和 8 位偏移量的地址寄存器间接寻址、16 位偏移量的程序计数器间接寻址、变址和 8 位偏移量程序计数器寻址、绝对短数据寻址和绝对长数据寻址。详情可参阅光盘 “ColdFire 编程参考手册”。

2.2.4 ColdFire异常和中断控制

中断指CPU正在处理某件事情的时候, 外部发生了某一事件, 请求CPU迅速去处理, CPU暂时中断当前的工作, 转入处理所发生的事件 (中断服务程序), 处理完后, 再回到原来中断的地方, 继续原来的工作, 这样的过程称为中断。中断服务程序所在的地址称为中断向量。实现中断功能的部件称为中断系统, 产生中断的请求源称为中断源。把由软件指令引起的一个程序的中断则称为软件中断; 而由某一特定功能引脚或基于芯片设备产生的中断则称为硬件中断。在嵌入式系统中, 中断是一个很重要的概念, 为了实时响应外界发生的事件, 必须依靠中断系统来实现。

异常是CPU强行从正常的程序执行切换到由某些内部或外部条件所要求的处理任务上去, 这些任务是优先于处理器正在执行的任务的。引起异常的外部条件是来自外围设备、硬件断点请求、访问错误和复位等; 引起异常的内部条件是指令、不对齐错误、违反特权级和跟踪等。ColdFire处理器把硬件复位和硬件中断都归类为异常, 把硬件复位看做是一种具有最高优先级的异常; 把来自外围设备的强行任务切换请求称为中断。处理器对复位、中断、异常具有同样的处理过程, 在谈及这个处理过程时统称为异常。

处理器在指令流水线的译码或者执行阶段识别异常, 若检测到一个异常, 则强行中止后面尚未流到该阶段的指令。对于在指令译码阶段检测到的异常, 以及对于与执行阶段有关的指令异常来说, 由于引起的异常与该指令本身无关, 指令并没有得到正确执行,

所以为该类异常保存的程序计数器的值指向引起该异常的指令，以便异常返回后重新执行。对于中断和跟踪异常，异常与指令本身有关，处理器在执行完当前这条指令才识别和检测这类异常，故为该类异常保存的PC值是指向要执行的下一条指令。

ColdFire 中由中断控制器处理所有异常和中断。ColdFire 支持 7 级中断嵌套，7 级中断优先级最高，依次递减，例如 2 级可以中断 1 级、3 级可以中断 2 级，等。当中断发生时，CPU 处理中断的流程是先挂起主程序，将机器状态信息保存到系统堆栈中，从中断向量表中取得中断向量找到中断处理函数入口，然后跳转到预先定义好中断处理程序入口，还需要根据该中断的中断级别设置 SR 寄存器的[I₂I₁I₀]，这样在执行中断程序时所有低级别的中断被有效屏蔽，执行中断处理程序。一旦 CPU 处理完中断，机器状态将被恢复，重新执行主程序。

异常向量表（Exception Vector Tablet, EVT）共有256个，如表2-4，ColdFire处理器使用前64个向量，用以处理复位、访问错误、地址错误、自动向量中断、系统调用；后面的192个向量留给用户使用，分配给I/O外设。

表 2-4 ColdFire 异常向量表

向量号	向量偏移值	入栈的程序计数器	分配
0	0x000	（特殊向量号与向量地址）	堆栈指针 SP 初始值
1	0x004	（特殊向量号与向量地址）	程序计数器 PC 初始值
2	0x008	出错指令	访问错误
3	0x00C	出错指令	地址错误
4	0x010	出错指令	非法指令
5	0x014	出错指令	除零错误
6~7	0x018~0x01C	—	保留
8	0x020	出错指令	特权违规
9	0x024	下一条指令	追踪
10	0x028	出错指令	无法执行行-a 操作码
11	0x02C	出错指令	无法执行行-f 操作码
12	0x030	下一条指令	调试中断
13	0x034	—	保留
14	0x038	出错指令	格式错误
15~23	0x03C~0x05C	—	保留
24	0x060	下一条指令	伪中断
25~31	0x064~0x07C	—	保留
32~47	0x080~0x0BC	下一条指令	陷阱指令#0~15
48~63	0x0C0~0x0FC	—	保留
64~255	0x100~0x3FC	下一条指令	用户定义中断

2.3 MCF52233的引脚功能与硬件最小系统

本书以80引脚LQFP封装的MCF52233芯片为例阐述ColdFire的编程和应用。在实际应用时MCF5223x系列的编程基本相似。实际上，以ColdFire为核心的系列微控制器的应用系统设计方法与编程方法基本上是一致的，只要以其中一个芯片为蓝本学习领会即可。

关键是必须深刻理解硬件系统及软件编程的基本要素。

2.3.1 MCF52233的引脚功能

图2-4给出的是80引脚LQFP封装的MCF52233的引脚图。每个引脚都可能有多重复用功能，有的引脚有两个复用功能，有的有四个复用功能，系统设计时必须注意只能使用其中的一个功能。一般情况下，需要按引脚功能分类了解MCU引脚功能情况，光盘中的“…\第02章 阅读材料\引脚功能表.doc”给出了按功能模块进行分类的80引脚LQFP封装MCF52233的引脚功能。进行硬件最小系统设计时，一般以引脚的第一功能作为引脚名进行原理图设计，若实际使用的是其另一功能，可以用括号加以标注，这样设计的硬件最小系统就比较通用。本书第16章给出了基于硬件构件的硬件原理图设计基本规则，在学习MCF52233的最小系统设计时，可先浏览该章主要内容。

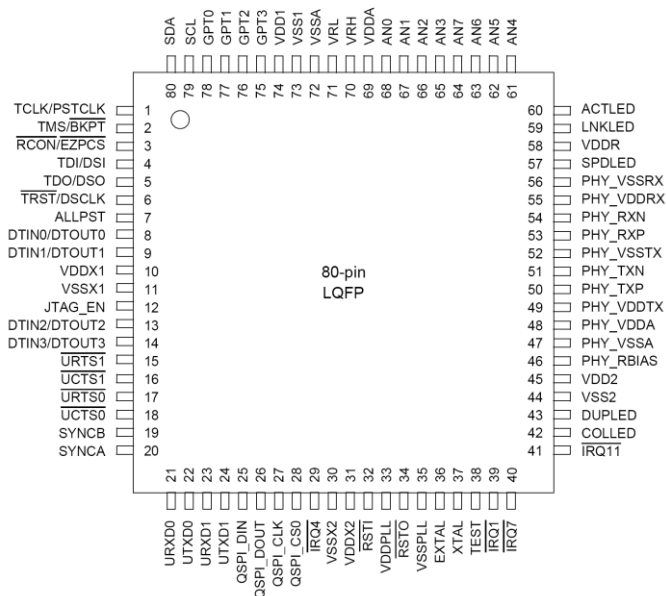


图2-4 80引脚LQFP封装的MCF52233的引脚分配图

80引脚LQFP封装MCF52233的引脚功能分类简介如下。

(1) 电源类信号

电源和地信号包括：VDDPLL、VSSPLL、VDDA、VSSA、VRH、VRL、VDDR、VSSX1、VDDX1、VDDX2、VSSX2、VDD2、VSS2、VDD1、VSS1、PHY_RBIAS、PHY_VDDA5、PHY_VDDR5、PHY_VDDTX5、PHY_VSSA。其中VDDPLL和VSSPLL是PLL模拟电源，为PLL模拟电路提供电源；VDDA和VSSA为ADC提供模拟电源；VRH和VRL为ADC模块提供模拟参考电源；PHY_RBIAS、PHY_VDDA5、PHY_VDDR5、PHY_VDDTX5和PHY_VSSA为EPHY模块提供电源；VDDR是以太网指示灯模块的电源；

其他引脚为MCU提供电源。

(2) 复位信号

复位信号包括复位输入引脚RSTI和复位输出引脚RSTO。

(3) 主要功能模块引脚

UART信号：包括串行数据输入URXD[1-0]、串行数据输出UTXD[1-0]、请求发送URTS[1-0]、清发送UCTS[1-0]。

ADC信号：包括AN[7:0]，ADC模块的输入引脚。

QSPI：包括串行数据输入QSPI_DIN、串行数据输出QSPI_DOUT、串行时钟QSPI_CLK和片选QSPI_CS0。

DMA定时器信号：包括DTIN/DTOUT[3-0]，32位定时器，可以产生DMA请求。

通用定时器信号：包括GPT[3-0]，16定时器。

I2C信号：包括串行时钟信号SCL和串行数据信号SDA。

快速以太网微控制器FEC信号：包括输入输出引脚PHY_RXP、PHY_RXN、PHY_TXN、PHY_TXP和指示灯引脚ACTLED、LNKLED、SPDLED、DUPLD、COLLED。

外部中断信号：包括IRQ1、IRQ4、IRQ11和IRQ7，可以作为外部中断源。

(4) 其他

PLL和时钟信号：包括外部时钟输入EXTAL、晶振引脚XTAL和时钟输出CLKOUT。

调试支持信号：包括JTAG_EN、TCLK/PSTCLK、TMS/BKPT、TRST/DSCLK、TDI/DSI、TDI/DSO、ALLPST，信号的具体描述在下一节介绍。

模式选择：复位配置引脚RCON。

2.3.2 MCF52233的硬件最小系统

MCU的硬件最小系统是指可以使内部程序运行的所必须的外围电路，也可以包括写入器接口电路。使用一个芯片，必须完全理解其硬件最小系统。当MCU工作不正常时，首先查找最小系统中可能出错的元件。一般情况下，MCU的硬件最小系统由电源、晶振及复位等电路组成。芯片要能工作，必须有电源与工作时钟，至于复位电路则提供不掉电情况下MCU重新启动的手段。由于Flash存储器制造技术的发展，大部分芯片提供了在板或在系统（On System）写入程序功能，即把空白芯片焊接到电路板上后，再通过写入器把程序下载到芯片中。这样，硬件最小系统应该把写入器的接口电路也包含在其中。基于这个思路，MCF52233芯片的硬件最小系统包括电源及其滤波电路、复位电路、晶振电路及PLL滤波电路、写入器接口电路。完整的电路在光盘的“…\第02章 阅读材料\mcf52233原理图pdf”目录中。下面分别对这些电路给出简明分析。

1. 电源及其滤波电路

电路中需要大量的电源类引脚用来提供足够的电流容量。所有的电源引脚必须外接适当的滤波电容抑制高频噪音。图2-5给出了MCF52233硬件最小系统的电源和地连接图。

电源 (VDDx) 与地 (VSSx) 包括很多引脚, 如VDDA、VSSA、VDD1、VSS1、VDDX1和VSSX1等。一些电源与地引脚仅用于外接滤波电容, 内部已经连接到电源与地, 芯片参考手册指出不需要再外接电源。至于电容外接, 是由于集成电路制造技术所限, 无法在IC内部通过光刻的方法制造这些电容。电源滤波电路, 用于改善系统的电磁兼容性, 降低电源波动对系统的影响, 增强电路工作稳定性。为标识系统通电与否, 可以增加一个电源指示灯。

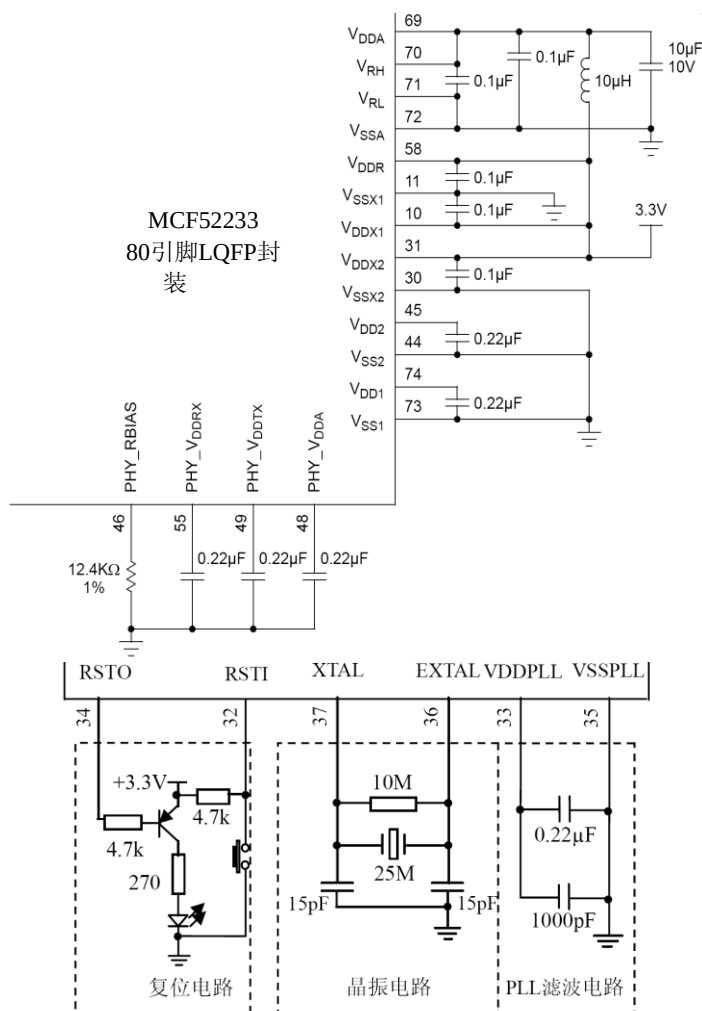


图2-6 MCF52233硬件最小系统的复位电路、晶振电路及PLL滤波电路

2. 复位电路

图2-6给出了MCF52233硬件最小系统的复位电路、晶振电路及PLL滤波电路。复位,意味着MCU一切重新开始。复位引脚有复位输入引脚 $\overline{\text{RSTI}}$ 和复位输出引脚 $\overline{\text{RSTO}}$ 。若 $\overline{\text{RSTI}}$ 信号有效(低电平),MCU复位,此时 $\overline{\text{RSTO}}$ 会产生一个低电平脉冲(512个时钟周期),肉眼观察,图2-6中复位电路的发光二极管会闪一下,这是正常复位情况^①。复位电路原理如下:正常工作时复位输入引脚 $\overline{\text{RSTI}}$ 通过4.7K欧姆电阻接到电源正极,所以应为高电平。若按下复位按钮,则 $\overline{\text{RSTI}}$ 脚接地,为低电平,导致芯片复位。若芯片能正常复位, $\overline{\text{RSTO}}$ 应该输出低电平,图中PNP型三极管导通,发光二极管亮。需要注意的是,如果 $\overline{\text{RSTI}}$ 或 $\overline{\text{RSTO}}$ 引脚被一直拉低,MCU将不能正常工作。

从引起MCU复位的内部与外部因素区分,复位分为外部复位与内部复位两种。外部复位有上电复位、按下“复位按钮”复位。内部复位有看门狗定时器复位、PLL失锁复位、PLL丢失时钟复位、软件复位、低电压复位等。

从复位时芯片是否处于上电状态区分,复位可分为冷复位与热复位。芯片从无电状态上电的复位属于冷复位,芯片处于带电状态的复位叫热复位。冷复位后,MCU内部RAM的内容是随机的。而热复位后,MCU内部RAM的内容保持复位前的内容,即热复位并不会引起RAM中内容的丢失。

从CPU响应快慢区分,复位可分为异步复位与同步复位。异步复位源提出的复位请求一般表示一种紧要的事件,因此复位控制逻辑不等到当前总线周期结束,复位立即有效。异步复位源有上电、丢失时钟、低压检测LVD、失去锁定、 $\overline{\text{RSTI}}$ 引脚(处在STOP模式)等。同步复位的处理方法与异步复位不同,当一个同步复位源提出一个复位请求时,复位控制器并不使之立即起作用,而是等到当前总线周期结束,这是为了保护数据的完整性。在该总线周期结束后,下一个系统时钟的上升沿时,复位才有效。同步复位有看门狗定时器、软件、 $\overline{\text{RSTI}}$ 引脚(不处在STOP模式)等。

3. 晶振电路及PLL滤波电路

晶振电路和PLL锁相环电路为芯片提供准确的工作时钟。

晶体振荡器分为无源晶振和有源晶振两种类型。需要外接电源的晶振称为有源晶振。无源晶振与有源晶振的英文名称不同,无源晶振为crystal(晶体),而有源晶振则叫做oscillator(振荡器)。无源晶振是有两个引脚的无极性元件,需要借助于时钟电路才能产生振荡信号,自身无法振荡起来。图2-6的硬件最小系统中使用无源晶振,无源晶振的符号为“—□—”,注意与电阻符号的区别。与晶振电路相关的引脚有晶体振荡器或外部时钟输入EXTAL引脚,晶体振荡器输出引脚XTAL。Freescale公司生产的MCU,其内部大多含有锁相环(Phase-Locked Loop, PLL)电路。锁相环PLL电路兼有频率放大和信号提纯的功能。通过内部PLL系统可以通过较低的外部时钟信号获得较高的内部工作频率,以降低因高速开关时钟所造成的高频信号干扰,提高系统抗干扰性。本芯片的PLL电路,由芯片内部提供电源,但其引脚VDDPLL和VSSPLL引出,供外接滤波电容之用。

^①这里给出的是通俗描述,目的是为了理解复位过程与检查电路之用,更为科学的描述见参考手册。

MCU系统时钟电路和电源电路在布PCB板时，要按照以下规则布线，才能使得系统的电磁兼容性得到保证：（1）用地线将晶振电路圈起来，各引线要尽量短。（2）石英晶振下面和对噪声特别敏感的器件下面不要走线。（3）PLL滤波电路要尽量靠近MCU，每个电源端和接地端都要接一个去耦电容，去耦电容也要尽量靠近MCU。

4. 写入器接口电路

MCF52233支持后台调试模式(Background Debug Mode, BDM)和JTAG(Joint Test Action Group, 联合测试行动小组)^①。后台调试模式BDM是由Freescall半导体公司自定义的片上调试规范。BDM调试方式为开发人员提供了底层的调试手段。开发人员可以通过它向目标板下载程序，同时也可以通过BDM调试器对目标板MCU的Flash进行写入、擦除、应用程序的下载和在线更新、在线动态调试和编程、读取CPU各个寄存器的内容、MCU内部资源的配置与修复、程序的加密处理等操作。而这些仅需要向CPU发送几个简单的指令就可以实现，从而使调试软件变得方便、快捷。

表 2-5 BDM 调试端口信号含义

信号	功能描述	所在引脚	方向（相对于MCU）
TCLK (Test Clock)	用于同步JTAG逻辑		
BKPT (Breakpoint)	断点中断请求输入信号。在当前指令执行完后处理器进入停机状态，停机状态在PST[3:0]上对应的值是0x0f。	2	输入
DSI(Development Serial Input)	调试模块的数据输入信号	4	输入
DSO(Development Serial Output)	调试模块输出的应答信号	5	输出
DSCLK(Development Serial Clock)	串行时钟输入信号。最大的工作频率是PSTCLK的1/5。在上升沿，DSI信号线的数据被采样，DSO则改变状态。	6	输入
ALLPST(All Processor Status Outputs)	处理器状态PST[3:0] ^② 的逻辑与。	7	输出
JTAG_EN (JTAG Enable)	当该引脚为低电平时，进入BDM模式；当该引脚为高电平时，进入JTAG模式。	21	输入

图2-7是MCF52233的写入模块电路，其中，M52233-BDM-PORT是与写入器连接的26引脚插座，该插座的引脚信号除了上述的几种信号外，还包括3.3V电源输入信号、地线、复位输入信号以及复位输出信号。

^① JTAG 是一种国际标准测试协议（IEEE 1149.1 兼容），有助于系统诊断和制造测试。该模块通过标准 4 引脚测试访问端口(TAP)和 JTAG 复位引脚 TRST 可以访问所有的数据和芯片控制引脚。在 ColdFire 中，JTAG 一般用于出厂测试和加密后的整体擦除，而普通用户不使用，本书不做介绍。

^② 处理器状态 PST[3:0] (Processor Status Outputs)：报告处理器状态的输出信号，方向为输出，它指示了处理器当前的状态，但与当前的总线传输无关，它也是在每个处理器时钟周期被刷新。有的芯片没有这四个引脚（比如 80 引脚 LQFP 封装的 MCF52233），使用 ALLPST 来代替。

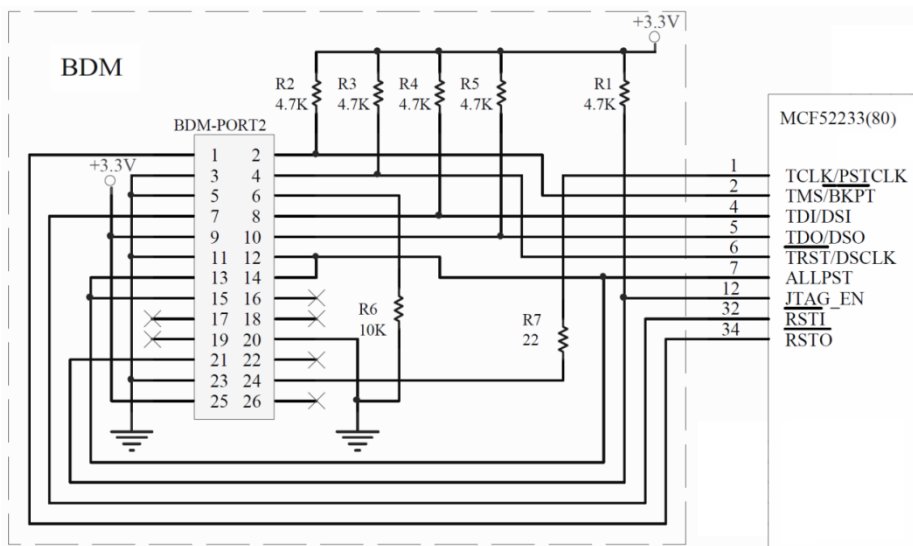


图2-7 MCF52233写入模块电路

写入器可以通过并口或USB接口与PC机连接。鉴于USB接口使用方便，苏州大学提供的“32位ColdFire写入器”为USB2.0接口，适用于集成开发环境CodeWarrior for ColdFire 6.3。一般编程时，Flash未被加密，可以使用BDM模式与目标板通信，但若Flash被加密，必须首先擦除密码，方可通过BDM模式对Flash区重新操作。具体方法见光盘中“《32位ColdFire写入器》使用说明”。

2.3.3 硬件最小系统测试方法

前面介绍了硬件最小系统的设计，给出了硬件最小系统元件的参考值（更详细的元件信息请参考光盘的“…\第02章 阅读材料\MCF52233 原理图 pdf”目录）。硬件最小系统电路原理图如图2-8。

根据原理图制作了印刷电路板后，就开始硬件电路板的焊接和测试了。具体过程如下：

（1）焊接电源及其滤波电路、复位电路、晶振电路、PLL 滤波电路以及写入器接口电路。注意：电源的滤波电容不可漏焊，否则芯片所受干扰较大，影响调试。

（2）在确保电源和地未短路的情况下接通电源，测量电压是否正常，检查按下复位按钮是否能够复位（观察复位指示灯）。

（3）将写入器与电路板连接，启动开发环境 CodeWarrior 6.3，对目标 MCU 进行擦除，如果成功则说明最小系统工作正常。

（4）将第一个样例程序编译、连接生成 S19 文件，并下载到 Flash 中，观察小灯闪烁情况。

（5）硬件最小系统测试通过以后就可以进行其他模块焊接。正确的做法是，焊完一个模块后，应紧接着测试该模块工作是否正常，切忌焊接多个模块后再进行测试，因为一旦出现问题，就很难定位具体是哪个模块的问题。

第3章 第一个样例程序及ColdFire工程组织

本章阐述“入门”过程的(5) — (8)步,通过这个过程,完成第一个 ColdFire 工程的入门。利用 GPIO 模块编程控制发光二极管作为入门例子,给出 ColdFire 工程组织、框架,阐述各个文件的功能,主要目的是使读者理解程序框架和工作过程。在此基础上进行实际环境的编译、链接生成机器码、将机器码下载芯片内部 Flash 中,进行调试、运行。重点是透彻理解第一工程的执行过程。

3.1 通用I/O接口基本概念及连接方法

1. I/O接口的概念

I/O 接口,即输入输出接口,是微控制器同外界进行交互的重要通道。这里的接口英文是 port,也可以翻译为“端口”,另一个英文单词是 interface,也翻译为接口。从中文文字面看,接口与端口似乎有点区别,但在嵌入式系统中它们的含义是相同的。有时 I/O 引脚称为接口(interface),而把用于对 I/O 引脚进行编程的寄存器称为端口(port),实际上它们是紧密相连的。因此,不必深究它们之间的区别。有些书中甚至直接称 I/O 接口(端口)为 I/O 口。在嵌入式系统中,接口千变万化,种类繁多,有显而易见的人机交互接口,如操纵杆、键盘、显示器;也有无人介入的接口,如网络接口、机器设备接口。

2. 通用I/O

所谓通用 I/O,也记为 GPIO (General Purpose I/O),即基本的输入/输出,有时也称并行 I/O,或普通 I/O,它是 I/O 的最基本形式。本书中使用正逻辑,电源(Vcc)代表高电平,对应数字信号“1”;地(GND)代表低电平,对应数字信号“0”。作为通用输入引脚,MCU 内部程序可以通过端口寄存器读取该引脚,知道该引脚是“1”(高电平)或“0”(低电平),即开关量输入。作为通用输出引脚,MCU 内部程序通过端口寄存器向该引脚输出“1”(高电平)或“0”(低电平),即开关量输出。大多数通用 I/O 引脚可以通过编程来设定工作方式输入或输出,称之为双向通用 I/O。

3. 上拉下拉电阻与输入引脚的基本接法

芯片输入引脚的外部有三种不同的连接方式:带上拉电阻的连接、带下拉电阻的连接和“悬空”连接。通俗地说,若 MCU 的某个引脚通过一个电阻接到电源(Vcc)上,这个电阻被称为“上拉电阻”。与之相对应,若 MCU 的某个引脚通过一个电阻接到地(GND)上,则相应的电阻被称为“下拉电阻”。这种做法使得,悬空的芯片引脚被上拉电阻或下拉电阻初始化为高电平或低电平。根据实际情况,上拉电阻与下拉电阻可以取值在 $1\text{K}\Omega \sim 10\text{K}\Omega$ 之间,其阻值大小与静态电流及系统功耗相关。

图 3-1 给出了一个 MCU 的输入引脚的三种外部连接方式,假设 MCU 内部没有上拉

或下拉电阻，图中的引脚 I3 上的开关 K3 采用悬空方式连接就不合适，因为 K3 断开时，引脚 I3 的电平不确定。在图 3-1 中， $R1 \gg R2$ ， $R3 \ll R4$ ，各电阻的典型取值为： $R1=20K$ ， $R2=1K$ ， $R3=10K$ ， $R4=200K$ 。

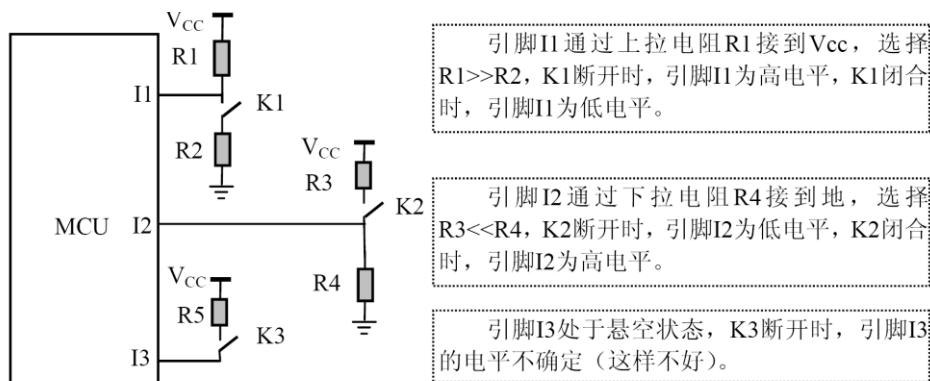


图3-1 I/O口输入电路

4. 输出引脚的基本接法

作为通用输出引脚，MCU 内部程序向该引脚输出高电平或低电平来驱动器件工作，即开关量输出。如图 3-2 所示。

一种接法是 O1 引脚直接驱动发光二极管 LED，当 O1 引脚输出高电平时，LED 不亮；当 O1 引脚输出低电平时，LED 点亮。这种接法的驱动电流一般在 2mA~10mA。

另种接法是 O2 引脚通过一个 NPN 三极管驱动蜂鸣器，当 O2 脚输出高电平时，蜂鸣器响；O2 脚输出低电平时，蜂鸣器不响。这种接法的驱动电流可达 100mA 左右，而 O2 引脚控制电流可以在几个 mA 左右。

若负载需要更大的驱动电流，就必须另外的驱动电路，但对 MCU 编程来说，没有任何影响。

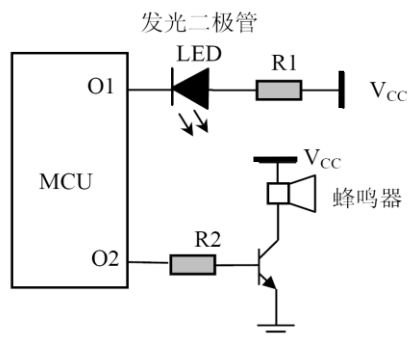


图3-2 I/O口输出电路

3.2 MCF52233的GPIO

3.2.1 MCF52233 GPIO编程的基本原理

MCF52233 的大部分引脚具有多重功能，可以通过编程设定使用其中一种功能。GPIO 作为基本功能，MCF52233 有 13 个 GPIO 口。图 3-3 给出了 MCF52233 的 13 个 GPIO 口

的引脚分布情况。

每个 GPIO 口的名称由两位英文字母组成，分别是 AN、DD、GP、AS、TA、TC、TD、UA、UB、UC、QS、LD、NQ。

GPIO 的基本编程方法：

- (1) 通过“引脚配置寄存器”配置相应引脚为 GPIO 功能；
- (2) 通过“数据方向寄存器”设置相应引脚为输入或输出；
- (3) 若是输出引脚，则通过“端口输出数据寄存器”设置引脚输出高电平或低电平；
- (4) 若是输入引脚，则通过“端口引脚数据/设置数据寄存器”获得引脚的状态。

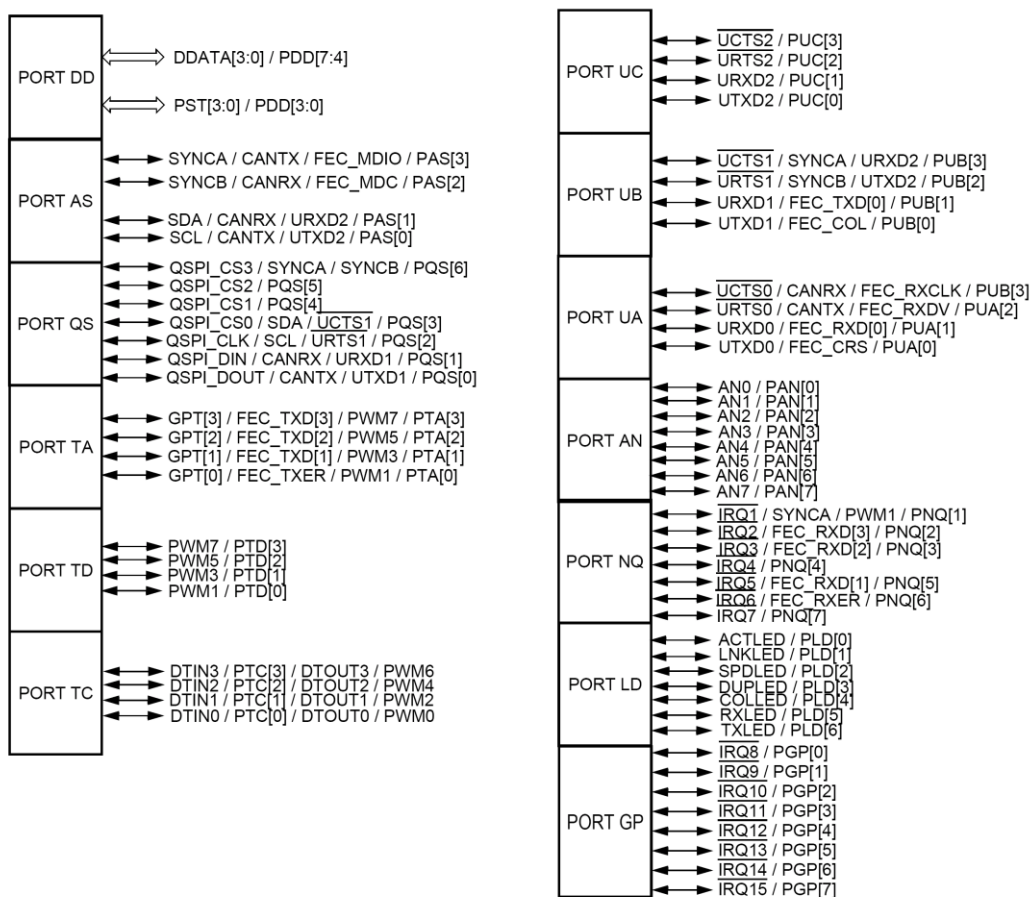


图3-3 GPIO模块框图

3.2.2 GPIO模块寄存器与GPIO的编程基本方法

GPIO 模块的每个口最多对应 8 个 GPIO 引脚，但各个 GPIO 口的编程寄存器均为 8 位，只是没有对应引脚的位无效。

GPIO 模块寄存器的命名有一定的规范，所有寄存器都在 MCF52233 芯片寄存器及相

关位定义头文件 mcf52233.h 中定义。其中“引脚配置寄存器”是前缀 P+该端口的名称+PAR。

“端口输出数据寄存器”是 PORT+该端口的名称，“端口输出方向寄存器”是 DDR+该端口的名称。所有寄存器的位编号从 0 开始，且最低位编号为 0。例如，一个 8 位寄存器的最低位编号为 0，最高位编号为 7。

例如：TC 端口的引脚配置寄存器标记为 PTCPAR、端口输出数据寄存器记为 PORTTC、端口输出方向寄存器记为 DDRTC。

1. 操作GPIO的基本寄存器

1) 引脚配置寄存器PAR (Pin Assignment Register)

可以设置当前 I/O 口是使用 GPIO 功能还是使用其复用的特殊功能。引脚可以有两种、或四种功能。PANPAR、PDDPAR、PGPPAR、PLDPAR、PTDPAR、PUCPAR 引脚有两个功能，由一位决定该引脚的功能，该位为 0 则为 GPIO 功能。PNQPAR、PQSPAR、PASPAR、PTAPAR、PTCPAR、PUAPAR、PUBPAR 引脚有四个功能，由两位决定该引脚的功能，相应两位为 00 则选择 GPIO 功能。实际编程需要查阅参考手册。本书的例程已经给出了完整的封装，可以直接使用。

2) 端口数据方向寄存器DDR (Port Data Direction Register)

这些位分别控制着端口引脚是输入还是输出，若为 0，则引脚为输入，若为 1，则引脚为输出。复位时为 0x00。

3) 端口输出数据寄存器PORT (Port Output Data Register)

若引脚被配置为输出，PORTn 寄存器中每一位数据决定了对应引脚的输出电平。复位时 PORTn 寄存器的所有被使用的位为 1。

4) 端口引脚数据/设置数据寄存器SET (Port Pin Data/Set Data Register)

SETn 寄存器反应引脚的当前状态。若引脚被配置成输入，通过读取该寄存器，获得引脚状态。同时，也向 SETn 寄存器写 1 置位 PORTn 寄存器的相应位，写 0 无影响。

2. GPIO的基本编程方法举例

在对有关基本寄存器功能有所了解后，下面将以对 TC 口第 3 脚的 GPIO 功能演示代码为例让读者对上述寄存器有更深入的了解，完整工程请参阅光盘中本章的例程。

首先，为了程序通用性，对要设置的引脚进行宏定义，如需对第 2 脚操作只需将下面的“3”修改为“2”即可。现在，设要使用的引脚名叫“RUNpin”，接在 TC 口的第 3 引脚上，则需使用如下宏定义：

```
#define RUN_PORT  PORT_TC  //灯使用的端口
#define RUNpin    3        // 用 RUNpin 代替“3”
```

注意，PORT_TC 是一个代表 TC 端口寄存器的 32 位地址指针，在芯片的映像文件中定义。这样，程序员就可以用普通 C 语言对该寄存器进行操作了。

当需要对寄存器的具体位进行设置，而不改变该寄存器中其他位的值时，通常会使用

位操作，例如：

```
RUN_PORT &= 0b11110111 //表示将 RUN_PORT 的第 3 位设置为 0
```

```
RUN_PORT |= 0b00001000 //表示将 RUN_PORT 的第 3 位设置为 1
```

为了程序通用性，以上两行代码需要写成：

```
RUN_PORT &= ~(0x01<< RUNpin) //将 RUN_PORT 的第 3 位设置为 0
```

```
RUN_PORT |= 0x01<< RUNpin //将 RUN_PORT 的第 3 位设置为 1
```

这样做的好处是，当需要改变所使用的引脚后，只需要改变 RUN_PORT 和 RUNpin 的宏定义，这也是软件工程的基本原则的体现。另外，若要对连续几位进行操作，则可以用以下代码来实现：

```
RUN_PORT &= ~(0x03<< RUNpin) //将 RUN_PORT 的第 3、4 位都设置为 0
```

```
RUN_PORT |= 0x03<< RUNpin //将 RUN_PORT 的第 3、4 位都设置为 1
```

以上的例子说明了怎样设置具体寄存器的单独 1 位或连续 2 位的位值，若要对连续的 3 位或更多的位进行设置，只需要更改 0x03 为合适的值即可（3 位对应 0x07）。

1) 配置 TC 口的 RUNpin 引脚为 GPIO 功能

TC 口有四个复用功能，根据“引脚配置寄存器”的编程方法，TC 口的每个引脚功能具体定义为何种功能，由 TC 口引脚配置寄存器 PTCPAR 中的相应两位决定。在本书的工程实例中，PTCPAR 被加上“MCF_GPIO_”前缀，变成 MCF_GPIO_PTCPAR，用来表示 TC 口引脚配置寄存器，其他寄存器类似。

```
MCF_GPIO_PTCPAR &= ~((0x03)<<(2*( RUNpin ))) //定义 TC.RUNpin 为 GPIO
```

2) 设置 TC 口的 RUNpin 引脚为输出

```
MCF_GPIO_DDRTC |= (0x01<<RUNpin) //设置 TC. RUNpin 为输出
```

3) 在 TC 的 RUNpin 引脚上输出高/低电平

```
MCF_GPIO_PORTTC |= (0x01<<RUNpin) //TC. RUNpin=1(高电平)
```

```
MCF_GPIO_PORTTC &= ~(0x01<< RUNpin) // TC. RUNpin=0(低电平)
```

4) 设置 TC 口 RUNpin 引脚为输入并读取 RUNpin 引脚状态

```
MCF_GPIO_DDRTC &= ~(0x01<<RUNpin) //设置 TC. RUNpin 为输入
```

```
value = MCF_GPIO_SETTC &(0x01<< RUNpin) //获取 TC. RUNpin 状态
```

3. GPIO 功能的其他寄存器

除了基本编程寄存器外，还有其他几个寄存器对 I/O 口进行操作。

1) 清零寄存器 CLR (Port Clear Output Data Register)

向 CLRN 写 0 可以清零 PORTn 寄存器的相应位，写 1 无影响。读 CLRN 寄存器返回 0。在编程中可不使用该寄存器，而是使用“端口输出数据寄存器”控制输出高/低电平。

2) 引脚线或寄存器 PWOR (Pin Wired OR Register)

引脚线或寄存器 PWOR (Pin Wired OR Register) 可读写并且每位复位为 0。0 为正常

操作；1 为线或操作。若设置为线或，需要使用上拉电阻。

3) 引脚驱动能力寄存器 PDSR (Pin Drive Strength Register)

引脚驱动能力寄存器 PDSR (Pin Drive Strength Register) 包括 PDSR0 和 PDSR1，可读写，在单片模式时 (MCF52233 默认状态) 下复位为 0，在 EzPort 和 FAST 模式时为 1。0 引脚配置为低驱动能力 (2mA)；1 引脚配置为高驱动能力 (10mA)。

3.3 开发套件 CodeWarrior 开发环境与 ColdFire 写入器

嵌入式软件开发有别于桌面软件开发的一个显著的特点，是它一般需要一个交叉编译和调试环境，即编辑和编译软件在通常的 PC 机上进行，而编译好的软件需要通过写入工具下载到目标机上执行，如 MCF52233 的目标机上。由于主机和目标机处理器的体系结构彼此不同，从而增加了嵌入式软件开发的难度。所以选择一些好的开发套件有助于对目标机的学习与开发。本书将介绍 Freescale 公司的 CodeWarrior for ColdFire V6.3 集成开发环境（简称 CW 环境）、苏州大学的 WA-MCF52233EVB 硬件评估板以及 WA-I 型 32 位 ColdFire 写入器。

3.3.1 CodeWarrior 开发环境简介与基本使用方法

1. CodeWarrior 环境功能和特点

CodeWarrior 开发环境(简称 CW 环境)是 Freescale 公司研发的面向 Freescale MCU 与 DSP 嵌入式应用开发的商业软件工具，其功能强大，是 Freescale 向用户推荐的产品。

CodeWarrior 分为 3 个版本：特别版 (Special Edition)、标准版和专业版。特别版是免费的，用于教学目的，对生成的代码量有一定限制，C 语言代码不得超过 12KB，对工程包含的文件数目也限制在 30 个以内。标准版和专业版没有这种限制。3 个版本的区别在于用户所获取的授权文件 (license) 不同，特别版的授权文件随安装软件附带，不需要特殊申请，标准版和专业版的授权文件需要付费。CodeWarrior 特别版、标准版和专业版的定义随所支持的微处理器的不同而不同，如 CodeWarrior for HC08 V6.0、CodeWarrior for HC12 V4.6、CodeWarrior for ColdFire V6.3 等。

CW 环境包括以下几个功能模块：编辑器、源码浏览器、搜索引擎、构造系统、调试器、工程管理器。编辑器、编译器、连接器和调试器对应开发过程的四个主要阶段，其它模块用以支持代码浏览和构造控制，工程管理器控制整个过程。该集成环境是一个多线程应用，能在内存中保存状态信息、符号表和对对象代码，从而提高操作速度；能跟踪源码变化，进行自动编译和连接。

2. CW 环境安装与设置

CW 环境安装没有什么特别之处，在 Windows 操作系统上，只要按照安装向导单击鼠标就可以自动完成。

需要说明的是,安装完毕以后要上网注册以申请使用许可 (license key)。无论是下载的软件还是申请到的免费光盘,安装后都要通过因特网注册,以申请使用许可 (licenseKey)。这里可通过登录其网站,单击“Request a Key”实现。由于这一注册过程是在网上自动实现的,故只要网络通畅,这个往返过程在数分钟之内即可完成。申请后会通过 E-MAIL 得到一个 License.dat 文件。将该文件复制到相应目录下即可,例如:“C:\Program Files\Freescale\CodeWarrior for ColdFire V6.3\”。对于免费的特别版本,安装好后用 License.dat 覆盖安装目录下的 License.dat。

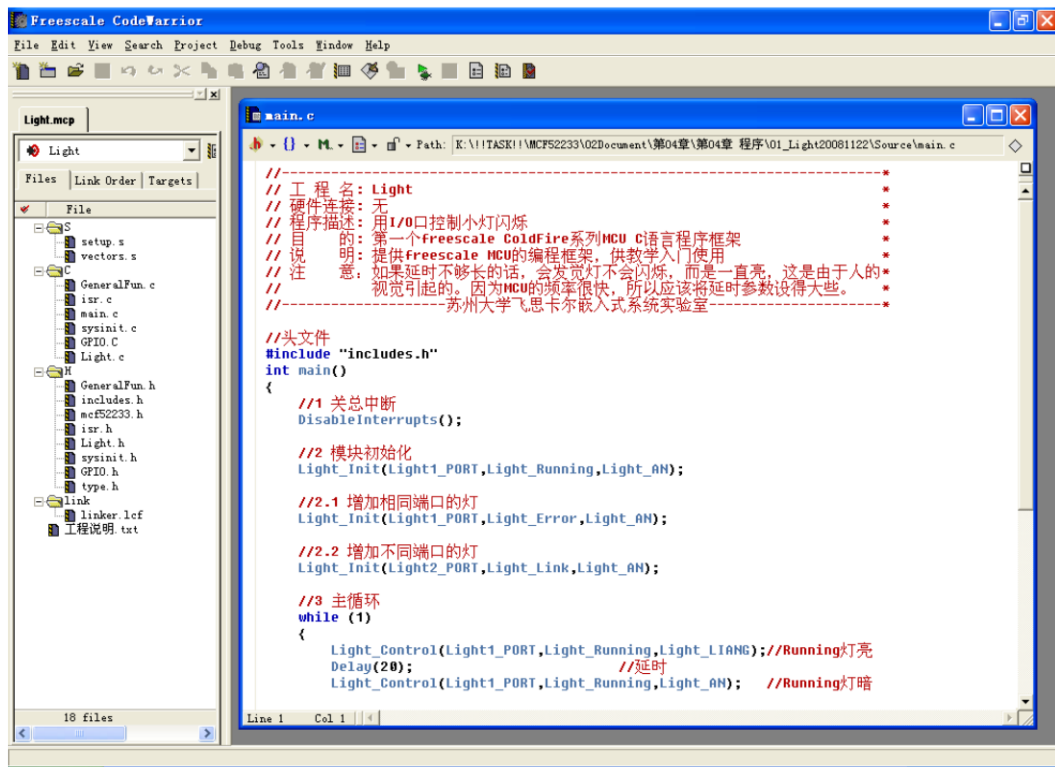


图3-4 CW环境运行界面

CW 环境的运行界面如图 3-4 所示。

由于 CodeWarrior IDE 安装后的默认字体是 Courier New,对中文的支持不完善,因此建议修改字体。选择 Edit->Perferences...,弹出 IDE Preferences 对话框。在 Font&Tabs 选项设置字体为 Fixedsys,Script 为 CHINESE_GB2312,由于 Tab 在不同文本编辑器解释不同,建议在 Tab Inserts Spaces 前打钩,使 Tab 键插入的是多个空格。

光盘中有相关简明使用方法的文档,本章随后的内容也有引导读者如何进行编辑、编译、程序运行、调试等阐述。

3.3.2 WA- I 型32位ColdFire写入器

开发人员可以通过 WA- I 型 32 位 ColdFire 写入器对目标板中的 Flash 进行擦除、写

入等操作。将机器码下载到 Flash 后，可以进行程序的运行、调试。图 3-5 给出了写入器的实物图。使用该写入器时，一端连接 PC 的 USB 口，另一端连接目标板 BDM 接口。



图3-5 ColdFire系列MCU写入器实物图

3.3.3 MCF52233EVB硬件评估板

SDMCF52233EVB 硬件评估板如图 3-6 所示。该硬件评估板使用 80 引脚的 MCF52233CAF60 芯片，带以太网接口、RS232 串口、一个复位按钮、两个按键、引出所有 I/O 口、并带有与苏州大学 Freescale 嵌入式系统实验室的 MT-扩展板- I 型的对接口。外接 5V 直流电源，板内转为 3.3V 给芯片供电。



图3-6 MCF52233EVB硬件评估板

本书附录中提供了有关硬件材料的购买信息。

3.4 CW工程文件组织

嵌入式系统工程往往包含很多文件，如：程序文件、头文件、与编译调试相关的信息文件、工程说明文件以及工程目标代码文件等。工程文件的合理组织对一个嵌入式系统工程犹为重要，它不但会提高项目的开发效率，同时也降低项目的维护难度。

嵌入式系统的文件组织方法以硬件对象为核心来展开，系统中每个对象应包含相关的头文件、程序文件及说明文件等。以硬件对象的方式来组织文件，会使得工程结构清

晰，调试定位方便，后期维护容易，这也是嵌入式系统软件工程的基本思想。

3.4.1 工程文件的组织

将光盘中的文件夹“...\\02-MCF52233Program\\01_Light”工程文件夹添加到安装文件所在的文件夹下，比如“C:\\Program Files\\Freescall\\CodeWarrior for ColdFire V6.3\\Stationery\\ColdFire”文件夹下。该模板是苏州大学飞思卡尔嵌入式系统实验室专门为 MCF52233EVB 开发板设计的工程模板。读者在新建工程时可以选择该模板。该模板方便易懂，与 CodeWarrior 提供的 CF_m52233DEMO 的工程模板相比文件少，去掉了一些初学者不易理解且不是必须的文件，同时应用底层软件构件的概念改进了程序结构，目的是一开始就引导读者进行规范的文件组织与编程。

新建工程有两种方法，一种是使用工程模板，另一种是使用已存在的工程来建立另外一个工程。

第一种方法的操作步骤如下：

选择 File->New，弹出新建对话框，选择“ColdFire Stationery”，在 Project name 中输入工程名，在 Location 中选择工程所在目录。单击确定即可。

新建工程后，通常要设置某些工程属性，例如新建 Light 工程后，选择菜单 Edit->Demo Settings...，弹出 Demo Settings 对话框，选择 Target Settings，将 Target Name 修改成 Light；选择 ColdFire Target，将 File Name 修改成 Light.elf，这样开发环境在编译后生成的目标文件名即为 Light。

第二种方法是使用已存的工程来建立另一个工程。当在已有工程的基础上，做另一个项目时，比如在 Light 工程的基础上编写 LCD 程序，需要进行如下设置：

- (1) 更改工程文件夹名为 LCD
- (2) 更改 Light.mcp 为 LCD.mcp
- (3) 文件夹 Light_Data 更改为 LCD_Data
- (4) 将 bin 文件夹的所有内容删掉
- (5) 打开该工程，选择菜单 Edit->Light Settings...，弹出 Light Settings 对话框，选择 Target Settings，将 Target Name 修改成 LCD；选择 ColdFire Target，将 File Name 修改成 LCD.elf。

下面以 3.5 小节所述的控制小灯闪烁工程为例，介绍基于 CW 环境的嵌入式工程文件组织方法。图 3-7 给出了该工程相关源文件的树型结构，可分为“汇编程序”、“头文件”、“C 语言程序”、“链接文件”、“工程说明文件”五个部分。其中“头文件”又可分为“总体框架头文件”与“软件构件头文件”；相应地，“C 语言程序”分为“总体框架程序”与“软件构件”两部分。

“汇编程序”中包含的文件与工程初始化相关，包括工程启动文件“setup.s”与中断向量表文件“vectors.s”，由于源代码完全使用汇编语言编写，故将其独立作为一类。对于其他的文件，可以很容易找到一个头文件和一个源文件的对应关系，例如，includes.h 对应 main.c 文件，isr.h 对应 isr.c 文件，而 GPIO.h 对应 GPIO.c 文件，它们都是按照构件化

的原则来进行切割和组织的。

请特别注意 `main.c` 和 `isr.c` 这两个文件，从源程序角度来看，一般嵌入式软件的执行流程如下：系统启动并初始化后，程序根据 `main.c` 中定义的主循环顺序执行；当遇到中断请求时，转而执行 `isr.c` 中定义的相应中断处理程序；中断处理结束，则返回中断处继续顺序执行。由于 `main.c` 和 `isr.c` 文件反映了软件系统的整体执行流程，故而在工程文件组织时，将它与其余 C 语言程序文件分开管理，同样，与这两个文件对应的 `includes.h` 和 `isr.h` 文件也被单独拿出，放在头文件的根目录下。

此外，与总体框架程序相关的头文件和源文件分别放在了 `Frame_H` 和 `Frame_C` 子文件夹中，以归类管理。`Frame_H` 里包含了 `type.h`、`mcf52233.h`、`GeneralFun.h` 和 `sysinit.h` 四个头文件。`type.h` 用于类型别名定义，它将 C 语言中用于变量类型定义的关键字简化定义成比较简短的形式，这样，开发者在定义变量时，可以不必敲入冗长的变量定义关键字，同时，这也为不同编译体系间的代码复用和移植提供了方便。`mcf52233.h` 是 MCF52233 芯片寄存器及相关位定义头文件，它可被视为芯片的接口文件，没有这个文件，就不可能对该芯片进行任何操作。`sysinit.h` 与 `Frame_C` 子文件夹中的 `sysinit.c` 对应，它定义了系统初始化时的基本参数，如系统时钟等，而 `sysinit.c` 文件则包含实际初始化代码。`GeneralFun.h` 与 `GeneralFun.c` 对应，它提供常用且基本的软件功能性子函数，如延时子函数等。这些文件对于系统的正常运行都是不可或缺的。

以上所说，都是系统正常运行所必需的，它们只是实现了“最小系统”。若要系统能做实际的事情，还必须添加相关功能代码。按照构件化原则，这些代码被安置于 `Component_H` 与 `Component_C` 子文件夹中。每个功能实体，或叫作“构件”，都对应一个 `.c` 文件和一个 `.h` 文件。例如，用于指示灯控制的“Light”构件，就对应了 `Light.c` 和 `Light.h`，它们分别放置于 `Component_C` 和 `Component_H` 子文件夹中。构件主要是按照功能进行划分的，除了这里定义的“Light”构件和“GPIO”构件，以后的章节还会出现“串行通信”、“键盘”、“LED”、“液晶”等构件，它们都是包含在这两个文件夹里的。

另外，嵌入式系统工程框架中还必须包括后缀名为 `.lcf` 的特殊格式文件，它是一个地址链接文件，用于告诉编译器代码是如何安放在具体的地址空间的。了解该文件的格式，有助于全面理解嵌入式系统的运作，本章后续章节会简要介绍它的内容。特别注意，要重视工程说明文件，必要的工程说明对于软件开发者是非常必要的。

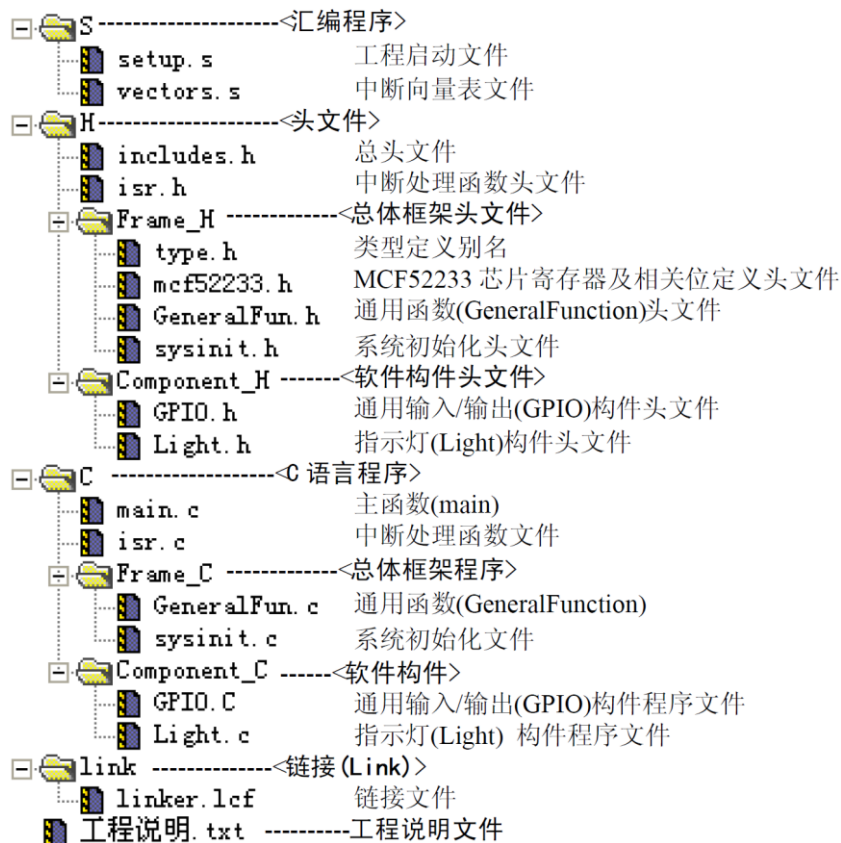


图3-7 小灯闪烁工程相关源文件的树型结构

3.4.2 初始化相关文件

1. 汇编程序

1) 启动文件setup.s

由于启动模块主要完成对目标机系统的初始化,包括初始化基本硬件模块,设置堆栈,配置中断向量基址,分配 RAM 空间等,因此其代码与微处理器硬件体系架构关系密切,一般使用汇编语言实现。该文件不建议修改。ColdFire 启动模块程序实现步骤主要如下:

- (1) 屏蔽所有中断。
- (2) 允许使用内部 RAM。
- (3) 设置堆栈指针,使其映射到 RAM 空间。
- (4) 初始化内部外设基址寄存器 (Internal Peripheral System Base Address Register, IPSBAR)。
- (5) 允许使用内部 Flash 存储器。
- (6) 调用 MCF52233 的各个内部模块的初始化函数,主要包括设置 IRQ (Interrupt

ReQuest) 中断, 初始化看门狗, 配置系统时钟, 设置片内 RAM, 初始化程序运行环境等。

(7) 跳转到主函数执行。

启动文件包含启动代码, 是硬件复位后的函数入口, 启动代码会初始化 C 语言的运行环境, 然后进入 C 程序代码。具体代码如下:

```
//-----*
//函数名:asm_startmeup                                     *
//功 能:硬件复位后的函数入口。内存映射根据链接文件的定义设置, *
//      然后调用更高级的系统初始化例程。最后程序跳转到“main”程序 *
//参 数: 无                                               *
//返 回: 无                                               *
//-----*
asm_startmeup:
_asm_startmeup:

//备份芯片信息D0和D1①: D0寄存器提供处理器配置信息, D1提供内部存储器配置的信
息
//先备份D0和D1到D6和D7, 后面会使用D0和D1,
//而D0和D1在系统加电之初保存有芯片相关信息
    move.l    d0, d6
    move.l    d1, d7

    //初始化RAMBAR1: 定位SRAM并使其生效
    //设置RAMBAR1的SPV和V位为1,
    //SPV: 允许DMA和FEC访问内存
    //V   : 设置SRAM模块有效。

move.l    #__SRAM, d0
    add.l    #0x21, d0
    movec    d0, RAMBAR1

//定位堆栈指针
move.l    #__SP_INIT, sp

    //初始化IPSBAR
move.l    #__IPSBAR, d0
    add.l    #0x1, d0
move.l    d0, 0x40000000

    //初始化FLASHBAR
move.l    #__FLASH, d0
    cmp.l    #0x00000000, d0
    bne      change_flashbar
    add.l    #0x61, d0
    movec    d0, RAMBAR0

_continue_startup:
```

^① D0 和 D1 内所包含的详细信息请参见光盘中“ColdFire 编程参考手册.pdf”文件。

```

//定位堆栈指针
move.l  #_SP_INIT, sp

//系统初始化
jsr     mcf52233_init

//跳转到main程序执行
jsr     main

```

2) 异常向量表文件vectors.s的定义

向量表定义在前面的章节已经介绍了，该文件只有在增加中断和设置 Flash 密码时才需要修改，增加中断的方法在 3.6 节介绍。在异常向量表的后面是 CFM 配置域的内容，地址从 0x400~0x417，里面存放了加密 Flash 的一些信息，一般设置为全零即不加密，加密的方法请参考“第 10 章 Flash 存储器在线编程”。异常向量表文件如下：

```

//-----*
//                                           *
//                               异常向量表           *
//                                           *
//-----*
VECTOR_TABLE:
_VECTOR_TABLE:
INITSP:    .long  __SP_INIT  /*Initial SP*/
INITPC:    .long  start      /* Initial PC*/
vector02:  .long  _asm_exception_handler /* Access Error */
...
vectorFE:  .long  _irq_handler
vectorFF:  .long  _irq_handler
//在此处添加密码
KEY_UPPER: .long  0x00000000
KEY_LOWER: .long  0x00000000
CFMPROT:   .long  0x00000000
CFMSACC:   .long  0x00000000
CFMDACC:   .long  0x00000000
CFMSEC:    .long  0x00000000

```

2. 芯片相关文件

1) 映像寄存器定义文件mcf52233.h

mcf52233.h 中定义了编程时需要访问的外设寄存器，该文件不修改。比如，在 mcf52233.h 中定义 TC 口的数据寄存器的定义形式如下：

```
#define MCF_GPIO_PORTTC      (*(vuint8 *)(&__IPSBAR[0x100008]))
```

这个定义看起来很复杂，其实也可以将它分解成几个很简单的部分来看：

(1) vuint8 是在 Type.h 文件中定义为 volatile unsigned char 的，就是说(vuint8 *)和 (volatile unsigned char *) 是相同的意思。(vuint8 *)是 C 语言中的强制类型转换，它的作

用是把 (&__IPSBAR[0x100008]) 转换为一个地址指针, 其中 volatile 并不是必要的, 他只是告诉编译器, 这个值与外界环境有关, 不要对其优化, volatile 的具体用法在后面的 Type.h 中介绍。

(2) 接下来在外面加一个*号, 就表示 __IPSBAR[0x100008] 的内容。经过这个宏定义之后, MCF_GPIO_PORTTC 就可以作为一个普通的变量来操作, 所有出现 MCF_GPIO_PORTTC 的地方编译的时候都会被替换成 (* (vuint8 *) (&__IPSBAR[0x100008]))

(3) 外面一层括号是为了保证里面的操作不会因为运算符优先级或其他不可预测的原因被改变而无法得到预期的结果。

这种定义方法适合所有的 C 编译器, 可移植性好, 但 MCF_GPIO_PORTTC 并不是一个真正的变量, 只是一个宏名, 当调试一个程序的时候, 无法在调试窗口观察它的值。

2) 系统初始化文件 sysinit.h 与 sysinit.c

系统初始化操作是由 sysinit.h 与 sysinit.c 来实现的, 具体内容请参见随书网上光盘。

3.4.3 主程序、中断程序及其他文件

1. 总头文件 Includes.h 和主程序文件 main.c

Includes.h 文件包含主函数 (main) 文件中用到的头文件、外部函数或变量引用、有关常量和全局变量定义以及内部函数声明。而 main.c 文件是工程任务的核心文件, 里面包含了一个主循环, 对具体事务过程的操作几乎都是添加在该主循环中。

2. 中断文件 isr.h 与 isr.c

中断头文件 isr.h 内容如下:

```
//-----*
// 文件名:  isr.h  (中断控制头文件)           *
// 说 明:  中断控制头文件                     *
//-----*

#ifndef ISR_H          //防止重复定义
#define ISR_H
    //1. 头文件
    #include "mcf52233.h"  //CodeFire 52233 MCU映像寄存器名定义
    //2. 中断操作语句
    //2.1 总中断
    #define DisableInterrupts()  asm(move.w  #0x2700, sr)    //关总中断
    #define EnableInterrupts()   asm(move.w  #0x2000, sr)    //开总中断
    //2.2 添加模块中断

#endif
```

中断定义文件 isr.c 的格式:

```

//-----*
//文 件 名: isr.c (中断处理函数文件) *
//说 明: 中断处理函数 *
//本文件包含: *
//      (1)asm_exception_handler: 中断处理程序 *
//      (2)irq_handler: 中断处理程序 *
//硬件连接: 无 *
//-----*
#include "isr.h"
//asm_exception_handler中断处理程序
__interrupt__
void asm_exception_handler(void)
{
    DisableInterrupts(); //关总中断
    //添加用户代码
    EnableInterrupts(); //开总中断
}
//irq_handler中断处理程序
__interrupt__
void irq_handler(void)
{
    DisableInterrupts(); //关总中断
    //添加用户代码
    EnableInterrupts(); //开总中断
}

```

3. 芯片无关文件

1) 数据类型别名定义文件Type.h

在 C 工程中有一个“Type.h”文件，它给 C 语言中的类型名起别名，这样使程序中的类型名更清晰，同时，也便于程序移植到不同的 MCU 中。在多个程序文件中都有可能用到类型别名定义，为了防止在一个文件中多次包含“Type.h”。因此，在“Type.h”中需要加入条件编译语句。

```

//-----*
// 文件名: type.h (变量类型别名文件) *
// 说 明: 定义变量类型的别名，目的: *
//      (1)缩短变量类型书写长度; (2)方便程序移植。可以根据需要自行添加。 *
//-----*
#ifndef TYPE_H //防止重复定义
#define TYPE_H
typedef char      int8;    // 8 位有符号数
typedef short int int16;   // 16 位有符号数
typedef int       int32;   // 32 位有符号数
typedef unsigned char uint8; // 8 位无符号数
typedef unsigned short int uint16; // 16 位无符号数
typedef unsigned long int uint32; // 32 位无符号数

typedef float      fp32;    //单精度浮点数

```

```

typedef double      fp64;    //双精度浮点数
//不优化数据类型
typedef volatile int8    vint8;  // 8 位无符号数
typedef volatile int16   vint16; // 16 位无符号数
typedef volatile int32   vint32; // 32 位无符号数
typedef volatile uint8    vuint8; // 8 位有符号数
typedef volatile uint16   vuint16; // 16 位有符号数
typedef volatile uint32   vuint32; // 32 位有符号数
typedef volatile float    vfp32;  //单精度浮点数
typedef volatile double   vfp64;  //双精度浮点数
#endif

```

在定义不优化数据类型时, 使用关键字 **volatile**。**volatile** 关键字用于提醒编译器, 它后面所定义的变量不能随意进行优化, 因此, 编译器会安排该变量使用系统存储区的具体地址单元, 编译后的程序每次需要存储或读取该变量时, 都会直接访问该变量的地址。若没有 **volatile** 关键字, 则编译器可能会暂时使用 CPU 寄存器来存储, 以优化存储和读取, 这样, CPU 寄存器和变量地址的内容很可能会出现不一致现象。

下面举例说明。在开发中, 常需等待某个事件的触发, 所以会写出这样的程序:

```

unsigned char flag;
void test()
{
    do1();
    while(flag==0);
    do2();
}

```

这段程序等待内存变量 **flag** 的值变为 1 之后才运行 **do2()**。变量 **flag** 的值由别的程序更改, 这个程序可能是某个硬件中断服务程序。例如: 如果某个按钮按下的话, 就会产生中断, 在按键中断程序中修改 **flag** 为 1, 这样上面的程序就能够得以继续运行。但是, 编译器并不知道 **flag** 的值会被别的程序修改, 因此它在进行优化的时候, 可能会把 **flag** 的值先读入某个寄存器, 然后等待那个寄存器变为 1。如果不幸进行了这样的优化, 那么 **while** 循环就变成了死循环, 因为寄存器的内容不可能被中断服务程序修改。为了让程序每次都读取真正 **flag** 变量的值, 就需要定义为如下形式:

```
volatile unsigned char flag;
```

2) 通用函数文件 GeneralFun.h 和 GeneralFun.c

在 C 工程中, “GeneralFun.h” 文件中定义了经常使用的一些函数和宏, 比如延时。另外由于需要频繁地操作某个寄存器某位, 比如置位、清零等, 所以我们也设置了对寄存器位操作的宏定义。用户可以修改该文件添加一些经常使用的函数和宏。GeneralFun.c 则用于定义具体的通用函数。

```

#define BSET(bit, Register) ((Register) |= (1<<(bit))) //设置寄存器中某一位为1
#define BCLR(bit, Register) ((Register) &= ~(1<<(bit))) //设置寄存器中某一位为0
#define BGET(bit, Register) (((Register)>> (bit)) & 1) //得到寄存器中某一位状态

```

4. 工程说明文件

该文件用于记录或给出工程实例的说明信息。

3.4.4 链接文件及机器码文件

1. 链接文件linker.lcf

C 工程中的链接文件是 `linker.lcf`，该文件提供了工程在链接时所需要的参数信息。一般情况下，该文件不需要修改。链接文件的源代码如下：

```

/*链接地址的定义*/
MEMORY
{
    flash    (RX)    : ORIGIN = 0x00000000, LENGTH = 0x00040000
    sram     (RWX)    : ORIGIN = 0x20000000, LENGTH = 0x00007C00
    ipsbar   (RWX)    : ORIGIN = 0x40000000, LENGTH = 0x0
}

SECTIONS
{
    .ipsbar : {} > ipsbar
    .flash :
    {
        vectors.s (.text)
        . = ALIGN(0x10);
        *(.text)
        . = ALIGN(0x10);
        *(.rodata)
        __DATA_ROM = .;
    } > flash

    ... (部分内容省略)

    __FLASH = ADDR(.flash);
    __FLASH_SIZE = 0x00040000;
    __SRAM = 0x20000000;
    __SRAM_SIZE = 0x00008000;

    ... (部分内容省略)

    __heap_addr = __HEAP_START;
    __heap_size = __HEAP_END - __HEAP_START;
    __SP_INIT = __SP_INIT;
    __STACK_SIZE = __SP_INIT - __SP_END;
}

```

上面的代码首先用 **MEMORY** 命令定义了一些存储空间，**Flash** 指定用户程序可存放的区域，**Flash** 段是从 `0x0` 开始，长度是 `0x40000`，共 256K 的存储空间。该段的数据是可

读的和可执行的。

Sram 段是从 0x20000000 开始，长度是 0x00007C00，为 32K 的存储空间。全局变量和静态变量存放在该段。该段的数据是可读写和可执行的。

ipsbar 段从 0x40000000 开始，定义了内部外设系统基址寄存器的地址。该段的数据是可读写和可执行的。

SECTIONS 命令为程序的代码和数据指定存放地址。SECTIONS 命令是脚本文件中最基本的命令，其功能非常强大。

样例代码中 SECTIONS{} 所给出的便是与链接序列相关的信息。以上面链接文件中黑色底纹的代码为例，.flash: {*} > flash 指定了“{}”中的内容是用来定义源代码文件在 Flash 程序存储空间（在 MEMORY{} 中已经把 Flash 放在了首地址为 0x00000000、长度为 0x00040000、大小为 256K 的逻辑空间）中的安放顺序的。下面进行代码解读：

异常向量表文件 vectors.s 优先放在低地址，(.text) 则指明该文件为文本文件。

通配符“*”用来代替任意文件，所以*(.text) 的意思就是其他所有的文本文件放在紧随异常向量表文件之后的地址空间。类似的，*(.rodata) 的意思则是所有.rodata 格式的文件安排紧随其后的位置。ALIGN (0x10) 用于指定对齐方式，0x10: 对齐单元大小为 16B，意思是在链接定址时，该文件的首地址必然定位到上一链接文件之后的第一个 16B 的整数倍的地址处。

如果没有链接器命令文件或文件里没有顺序操作命令，链接器就按照输入文件的出现顺序来分配段地址，文件内的各段也按照段的属性排列（代码段、可固化数据、可写数据以及初始化数据），所有段中第一个段的首地址为 0x00000000，其他段逐个紧随。

2. 目标文件.s19文件

在编译连接过程中，CodeWarrior 会生成三个文件分别是 elf、s19、.elf.xMAP。其中 S19 格式的文件写入到 Flash 中。

ELF (Executable and Linking Format) 是可执行连接格式。可执行连接格式是 UNIX 系统实验室 (USL) 作为应用程序二进制接口 (Application Binary Interface, ABI) 而开发和发布的。工具接口标准委员会 (TIS) 选择了正在发展中的 ELF 标准作为工作在 32 位 INTEL 体系上不同操作系统之间可移植的二进制文件格式。elf.xMAP 文件中记录了存储器的分配情况。这两种格式本书不涉及。下面详细介绍目标代码文件 S19 文件。

源文件经编译、连接后可获得目标码文件 (.S19 文件)，它是 Freescale MCU 的机器码文件，通过写入程序可将机器码下载到目标 MCU 内的 Flash 存储器，我们可以通过一个叫做 CF Flasher 的程序来读出写入到 Flash 的 S19 的内容，观察实际写入到 Flash 的内容是否和 S19 的内容相同。该程序也可以擦写 Flash。该程序的安装文件简要使用说明在“…\第 04 章 阅读材料”目录下。

目标代码文件是以 S 记录格式表示的机器码文件。S 记录格式是 Freescale 公司的十六进制目标代码文件，它将目标程序和数据以 ASCII 码格式表示，可直接显示和打印。目标文件由若干行 S 记录构成，每行 S 记录可以用 CR/LF/NUL 结尾。一行 S 记录由下列五部分组成，分别说明如下。

(1) 类型

表示 S 记录的类型。有 8 种记录类型 S0、S1、S2、S3、S5、S7、S8、S9。这是为了满足不同的编码、解码及传送方式的需求，表 3-1 给出了 S 记录的格式。

表 3-1 S 记录格式

类型	记录长度	地址	编码/数据	校验和
2字节	2字节	2、3或4字节	0~n字节	1字节

S0—该记录包含 S19 文件的文件名信息。

S1—该记录包含代码/数据以及两个字节存储其代码/数据的存储器首地址。

S2—该记录包含要写到 Flash 的扩展地址处的代码/数据以及三个字节存储其代码/数据的存储器首地址。

S3—该记录包含要写到 Flash 的扩展地址处的代码/数据以及四个字节存储其代码/数据的存储器首地址。

S7—S3 记录的结束记录。

S8—S2 记录的结束记录。

S9—S1 记录的结束记录。

每个 S 记录块都使用唯一的终止记录。

(2) 记录长度

表示该记录行中字符对的数目，不包括类型和记录长度。

(3) 地址

它可以是 2 个字节、3 个字节或 4 个字节，取决于记录类型。S1 记录、S9 记录均是 2 个字节，S2 记录、S8 记录是 3 个字节，S3 记录、S7 记录是 4 个字节。它表示其后的代码/数据部分将要装入的存储器起始地址。

(4) 代码/数据

就是实际的目标代码或数据，这一部分将被下载到目标芯片的存储器并运行。其字节数是由“记录长度”域的实际数值减去地址长度和校验码长度的值而得到的。

(5) 校验和

为 1 个字节，它是“记录长度”、“地址”、“代码/数据”三个部分所有字节之和的反码的低 8 位，用于校验。

下面是 01_Light 工程中的 Light.elf.S19 的部分内容。

```
S0030000FC
...
S321000006AC4A8067E44E5E4E754E56000043F94000000041F9200000002008028084
...
S70500000400F6
```

第一行为 S0 记录，表示文件名信息。S0 之后的 03 是十六进制数（十进制数 3），表示后面有 3 个字节的数据；随后的"0000"是 2 字节地址，"0000"表示本行信息不是程序/数据，不需要装入存储空间；最后的 FC 是本记录的校验和。其中 S321000006AC4A8067E44E5E4E754E56000043F94000000041F9200000002008028084 的前

两个符号 S3 表示这一行是 S3 记录，其后的“21”是十六进制数（十进制数的 33），表示在此行其后有 33 个字节的数据，包括 4 个字节的地址 000006AC、28 个字节的代码/数据，最后字节 84 为校验和。该行记录所表示的实际代码/数据 4A8067E44E5E4E754E56000043F94000000041F92000000020080280 将被装入起始地址为 0x000006AC 的 MCU 存储器中。

最后一行是 S7 记录，S7 之后的 05 是十六进制 0x05，表示其后有 5 个字节的/数据。00000400 为 4 个字节的地址，F6 是校验和。

3.5 第一个应用实例：控制小灯闪烁

本书用 MCF52233 控制多个发光二极管指示灯的例子开始我们的程序之旅，程序中使用了 GPIO 构件来编写指示灯程序。指示灯是最简单不过的硬件对象了，当灯两端引脚上有足够高的正向压降时，它就会发光。在本书的工程实例中，灯的正端引脚接 MCF52233 的普通 I/O 口，负端引脚过电阻接地。当在 I/O 引脚上输出高或低电平时，指示灯就会亮或暗。如图 3-7 所示，为使用 ColdFire 52233 的普通 I/O 口控制指示灯闪烁的 C 语言工程实例的文件组织，其工程文件在光盘“...\\02-MCF52233Program\\01_Light”文件夹内。

为了复用代码，提高编程效率和增强代码的可移植性，以下控制小灯闪烁的实例编程中使用了构件化的思想。关于构件的详细内容在“第 16 章 基于硬件构件的嵌入式系统开发方法”阐述。

3.5.1 GPIO 构件

GPIO 引脚可以被定义成输入、输出两种情况。若是输入，程序需要获得引脚的状态（逻辑 1 或 0）。若是输出，程序可以设置引脚状态（逻辑 1 或 0）。MCU 的 GPIO 引脚分为许多端口（Port），每个口有若干引脚。为了实现对所有 GPIO 引脚统一编程，设计了 GPIO 构件（由 GPIO.h、GPIO.c 两个文件组成）。这样，要使用 GPIO 构件，只需要将这两个文件加入到所建工程中，方便了对 GPIO 的编程操作。实际上，若只是使用构件，只需看头文件中的相关函数说明。

1. GPIO 构件的头文件 GPIO.h

```
//-----*
// 文件名:  GPIO.h (GPIO构件头文件)                *
// 说 明:  包含了GeneralFun.h文件                    *
//-----*

#ifndef GPIO_H          //防止重复定义
#define GPIO_H
    //1. 头文件
    #include "mcf52233.h"    //CodeFire 52233 MCU映像寄存器名定义
    #include "sysinit.h"    //配置头文件
    #include "GeneralFun.h" //包含通用函数头文件
    //2. 功能接口(函数声明)
```

```

//-----*
//函数名: GPIO_Init                                *
//功 能: 初始化GPIO                                *
//参 数: port:端口名;                              *
//       pin:指定端口引脚;                          *
//       direction:引脚方向, 0=输入, 1=输出;         *
//       state:初始状态, 0=低电平, 1=高电平;        *
//返 回: 无                                          *
//-----*
void GPIO_Init(uint8 port,uint8 pin,uint8 direction,uint8 state);

//-----*
//函数名: GPIO_Get                                *
//功 能: 获得引脚状态                              *
//参 数: port: 端口名;                              *
//       pin: 指定端口引脚                          *
//返 回: state:得到引脚状态                        *
//-----*
uint8 GPIO_Get(uint8 port,uint8 pin);

//-----*
//函数名: GPIO_Set                                *
//功 能: 设置引脚状态                              *
//参 数: port:端口名;                              *
//       pin:指定端口引脚;                          *
//       state: 状态, 0=低电平, 1=高电平            *
//返 回: 无                                          *
//-----*
void GPIO_Set(uint8 port,uint8 pin,uint8 state);

//3. 宏定义
//ColdFire端口名与偏移地址的对应关系
//每个端口的引脚并不是全部使用, 请参考数据手册。
//80引脚的MCF52233可用引脚
#define PORT_NQ 0 //1. 4. 7
#define PORT_AN 2 //0~7
#define PORT_AS 3 //0. 1. 2. 3
#define PORT_QS 4 //0. 1. 2. 3
#define PORT_TA 6 //0. 1. 2. 3
#define PORT_TC 7 //0. 1. 2. 3
#define PORT_TD 8 //
#define PORT_UA 9 //0. 1. 2. 3
#define PORT_UB 10 //0. 1. 2. 3
#define PORT_UC 11 //
#define PORT_DD 12 //
#define PORT_LD 13 //0~4
#define PORT_GP 14 //3
//根据GPIO端口的偏移地址得到各个寄存器的地址
//端口寄存器
#define MCF_GPIO_PORT(x) (*(vuint8 *)(&__IPSBAR[0x100008+(x)]))

```

```

//方向寄存器
#define MCF_GPIO_DDR(x) (*(vuint8 *)(&__IPSBAR[0x100020+(x)]))
//数据/置位寄存器
#define MCF_GPIO_SET(x) (*(vuint8 *)(&__IPSBAR[0x100038+(x)]))
//复用寄存器
//MCF_GPIO_PNPARG的位定义和宏定义
#define MCF_GPIO_PNPARG(x) ((0x0003)<<(2*(x)))
//MCF_GPIO_PANPAR的位定义和宏定义
#define MCF_GPIO_PANPAR(x) ((0x1)<<(x))
//MCF_GPIO_PASPAR的位定义和宏定义
#define MCF_GPIO_PASPAR(x) ((0x03)<<(2*(x)))
//MCF_GPIO_PQSPARG的位定义和宏定义
#define MCF_GPIO_PQSPARG(x) ((0x0003)<<(2*(x)))
//MCF_GPIO_PTAPARG的位定义和宏定义
#define MCF_GPIO_PTAPARG(x) ((0x03)<<(2*(x)))
//MCF_GPIO_PTCPARG的位定义和宏定义
#define MCF_GPIO_PTCPARG(x) ((0x03)<<(2*(x)))
//MCF_GPIO_PTDPAR的位定义和宏定义
#define MCF_GPIO_PTDPAR(x) ((0x01)<<(x))
//MCF_GPIO_PUAPARG的位定义和宏定义
#define MCF_GPIO_PUAPARG(x) ((0x03)<<(2*(x)))
//MCF_GPIO_PUBPAR的位定义和宏定义
#define MCF_GPIO_PUBPAR(x) ((0x03)<<(2*(x)))
//MCF_GPIO_PUCPAR的位定义和宏定义
#define MCF_GPIO_PUCPAR(x) ((0x01)<<(x))
//MCF_GPIO_PDDPAR的位定义和宏定义
#define MCF_GPIO_PDDPAR(x) ((0x01)<<(x))
//MCF_GPIO_PLDPARG的位定义和宏定义
#define MCF_GPIO_PLDPARG(x) ((0x01)<<(x))
//MCF_GPIO_PGPPARG的位定义和宏定义
#define MCF_GPIO_PGPPARG(x) ((0x01)<<(x))
#endif

```

2. GPIO构件的程序文件GPIO.c

```

//-----*
// 文件名: GPIO.c (GPIO驱动程序文件) *
//-----*
#include "GPIO.h" //包含GPIO头文件
//-----*
//函数名: GPIO_Init *
//功 能: 初始化GPIO *
//参 数: port:端口名; *
//      pin:指定端口引脚; *
//      direction:引脚方向,0=输入,1=输出; *
//      state:初始状态,0=低电平,1=高电平; *
//返 回: 无 *
//-----*

```

```
void GPIO_Init(uint8 port, uint8 pin, uint8 direction, uint8 state)
{
    //1. 配置引脚功能为GPIO
    switch(port)
    {
        case PORT_NQ:
            MCF_GPIO_PNQPARG &= ~MCF_GPIO_PNQPARG(pin);
            break;
        case PORT_AN:
            MCF_GPIO_PANPARG &= ~MCF_GPIO_PANPARG(pin);
            break;
        case PORT_AS:
            MCF_GPIO_PASPARG &= ~MCF_GPIO_PASPARG(pin);
            break;
            MCF_GPIO_PQSPARG &= ~MCF_GPIO_PQSPARG(pin);
            break;
            MCF_GPIO_PTAPARG &= ~MCF_GPIO_PTAPARG(pin);
            break;
            MCF_GPIO_PTCPARG &= ~MCF_GPIO_PTCPARG(pin);
            break;
            MCF_GPIO_PTDPAR &= ~MCF_GPIO_PTDPAR(pin);
            break;
            MCF_GPIO_PUAPARG &= ~MCF_GPIO_PUAPARG(pin);
            break;
            MCF_GPIO_PUBPAR &= ~MCF_GPIO_PUBPAR(pin);
            break;
            MCF_GPIO_PUCPAR &= ~MCF_GPIO_PUCPAR(pin);
            break;
            MCF_GPIO_PDDPAR &= ~MCF_GPIO_PDDPAR(pin);
            break;
            MCF_GPIO_PLDPARG &= ~MCF_GPIO_PLDPARG(pin);
            break;
            MCF_GPIO_PGPPARG &= ~MCF_GPIO_PGPPARG(pin);
            break;
        default:
            break;
    }
    //2. 设置引脚方向
    if(direction == 1)
    {
        BSET(pin, MCF_GPIO_DDREG(port));    //定义引脚为输出(相应位为1)
    }
    else
    {
        BCLR(pin, MCF_GPIO_DDREG(port));    //定义引脚为输入(相应位为0)
    }
    //3. 设置引脚状态
    if(state == 1)
    {
```

```

        BSET(pin, MCF_GPIO_PORT(port));    //输出高电平(1)
    }
    else
    {
        BCLR(pin, MCF_GPIO_PORT(port));    //输出低电平(0)
    }
}

//-----*
//函数名: GPIO_Get                                *
//功 能: 获得引脚状态                            *
//参 数: port: 端口名;                            *
//      pin: 指定端口引脚                        *
//返 回: state:得到引脚状态                      *
//-----*
uint8 GPIO_Get(uint8 port, uint8 pin)
{
    uint8 reval=0;
    reval = BGET(pin, MCF_GPIO_SET(port)); //得到引脚的状态
    return reval;
}

//-----*
//函数名: GPIO_Set                                *
//功 能: 设置引脚状态                            *
//参 数: port:端口名;                            *
//      pin:指定端口引脚;                        *
//      state: 状态, 0=低电平, 1=高电平          *
//返 回: 无                                        *
//-----*
void GPIO_Set(uint8 port, uint8 pin, uint8 state)
{
    if(state == 1)
    {
        BSET(pin, MCF_GPIO_PORT(port));    //输出高电平(1)
    }
    else
    {
        BCLR(pin, MCF_GPIO_PORT(port));    //输出低电平(0)
    }
}

```

3.5.2 Light构件

控制指示灯的亮或暗，通过调用 GPIO 构件完成。设有三盏灯，功能分别为“运行指示灯”、“故障指示灯”、“通信指示灯”。现在要为这三盏灯分别起个名字，分别叫 Light_Run、Light_Error、Light_Link。它们所接在的 MCU 的 GPIO 端口的名字分别叫做：Light_Run

_PORT、Light_Error_PORT、Light_Link_PORT。这样它们具体接在 MCU 的哪个端口，哪个引脚，只要在 Light.h 中给出具体宏定义就可以了。

指示灯的亮、暗的逻辑值与实际状态的对应取决于实际电路的接法（见 3.1 节），为了程序的可复用性，因此，使用两条宏定义语句来描述这种对应关系。

1. Light构件的头文件Light.h

```
//-----*
// 文件名: Light.h (指示灯驱动程序头文件) *
//-----*

#ifndef Light_H          //防止重复定义
#define Light_H

    //1 头文件
    #include "GPIO.h"     //包含GPIO头文件
    //2 灯控制
    //2.1 灯控制引脚定义
    //    Light_Run灯
    #define Light_Run_PORT PORT_TC //灯使用的端口
    #define Light_Run 1           //灯使用的引脚
    //    Light_Error灯
    #define Light_Error_PORT PORT_TC //灯使用的端口
    #define Light_Error 2           //灯使用的引脚
    //    Light_Link灯
    #define Light_Link_PORT PORT_LD //灯使用的端口
    #define Light_Link 2           //灯使用的引脚
    //2.2 灯状态宏定义
    #define Light_ON 1           //灯暗(对应高电平)
    #define Light_OFF 0         //灯亮(对应低电平)

    //2.3 灯控制相关函数声明
    //-----*
    //函数名: Light_Init *
    //功 能: 初始化指示灯状态 *
    //参 数: port:端口名; *
    //        name:指定端口引脚号; *
    //        state:初始状态, 1=高电平, 0=低电平 *
    //返 回: 无 *
    //-----*

    void Light_Init(uint8 port, uint8 name, uint8 state);
    //-----*
    //函数名: Light_Control *
    //功 能: 控制灯的亮和暗 *
    //参 数: port:端口名; *
    //        name:指定端口引脚号; *
    //        state:状态, 1=高电平, 0=低电平 *
    //返 回: 无 *
    //-----*

    void Light_Control(uint8 port, uint8 name, uint8 state);
```

```
#endif
```

2. Light构件的程序文件Light.c

```
//-----*
//文件名: Light.c (小灯驱动函数文件) *
//-----*
#include "Light.h" //(指示灯驱动程序头文件)

//-----*
//函数名: Light_Init *
//功 能: 初始化指示灯状态 *
//参 数: port:端口名; *
//      name:指定端口引脚号; *
//      state:初始状态, 1=高电平, 0=低电平 *
//返 回: 无 *
//-----*
void Light_Init(uint8 port, uint8 name, uint8 state)
{
    GPIO_Init(port, name, 1, state); //初始化指示灯
}

//-----*
//函数名: Light_Control *
//功 能: 控制灯的亮和暗 *
//参 数: port:端口名; *
//      name:指定端口引脚号; *
//      state:状态, 1=高电平, 0=低电平 *
//返 回: 无 *
//-----*
void Light_Control(uint8 port, uint8 name, uint8 state)
{
    GPIO_Set(port, name, state); //控制引脚状态
}
```

3.5.3 Light测试工程主程序

在 includes.h 文件中需要包含 Light.h, 这样在该工程中就可以调用 Light 构件的接口函数。首先调用 Light_Init 函数, 初始化所需的每一盏指示灯。注意, 初始化时, 要让每一盏灯初始状态为“暗”。随后, 通过 Light_Control 函数控制指示灯亮、暗。在指示灯亮暗之间增加适当的延时后, 就能够在程序运行时, 较明显的看到指示灯闪烁的现象。

```
//-----*
//工 程 名: Light *
//硬件连接: 见工程说明 *
//程序描述: 用I/O口控制小灯闪烁 *
//目 的: 第一个Freescale ColdFire系列MCU C语言程序框架 *
//-----*
```

```

// 说    明：提供Freescale MCU的编程框架，供教学入门使用                *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*

#include "includes.h"    //包含总头文件

int main(void)
{
    //1 关总中断
    DisableInterrupts();
    //2 三盏指示灯初始化
    Light_Init(Light_Run_PORT, Light_Run, Light_OFF);
    Light_Init(Light_Error_PORT, Light_Error, Light_OFF);
    Light_Init(Light_Link_PORT, Light_Link, Light_OFF);

    //3 主循环
    while (1)
    {
        //Light_Run灯先亮后暗
        Light_Control(Light_Run_PORT, Light_Run, Light_ON);
        Delay(100);                //延时适当时间
        Light_Control(Light_Run_PORT, Light_Run, Light_OFF);
        Delay(100);
        //Light_Error灯先亮后暗
        Light_Control(Light_Error_PORT, Light_Error, Light_ON);
        Delay(100);
        Light_Control(Light_Error_PORT, Light_Error, Light_OFF);
        Delay(100);
        //Light_Link灯先亮后暗
        Light_Control(Light_Link_PORT, Light_Link, Light_ON);
        Delay(100);
        Light_Control(Light_Link_PORT, Light_Link, Light_OFF);
        Delay(100);
    }
    return 0;
}

```

3.6 理解第一个C工程的执行过程

当 MCF52233 芯片上电复位或热复位后，系统程序的执行流程如下：从复位向量处取出程序执行的首地址，跳转并按该地址执行；执行 setup.s 文件中的 _asm_startmeup 函数，进行系统的初始化，并最终跳转到 main 主函数入口继续执行；在系统带电的状态下，硬件中断机制始终开启，并实时地“监听”内外环境而恰当地激发特定的事务处理过程。下面具体来看各个过程是怎样工作的。

1. 系统上电

系统在加电过后，芯片内的硬件机制会产生加电复位中断，这时系统到异常向量表中

查找复位向量地址，并转向这个地址继续执行。在本书所有工程样例，到 `vectors.s` 文件中都可以找到异常向量表，在该表开始的前两行分别是堆栈初始化指针和复位中断向量地址：

```
INITSP: .long  __SP_INIT      // Initial SP
INITPC: .long  start         // Initial PC
```

`__SP_INIT` 是在 `linker.lcf` 链接文件中定义的，它给出了系统在复位过程中需要给堆栈初始化写入的值，`start` 是一个汇编语言的标号，它引领了复位后程序执行的第一条代码。查看 `.S19` 代码文件，可以看到前两行为：

```
S0030000FC
S321000000002000800000000418000007D0000007D0000007D0000007D0000007D00EF
```

`__SP_INIT` 编译后的值为 `20008000`（该值为片内 RAM 地址的最后一个地址，也就是说堆栈 SP 指向 RAM 的最高地址，堆栈使用时是向低地址方向增长），放在地址 `0x00000000` 处，`start` 编译后的值为 `00000418`，放在地址 `0x00000004` 处。

系统复位后，会用 `__SP_INIT` 初始化 SP，并用 `start` 初始化 PC，之后程序将从标号“`start`”所代表的地址处开始执行，“`start`”标号的定义在 `vectors.s` 文件的最后，它的代码如下：

```
start:
move.w  #0x2700,sr
jmp     _asm_startmeup
```

`start` 编译后的值为 `00000418`，在 `.S19` 代码文件中可找到该地址所对应的行：

```
S3210000040C0000000000000000000000000000046FC27004EF90000080000000000000016
46FC2700 就是 move.w  #0x2700,sr 指令所对应的机器码。
```

`4EF900000800` 就是 `jmp _asm_startmeup` 指令所对应的机器码。`00000800` 为 `_asm_startmeup` 所指向的第一条指令的地址。

`move.w #0x2700,sr` 指令是系统启动后程序执行的第一条代码。之后程序就跳转到标号“`_asm_startmeup`”处执行。“`_asm_startmeup`”的定义在启动文件 `setup.s` 中，它所代表的地址为 `00000800`，该地址在 `.S19` 代码文件中的行为：

```
S321000007FC588F4E732C002E01203C200000000680000000214E7B0C05203C40003F
```

2. 执行 `setup.s` 文件中的 `_asm_startmeup` 函数

执行 `setup.s` 文件中的 `_asm_startmeup` 函数，并转向 `main` 函数。`setup.s` 文件是系统启动文件，它的主体代码在前面 3.4.1 中已经给出。这段代码主要作用是设置存储映射的基本参数，以安排 SRAM、Flash 和 IPSBAR 在 4G 的虚拟寻址空间中的位置。具体包括以下过程：备份芯片信息；初始化 RAMBAR1，定位 SRAM 并使其生效；定位堆栈指针；初始化 IPSBAR；初始化 FLASHBAR；系统模块初始化并跳转到 `main` 主程序。

值得注意的是程序一进入 `_asm_startmeup` 函数就先备份 D0 和 D1 寄存器的内容到 D6 和 D7。这是因为 D0 和 D1 在上电初始就保存有芯片的版本信息，D0 和 D1 在后面的程序中必然会用到，为了以后还能有机会访问芯片的版本信息，备份是个不错的选择。

备份芯片信息的两条指令是 `_asm_startmeup` 函数的开始指令，在地址 `00000800` 处

存放的 2C002E01，正是以上两条指令的机器码。

若代码执行顺利，程序会执行到如下指令：

```
jsr  mcf52233_init  //系统初始化
jsr  main           //转主程序
```

在系统初始化过程中编程者可以加入任何模块的初始化函数。

程序在开始执行 `jsr mcf52233_init` 之前，会把当前指令的下条指令地址压入堆栈，在返回时，该地址将从堆栈中取出并放到指令指针（PC）中，那么就可以执行下条指令 `jsr main` 了，它也是一条跳转指令，目的地址为 `main` 函数所在的地址。

当执行到“`jsr main`”时，程序就会转向用户自定义的 `main` 主程序中执行。以上所说的，都是正常的事务执行过程。当然，从根本上讲，这个过程也是由系统复位中断来触发。同样，芯片内部的硬件中断机制也能为其他中断型事件提供事务处理的机会。

当开发者在系统初始化过程中加入了开启某些中断响应的命令时，硬件中断机制将会实时检测芯片内外的状况，以决定是否激发对应的中断处理例程，而不需要主程序的干预。这样的话，整个程序的执行可被视为具有 2 条主线，一方面 `main` 主程进行着正常的事务操作，另一方面，硬件中断机制会独自的实时“监听”内外环境而恰当的激发特定的事务处理过程。

3. 中断程序的执行

当某个中断发生后，MCU 将转到异常向量表文件 `vectors.c` 所指定的中断入口地址处开始执行中断服务程序（Interrupt Service Routine，ISR）。在这个过程中，系统必然会保存“上下文”（CPU 寄存器的内容），在中断处理结束前，必须恢复该“上下文”，以便继续执行原来的程序。中断的执行实际上是在抢夺主程序的执行时间。4.6 节将具体介绍了在 CW 工程中添加中断的方法。

第4章 异步串行通信与直接存储器访问DMA

目前几乎所有的台式电脑都带有 9 芯的异步串行通信口，简称串行口或 COM 口。由于历史的原因，通常所说的串行通信就是指异步串行通信。USB、以太网等也用串行方式通信，但与这里所说的异步串行通信物理机制不同。

有的台式电脑带有两个串行口：COM1、COM2 口，部分笔记本电脑也带有串行口。随着 USB 接口的普及，串行口的地位逐渐降低，但是作为设备间简便的通信方式，在相当长的时间内，串行口还不会消失，在市场上也可很容易的购买到 USB 到串行口的转接器。因为简单且常用的串行通信只需要三根线（发送线、接收线和地线），所以串行通信仍然是 MCU 与外界通信的简便方式之一。

实现异步串行通信功能的模块在一部分 MCU 中被称为通用异步收发器（Universal Asynchronous Receiver/Transmitters, UART），在另一些 MCU 中被称为串行通信接口（Serial Communication Interface, SCI）。串行通信接口可以将终端或个人计算机连接到 MCU，也可将几个分散的 MCU 连接成通信网络。

本章主要介绍 MCF52233 的 UART 模块和直接存储器访问 DMA 模块的工作原理以及编程实例，这些编程实例都使用了基于构件的编程思想，读者在阅读时可以仔细体会，以求得对编程方法更深刻的理解。下文所出现的 UART 和 DMA 字眼，在没有其他说明的情况下，都是特指 MCF52233 的 UART 模块和 DMA 模块。

4.1 异步串行通信的基础知识

本节简要概括了串行通信中常用的基本概念，为学习 MCU 的串行接口编程做准备。对于已经了解这方面知识的读者，可以略读本节。

4.1.1 基本概念

“位”（bit）是单个二进制数字的简称，是可以拥有两种状态的最小二进制值，分别用“0”和“1”表示。在计算机中，通常一个信息单位用 8 位二进制表示，称为一个“字节”（byte）。串行通信的特点是：数据以字节为单位，按位的顺序（例如最高位优先）从一条传输线上发送出去。这里至少涉及到以下几个问题：第一，每个字节之间是如何区分开的？第二，发送一位的持续时间是多少？第三，怎样知道传输是正确的？第四，可以传输多远？这些问题属于串行通信的基本概念。串行通信分为异步通信与同步通信两种方式，本节主要给出异步串行通信的一些常用概念。正确理解这些概念，对串行通信编程是有益的。

1. 异步串行通信的格式

在 MCU 的英文芯片手册上，通常说的异步串行通信采用的是 NRZ 数据格式，英文

全称是：“standard non-return-zero mark/space data format”，可以译为：“标准不归零传号/空号数据格式”。这是一个通信术语，“不归零”的最初含义是：用负电平表示一种二进制值，正电平表示另一种二进制值，不使用零电平。“mark/space”即“传号/空号”分别是表示两种状态的物理名称，逻辑名称记为“1/0”。对学习嵌入式应用的读者而言，只要理解这种格式只有“1”、“0”两种逻辑值就可以了。图 4-1 给出了 8 位数据、无校验情况的传送格式。



图4-1 串行通信数据格式

这种格式的空闲状态为“1”，发送器通过发送一个“0”表示一个字节传输的开始，随后是数据位（在 MCU 中一般是 8 位或 9 位，可以包含校验位）。最后，发送器发送 1 到 2 位的停止位，表示一个字节传送结束。若继续发送下一字节，则重新发送开始位，开始一个新的字节传送。若不发送新的字节，则维持“1”的状态，使发送数据线处于空闲。从开始位到停止位结束的时间间隔称为一帧（frame）。所以，也称这种格式为帧格式。

每发送一个字节，都要发送“开始位”与“停止位”，这是影响异步串行通信传送速度的因素之一。同时因为每发送一个字节，必须首先发送“开始位”，所以称之为“异步”（Asynchronous）通信。

2. 串行通信的波特率

位长（Bit Length），也称为位的持续时间（Bit Duration）。其倒数就是单位时间内传送的位数。人们把每秒内传送的位数叫做波特率（Baud Rate）。波特率的单位是：位/秒，记为 bps。bps 是英文 bit per second 的缩写，习惯上这个缩写不用大写，而用小写。通常情况下，波特率的单位可以省略。

通常使用的波特率有 600、900、1200、1800、2400、4800、9600、19200、38400、57600、115200 等。在包含开始位与停止位的情况下，发送一个字节需 10 位，很容易计算出，在各波特率下，发送 1K 字节所需的时间。显然，这个速度相对于目前许多通信方式而言是慢的，那么，异步串行通信的速度能否提得很高呢？答案是不能。因为随着波特率的提高，位长变小，以致于很容易受到电磁源的干扰，通信就不可靠了。当然，还有通信距离问题，距离小，可以适当提高波特率，但这样毕竟提高的幅度非常有限，达不到大幅度提高的目的。

3. 奇偶校验

在异步串行通信中，如何知道传输是正确的？最常见的方法是增加一个位（奇偶校验位），供错误检测使用。字符奇偶校验检查（Character Parity Checking）称为垂直冗余检查（Vertical Redundancy Checking, VRC），它是为每个字符增加一个额外位使字符中“1”的个数为奇数或偶数。奇数或偶数依据使用的是“奇校验检查”还是“偶校验检查”而定。当使用“奇校验检查”时，如果字符数据位中“1”的数目是偶数，校验位应为“1”，如

果“1”的数目是奇数，校验位应为“0”。当使用“偶校验检查”时，如果字符数据位中“1”的数目是偶数，则校验位应为“0”，如果是奇数则为“1”。

这里列举奇偶校验检查的一个实例，看看 ASCII 字符“R”，其位构成是 1010010。由于字符“R”中有 3 个位为“1”，若使用奇校验检查，则校验位为 0；如果使用偶校验检查，则校验位为 1。

在传输过程中，若有 1 位（或奇数个数据位）发生错误，使用奇偶校验检查，可以知道发生传输错误。若有 2 位（或偶数个数据位）发生错误，使用奇偶校验检查，就不能知道已经发生了传输错误。但是奇偶校验检查方法简单，使用方便，发生 1 位错误的概率远大于 2 位的概率，所以“奇偶校验”这种方法还是最为常用的校验方法。几乎所有 MCU 的串行异步通信接口，都提供这种功能。

4. 串行通信的传输方式

在串行通信中，经常用到“单工”、“双工”、“半双工”等术语。它们是串行通信的不同传输方式。下面简要介绍这些术语的基本含义。

(1) 单工 (Simplex): 数据传送是单向的，一端为发送端，另一端为接收端。这种传输方式中，除了地线之外，只要一根数据线就可以了。有线广播就是单工的。

(2) 全双工 (Full-duplex): 数据传送是双向的，且可以同时接收与发送数据。这种传输方式中，除了地线之外，需要两根数据线，站在任何一端的角度看，一根为发送线，另一根为接收线。一般情况下，MCU 的异步串行通信接口均是全双工的。

(3) 半双工 (Half-duplex): 数据传送也是双向的，但是在这种传输方式中，除地线之外，一般只有一根数据线。任何时刻，只能由一方发送数据，另一方接收数据，不能同时收发。

4.1.2 RS-232C 总线标准

现在回答“可以传输多远”这个问题。MCU 引脚输入/输出一般使用 TTL (Transistor Transistor Logic) 电平，即晶体管-晶体管逻辑电平。而 TTL 电平的“1”和“0”的特征电压分别为 2.4V 和 0.4V (目前使用 3V 供电的 MCU 中，该特征值有所变动)，即大于 2.4V 则识别为“1”，小于 0.4V 则识别为“0”。它适用于板内数据传输。若用 TTL 电平将数据传输到 5m 之外，那么可靠性就很值得考究了。为使信号传输得更远，美国电子工业协会 EIA (Electronic Industry Association) 制订了串行物理接口标准 RS-232C。RS-232C 采用负逻辑，-15V~-3V 为逻辑“1”，+3V~+15V 为逻辑“0”。RS-232C 最大的传输距离是 30m，通信速率一般低于 20Kbps。当然，在实际应用中，也有人用降低通信速率的方法，通过 RS-232 电平，将数据传送到 300m 之外，这是很少见的，且稳定性很不好。

RS-232C 总线标准最初是为远程数据通信制订的，但目前主要用于几米到几十米范围内的近距离通信。有专门的书籍介绍这个标准，但对于一般的读者，不需要掌握 RS-232C 标准的全部内容，只要了解本节介绍的这些基本知识就可以使用 RS-232。目前一般的 PC 机均带有 1 到 2 个串行通信接口，人们也称之为 RS-232 接口，简称“串口”，它主要用于连接具有同样接口的室内设备。早期的标准串行通信接口是 25 芯插头，这是 RS-232C 规

定的标准连接器（其中：2条地线，4条数据线，11条控制线，3条定时信号，其余5条线备用或未定义）。

后来，人们发现在计算机的串行通信中，25芯线中的大部分并不使用，逐渐改为使用9芯串行接口。一段时间内，市场上还有25芯与9芯的转接头，方便了两种不同类型之间的转换。后来，使用25芯串行插头极少见到，25芯与9芯转接头也极少有售。因此，目前几乎所有计算机上的串行口都是9芯接口。图4-2给出了9芯串行接口的排列位置，相应引脚含义见表4-1。

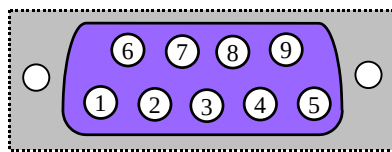


图4-2 9芯串行接口排列

在 RS-232 通信中，常常使用精简的 RS-232 通信，通信时仅使用3根线：RXD（接收线）、TXD（发送线）和 GND（地线）。其他为进行远程传输时接调制解调器之用，有的也可作为硬件握手信号，初学时可以忽略这些信号的含义。

表 4-1 9 芯串行接口引脚含义表

引脚号	功 能	引脚号	功 能
1	接收线信号检测(载波检测DCD)	6	数据通信设备准备就绪(DSR)
2	接收数据线(RXD)	7	请求发送(RTS)
3	发送数据线(TXD)	8	允许发送(CTS)
4	数据终端准备就绪(DTR)	9	振铃指示
5	信号地(SG)		

4.1.3 电平转换电路原理

在 MCU 中，若用 RS-232C 总线进行串行通信，则需外接电路实现电平转换。在发送端，需要用驱动电路将 TTL 电平转换成 RS-232C 电平；在接收端，需要用接收电路将 RS-232C 电平转换为 TTL 电平。电平转换器不仅可以由晶体管分立元件构成，也可以直接使用集成电路。目前使用 MAX232 芯片较多，该芯片使用单一+5V 电源供电实现电平转换。图 4-3 给出了 MAX232 的引脚说明。

引脚含义简要说明如下：

Vcc（16脚）：正电源端，一般接+5V

GND（15脚）：地

VS+（2脚）：VS+=2Vcc-1.5V=8.5V

VS-（6脚）：VS-=-2Vcc-1.5V=-11.5V

C2+、C2-（4、5脚）：一般接 1μF 的电解电容

C1+、C1-（1、3脚）：一般接 1μF 的电解电容

输入输出引脚分两组，基本含义见表 4-2。在实际使用时，若只需要一路串行通信接口，可以使用其中的任何一组。利用 MAX232 将 TTL 电平转换为 RS-232 电平的电路接法稍后结合串行通信接口的外围硬件电路讨论。

焊接到 PCB 板上的 MAX232 芯片检测方法：正常情况下，(1) T1IN=5V，则 T1OUT=-9V；T1IN=0V，则 T1OUT=9V。(2) 将 R1IN 与 T1OUT 相连，令 T1IN=5V，则 R1OUT=5V；

令 $T1IN=0V$ ，则 $R1OUT=0V$ 。

所有型号 MCU 的串行通信接口，都具有发送引脚 TxD 、接收引脚 RxD ，它们是 TTL 电平引脚。要利用这两个引脚与外界实现异步串行通信，还必须将 TTL 电平转为 RS-232 电平，这可利用上节介绍的 MAX232 来完成。本节从通用角度讨论串行通信接口的电路设计、基本编程结构与编程原理，为实际编程做准备。

表 4-2 MAX232 芯片输入输出引脚分类与基本接法

组别	TTL电平引脚	方向	典型接口	232电平引脚	方向	典型接口
1	11	输入	接MCU的TXD	13	输入	接到9芯接口的2脚RXD
	12	输出	接MCU的RXD	14	输出	接到9芯接口的3脚TXD
2	10	输入	接MCU的TXD	8	输入	接到9芯接口的2脚RXD
	9	输出	接MCU的RXD	7	输出	接到9芯接口的3脚TXD

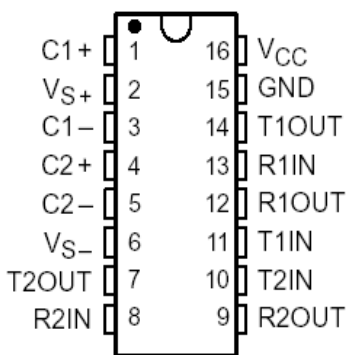


图4-3 MAX232引脚

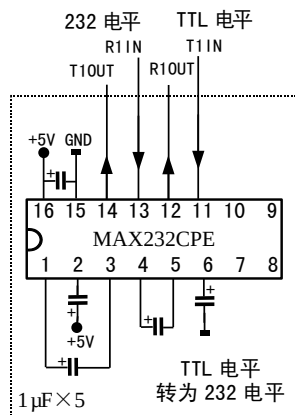


图4-4 串行通信接口电平转换电路

具有串行通信接口的 MCU，一般具有发送引脚（TXD）与接收引脚（RXD），不同公司或不同系列的 MCU，使用的引脚缩写名可能不一致，但含义相同。串行通信接口的外围硬件电路，主要目的是将 MCU 的发送引脚 TXD 与接收引脚 RXD 的 TTL 电平，通过 RS-232 电平转换芯片转换为 RS-232 电平。图 4-4 给出了基本串行通信接口的电平转换电路。

基本工作过程是：

发送过程：MCU 的 TXD（TTL 电平）经过 MAX232 的 11 脚(T1IN)送到 MAX232 内部，在内部 TTL 电平被“提升”为 232 电平，通过 14 脚(T1OUT)发送出去。

接收过程：外部 232 电平经过 MAX232 的 13 脚（R1IN）进入到 MAX232 的内部，在内部 232 电平被“降低”为 TTL 电平，经过 12 脚（R1OUT）送到 MCU 的 RXD，进入 MCU 内部。

进行 MCU 的串行通信接口编程时，只针对 MCU 的发送与接收引脚，与 MAX232 无关，MAX232 只是起到电平转换作用。

4.2 MCF52233的UART模块的物理结构

MCF52233 的通用异步收发器 UART，支持半双工和全双工的数据传输。其波特率可根据具体应用编程设定，典型的 UART 数据帧有 7 位或 8 位，并有 1 或 2 位停止位，同时它也支持 5 或 6 位的数据帧。UART 发送器的硬件可产生并发送奇偶校验位，而接收器的奇偶校验硬件则能据此确保接收数据的完整性。UART 能适时的向 CPU 或 DMA 控制器发出中断或请求。利用 DMA 控制器，应用程序代码可在很少或根本没有 CPU 干预的情况下，发送和接收数据。

本节主要阐述 MCF52233 的 UART 模块的结构，包括波特率产生器、发送器和接收器以及 DMA 请求产生器等，对于初次接触带 DMA 的 UART 模块的读者来说，阅读起来困难较大，不易理解，因此，建议在阅读本节时，结合后续章节所介绍的 UART 和 DMA 编程结构，或直接参考 MCF5223x 的英文手册来理解。也可以先略读本节，待基本理解并验证本章的程序样例后，再回过头来理解 MCF52233 的 UART 的物理结构。

MCF5223x 最多可包括 3 个相同且独立的 UART 模块，每个模块都含有相互独立的发送器和接收器，图 4-5 所示为模块的结构框图，其主要组成结构有 UART 寄存器组、内部通道控制逻辑、中断控制逻辑、DMA 请求逻辑、串行通信通道和可编程时钟发生器等，其中，串行通信通道包括发送器和接收器两个部分。

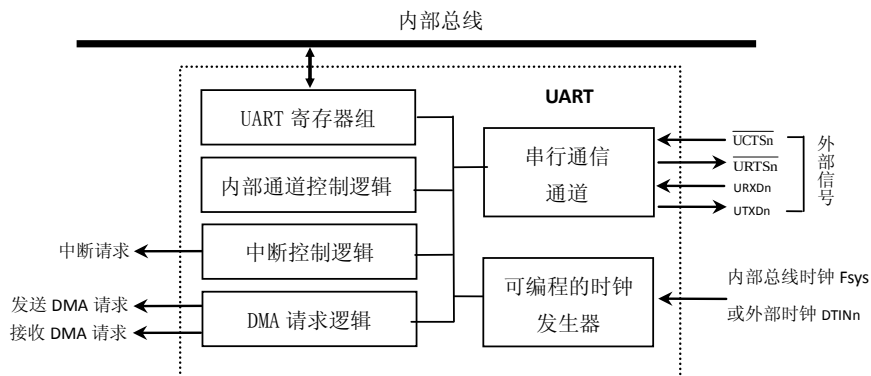


图4-5 UART模块结构框图

1. 外部引脚

UART 的外部引脚有：

- (1) 发送数据引脚：UTXDn
- (2) 接收数据引脚：URXDn
- (3) 请求发送引脚：URTSn
- (4) 允许发送引脚：UCTSn

引脚名中的“U”是 UART 的简写，“n”表示模块的编号，取 0、1 或 2。通常情况串行通信只使用发送数据引脚 UTxDn 与接收数据引脚 URxDn，若要使用流控制则需使用请求发送引脚 URTSn 和允许发送引脚 UCTSn。关于流控制问题见 4.9 节。

MCF52233 集成了四通道的 DMA 控制器，开发者可以根据需要将具体通道分配给

UART 模块的发送器或接收器，而这要依赖于图 4-5 中 UART 寄存器组和 DMA 请求逻辑的硬件实现。由于 UART 模块可以直接与 DMA 模块进行关联编程，因此，DMA 也放在本章讨论。

2. 波特率发生器的内部结构

UART 模块时钟可以来源于系统的总线时钟 (F_{sys})，或 4 个 UART DMA 定时器输入引脚 $DTIN_n$ 中的一个。图 4-6 所示的功能框图描述了 UART 的时钟选择逻辑。

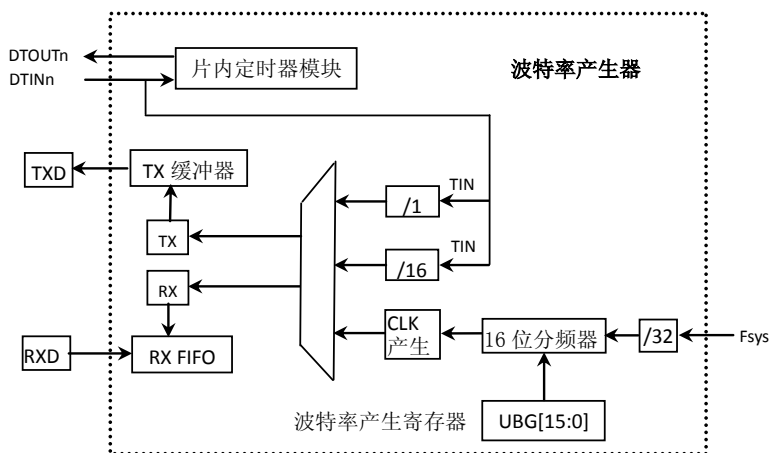


图4-6 波特率发生器的内部结构

若使用系统总线时钟 F_{sys} ，首先被 32 分频，然后按 16 位 UART 波特率产生寄存器 UBG 决定的分频因子进行分频。若选择 DMA 定时器输入引脚 $DTIN_n$ 作为串行通信模块的输入时钟，则软件可以选择同步（1 分频）或异步（16 分频）两种模式，有关这方面的详细内容，请参考 MCF5223x 的芯片手册。用来设置发送和接收时钟的是 UART 时钟选择寄存器（UART Clock Select Register, UCSR）。

3. 发送器的内部结构

发送器包含双缓冲区，该双缓冲区由 UART 发送缓冲寄存器（Transim Buffer Register, TBR）和串行移位器组成，如图 4-7 所示。通过写发送缓冲寄存器（或叫发送缓冲区）的存储器 I/O 地址来实现字节的发送。

对于数据发送，当发送器已经为下一个要发送的字符准备就绪时，CPU 写 UART 的发送缓冲寄存器，之后发送器会自动添加起始位，如果编程为允许，也可以添加 1 个奇偶位，1 个或 2 个停止位。UART 发送出去的字符长度（可编程为 5-8 位）取决于 UART 模式寄存器 1（UART Mode Register 1, UMR1）中 B/C 位域的值。

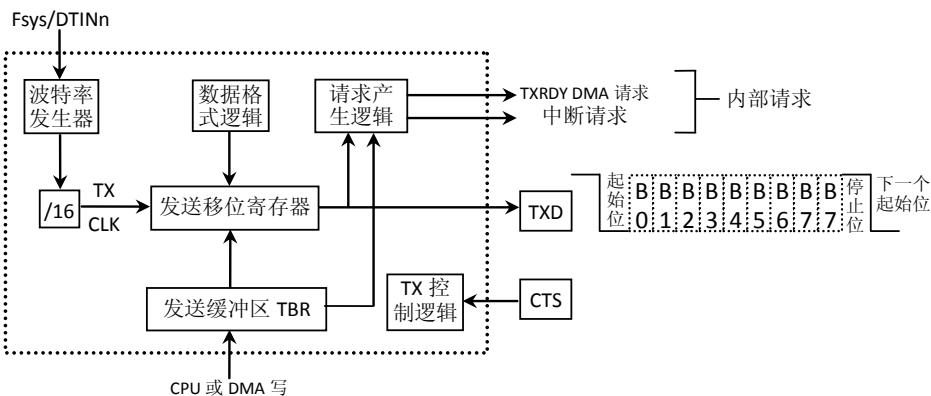


图4-7 发送器的内部结构

当字符被完全移出时，如果发送缓冲区中有字符，则它将被立即传送到发送移位寄存器，且发送缓冲区变成空。这时，UART 状态寄存器（UART Status Register, USR）中的发送就绪状态标志（TXDRY）自动置“1”，表示已为下个字符准备就绪。此外，开发者可通过寄存器设置，使 UART 在这时能向 CPU 发出 UART 发送中断，或向 DMA 控制器发出 UART 发送请求。

用 UART 中断屏蔽寄存器（UART Interrupt Mask Register, UIMR）中的 TXRDY 屏蔽位可将 UART 中断状态寄存器（UART Interrupt Status Register, UISR）的 TXRDY 位设置成允许。UISR 和 UIMR 寄存器共享相同的 I/O 存储地址，对该地址的读操作将访问 UISR，而写操作则访问 UIMR。

4. 接收器的内部结构

接收器的前端部分由串行移位器和三阶先进先出（FIFO）队列组成。与发送器类似，应用软件不能直接存取串行移位寄存器，三阶的 FIFO 作为整体映射到相同的存储器 I/O 地址，应用程序通过读该地址来取得接收器的数据。

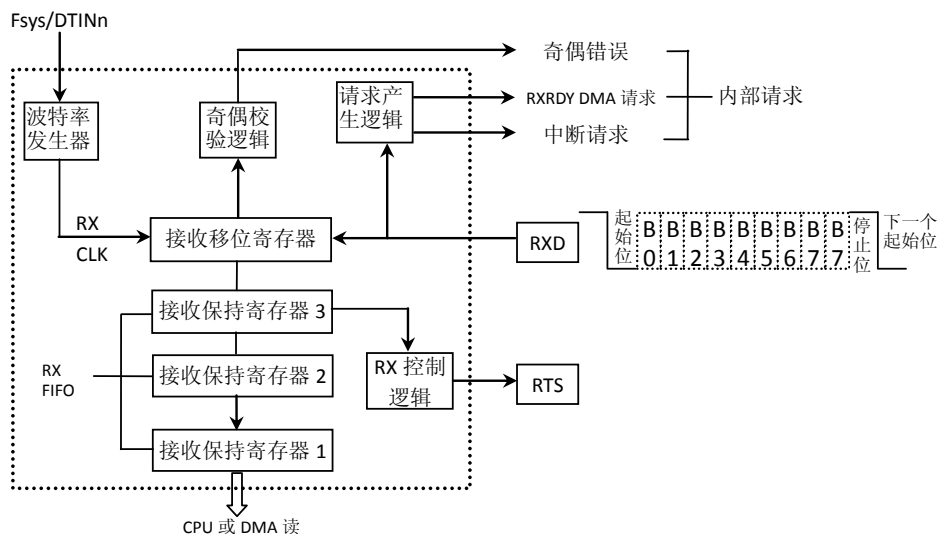


图4-8 接收器的内部结构

当串行移位寄存器中装入（接收）字符之后，UART 将自动地把字节数据装入到接收 FIFO，DMA 或 CPU 将从三阶的 FIFO 顶端读取接收到的字符。

图 4-8 为 UART 接收器的功能框图，图中所示收到的帧有 8 个数据位和 1 个停止位。在收到字符后，UART 自动将接收就绪标志位（RXRDY）置位，如果接收 FIFO 满，则 FIFO 满标志（FFULL）也被置位，该情况下，后面接收到的字符可能会丢失。这时，根据开发者的编程设定，UART 会向 CPU 发送 UART 接收中断或向 DMA 控制器发送 UART 接收请求，CPU 或 DMA 将响应并将 UART 保持寄存器中的数据传送到系统存储器中。

UART 可以产生多个中断请求，但只使用同一个中断向量，因此需要查询 UART 状态寄存器，以确定中断请求产生的原因。在 DMA 提供服务的过程中，UART 将在字符读取时自动清零 RXRDY 标志。

5. DMA请求产生器

UART 可以配置为从发送器或接收器中产生 DMA 传送请求，从而取代对 CPU 产生的中断。如果允许，当发送器设置 UART 中断状态寄存器（UISR）中的发送就绪（TXRDY）标志，准备发送字符时，发送器可产生 DMA 请求。

在发送字符时，DMA 从存储器队列中启动读操作，并且写入 UART 发送缓冲区（TB）中。当单个字符被 UART 完全发送后，UART 将告知 DMA 写入下个等待发送的字符，这样字符将逐个被发送，而 DMA 在没有 CPU 干预的情况下，将数据从系统存储器送到 UART 发送器。一旦整个报文都被发送到 UART，如果允许，DMA 会向 CPU 产生数据传送结束中断。中断服务程序可以查询 UART 中断状态寄存器（UISR）以确定发送结束时的状态。同样，接收器也能产生 DMA 请求以取代 CPU 中断。

4.3 MCF52233的UART模块的编程结构

表 4-3 给出了 MCF52233 的 UART 模块各寄存器相关信息,最左边一列给出了各寄存器的映射地址,它们是相对于内部外设系统基地址寄存器 IPSBAR (Internal Peripheral System Base Address Register) 的偏移量。最后三列分别给出了相应寄存器的位宽、访问权限和复位时的状态信息。表中给出的都是 UART0 的地址,对于其他两个 UART 来说,同名寄存器的地址相对于 UART0 的偏移分别为 0x40 和 0x80。如, UART0 的模式寄存器的基址偏移为 0x00_0200,则 UART1 为 0x00_0240,而 UART2 为 0x00_0280,其他寄存器类似。

表 4-3 UART 模块存储映射

基址寄存器偏移 UART0	寄存器	位宽	权限	复位值
0x00_0200	UART模式寄存器 (UMR1和UMR2)	8	读/写	0x00
0x00_0204	UART状态寄存器 (USR)	8	只读	0x00
	UART时钟选择寄存器 (UCSR)	8	只写	0x00
0x00_0208	UART命令寄存器 (UCR)	8	只写	0x00
0x00_020C	UART接收缓冲区 (URB)	8	只读	0xFF
	UART发送缓冲区 (UTB)	8	只写	0x00
0x00_0210	UART输入端口改变寄存器 (UIPCR)	8	只读	0x00
	UART辅助控制寄存器 (UACR)	8	只写	0x00
0x00_0214	UART中断状态寄存器 (UISRn)	8	只读	0x00
	UART中断屏蔽寄存器 (UIMR)	8	只写	0x00
0x00_0218	UART波特率发生器寄存器1 (UBG1)	8	只写	0x00
0x00_021C	UART波特率发生器寄存器2 (UBG2)	8	只写	0x00
0x00_0234	UART输入端口寄存器 (UIP)	8	只读	0xFF
0x00_0238	UART输出端口位命令设置寄存器 (UOP1)	8	只写	0x00
0x00_023C	UART输出端口位命令复位寄存器 (UOP0)	8	只写	0x00

以上寄存器的用法在 MCF5223x 的芯片手册上有详细的说明,下面按初始化顺序简要阐述基本编程需要使用的寄存器。注意,下面所列寄存器名中的“n”表示 UART 模块编号,取 0、1 和 2。

1. UART命令寄存器 (UCRn)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	保留	MISC			TC		RC	
复位	0							

D6~D4—MISC, 复位命令位域。000: 无命令, 001: 复位模式寄存器指针, 010: 复位接收器, 011: 复位发送器, 100: 复位错误状态信息, 101: 复位状态改变中断标志, 110: 开启中止, 111: 停止中止。

D3~D2—TC, 发送器控制位域。00: 无操作; 01: 使能; 10: 关闭; 11: 保留。

D1~D0—RC, 接收器控制位域。00: 无操作; 01: 使能; 10: 关闭; 11: 保留。

2. UART模式寄存器1 (UMR1n)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RXRTS	RXIRQ/FFULL	ERR	PM		PT	B/C	
复位	0							

D7—RXRTS，发送器请求发送位。

D6—RXIRQ/FFULL，接收器中断选择位。0：RXRDY 为源产生中断或 DMA 请求；

1：FFULL 为源产生中断或 DMA 请求。

D5—ERR，错误模式位。

D4~D2—PM&PT，奇偶校验模式&奇偶校验类型位。奇偶校验选择如表 4-4 所示。

表 4-4 奇偶校验选择

PM	奇偶校验模式	奇偶校验类型 (PT=0)	奇偶校验类型 (PT=1)
00	奇偶校验	偶校验	奇校验
01	强制奇偶校验	低位奇偶校验	高位奇偶校验
10	无奇偶校验		
11	多点模式	数据字符	地址字符

D1~D0—B/C，每个字符的位数。选择每个要发送字符的数据位的个数。这些位不包括开始位、奇偶校验位和停止位。00：5 位；01：6 位；10：7 位；11：8 位。

3. UART模式寄存器2 (UMR2n)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	CM		TXRTS	TXCTS	SB			
复位	0							

D7~D6—CM，通道模式。00：普通；01：自动回应；10：本地回环；11：远端回环。

D5—TXRTS，准备发送。

D4—TXCTS，允许发送。

D3~D0—SB，停止位长度。

4. UART时钟选择寄存器 (UCSRn)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RCS				TCS			
复位	0							

D7~D4—RCS，指定接收器时钟源。

D3~D0—TCS，指定发送器时钟源。

接收器和发送器的时钟源，可选择内部总线时钟作为时钟源，也可选择外部时钟 DTINn（1 分频或 16 分频）。1101：内部总线时钟，1110：DTINn 16 分频，1111：DTINn 1 分频，其他：发送器和接收器使用相同时钟源。例如，要使两者都用内部总线时钟，则需将 UCSRn 设置为 0xDD。

5. UART波特率发生器寄存器 (UBG1和UBG2)

UART 波特率发生器寄存器是一个 16 位寄存器，可以分为两个 8 位寄存器：UBG1

和 UBG2, UBG1 存放波特率值的高 8 位, UBG2 存放波特率值的低 8 位。

6. UART 状态寄存器 (USRn)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RB	FE	PE	OE	TXEMP	TXRDY	FFULL	RXRDY
复位	0							

D7—RB, 接收到 Break 字位。0: 未收到; 1: 收到。关于 Break 字的含义见 4.9 节。

D6—FE, 帧差错位。0: 无; 1: 有。仅当 RXRDY 为 1 时才有效。

D5—PE, 奇偶校验错误位。0: 无; 1: 有。仅当 RXRDY 为 1 时才有效。

D4—OE, 溢出错误位。0: 无溢出发生; 1: 有溢出发生。

D3—TXEMP, 发送器空位。0: 发送器缓存不空; 1: 发送器空。

D2—TXRDY, 发送器准备完毕位。0: CPU 已写发送器的保持寄存器或发送器不可用; 1: 发送器的保持寄存器为空并且准备好发送字符。

D1—FFULL, FIFO 满位。0: FIFO 未满足但是可能保留两个未读的字符; 1: 收到字符且接收 FIFO 已满。当 FIFO 满时, 任何到达的字符都会丢失。

D0—RXRDY, 接收器准备完毕位。0: 接收 FIFO 缓冲区的数据全部读取结束; 1: 若干字符到达并在接收缓冲 FIFO 中等待。

7. UART 中断状态/屏蔽寄存器 (UISRn/UIMRn)

数据位	D7	D6-D3	D2	D1	D0
定义	COS	0	DB	FFULL/RXRDY	TXRDY
复位	0	0	0	0	0

UART 最多可允许产生 4 个中断请求: 1 个来自于发送器, 3 个来自于接收器。UART 中断状态寄存器 (UISR) 提供这 4 个中断请求的标志。其中最常用的是接收中断。

D7—COS, 状态改变 (Change-Of-State) 中断标志。CTS 引脚上任何电平的转变会使得该中断标志位被置位。通过读取 UART 输入端口改变寄存器 (UIPCR) 中的值来得到 CTS 状态。读 UIPCR 将自动取消 COS 中断标志位。

D2—DB, Break 信号的中断标志。只要接收器检测到 Break 字, 该标志即被置位。接收器在收到的字符为全零且没有检测到停止位时, 断言为 Break 条件。写 UART 命令寄存器 MISC 位域的值为 “101”, 可将 DB 标志清零。

D1—FFULL/RXRDY, FIFO 满/接收就绪中断标志 (也称接收中断标志)。当接收器的 RB 中有数据, 该标志置位。用户可以选择 RXRDY 或 FIFO 满作为中断请求源, 该选择是通过 UART 模式寄存器 1 (UMRI) 中的接收中断选择控制位 (RXIRQ/FFULL) 来实现的。当 CPU 读 RB 时, RXRDY 中断标志将自动清零。

D0—TXRDY, 发送器准备完毕中断标志。当发送缓冲寄存器空时 UART 发送器置位该标志。

4.4 基于构件方法的UART编程

4.4.1 UART构件的函数原型设计

UART 具有初始化、接收和发送三种基本操作。按照构件的思想,可将它们封装成三个独立的功能函数,初始化函数完成对 UART 模块的工作属性的设定,接收和发送功能函数则完成实际的通信任务。对 UART 模块进行编程,实际上已经涉及到对硬件底层寄存器的直接操作,因此,可将初始化、接收和发送三种基本操作所对应的功能函数共同放置在命名为 UART.c 的文件中,并按照相对严格的构件设计原则对其进行封装,同时配以命名为 UART.h 的头文件,用来定义模块的基本信息和对外接口。

按照模块所具有的基本操作来确定构件中应该具有哪些功能集合,是很自然也很重要的事情。但是,要实现编程的构件化,对具体的函数原型的设计则是重中之重。函数原型设计的好坏直接体现构件化编程的成败。下面就以 UART 的初始化、接收和发送三种基本操作为例,来说明实现构件化的全过程。

需要说明的是,实现构件化编程的 UART 软件模块应当具有以下几个特点:

(1) UART 模块是最底层的构件,它主要向上提供三种服务,分别是 UART 模块的初始化、接收单个字节和发送单个字节,向下则直接访问模块寄存器,实现对硬件的直接操作。另外,从现实使用角度出发,它还需要封装接收 N 个字节和发送 N 个字节的子功能函数。

(2) UART 模块在软件上对应 1 个 UART.c 程序源代码文件和 1 个 UART.h 头文件,当需要对它进行移植时,大多数情况下只需简单拷贝这两个文件即可,无需对源代码文件和头文件进行修改,只有当实施不同芯片之间的移植时,才需要修改头文件中与硬件相关的宏定义。

(3) 上层构件或软件在使用该构件时,严格禁止通过全局变量来传递参数,所有的数据传递都直接通过函数的形式参数来接收。这样做不但使得接口简洁,更加避免了全局变量可能引发的安全隐患。

UART 模块是最基本最底层的构件,它在满足功能完整性的情况下,具有不可再分的特性,可以把它叫做“元构件”。下面简要介绍,按照构件化的思想对 UART 模块进行编程的过程。

通过以上分析,可以设计 UART 构件的 5 个基本功能函数。

(1) 初始化

```
void UartInit(uint8 uartNo, int32 sysclk, int32 baud, uint8 intStatusInit)
```

(2) 发送单个字节

```
void UartSend1(uint8 uartNo, uint8 ch)
```

(3) 接收单个字节

```
uint8 UartRel (uint8 uartNo, uint8 *p)
```

(4) 发送 N 个字节

```
void UartSendN(uint8 uartNo, uint8 n, uint8 ch[])
```

(5) 接收 N 个字节

```
uint8 UartReN(uint8 uartNo, uint8 n, uint8 ch[])
```

4.4.2 UART构件的头文件

与 UART 通信子函数相关的文件有头文件 **UART.h**，以及包含 UART 初始化和收发子函数的程序文件 **UART.c**。

头文件 **UART.h** 中的内容可分为两个主要的部分，它们分别是 5 个函数原型的声明和外设模块寄存器相关信息的定义。前者给出了本 UART 构件对上层构件或软件所提供的接口函数，而后者则指明了本“元构件”与具体硬件相关的信息。

串行通信头文件 **UART.h** 含有串行通信寄存器和标志位定义，以及串行通信相关函数声明，下表简要给出了它的主要内容。以初始化函数 **UartInit** 为例，通道号、当前的系统时钟、希望实现的通信波特率以及初始化时是否使能中断，都被设计为函数参数。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。文件还给出了必要的硬件相关信息，当要把该构件移植到其他芯片时，就必须检查并修改这些信息。

```
//-----*
//Uart.h串行通信头文件                                *
//-----*
#ifndef UART_H
#define UART_H
//1. 对外引用
#include "type.h"          //类型定义头文件
extern uint8 __IPSBAR[];   //引用外部标号
//2. 外用UART通信函数声明
//-----*
//函数名: UartInit                                          *
//功 能: 初始化UARTx模块。x代表0, 1, 2                    *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2 *
//          如果uartNo大于2, 则认为是2                      *
//          int32 sysclk: 系统总线时钟, 以MHz为单位          *
//          int32 baud: 波特率, 如 4800, 9600, 19200, 38400, 57600, 115200 *
//          其他的波特率在MCF52233的评估板上没有测试过, 若要使 *
//          用该函数, 请自行测试, 一般来说, 速度慢一点, 通信会 *
//          更稳定                                           *
//          uint8 intStatusInit: 初始化中断的状态 1表示开中断 其他表示不开 *
//返 回: 无                                                *
//说 明: uartNo=0表示使用UART0模块, 依此类推。            *
//-----*
void UartInit(uint8 uartNo, int32 sysclk, int32 baud, uint8 intStatusInit);

//-----*
//函数名: UartSend1                                          *
//功 能: 串行发送1个字节                                    *
```

```

//参 数: uint8 uartNo: 第uartNo个UART模块, 其中uartNo取值为0, 1, 2      *
//      uint8 ch:      要发送的字节                                      *
//返 回: 无                                                            *
//说 明: uartNo=0表示使用UART0模块, 依此类推                          *
//-----*
void UartSend1(uint8 uartNo, uint8 ch);

//-----*
//函数名: UartSendN                                                    *
//功 能: 串行发送N个字节                                              *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2    *
//      uint8 n:      发送的字节数                                      *
//      uint8 ch[]:   待发送的数据                                    *
//返 回: 无                                                            *
//说 明: uartNo=0表示使用UART0模块, 依此类推。                        *
//      调用了UartSend1函数                                          *
//-----*
void UartSendN(uint8 uartNo, uint8 n, uint8 ch[]);

//-----*
//函数名: UartRe1                                                    *
//功 能: 从串口接收1个字节的数据                                      *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2    *
//返 回: 接收到的数据 (若接收失败, 返回0xff)                          *
//      uint8 *p:     接收成功标志的指针 (0表示成功, 1表示不成功)      *
//说 明: 参数*p带回接收标志, *p = 0, 收到数据; *p = 1, 未收到数据      *
//      uartNo=0表示使用UART0模块, 依此类推。                        *
//-----*
uint8 UartRe1 (uint8 uartNo, uint8 *p);

//-----*
//函数名: UartReN                                                    *
//功 能: 从串口接收N个字节的数据                                      *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2    *
//      uint8 n:      要接收的字节数                                    *
//      uint8 ch[]:   存放接收数据的数组                                *
//返 回: 接收标志= 0 接收成功, = 1 接收失败                            *
//说 明: uartNo=0表示使用UART0模块, 依此类推。                        *
//      调用了UartRe1函数                                          *
//-----*
uint8 UartReN(uint8 uartNo, uint8 n, uint8 ch[]);

//-----*
//函数名: UartSendString                                              *
//功 能: 串口传输字符串                                              *
//参 数: uint8 uartNo: 第uartNo个UART模块; p: 要传输的字符串的指针    *
//返 回: 无                                                            *
//说 明: 字符串以'\0'结束                                            *
//      调用了UartSend1函数                                          *

```

```

//-----*
void UartSendString(uint8 uartNo, char *p);

//3. 模块相关的宏定义
//示例波特率的宏定义
#define UART_BAUD          115200    //115200
//标志位掩码的定义
#define MCF_UART_RXD_MASK    (0x04)    //UART接收引脚定义掩码
#define MCF_UART_TXD_MASK    (0x01)    //UART发送引脚定义掩码
#define MCF_UART_RX_TEST     (0x01)    //UART接收缓冲区满标志位掩码
#define MCF_UART_TX_TEST     (0x04)    //UART发送缓冲区空标志位掩码
#define MCF_UART_RESET_TX    (0x30)    //UART命令寄存器复位发送器掩码
#define MCF_UART_RESET_RX    (0x20)    //UART命令寄存器复位接收器掩码
#define MCF_UART_RESET_MR    (0x10)    //UART复位模式寄存器掩码
#define MCF_UART_NONE_PARITY (0x10)    //UART模式寄存器无奇偶校验掩码
#define MCF_UART_BIT_8       (0x03)    //UART模式寄存器8位传输掩码
#define MCF_UART_CM_NORMAL   (0x00)    //UART模式寄存器正常通道模式掩码
#define MCF_UART_STOP_BITS_1 (0x07)    //UART模式寄存器1位停止位掩码
#define MCF_UART_RCS_SYSCLK   (0xD0)    //UART时钟选择寄存器接收时钟选择掩码
#define MCF_UART_TCS_SYSCLK   (0x0D)    //UART时钟选择寄存器接收时钟选择掩码
#define MCF_UART_RX_ENABLED   (0x01)    //UART命令寄存器使能接收器掩码
#define MCF_UART_TX_ENABLED   (0x04)    //UART命令寄存器使能发送器掩码
//ColdFire的UART模块相关寄存器的定义
//x=0表示UART0模块 x=1表示UART1模块 x=2表示UART2模块
//UART的IO端口功能分配寄存器定义, 其中x=0表示UART0端口, 依此类推
#define MCF_GPIO_PUPAR(x)    (*(vuint8 *)(&__IPSBAR[0x100071+(x)]))
//UART模式寄存器
#define MCF_UART_UMR(x)      (*(vuint8 *)(&__IPSBAR[0x000200+((x)*0x040)]))
//UART状态寄存器
#define MCF_UART_USR(x)      (*(vuint8 *)(&__IPSBAR[0x000204+((x)*0x040)]))
//UART时钟选择寄存器
#define MCF_UART_UCSR(x)     (*(vuint8 *)(&__IPSBAR[0x000204+((x)*0x040)]))
//UART命令寄存器
#define MCF_UART_UCR(x)      (*(vuint8 *)(&__IPSBAR[0x000208+((x)*0x040)]))
//UART接收数据寄存器
#define MCF_UART_URB(x)      (*(vuint8 *)(&__IPSBAR[0x00020C+((x)*0x040)]))
//UART发送数据寄存器, 接收和发送数据寄存器使用同一个地址
#define MCF_UART_UTB(x)      (*(vuint8 *)(&__IPSBAR[0x00020C+((x)*0x040)]))
//UART中断状态/掩码寄存器
#define MCF_UART_UIMR(x)     (*(vuint8 *)(&__IPSBAR[0x000214+((x)*0x040)]))
//UART波特率寄存器1(高位)
#define MCF_UART_UBG1(x)     (*(vuint8 *)(&__IPSBAR[0x000218+((x)*0x040)]))
//UART波特率寄存器2(低位) Baud=fsys/(32*divider)
#define MCF_UART_UBG2(x)     (*(vuint8 *)(&__IPSBAR[0x00021C+((x)*0x040)]))
//定义UART中断相关的寄存器
//UART的中断控制器寄存器
#define MCF_UART_ICR_LEVEL(x) (*(vuint8 *)(&__IPSBAR[0x000C4D+(x)]))
//中断掩码控制器
#define MCF_UART_IMRL        (*(vuint32*)(&__IPSBAR[0x000C0C]))

```

```
//中断掩码控制器中UART掩码设置(13, 14, 15位)
#define MCF_UART_IMRL_MASK(x) (0x00002000<<x)
#endif
```

4.4.3 UART构件的源程序文件

1. UART构件的初始化功能函数: UartInit

一般情况下使用内部总线时钟 Fsys 作为 UART 的时钟源, 则初始化函数中的分频因子计算方法如下:

由公式:

$$\text{波特率} = F_{\text{sys}} / (32 \times \text{分频因子})$$

计算分频因子:

$$\text{分频因子} = F_{\text{sys}} / (32 \times \text{波特率})$$

例如, 使用 60MHz 系统时钟频率, 且需要设置的波特率为 115200, 则有:

$$\text{分频因子} = 60000000 / (32 \times 115200) = 16.3$$

取整为 16, 即十六进制的 \$0010, 编程将 \$0010 写入 16 位 UART 波特率寄存器 UBG1:UBG2 中。

```
//-----*
//函数名: UartInit *
//功 能: 初始化UARTx模块。x代表0, 1, 2 *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2 *
//          如果uartNo大于2, 则认为是2 *
//          int32 sysclk: 系统总线时钟, 以MHz为单位 *
//          int32 baud: 波特率, 如 4800, 9600, 19200, 38400, 57600, 115200 *
//          其他的波特率在MCF52233的评估板上没有测试过, 若要使 *
//          用该函数, 请自行测试, 一般来说, 速度慢一点, 通信会 *
//          更稳定 *
//          uint8 intStatusInit: 初始化中断的状态 1表示开中断 其他表示不开 *
//返 回: 无 *
//说 明: uartNo=0表示使用UART0模块, 依此类推。 *
//-----*
void UartInit(uint8 uartNo, int32 sysclk, int32 baud, uint8 intStatusInit)
{
    register uint16 ubgs;
    if(uartNo > 2)
    { //若传进的通道号大于2, 则按照2来接受
        uartNo = 2;
    }
    //1. 将Ux口初始化为UART功能
    MCF_GPIO_PUPAR(uartNo) |= (0
        | MCF_UART_RXD_MASK
        | MCF_UART_TXD_MASK);
    //2. 复位接收器和发送器以及模式寄存器
    MCF_UART_UCR(uartNo) = MCF_UART_RESET_TX; //复位发送器
```

```

MCF_UART_UCR(uartNo) = MCF_UART_RESET_RX; //复位接收器
MCF_UART_UCR(uartNo) = MCF_UART_RESET_MR; //复位模式寄存器
//3. 配置UART模式:无奇偶校验、8位数据、正常通道模式、1停止位
MCF_UART_UMR(uartNo) = (0
    | MCF_UART_NONE_PARITY
    | MCF_UART_BIT_8 );
MCF_UART_UMR(uartNo) = (0
    | MCF_UART_CM_NORMAL
    | MCF_UART_STOP_BITS_1);
//4. 选择预分频后的内部总线时钟作为收发器的时钟源
MCF_UART_UCSR(uartNo) = (0
    | MCF_UART_RCS_SYSCLK
    | MCF_UART_TCS_SYSCLK);
//5. 屏蔽所有的UART中断
MCF_UART_UIMR(uartNo) = 0;
//6. 计算波特率并设置:UBG = fsys/(波特率*32)
ubgs = (uint16)((sysclk*1000000)/(baud * 32));
MCF_UART_UBG1(uartNo) = (uint8)((ubgs & 0xFF00) >> 8);
MCF_UART_UBG2(uartNo) = (uint8)(ubgs & 0x00FF);
//7. 使能接收器和发送器
MCF_UART_UCR(uartNo) = (0
    | MCF_UART_RX_ENABLED
    | MCF_UART_TX_ENABLED);
//8. 决定是否开放UARTx接收中断
if(1 == intStatusInit)
{ //开串行接收中断
    MCF_UART_ICR_LEVEL(uartNo) = 0x3F;
    MCF_UART_UIMR(uartNo) = 0x02;
    MCF_UART_IMRL &= ~MCF_UART_IMRL_MASK(uartNo);
}
else
{ //禁串行接收中断
    MCF_UART_ICR_LEVEL(uartNo) = 0x00;
    MCF_UART_UIMR(uartNo) = 0x00;
    MCF_UART_IMRL |= MCF_UART_IMRL_MASK(uartNo);
}
}

```

2. UART构件单字节发送功能函数: UartSend1

```

//-----*
//函数名: UartSend1                                *
//功 能: 串行发送1个字节                            *
//参 数: uint8 uartNo: 第uartNo个UART模块, 其中uartNo取值为0, 1, 2 *
//      uint8 ch:      要发送的字节                    *
//返 回: 无                                            *
//说 明: uartNo=0表示使用UART0模块, 依此类推          *
//-----*
void UartSend1(uint8 uartNo, uint8 ch)

```

```

{
    if(uartNo > 2)
    { //若传进的通道号大于2, 则按照2来接受
        uartNo = 2;
    }
    //等待FIFO中有空闲缓冲区可用
    while (!(MCF_UART_USR(uartNo) & MCF_UART_TX_TEST));
    //发送数据
    MCF_UART_UTB(uartNo) = ch;
}

```

3. UART构件的单字节接收功能函数: UartRe1

```

//-----*
//函数名: UartRe1                                     *
//功 能: 从串口接收1个字节的数据                     *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2 *
//返 回: 接收到的数(若接收失败, 返回0xff)             *
//      uint8 *p:      接收成功标志的指针(0表示成功, 1表示不成功) *
//说 明: 参数*p带回接收标志, *p = 0, 收到数据; *p = 1, 未收到数据 *
//      uartNo=0表示使用UART0模块, 依此类推。           *
//-----*
uint8 UartRe1 (uint8 uartNo, uint8 *p)
{
    uint16 k;
    uint8 i;
    if(uartNo > 2)
    { //若传进的通道号大于2, 则按照2来接受
        uartNo = 2;
    }
    //等待接收数据
    for(k=0; k < 0xfbbb; k++)
        if (MCF_UART_USR(uartNo) & MCF_UART_RX_TEST)
        {
            i = MCF_UART_URB(uartNo);
            *p = 0x00;
            break;
        }
    //接收错误
    if (k >= 0xfbbb)
    {
        i = 0xff;
        *p = 0x01;
    }
    return i;
}

```

4. UART构件的多字节发送功能函数：UartSendN

```
//-----*
//函数名: UartSendN                                     *
//功 能: 串行发送N个字节                               *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2 *
//      uint8 n:      发送的字节数                       *
//      uint8 ch[]:   待发送的数据                       *
//返 回: 无                                              *
//说 明: uartNo=0表示使用UART0模块, 依此类推。          *
//      调用了UartSend1函数                             *
//-----*
void UartSendN(uint8 uartNo, uint8 n, uint8 ch[])
{
    uint8 i;
    if(uartNo > 2)
    { //若传进的通道号大于2, 则按照2来接受
        uartNo = 2;
    }
    for(i=0; i<n; i++)
    { //发送n个字节
        UartSend1(uartNo, ch[i]);
    }
}
```

5. UART构件的多字节接收功能函数：UartReN

```
//-----*
//函数名: UartReN                                       *
//功 能: 从串口接收N个字节的数据                       *
//参 数: uint8 uartNo: 第uartNo个UART模块。其中uartNo取值为0, 1, 2 *
//      uint8 n:      要接收的字节数                     *
//      uint8 ch[]:   存放接收数据的数组                 *
//返 回: 接收标志= 0 接收成功, = 1 接收失败             *
//说 明: uartNo=0表示使用UART0模块, 依此类推。          *
//      调用了UartRel函数                             *
//-----*
uint8 UartReN(uint8 uartNo, uint8 n, uint8 ch[])
{
    uint8 m, fp;
    m = 0;
    if(uartNo > 2)
    { //若传进的通道号大于2, 则按照2来接受
        uartNo = 2;
    }
    while (m < n)
    { //接收n个字节, 若任一字节接收失败, 即返回接收失败
        ch[m] = UartRel(uartNo, &fp);
        if (fp == 1)

```

```

        {
            return 1;
        }
        m++;
    }
    return 0;
}

```

4.4.4 UART构件的测试工程

如第3章的“3.4 CW 工程文件组织”一节所述，嵌入式软件中，main.c 和 isr.c 这两个文件反映了软件系统的整体执行流程。当系统启动并初始化后，程序根据 main.c 中定义的主循环顺序执行；一旦遇到中断请求，立即转去执行 isr.c 中定义的相应中断处理程序；当中断处理程序运行结束后，再返回中断处继续顺序执行。

在 main.c 和 isr.c 中，可通过调用上一小节介绍的5个功能接口函数实现 UART 模块收发数据，实现方式有两种：查询方式和中断方式。

光盘中的 UART 构件的测试工程“...\ Chap04_UART+DMA\01-UartSrchr”采用查询方式实现：UART0 模块首先向 PC 机发送字符串“Hello! World!”，然后等待接收 PC 机从串口发送来的数据，若成功接收到1个数据，则立即将该数据回发给 PC 机，随后继续等待接收1个数据并回发，如此循环。主函数文件 main.c 如下编写：

```

//-----*
// 工 程 名: UartSrchr*
// 硬件连接: 接PC机串口*
// 程序描述: 使用UART0和PC机通信*
//          初始时发送"Hello! World!"到PC机1次*
//          同时等待接收PC机发送的数据，当接收到了来自PC机的数据后立即回发*
// 目    的: 初步掌握利用查询方式进行串行通信的基本知识*
// 说    明: 波特率为38400, 使用UART0口*
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
#include "includes.h" //头文件
uint8 msg[14]= "Hello! World! ";
int main(void)
{
    uint8 m, fp;
    //1. 关总中断
    DisableInterrupts();
    //2. UART初始化
    //使用UART0模块, 使用系统时钟作为UART的时钟源
    //波特率为38400, 初始化时禁止接收中断
    UartInit(0, SYSTEM_CLOCK, 38400, 0);
    //3. 开总中断
    EnableInterrupts();
    //4. 发送msg
    UartSendN(0, 14, msg);
    //5. 主循环

```

```

while (1)
{
    //等待成功接收串口的数据
    m = UartRel(0, &fp);
    if(0 == fp)
    { //成功接收串口数据, 则立即回发
        UartSend1(0, m);
    }
}
return 0;
}

```

在本书的光盘中, 有两个用来测试串行通信的高端程序, 运行界面如图 4-9 和 4-10 所示, 其文件在“...测试工具”文件夹内。建议读者充分使用这些有效工具。



图4-9 串口调试工具软件界面



图4-10 串行通信收发测试软件界面

4.5 ColdFire第一个带有中断功能的实例

采用中断方式收发数据时, 需编写中断处理程序。在 CW 环境下使用 ColdFire 芯片中断的步骤是:

- (1) 在 main.c 中, 依照“关总中断→开模块中断→开总中断”的顺序打开模块中断;
- (2) 在 isr.c 文件中, 编写中断服务程序;
- (3) 在 vectors.s 文件中, 修改中断向量表;

ColdFire 单片机开始运行后, 系统状态寄存器 SR 中 16、17 和 18 这三位的默认值都是“1”, 即关闭所有中断, 所以要使用中断, 必须更改这三位的值。它就相当于一个总闸, 如果总闸不开, 所有中断都不可能发生。操作状态寄存器 SR, 需要汇编指令来实现:

```

move.w #0x2700,sr    //关总中断
move.w #0x2000,sr    //开总中断

```

为了方便代码移植, 在 isr.h 文件中做了如下定义:

```

#define DisableInterrupts() asm(move.w #0x2700,sr)    //关总中断
#define EnableInterrupts()  asm(move.w #0x2000,sr)    //开总中断

```

下面以 UART0 接收中断为例，实现以下功能：主程序每隔 1 秒让 UART0 模块向 PC 机发送用于握手的字符串“Hello! World!”；同时，串口等待接收从 PC 机发来的数据，一旦接到数据，马上将该数据回发给 PC 机。串口接收程序使用中断来实现，中断处理程序执行完毕后，又回到主程序，继续每隔 1 秒向 PC 机发送一遍“Hello! World!”。

1. 主函数文件（main.c）

```
//-----*
// 工 程 名: UartIntRe                                     *
// 硬件连接: 接PC机串口                                   *
// 程序描述: 使用UART0和PC机通信                         *
//           定时发送“Hello! World!”到PC机, 1秒1次      *
//           同时等待接收PC机发送的数据, 当接收到了来自PC机的数据后立即回发*
// 目    的: 初步掌握利用中断方式进行串行通信的基本知识 *
// 说    明: 波特率为38400, 使用UART0口                 *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
#include "includes.h" //头文件
uint8 msg[14]= "Hello! World! ";
int main(void)
{
    //1. 关总中断
    DisableInterrupts();
    //2. UART初始化
    //使用UART0模块, 使用系统时钟作为UART的时钟源
    //波特率为38400, 初始化时开接收中断
    UartInit(0, SYSTEM_CLOCK, 38400, 1);
    //3. 开总中断
    EnableInterrupts();
    //4. 主循环
    while (1)
    {
        Delay(100); //延时1.0s
        UartSendN(0, 14, msg);
    }
    return 0;
}
```

2. 中断处理函数文件（isr.c）

在程序中，一般使用“__interrupt__”来标识函数是中断或异常处理程序。以下简要给出了与本工程相关的中断函数定义，要了解整个文件，请查看光盘。

```
//-----*
//文 件 名: isr.c （中断处理函数文件）                 *
//说    明: 中断处理函数                               *
//本文件包含:                                           *
//           (1) asm_exception_handler: 中断处理程序      *
//           (2) irq_handler:           中断处理程序      *
//           (3) Uart0_handler:         Uart0中断处理程序  *
//-----*
```

```

//硬件连接：无 *
//-----*
.....
//Uart0中断处理程序
__interrupt__
void Uart0_handler(void)
{
    uint8 reBuff[1];
    DisableInterrupts(); //关总中断
    if(UartReN(0, 1, reBuff)==0)
        UartSendN(0, 1, reBuff);
    EnableInterrupts(); //开总中断
}

```

3. 中断向量表文件（vectors.s）

编写好中断处理函数后，接下来需要修改中断向量表。在 **vectors.s** 文件中，找到对应位置，把定义的中断服务程序名写入。这里要注意的是，函数名前必须加下划线“_”，这是 CW 环境对汇编程序代码的要求。以下简要给出了与本工程相关的中断向量表内容，要了解整个文件，请查看光盘。

```

//-----*
// 文件名： vectors.s *
// 说 明： MCF52233中断向量表 *
//-----*
//外部标记符引用声明
.extern __SP_INIT
.extern _asm_startmeup
.extern _asm_exception_handler
.extern _irq_handler
.extern _Uart0_handler
//全局标记符定义
.global VECTOR_TABLE
.global _VECTOR_TABLE
.global start
.text
//-----*
//中断向量表 *
//-----*
VECTOR_TABLE:
_VECTOR_TABLE:
INITSP:    .long    __SP_INIT           // SP的初始化值(Initial SP)
INITPC:    .long    start               // PC的初始化值(Initial PC)
vector02:  .long    _asm_exception_handler // 存取错误(Access Error)
vector03:  .long    _asm_exception_handler // 地址错误(Address Error)
.....
vector4D:  .long    _Uart0_handler       // UART0中断(UART 0 Rx/Tx)
.....
//复位中断处理-----*

```

```
//功 能：系统复位时，进入复位向量指向的首条要执行的指令      *  
//-----*  
start:  
    move.w #0x2700, sr      //初始化系统状态寄存器SR  
    jmp     _asm_startmeup  
    .end
```

4.6 MCF52233的四通道DMA控制器

4.6.1 DMA概述

DMA 控制器的作用是在大容量存储设备和内存之间传送批量数据。通常 I/O 外设会通过中断状态位，来告知系统数据是否已准备好或是否有对数据的请求。为了及时地响应这些事件，处理器经常会被中断。在该过程中 CPU 挂起当前进程，保存设备状态，服务于此事件，清除 I/O 请求标志，然后返回到被挂起的主进程。为避免每次事件都去中断 CPU，ColdFire 可使用 DMA 通道来响应 I/O 外设请求，实现对请求数据的传送，而不用每次都去中断 CPU 内核。

由于 I/O 数据的传输由 DMA 控制器完成，这就增加了 CPU 的带宽，使 CPU 在有限的时间内可以执行更多更复杂的算法和任务。某些 ColdFire 微控制器集成了具有 16 个通道的加强型直接内存存取（enhanced Direct Memory Access，eDMA）引擎，而其他的 ColdFire 微控制器则集成了带有 4 通道的 DMA 控制器。MCF52233 使用的就是 4 通道的 DMA 控制器。

4.6.2 MCF52233四通道DMA特点

MCF52233 的四通道 DMA 控制器可高效完成系统存储器到 I/O 外设的直接数据传送操作，且有相对较高的速度，很少或不需要 CPU 干预，4 个通道都能传送字节、字、长字数据或突发传送 16 字节数据。

四个通道都可由软件通过片上硬件的多路选择器来指定其归属，有 10 个 I/O 外设可以被指定给每个 DMA 通道。这种路由设定的功能使得通道分配非常灵活，每个 DMA 通道都可分配给 3 个 UART 接收器/发送器或 4 个 DMA 定时器。DMA 通道可编程设置总线传送带宽，也可配置成周期窃取。

MCF52233 的四通道 DMA 控制器具有以下特点：4 个独立的可编程 DMA 控制通道；源或目的访问的地址自动对齐；双地址传送；每个通道都有外部请求；能进行 8、16 和 32 位的数据传送或用 16 字节缓冲器进行 128 位的突发数据传送；具有连续传送或周期窃取传送两种模式；源数据或目的数据具有独立的传送带宽；具有独立的源地址和目的地址寄存器；具有源地址和目的地址的模块寻址能力；自动通道链接。

4.6.3 MCF52233的DMA控制器编程结构

图 4-11 的框图描述了四通道 DMA 控制器的编程结构，每个通道都有五个可编程寄存器，用于实现在特定的 MCU 存储器映射地址之间的数据传送。这些寄存器包括：

- (1) 源地址寄存器（Source Address Register, SAR）
- (2) 目的地址寄存器（Destination Address Register, DAR）
- (3) 字节计数寄存器（Byte Count Register, BCR）
- (4) DMA 状态寄存器（DMA Status Register, DSR）
- (5) DMA 控制寄存器（DMA Control Register, DCR）

另外，还有一个用于指定 4 个通道与请求源的逻辑连接关系的 DMA 请求控制寄存器（DMA Request Control Register, DMAREQC）。表 4-5 列出了 DMA 请求控制寄存器和通道 0 各寄存器的地址映射。

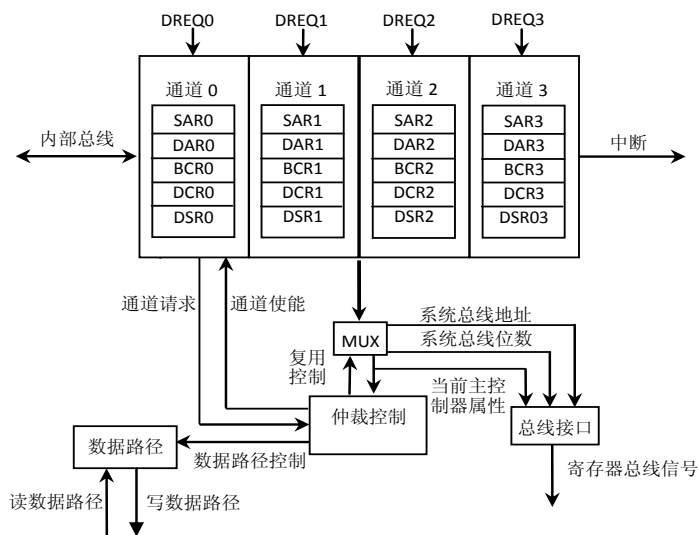


图4-11 DMA功能模块框图

对于其他通道，同名寄存器的地址相对于通道 0 的偏移分别为 0x10、0x20 和 0x30，例如，SAR0 的地址为 0x00_0100，则 SAR1、SAR2 和 SAR3 的地址分别为 0x00_0110、0x00_0120 和 0x00_0130。

表 4-5 DMA 控制器模块寄存器地址映射

DMA通道	IPSBAR偏移	[31:24]	[23:16]	[15:8]	[7:0]
—	0x00_0014	DMA请求控制寄存器（DMAREQC）			
0	0x00_0100	源地址寄存器 0（SAR0）			
	0x00_0104	目的地址寄存器 0（DAR0）			
	0x00_0108	字节计数寄存器 0（BCR0）和DMA状态寄存器 0（DSR0）			
	0x00_010C	DMA控制寄存器 0（DCR0）			

1. 源地址寄存器 (SARn) 和目的地址寄存器 (DARn)

可读/写, 包含 32 位源和目的存储器地址或者输入/输出外设地址。

2. 字节计数寄存器 (BCRn)

与 DMA 状态寄存器共用相同的地址, 低 24 位是字节计数寄存器字段, 高 8 位是 DMA 状态寄存器。该寄存器用于写入需要传送的字节数。在字节、字、长字或线性访问时, BCRn 在每次传送后分别减少 1、2、4 或 16。当字节计数器减少到 0 时, DMA 状态寄存器 (DSR) 中的 DONE 状态位置位, 这表示所有传送都成功完成; 如果传送完成中断允许, 则 DMA 向 CPU 内核发出中断请求。

3. DMA 状态寄存器 (DSRn)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	—	CE	BES	BED		REQ	BSY	DONE
复位	0	0	0	0	0	0	0	0

DMA 状态是 8 位字段, 它在 BCR 的最高字节中。在 DMA 通道传送操作中, 这个寄存器中的状态位用于表示成功传送或有什么地方出错。DSR 每个状态位简单描述如下:

D6—CE, 配置错误状态。当 BCR、SAR 与 DAR 与请求的传送大小不符时, 或当 DMA 处于传送操作的激活状态且 BCR=0 时, DMA 将检测到该错误条件。向 DSR.DONE 写 “1” 可清除 CE。

D5—BES, 源总线错误。在传输中进行读操作时, 如果出现总线错误, 则 DMA 通道传送被取消, 且该位被置位。

D4—BED, 目标总线错误。在传输中进行写操作时, 如果出现总线错误, 则 DMA 通道传送被取消, 且该位被置位。

D2—REQ, 请求。如果 I/O 外设被编程产生 DMA 请求, 而 DMA 通道还没有分配给 I/O 设备, REQ 将被置位。此时, I/O 外设请求连续挂起, 直到 DMA 通道允许并且 I/O 外设得到服务。

D1—BSY, 忙。当前通道处于数据传送操作期间, 该位将被置位。

D0—DONE, 传送完成。当由字节计数器 (BCR) 确定的全部字节传输完成时, 该状态位被置位。如果传送完成中断允许, 则 DMA 通道将向 CPU 发出中断信号。对于编程者来说, 在退出中断服务程序之前, 中断处理过程应在此位写逻辑 “1” 以清零 DONE 位。清除 DONE 位, 将会自动清零 DSR 中的所有状态位。

4. DMA 控制寄存器 (DCRn)

数据位	D31	D30	D29	D28	D27~D25	D24	D23	D22	D21~D20	D19	D18~D17	D16
定义	INT	EEXT	CS	AA	BWC	保留	SINC	SSIZE	DINC	DSIZE	START	
复位	0		0		0	0	0	0	0	0	0	0

该 DMA 控制寄存器可以控制 DMA 通道的所有操作, 各位的简要描述如下:

D31—INT, 传送完成时触发中断。置位该位, 将使得 DMA 在完成请求传送或出现错误时, 向 CPU 发出中断请求。例如, 使用带 DMA 的 UART 来接收批量数据时, 需要

置位该位，以使得传送完成时，DMA 能告知 CPU 已经完成了批量接收。

D30—EEXT，外部请求使能。通过 EEXT 控制位使外部请求得到允许并启动外部传送操作。对编程者来说，当需要将 DMA 通道分配给 I/O 外设且使用 I/O 外设寄存器作为传送源时，该位应被置位，以使得 DMA 能监听外设的请求。另外，若要使用 DMA 通道的软件触发来传送大量数据时，则需置位 START 位。

D29—CS，周期挪用。置位该位 DMA 将启用周期挪用（窃取）模式，否则启用连续传送模式。当 DMA 通道被编程为连续执行读写传送时，直到 BCR 减到 0，DMA 才释放总线的使用权。而周期挪用操作每次请求完成 1 次读/写传送后，马上释放总线使用权，这样操作的优点是，DMA 访问的存储器资源每次传送后是空闲的，因此允许其他总线主控制器访问这些资源；缺点是，传送大量数据时需要占用较长的数据传送时间。

D28—AA，自动对齐。自动对齐允许 DMA 以最佳长度传送数据块，该最佳长度基于地址、字节计数和可编程长度。

D27~D25—BWC[2:0]，带宽控制位域。在传送大量数据时，这 3 位字段控制 DMA 通道可用的带宽。如果不使用带宽控制（BWC=000），则通道将连续产生所需数量的读/写周期，直到字节计数器溢出（BCR=0）。为了减少其他总线主控制器连接相同资源的等待时间，在完成数据传送后，DMA 通道可强制等待，以便内部总线仲裁器把总线控制权交给其他发出请求的主控制器。当 BWC 字段为非零时，在释放总线控制权前，DMA 通道将以 BWC 中设定的带宽传送大量数据。表 4-6 所列为 BWC 取值的定义。

表 4-6 带宽控制编码

BWC	每块千字节数
000	DMA有优先级，在传输完成前释放总线控制权
001	在16KB后释放总线控制权
010	在32KB后释放总线控制权
011	在64KB后释放总线控制权
100	在128KB后释放总线控制权
101	在256KB后释放总线控制权
110	在512KB后释放总线控制权
111	在1024KB后释放总线控制权

D22~D20—SINC&DINC，源地址和目的地址指针增量允许位。每次成功传送后，这两位控制是否有源/目的地址增量。如果编程为有增量，则 DMA 按照源/目的数据单位长度定义的值（通过对 SSIZE&DSIZE 位域来设置）增加源/目的地址寄存器。在访问内存区域时，源/目的地址增量常被使用；但当访问 I/O 设备的数据寄存器或 FIFO 时，源和目标指针就不该设有增量。

D19~D17—SSIZE&DSIZE，源和目的单位长度。这两个字段决定了在每个源/目的总线周期内传送的数据单位的长度。在每次请求中，SSIZE 或 DSIZE 可编程通过 DMA 通道传送字节、字、长字或者 16 字节数据。00：字；01：字节；10：长字；11：16 字节。

D16—START，开始传送。通过软件置位该位后，DMA 通道立即得到总线仲裁，并开启从源地址到目的地址的传送。通常，若需要把外设 I/O 地址作为传送源时，该位应被清零，而外部请求使能位 EEXT 则被置位，以使得 DMA 能实时监听外设的请求。

5. DMA 请求源

有 10 种可能的请求（4 个 DMA 定时器和 6 个 UART），而 DMA 控制器只有 4 个通道，每个请求都可以逻辑地连接到 4 个通道之一。DMA 请求控制寄存器 DMAREQC 提供高效的软件路由表，来实现从 10 个 DMA 请求信号到 4 路通道的逻辑连接。DMAC3 控制 DMA 通道 3，DMAC2 控制 DMA 通道 2，依此类推。表 4-7 所示为请求与 DMA 通道之间的逻辑连接关系。DMAREQC 的地址编码表如表 4-5 所示。

表 4-7 DMAREQC[DMAn]编码表

编 码	请求源	编 码	请求源
0100	DMA 定时器 0	1001	UART1 接收器
0101	DMA 定时器 1	1010	UART2 接收器
0110	DMA 定时器 2	1100	UART0 发送器
0111	DMA 定时器 3	1101	UART1 发送器
1000	UART0 接收器	1110	UART2 发送器

4.7 MCF52233的DMA编程要点

DMA 模块支持内部请求和外部请求，分别由 DMA 控制寄存器 DCR 的 START 和 EEXT 位控制。对于内部请求，设置 DCRn 的 START 位，使得 DMA 通道立即进行总线仲裁，以便于启动所请求的传送。例如，从外部存储器（如 DRAM）到内部存储器传送成块数据就属这种情况。

而对于外部请求来说，应用软件编程设定后，在开始传送操作之前，DMA 通道持续等待，直到片上外设发出请求信号。举个外部请求的例子，片上外设（如 UART）当准备好传送数据时，就向 DMA 控制器发送传送请求，DMA 将响应并按照应用软件的编程设定来进行传送。在数据传送开启前，DMA 控制器将验证控制寄存器的 DCRn[SSIZ:DSIZ] 位域，如果它们与源地址和目的地址的内容不相符，配置错误位 DSRn 的 CE 将被置位。如果检测到未对齐，DMA 则不会执行任何传送操作，并根据 DMA 控制寄存器 DCR 的配置发出中断请求。

根据 DMA 控制寄存器（DCRn）中设定的数据传送单位长度，读/写传送将从源地址读数据，并将它们写到目的地址。在成功完成传送操作后，源地址和目的地址寄存器（SARn 和 DARn）根据 DCRn 中编程设置来实现自增。而字节计数寄存器 BCRn 则根据请求数据传送的单位长度来减 1、2、4 或 16 字节。

在周期挪用模式 DCRn 的 SC=1 下，完成从源地址到目的地址的数据传送后，DMA 通道立即释放总线，而在连续模式 DCRn 的 SC=0 下，DMA 通道连续传送字节数据，直到字节计数寄存器 BCR 为零，整个传送操作成功完成之时。DMA 用置位 DONE 位来告知传送操作成功完成；这时如果软件允许，DMA 可向 CPU 产生传送完成中断。

4.7.1 DMA初始化

要使用 DMA 控制器，必须先对它进行初始化操作。对于通常的 DMA 初始化操作来

说，主要有以下步骤：

(1) 设置总线控制权限寄存器 MPR (Master Privilege Register) 为 0x04，以允许 DMA 控制器访问内部系统总线。MPR 相对内部外设系统基地寄存器 (IPSBAR) 的地址偏移为 0x0020，要了解更多信息，请参考随书光盘中的 MCF5223x 芯片手册。

(2) 设置 DMA 请求控制寄存器 (DMAREQC)，将 DMA 具体的通道逻辑地连接到 10 个 I/O 请求源的某一个上。例如，将 UART0 接收器和发送器连接到 DMA 通道 3 和 DMA 通道 4，则需写二进制值“1000”到 DMAC3，写二进制值“1100”到 DMAC2。

(3) 设置相关模块的中断屏蔽寄存器，禁止常规中断，以允许 DMA 请求，如 UART 模块的中断屏蔽寄存器 (UIMR)。例如，若要从 UART0 对 DMA 产生传送请求，需通过设置 UART0 中断屏蔽寄存器 (UIMR0)，将 UIMR0 的 TXRDY 位清零，来禁止 UART0 的中断；类似地，要从 UART0 产生接收 DMA 请求，则应将 UIMR0 的 RXRDY 位清零。这样设置的作用是，在 UART 产生发送和接收 DMA 请求时，可以防止 TXRDY 和 RXRDY 产生对 CPU 的中断。

(4) 设置中断控制模块 ICM (Interrupt Controller Module) 里的中断控制寄存器 ICR (Interrupt Control Register) 和中断屏蔽寄存器 (Interrupt Mask Register) 的低字段 IMRL 为适当的值，以控制是否使能具体的请求源所对应的 DMA 传送结束中断。

4.7.2 DMA 启动序列与重启序列

1. DMA 启动序列

对于相对简单的任务，设置 DMA 模块需要以下步骤：

- (1) 向 SARn 写入 32 位源地址。
- (2) 向 DARn 写入 32 位目的地址。
- (3) 向 BCRn 写入要传送的字节数量。

(4) 通过写 DCRn，设置传送过程中的相关属性。若要用软件来启动传送，则需置位 START 位，这时 DMA 传送马上开始；否则置位 DCRn 中的 EEXT 位来启动监听片上 I/O 请求。注意，对于 UART 请求，DMA 通道应该配置为周期挪用模式，源和目的数据的单位长度应该为单字节，这样，每次 UART DMA 请求就只需进行单字节传送。

特别注意，在写 DCRn 时，如果置位了 START 位或 EEXT 位，DMA 控制器将立即开始工作，换言之，若 START 或 EEXT 位处于清零的状态，则 DMA 将不会理会前面所说步骤的任何设置。建议在编程时，先配置好其他标志位，最后再来设置它们。

2. DMA 重启序列

通常情况下，在对 DMA 控制器以及相关模块进行 DMA 传送初始化后，使用软件直接运行 DMA 启动序列，就可以启动正常的 DMA 的传输了。但是，每次启动传送都去执行完整的启动序列是没有必要的。

编程者可以根据实际需求来决定重启序列应该包含哪些步骤，最简单的办法就是把启动序列中的四个步骤全部囊括。需要注意的是，不管怎样，要重启 DMA 传送，必须要有

对 DCRn 中 START 位或 EEXT 位的置位操作，其他的设置根据具体应用而定。

4.8 基于构件的带DMA的UART编程实例

在本章的 4.4 节已经详细介绍了基于构件的 UART 编程实例，本节主要在 4.4 节的基础上介绍带 DMA 的 UART 编程实例。带 DMA 的 UART 具有接收和发送初始化、启动 DMA 接收和发送以及重新启动 DMA 接收这五种基本操作。按照构件化的思想，可将它们封装成五个独立的功能函数，初始化过程主要完成 DMA 通道供给 UART 模块的发送器和接收器使用的一些相关设置，启动 DMA 接收和发送功能函数则完成利用 DMA 进行数据传输需要的一些规则，重新启动 DMA 接收功能函数则完成对 DMA 数据接收规则的重新设置。

4.8.1 DMA构件的函数原型设计

参照本章 4.8 节对 DMA 初始化、DMA 启动序列和 DMA 重新启动的描述，本章的编程实例设计了 5 个基本的功能函数，以下是各功能函数的原型设计：

(1) DMA 发送和接收初始化

```
void DmaUartTxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt)
```

```
void DmaUartRxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt)
```

(2) DMA 发送和接收启动序列

```
void DmaUartTxStrt(uint8 dmaCh, uint8 reqNo, uint8* seBuff, uint16 buffSize)
```

```
void DmaUartRxStrt(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize)
```

(3) DMA 接收重新启动序列

```
void DmaUartRxNext(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize)
```

4.8.2 DMA构件的头文件

与使用 DMA 功能的 UART 通信子函数相关的文件有头文件 DMA.h 和 DMA.c。

头文件 DMA.h 中的内容可分为两个主要的部分，它们分别是 5 个函数原型的声明和外设模块寄存器相关信息的定义。前者给出了使用 DMA 的 UART 构件对上层构件或软件所提供的接口函数，而后者则指明了本“元构件”与具体硬件相关的信息。

DMA.h 中含有相关的 DMA 寄存器和标志位定义，以及使用 DMA 功能的 UART 需要配置的相关函数声明。以初始化接收函数 DmaUartRxInit 为例，要设置的 DMA 通道号、UART 接收器的编号、初始化时是否开启某些中断，都被设计为函数参数。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。文件还给出了必要的硬件相关信息，当要把该构件移植到其他芯片时，就必须检查并修改这些信息。

```

//-----*
//文件名: DMA.h (DMA头文件) *
//-----*
#ifndef DMA_H
#define DMA_H
//1. 对外引用
#include "type.h" //类型定义头文件
extern uint8 __IPSBAR[]; //引用外部标号
//2. 外用通信函数声明
//-----*
//函数名: DmaUartRxInit *
//功 能: 初始化UARTx接收器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//        uint8 reqNo: UART接收器编号, 取值为0, 1, 2; *
//        uint8 enInt: 初始化后是否使能本模块中断(1:开中断, 0:关中断) *
//返 回: 无 *
//实 例: DmaUartRxInit(0, 0, 0)表示将UART0接收器连接到DMA0通道, 且初始禁中断 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt);

//-----*
//函数名: DmaUartRxStrt *
//功 能: 启动UARTx接收器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//        uint8 reqNo: UART接收器编号, 取值为0, 1, 2; *
//        uint8* reBuff: 目的地址; *
//        uint16 buffSize: 接收数据的个数; *
//返 回: 无 *
//实 例: DmaUartRxStrt(0, 0, (uint8 *)reRingBuff, 10)表示将UART0接收器连接到 *
//        DMA0通道且把接收到的10个数据传送到目的地址为reRingBuff的内存区域 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxStrt(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize);

//-----*
//函数名: DmaUartRxNext *
//功 能: 重启UARTx模块的DMAy通道。x代表0, 1, 2; y代表0, 1, 2, 3 *
//参 数: uint8 dmaCh: 第dmaCh个DMA通道。其中dmaCh取值为0, 1, 2, 3 *
//        uint8 reqNo: UART模块编号, 取值为0, 1, 2 *
//返 回: 无 *
//实 例: DmaUartRxNext(0, 0, (uint8 *)reRingBuff, 10)表示将UART0接收器连接到 *
//        DMA0通道, 且把接收到的10个数据传送到目的地址为reRingBuff的内存区域 *
//        中 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxNext(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize);

//-----*

```

```

//函数名: DmaUartTxInit *
//功 能: 初始化UARTx发送器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//      uint8 reqNo: UART发送器编号, 取值为0, 1, 2; *
//      uint8 enInt: 初始化后是否使能本模块中断(1:开中断, 0:关中断) *
//返 回: 无 *
//实 例: DmaUartTxInit(0, 0, 0)表示将UART0接收器连接到DMA0通道且初始时禁中断*
//说 明: 参数不能越界 *
//-----*
void DmaUartTxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt);

//-----*
//函数名: DmaUartTxStrt *
//功 能: 启动UARTx发送器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//      uint8 reqNo: UART发送器编号, 取值为0, 1, 2; *
//      uint8* seBuff: 首地址; *
//      uint16 buffSize: 发送数据的个数; *
//返 回: 无 *
//实 例: DmaUartTxStrt(0, 0, (uint8*)reRingBuff, 10)表示将UART0发送器连接到 *
//      DMA0通道且从首地址为reRingBuff的内存区写10个数据到发送缓冲寄存器 *
//说 明: 参数不能越界 *
//-----*
void DmaUartTxStrt(uint8 dmaCh, uint8 reqNo, uint8* seBuff, uint16 buffSize);

//-----*
//函数名: DmaReady *
//功 能: 检查具体DMA通道当前是否正在传送 *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//返 回: 1:完成传送0:正在传送 *
//实 例: DmaReady(2)表示检查DMA2通道当前是否正在传送 *
//说 明: 参数不能越界 *
//-----*
uint8 DmaReady(uint8 dmaCh);

//3. 模块相关的宏定义
//标志位掩码的定义
#define MCF_INTC_MASK(x) (0x0200<<(x)) //DMA中断源(9-12)掩码
#define MCF_INTC_ICR_IL(x) (((x)&0x07)<<3) //中断级别掩码
#define MCF_INTC_ICR_IP(x) (((x)&0x07)<<0) //中断优先级掩码
#define MCF_DMA_DSR_DONE (0x01) //DSR清零掩码
#define MCF_DMA_DCR_START (0x10000) //DSR开始传输掩码
#define MCF_DMA_DCR_DSIZE(x) (((x)&0x3)<<0x11) //DCR目的数据大小掩码
#define MCF_DMA_DCR_DINC (0x80000) //DCR目的地址递增掩码
#define MCF_DMA_DCR_SSIZE(x) (((x)&0x3)<<0x14) //DCR源数据大小掩码
#define MCF_DMA_DCR_SINC (0x400000) //DCR源地址递增掩码
#define MCF_DMA_DCR_BWC(x) (((x)&0x7)<<0x19) //DCR传输块的字节数掩码
#define MCF_DMA_DCR_AA (0x10000000) //DCR自动对齐掩码
#define MCF_DMA_DCR_CS (0x20000000) //DCR允许周期挪用掩码

```



```

#define MCF_DMA_DCR_EEXT      (0x40000000)    //DCR外部请求使能掩码
#define MCF_DMA_DCR_INT      (0x80000000)    //DCR传输完成中断掩码
#define MCF_SCM_DMAC(x)      ((x)&0xF)        //DCR中请求源掩码
//模块相关寄存器的定义
//总线主控制权限寄存器
#define MCF_SCM_MPR          (*(vuint8 *)(&__IPSBAR[0x20]))
//DMA请求控制寄存器
#define MCF_SCM_DMAREQC      (*(vuint32*)(&__IPSBAR[0x14]))
//UART中断状态/掩码寄存器
#define MCF_UART_UIMR(x)    (*(vuint8 *)(&__IPSBAR[0x000214+((x)*0x040)]))
//中断控制寄存器
#define MCF_INTC0_ICR(x)    (*(vuint8 *)(&__IPSBAR[0x000C49+(x)]))
//中断屏蔽寄存器
#define MCF_INTC0_IMRL      (*(vuint32*)(&__IPSBAR[0x000C0C]))
//UART接收数据寄存器
#define MCF_DMA_URB(x)      (*(vuint8 *)(&__IPSBAR[0x20C+((x)*0x040)]))
//UART发送接收数据寄存器
#define MCF_DMA_UTB(x)      (*(vuint8 *)(&__IPSBAR[0x20C+((x)*0x040)]))
//DMA源地址寄存器
#define MCF_DMA_SAR(x)      (*(vuint32*)(&__IPSBAR[0x100 + ((x)*0x10)]))
//DMA目的地址寄存器
#define MCF_DMA_DAR(x)      (*(vuint32*)(&__IPSBAR[0x104 + ((x)*0x10)]))
//DMA状态寄存器
#define MCF_DMA_DSR(x)      (*(vuint8 *)(&__IPSBAR[0x108 + ((x)*0x10)]))
//DMA字节计数寄存器
#define MCF_DMA_BCR(x)      (*(vuint32*)(&__IPSBAR[0x108 + ((x)*0x10)]))
//DMA控制寄存器
#define MCF_DMA_DCR(x)      (*(vuint32*)(&__IPSBAR[0x10C + ((x)*0x10)]))
#endif

```

4.8.3 DMA构件的源程序文件

1. 使用DMA功能的UART模块的初始化功能函数: DmaUartTxInit 和 DmaUartRxInit

```

//-----*
//函数名: DmaUartTxInit                                *
//功 能: 初始化UARTx发送器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//      uint8 reqNo: UART发送器编号, 取值为0, 1, 2; *
//      uint8 enInt: 初始化后是否使能本模块中断(1:开中断, 0:关中断) *
//返 回: 无 *
//实 例: DmaUartRxInit(0, 0, 0) 表示将UART0接收器连接到DMA0通道且初始时禁中断*
//说 明: 参数不能越界 *
//-----*
void DmaUartTxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt)
{
    //1. 使能DMA控制器

```

```

MCF_SCM_MPR      |= 0x04;
//2. 在DMA请求控制寄存器中使用DMA(dmaCh)通道的UART(reqNo)发送器请求
MCF_SCM_DMAREQC |= MCF_SCM_DMAC(reqNo+0x0C)<<(dmaCh*4);
//3. 设置UART中断屏蔽寄存器中的中断使能位
MCF_UART_UIMR(reqNo) = 0x02;
//4. 设置DMA中断的级别和优先级(DMA通道(0-3)中断源是9-12)
MCF_INTC0_ICR(dmaCh) = (MCF_INTC_ICR_IL(0x04) |
                        MCF_INTC_ICR_IP(0x03));

//5. 使能DMA(dmaCh)中断
if(1==enInt)
{ //使能DMA(dmaCh)中断
    MCF_INTC0_IMRL  &= ~MCF_INTC_MASK(dmaCh);
}
else
{ //禁止DMA(dmaCh)中断
    MCF_INTC0_IMRL  |=  MCF_INTC_MASK(dmaCh);
}
}
//-----*
//函数名: DmaUartRxInit                                *
//功 能: 初始化UARTx接收器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//        uint8 reqNo: UART接收器编号, 取值为0, 1, 2; *
//        uint8 enInt: 初始化后是否使能本模块中断(1:开中断, 其他:关中断) *
//返 回: 无 *
//实 例: DmaUartRxInit(0, 0, 0)表示将UART0接收器连接到DMA0通道, 且初始禁中断 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxInit(uint8 dmaCh, uint8 reqNo, uint8 enInt)
{
    //1. 使能DMA控制器
    MCF_SCM_MPR      |= 0x04;
    //2. 在DMA请求控制寄存器中使用DMA(dmaCh)通道的UART(reqNo)接收器请求
    MCF_SCM_DMAREQC |= MCF_SCM_DMAC(reqNo+0x08)<<(dmaCh*4);
    //3. 设置UART中断屏蔽寄存器中的中断使能位
    MCF_UART_UIMR(reqNo) = 0x02;
    //4. 设置DMA中断的级别和优先级(DMA通道(0-3)中断源是9-12)
    MCF_INTC0_ICR(dmaCh) = (MCF_INTC_ICR_IL(0x04) |
                            MCF_INTC_ICR_IP(0x03));

    //5. 使能DMA(dmaCh)中断
    if(1==enInt)
    { //使能DMA(dmaCh)中断
        MCF_INTC0_IMRL  &= ~MCF_INTC_MASK(dmaCh);
    }
    else
    { //禁止DMA(dmaCh)中断
        MCF_INTC0_IMRL  |=  MCF_INTC_MASK(dmaCh);
    }
}
}

```

2. 使用DMA功能的UART模块发送启动功能函数: DmaUartTxStrt

```
//-----*
//函数名: DmaUartTxStrt *
//功 能: 启动UARTx发送器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//        uint8 reqNo: UART发送器编号, 取值为0, 1, 2; *
//        uint8* seBuff: 首地址; *
//        uint16 buffSize: 发送数据的个数; *
//返 回: 无 *
//实 例: DmaUartTxStrt(0, 0, (uint8*)reRingBuff, 10) 表示将UART0发送器连接到 *
//        DMA0通道且从首地址为reRingBuff的内存区写10个数据到发送缓冲寄存器 *
//说 明: 参数不能越界 *
//-----*
void DmaUartTxStrt(uint8 dmaCh, uint8 reqNo, uint8* seBuff, uint16 buffSize)
{
    //1. 对DMA状态寄存器DSR清零
    MCF_DMA_DSR(dmaCh) = MCF_DMA_DSR_DONE;
    //2. 设置源地址为seBuff
    MCF_DMA_SAR(dmaCh) = (unsigned long) seBuff;
    //3. 设置目的地址为发送缓冲区的地址
    MCF_DMA_DAR(dmaCh) = (unsigned long) & MCF_DMA_UTB(reqNo);
    //4. 设置DMA的字节计数寄存器为buffSize
    MCF_DMA_BCR(dmaCh) = buffSize;
    //5. 设置DMA的控制寄存器: 允许外部请求、使能周期窃取、
    //    自动对齐、源地址加1、源数据和目的数据为1字节
    MCF_DMA_DCR(dmaCh) = (MCF_DMA_DCR_EEXT |
                          MCF_DMA_DCR_CS |
                          MCF_DMA_DCR_AA |
                          MCF_DMA_DCR_SINC |
                          MCF_DMA_DCR_SSIZE(01) |
                          MCF_DMA_DCR_DSIZE(01));
}
```

3. 使用DMA功能的UART模块接收启动功能函数: DmaUartRxStrt

```
//-----*
//函数名: DmaUartRxStrt *
//功 能: 启动UARTx接收器的DMAy通道(x取0, 1, 2; y取0, 1, 2, 3) *
//参 数: uint8 dmaCh: DMA 通道编号, 取值为0, 1, 2, 3; *
//        uint8 reqNo: UART接收器编号, 取值为0, 1, 2; *
//        uint8* reBuff: 目的地址; *
//        uint16 buffSize: 接收数据的个数; *
//返 回: 无 *
//实 例: DmaUartRxStrt(0, 0, (uint8 *)reRingBuff, 10) 表示将UART0接收器连接到 *
//        DMA0通道且把接收到的10个数据传送到目的地址为reRingBuff的内存区域 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxStrt(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize)
```

```

{
    //1. 对DMA状态寄存器DSR清零
    MCF_DMA_DSR(dmaCh) = MCF_DMA_DSR_DONE;
    //2. 设置源地址为UARTx接收数据寄存器的地址
    MCF_DMA_SAR(dmaCh) = (unsigned long) & MCF_DMA_URB(reqNo);
    //3. 设置目的地址为reBuff
    MCF_DMA_DAR(dmaCh) = (unsigned long) reBuff;
    //4. 设置DMA的字节计数寄存器为buffSize
    MCF_DMA_BCR(dmaCh) = buffSize;
    //5. 设置DMA的控制寄存器:传输完成产生中断、允许外部请求、使能周期窃取、
    // 自动对齐、源地址加1、源数据和目的数据为1字节
    MCF_DMA_DCR(dmaCh) = (MCF_DMA_DCR_INT |
                          MCF_DMA_DCR_EEXT |
                          MCF_DMA_DCR_CS |
                          MCF_DMA_DCR_AA |
                          MCF_DMA_DCR_DINC |
                          MCF_DMA_DCR_SSIZE(01) |
                          MCF_DMA_DCR_DSIZE(01));
}

```

4. 使用DMA的UART模块接收重启功能函数: DmaUartRxNext

```

//-----*
//函数名: DmaUartRxNext *
//功 能: 重启UARTx模块的DMAy通道。x代表0, 1, 2; y代表0, 1, 2, 3 *
//参 数: uint8 dmaCh: 第dmaCh个DMA通道。其中dmaCh取值为0, 1, 2, 3 *
//        uint8 reqNo: UART模块编号, 取值为0, 1, 2 *
//返 回: 无 *
//实 例: DmaUartRxNext(0, 0, (uint8 *)reRingBuff, 10)表示将UART0接收器连接到 *
//        DMA0通道, 且把接收到的10个数据传送到目的地址为reRingBuff的内存区域 *
//        中 *
//说 明: 参数不能越界 *
//-----*
void DmaUartRxNext(uint8 dmaCh, uint8 reqNo, uint8* reBuff, uint16 buffSize)
{
    //1. 对DMA状态寄存器DSR清零
    MCF_DMA_DSR(dmaCh) = MCF_DMA_DSR_DONE;
    //2. 设置目的地址为reBuff
    MCF_DMA_DAR(dmaCh) = (unsigned long)reBuff;
    //3. 设置DMA的字节计数寄存器为buffSize
    MCF_DMA_BCR(dmaCh) = buffSize;
}

```

4.8.4 DMA方式收发测试实例

1. DMA方式收发测试工程主函数文件main.c

```

//-----*
// 工程名: UartDmaRe *
// 硬件连接: 接PC机串口 *
// 程序描述: 使用带有DMA模块的UART和PC机进行通信. *
//          UART模块首先发送数据到PC机, 然后用DMA请求接收PC机发送的数据并 *
//          用DMA请求发送接收到的数据 *
// 目的: 初步掌握利用DMA请求中断方式进行串行通信的基本知识 *
// 说明: 波特率为115200, 使用UART0口 *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
#include "includes.h" //头文件
int main(void)
{
    uint8 msg0[20] = " UART Hello World!";
    uint8 msg1[20] = " DMA Hello World!";
    uint8 msg2[20] = " Return with Dma: ";
    uint8 msg3[42] = " 苏州大学飞思卡尔嵌入式系统实验室2009.01 ";
    //1. 关总中断
    DisableInterrupts();
    //2. 初始化UART0, 使用系统时钟, 波特率为115200, 初始时不开UART中断
    UartInit(0, SYSTEM_CLOCK, 115200, 0);
    //3. 将DMA3通道指定给UART0接收器, 并开启DMA传送完成中断
    DmaUartRxInit(0x03, 0x00, 0x01);
    //4. 将DMA1通道指定给UART0发送器, 并禁止DMA传送完成中断
    DmaUartTxInit(0x01, 0x00, 0x00);
    //5. 开总中断
    EnableInterrupts();
    //6. 启动DMA3通道的UART0接收传送
    DmaUartRxStrt(0x03, 0x00, (uint8 *)reRingBuff, 15);
    //7. 利用UART0模块发送消息msg0
    UartSendN(0, 18, (uint8*)msg0);
    UartSend1(0, 0x0D); UartSend1(0, 0x0A); //换行符
    //8. 使用DMA1和UART0发送器来发送消息msg1
    DmaUartTxStrt(0x01, 0x00, (uint8*)msg1, 18);
    //9. 使用DMA1和UART0发送器来发送消息msg3
    while(DmaReady(0x01)); //等待传送完成
    UartSend1(0, 0x0D); UartSend1(0, 0x0A); //换行符
    DmaUartTxStrt(0x01, 0x00, (uint8*)msg3, 42);
    //主循环
    while(1)
    {
        //检查接收数组是否有数据更新
        if( 0x01 == dmaReFlag )//有数据
        {
            //发送换行符
            UartSend1(0, 0x0D); UartSend1(0, 0x0A);
            //利用UART0模块发送消息msg2
            UartSendN(0, 18, (uint8*)msg2);
            //使用DMA1和UART0发送器来发送收到的数据
            DmaUartTxStrt(0x01, 0x00, (uint8*)reRingBuff, 15);

```

```

        //接收数据完成标志清零
        dmaReFlag = 0x00;
    }
}
return 0;
}

```

2. 中断函数 (isr.c)

```

//-----*
//文 件 名: isr.c (中断处理函数文件) *
//说 明: 中断处理函数 *
//本文件包含: *
//      (1)asm_exception_handler: 中断处理程序 *
//      (2)irq_handler: 中断处理程序 *
//      (3)DMA3_isr: DMA中断处理程序 *
//硬件连接: 无 *
//-----*
#include "Uart.h"
#include "isr.h"
extern volatile unsigned char reRingBuff[255];
extern uint8 dmaReFlag;

.....

//DMA3通道传送完成中断
__interrupt__
void DMA3_isr(void)
{
    //重启DMA3通道传送
    DmaUartRxNext(0x03, 0x08, (uint8*)reRingBuff, 15);
    //接收数据完成标志置位表示数据已接收到
    dmaReFlag = 0x01;
}

```

4.9 进一步讨论

前几节介绍了带 DMA 控制器的串行通信基本知识和编程方法。本节将讨论串行通信的一些扩展应用。

4.9.1 流控制与 Break 信号

串口通信通常只用到 9 芯串口的 2 (RXD)、3 (TXD) 和 5 (GND) 三个管脚。像这样的简单三线连接,使用的数据线少,通信中可节约通信成本,增加稳定性,能满足绝大多数的需求。但严格来说,当两个系统进行串行通信时,在接收方收完数据之前,应当禁止发送者发送新的数据,这个过程称为流控制 (flow control) 或握手 (shake hands)。流

控制常有软件方式和硬件方式。

软件握手又称为 XON/XOFF 方式，用于接受者和发送者之间无法实现硬件握手的情况。它用专门的字符来启动（XON）或停止（XOFF）数据流。这些字符定义在美国标准信息交换码（ASCII 码）中。软件握手用两个字符来实现流控制，一个代表请求对方暂停传输，另一个代表清除暂停的请求，继续数据传送。通常情况下用 ASCII 码为 0x13 和 0x11 二进制数据来表示，这样，在传输的数据中应禁止包含这两个字符，因为它们会被接收者理解为流控制字符而不是数据。假如发送的只是 ASCII 字符，就不会存在这样的问题，但发送二进制数据的话就会遇到这种情况。常见的解决办法是在发送前对二进制数据进行预处理，将它们用 ASCII 字符来代替。

在 RS-232 标准中，硬件握手使用 RTS（Request To Send，请求发送）和 CTS（Clear To Send，允许发送）两个信号。接收方当准备好接收数据时就将 CTS 信号设置成 0 电平，没有准备好就设置为 1 电平。同样，发送方当准备好发送数据时，就将 RTS 设置成 0 电平，没有准备好就将其设置为 1 电平。硬件流控制使用的是专门的信号设置，而软件要完成同样的任务需要发送和接收额外的数据，所以硬件方式比软件方式速度要快得多。

正常情况下一个数据接收或发送信号保持在高电平，直到一个新数据要传送。如果这个信号跳变到低电平并持续一个较长的时钟周期（通常是 1/4 到 1/2 秒），便说明这是一个 Break（停顿）状态。它可以用来实现收发双方的同步，有时也用作复位或改变通信设备的操作模式，例如调制解调器。

4.9.2 延长串口通信的距离

串口使用一根发送信号线和一根接收信号线来构成共地的传输形式，这种共地传输容易产生共模干扰，抗噪性能弱。本章 4.1 节已经介绍了，RS-232C 最大的传输距离大约是 30m，通信速率一般低于 20Kbps。当然，可以通过降低传输速率的方法来提升传输的距离，但除此之外，还可通过级联信号维持电路来解决这个问题。例如，MAX232 芯片具有两组 232 电平与 TTL 电平转换通路，在传输通路中，信号强度降低后，可先把 232 电平转化成 TTL 电平，再将 TTL 电平回转为 232 电平，这样获得的 232 电平信号又具有了强信号的特性，可保证传输更远的距离。从理论上来说，这种方法可以多次使用，传输距离想要多远就有多远，但这增加了系统开发的成本。

4.9.3 串口的扩展

通用 PC 的 COM 接口有 1 或 2 个，当 PC 机上需要挂载多个串口设备时，这显然不能满足需求，需要串口扩展。同样在以 MCU 为核心的多主机串口通信网络中，也极有可能出现一对多的串行通信方式。

目前比较通用的串口扩展方案有两种。一是用硬件实现，使用多串口单片机或专用串口扩展芯片，可选的扩展芯片有 TI 等公司研发的 16C554 系列串口扩展芯片，该系列芯片通过并行口扩展串行口，功能比较强大、通讯速度高，但控制复杂，同时价格较高，主要的应用场合是 PC 机串口扩展产品。而多串口单片机也同样存在价格高的缺点。

另一种串口扩展方案就是用软件实现，当然，这种方案需要在硬件连接上使用总线式连接，软件上用广播式传输控制。具体来说，一个串口挂载多个串口设备，PC 机主动发送数据或控制信号到各个终端，终端则被动响应并适时回复。在终端回复的过程中，线路连接上需要考虑各个终端的信息干扰，可以在每个终端到回发线之间的连接一个单向传输的二极管，以解决此问题。软件模拟串口的缺点：一是采样次数低，分摊到每个终端设备的传输时间有限；二是不能实现高波特率通讯，一般不要使用高于 9600bps 的波特率，以保证传输正确性。

第5章 GPIO的应用实例--键盘、LED与LCD

键盘、数码管显示 LED 和液晶显示 LCD 是 MCU 的常用外部器件，它们一般通过通用 I/O 口与 MCU 进行通信，本章把它们作为 GPIO 的应用实例来看待。键盘是嵌入式应用的输入设备。识别键盘是否有键被按下的方法有查询法、定时扫描法与中断法等。LED 和 LCD 是嵌入式应用的输出设备。结合第 3 章介绍的 GPIO 构件，本章给出了 4×4 键盘、8 段码数码管显示 LED 与字符型 LCD 的编程实例。在实际应用中所采用的器件和本章所用的器件也许有所差异，但将其封装成构件的方法和过程是一样的，通过对构件封装过程的学习加深对构件思想的理解，相信对读者后续的学习会有一定的帮助。另外要注意，由于所采用的器件不同，硬件电路也会有很大差别，要充分考虑驱动电流等问题。

5.1 键盘

本节首先简要阐明有关键盘识别的一些基本问题，随后给出 MCF52233 MCU 的键盘的工作原理、基于构件思想的键盘构件头文件以及源程序文件的设计思想。

5.1.1 键盘模型及接口

键盘是由若干个按键组成的开关矩阵，它是最简单的 MCU 数字量输入设备。操作人员通过键盘输入数据或命令，实现简单的人机通信。

首先，我们应了解键盘模型及接口的基本概念和原理，以便为后续编程作准备。键盘的基本电路为接触开关，有通、断两种状态分别表示 0 和 1，详细电路及原理图参见光盘该章节的参考资料。MCU 通过检测与键盘相连接的 I/O 口来确定键盘状态。

其次，键盘接口按照不同的标准有不同的分类方法。按键盘排布的方式可分成独立方式和矩阵方式；按读入键值的方式可分为直读方式和扫描方式；按是否进行硬件编码可分成非编码方式和硬件编码方式；按微处理器响应方式可分为中断方式和查询方式。本章将使用查询方法编程。将以上各种方式组合可构成不同的键盘接口方式。以下介绍两种较为常用的键盘接口方式。

1. 独立方式

独立方式是指将每个独立按键按一对一的方式直接接到 I/O 输入线上。读键值时直接读 I/O 口，每个键的状态通过读入键值来反映，所以也称这种方式为一维直读方式，习惯称为独立式。这种方式查键实现简单，但占用 I/O 资源较多，一般在按键数量较少的情况下采用。

2. 矩阵方式

矩阵方式是用 n 条 I/O 线组成行输入口， m 条 I/O 线组成列输出口，在行列线的每一

个交点上设置一个按键。读键值方法一般采用扫描方式，即输出口按位轮流输出低电平，再从输入口读入键信息，最后获得键码。这种方式占用 I/O 线较少，在实际应用系统中采用较多。在使用这种方法时需要考虑，硬件连接时需要选择键盘哪些引脚作为 MCU 的输入，哪些作为 MCU 的输出，作为 MCU 的输入的键盘引脚需要上拉。

设计键盘的时候，通常小于 4 个按键的应用，可以使用独立式接口。如果多于 4 个按键，为了减少对微处理器的 I/O 资源的占用，可以使用矩阵式键盘。那么键盘编程应该注意哪些问题呢？下节我们就讨论这个问题，这对于实际应用非常重要。

5.1.2 键盘编程的基本问题

对于键盘编程我们至少应该了解下面几个问题：

第一，如何识别键盘上的按键？

第二，如何区分按键是否真正地被按下，还是抖动？

第三，如何处理重键问题？

了解这些问题有助于键盘编程。

1. 键的识别

如何知道键盘上哪个键被按下就是键的识别问题。若键盘上闭合键的识别由专用硬件实现，称为编码键盘；而靠软件实现的称为未编码键盘。在这里主要讨论未编码键盘的接口技术和键盘输入程序的设计。识别是否有键被按下，主要有查询法、定时扫描法与中断法等。而要识别键盘上哪个键被按下主要有行扫描法与行反转法。

2. 抖动问题

当键被按下时，会出现所按的键在闭合位置和断开位置之间跳几下才稳定到闭合状态的情况，当释放一个按键时也会出现类似的情况，这就是抖动问题。抖动持续的时间因操作者而异，一般为 5~10ms 之间，稳定闭合时间一般为十分之几秒~几秒，由操作者的按键动作所确定。在软件上，解决抖动的方法通常是延迟等待抖动的消失或多次识别判定。

3. 重键问题

所谓重键问题就是有两个及两个以上按键同时处于闭合状态的处理问题。在软件上，处理重键问题通常有连锁法与巡回法。

5.1.3 键盘编程实例

本节给出一个实际的 4×4 键盘编程实例，讲述如何扫描键盘取得键值和键盘编码等问题。读者可参见图 5-1 来正确理解 MCU 键盘接口方法与编程技术，图中列线 (n1~n4) 通过电阻接 VCC (+3.3V)，当键盘上没有键闭合时，所有的行线和列线断开，列线 n1~n4 都呈高电平。当键盘上某一个键闭合时，则该键所对应的行线与列线短接。例如，第 2 排第 3 个按键被按下闭合时，行线 m2 和列线 n3 短路，此时 n3 线上的电平由 m2 的电位所决定。那么如何确定键盘上哪个按键被按下呢？可以把列线 n1~n4 接到 MCU 的输入

口, 行线 m1~m4 接到 MCU 的输出口, 则在 MCU 的控制下, 使行线 m1 为低电平 (0), 其余三根行线 m2、m3、m4 都为高电平 (1), 并读列线 n1~n4 状态。如果 n1~n4 都为高电平, 则 m1 这一行上没有键闭合, 如果读出列线 n1~n4 的状态不全为高电平, 那么为低电平的列线和 m1 相交的键处于闭合状态; 如果 m1 这一行上没有键闭合, 接着使行线 m2 为低电平, 其余行线为高电平, 用同样方法检查 m2 这一行上是否有键闭合; 依此类推, 最后使行线 m4 为低电平, 其余的行线为高电平, 检查 m4 这一行上是否有键闭合。这种逐行逐列地检查键盘状态的过程称为对键盘的一次扫描。

下面介绍键值计算方法:

键盘与 MCU 接线见图 5-1, 图 5-2 给出了键的定义符号“0”~“9”、“A”~“D”、“*”、“#”等。如何识别“1”键呢? 这里将列线 n1~n4 分别接 AN4~AN7, 且编程时将 AN4~AN7 定义为输入并进行上拉, 行线 m1~m4 分别接 AN0~AN3, 且编程时将 AN0~AN3 定义为输出, 那么“1”键对应于: AN7~AN0=11101110, 即\$EE; “2”键对应于: AN7~AN0=11011110, 即\$DE; ……; “D”键对应于: AN7~AN0=01110111, 即\$77。前者“1”、“2”、“D”就是我们的定义值, 后者\$EE、\$DE、\$77 就是“键值”, 这种情况“键值”是一个字节。这样, 按图 5-1 的接法可以得出键值表, 见图 5-2。键值可以通过扫描法获得, 由键值通过查表法编程得到定义值。

从软件角度来看, 键盘构件包含头文件 KB.h 和程序文件 KB.c。由图 5-1 可看出, 键盘与 MCU 的 AN 口连接, 在进行键盘按键扫描时, 需设置 AN 口某些引脚的输出或读取某些引脚的状态, 显然, 在编写键盘构件时, 需调用 GPIO 构件。

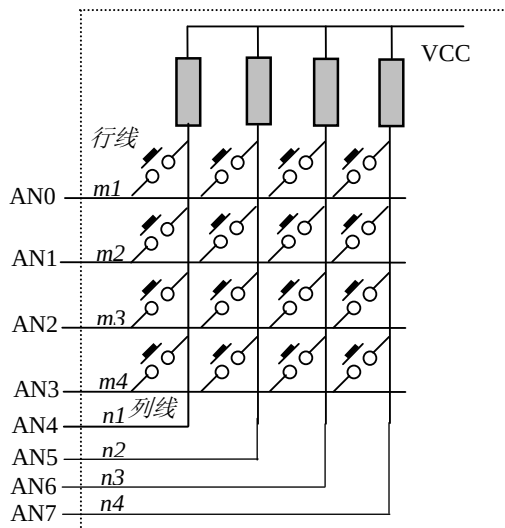


图5-1 4×4键盘与MCU接法示例

1 EE	2 DE	3 BE	A 7E	定义值
4 ED	5 DD	6 BD	B 7D	键值
7 EB	8 DB	9 BB	C 7B	
* E7	0 D7	# B7	D 77	

图5-2 键盘定义

1. 键盘构件的头文件 (KB.h)

```
// [KB.h] 键盘扫描识别头文件
// 说明: 本文件与具体的芯片型号有关
#ifndef KB_H // 防止重复定义
#define KB_H
```

```

//1. 头文件
#include

"mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义

#include "type.h"

//类型定义头文件

#include

"sysinit.h" //配置头文件

#include "GPIO.h"

//GPIO构件头文件
//2. 功能接口(函数声明)
//-----*
//函数名: KB_Init *
//功 能: 初始化键盘口方向和数据寄存器 *
//参 数: 无 *
//返 回: 无 *
//-----*
void KB_Init(void);

//-----*
//函数名:KB_Scan1扫描读取键值函数 *
//功 能: 扫描一 *
次4*4键盘上的按键, 读取键值; *
//参 数: 无 *
* //返 回: 扫描到 *
的键值。若无按键, 返回0xff *
//-----*
INT8U
KB_Scan1(void);

//-----*
// 函 数 *
名:KB_ScanN 多次扫描键盘函数 *
//功 能: 多次扫 *
描键盘, 消除"抖动" *
//参 *
数:KB_count:扫描键盘的的次数 *
//返 回: 多次扫 *
描键盘得到的键值, *
// 正常按 *
键--返回键值; 非正常按键(包括无按键)--返回0xFF *
// 内部调用函 *
数:KB_Scan1 *
//-----*
INT8U
KB_ScanN(INT8U KB_count);

```

```

//-----
-----*
// 函数名:KB_Def
键值转为定义值函数      *
为定义值                *
数:KB_valve:键值        *
//      参
// 返 回:键定义
//-----
-----*
INT8U
KB_Def(INT8U KB_valve);

//-----
-----*
//3. 宏定义
//键盘所接端口
#define KB_PORT  PORT_AN
#endif

```

需说明的是，在实际使用中，若键盘与MCU所连接的端口不是AN口，而是其他端口，则仅需修改宏定义语句：`#define KB_PORT PORT_AN`

2. 键盘构件的程序文件（KB.c）

```

//-----*
// 文件名: KB.c （键盘扫描识别驱动程序文件）      *
//-----*
//头文件
#include "KB.h"    //键盘扫描识别头文件

//-----*
//函数名: KB_Init      *
//功 能: 初始化键盘口方向和数据寄存器      *
//参 数: 无      *
//返 回: 无      *
//-----*
void KB_Init(void)
{
    //定义四根行线为输出
    uint8 i;
    for(i=0;i<4;i++)
    {
        GPIO_Init(KB_PORT, i, 1, 0);
    }
    //定义四根列线为输入

```

```

        for(i=4;i<8;i++)
        {
            GPIO_Init(KB_PORT, i, 0, 0);
        }
    }

//-----*
//函数名:KB_Scan1扫描读取键值函数
//功 能:扫描一次4*4键盘上的按键, 读取键值;
//参 数:无
//返 回:扫描到的键值。若无按键, 返回0xff
//-----*
uint8 KB_Scan1(void)
{
    uint8 i, j, tmp;
    //扫描
    for(i=0;i<4;i++)
    {
        //使第0-4行为1
        for(j=0;j<4;j++)
        {
            GPIO_Set(KB_PORT, j, 1);
        }
        asm(NOP);
        asm(NOP);
        //使第i行为0
        GPIO_Set(KB_PORT, i, 0);
        asm(NOP);
        asm(NOP);
        if ((MCF_GPIO_SET(KB_PORT) & 0xF0) != 0xF0)
            return MCF_GPIO_SET(KB_PORT); //读取键值
        else
            tmp=0xFF;
    }
    return(tmp);
}

//-----*
//函数名:KB_ScanN 多次扫描键盘函数
//功 能:多次扫描键盘, 消除“抖动”
//参 数:KB_count:扫描键盘的的次数
//返 回:多次扫描键盘得到的键值,
//      正常按键--返回键值; 非正常按键(包括无按键)--返回0xFF
//内部调用函数:KB_Scan1
//-----*
uint8 KB_ScanN(uint8 KB_count)
{
    uint8 i, KB_value_last, KB_value_now;
    if (0 == KB_count || 1 == KB_count)

```

```

        return KB_Scan1();
//先扫描一次得到的键值,便于下面比较
KB_value_now = KB_value_last = KB_Scan1();
//以下多次扫描消除误差
for (i=0; i<KB_count-1; i++)
{
    KB_value_now = KB_Scan1();
    if (KB_value_now == KB_value_last)
        return KB_value_now;    //返回扫描的键值
    else
        KB_value_last = KB_value_now;
}
return 0xDD;    //返回出错标志
}

//-----*
//函数名:KB_Def 键值转为定义值函数    *
//功 能:键值转为定义值    *
//参 数:KB_valve:键值    *
//返 回:键定义值    *
//-----*
const uint8 KB_Table[]=
{
    0xEE, '1', 0xDE, '2', 0xBE, '3', 0x7E, 'A',
    0xED, '4', 0xDD, '5', 0xBD, '6', 0x7D, 'B',
    0xEB, '7', 0xDB, '8', 0xBB, '9', 0x7B, 'C',
    0xE7, 'F', 0xD7, '0', 0xB7, 'E', 0x77, 'D',
    0x00
};
uint8 KB_Def(uint8 KB_valve)
{
    uint8 KeyPress;
    uint8 i;
    uint8 j;
    i = 0;
    while (1)
    {
        j = KB_Table[i];
        //与0(定义表结束标志)相比较
        if (j == 0)    //!=0,即表中无次定义值
        {
            KeyPress = 0xFF;    //返回0xFF
            break;
        }
        else
        {
            //与键值相比较
            if (j == KB_valve)    //找到
            {

```

```

        KeyPress = KB_Table[i+1];    //取键定义值
        break;
    }
    else
    {
        i += 2;                      //指向下一个键值, 继续判断
    }
}
}
return KeyPress;                    //返回键定义值
}

```

3. 测试工程的主函数 (main.c)

```

//-----*
//工程名: MCF52233键盘测试程序*
//硬件连接: AN0-AN7分别与键盘模块的插线孔1-8脚连接*
//程序描述: 按下键盘按键, 串口发送对应的键值和定义值 (十六进制显示)*
//目 的: 掌握键盘扫描、键值识别、键盘编码等基本知识*
//说 明: 波特率为9600, 使用UART0口*
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
//头文件
#include "Includes.h" //总头文件
uint8 msg[15]= "Test Keyboard!";
//主函数
int main(void)
{
    //1. 禁止总中断
    DisableInterrupts();
    //2. 模块初始化
    UartInit(0, 60, 9600, 0); //串行口初始化, 不使用中断
    UartSendN(0, 14, msg); //发送字符串: Test Keyboard!
    KB_Init(); //键盘初始化
    //3. 开放总中断
    EnableInterrupts();
    //主循环
    while (1)
    {
        uint8 KB_valueN, KB_DefValue;
        KB_valueN = KB_ScanN(10); //扫描键值, 存于KB_valueN中
        Delay(20); //延迟去抖动
        if (KB_valueN != 0xFF)
        {
            KB_DefValue = KB_Def(KB_valueN); //键值转化为定义值
            UartSend1(0, KB_valueN); //发送键值
            UartSend1(0, KB_DefValue); //发送定义值
            KB_valueN = 0xFF;
        }
    }
}

```

}

测试结果：运行程序后，UART0 模块首先向 PC 机发送“Test KeyBoard!”，若 PC 机收到则表明串行通信正常。然后，按下 4*4 键盘上的任一按键后，UART0 模块向 PC 机发送对应的键值和定义值。注意，这两个值需用十六进制显示。

5.2 数码管LED

本节在介绍 8 段数码管 LED 显示编程原理的基础上，给出了扫描法实现四连排 LED 显示实例。

5.2.1 扫描法编程原理

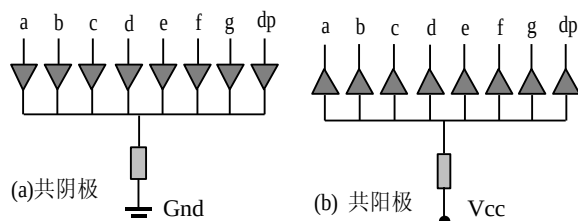


图5-3 数码管

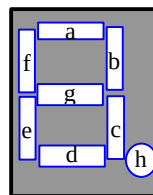


图5-4 数码管外形

对于 LED 编程我们需要了解下列几个问题：第一，所用 LED 是几段，是共阴还是共阳？第二，所选 LED 的电气参数怎样？如额定功率、额定电流是多少？如果我们对上述两个问题有明确的了解，那么对 LED 编程和封装 LED 构件就变得容易多了。

LED 的选择需要根据实际应用需求来决定，若只需要显示数字“0”~“9”，则只需七段 LED 就够了，若同时又要显示小数点，则需使用八段 LED。8 段数码管由 8 个发光二极管（Light-Emitting Diode，LED）组成。MCU 是通过 I/O 脚来控制 LED 某段发光二极管的亮灭从而达到显示某个数字的目的。那么怎样才能使 LED 发光二极管亮灭呢？首先应了解所选用的是共阴极数码管还是共阳极数码管。若共阴数码管，则公共端需要接地，若为共阳则公共端接电源正极（参见图 5-3）。例如，图 5-4 的 8 段数码管分别由 a、b、c、d、e、f、g 位段和小数点位段 h（或记为 dp）组成。共阴极 8 段数码管的信号端高电平有效，只要在各位段加上高电平信号即可使相应的位段发光，比如要使 a 段发光，则在 a 段加上高电平即可。共阳极的 8 段数码管则相反，在相应的位段加上低电平即可使该位段发光。因而一个 8 段数码管就必须有 8 位（即 1 个字节）数据来控制各个位段的亮灭。比如对共阳极 8 段数码管，AN0~7 分别接 a~g、dp，即 MCF_GPIO_PORTAN=0b01111111 时 dp 段亮；当 MCF_GPIO_PORTAN=0b10000000 时除 dp 位段外，其他位段均亮。到此我们基本弄清对一个 LED 编程的原理了，下面我们需要注意的是在进行硬件连接时需要注意所选用的 LED 的电气参数，如能承受的最大电流，额定电压。根据其电气参数来选择使用限流电阻或电流放大电路。

下面介绍如何进行多个 LED 编程，因为在实际应用中往往需要多个 LED 协同使用。那么是不是如上段述说一样，有几个 8 段数码管，就必须有几个字节的数据来控制各个数码管的亮灭。这样控制虽然简单，却不切实际，MCU 也不可能提供这么多的端口用来控制数码管，为此往往是通过一个称为数据口的 8 位数据端口来控制位段。而 8 段数码管的公共端，原来接到固定的电平（对共阴极是 GND，对共阳极是 Vcc），现在接 MCU 的一个输出引脚，由 MCU 来控制，通常叫“位选信号”，而把这些由 n 个数码管合在一起的数码管组称为 n 连排数码管。这样 MCU 的两个 8 位端口就可以控制一个 8 连排的数码管。若是要控制更多的数码管，则可以考虑外加一个译码芯片。图 5-5 所示为一个四连排的共阴极数码管，它们的位段信号端（称为数据端）接在一起，可以由 MCU 的一个 8 位端口控制，同时还有 4 个位选信号（称为控制端），用于分别选中要显示数据的数码管，可用 MCU 另一个端口的 4 个引脚来控制。如图 5-5 所示，每个时刻只让一个数码管有效，由于人眼的“视觉暂留”（约 100ms 左右），看起来则是同时显示的。

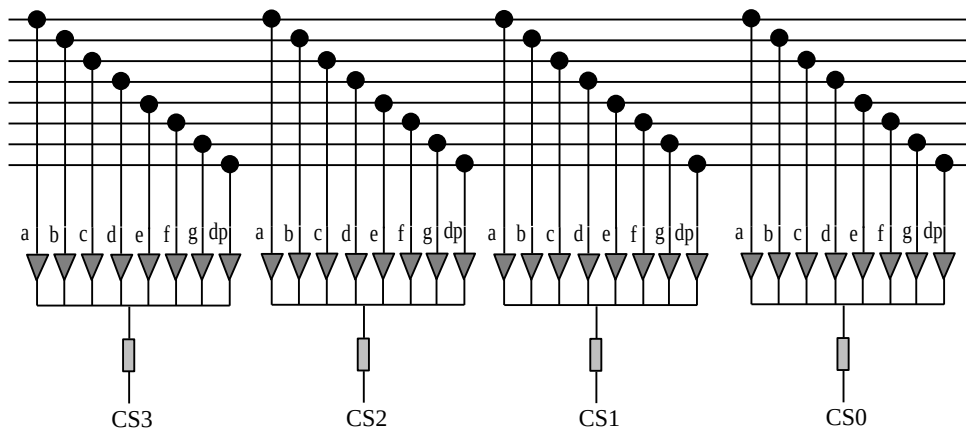


图 5-5 四连排共阴极 8 段数码管

5.2.2 数码管编程实例

图 5-6 给出了一个共阴极四连排 8 段数码管的硬件构件连接实例。利用 MCU 的 AN 口控制 8 个位段（数据）。图中 AN7~0 分别接 h~a 位段，TC 口 3~0 脚作为片选端。

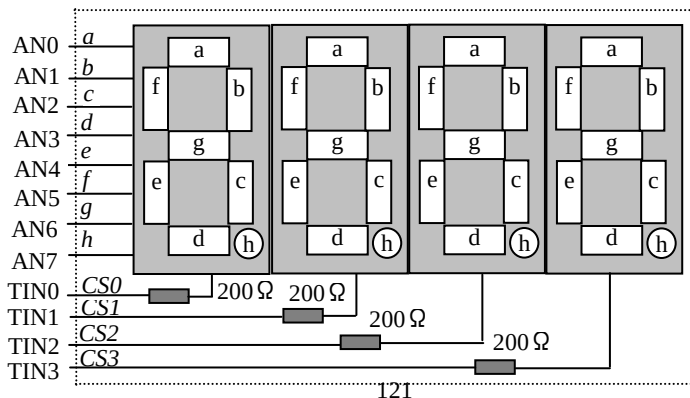


图5-6 MCU与一个四连排数码管的连接

以下给出 MCF52233 对上述四连排 LED 的 C 语言编程实例。

1. LED构件的头文件（LED.h）

```

//-----*
// 文件名:  LED.h                                     *
// 说 明:  小灯初始化头文件:包括数据线, 片选线引脚定义。与具体的芯片相关 *
//                                              *
//-----*

#ifndef LED_H
#define LED_H

//1. 头文件
#include

"mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义

#include "type.h"

//类型定义头文件

#include

"sysinit.h" //配置头文件

#include "GPIO.h"

//2. 功能接口(函数
声明)
//-----
-----*
// 函数名:LEDinit
四连排LED初始化 *
//功能:定义LED控
制引脚的数据口和位选口为输出 *
// 参 数 : 无
*
// 返 回 : 无
*
//-----
-----*
void
LEDInit(void);

//-----*
//函数名:LEDShow1 在1个LED上显示数字 *
//功能:在第i个LED上显示数字c(要查表转码) *
//参数: *
//      (1)i:要显示的LED位号(从右到左1-3) *
//      (2)c:要显示的数字(0-9) *

```

```

//返回:无 *
//-----*
void LEDShow1(uint8 i, uint8 c);

//-----*
//3. 宏定义
#define LED_DATA

PORT_AN //定义LED数据口为AN口
#define LED_CS

PORT_TC //定义LED片选口为TC口

#endif

```

2. LED构件的程序文件（LED.c）

```

//-----*
// 文件名: LED.c (LED驱动程序文件) *
//-----*
#include "LED.h" //该头文件包含LED驱动相关头文件

//-----*
//函数名:LEDInit 四连排LED初始化 *
//功能:定义LED控制引脚的数据口和位选口为输出 *
//参数:无 *
//返回:无 *
//-----*
void LEDInit(void)
{
    //初始化LED数据口、位选口为输出
    uint8 i;
    for(i=0;i<8;i++)
    {
        GPIO_Init(LED_DATA, i, 1, 0);
        GPIO_Init(LED_CS, i, 1, 0);
    }
}

//-----*
//函数名:LEDShow1 在1个LED上显示数字 *
//功能:在第i个LED上显示数字c(要查表转码) *
//参数: *
// (1)i:要显示的LED位号(从右到左1-3) *
// (2)c:要显示的数字(0-9) *
//返回:无 *
//-----*
//显示码表
const uint8 Dtable[10] =
// 0 1 2 3 4 5 6 7 8 9

```

```

    {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
//片选表
const uint8 CStable[4] =
// 0 1 2 3
    {0xF7, 0xFB, 0xFD, 0xFE};

void LEDShow1(uint8 i, uint8 c)
{
    MCF_GPIO_PORT(LED_CS) = CStable[i];
    MCF_GPIO_PORT(LED_DATA) = Dtable[c];
}

```

3. 测试工程的主函数 (main.c)

```

//-----*
// 工程名: LED数码管 *
// 硬件连接: (1)AN口的AN0-AN7分别接LED的数据线LED1_LD0-LED1_LD7, 即a-h *
//           (2)TC口的TIN0-TIN3分别接LED的位选线LED1_CS0-LED1_CS3 *
// 程序描述: 先向PC机串口发送' Test LED!', 然后LED数码管显示' 0123' *
// 目的: 巩固I/O引脚编程, 掌握数码管使用 *
// 说明: 可通过修改LEDbuf值更换数码管显示内容 *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
#include "includes.h" //头文件
uint8 msg[10]= "Test LED!";
uint8 LEDbuf[4]; //存放需要LED显示数据
int main(void)
{
    LEDbuf[0] = '0';
    LEDbuf[1] = '1';
    LEDbuf[2] = '2';
    LEDbuf[3] = '3';
    //1. 关总中断
    DisableInterrupts();
    //2. 模块初始化
    LEDInit();//LED初始化
    UartInit(0, 60, 9600, 0); //串口初始化, 不使用中断
    UartSendN(0, 9, msg); //发送字符串:Test LED!
    //3. 开总中断
    EnableInterrupts();
    //主循环
    while (1)
    {
        uint8 i, c;
        uint16 j;
        for (i = 0; i <= 3; i++)
        {
            c = LEDbuf[i] - '0'; //将字符转换为数字
            LEDShow1(3-i, c); //显示该数字
            for (j = 0; j <= 100; j++); //延时
        }
    }
}

```

```

    }
}
}

```

测试结果：观察 LED，显示“0123”4 位数字。修改 main.c 中的 LEDbuf[] 数组的值可在 LED 上观察到相应数值，以验证程序的正确性。假如要是换用其它端口作为 LED 的数据端口和片选信号，则只需修改“LED.h”中的宏定义语句：

```

#define LED_DATA  PORT_AN    //定义LED数据口为AN口
#define LED_CS     PORT_TC    //定义LED片选口为TC口

```

5.3 液晶LCD

5.3.1 HD44780的编程结构

LCD 的编程我们将以 HD44780 为例来展示构件的封装过程，当然只要我们熟悉一款 LCD 的构件编程，那么即使你需要换用其它公司产品也是非常容易和轻松的，因为这正是构建思想的目的所在。从编程角度看，HD44780 内部主要由指令寄存器（IR）、数据寄存器（DR）、忙标志（BF）、地址计数器（AC）、显示数据寄存器（DD RAM）、字符发生器 ROM（CG ROM）、字符发生器 RAM（CG RAM）及时序发生电路构成。下面我们将简单了解一下上述寄存器的功能，至于 HD44780 的指令集请参阅光盘中该章相关资料。

1. 指令寄存器（IR）

指令寄存器 IR 用于 MCU 向 HD44780 写入指令码。IR 只能写入，不能读出。当 RS=0、 $R/\overline{W}=0$ 时，数据线 DB7~DB0 上的数据写入 IR。

2. 数据寄存器（DR）

DR 用于寄存数据。当 RS=1、 $R/\overline{W}=0$ 时，数据线 DB7~DB0 上的数据写入数据寄存器 DR，同时 DR 的数据由内部操作自动写入 DD RAM 或 CG RAM。当 RS=1、 $R/\overline{W}=1$ 时，内部操作将 DD RAM 或 CG RAM 送到 DR 中，通过 DR 送到数据总线 DB7~DB0 上。

3. 忙标志（BF）

令 RS=0、 $R/\overline{W}=1$ ，在 E 信号高电平的作用下，BF 输出到总线的 DB7 上，MCU 可以读出判别。BF=1，表示该液晶正在进行内部操作，不能接受外部指令或数据。

4. 地址计数器（AC）

AC 作为 DD RAM 或 CG RAM 的地址指针。如果地址码随指令写入 IR，则 IR 的地址码部分自动装入地址计数器 AC 之中，同时选择了相应的 DD RAM 或 CG RAM 单元。

AC 具有自动加 1 或自动减 1 功能。当数据从 DR 送到 DD RAM（或 CG RAM），AC 自动加 1。当数据从 DD RAM（或 CG RAM）送到 DR，AC 自动减 1。当 RS=0、 $R/\overline{W}=1$ 时，在 E 信号高电平的作用下，AC 所指向的内容送到 DB7~DB0。

5. 显示数据寄存器 (DD RAM)

DD RAM 用于存储显示数据, 共有 80 个字符码。对于不同的显示行数及每行字符个数, 所使用的地址不同, 具体的对应关系, 可参阅使用说明书, 光盘中也有相应的介绍。

6. 字符发生器ROM (CG ROM)

CG ROM 由 8 位字符码生成 5×7 点阵字符 160 种和 5×10 点阵字符 32 种, 8 位字符编码与字符的对应关系, 可以直接使用。其中大部分与 ASCII 码兼容, 对应关系图见光盘中资料。

7. 字符发生器RAM (CG RAM)

CG RAM 是提供给用户自定义特殊字符用的, 它的容量仅为 64 字节, 编址为 00~3FH。作为字符字模使用的仅是一个字节中的低 5 位, 每个字节的高 3 位留给用户作为数据存储器使用。如果用户自定义字符由 5×7 点阵构成, 可定义 8 个字符。

5.3.2 点阵字符型LCD编程实例

通过上文描述, 我们对该款 LCD 的接口及指令集有了一定的了解, 这也是封装 LCD 构件的必要准备。假如你不使用该款 LCD 而需要封装其它公司产品的构件, 其操作过程也是一样的。

本节给出点阵字符型 LCD 的一个编程实例。在实验板上, LCD 的数据线 7~14 脚 (D0~D7) 分别与 MCU 的 AN0~AN7 连接, LCD 的控制线 RS、 $R/\bar{W}$ 、E (4、5、6 脚) 分别与 MCU 的 GPT3、GPT2、GPT1 连接, 图 5-7 给出了 LCD 硬件构件原理图。LCD 的 1、2、3 脚为供电电源与亮度调节引脚。实验箱的扩展板上还留有一排插孔, 当 LCD 接在下排插孔时, LCD 控制线和数据线与 MCU 的连接可以自行定义, 并通过下部接线孔接线。以下给出 MCF52233 的 LCD 编程实例。

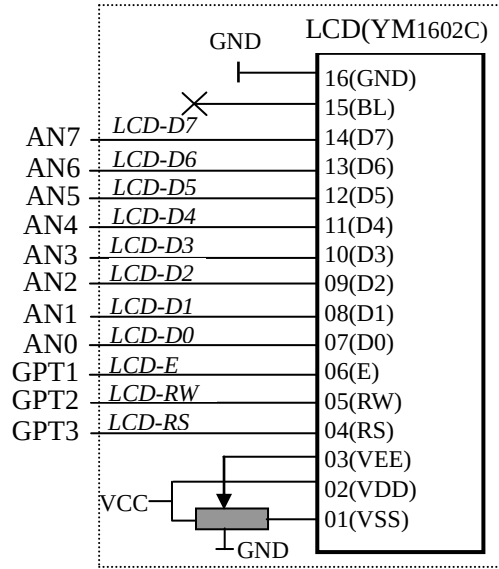


图5-7 MCF52233与LCD的连接

1. LCD函数定义头文件（LCD.h）

```
//[LCD.h]液晶显示头文件-----
#ifndef LCD_H                //防止重复定义
#define LCD_H

//1. 头文件
#include

"mcf52233.h"                //MCU映像寄存器名

//类型别名定义
#include "type.h"

"sysinit.h"                //配置头文件
#include "GPIO.h"           //包含GPIO构件头文件

//2. 功能接口(函数
声明)
//-----
*
//函数名:LCDInit
初始化LCD函数
*
//功 能:初始化
Lcd(HD44780), 设置显示方式, 清屏, AC自动+1
*
//参 数: 无
*
//返 回: 无
*
//内 部 调 用 函
```

```

数:LCD_Command *
//-----
-----*
void
LCDInit(void);

//-----
-----*
//函数名:LCDShow:
在HD44780显示屏上显示数据 *
//功 能:在
HD44780显示屏显示str所指向的32个数据 *
//参 数:str:待显
示的数组 *
//返 回:无
*
//内 部 调 用 函
数:LCD_Command *
//-----
-----*
void
LCDShow(uint8 str[]);

//-----
-----*
//3. 宏定义
#define LCD_DATA

PORT_AN //LCD数据端口

#define
LCD_CONTROL PORT_TA //LCD控制信号端口

#define LCD_E
1 //LCD使能信号接控制端口的1脚
#define LCD_RW
2 //LCD读写信号接控制端口的2脚
#define LCD_RS
3 //LCD寄存器选择信号控制端口的3脚

#endif

```

2. LCD函数定义 (LCD.c)

```

//-----*
// 文件名: LCD.c (LCD驱动程序文件) *
//-----*
#include "LCD.h" //包含LCD头文件
//内部函数原型说明
void LCD_Command(uint8 cmd);

//-----*

```

```

//函数名:LCDInit 初始化LCD函数                                     *
//功 能:初始化Lcd(HD44780), 设置显示方式, 清屏, AC自动+1         *
//参 数:无                                                         *
//返 回:无                                                         *
//内部调用函数:LCD_Command                                         *
//-----*
void LCDInit(void)
{
    uint16 i;

    //初始化数据端口的8个引脚输出0
    for(i=0;i<8;i++)
    {
        GPIO_Init(LCD_DATA, i, 1, 0);
    }
    //初始化控制端口的3个引脚
    GPIO_Init(LCD_CONTROL, LCD_E, 1, 1); //E=1
    GPIO_Init(LCD_CONTROL, LCD_RS, 1, 0); //RS=0
    GPIO_Init(LCD_CONTROL, LCD_RW, 1, 0); //RW=0

    //设置指令
    LCD_Command(0x38);        //5*7点阵模式, 2行显示, 8位数据总线
    LCD_Command(0x08);        //关显示, 关光标显示, 不闪烁
    LCD_Command(0x01);        //清屏
    for (i=0; i<20000; i++); //延时>1.6ms
    LCD_Command(0x06);
    LCD_Command(0x14);        //光标右移一个字符位, AC自动加1
    LCD_Command(0x0C);        //开显示, 关光标显示, 不闪烁
}

//-----*
//函数名:LCDShow:在HD44780显示屏上显示数据                         *
//功 能:在HD44780显示屏显示str所指向的32个数据                 *
//参 数:str:待显示的数组                                         *
//返 回:无                                                         *
//内部调用函数:LCD_Command                                         *
//-----*
void LCDShow(uint8 str[])
{
    uint8 i;
    //1. LCD初始化
    LCDInit();
    //2. 显示第1行16个字符
    //2.1 设置显示首地址, RS, R/W = 00(写的是指令)
    GPIO_Set(LCD_CONTROL, LCD_RS, 0);
    GPIO_Set(LCD_CONTROL, LCD_RW, 0);
    LCD_Command(0x80);        //后7位为DD RAM地址(0x00)
    //2.2 RS, R/W = 10(写的是数据)
    GPIO_Set(LCD_CONTROL, LCD_RS, 1);

```

```

GPIO_Set(LCD_CONTROL, LCD_RW, 0);
//2.3 将要显示在第1行上的16个数据逐个写入DD RAM中
for (i = 0; i < 16; i++)
{
    LCD_Command(str[i]);
}
//3. 显示第2行16个字符
//3.1 设置显示首地址, RS, R/W = 00(写的是指令)
GPIO_Set(LCD_CONTROL, LCD_RS, 0);
GPIO_Set(LCD_CONTROL, LCD_RW, 0);
LCD_Command(0xC0); //后7位为DD RAM地址(0x40)
//3.2 RS, R/W = 10(写的是数据)
GPIO_Set(LCD_CONTROL, LCD_RS, 1);
GPIO_Set(LCD_CONTROL, LCD_RW, 0);
//3.3 将要显示在第2行上的16个数据逐个写入DD RAM中
for (i = 16; i < 32; i++)
{
    LCD_Command(str[i]);
}

//-----函数LCD_Command调用的内部延迟子程序-----*
void Delay40us();
void Delay12NOP();

//-----*
//函数名:LCD_Command 执行给定的cmd命令 *
//功 能:执行给定的cmd命令,且延时 *
//参 数:cmd:待执行的命令 *
//返 回:无 *
//-----*
void LCD_Command(unsigned char cmd)
{
    //1. 等待 > 40us
    Delay40us();
    //2. 数据送到LCD的数据线上
    MCF_GPIO_PORT(LCD_DATA)=cmd;
    //3. 给出E信号的下降沿(先高后低),使数据写入LCD
    GPIO_Set(LCD_CONTROL, LCD_E, 1);
    Delay12NOP();
    GPIO_Set(LCD_CONTROL, LCD_E, 0);
    //4. 等待 > 40us
    Delay40us();
}

//-----*
//函数名:Delay40us *
//功 能:延时>40us *

```

```

//参 数:无 *
//返 回:无 *
//-----*
void Delay40us()
{
    uint16 i;
    for (i=0; i<800; i++);
}

//-----*
//函数名:Delay12NOP *
//功 能:延时 *
//参 数:无 *
//返 回:无 *
//-----*
void Delay12NOP()
{
    uint8 i;
    for (i=0; i<13; i++) asm(NOP);
}

```

3. 测试工程的主函数 (main.c)

```

//-----*
//工 程 名:MCF52233液晶(LCD) *
//硬件连接: (1) AN口的AN0-AN7与LCD数据线的D0-D7连接(若LCD接上排孔则不必连接) *
//           (2) GPT1与LCD的使能信号 E 连接 *
//           (3) GPT2与LCD的读写信号 RW 连接 *
//           (4) GPT3与LCD的片选信号 RS 连接 *
//程序描述: *
//           (1) 初始显示: *
//               "Wait Receiving.." *
//               "Soochow 2009. 03." *
//           (2) MCU等待接收PC发来的32个字符, 并送LCD显示 *
//目    的:熟悉LCD编程 *
//说    明:波特率为9600, 使用UART0口 *
//-----苏州大学嵌入式系统实验室-----*

#include "includes.h" //头文件
uint8 msg[10]= "Test LCD!";
int main(void)
{
    //定义初始显示缓存并赋初值
    uint8 g_DisplyInit[33] = "Wait Receiving..Soochow 2009. 03.";
    //receive_data中存放从串口接收来的要被显示的32个数据
    uint8 receive_data[32];
    //1. 关总中断
    DisableInterrupts();
    //2. 模块初始化

```

```
//使用UART0模块,使用系统时钟作为UART的时钟源,波特率为9600,不使用中断
UartInit(0, 60, 9600, 0);
UartSendN(0, 9, msg); //发送字符串:Test LCD!

LCDInit(); //LCD初始化
//3. 开总中断
EnableInterrupts();
// 主循环
LCDShow((uint8 *)g_DisplyInit); //显示初始字符
while (1)
{
    //接收PC发送的32个数据,并存放至receive_data数组中
    if (0 == UartReN(0, 32, receive_data)) //接收成功
    {
        //LCD显示MCU从串口接收到的32个数据并返回给PC
        LCDShow((uint8 *)receive_data);
        UartSendN(0, 32, receive_data);
    }
}
```

测试结果: 运行程序后, 首先向 PC 发送“Test LCD!”, 然后在 LCD 分两行显示“Wait Receiving..”和“Soochow 2009.03.”。随后循环等待 PC 机发送过来的 32 个字节数据并显示在 LCD 上。

第6章 MCF52233的AD转换模块

在过程控制和仪器仪表中,多数情况下是由嵌入式计算机进行实时控制及实时数据处理的。计算机所加工的信息是数字量,而被测控对象往往是一些连续变化的模拟量(如温度、压力、速度或流量等)。模数(A/D, Analog / Digital)转换模块是计算机与外界连接的纽带,是大部分嵌入式应用中必不可少的重要组成部分,该部分的性能直接影响到嵌入式设备的总体性能。

本章首先简要阐述 A/D 转换的基础知识,接着给出 MCF52233 单片机内部 A/D 转换模块的基本编程方法,并封装了 A/D 转换构件,可供实际开发参考使用。

6.1 AD转换的基本问题

A/D 转换模块(Analog To Digital Convert Module)即模/数转换模块,其功能是将电压信号转换为相应的数字信号。实际应用中,这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过 A/D 转换后,MCU 就可以处理这些物理量。进行 A/D 转换,应该了解以下一些基本问题:第一,采样精度是多少;第二,采样速率有多快;第三,滤波问题;第四,物理量回归等。

1. 采样精度

采样精度就是指数字量变化一个最小量时模拟信号的变化量,即我们通常所说的采样位数。通常,MCU 的采样位数为 8 位,某些增强型的可达到 10 位,而专用的 A/D 采样芯片则可达到 12 位、14 位、甚至 16 位。设采样位数为 N ,则最小的能检测到的模拟量变化值为 $1/2^N$ 。例如以 MCF52233 为例,其采样精度最高为 12 位,参考电压为 3.3V,则检测到的模拟量变化为 $3.3/2^{12}=0.8\text{mV}$ 。

2. 采样速率

采样速率是指完成一次 A/D 采样所要花费的时间。在多数的 MCU 中要花费大于 15~20 个指令周期。因而此速率和所选器件的工作频率有很大关系。

3. 滤波

为了使采样的数据更准确,必须对采样的数据进行筛选去掉误差较大的毛刺。通常我们采用中值滤波和均值滤波来提高采样精度。所谓中值滤波,就是将 M 次连续采样值按大小进行排序,取中间值作为滤波输出。而均值滤波,是把 N 次采样结果值相加,然后再除以采样次数 N ,得到的平均值就是滤波结果。若要得到更高的精度,可以通过建立其他误差模型分析方式来实现。

4. 物理量回归

在实际应用中,得到稳定的 A/D 采样值以后,还需要把 A/D 采样值与实际物理量对应起来,这一步称为物理量回归。A/D 转换的目的是把模拟信号转化为数字信号,供计算机进行处理,但必须知道 A/D 转换后的数值所代表的实际物理量的值,这样才有实际意义。例如,利用 MCU 采集室内温度, A/D 转换后的数值是 126,实际它代表多少温度呢?如果当前室内温度是 25.1℃,则 A/D 值 126 就代表实际温度 25.1℃。实现这种对应关系有多种方法,本章将在 6.4 节的实际应用模型中对物理量回归概念做进一步说明。

6.2 MCF52233的AD转换结构与特性

MCF52233 的 A/D 转换(ADC)由 5 个主要功能模块组成,即输入多路复用器、采样和保持单元、转换器单元、结果寄存器和控制单元。虽然 ADC 是 MCF52233 上的一个独立模块,包含有两个独立的转换器 A 和 B,它们共享一个公共参考电压和数字控制模块,但各自有独立的输入多路复用器、采样和保持单元以及转换逻辑。因此,ADC 模块可以实现两路模拟量同时采样。

ADC 的转换过程可以由同步信号(SYNCn)或者向控制寄存器的开始位 STARTn 写 1 来启动。扫描顺序由通道列表寄存器 1/2(ADLST1/ADLST2)决定,这两个寄存器共有 8 个采样槽 SAMPLEn ($0 \leq n \leq 7$),可分别对应一路采样通道 ANn ($0 \leq n \leq 7$)。

ADC 有两种基本扫描模式:顺序扫描和并行扫描。在顺序扫描过程中,从采样槽 0 到 7 依次进行扫描;在并行扫描过程中,转换器 A 从采样槽 0 到 3 依次扫描,转换器 B 从采样槽 4 到 7 依次扫描。ADC 模块还提供了采样禁止寄存器 ADSDIS,用于禁止某些采样槽。并行扫描可以是同步的或者是非同步的。同步扫描时,A 和 B 两个转换器的扫描是同时进行的,同时产生转换结果对。它们共享开始、停止、同步、扫描结束中断使能控制位。当转换器 A 和 B 中的任意一个遇到被禁止的采样槽时,两个转换器都会终止扫描。非同步扫描时,两个转换器的并行扫描是独立的,相对于同步扫描,它们有各自的开始、停止、同步、扫描结束中断使能控制位。当转换器 A 或 B 遇到被禁止的采样槽时,就会终止自身的扫描,但不会影响另一个转换器的扫描。

ADC 可以配置为执行单次扫描和停止,触发时执行一次扫描,或重复的扫描采样槽序列直到手动停止。在单次扫描中,触发器必须为每次扫描进行重新装备,装备可以在前一个出发后,甚至在前一个扫描还在进行中随时进行。在扫描采样槽序列结束时会产生一些可选的中断,包括表明一次扫描结束、采样结果超出范围或者满足了一些不同的过零条件。

ADC 的输入模式分为单端输入模式和差分输入模式,由控制 1 寄存器(CTRL1)的通道配置域 CHNCFG 决定。在单端输入模式下,设 V_{IN} 是某一路 ANn 单端输入模拟电压, V_{REFH} 是 ADC 模块的正参考电压(3.3V), V_{REFL} 是 ADC 模块的负参考电压(0V),那么该路 12 位 AD 转换结果是: $((V_{IN}-V_{REFL})/(V_{REFH}-V_{REFL}))*4095$ 。所以,当单端输入电压为 3.3V 时,AD 转换有最大值 4095;而如果输入 0V,则有最小值 0。在差分模式下,8 路模拟量

输入分成 AN0-1、AN2-3、AN4-5 和 AN6-7 共 4 个差分对。序号为偶数的模拟量输入 ANn (n = 0,2,4,6) 作为差分信号的正端, 序号为奇数的模拟量输入 ANn (n = 1,3,5,7) 作为差分信号负端。设 V_{IN1} 和 V_{IN2} 分别是差分对中的正端和负端模拟电压, 那么相应差分模拟输入的 12 位转换结果是: $((V_{IN1} - V_{IN2}) / (V_{REFH} - V_{REFL})) * 4095$ 。所以, 当差分信号正负端的压差为 3.3V 时, AD 转换有最大值 4095; 当压差为 -3.3V, 则有最小值 0; 而压差为 0 时, 对应的 AD 值是 2048。

以上计算得到的是扫描每个采样槽 SAMPLEn 的原始转换结果, 该结果是一个无符号数, 由于 MCF52233 ADC 模块的转换精度为 12 位, 所以它的范围是 0~4095, 该结果要经由 ADC 的加法器进行偏移量修正, 即要减去对应偏移量寄存器 ADOFSn 中的偏移值, 得到一个有符号数, 存入对应的结果寄存器 ADRSLTn 中。如果偏移量设为 0, 那么修正后的结果就是一个无符号数 (正数, 符号位为 0)。同时, 原始转换结果要与上限和下限寄存器中的值比较, 结果寄存器中的值要进行过零检验。要注意的是, 结果、偏移量、上限和下限寄存器都是 16 位的, 它们的后 3 位不能写, 且读出总是 0, 因此, 实际上 12 位的原始结果要先向右移 3 位, 才能进行偏移量修正等操作。

MCF52233 ADC 的主要特性总结如下:

- (1) 12 位的转换精度。
- (2) A/D 转换的时钟频率最高为 5MHz。
- (3) 采样速率高达每秒 1.66 百万次。
- (4) 一次一路 A/D 转换时间为 8.5 个 ADC 时钟周期 (1.7 μ s), 后续每路为 6 个 ADC 时钟周期 (1.2 μ s)。
- (5) 在并行模式下, 8 路 A/D 转换总时间只要 26.5 个 ADC 时钟周期 (5.3 μ s)。
- (6) 有并行和顺序两种采样模式。
- (7) 内部的多路复用器可以选择 8 路输入中的 2 路。
- (8) 电源节省模式可以自动关闭/启动整个或部分 ADC 模块。
- (9) 没有被选择的输入口能够容纳电流的流动而不会影响到 ADC 性能, 并且能够在工业噪声环境中工作。
- (10) 在一次扫描结束后可以允许超限或过零中断。
- (11) 可以对采样进行修正, 即将最初的 ADC 值减去预先编程设定的偏移量从而得到最终的 ADC 转换结果。
- (12) ADC 结果可以是有符号数或无符号数。
- (13) 这些输入引脚可以是单端输入、差分输入, 或是两种输入方式兼有。

6.3 MCF52233 的 AD 模块编程寄存器

MCF52233 单片机的 8 路 12 位 A/D 转换模块引脚为 AN7/PAN7~AN0/PAN0, 每一个引脚都可以实现模拟量 (ANx) / 数字量 (PANx) 的复用输入。

MCF52233 的 A/D 转换模块有 43 个寄存器, 它们是: 2 个控制寄存器 (CTRL1~CTRL2)、1 个过零控制寄存器 (ADZCC)、2 个通道列表寄存器 (ADLST1 和 ADLST2)、

1 个采样禁止寄存器 (ADSDIS)、1 个状态寄存器 (ADSTAT)、1 个限制状态寄存器 (ADLSTAT)、1 个过零状态寄存器 (ADZCSTAT)、8 个结果寄存器 (ADRSLTn)、8 个下限寄存器 (ADLLMTn)、8 个上限寄存器 (ADHLMn)、8 个偏移量寄存器 (ADOFSn)、1 个电源控制寄存器 (POWER)、1 个电压参考寄存器 (CAL)。通过对这些寄存器的编程, 就可以获取 A/D 转换数据。

1. 控制寄存器1 (Control 1 Register, CTRL1)

CTRL1 的地址为 0x19_0000。定义为:

位	D15	D14	D13	D12	D11	D10	D9	D8	D7:4	D3	D2:0
读		STOP0		SYNC0	EOSIE0	ZCIE	LLMTIE	HLMTIE	CHNCFG		SMODE
写			START0								
复位	0	1	0	1	0	0	0	0	0:0	0	101

D14—STOP0 位, 停止转换位 0。0: 正常工作模式; 1: 停止模式。当该位为 1 时, 当前扫描被停止, 也不能开始新的扫描。SYNC0 信号输入或向 START0 位写 1 都会被忽略, 直到 STOP0 位被清零。当 ADC 在停止模式下, 结果寄存器可以被处理器修改, 这和 AD 核的处理是一样的。因此, 上下限检验、过零和相关中断都可以是被允许的。

D13—START0 位, 开始转换位 0。0: 无动作。1: 启动扫描。该位为只写位, 在写之前, ADC 必须有稳定的电源配置。

D12—SYNC0 位, 同步使能位 0。0: 扫描动作只能通过写 START0 位来初始化; 1: 扫描动作可以由 SYNC0 脉冲输入或写 START0 位来初始化。设置该位后, SYNC0 端有正沿输入时, 可以初始化一次转换。在扫描期间如果还有 STNC0 信号输入, 将被忽略。在一次顺序或一次并行扫描模式下, 只有第一个 SYNC0 输入信号是有效的, 后续 STNC0 信号将被忽略, 除非重写 SYNC0 位。这个重写动作可以在任何时候发生, 包括在扫描期间。

D11—EOSIE0 位, 扫描结束中断使能位 0。0: 禁止该中断; 1: 使能该中断。该位使能扫描结束中断。在循环扫描模式下, 每一次迭代完成都会触发中断。

D10—ZCIE 位, 过零中断使能位。0: 禁止该中断; 1: 使能该中断。设置该位之后, 如果当前转换结果和上一个转换结果相比, 符号发生改变, 且 ADZCC 寄存器中做了相关设置 (允许过零), 那么就会发生过零中断。

D9—LLMTIE, 超下限中断使能位。0: 禁止该中断; 1: 使能该中断。某个采样槽的原始采样值在进行偏移量修正之前和相应的下限寄存器 ADLLMTn 比较, 如果当前结果值小于下限寄存器中的值, 并且设置了 LLMTIE 位, 就会发生过下限中断。

D8—HLMTIE, 超上限中断使能位。0: 禁止该中断; 1: 使能该中断。某个采样槽的原始采样值在进行偏移量修正之前和相应的上限寄存器 ADHLMn 比较, 如果当前结果值大于上限寄存器中的值, 并且设置了 HLMTIE 位, 就会发生过上限中断。

D7~D4—CHNCFG 位, 通道配置位。该字段将输入通道配置为单端或差分输入, 如表 6-1 所示。

表 6-1 通道配置表

CHNCFG	输入	描述
xxx1	AN0~AN1	配置为差分对 (AN0为+, AN1为-)
xxx0		两个通道都配置为单端输入
xx1x	AN2~AN3	配置为差分对 (AN2为+, AN3为-)
xx0x		两个通道都配置为单端输入
x1xx	AN4~AN5	配置为差分对 (AN4为+, AN5为-)
x0xx		两个通道都配置为单端输入
1xxx	AN6~AN7	配置为差分对 (AN6为+, AN7为-)
0xxx		两个通道都配置为单端输入

D2~D0—SMODE 位, 扫描模式控制位。该字段控制 ADC 模块的扫描模式, 如表 6-2 所示。

表 6-2 SMODE 所对应的扫描模式

SMODE	扫描模式
000	顺序扫描一次
001	并行扫描一次
010	顺序循环扫描
011	并行循环扫描
100	触发顺序扫描
101	触发并行扫描 (默认值)
110、111	保留

2. 控制寄存器2 (Control 2 Register, CTRL2)

CTRL2 的地址为 0x19_0002。该寄存器的位定义取决于 ADC 工作在顺序模式还是并行模式。

如果工作在顺序模式下, CTRL2 定义为:

位	D15:5	D4:0
读	0	DIV
写		
复位	0:0	00010

D4~D0—DIV 位, 时钟分频选择位。该位控制分频电路, 使其产生的 ADC 时钟为系统时钟的 $1/(2*(DIV+1))$ 。DIV 必须确保 ADC 时钟不会超过 5MHz。具体见表 6-3 所示。

表 6-3 不同转换时钟源的 ADC 时钟频率

DIV	分频因子	ROSC旁路400kHz	ROSC正常8MHz	PLL64MHz	外部CLK
		200kHz Sys Clock	4MHz Sys Clock	32MHz Sys Clock	CLK/2 Sys Clock
00000	2	100kHz	2.00MHz	16.0MHz	CLK/4
00001	4	50kHz	1.00MHz	8.00MHz	CLK/8
00010	6	33.33kHz	500kHz	5.33MHz	CLK/12
00011	8	25kHz	250kHz	4.00MHz	CLK/16
00100	10	20kHz	125kHz	3.20MHz	CLK/20
—	—	—	—	—	—
11111	64	3.125kHz	62.5kHz	500kHz	CLK/128

如果工作在并行模式下，CTRL2 定义为：

位	D15	D14	D13	D12	D11	D10:6	D5	D4:0
读		STOP1		SYNC1	EOSIE1	0	SIMULT	DIV
写			START1					
复位	0	1	0	1	0	0	0	00010

当 ADC 工作在并行模式下，控制寄存器 2 用于控制转换器 B 的操作。如果是同步并行扫描，转换器 A 和 B 一起操作，它们由控制寄存器 1 的 STOP0、START0、SYNC0 和 EOSIE0 位共同控制，此时控制寄存器 2 的 STOP1、START1、SYNC1 和 EOSIE1 位对转换器 B 无效。如果是非同步并行扫描，以上控制寄存器 2 的这些位和 SYNC1 输入用于控制转换器 B 扫描，而控制寄存器 1 的相应位和 SYNC0 用于控制转换器 A 扫描。

D14—STOP1 位，停止转换位 1。0：转换器 B 处于正常工作模式；1：转换器 B 处于停止模式。非同步并行扫描时，当该位为 1 时，转换器 B 的当前扫描被停止，也不能开始新的扫描。SYNC1 信号输入或向 START1 位写 1 都会被忽略，直到 STOP1 位被清零。当 ADC 在停止模式下，结果寄存器可以被处理器修改，这和 AD 核的处理是一样的。因此，上下限检验、过零和相关中断都可以是被允许的。

D13—START1 位，开始转换位 1。该位为只写位。0：无动作；1：执行开始命令。非同步并行扫描时，向该位写 1，启动转换器 B 的一次扫描，该位为只写位。在当前扫描结束前再次写 1 会被忽略；在写该位之前，ADC 必须有稳定的电源配置。

D12—SYNC1 位，同步使能位 1。0：扫描动作只能通过写 START1 位来初始化；1：扫描动作可以由 SYNC1 脉冲输入或写 START1 位来初始化。非同步并行扫描时，设置该位后，SYNC1 端有正沿输入时，可以初始化转换器 B 一次转换。在扫描期间如果还有 STNC1 信号输入，将被忽略。在一次顺序或一次并行扫描模式下，只有第一个 SYNC1 输入信号是有效的，后续 STNC1 信号将被忽略，除非重写 SYNC1 位。这个重写动作可以在任何时候发生，包括在扫描期间。

D11—EOSIE1 位，扫描结束中断使能位 1。0：禁止该中断；1：使能该中断。非同步并行扫描时，该位使能转换器 B 的扫描结束中断。在循环扫描模式下，每一次迭代完成都会触发中断。

D5—SIMULT 位，同步模式位。该位只影响并行扫描模式。0：非同步并行扫描；1：同步并行扫描。

D4~D0—DIV 位，时钟分频选择位。与顺序模式下定义相同。

3. 过零控制寄存器（Zero Crossing Control Register, ADZCC）

ADC 过零逻辑可以监视采样槽 0-7 的当前转换结果和上一次转换结果的符号位是否发生改变，并且可以定义过零方向。ZCE0-7 分别监视采样槽 SAMPLE0-7 的 AD 转换，当关闭对某个采样槽的过零监视时，即使该采样槽的转换结果和上一次转换结果之间有符号的改变，也不会更新过零状态寄存器 ADZCSTAT。ADZCC 的地址为 0x19_0004。该寄存器复位为 0。ADZCC 定义为：

位	D15:14	D13:12	D11:10	D9:8	D7:6	D5:4	D3:2	D1:0
读/写	ZCE7	ZCE6	ZCE5	ZCE4	ZCE3	ZCE2	ZCE1	ZCE0

D15~D0—ZCEn 位，过零使能位。00、01、10、11 分别代表过零禁止、使能正变负的过零、使能负变正的过零和使能任何符号变化的过零。

4. 通道列表寄存器1和2（Channel List Registers 1、2，ADLST1、2）

这两个寄存器定义了对模拟量输入的扫描顺序，它们提供了 8 个采样槽 SAMPLEn，每个采样槽可对应 8 路模拟量输入 ANn 的任意一路。因此，可以以任意次序采样这些模拟量输入，还可以多次采样同一个模拟量输入。如果在采样禁止寄存器 ADSDIS 中开放所有采样槽，那么顺序扫描模式以 SAMPLE0-7 的次序进行扫描，并行扫描模式中转换器 A 以 SAMPLE0-3 的次序，转换器 B 以 SAMPLE4-7 的次序进行扫描。由于转换器 A 只能采样模拟量输入 AN0-3，而转换器 B 只能采样模拟量输入 AN4-7，所以 ADLST1 寄存器 SAMPLE3-0 域值的范围是 0b000-0b011，ADLST2 寄存器 SAMPLE4-7 域值的范围是 0b100-0b111。如果 SAMPLEn 域值超出所定义的范围，不会招致损害，但转换结果是未定义的。

ADLST1 的地址为 0x19_0006。定义为：

位	D15	D14:12	D11	D10:8	D7	D6:4	D3	D2:0
读		SAMPLE3		SAMPLE2		SAMPLE1		SAMPLE0
写								
复位	0	011	0	010	0	001	0	000

ADLST2 的地址为 0x19_0008。定义为：

位	D15	D14:12	D11	D10:8	D7	D6:4	D3	D2:0
读		SAMPLE7		SAMPLE6		SAMPLE5		SAMPLE4
写								
复位	0	111	0	110	0	101	0	100

SAMPLEn—采样输入通道选择 n 位。该字段的设置如表 6-4 所示。

表 6-4 ADC 输入转换的采样位数

顺序模式	SAMPLEn[2:0]		ADC输入引脚选择	
	并行模式			
n=0,1,2,...,7	n=0,1,2,3 (转换器A)	n=4,5,6,7 (转换器B)	单端	多端
000	000		AN0	AN0+,AN1-
001	001		AN1	
010	010		AN2	
011	011		AN3	AN2+,AN3-
100		100	AN4	AN4+,AN5-
101		101	AN5	
110		110	AN6	
111		111	AN7	AN6+,AN7-

5. 采样禁止寄存器（Sample Disable Register，ADSDIS）

采样禁止寄存器 ADSDIS 配合通道列表寄存器 1 和 2 使用，它提供了使能所需采样槽 SAMPLEn 的功能。复位时 SAMPLE0-SAMPLE7 都是被使能的。例如，在顺序扫描模

式下, DS5 被置位, 那么 SAMPLE5 被禁止, 扫描始于 SAMPLE0, 终于 SAMPLE4, SAMPLE5 及其之后的采样槽将不被扫描。再例如, 在并行模式下, DS5 和 DS1 被置位, 那么就只扫描 SAMPLE0 和 SAMPLE4。ADSDIS 的地址为 0x19_000A。该寄存器复位为 0。ADSDIS 定义为:

位	D15:8	D7:0							
读	0:0	DS7	DS6	DS5	DS4	DS3	DS2	DS1	DS0
写									

DSn—禁止允许/禁止采样位。0: 使能 SAMPLEn; 1: 禁止 SAMPLEn 及其之后的所有采样槽, 是否真正被禁止, 取决于转换模式 (顺序/并行) 以及 SIMULT 的值。

6. 状态寄存器 (Status Register, ADSTAT)

状态寄存器 ADSTAT 跟踪了 ADC 模块的当前转换状态, 除了 CIP0 和 CIP1, 该寄存器中的所有其他状态位都是粘性的, 即它们必须在执行完指定的操作后才能被清除, 而不会在下一次扫描周期开始时自动被清除。ADSTAT 的地址为 0x19_000C。定义为:

位	D15	D14	D13	D12	D11	D10	D9	D8	D7:0
读	CIP0	CIP1	0	EOSI1	EOSI0	ZCI	LLMTI	HLMTI	RDY7:0
写									
复位	0	1	0	0	0	0	0	0	0

D15、D14—CIP0 位、CIP1 位, 正在转换位 0、1。CIP0 指示是否正在进行扫描。CIP0 用于顺序扫描和同步并行扫描。如果是在非同步并行扫描模式下, CIP0 用于指示转换器 A 的扫描是否在处理中, 而 CIP1 用于指示转换器 B 的扫描是否在处理中。这两位在下一次扫描周期开始时会被自动清零。CIP0、CIP1 = 0, 空闲状态; CIP0、CIP1 = 1, 一次扫描周期正在处理中, 在该状态下, ADC 将忽略所有的 SYNC 脉冲和开始命令。

D12、D11—EOSI1、EOSI0 位, 扫描结束中断请求位 1、0。该两位表明自从上一次读或者复位状态寄存器, 一次扫描周期是否完成, 只是用于不同的扫描模式。EOSI0 用于顺序扫描、同步并行扫描和非同步并行扫描的 A 转换器, 而 EOSI1 用于非同步并行扫描的 B 转换器。在循环扫描模式下, 这两个中断请求位在每一次循环迭代完成都会置位。如果没有使能扫描完成中断, 那么这两位可用于查询是否扫描完成。向 EOSI1、EOSI0 位写 1 可以清零这两位, 并且它们不能软件置位。EOSI1、EOSI0 = 00, 一个扫描周期还没结束, 没有未处理的扫描结束的中断请求; EOSI1=1 或 EOSI0 = 1, 一个扫描周期已结束, 有未处理的扫描结束中断请求。

D10—ZCI 位, 过零中断请求位。如果那些使能过零检查的采样槽的转换结果和上一次转换结果相比 (比较的是结果寄存器中的值), 符号位发生了变化 (这种变化方式可以在过零控制寄存器中定义), 那么该位被置位, 如果 CTRL1[ZCIE] = 1, 则产生中断。否则, 没有过零中断请求, 该位为零。一有过零事件发生, 该位就会被置位, 而不是要等到一次扫描周期结束。向过零状态寄存器 ADZCSTAT 中所有有效的过零状态位 ZCSn 写 1, 才能清零 ZCI 位。

D9—LLMTI 位, 超下限中断请求位。0: 没有超下限中断请求; 1: 有超下限中断请求。下限寄存器 ADLLMTn 中如果是一个非 0 值, 那么就对相应采样槽的原始转换结果

进行下限检查，检查在转换完成时发生，而不一定要等到扫描结束。向限制状态寄存器 ADLSTAT 中所有有效的超下限状态位 LLSn 写 1，才能清零 LLMTI 位。

D8—HLMTI 位，超上限中断请求位。0：没有超上限中断请求；1：有超上限中断请求。上限寄存器 ADHLMtN 中的值只要不是 0x7FF8，那么就对相应采样槽的原始转换结果进行上限检查，检查在转换完成时发生，而不一定要等到扫描结束。向限制状态寄存器 ADLSTAT 中所有有效的超上限状态位 HLSn 写 1，才能清零 HLMTI 位。

D7~D0—RDYn 位，转换结果准备好位。RDY0-7 与采样槽 SAMPLE0-7 以及结果寄存器 ADRSLT0-7 一一对应。采样槽 SAMPLEn 转换完成并将结果存于相应的 ADRSLTn，对应的 RDYn 标志就会被置位，表明转换结果准备好，可以读了；否则该位为 0。读结果寄存器后，相应的 RDYn 就被清零。在调试期间，如果调试器读了某个结果寄存器，相应的 RDYn 也会被清零。要注意，在所有使能的采样槽转换完成之前，不要启动新的扫描。

7. 限制状态寄存器（Limit Status Register, ADLSTAT）

ADC 限制状态寄存器 ADLSTAT 记录了采样槽的原始转换结果和各自的上下限寄存器的比较结果。例如，如果采样槽 0 的原始转换结果比它的上限值大，那么 HLS0 就会被置位，如果 CTRL1[HLMTIE] = 1，会发生超上限中断。这些状态位是粘性的，它们不会再下一次转换周期开始时自动清零，必须向 HLSn 或 LLSn 写 1 才能清零。ADLSTAT 的地址为 0x19_000E。该寄存器各位复位时为 0。ADLSTAT 定义为：

位	D15:8	D7:0
读/写	HLS7:0	LLS7:0

D15~D8—HLSn 位，超上限状态位。0：采样槽 n 的原始转换值小于等于相应的上限值；1：采样槽 n 的原始转换值大于相应的上限值。

D7~D0—LLSn 位，超下限状态位。0：采样槽 n 的原始转换值大于等于相应的下限值；1：采样槽 n 的原始转换值小于相应的下限值。

8. 过零状态寄存器（Zero Crossing Status Register, ADZCSTAT）

ADC 过零状态寄存器 ADZCSTAT 记录了结果寄存器 ADRSLTn 中的当前值和上一次值的符号位比较结果。例如，如果采样槽 0 的转换结果和上一次比较，符号位发生了改变，并且过零控制寄存器 ADZCC 的 ZCE0 位设为 0b11，那么任意过零方向都会使 ZCS0 置位，如果 CTRL1[ZCIE] = 1，会发生过零中断。向 ZCSn 位写 1，可以清零 ZCSn 位。ADZCSTAT 的地址为 0x19_0010。该寄存器各位复位为 0。ADZCSTAT 定义为：

位	D15:8	D7:0
读	0:0	ZCSn
写		

D7~D0—ZCSn 位，过零状态位。0：转换结果未改变符号，或者过零检查被禁止；1：转换结果的符号有变化。

9. 结果寄存器（Result Registers, ADRSLTn）

采样槽 SAMPLEn 的最终转换结果被载入到对应的结果寄存器 ADRSLTn 中。例如，在同步并行扫描模式下，SAMPLE0 和 SAMPLE4 同步扫描，它们的转换结果分别放在的 ADRSLT0 和 ADRSLT4 中。ADRSLTn 中只能写 RSLT 部分，SEXT 不能直接写入，它是由原始转换结果和偏移量相减后决定的。

结果寄存器中的数可以理解为由符号整数或有符号定点（小）数。作为定点数，结果寄存器中的值可以直接使用；作为整数，就有两种处理方式：将结果连同符号位向右移 3 位；或者全盘接受寄存器中的结果格式，但要清楚这个结果的低三位添了 0。

负的结果（SEXT = 1）总以补码形式表示。如果应用程序中要求结果总为正数，那么相应的偏移量寄存器 ADOFSn 必须设为 0。

ADRSLTn 地址为 0x19_0012~0x19_0020。这 8 个寄存器用来存放一次扫描后的转换结果。该寄存器各位复位为 0。ADRSLTn 定义为：

位	D15	D14:3	D2:0
读	SEXT	RSLT	000
写			

D15—SEXT 位，符号扩展位。0：结果为正数；1：结果为负数。

D14~D3—RSLT 位，转换结果位。

10. 下限寄存器（Low Limit Registers, ADLLMTn）和上限寄存器（High Limit Registers, ADHLMTn）

ADC 转换的原始结果在进行偏移量修正之前，可能会跟上、下限寄存器中的值比较，如果超出范围会置相关标志位。如果上限寄存器中写入 0x7FF8，下限寄存器中写入 0x0000 就会禁止对结果超限的检查。复位时，超限检查是被禁止的。

下限寄存器 ADLLMTn 的地址为 0x19_0022~0x19_0030。这些寄存器复位为 0。ADLLMTn 定义为：

位	D15	D14:3	D2:0
读	0	LLMT	000
写			

D14~D3—LLMT 位，下限位。

上限寄存器 ADHLMTn 的地址为 0x19_0032~0x19_0040。这些寄存器复位为 0。ADHLMTn 定义为：

位	D15	D14:3	D2:0
读	0	HLMT	000
写			

D14~D3—HLMT 位，上限位。

11. 偏移量寄存器（Offset Registers, ADOFSn）

原始 AD 转换值要减去相应偏移量寄存器中的值后，才存入到对应的结果寄存器中。如果要获得无符号 AD 转换结果（0x0000-0x7FF8），那么相应的偏移量寄存器必须写入

0x0000。ADOF_{Sn} 的地址为 0x19_0042~0x19_0050。该寄存器复位为 0。ADOF_{Sn} 定义为：

位	D15	D14:3	D2:0
读	0	OFFSET	000
写			

D14~D3—OFFSET 位，偏移量位。

12. 电源控制寄存器（Power Control Register, POWER）

该寄存器管理 ADC 模块的电源。在 ADC 中每个转换器和参考电压发生器都能独立的被配置为掉电状态，掉电后，相应的模块就不消耗电能。从一个掉电的转换器引用到的扫描结果是未定义的。要使用 ADC 模块，参考电压发生器和至少一个 ADC 转换器必须上电。每一个转换器和参考电压发生器有一个手动电源控制位用于强制相应的模块进入掉电状态，相应的也有自动上电和掉电模式。当两个转换器都没有在处理扫描，ADC 模块是空闲的。如果两个转换器中至少有一个在扫描，那么 ADC 模块处在激活状态。ADC 模块有两种电流模式：正常电流模式用于给工作频率在 100KHz 以上的转换器提供电能；低功耗电流模式供给更少的电能，它仅为工作在 100KHz 频率下的转换器提供电能。激活电流模式后并不会影响一次 AD 转换所需的 ADC 时钟周期数或是转换精度。当空闲作为省电策略的一种，ADC 模块就会在这两种电流模式之间切换。两个转换器总是处在同样的电流模式下。在自动掉电和自动低功耗电源模式下，当 ADC 模块在空闲和激活状态切换时，需要一段启动延时时间，用于切换时钟或电源配置。延时所需的 ADC 时钟数在 PUDELAY 域定义。POWER 的地址是 0x19_0052，定义为：

位	D15	D14	D13	D12	D11	D10	D9:4	D3	D2:0
读	ASB	0	0	PSTS2	PSTS1	PSTS0	PUDELAY	APD	PD2:0
写									
复位	0	1	0	0	0	0	001101	0	111

D15—ASB 位，自动低功耗位。0：禁止自动低功耗模式；1：使能自动低功耗模式。该位用于选择自动低功耗模式。如果 APD 被置位，那么 ASB 将被忽略。自动低功耗模式选择低功耗时钟作为 ADC 时钟源，并且将转换器的电源模式设为低功耗电源模式。在扫描开始时，转换时钟被选为 ADC 时钟，在使能转换器后到扫描开始之前会强制插入 PUDELAY 个 ADC 时钟周期的延时，以稳定对 ADC 的供电。在延时之后，ADC 对扫描进行初始化。当 ADC 回到空闲状态，转换器回到低功耗电流模式并且 ADC 时钟使用低功耗时钟。

D12—PSTS2 位，参考电压电源状态位。0：当前参考电压电路使能；1：当前参考电压电路禁止。

D11—PSTS1 位，转换器 B 电源状态位。0：当前转换器 B 使能；1：当前转换器 B 禁止。置位 PD1 位后，该位立即也被置位。如果自动掉电模式未被激活（APD=0），手动上电转换器 B（清零 PD1），在启动扫描前应等待 PUDELAY 个 ADC 时钟周期的延时。该位可以作为一个状态位来被查询，以决定转换器 B 是否准备好被操作。在自动掉电模式下，该位表明转换器 B 电源的当前状态。

D10—PSTS0 位，转换器 A 电源状态位。0：当前转换器 A 使能；1：当前转换器 A

禁止。功能与 D11 类似。

D9~D4—PUDELAY 位，上电延时位。在转换器 A 和 B 上电之后（清零 PD0 和 PD1 后）启动扫描之前需要一段延时时间，由 PUDelay 位决定延时所需的 ADC 时钟周期数。在自动掉电和自动低功耗模式下，在 ADC 从空闲到激活和启动扫描之间的延时也是由该位决定。默认延时 13 个 ADC 时钟周期，这个值能够满足任何电源模式的要求。可以减小 PUDelay 值，以缩短上电延时时间而不会影响到转换精度。如果 PUDelay 值设置的太小，那么在扫描转换刚开始时，转换精度会降低。

D3—APD 位，自动掉电模式位。0：未激活自动掉电模式；1：激活自动掉电模式。在自动掉电模式下，当转换器未用于扫描时就会被禁止。该位的优先权高于 ASB 位。在 APD 模式下，在转换器从空闲态到开始扫描之间有 PUDelay 个 ADC 时钟周期的延时，扫描完成后，转换器再次被禁止。如果在扫描期间，置位 ASB 或 APD，扫描不会受影响，并且在转换器都完成转换进入空闲态后，ADC 模块就处于低电能状态。ADC 在循环扫描模式下永远不会进入低电能状态，因此该模式下，ASB 和 APD 位是没有用的。

D2—PD2 位，参考电压电路掉电控制位。0：手动上电电压参考电路；1：由 PD0 和 PD1 控制电压参考电路掉电（默认）。参考电压电路由两个转换器所共享。如果置位 PD2 位，那么当转换器 A 或转换器 B 上电（PD0 = 0 或 PD1 = 0）时，该参考电压电路也处于上电激活状态，当转换器 A 或转换器 B 掉电，该参考电压电路自动掉电。

D1—PD1 位，转换器 B 的手动掉电。0：ADC 转换器 B 上电；1：ADC 转换器 B 掉电。置位该位立刻强制转换器 B 掉电，之后再引用转换器 B 的扫描结果是无效的。当清零 PD1 位之后，转换器 B 持续上电（AP0 = 0），或在需要时自动上电（APD = 1）。当在除了自动掉电以外模式的电源模式下，清零 PD1 位给转换器 B 上电，需要有上电延时以使得转换器内的电源稳定，可以查询 PSTS1 位，检查延时是否完成。

D0—PD0 位，转换器 A 的手动掉电。0：ADC 转换器 A 上电；1：ADC 转换器 A 掉电。置位该位立刻强制转换器 A 掉电，之后再引用转换器 A 的扫描结果是无效的。当清零 PD0 位之后，转换器 A 持续上电（AP0 = 0），或在需要时自动上电（APD = 1）。当在除了自动掉电以外模式的电源模式下，清零 PD0 位给转换器 A 上电，需要有上电延时以使得转换器内的电源稳定，可以查询 PSTS0 位，检查延时是否完成。

13. 电压参考寄存器（Voltage Reference Register, CAL）

电压参考寄存器 CAL 的地址为 0x19_0054，该寄存器复位为 0。CAL 定义为：

位	D15	D14	D13:0
读	SEL_VREFH	SEL_VREFL	0:0
写			

D15—SEL_VREFH 位，V_{REFH} 源选择位。0：V_{REFH} 的源来自内部 V_{RH}；1：V_{REFH} 的源来自外部引脚 AN2。

D14—SEL_VREFL 位，V_{REFL} 源选择位。0：V_{REFL} 的源来自内部 V_{RL}；1：V_{REFL} 的源来自外部引脚 AN6。

6.4 基于构件的AD编程

6.4.1 AD构件的函数原型设计

AD 模块具有初始化、采样、中值滤波、均值滤波等操作。按照构件的思想，可将它们封装成独立的功能函数，以下是各功能函数的原型设计：

- (1) AD 转换模块初始化

```
void ADinit (void)
```

- (2) 采样一次 ANn (0-7) 路模拟量的 AD 值

```
uint16 AOnce(uint16 ANn)
```

- (3) 采用中值滤波，获取 ANn (0-7) 路模拟量的 AD 值

```
uint16 AMid(uint16 ANn)
```

- (4) 采用均值滤波，获取 ANn (0-7) 路模拟量的 AD 值，N 为均值滤波次数

```
uint16 ADave(uint8 N, uint16 ANn)
```

6.4.2 AD构件的头文件

AD 构件头文件中主要包括相关宏定义、AD 的功能函数原型说明等内容。

```
#ifndef _AD_H
#define _AD_H
//头文件
#include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
#include "sysinit.h" //配置头文件
#include "type.h" //类型定义头文件
#include "GeneralFun.h" //包含通用函数头文件
//-----*
//函数名:ADinit *
//功 能:AD转换模块初始化 *
//参 数:无 *
//返 回:无 *
//-----*
void ADinit (void); //AD转换模块初始化
//-----*
//函数名:AOnce *
//功 能:采集一次AN0-AN7中任意一路的模拟量的AD值 *
//参 数:ANn:模拟量输入端口号(范围:0-7) *
//返 回:无符号结果值(范围:0-4095) *
//-----*
uint16 AOnce(uint16 ANn); //采集某路模拟量的AD值
//-----*
//函数名:AMid *
//功 能:中值滤波,获取AN0-AN7中任意一路的模拟量的AD转换结果的中值滤波值 *
//参 数:ANn:模拟量输入端口号(范围:0-7) *
```

```

//返回:中值滤波后的结果(范围:0-4095) *
//-----*
uint16 ADmid(uint16 ANn); //中值滤波
//-----*
//函数名:ADave *
//功能:均值滤波,对 AN0-AN7 中任意一路的模拟量的 AD 转换结果求 N 次平均值, *
//      实现均值滤波 *
//参数:N:均值滤波次数(范围:0~255) *
//      ANn:模拟量输入端口号(范围:0-7) *
//返回:平均值滤波后的结果(范围:0-4095) *
//-----*
uint16 ADave(uint8 N, uint16 ANn); //均值滤波
#endif

```

6.4.3 AD构件的源程序文件

AD 构件程序文件的内容是给出 AD 各功能函数的实现过程。ADinit 函数主要完成 A/D 转换模块的初始化。ADmid 函数实现中值滤波, ADave 函数实现均值滤波。中值滤波的思想是:连续采样 3 次,并将 3 次采样结果按照从大到小的顺序排列,取中间值作为中值滤波的结果。均值滤波的思想是:先进行 n 次中值滤波,再将这 n 次中值结果相加,然后除以采样次数 n 得到最终结果。

```

//[AD.c]A/D转换驱动文件-----*
//本文件包含: *
//      (1)ADinit:A/D转换模块初始化 *
//      (2)ADonce:采集一次AN0-AN7中任意一路的模拟量的AD值 *
//      (3)ADmid:中值滤波 *
//      (4)ADave:均值滤波 *
//硬件连接: *
//      MCU的A/D模块ANx引脚接传感器采样脚 *
//-----*
#include "AD.h" //A/D转换驱动文件
//-----*
//函数名:ADinit *
//功能:AD转换模块初始化 *
//参数:无 *
//返回:无 *
//-----*
void ADinit (void) //AD转换模块初始化
{
    //1. 循环顺序扫描模式, AN0-7设为单端输入, 禁止中断, 停止转换
    MCF_ADC_CTRL1 = 0b010100000000010;
        //      ||| ||| ||| ||| ||| ||| ||| ||| SMODE=2, 循环顺序扫描
        //      ||| ||| ||| ||| ||| ||| ||| CHNCFG=0, 8路ANn都设为单端输入
        //      ||| ||| ||| ||| ||| ||| ||| HLMTIE=0, 禁止超上限中断
        //      ||| ||| ||| ||| ||| ||| ||| LLMTIE=0, 禁止超下限中断
        //      ||| ||| ||| ||| ||| ||| ||| ZCIE=0, 禁止过零中断

```

```

//      |||_____ EOSIE0=0, 禁止转换完成中断
//      ||_____ SYNC0=1, 由SYNC0输入或写START0启动扫描
//      |_____ START0=0, 未启动扫描
//      |_____ STOP0=1, 停止转换
//2. ADC时钟频率3MHz
MCF_ADC_CTRL2 = 0b0000000000001001;
//      |||_____ DIV=20, ADC时钟频率3MHz
//      (fBUS=60MHz, fAD=fBUS/(9+1)*2=3MHz)
//3. 转换器A和B和参考电压电路上电, 上电延时10个ADC时钟周期
MCF_ADC_POWER = 0b0000000010100100;
// | | | | | | | | _____ PD0=0, 转换器A上电
// | | | | | | | | _____ PD1=0, 转换器B上电
// | | | | | | | | _____ PD2=1, 参考电压电路上电依赖于转换器A/B
// | | | | | | | | _____ APD=0, 禁止自动掉电
// | | | | | | | | _____ PUDELAY=10, 上电延时为10个ADC时钟周期
// | | | | | | | | _____ ASB=0, 禁止自动低功耗模式
//4. 使用内部参考电压VRH和VRL
MCF_ADC_CAL = 0b0000000000000000;
// || _____ SEL_VREFH=0, SEL_VREFL=0
//      使用内部参考电压VRH和VRL
//5. 使用A口的第一功能——AD转换
MCF_GPIO_PANPAR = 0xFFFF;
//6. 清停止0位, STOP0=0
MCF_ADC_CTRL1 &= 0xBFFF;
//7. 等待转换器A上电完成
while (MCF_ADC_POWER & MCF_ADC_POWER_PSTS0);
//8. 等待转换器B上电完成
while (MCF_ADC_POWER & MCF_ADC_POWER_PSTS1);
//9. 向START0写1, 启动扫描
MCF_ADC_CTRL1 |= 0x2000;
return;
}
//-----*
//函数名:ADonce *
//功 能:采集一次AN0-AN7中任意一路的模拟量的AD值 *
//参 数:ANn:模拟量输入端口号(范围:0-7) *
//返 回:无符号结果值(范围:0-4095) *
//-----*
uint16 ADonce(uint16 ANn) //采集某路模拟量的AD值
{
    uint16 result;
    //等待数据准备好
    while ((MCF_ADC_ADSTAT & (1 << ANn))==0);
    //取得结果
    result = (MCF_ADC_ADRSLT(ANn) & 0x7FF8) >> 3;
    return result;
}
//-----*
//函数名:ADmid *
```

```

//功 能:中值滤波,获取AN0-AN7中任意一路的模拟量的AD转换结果的中值滤波值      *
//参 数:ANn:模拟量输入端口号(范围:0-7)                                          *
                                                                    *
//返 回:中值滤波后的结果(范围:0-4095)                                          *
//-----*
uint16 ADmid(uint16 ANn) //中值滤波
{
    uint16 i,j,k,tmp;
    //1.取 3 次 A/D 转换结果
    i = ADonce(ANn);
    j = ADonce(ANn);
    k = ADonce(ANn);
    //2.取中值
    if (i > j)
    {
        tmp = i; i = j; j = tmp;
    }
    if (k > j) tmp = j;
    else if(k > i) tmp = k;
    else tmp = i;
    return tmp;
}

//-----*
//函数名:ADave                                                                    *
//功 能:均值滤波,对AN0-AN7中任意一路的模拟量的AD转换结果求N次平均值,          *
//      实现均值滤波                                                                *
//参 数:N:均值滤波次数(范围:0~255)                                                *
//      ANn:模拟量输入端口号(范围:0-7)                                            *
//返 回:平均值滤波后的结果(范围:0-4095)                                          *
//-----*
uint16 ADave(uint8 N, uint16 ANn) //均值滤波
{
    uint32 tmp = 0;
    uint8 i;
    for(i = 0; i < N; i++)
        tmp += ADmid(ANn);
    tmp = tmp / N;
    return (uint16)tmp;
}

```

在 AD 的初始化函数 ADinit 中没有对过零控制寄存器 (ADZCC)、通道列表寄存器 1 和 2 (ADLST1 和 ADLST2)、采样禁止寄存器 (ADSDIS)、上限和下限寄存器 (ADLLMTn 和 ADHLMtn)、以及偏移量寄存器 (ADOFsn) 等进行设置,而是使用这些寄存器复位时的默认值。默认情况下,不对结果寄存器中的值进行过零检查;采样槽 SAMPLE0-7 分别与模拟量输入通道 ANn0-7 一一对应;开放全部 8 个采样槽;不对原始转换结果进行超上下限检查;8 个结果寄存器中存放无符号转换结果。初始化时,还将 AD 模块配置为单端输入的循环顺序扫描模式,在随书光盘中,提供了单端差分混合型输入的单次非同步并

行扫描的 AD 程序实例，两者的主要差别在于对控制寄存器 1 和 2 的设置，这里不再展开介绍。值得注意的是，如果 AN_x 和 AN_y 是一对差分输入，那么读 AN_x 或 AN_y 对应的结果寄存器都能获得该差分输入的转换结果。通过这两个例子的比较学习，对于 MCF52233 AD 模块的其他运行模式，读者应该也能举一反三、触类旁通了。

6.5 MCF52233 AD模块编程实例——物理量采集回归系统

模拟量数据采集的典型过程为：实际物理量→传感器→变换器→放大器→A/D 转换器→计算机（MCU、DSP 或通用计算机）。按照这个流程，本节给出的物理量采集回归系统，分为数据采集转换和 AD 值回归两个部分，分别由低端 MCU 方和高端 PC 方来完成。本系统采用 MCF52233 内置的 ADC 模块配合各种传感器和相应的放大电路完成物理量的采集和转换。MCF52233 的 ADC 模块有 AN0~AN7 共 8 路模拟量输入，首先依次采集这 8 路模拟量，并将这 8 个 12 位转换结果（在 MCU 中每个转换结果用 16 位变量存储）存入内存；然后 MCU 和 PC 通过串口交换数据：MCU 向 PC 发送 0x56，表示 MCU 数据采集转换完成，等待 PC 查询接收，PC 再发给 MCU 0x56 和 0x2F 两个字节，通知 MCU 可以发送 AD 结果值；最后 MCU 将 8 个转换结果共 16 字节发送给 PC 进行处理。

6.5.1 低端软件设计

在模拟量数据采集过程中，不可避免地会受到随机噪声的干扰，造成采集的数字量不准确，有时甚至很难满足实际的应用。为了防止脉冲干扰，可采用一些滤波方法，使得最终传送给高端的结果更准确稳定。常用的滤波方法有中值滤波和均值滤波。本系统采用了均值滤波，滤波次数取值为 200。

低端软件的主程序（main.c）的执行流程是：

- （1）对运行指示灯、UART 模块和 AD 模块进行初始化；
- （2）点亮运行指示灯；
- （3）采用均值滤波，将 8 路模拟量进行 A/D 转换，结果存入 ADresult 数组；
- （4）向 PC 发送握手信号 0x56；
- （5）若接收到 PC 的应答信号：0x56 和 0x2F，则向 PC 发送 16 字节的 AD 转换值；
- （6）熄灭运行指示灯，延迟；
- （7）转第（2）步；

显然，按照构件的程序设计思想，在主程序的实现过程中，需调用 AD 构件、UART 构件以及 Light 构件的相关功能函数。具体代码如下：

```
//-----*
// 工 程 名: AD *
// 硬件连接: AN0~AN7接传感器的采样脚 *
// 程序描述: 响应高端请求, 获取8路AD转换结果, 并均值滤波, 通过串口发出去. *
// 目 的: 掌握AD转换的基本编程方法 *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
```

```

//头文件
#include "includes.h"
int main(void)
{
    uint16 ADresult[8]; //存放8路AD转换结果
    uint8 i, flag;       //是否接收到PC发送的应答信号的标志
    uint8 RevData[2];    //存放PC发送的两个字节的应答信号
    //1. 关总中断
    DisableInterrupts();
    //2. 模块初始化
    Light_Init(Light_Run_PORT, Light_Run, Light_OFF); //运行指示灯初始化
    UartInit(0, SYSTEM_CLOCK, 9600, 0); //UART0初始化
    ADinit(); //AD初始化
    //3. 主循环
    while (1)
    {
        //3.1 点亮Light_Run
        Light_Control(Light_Run_PORT, Light_Run, Light_ON);
        //3.2 采样8路模拟量并进行A/D转换
        for(i=0; i<8; i++)
        {
            ADresult[i]=ADave(200, i);
        }
        //3.3 发送握手0x56, MCU通知PC, 数据采集转换完成
        UartSend1(0, 0x56);
        //3.4 接收PC握手, PC通知MCU, 可以发送AD数据
        flag=UartReN(0, 2, RevData);
        if((flag==0)&&(RevData[0]==0x56)&&(RevData[1]==0x2f))
        {
            UartSendN(0, 16, (uint8 *)ADresult); //向PC方发送数据
        }
        //3.5 熄灭Light_Run
        Light_Control(Light_Run_PORT, Light_Run, Light_OFF);

        Delay(40);
    }
    return 0;
}

```

6.5.2 高端软件设计

MCU 方传送给 PC 的 AD 值（数字量）和实际物理量是非线性的。对于非线性问题，通常的处理方法是：首先在硬件上尽可能减少非线性，然后在软件上再做一些处理。在本系统中，PC 方对得到的 AD 值采用分段直线回归的方法，以实现快速、动态、准确的 AD 转换校正。

动态的分段直线在线校正的方法，即在线增加校正点，用线性动态校正实际曲线方法去反应模拟量（物理量）与数字量之间的复杂曲线关系。该动态在线校正的基本思想是在

屏蔽物理量至数字量之间通路的基础上,根据现场的需要,在线进行增加校正点,从校正点开始,过该点的回归直线一分为二,重新进行最小二乘估计逐段拟合线性参数。

动态在线校正方式的目的是寻找实际物理量 y 与计算机所能识别的数字量 x 之间的相邻校正点间的线性关系。不妨假设每个线性关系为 $y_i=a_ix_i+b_i$, $i=1, \dots, n$ (n 为分段直线个数)。其中, x_i , y_i 已知。 x_i 可由数据采集系统获得,它表示一条回归直线段的起始点或终点数字量值; y_i 可通过测量获得,它表示实际的物理量; a_i 和 b_i 是回归计算的参数。为了更好的表示每个线性关系的结果以及方便的进行模拟量数据校正,本系统使用后台数据库作为其数据结构以及数据存储的工具,具体格式如表 6-5 所示。

表 6-5 动态在线校正方式的数据结构及数据存储实例

变量	x_i	a_i	b_i
数据1	0	0.2840695	0
数据2	403	0.2516545	0.9732352
数据3	814	0.2081001	0.6105005
数据4	1633	0.1810512	-6.610500
数据5	2453	0.1306861	8.910500
数据6	3700	0.0566861	0
数据7	4095	1	0

根据上述分析,PC 方软件功能主要包括输入当前该数字量对应的物理量、计算动态在线校正的回归参数、将回归参数写入数据库以及计算某个具体数字量对应的物理量。PC 方软件的运行界面如图 6-1 所示。

PC 程序运行后,先选择采集路数,如第 3 路温度采集,采集的物理量对象是温度,单击“开始”按钮,在界面左侧的图片框中会显示关于温度物理量和 AD 值的校正坐标系,如果初始时在数据库中没有任何有关该路采集的校正数据,那么系统会要求用户输入初始的最低和最高物理量和相应的 AD 值。当开始采集后,在界面右侧会用红字实时显示采集来的 AD 和相应物理量值。单击“校准”按钮,则可实时校正某个 AD 值对应的物理量值。如果之前的某次校准不准确,用户可以在删除选择下拉框中删除某次校准数据。

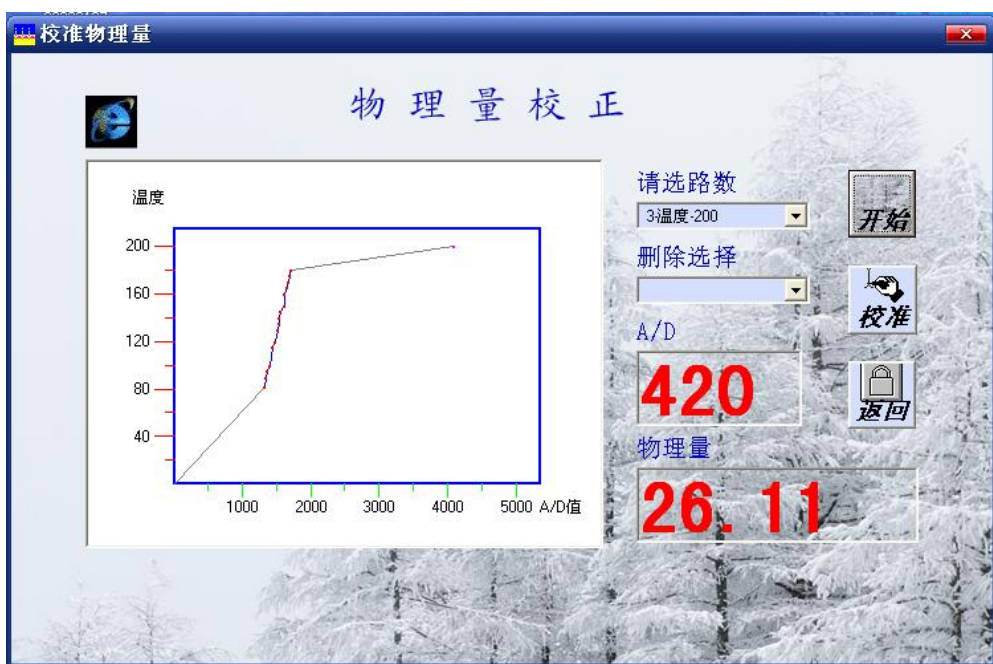


图6-1 高端程序运行界面

6.6 常用传感器简介及一个典型信号采样电路设计

传感器是指把物理量或化学量转变成电信号的器件，它是实现测试与自动控制系统的首要环节。如电子计价秤中所安装的称重传感器，是电子计价秤的重要部件，它担负着将重量转换成电信号的任务，该电信号被放大器放大并经 A/D 转换后，由显示器件给出称重信息。如果没有传感器对原始参数进行精确可靠的测量，无论是信号转换或信息处理都将无法实现。传感器的种类可分为力、热、湿、气、磁、光、电等。各种传感器都是根据相关材料在不同环境下会表现出不同的物理特性研制而成。下面先介绍一些简单的传感器，再以电阻型传感为例，给出其采集电路的设计和应用。

6.6.1 常用传感器简介

1. 温度传感器

温度传感器是利用一些金属、半导体等材料与温度有关的特性制成的，这些特性包括热膨胀、电阻、电容、磁性、热电势、热噪声、弹性及光学特征，根据制造材料将其分为热敏电阻传感器、半导体热电偶传感器、PN 结温度传感器和集成温度传感器等类型。热敏电阻传感器是一种比较简单的温度传感器，其最基本电气特性是随着温度的变化自身阻值



图6-2 热敏电阻器

也随之变化。图 6-2 是 NTC 热敏电阻器。

与热敏电阻相类似还有光敏、湿敏传感器，它们随着光线强弱、湿度不同，阻值也有所差别。下面以热敏电阻传感器介绍这类传感器的使用方法。

在实际应用中，将热敏电阻接入图 6-3 的电路图中，热敏电阻和一个特定阻值的电阻串联，由于热敏电阻会随着环境温度的变化而变化，因此 A/D 采样点的电压也会随之变化，A/D 采样点的电压为：

$$V_{A/D} = \frac{X}{R_{\text{热敏}} + X} \times V_{\text{REF}}$$

式中 x 是一特定阻值，根据实际热敏电阻的不同而加以选定。

假如热敏电阻阻值增大，采样点的电压就会减小，A/D 值也相应减小；反之，热敏电阻阻值减小，采样点的电压就会增大，A/D 值也相应增大。所以采用这种方法，MCU 就会获知外界温度的变化。如果想知道外界的具体温度值，就需要进行物理量回归操作，也就是通过 A/D 采样值，根据采样电路及热敏电阻温度变化曲线，推算当前温度值。

2. 灰度传感器

所谓灰度也可认为是亮度，简单的说就是色彩的深浅程度。灰度传感器的主要工作原理是它使用两只二极管，一只为发白光的高亮度发光二极管，另一只为光敏探头。

通过发光管发出超强白光照射在物体上，通过物体反射回来落在光敏二极管上，由于照射在它上面的光线强弱的影响，光敏二极管的阻值在反射光线很弱（也就是物体为深色）时为几百 $K\Omega$ ，一般光照度下为几 $K\Omega$ ，在反射光线很强（也就是物体颜色很浅，几乎全反射时）为几十 Ω 。这样就能检测到物体的颜色的灰度了。图 6-4 为设计好的一个灰度传感器实物图。

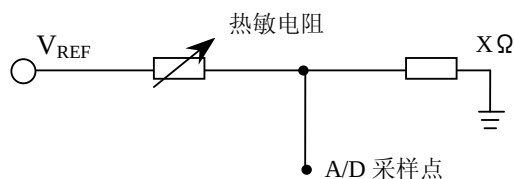


图6-3 热敏电阻传感器电路原理图



图6-4 灰度传感器实物图

3. 红外传感器

红外线是著名科学家赫歇尔在一次科学实验中发现的，他发现在太阳的可见光线以外存在着一种神奇的光线，人的肉眼无法看见这种光线，但它的物理特性与可见光线极为相似，有着明显的热辐射。由于它位于可见光中红光的外侧，故而称之为红外线，红外线的波长范围很宽，介于 $0.75 \sim 1000$ 微米之间，频率低于可见光。

红外传感器模块（如图 6-5 所示）可用来测量障碍物的距离，由红外发射管和红外接收管组成。其基本原理是：发射管发射红外光，当前方没有障碍物时，红外线就一直往前照射，一旦遇到障碍物后，红外光就会反射回来。红外线的反射光愈强，则说明障碍物的距离愈近。但当障碍物表面是黑色时，红外线容易被吸收，从而使反射光的强度减弱，这是它的不足之处。对于其他颜色的障碍物红外线都有很好的反射效果。

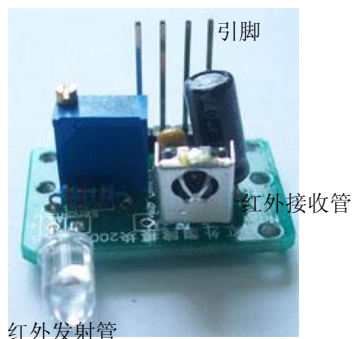


图6-5 红外传感器模块实物图

图 6-5 实物图中共有 4 个引脚接出，分别为接收信号引脚（IR）、电源+5V、地（GND）、发射信号引脚（ISR）。使用时，ISR 引脚接 MCU 某个具有 PWM 功能的 I/O 口，MCU 在该引脚上产生 40KHZ 的方波。IR 引脚接 MCU 某个具有输入捕捉功能的 I/O 口，当有反射的红外信号时，在该引脚上会产生下降沿信号。某次检测开始时，MCU 连续发送 N 个方波，并开始计时，输入捕捉 I/O 口有下降沿信号产生时，计时结束，根据这个延时及红外光的传播速度，就可以计算出障碍物的距离。

接收管实质是一种光电二极管，将光信号转变为电信号的元件。它的特点是被特定频率的光照射后，其阻值变小，光电二极管中就会有电流，光强度越大，阻值会越小。

接收管原理图如图 6-6 所示。当没有红外光照射时，光电二极管 P 的阻值很大， $V_{A/D} \approx 0$ ；当有光照射时，随着光照强度的增大， $V_{A/D}$ 也逐渐增大， $V_{A/D}$ 最大值 $\approx +5V$ 。

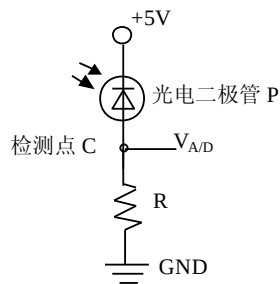


图6-6 红外接收管

4. 超声波传感器

超声是指当声音的频率高到超过人耳听觉的频率极限（根据大量调查，取整数 20000 赫兹）时，人们就觉察不出声音的存在，因而称这种高频率的声音为“超”声。超声波传感器按其工作原理可分为压电式、磁致伸缩式、电磁式等，而以压电式最为常用。压电式超声波传感器常用的材料是压电晶体和压电陶瓷，利用压电材料的压电效应来工作的。逆压电效应将高频电振动转换为高频机械振动，从而产生超声波，可作为发射器；而利用正压电效应，将超声振动波转换成电振荡，可作为接收器。所以超声波传感器实质上是一种可逆的换能器。

6.6.2 电阻型传感器采样电路设计

电阻型传感器即自身等效为一个电阻，电阻的阻值随外部信号的变化而变化，可用来采集温度等。

对于电阻型传感器的采集电路，基本思想是将电阻变化转化为电压变化，然后利用 AD 转换芯片得到电压值，最后利用 AD 值和外部信号的对照表得出当前外部信号的值。常用设计有恒流激励电路和恒压激励电路。

在恒流激励电路中,用恒定电流激励传感器,然后采集传感器两端电压值。由于传感器两端电压变化微弱,在用 AD 转换测电压之前,需先对电压值进行运算放大。最后,建立 AD 值同外部温度的对照表,在获得电压的 AD 值后,可使用对照表查出当前温度。通常,对照表是通过最后试验测试建立的,一般做法是首先测得特征点的 AD 值,而特征点之间的 AD 值通过线性插值得出。

通常,电阻型传感器采集电路由三部分组成:传感器接口、恒流源电路和放大电路。

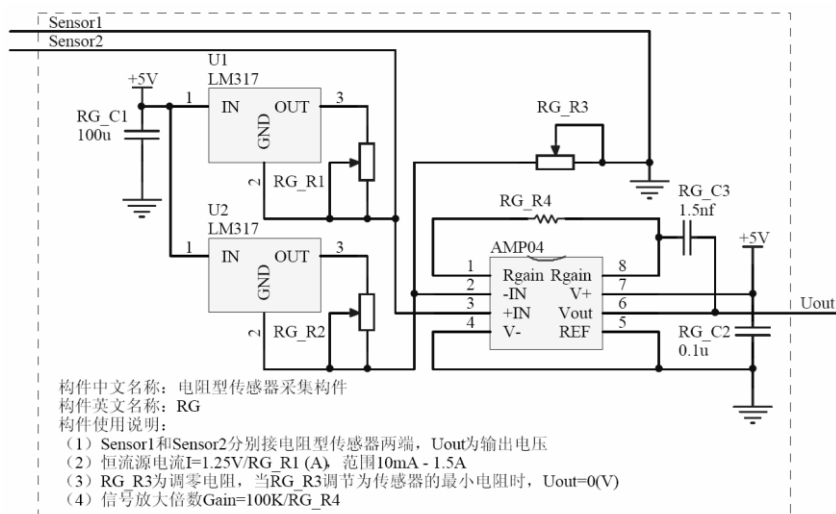


图6-7 电阻型传感器通用采集电路

图 6-7 为电阻型传感器通用采集电路，其中，Sensor1 和 Sensor2 为传感器接口，使用两个可调稳压器 LM317 提供 1.25V 的内部参考电压，一路供传感器使用，另一路供零参考电压电路使用，电阻 RG_R1 和 RG_R2 的阻值相等。恒流源的输出电流计算公式为 $I_{out}=1.25/RG_R1$ (A)。在传感器的可接受范围内，应采用尽可能大的输出电流，即 RG_R1 和 RG_R2 的阻值应尽可能的小，这样将在传感器两端产生尽可能大的电压差。电路对电源要求较高，在电源输入端加入电容 RG_C1 (100μF)，目的是过滤低频信号。

零参考电压电路中的 RG_R3 为精确电位器，用来调整零点。该电路中还使用了 AMP04 对传感器两端的电压进行放大。AMP04 放大电路的放大倍数由 RG_R4 的阻值决定，放大倍数 $\text{Gain} = 100\text{K} / \text{RG_R4}$ ，输出电压 $\text{Uout} = ((+IN) - (-IN)) * \text{Gain}$ 。

由上面的分析得知，可通过调节 RG_R1 和 RG_R2 的阻值来获得需要的电流，调节 RG_R4 的阻值来获得需要的放大倍数。

假设有一个传感器，其允许的输入电流为 5~10mA，电阻变化范围 10~20Ω，需要的放大倍数为 50，那么，如何配置 RG R1 和 RG R4 的阻值呢？

根据公式 $I_{out}=1.25/RG_R1$ ， I_{out} 设为传感器允许的最大电流即 10mA，因此 $RG_R1=1.25V/10\text{ mA}=125\Omega$

根据公式 $\text{Gain}=100\text{K}/\text{RG R4}$ ，放大倍数为 50，则 $\text{RG R4}=100\text{K}/50=2\text{K}\Omega$

第7章 定时器相关模块及其应用

本章讨论 MCF52233 可编程中断定时器 (PIT)、通用定时器 (GPT) 及脉宽调制模块 (PWM) 的基本原理, 并按照构件化编程原则给出各个模块的编程实例。

7.1 可编程中断定时器PIT

MCF52233 包括两个可编程中断定时器 (Programmable Interrupt Timers, PIT) 模块: PIT0 和 PIT1。每个 PIT 模块主要由一个 16 位的定时器组成, 提供了一个精确的周期性中断功能。使用 PIT 模块可以实现大量与定时器/计数器相关的应用, 例如系统时钟和周期性触发中断等功能。

7.1.1 PIT模块的编程寄存器

在介绍可编程中断定时器 PIT 模块的编程方法之前, 先简要介绍用于 PIT 模块的编程的三个寄存器: PIT 控制和状态寄存器 (PCSRn)、PIT 模数寄存器 (PMRn) 及 PIT 计数寄存器 (PCNTRn)。MCF52233 有两个可编程中断定时器 PIT 模块, 这里 “n” 表示 PIT 模块的编号, 取 0 或 1。

1. PIT控制和状态寄存器 (PCSRn)

PIT 控制和状态寄存器 PCSRn 是一个 16 位的寄存器, 主要用于配置 PIT 模块的操作, 复位后全为 0。PCSRn 中的 “n” 表示模块的编号, 取 0 或 1。

数据位	D15—D12	D11—D8	D7	D6	D5	D4	D3	D2	D1	D0
定义		PRE		DOZE	DBG	OVW	PIE	PIF	RLD	EN

D15~D12: 保留位。

D11~D8—PRE, 这四位用于设置 PIT 模块的计数时钟频率 (PRE[3:0])。计算公式如下, 其中 f_{sys} 为系统总线时钟, PRE[3:0]取值范围为 0000~1111, 对应的十进制数值为 0~15。

$$\text{PIT 模块的计数时钟频率} = f_{\text{sys}} \div 2 \div 2^{\text{PRE}[3:0]}$$

D7: 保留位。

D6—DOZE, 休眠模式位。0: 在休眠模式下, PIT 模块的操作不受影响。1: 在休眠模式下, PIT 模块是停止的。当离开休眠模式后, PIT 模块从进入休眠模式之前的状态继续运行。

D5—DBG, 调试模式位。该位控制 PIT 模块在调试模式下的功能。0: 启动 PIT 定时器。1: 停止 PIT 定时器。在调试模式下, 该位可以正常进行读写操作, 当离开调试模式后, PIT 模块从进入调试模式之前的状态继续运行, 但是在调试模式下所进行的修改依然保留。

D4—OVW, 重写位。0: 当 PIT 计数器到达 0x0000 后, PIT 模数寄存器 PMRn 中的

值将替代 PIT 计数器中的值。1: PMRn 中的值将立即替代 PIT 计数器中的值;

D3—PIE, PIT 中断使能位。若该位为 1, 则使能 PIF 中断请求, 反之则禁止 PIF 中断请求。

D2—PIF, PIT 计数器为 0 中断标志位。当 PIT 计数器到达 0x0000 后, 该位被置 1。此时如果 PCSRn 寄存器中的中断使能位 PIE 为 1, 则会产生相应中断请求。通过向该位写 1 或者写模数寄存器 PMRn 将清零该位。

D1—RLD, 重载位。0: 计数器载入 0xFFFF 然后继续进行减 1 计数。1: 当 PIT 计数器达到 0x0000 后, 计数器将载入 PMRn 中的值, 进行减 1 计数。

D0—EN, PIT 使能位。0: PIT 模块是禁止的, 计数器处于停止状态。1: PIT 是使能的, PIT 正常工作。

2. PIT模数寄存器 (PMRn)

该 16 位可读写寄存器保存将载入到 PIT 计数器中的值。当计数器达到 0x0000 并且 PCSRn 的 RLD 位为 1 时, 该寄存器中的值将载入到 PIT 计数器中。特别地, 当 PCSRn 的 OVW 为 1 时, PMRn 寄存器中的值将立即载入到 PIT 计数器中而不需要等待计数器计数到 0x0000。复位后, 该寄存器值为 0xFFFF。

3. PIT计数寄存器 (PCNTRn)

该 16 位只读寄存器保存 PIT 计数器的值。读该寄存器时要一次性读 16 位, 而不能分别读高 8 位和低 8 位。实际上, 计数寄存器简称计数器。

7.1.2 PIT模块的运行模式与编程方法

1. PIT模块的运行模式

可编程中断定时器模块有两种运行方式: 自由运行方式和配置运行方式。PIT 模块的计数器是减 1 计数, 自由运行方式是从 0xFFFF 开始减 1 计数, 到 0 置 PIT 控制和状态寄存器 PCSRn 的 PIF 位为 1。配置运行方式是以模数寄存器 PMRn 的值为基数开始减 1 计数, 到 0 置 PIF 位为 1。通过设置 PCSRn 中的 RLD=0, 使 PIT 模块工作于自由运行方式。通过设置 PCSRn 中的 RLD=1, 使 PIT 模块工作于配置运行方式。

注意两点:

(1) 当 PCSRn 寄存器的重写位 OVW 为 1 时, PMRn 寄存器中的值将立即载入到计数器中, 而不需要等待计数器到达 0x0000。

(2) PIT 计数器的值在 PIT 时钟的上升沿改变, 设 f_{sys} 为系统总线时钟频率, 因此 PIT 的计数周期为:

$$(2 \times \text{PRE}[3:0] \div f_{sys}) \times (\text{PMRn}[15:0] + 1) \text{ 秒}$$

2. PIT模块的编程方法

可编程中断定时器 PIT 模块的编程主要由两部分组成: PIT 的初始化程序和 PIT 的中断程序。

(1) PIT 的初始化程序

在 PIT 的初始化程序中，主要对 PIT 模块的时钟频率和一些操作方式进行设置。PIT 的时钟频率的计算方法见 7.1.1 节中的“PIT 控制和状态寄存器”。PIT 常见的操作方式有：是否产生中断、调试模式和休眠模式下的操作、计数器计数数值的来源等，用户可以根据自己的需要设置这些操作方式。

(2) PIT 的中断程序

在使能 PIT 中断之前，应该禁止全部的中断，防止“虚假”的中断发生，使能 PIT 中断后，还要在中断屏蔽寄存器中使能 PIT 中断。在离开 PIT 中断程序前，应该清除中断标志。PIT 的初始化程序和中断程序见 7.1.3 节的应用实例。

7.1.3 PIT模块的编程实例

本编程实例的功能是：每隔 1 秒切换一次指示灯的状态，即若原来指示灯“亮”，则置指示灯“暗”；反之，若原来指示灯“暗”，则置指示灯“亮”。指示灯与 TA 端口的第 1 引脚（GPT1）连接。按照构件的思想，编程实例涉及 GPIO 构件、Light（指示灯）构件、PIT 构件以及 PIT 中断处理等。

1. PIT构件的头文件（PIT.h）

```
//-----*
// 文件名： PIT.h（可编程中断定时器构件头文件）      *
//-----*
#ifndef PIT_H
#define PIT_H
    //1. 头文件
    #include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
    #include "Type.h"     //类型别名定义
    #include "sysinit.h"  //配置头文件
    #include "GeneralFun.h" //通用函数头文件
    //2. PIT设置结构体
    typedef struct
    {
        uint8 CHANNEL : 1; //PIT通道(0~1)
        uint8 PRE      : 4; //PIT时钟分频因子(0~15)
        uint8 OVW      : 1; //PIT计数立即重写(0~1)
        uint8 RLD      : 1; //PIT重载方式(0~1)
        uint8 DBG      : 1; //PIT调试模式(0~1)
        uint8 DOZE     : 1; //PIT休眠模式(0~1)
        uint8 IL       : 3; //PIT中断级别(1~7)
        uint8 IP       : 3; //PIT中断优先级(0~7)
        uint16 Module  : 16; //PIT模数寄存器
    }PITSet;
    //3. 中断开放与关闭宏定义
    #define EnablePIT(channel) (MCF_PIT_PCSR(channel) |= 0x0001)
    #define DisablePIT(channel) (MCF_PIT_PCSR(channel) &= ~(0x0001))
```

```

//4. 功能接口(函数声明)
//-----*
//函数名: PIT_Init
//功 能: 初始化PIT
//参 数: setinfo: 与PIT设置相关的结构体
//返 回: 无
//说 明: 初始化结束后,PIT模块是禁止的,使用时应当先使能PIT模块
//适用范围:ColdFire芯片
//修改说明:无
//-----*
void PIT_Init(PITSet setinfo);
#endif

```

2. PIT构件的源程序文件 (PIT.c)

```

//[PIT.c]可编程中断定时器-----*
//本文件包含:
// PITInit:PIT初始化
//-----*
//头文件
#include "PIT.h" //PIT头文件
//寄存器位宏定义
#define PIT_PCSR_DOZE 6
#define PIT_PCSR_DBG 5
#define PIT_PCSR_OVW 4
#define PIT_PCSR_PIE 3
#define PIT_PCSR_RLD 1
#define PIT_PCSR_EN 0
//-----*
//函数名: PIT_Init
//功 能: 初始化PIT
//参 数: setinfo: 与PIT设置相关的结构体
//返 回: 无
//说 明: 初始化结束后,PIT模块是禁止的,使用时应当先使能PIT模块
//适用范围:ColdFire芯片
//修改说明:无
//-----*
void PIT_Init(PITSet setinfo)
{
    //1. 禁止PIT
    BCLR(PIT_PCSR_EN, MCF_PIT_PCSR(setinfo.CHANNEL));
    //2. 设置PIT时钟, PIT的OVW、RLD、DOZE和DBG功能位
    MCF_PIT_PCSR(setinfo.CHANNEL) = 0
                                | setinfo.PRE<<8
                                | setinfo.OVW<<PIT_PCSR_OVW
                                | setinfo.RLD<<PIT_PCSR_RLD
                                | setinfo.DBG<<PIT_PCSR_DBG
                                | setinfo.DOZE<<PIT_PCSR_DOZE;

    //3. 写模数寄存器同时清中断标志位

```

```

MCF_PIT_PMR(setinfo.CHANNEL) = 0 | setinfo.Module;
//4. 设置PIT中断级别并清PIT中断屏蔽标志
if (setinfo.CHANNEL == 0)
{
    MCF_INTC0_ICR55 = MCF_INTC_ICR_IL(setinfo.IL)
                    | MCF_INTC_ICR_IP(setinfo.IP);
    MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK55;
}
else if (setinfo.CHANNEL == 1)
{
    MCF_INTC0_ICR56 = MCF_INTC_ICR_IL(setinfo.IL)
                    | MCF_INTC_ICR_IP(setinfo.IP);
    MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK56;
}
MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
}

```

3. 测试工程的主函数（main.c）

主函数 main.c 所要做的工作是，对 PIT 模块和小灯模块进行初始化，允许 PIT 中断和使能 PIT 模块，随后进入死循环，等待中断。

为保证中断时间间隔约为 1 秒，PIT 模块初始化函数 PIT_Init 的入口参数 PITSetInfo 应赋值如下：

```

PITSet PITSetInfo =
{
    0,    //选择 PIT0
    15,   //PIT0 时钟频率为 1831Hz
    1,    //PIT 模数寄存器值立即生效
    1,    //PIT 从模数寄存器取值
    1,    //PIT 在调试模式下禁止
    1,    //PIT 在休眠模式下禁止
    3,    //PIT 中断级别 3
    3,    //PIT 中断优先级 3
    915,  //PIT 模数寄存器值为 915
};

```

4. PIT中断处理函数（PIT0_handler）

```

//PIT0_handler中断处理程序
__interrupt__
void PIT0_handler(void)
{
    DisableInterrupts();    //关总中断
    MCF_PIT0_PCSR |= 1<<2; //清PIT0中断标志
    Light_Change(Light_Run_PORT, Light_Run);    //切换指示灯的亮暗状态
    EnableInterrupts();    //开总中断
}

```

```
}
```

为了切换指示灯的亮暗状态，在 Light 构件中添加了功能函数 Light_Change，实现方法详见光盘中本工程实例。

7.2 通用定时器GPT模块

7.2.1 计数器/定时器的基本工作原理

在嵌入式应用系统中，有时要求能对外部脉冲信号或开关信号进行计数，这可通过计数器来完成。有时要求设备每间隔一定时间开启并在一段时间后关闭，有时要求指示灯不断地闪烁，这可利用定时信号来完成。个人计算机也经常要用到定时信号，如系统日历时钟的计时、产生不同频率的声源等。在计算机系统中，计数与定时问题的内容是一致的，只不过是同一个问题的两种表现形式。本节首先讨论计数与定时的一般方法，比较各种计数与定时方式的优缺点。然后阐述 MCF52233 MCU 定时接口模块的基本功能。

1. 实现计数与计时的基本方法

实现计数与计时的基本方法有三种：

（1）完全硬件方式

在过去许多仪器仪表或设备中，需要进行延时、定时或计数，经常使用数字逻辑电路实现，即完全用硬件电路实现计数/定时功能。若要改变计数/计时的要求，必须改变电路参数，缺点是通用性、灵活性差。微型电子计算机出现以后，特别是单片微型计算机的发展与普及，这种完全硬件方式实现定时与计数已很少使用。

（2）完全软件方式

在计算机中，通过编程，利用计算机执行指令的时间实现定时，称为完全软件方式，简称软件方式。在这种方式中，一般是根据所需要的时间常数来设计一个延时子程序，延时子程序中包含一定的指令，设计者要对这些指令的执行时间进行严密的计算或者精确的测试，以便确定延时时间是否符合要求。当时间常数比较大时，常常将延时子程序设计为一个循环程序，通过循环常数和循环体内的指令来确定延时时间。这样，每当延时子程序结束以后，可以直接转入下面的操作，也可以用输出指令产生一个信号作为定时输出。这种方法的优点是节省硬件。主要缺点是执行延时程序期间，CPU 一直被占用，所以降低了 CPU 的效率，也不容易提供多作业环境；另外，设计延时子程序时，要用指令执行时间来拼凑延时时间，显得比较麻烦。不过，这种方法在实际中还是经常使用的，尤其是在已有系统上做软件开发时，以及延时时间较小而重复次数又较少时，常用软件方法来实现定时。在计算机控制软件开发过程中，作为粗略的延时，经常使用这种方式。

(3) 可编程计数器 / 定时器

利用专门的可编程计数器 / 定时器实现计数与定时,克服了完全硬件方式与完全软件方式的缺点,综合了它们的优点,其计数/计时功能可由程序灵活地设置,设定之后与 CPU 并行地工作,不占用 CPU 的工作时间。应用可编程计数器 / 定时器,在简单的软件控制下,可以产生准确的时间延时。这种方法的主要思想是根据需要的定时时间,用指令对计数器 / 定时器设置定时常数,并用指令启动计数器 / 定时器。计数器 / 定时器开始计数,当计到确定值时,便自动产生一个定时输出。这种方法最突出的优点是计数时不占用 CPU 的时间,如果利用计数器 / 定时器产生中断信号就可以建立多作业的环境,大大提高了 CPU 的利用率。另外,计数器 / 定时器本身的开销不是很大,因此,这种方法在微机应用系统中已被广泛使用。

2. MCF52233的通用定时器模块GPT概述

某些型号的单片机提供多个独立的定时器,如 MC68HC908GP32 芯片有两个定时器,分别叫定时器 1、定时器 2,它们的工作原理是一致的。而有的单片机只提供一个定时器,如 MCF52233 芯片只提供了一个通用定时器模块 GPT (General Purpose Timer)。

定时器的核心是一个处于不断加 1 的 16 位计数寄存器,简称计数器。计数器的时钟频率由总线时钟经过预定的分频因子分频得到,定时器的所有动作都以该经过分频的频率作为参考。从 MCU 的角度看,真正的时间间隔被这个计数器的定时计数所代替。所以在任何时候可以通过读取计数器的值确定经历的时间。

在定时器内部有个控制寄存器,通过对它某些位的设置,就可以确定多少时间计数器加 1,即定时间隔,定时间隔可以达到几 ms 到几 s。

通过对控制和状态寄存器的某位进行设置,可以决定在计数器溢出时,是否允许中断,利用这个中断,可以编写中断例程,实现希望的功能。

另外, MCF52233 的通用定时器模块 GPT 还具有输入捕捉、输出比较、脉冲累加功能。该定时器包含 4 个 16 位的输入捕捉/输出比较通道和 1 个 16 位的脉冲累加器。输入比较功能可用于检测一个选定的跳变沿并记录发生跳变的时间;输出比较功能可用于产生一个期望的输出信号,或用于定时器软件延时;脉冲累加器可当作事件计数器来使用,或作为一个门控时间累加器来使用。

MCF52233 的通用定时器模块 GPT 主要有 29 个寄存器。按照功能分,可以分为定时器计数寄存器、与输入捕捉相关的寄存器、与输出比较相关的寄存器以及与脉冲累加相关的寄存器。

MCF52233 的通用定时器模块 GPT 提供了 7 个中断源,见表 7-1 所示。

表 7-1 定时器中断

中断源	描述
C3F~C0F	定时器通道3~0的输入捕捉/输出比较中断
PAOVF	脉冲累加溢出中断
PAIF	脉冲累加输入中断
TOF	定时器溢出中断

定时器模块只检测中断并产生中断请求，没有对中断请求进行处理。因而用户需要自己编写中断响应程序来处理相应的中断事件。

7.2.2 定时器模块的定时计数功能

1. 定时计数功能相关寄存器

与定时器模块的定时计数功能相关的寄存器主要有：

(1) GPT计数寄存器（GPTCNT）

该寄存器为 16 位的只读寄存器，用于保存当前定时器的计数值。如果要读取该寄存器的值，应当使用一个 16 位的读方式进行读取，否则读出的数值将有可能不准。复位后，该寄存器所有位全为 0。

(2) GPT 系统控制寄存器 1（GPTSCR1）

该寄存器用于配置 GPT 的功能。该寄存器的 D6~D5 和 D3~D0 位为保留位，读为 0，写无效果。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	GPTEN			TFFCA				

D7—GPTEN，GPT 使能位。0：禁止 GPT；1：使能 GPT。在不需要 GPT 的时候建议将 GPT 禁止以降低功耗。

D4—TFFCA，定时器标志快速清零位。若置该位为 1，则不需要写 1 到相应的标志位来清零标志位，只需要读写相关的计数器或者通道寄存器就可以自动清零标志位。

(3) GPT系统控制寄存器2（GPTSCR2）

该寄存器用于配置 GPT 的功能。其中 D6 位为保留位，读为 0，写无效果。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TOI		PUPT	RDPT	TCRE	PR		

D7—TOI，GPT 溢出中断使能位。0：禁止 GPT 溢出中断；1：使能 GPT 溢出中断。

D5~D3：很少使用。

D2~D0—PR，选择 GPT 的计数器的时钟分频。GPT 的计数器的时钟频率计算方法如下：

$$f_{\text{GPT}} = f_{\text{sys}} \div 2 \div 2^{\text{PR}[2:0]}$$

其中 f_{sys} 为系统总线时钟，PR[2:0]取值范围从 000~111，对应的数值为 0~7。

(4) GPT标志寄存器2（GPTFLG2）

该寄存器的 D6~D0 位为保留位，读为 0，写无效果。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TOF							

D7—TOF，定时器溢出标志。当 GPT 的计数器从 0xFFFF 计数到 0x0000 后，该位被

置位, 如果此时 GPT 系统控制寄存器 2 (GPTSCR2) 的 TOI 位为 1, 那么会产生一个中断请求。

2. 定时器溢出中断编程实例

本编程实例的功能是: PC 通过串行口接收 MCU 发送来的时、分、秒信息并显示, 同时可将 PC 机的当前时间通过串行口发送到 MCU, 作为 MCU 的时间校准值。程序中定义了一个全局变量 `uint8 TimeOverCount`; 用于定时器溢出中断计数, 还定义了一个全局数组 `uint8 Time[3]`; 用于存放当前 MCU 的时、分、秒。按照构件的思想, 本实例设计了 GPT 构件。

(1) GPT构件的头文件 (GPT.h)

```
//-----*
// 文件名: GPT.h (通用定时器构件头文件) *
//-----*
#ifndef GPT_H
#define GPT_H
    //1. 头文件
    #include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
    #include "Type.h" //类型别名定义
    #include "sysinit.h" //配置头文件
    //2. GPT设置结构体
    typedef struct
    {
        uint8 PR : 3; //GPT预分频值(0~7)
        uint8 IL : 3; //GPT中断级别(1~7)
        uint8 IP : 3; //GPT中断优先级(0~7)
    }GPTSet;
    //3. 宏定义
    //使能定时器计时
    #define EnableGPT() { (MCF_GPT_GPTSCR1 |= MCF_GPT_GPTSCR1_GPTEN); }
    //禁止定时器计时
    #define DisableGPT() { (MCF_GPT_GPTSCR1 &= ~MCF_GPT_GPTSCR1_GPTEN); }
    //4. 功能接口(函数声明)
    //-----*
    //函数名: GPT_Init *
    //功 能: 初始化GPT *
    //参 数: setinfo: 与GPT设置相关的结构体GPTSet *
    //返 回: 无 *
    //说 明: 初始化结束后, GPT计时禁止, 禁止GPT溢出中断 *
    //-----*
    void GPT_Init(GPTSet setinfo);
#endif
```

(2) GPT构件的源程序文件 (GPT.c)

```

//[GPT.c]通用定时器文件-----*
//本文件包含: *
// GPT_Init:GPT初始化 *
//-----*
//头文件
#include "GPT.h" //GPT头文件
//-----*
//函数名: GPT_Init *
//功 能: 初始化GPT *
//参 数: setinfo: 与GPT设置相关的结构体GPTSet *
//返 回: 无 *
//说 明: 初始化结束后, GPT计时禁止, 禁止GPT溢出中断 *
//-----*
void GPT_Init(GPTSet setinfo)
{
    //1. 禁止GPT
    DisableGPT();
    //2. 设置GPT时钟
    MCF_GPT_GPTSCR2 |= MCF_GPT_GPTSCR2_PR(setinfo.PR);
    //3. 设置GPT中断级别并清GPT溢出中断屏蔽标志
    MCF_INTC0_ICR41 = MCF_INTC_ICR_IL(setinfo.IL)
                    | MCF_INTC_ICR_IP(setinfo.IP);
    MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK41;
    MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
    //4. 清GPT计数器
    MCF_GPT_GPTCNT = 0x0000;
}

```

(3) GPT溢出中断处理函数 (GPTTO_handler)

```

//GPTTO_handler中断处理程序
__interrupt__
void GPTTO_handler(void)
{
    DisableInterrupts(); //关总中断
    TimeOverCount++;
    if (TimeOverCount >= 7)
    {
        TimerUpdate(Time); //秒加1
        TimeOverCount = 0;
    }
    MCF_GPT_GPTFLG2 |= MCF_GPT_GPTFLG2_TOF; //清GPT溢出中断标志
    EnableInterrupts(); //开总中断
}

```

(4) Uart0中断处理程序 (Uart0_handler)

```
//Uart0中断处理程序
__interrupt__
void Uart0_handler(void)
{
    DisableInterrupts(); //关总中断
    UartReN(0, 3, Time); //接收“时分秒”,放入全局变量数组Time中
    EnableInterrupts(); //开总中断
}
```

(5) 秒加1计时函数 (TimerUpdate)

```
//-----*
//函数名: TimerUpdate (秒加1计时函数) *
//功 能: 秒加1,同时更新分和时的数值 *
//参 数: 无 *
//返 回: 无 *
//-----*
void TimerUpdate(uint8 Time[])
{
    Time[2]++; //秒数加1
    //更新分数
    if (Time[2] >= 60)
    {
        Time[2] = 0;
        Time[1]++;
        //更新时数
        if (Time[1] >= 60)
        {
            Time[1] = 0;
            Time[0]++;
            //时分秒清零
            if (Time[0] >= 24)
            {
                Time[0] = 0;
            }
        }
    }
}
```

7.2.3 定时器模块的输入捕捉功能

输入捕捉功能是 MCF52233 MCU 定时器模块的基本功能之一,本节讨论该模块输入捕捉功能的编程方法。

1. 输入捕捉的基本含义

(1) 与输入捕捉功能相关的引脚

定时器模块的 GPT3~GPT0 可用于定时器的输入捕捉引脚,也可用于定时器的输出比较引脚。因为输入捕捉与输出比较复用同一个引脚,一个引脚不能同时被用来作为输入捕捉和输出比较,只能选用其中一个功能,为此,定时器模块中提供了一个输入捕捉和输出比较的选择位,通过设置相应的位可以将对应的通道设置为既定的功能。

(2) 输入捕捉的基本含义

输入捕捉功能是用来监测外部的事件和输入信号。当外部事件发生或信号发生变化时,在指定的输入捕捉引脚上发生一个指定的跳变沿(可以指定该跳变是上升沿还是下降沿)。定时器捕捉到特定的跳变沿后,把计数寄存器当前的值锁存到通道寄存器。如果在输入捕捉控制寄存器中设定允许输入捕捉中断,系统会产生一次输入捕捉中断,利用中断处理软件可以得到事件发生的时刻或信号发生变化的时刻。

通过记录输入信号的连续的跳变沿,就可以用软件算出输入信号的周期和脉宽。例如,为了测量周期,只要捕捉到两个相邻的上升沿或下降沿的时间,两者相减就可以得到周期;为了测量脉宽就要记录相邻的两个不同极性的沿变化的时间。当测量的脉宽值小于定时器的溢出周期时,只要将两次的值直接相减(看成无符号数)。如果测量值大于定时器的溢出周期,那么在两次输入捕捉中断之间就会发生定时器计数的溢出翻转,这时直接将两个数相减就没有意义,需要考虑到定时器的溢出次数。



图7-1 输入捕捉过程

图 7-1 表示输入捕捉引脚的电平变化。假设触发方式是跳变沿触发,在图中的时刻 1 将计数器的值锁存在通道寄存器中,在输入捕捉中断中,把它另存到一个内存单元以防下次将内容覆盖。在图中的 2 时刻会再次进入中断。这次将通道寄存器的值和内存单元的值相减就得到了为低电平的时间。

2. 输入捕捉功能相关寄存器

与定时器模块的输入捕捉功能相关的寄存器主要有:

(1) GPT输入捕捉/输出比较选择寄存器(GPTIOS)

该寄存器用于选择 GPT 的 4 个通道作为输入捕捉使用还是输出比较使用。复位后,该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					IOS			

D7~D4: 保留位,读为 0,写无效果。

D3~D0: 对应 4 个通道 GPT[3:0]的输入捕捉/输出比较选择功能。0: 使能输入捕捉功能。1: 使能输出比较功能;

(2) GPT控制寄存器2 (GPTCTL2)

该寄存器用于配置引发输入捕捉的电平极性。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A

第 7~6 位控制通道 GPT3，第 5~4 位控制 GPT2，第 3~2 位控制通道 GPT1，第 1~0 位控制 GPT0。

若相应通道的 2 位为 00，则禁止输入捕捉功能。

若相应通道的 2 位为 01，则只有在上升沿时，输入捕捉才会发生。

若相应通道的 2 位为 10，则只有在下降沿时，输入捕捉才会发生。

若相应通道的 2 位为 11，则当在跳变沿（上升沿或者下降沿）时，输入捕捉才会发生。

(3) GPT标志寄存器1 (GPTFLG1)

该寄存器的第 7~4 位为保留位，读为 0，写无效果。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					CF			

D3~D0：即 CF[3:0]，对应 GPT 通道 3~0。当通道的输入捕捉或者输出比较发生时，相应通道的位会置位。

(4) GPT中断使能寄存器 (GPTIE)

该寄存器用于使能 GPT 的通道中断。复位后，该寄存器全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					CI			

D3~D0：即 CI[3:0]，对应 GPT 通道 3~0。若相应位置 1，则当 GPT 标志寄存器 1 (GPTFLG1) 中的 CF[3:0]的相应位为 1 时将产生中断请求。

(5) GPT通道寄存器 (GPTCn)

当 GPT 的通道配置成输入捕捉功能时，该寄存器保存当输入捕捉发生时的 GPT 计数器的值。当 GPT 的通道配置成输出比较功能时，该寄存器存放输出比较的值。

3. 输入捕捉中断编程实例

本编程实例的功能是：验证定时器通道 0 (GPT0) 输入捕捉的功能。扩展板开关接口接 GPT0 引脚，TC 端口的第 1 引脚 (TIN0) 接指示灯，当拨动开关时指示灯的状态将会亮-暗切换。

(1) 输入捕捉头文件 (GPTIC.h)

```
//-----*
// 文件名: GPTIC.h (通用定时器输入捕捉构件头文件)
//-----*
#ifndef GPTIC_H
#define GPTIC_H
```

```

//1. 头文件
#include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
#include "Type.h" //类型别名定义
#include "sysinit.h" //配置头文件
//2. GPTIC设置结构体
typedef struct
{
    uint8 channel : 2; //输入捕捉通道号(0~3)
    uint8 EDG : 2; //输入捕捉中断触发极性设置
    //00:输入捕捉禁止
    //01:上升沿捕捉
    //10:下降沿捕捉
    //11:跳变沿捕捉(上升沿或者下降沿)
    uint8 PR : 3; //GPT预分频值(0~7)
    uint8 IL : 3; //输入捕捉中断级别(1~7)
    uint8 IP : 3; //输入捕捉中断优先级(0~7)
}GPTICSet;
//3. 宏定义
//使能定时器计数
#define EnableGPT() { (MCF_GPT_GPTSCR1 |= MCF_GPT_GPTSCR1_GPTEN); }
//禁止定时器计数
#define DisableGPT() { (MCF_GPT_GPTSCR1 &= ~MCF_GPT_GPTSCR1_GPTEN); }
//4. 功能接口(函数声明)
//-----*
//函数名: GPTIC_Init *
//功 能: 初始化输入捕捉功能 *
//参 数: setinfo: 与输入捕捉设置相关的结构体GPTICSet *
//返 回: 无 *
//说 明: 初始化结束后, 定时器计数是禁止的, 输入捕捉中断是禁止的 *
//-----*
void GPTIC_Init(GPTICSet setinfo);
#endif

```

(2) 输入捕捉初始化文件 (GPTIC.c)

```

//[GPTIC.c]定时器输入捕捉中断文件-----*
//本文件包含: *
// GPTIC_Init:定时器输入捕捉中断初始化 *
//-----*
//头文件
#include "GPTIC.h" //GPTIC头文件
//-----*
//函数名: GPTIC_Init *
//功 能: 初始化定时器输入捕捉功能 *
//参 数: setinfo: 与PIT设置相关的结构体GPTICSet *
//返 回: 无 *
//说 明: 初始化结束后, 定时器计数是禁止的, 输入捕捉中断是禁止的 *
//-----*
void GPTIC_Init(GPTICSet setinfo)

```

```

{
    //1. 配置TA引脚为GPT功能
    MCF_GPIO_PTAPAR |= 0x01<<(setinfo.channel*2);
    //2. 禁止GPT
    DisableGPT();
    //3. 设置GPT时钟
    MCF_GPT_GPTSCR2 |= MCF_GPT_GPTSCR2_PR(setinfo.PR);
    //4. 设置GPT通道功能为输入捕捉功能
    MCF_GPT_GPTIOS &= ~(1<<setinfo.channel);
    //5. 设置输入捕捉触发极性
    MCF_GPT_GPTCTL2 |= 0|(setinfo.EDG<<(setinfo.channel*2));
    //6. 设置输入捕捉中断级别并清除输入捕捉中断屏蔽标志
    switch (setinfo.channel)
    {
        case 0:
            MCF_INTCO_ICR(44) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTCO_IMRH &= ~MCF_INTC_IMRH_MASK44;
            MCF_INTCO_IMRL &= ~MCF_INTC_IMRL_MASK44;
            break;
        case 1:
            MCF_INTCO_ICR(45) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTCO_IMRH &= ~MCF_INTC_IMRH_MASK45;
            MCF_INTCO_IMRL &= ~MCF_INTC_IMRL_MASK45;
            break;
        case 2:
            MCF_INTCO_ICR(46) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTCO_IMRH &= ~MCF_INTC_IMRH_MASK46;
            MCF_INTCO_IMRL &= ~MCF_INTC_IMRL_MASK46;
            break;
        case 3:
            MCF_INTCO_ICR(47) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTCO_IMRH &= ~MCF_INTC_IMRH_MASK47;
            MCF_INTCO_IMRL &= ~MCF_INTC_IMRL_MASK47;
            break;
        default:
            break;
    }
    //7. 清GPT计数器
    MCF_GPT_GPTCNT = 0x0000;
}

```

(3) 输入捕捉中断处理函数 (GPTIC0_handler)

```

//GPTIC0_handler中断处理程序
__interrupt__
void GPTIC0_handler(void)
{
    uint16 i;
    DisableInterrupts();    //关总中断
    for (i=0; i<40000; i++); //延时防“抖动”
    Light_Change(Light_Run_PORT, Light_Run);    //切换指示灯状态
    MCF_GPT_GPTFLG1 |= 0x01; //清GPTIC0溢出中断标志
    EnableInterrupts();    //开总中断
}

```

7.2.4 定时器模块的输出比较功能

输出比较功能也是 MCF52233 MCU 定时器模块的基本功能之一，本节简要讨论输出比较功能的基本编程方法。

1. 输出比较的基本含义

(1) 与输出比较功能相关的引脚

MCF52233 的定时器模块中输入捕捉与输出比较的引脚是同一引脚，可以定义成不同的功能。这已在 7.2.3 节与输入捕捉功能相关的引脚中进行了介绍，在此不再赘述。

(2) 输出比较的含义

输出比较的功能是用程序的方法在规定的时刻输出需要的电平，实现对外部电路的控制。当定时器的某一通道用作输出比较功能时，输出比较功能将根据软件设置的极性、占空比和频率输出周期性的脉冲。当定时器计数器的值与通道的通道寄存器的值相等时，定时器就可以在该通道的引脚上输出高电平、低电平或对引脚上的电平进行翻转。通道上的输出比较会置位相应的标志位，这时，如果允许在发生输出比较时产生中断的话，还会产生一个中断。

与使用延时来得到所需输出电平的方法相比，使用输出比较的优势在于可以得到非常精确的输出时间间隔。硬件的比较功能不受其它中断的影响，而且对用户程序没有额外的负担。

输出比较最简单、最常用的功能就是产生一定间隔的脉冲。典型的应用实例就是实现软件的串行通讯。用输入捕捉作为数据输入，而用输出比较作为数据输出。首先根据通讯的波特率向通道寄存器写入延时的值，根据待传的数据位确定有效输出电平的高低。在输出比较中断处理程序中，重新更改通道寄存器的值，并根据下一位数据改写有效输出电平控制位。

2. 输出比较功能相关寄存器

与输出比较功能相关的寄存器主要是 GPT 控制寄存器 1 (GPTCTL1)。该寄存器用

于输出比较的输出电平的选择。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0

D7~D6 位控制通道 GPT3，D5~D4 位控制 GPT2，D3~D2 位控制通道 GPT1，D1~D0 位控制 GPT0。

若相应通道的 2 位为 00，则 GPT 与输出引脚断开连接。

若相应通道的 2 位为 01，则当输出比较发生时，反转输出电平极性，即若原来输出高电平，则输出低电平，反之亦然。

若相应通道的 2 位为 10，则当输出比较发生时，输出低电平。

若相应通道的 2 位为 11，则当输出比较发生时，输出高电平。

3. 输出比较中断编程实例

本编程实例的功能是：GPT0 输出指定低电平时间的方波以验证输出比较的功能。特殊地，当设置的低电平时间为 32768 时，用万用表量 GPT0 引脚，可以测量到 1.65V 左右的电压（系统 3.3V 电压的一半）。

(1) 输出比较头文件（GPTOC.h）

```
//-----*
// 文件名: GPTOC.h (通用定时器输出比较构件头文件) *
//-----*

#ifndef GPTOC_H
#define GPTOC_H
    //1. 头文件
    #include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
    #include "Type.h" //类型别名定义
    #include "sysinit.h" //配置头文件
    //2. GPTOC设置结构体
    typedef struct
    {
        uint8 channel : 2; //输出比较通道号(0~3)
        uint8 PR : 3; //GPT预分频值(0~7)
        uint8 IL : 3; //输出比较中断级别(1~7)
        uint8 IP : 3; //输出比较中断优先级(0~7)
        uint16 Value : 16; //输出比较产生的方波的低电平时间(0~65535)
    }GPTOCSet;
    //3. 宏定义
    //使能定时器计数
    #define EnableGPT() { (MCF_GPT_GPTSCR1 |= MCF_GPT_GPTSCR1_GPTEN); }
    //禁止定时器计数
    #define DisableGPT() { (MCF_GPT_GPTSCR1 &= ~MCF_GPT_GPTSCR1_GPTEN); }
    //4. 功能接口(函数声明)
    //-----*
    //函数名: GPTOC_Init *
    //功 能: 初始化GPT输出比较功能 *
    //参 数: setinfo: 与输出比较设置相关的结构体GPTOCSet *
```

```

//返 回: 无 *
//说 明: 初始化结束后,GPT计数是禁止的, 输出比较中断是禁止的 *
//-----*
void GPTOC_Init(GPTOCSet setinfo);
#endif

```

(2) 输出比较初始化文件 (GPTOC.c)

```

//[GPTOC.c]定时器输出比较文件-----*
//本文件包含: *
// (1)GPTOC_Init:定时器输出比较初始化 *
//-----*
//头文件
#include "GPTOC.h" //GPTOC头文件
//-----*
//函数名: GPTOC_Init *
//功 能: 初始化GPT输出比较功能 *
//参 数: setinfo: 与定时器输出比较设置相关的结构体GPTOCSet *
//返 回: 无 *
//说 明: 初始化结束后,GPT计数是禁止的, 输出比较中断是禁止的 *
//-----*
void GPTOC_Init(GPTOCSet setinfo)
{
    //1. 配置TA引脚为GPT功能
    MCF_GPIO_PTAPAR |= 0x01<<(setinfo.channel*2);
    //2. 禁止GPT
    DisableGPT();
    //3. 设置GPT时钟
    MCF_GPT_GPTSCR2 |= MCF_GPT_GPTSCR2_PR(setinfo.PR);
    //4. 设置GPT通道功能为输出比较功能
    MCF_GPT_GPTIOS |= 1<<setinfo.channel;
    //5. 设置输出比较及溢出时的电平极性
    MCF_GPT_GPTCT11 |= (0x11<<(setinfo.channel*2)); //输出比较发生输出低电平
    MCF_GPT_GPTTOV |= 1<<setinfo.channel; //计数溢出时反转电平
    //6. 设置通道寄存器, 设置输出捕捉中断级别并清除输出捕捉中断屏蔽标志
    switch (setinfo.channel)
    {
        case 0:
            MCF_GPT_GPTC0 = setinfo.Value;
            MCF_INTC0_ICR(44) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK44;
            MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASK44;
            break;
        case 1:
            MCF_GPT_GPTC1 = setinfo.Value;
            MCF_INTC0_ICR(45) = MCF_INTC_ICR_IL(setinfo.IL)
                               | MCF_INTC_ICR_IP(setinfo.IP);
            MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK45;

```

```

        MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
        break;
    case 2:
        MCF_GPT_GPTC2 = setinfo.Value;
        MCF_INTC0_ICR(46) = MCF_INTC_ICR_IL(setinfo.IL)
            | MCF_INTC_ICR_IP(setinfo.IP);
        MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK46;
        MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
        break;
    case 3:
        MCF_GPT_GPTC3 = setinfo.Value;
        MCF_INTC0_ICR(47) = MCF_INTC_ICR_IL(setinfo.IL)
            | MCF_INTC_ICR_IP(setinfo.IP);
        MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK47;
        MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
        break;
    default:
        break;
}
//7. 清GPT计数器
MCF_GPT_GPTCNT = 0x0000;
}

```

(3) 输出比较中断处理函数 (GPTOC0_handler)

```

//GPTOC0_handler中断处理程序
__interrupt__
void GPTOC0_handler(void)
{
    DisableInterrupts();    //关总中断
    MCF_GPT_GPTFLG1 |= 0x01; //清GPTOC0溢出中断标志
    EnableInterrupts();     //开总中断
}

```

7.2.5 定时器模块的脉冲累加功能

1. 脉冲累加的基本含义

(1) 与脉冲累加功能相关的引脚

定时器模块提供了 1 个 16 位的脉冲累加器，该脉冲累加器与定时器的通道 3 共用引脚，外部引脚为 GPT3。

(2) 脉冲累加的含义

脉冲累加器的输入通道的有效沿有两种形式：一种是用于事件计数方式的极性沿，一种是用于门控时间累加方式的脉冲。

在事件计数方式下，输入引脚上每产生一个有效跳变沿就会使得脉冲累加器计数器的

值加 1。

在门控时间累加方式下，输入引脚上的有效电平将会触发脉冲累加器对 64 分频后的时钟进行计数。

2. 脉冲累加功能相关寄存器

与定时器模块的脉冲累加功能相关的寄存器主要有：

(1) 脉冲累加控制寄存器 (GPTPACTL)

该寄存器用于配置脉冲累加的相关功能。第 7 位为保留位，读为 0，写无效果。复位后，该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义		PAE	PAMOD	PEDGE	CLK		PAOVI	PAI

D6—PAE，脉冲累加使能位。0：禁止脉冲累加功能。1：使能脉冲累加功能。特别地，就算 GPT 模块未使能，脉冲累加也可以正常使用。

D5—PAMOD，脉冲累加模式选择位。0：选择事件计数模式。1：选择门控时间累加模式。

D4—PEDGE，脉冲累加有效沿选择位。在事件计数模式下，若该位置 1，则在上升沿时增加计数器，反之则在下降沿时增加计数器。在门时间累加模式下，若该位置 1，则有效电平是低电平，反之有效电平为高电平。

D3~D2—CLK，选择 GPT 计数器输入时钟 CLK[1:0]。

该两位为 00 时，输入时钟为 GPT 的预分频时钟。

该两位为 01 时，输入时钟为 PACLK。

该两位为 10 时，输入时钟为 PACLK/256。

该两位为 11 时，输入时钟为 PACLK/65536。

D1—PAOVI，脉冲累加溢出中断使能位。若该位为 1，则脉冲累加溢出标志位 (PAOVF) 置位将产生一个中断请求。

D0—PAI，脉冲累加输入中断使能位。若该位为 1，则脉冲累加输入标志位 (PAIF) 置位将产生一个中断请求。

(2) 脉冲累加标志寄存器 (GPTPAFLG)

该寄存器的 D7~D2 位为保留位，读为 0，写无效果。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义							PAOVF	PAIF

D1—脉冲累加溢出标志位 (PAOVF)。当脉冲累加器计数从 0xFFFF 到 0x0000 后，该位被置位，此时，如果脉冲累加控制寄存器 GPTPACTL 寄存器中的 PAOVI 位为 1，那么会产生一个中断请求。向该位写 1 将清零该位。

D0—脉冲累加输入标志位 (PAIF)。当检测到脉冲累加所设置的有效电平时，该位被置位，此时，如果脉冲累加控制寄存器 GPTPACTL 寄存器中的 PAI 位为 1，那么会产生一个中断请求。向该位写 1 将清零该位。

(3) 脉冲累加计数寄存器 (GPTPACNT)

该寄存器保存自上次复位后检测到的脉冲累加所设置的有效电平的次数。读该寄存器时，应当使用一个 16 位的读方式进行读取，否则读出的数值将有可能不准。

3. 脉冲累加器中断编程实例

本编程实例的功能是：允许脉冲累加器输入引脚 GPT3 上的下降沿引发中断。在中断处理函数中，除了将相应的标志位清零之外，还将脉冲累加器中的当前值通过串口发送给 PC 机。

(1) 脉冲累加头文件 (GPTPA.h)

```
//-----*
// 文件名: GPTPA.h (脉冲累加构件头文件) *
//-----*
#ifndef GPTPA_H
#define GPTPA_H
    //1. 头文件
    #include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
    #include "Type.h" //类型别名定义
    #include "sysinit.h" //配置头文件

    //2. 宏定义
    //使能定时器计数
    #define EnableGPT() { (MCF_GPT_GPTSCR1 |= MCF_GPT_GPTSCR1_GPTEN); }
    //禁止定时器计数
    #define DisableGPT() { (MCF_GPT_GPTSCR1 &= ~MCF_GPT_GPTSCR1_GPTEN); }
    //3. 功能接口(函数声明)
    //-----*
    //函数名: GPTPA_Init *
    //功 能: 初始化脉冲累加器, 下降沿累加 *
    //参 数: 无 *
    //返 回: 无 *
    //说 明: 初始化结束后, 定时器计数是禁止的, 脉冲累加器中断是禁止的 *
    //-----*
    void GPTPA_Init(void);
#endif
```

(2) 脉冲累加初始化文件 (GPTPA.c)

```
//[GPTPA.c]脉冲累加器-----*
//本文件包含: *
// GPTPA_Init:脉冲累加器初始化 *
//-----*
//头文件
#include "GPTPA.h" //PIT头文件
//-----*
//函数名: GPTPA_Init *
//功 能: 初始化脉冲累加器, 下降沿累加 *
```

```

//参 数: 无 *
//返 回: 无 *
//说 明: 初始化结束后, 定时器计数是禁止的, 脉冲累加器中断是禁止的 *
//-----*
void GPTPA_Init(void)
{
    //1. 配置TA. 3引脚为GPT功能
    MCF_GPIO_PTAPAR |= 0x40;
    //2. 禁止GPT
    DisableGPT();
    //3. 设置GPT时钟
    MCF_GPT_GPTSCR2 |= MCF_GPT_GPTSCR2_PR(6);
    //4. 设置GPT通道3功能为脉冲累加功能, 下降沿累加, 关闭脉冲累加中断
    MCF_GPT_GTPACTL = 0x40;
    //5. 设置脉冲累加中断级别并清屏蔽位
    MCF_INTC0_ICR(42) = MCF_INTC_ICR_IL(3)
                      | MCF_INTC_ICR_IP(3);
    MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK42;
    MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;
    //6. 清脉冲累加输入标志
    MCF_GPT_GTPAFLG |= MCF_GPT_GTPAFLG_PAIF;
    //7. 清脉冲累加计数器
    MCF_GPT_GTPACNT = 0x0000;
    //8. 清GPT计数器
    MCF_GPT_GPTCNT = 0x0000;
}

```

(3) 脉冲累加中断处理函数 (GPTPA_handler)

```

//GPTPA_handler中断处理程序
__interrupt__
void GPTPA_handler(void)
{
    uint16 i;
    DisableInterrupts(); //关总中断
    for (i=0; i<60000; i++); //延时防“抖动”
    //发送脉冲累加次数
    UartSend1(0, (uint8)(MCF_GPT_GTPACNT>>8));
    UartSend1(0, (uint8)MCF_GPT_GTPACNT);
    MCF_GPT_GTPAFLG |= MCF_GPT_GTPAFLG_PAIF; //清脉冲累加中断标志
    EnableInterrupts(); //开总中断
}

```

7.3 脉宽调制模块PWM与D/A转换

7.3.1 脉宽调制模块PWM基本概念

PWM (Pulse Width Modulation) 又称脉冲宽度调制模块, 是嵌入式应用系统的常用功能之一。它通过对一系列脉冲的宽度进行调制, 来等效地获得所需要的波形 (包括形状和幅值)。本节将给出 PWM 的基本概念, 深刻理解这些概念将有助于 PWM 的使用。

1. PWM的周期与占空比

PWM 会产生一个在高电平和低电平之间重复交替的输出信号, 这个信号被称为 PWM 信号, 通常定义 PWM 周期为该 PWM 信号的周期, PWM 占空比为信号处于高电平的时间 (或时钟周期数) 占整个信号周期的百分比。图 7-2 给出了在同一 PWM 周期下几个不同占空比的 PWM 信号示意图。

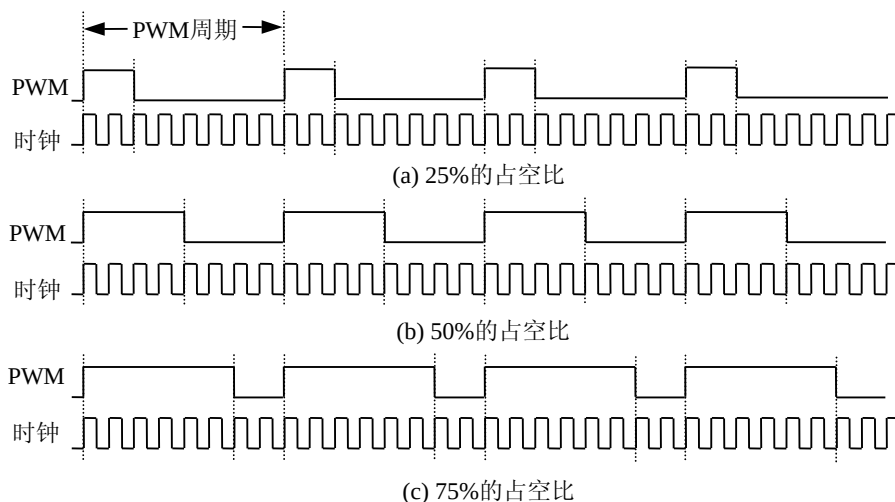


图7-2 PWM的周期与占空比

2. PWM的脉冲极性

PWM 的脉冲极性指的是 PWM 信号初始时输出的电平的极性, 若设置 PWM 信号初始时输出高电平, 那么当 PWM 的计数器与设置的占空比值相等时, PWM 信号变换输出电平的极性, 输出低电平, 此时的 PWM 信号如图 7-3 (a) 所示。

若设置 PWM 信号初始时输出低电平, 那么当 PWM 的计数器与设置的占空比值相等时, PWM 信号变换输出电平的极性, 输出高电平, 此时的 PWM 信号如图 7-3 (b) 所示。在实际使用时, 可以根据系统具体的需求来设计 PWM 的脉冲极性。

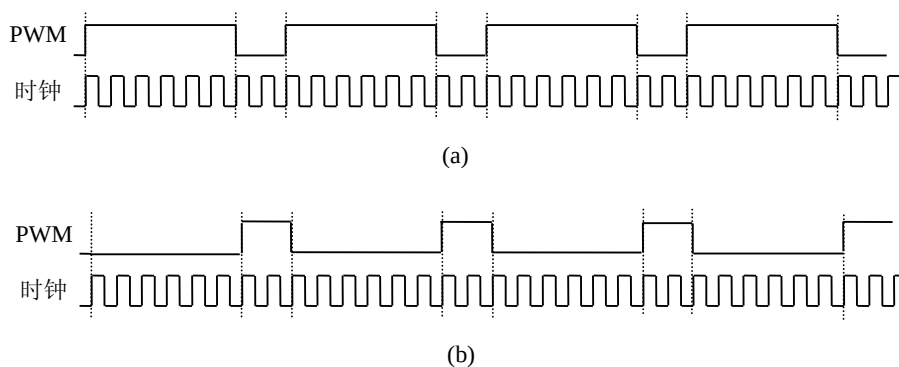


图7-3 PWM脉冲极性

3. PWM的脉冲对齐方式

PWM 的脉冲对齐方式有两种：左对齐与中心对齐。若设置 PWM 信号初始时输出高电平，左对齐的 PWM 信号如图 7-4 (a) 所示，当 PWM 的计数器与设置的占空比所对应的值相等时，PWM 信号变换输出电平的极性，输出低电平，当 PWM 的计数器与设置的 PWM 信号周期值相等时，PWM 的计数器重新清零继续递增计数，此时输出电平变换，输出高电平，以此类推。中心对齐的 PWM 信号如图 7-4 (b) 所示，当 PWM 的计数器与设置的占空比值相等时，PWM 信号变换输出电平的极性，输出低电平，当 PWM 的计数器与设置的 PWM 信号周期值相等时，PWM 的计数器递减继续计数，PWM 信号继续输出低电平，当 PWM 的计数器与设置的占空比值相等时，PWM 信号变换输出电平的极性，输出高电平，当 PWM 的计数器与设置的 PWM 信号周期值相等时，PWM 的计数器重新清零继续递增计数，此时输出电平变换，输出低电平，以此类推。

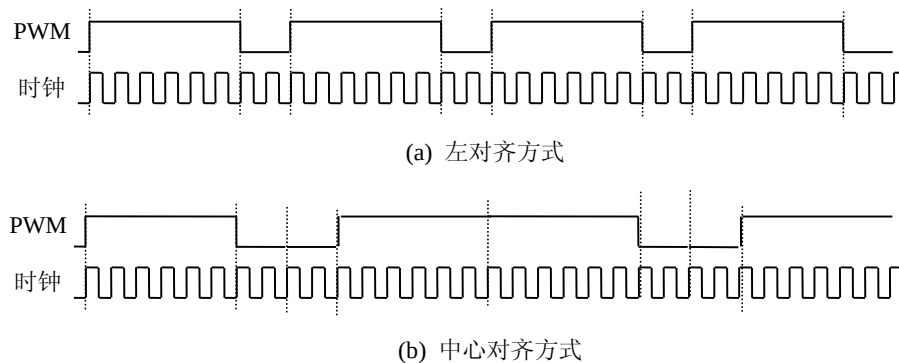


图7-4 PWM脉冲对齐方式

7.3.2 脉宽调制模块PWM的编程寄存器

在进行 MCF52233 的脉宽调制模块 PWM 编程之前，应当先理解该模块的寄存器。MCF52233 的 PWM 有 12 个寄存器，下面将依次介绍这些寄存器。

1. PWM使能寄存器 (PWME)

该寄存器中的每一位均控制每一个 PWM 通道的 PWM 信号是否输出。在运行模式下，如果所有的 PWM 通道都是禁止的，那么为了低功耗，预分频寄存器将会关闭。复位后该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0

PWME_n=1, 使能 PWM 通道 n 输出; PWME_n=0, 禁止 PWM 通道 n 输出。其中 n=0~7。

2. PWM极性寄存器 (PWMPOL)

该寄存器设置 PWM 的脉冲极性，当 PWM 信号已经输出时，更改 PWM 的脉冲极性将会导致在脉冲极性切换的过程中 PWM 的输出信号不稳。复位后该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0

PPOL_n=1, PWM 通道 n 的信号在周期的开始时输出高电平，到达占空比计数后输出低电平; PPOL_n=0, PWM 通道 n 的信号在周期的开始时输出低电平，到达占空比计数后输出高电平。其中 n=0~7。

3. PWM时钟选择寄存器 (PWMCLK)

每一个 PWM 通道都可以选择各自的时钟源。PWM 通道 0, 1, 4 和 5 可以选择时钟 A 或者 SA, PWM 通道 2, 3, 6 和 7 可以选择时钟 B 或者 SB。当 PWM 信号已经输出时，更改 PWM 的时钟将会导致在时钟切换的过程中 PWM 的输出信号不稳。复位后该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PCLK7	PCLK6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0

PCLK_n=1, PWM 通道 n 选择 SB 为时钟源; PCLK_n=0, PWM 通道 n 选择 B 为时钟源。其中 n=2, 3, 6, 7。

PCLK_n=1, PWM 通道 n 选择 SA 为时钟源; PCLK_n=0, PWM 通道 n 选择 A 为时钟源。其中 n=0, 1, 4, 5。

4. PWM预分频时钟选择寄存器 (PWMPRCLK)

该寄存器用于设置时钟 A 和 B 的时钟预分频值，当 PWM 信号已经输出时，更改预分频值将会导致在时钟切换的过程中 PWM 的输出信号不稳。复位后该寄存器默认全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义								

时钟 A 与时钟 B 的分频设置如下式：

$$f_A = f_{Bus} \div 2^{PCKA} \quad f_B = f_{Bus} \div 2^{PCKB}$$

其中，该寄存器的第 D6~D4 位为 PCKB, PCKB=000~111 所代表的数值为 0~7，该寄存器的 D2~D0 位为 PCKA, PCKA=000~111 所代表的数值为 0~7。

5. PWM中心对齐使能寄存器 (PWMCAE)

该寄存器设置 PWM 的脉冲对齐方式为左对齐或者中心对齐。只有当相应的 PWM 通道禁止时，才能进行设置。复位后该寄存器全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	CAE7	CAE6	CAE5	CAE4	CAE3	CAE2	CAE1	CAE0

CAEn=1, PWM 脉冲对齐方式为中心对齐; CAEn=0, PWM 脉冲对齐方式为左对齐。
其中 n=0~7。

6. PWM控制寄存器 (PWMCTL)

该寄存器可以对 PWM 模块进行不同的配置，包括将 2 路 8 位的 PWM 通道配置为 1 路 16 位的 PWM 通道，以及设置 PWM 在低功耗模式下的特性。复位后该寄存器全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1-D0
定义	CON67	CON45	CON23	CON01	PSWAI	PFRZ	

CONxy—PWM 通道的合并设置。CONxy=0, PWM 通道 x 与 PWM 通道 y 是独立的 8 位 PWM 通道; CONxy=1, PWM 通道 x 与 PWM 通道 y 合并为 1 路 16 位的 PWM 通道, PWM 通道 x 为高字节, PWM 通道 y 为低字节, 该 16 位的 PWM 信号从 PWM 通道 y 输出, 其极性、对齐方式、时钟选择和使能控制由 PWM 通道 y 决定, PWM 通道 x 在此模式下的任何设置均无效。这里的 x=6, 4, 2, 0, 对应的 y=7, 5, 3, 1。

D3—PSWAI, 休眠模式下 PWM 的时钟设置位。0: 休眠模式下时钟继续运行; 1: 休眠模式下停止时钟;

D2—PFRZ, 调试模式下 PWM 的计数器设置位。0: 调试模式下 PWM 计数器继续运行; 1: 调试模式下停止 PWM 计数器;

7. PWM分频寄存器A (PWMSCLA)

该寄存器设置时钟 SA, 时钟 SA 由时钟 A 分频而来。计算公式为:

$$f_{SA} = f_A \div (2 \times PWMSCLA)$$

PWMSCLA=00000001 ~ 11111111 对应的数值为 1 ~ 255, 特别的, 当 PWMSCLA=00000000 时, 对应的数值为 256。

8. PWM分频寄存器B (PWMSCLB)

该寄存器设置时钟 SB, 时钟 SB 由时钟 B 分频而来。计算公式为:

$$f_{SB} = f_B \div (2 \times PWMSCLB)$$

PWMSCLB=00000001 ~ 11111111 对应的数值为 1 ~ 255, 特别的, 当 PWMSCLB=00000000 时, 对应的数值为 256。

9. PWM通道计数寄存器 (PWMCNTn)

每一个 PWM 通道都有一个各自的 8 位自增/自减计数寄存器, 在不影响 PWM 通道正常工作的情况下, 用户可以读取计数器的值。向该寄存器写入任何值将清零该寄存器, 然后 PWM 通道将会按照设置好的时钟、周期、占空比和对齐方式重新产生 PWM 信号。当

PWM 通道禁止时，相应通道的计数寄存器将停止计数，当 PWM 通道重新使能时，PWM 计数器将从计数寄存器的值开始继续计数。

10. PWM通道周期寄存器（PWMPERn）

每一个 PWM 通道都有一个各自的 8 位周期寄存器。根据周期寄存器的值和 PWM 的计数时钟可以计算出 PWM 信号的周期，具体的计算方法如下。

当 PWM 脉冲对齐方式为左对齐时：

$$\text{PWM 信号周期} = \text{周期寄存器值} \div \text{时钟频率}$$

当 PWM 脉冲对齐方式为中心对齐时：

$$\text{PWM 信号周期} = \text{周期寄存器值} \div \text{时钟频率} \times 2$$

11. PWM通道占空比寄存器（PWMDTYn）

每一个 PWM 通道都有一个各自的 8 位占空比寄存器。根据周期寄存器的值、占空比寄存器的值和 PWM 脉冲极性可以计算出 PWM 信号的占空比，具体的计算方法如下。

当 PWM 脉冲初始时输出高电平时：

$$\text{PWM 信号占空比} = \text{占空比寄存器值} \div \text{周期寄存器值} \times 100\%$$

当 PWM 脉冲初始时输出低电平时：

$$\text{PWM 信号占空比} = (1 - \text{占空比寄存器值} \div \text{周期寄存器值}) \times 100\%$$

12. PWM关闭寄存器（PWMSDN）

该寄存器提供了在紧急情况下关闭 PWM 模块的功能。注意，如果该寄存器的第 0 位为 0，那么对该寄存器的其它位的设置是无效的。复位后该寄存器全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IF	IE	RESTART	LVL		PWM7IN	PWM7IL	SDNEN

D7—IF，PWM 中断标志位。当 PWM7IN 状态变化时会置该位。0：PWM7IN 状态没有变化；1：PWM7IN 状态有变化。写 1 可以清零该位，写 0 无效。

D6—IE，PWM 中断使能位。当使能 PWM 中断时，如果 PWM7IN 出现有效电平时将触发 CPU 中断。0：禁止 PWM 中断；1：使能 PWM 中断；

D5—RESTART，PWM 重启设置位。该位是只写位，读为 0。置该位后，当通道的相应计数器复位为 0 后，PWM 通道开始运行。

D4—LVL，PWM 关闭后输出电平设置位。该位设置当 PWM 关闭后输出的电平的极性。0：PWM 强制输出逻辑 0 电平；1：PWM 强制输出逻辑 1 电平。

D2—PWM7IN，PWM 通道 7 输入状态位。该位反映 PWMOUT7 引脚当前的电平状态，是只读位。

D1—PWM7IL，PWM 通道 7 输入有效极性设置位。0：PWM 通道 7 输入低电平有效；1：PWM 通道 7 输入高电平有效。

D0—SDNEN，PWM 紧急情况关闭使能位。0：禁止 PWM 紧急情况关闭；1：使能 PWM 紧急情况关闭；

7.3.3 脉宽调制模块PWM的编程实例

本编程实例的功能是：将相应的 PWM 通道引脚连接到小灯，实现小灯逐渐变亮的功能。按照构件的思想，定义了 PWM 构件。在 PWM 构件的头文件中，给出了 PWM 设置结构体类型定义：

```
typedef struct
{
    uint8 CHANNEL : 3; //PWM通道(0~7)
    uint8 POL      : 1; //PWM极性选择(0:负极性; 1:正极性)
    uint8 CAE      : 1; //PWM对齐方式选择(0:左对齐; 1:中心对齐)
    uint8 CLK      : 1; //PWM时钟选择
                        //对通道7、6、3、2, 0:选择时钟B; 1:选择时钟SB
                        //对通道5、4、1、0, 0:选择时钟A; 1:选择时钟SA
    uint8 SCLAorB : 8; //PWM时钟SA或者SB的分频因子选择, 若PWM时钟选择
                        //时钟A或者B时, 该8位无意义, 建议设置为0x00
                        //设置范围0x00~0xFF, 其中0x00对应的数值为256
    uint8 PRCLK    : 3; //PWM时钟A或者时钟B的分频因子选择(0~7)
    uint8 CON      : 1; //PWM通道合并使能(0:不合并; 1:合并为16位PWM)
}PWMSet;
```

PWM 构件提供的功能函数有：初始化函数 PWM_Init 以及 PWM 信号产生函数 PWMSetting。

1. PWM初始化函数：PWMInit

PWM 波的输出电压幅值和输出电流大小由引脚的输出电压和电流决定。在输出期望的 PWM 信号之前，要先对 PWM 进行初始化，一般包括 PWM 时钟的选择，PWM 脉冲极性的设置以及 PWM 脉冲对齐方式的设置等。在实际使用时应该根据要驱动的器件的特性选择合适的时钟。一般在初始化结束后，应当清零 PWM 的计数寄存器。PWM 初始化函数 PWMInit 定义如下：

```
//寄存器位宏定义
#define PWM_PWMCTL_CON67 7
#define PWM_PWMCTL_CON45 6
#define PWM_PWMCTL_CON23 5
#define PWM_PWMCTL_CON01 4
//-----*
//函数名: PWM_Init *
//功 能: 初始化PWM *
//参 数: setinfo: 与PWM设置相关的结构体 *
//返 回: 无 *
//说 明: 若设置的PWM通道<0或者>7, 那么强置PWM通道为0。PWM初始化完成后, *
//        PWM通道是禁止的。 *
//-----*
void PWM_Init(PWMSet setinfo)
{
    //1. 引脚功能配置为PWM功能
    if (setinfo.CHANNEL<0 || setinfo.CHANNEL>7)
```

```

    {
        setinfo.CHANNEL = 0;
    }

                                                                    switch
(setinfo.CHANNEL)
    {
        //通道0, 2, 4, 6使用TC端口
        case 0:
        case 2:
        case 4:
        case 6:

                                                                    MCF_GPIO_PTCPAR
|= 0x03<<setinfo.CHANNEL;
        break;
        //通道1, 3, 5, 7使用TA端口
        case 1:
        case 3:
        case 5:
        case 7:
            MCF_GPIO_PTAPAR |= 0x03<<(setinfo.CHANNEL-1);
            break;
        default:
            break;
    }
//2. PWM通道重定位
//若初始化2个8位的通道合并为1个16位的通道, 则置通道号为有效通道号
//即0, 1通道均重定位为1通道, 即2, 3通道均重定位为3通道,
//即4, 5通道均重定位为5通道, 即6, 7通道均重定位为7通道
if (setinfo.CON)
{
    if (setinfo.CHANNEL % 2 == 0)
    {
        setinfo.CHANNEL += 1;
    }
}
//3. 禁止PWM通道
BCLR(setinfo.CHANNEL, MCF_PWM_PWME);
//4. PWM时钟源选择
//PWM通道使用时钟SA、A或者SB、B
//PWM通道使用时钟SA或者SB
if (setinfo.CLK)
{
    BSET(setinfo.CHANNEL, MCF_PWM_PWMCLK);
}
//PWM通道使用时钟A或者B
else
{
    BCLR(setinfo.CHANNEL, MCF_PWM_PWMCLK);
}

```

```

//5. PWM时钟频率设置
//PWM时钟A或者时钟B设置
switch (setinfo.CHANNEL)
{
    //通道0, 1, 4, 5使用时钟A
    case 0:
    case 1:
    case 4:
    case 5:
        MCF_PWM_PWMPRCLK
|= 0x07 & setinfo.PRCLK;
        break;
    //通道2, 3, 6, 7使用时钟B
    case 2:
    case 3:
    case 6:
    case 7:
        MCF_PWM_PWMPRCLK |= 0x70 & (setinfo.PRCLK<<4);
        break;
    default:
        break;
}
//PWM时钟SA或者SB设置
if (setinfo.CLK)
{
    switch (setinfo.CHANNEL)
    {
        //通道0, 1, 4, 5使用时钟SA
        case 0:
        case 1:
        case 4:
        case 5:
            MCF_PWM_PWMSCLA
|= setinfo.SCLAorB;
            break;
        //通道2, 3, 6, 7使用时钟SB
        case 2:
        case 3:
        case 6:
        case 7:
            MCF_PWM_PWMSCLB |= setinfo.SCLAorB;
            break;
        default:
            break;
    }
}
//6. PWM输出极性设置
//PWM输出正极性
if (setinfo.POL)

```

```

    {
        BSET(setinfo.CHANNEL, MCF_PWM_PWMPOL);
    }
    //PWM输出负极性
    else
    {
        BCLR(setinfo.CHANNEL, MCF_PWM_PWMPOL);
    }
    //7. PWM对齐方式设置
    //PWM中心对齐方式
    if (setinfo.CAE)
    {
        BSET(setinfo.CHANNEL, MCF_PWM_PWMCAE);
    }
    //PWM中心对齐方式
    else
    {
        BCLR(setinfo.CHANNEL, MCF_PWM_PWMCAE);
    }
    //8. PWM通道合并成16位的设置
    //将两个8位的PWM通道合并为1路的16位PWM通道
    if (setinfo.CON)
    {
        switch (setinfo.CHANNEL)
        {
            case 0:
            case 1:

BSET(PWM_PWMCTL_CON01, MCF_PWM_PWMCTL);

                break;
            case 2:
            case 3:

BSET(PWM_PWMCTL_CON23, MCF_PWM_PWMCTL);

                break;
            case 4:
            case 5:

BSET(PWM_PWMCTL_CON45, MCF_PWM_PWMCTL);

                break;
            case 6:

BSET(PWM_PWMCTL_CON67, MCF_PWM_PWMCTL);

                break;
            case 7:

                break;
            default:
                break;
        }
    }
}

```

```

        //不合并通道
        else
        {
            switch (setinfo.CHANNEL)
            {
                case 0:
                case 1:

                    BCLR(PWM_PWMCTL_CON01, MCF_PWM_PWMCTL);

                    break;
                case 2:
                case 3:

                    BCLR(PWM_PWMCTL_CON23, MCF_PWM_PWMCTL);

                    break;
                case 4:
                case 5:

                    BCLR(PWM_PWMCTL_CON45, MCF_PWM_PWMCTL);

                    break;
                case 6:

                    BCLR(PWM_PWMCTL_CON67, MCF_PWM_PWMCTL);

                    break;
                case 7:

                    break;
                default:
                    break;
            }
        }
    }
}

```

2. PWM信号产生函数: PWMSetting

PWM 初始化完成以后, 通过对周期寄存器和占空比寄存器的设置可以产生期望的 PWM 信号。一般按照以下的顺序进行设置:

- (1) 禁止相应的 PWM 通道, 防止在设置时出现不规则的波形。
- (2) 按照实际的要求, 设置 PWM 的周期与占空比。
- (3) 清零 PWM 的计数寄存器。
- (4) 使能相应的 PWM 通道, 这样就能产生与设置的 PWM 周期与占空比对应的 PWM 信号。

PWM 信号产生函数 PWMSetting 定义如下:

```

//-----*
//函数名: PWMSetting                                *
//功 能: 设置指定通道PWM波的周期与占空比          *
//参 数: channel:通道号(0~7)                        *
//          con: 当前的设置是对8位单独的通道还是16位的合并通道 *
//          (0:设置的是单独通道; 1:设置的是合并通道)        *

```

```

//      period:PWM周期                                *
//      duty:PWM占空比                                *
//返 回: 无                                            *
//说 明: 对8位单独的通道PWM周期与占空比为8位的, 对16位合并的通道    *
//      PWM周期与占空比为16位的。如果通道号在0~7之外, 则置通道号为0    *
//-----*
void PWMSetting(uint8 channel, uint8 con, uint16 period, uint16 duty)
{
    //1. PWM通道重定位
    if (channel<0 || channel>7)
    {
        channel = 0;
    }
    //若要设置的是合并的16位的通道, 则置通道号为有效通道号
    //即0, 1通道均重定位为1通道, 即2, 3通道均重定位为3通道,
    //即4, 5通道均重定位为5通道, 即6, 7通道均重定位为7通道
    if (con)
    {
        if (channel % 2 == 0)
        {
            channel += 1;
        }
    }
    //2. 禁止PWM通道
    BCLR(channel, MCF_PWM_PWME);
    //3. 通道是否合并
    //合并16位通道的周期和占空比设置
    if (con)
    {
        //设置通道的周期与占空比并清通道计数寄存器
        switch (channel)
        {
            case 1:
                MCF_PWM_PWMPER0 = (uint8)(period>>8); //设置周期寄存器
                MCF_PWM_PWMDTY0 = (uint8)(duty>>8); //设置占空比寄存器
                MCF_PWM_PWMCNT0 = 0x00; //清通道计数寄存器

                MCF_PWM_PWMPER1 = (uint8)period; //设置周期寄存器
                MCF_PWM_PWMDTY1 = (uint8)duty; //设置占空比寄存器
                MCF_PWM_PWMCNT1 = 0x00; //清通道计数寄存器

                break;
            case 3:
                MCF_PWM_PWMPER2 = (uint8)(period>>8); //设置周期寄存器
                MCF_PWM_PWMDTY2 = (uint8)(duty>>8); //设置占空比寄存器
                MCF_PWM_PWMCNT2 = 0x00; //清通道计数寄存器

                MCF_PWM_PWMPER3 = (uint8)period; //设置周期寄存器

```

```

        MCF_PWM_PWMDTY3 = (uint8)duty;//设置占空比寄存器
        MCF_PWM_PWCNT3 = 0x00;//清通道计数寄存器
                                break;
        case 5:

MCF_PWM_PWMPER4 = (uint8)(period>>8);//设置周期寄存器
        MCF_PWM_PWMDTY4 = (uint8)(duty>>8);//设置占空比寄存器
        MCF_PWM_PWCNT4 = 0x00;//清通道计数寄存器

MCF_PWM_PWMPER5 = (uint8)period;//设置周期寄存器
        MCF_PWM_PWMDTY5 = (uint8)duty;//设置占空比寄存器
        MCF_PWM_PWCNT5 = 0x00;//清通道计数寄存器
                                break;
        case 7:

MCF_PWM_PWMPER6 = (uint8)(period>>8);//设置周期寄存器
        MCF_PWM_PWMDTY6 = (uint8)(duty>>8);//设置占空比寄存器
        MCF_PWM_PWCNT6 = 0x00;//清通道计数寄存器

MCF_PWM_PWMPER7 = (uint8)period;//设置周期寄存器
        MCF_PWM_PWMDTY7 = (uint8)duty;//设置占空比寄存器
        MCF_PWM_PWCNT7 = 0x00;//清通道计数寄存器
                                break;
        default:
            break;
    }
}
//单独8位通道的周期和占空比设置
else
{
    MCF_PWM_PWMPER(channel) = (uint8)period;//设置周期寄存器
    MCF_PWM_PWMDTY(channel) = (uint8)duty;//设置占空比寄存器
    MCF_PWM_PWCNT(channel) = 0x00;//清通道计数寄存器
}
//4. 使能PWM通道
BSET(channel, MCF_PWM_PWME);
}

```

7.3.4 脉宽调制模块PWM的进一步讨论

1. 周期和占空比设置的边界讨论

表 7-2 总结了周期和占空比设置的边界情况。其中 $n=0\sim7$ 。

表 7-2 周期和占空比设置的边界情况

PWMDTY n	PWMPER n	PPOL n	PWM通道 n 的输出
0x00	>0x00	1	一直输出低电平
0x00	>0x00	0	一直输出高电平

—	0x00	1	一直输出高电平
—	0x00	0	一直输出低电平
$\geq \text{PWMPERn}$	—	1	一直输出高电平
$\geq \text{PWMPERn}$	—	0	一直输出低电平

2. PWM通道的合并

在使用 MC52233 的 PWM 时,通过对 PWMCTL 寄存器的设置可以将 2 路 8 位的 PWM 通道合并成 1 路 16 位的 PWM 通道使用。但是这 2 路 PWM 通道在选择时有所限制,只能将通道 0 和 1、通道 2 和 3、通道 4 和 5、通道 6 和 7 进行合并。在合并之前,应当先禁止这 2 路 PWM 通道。合并后, PWM 信号从通道 1, 3, 5, 7 输出。合并后的 PWM 信号的时钟、脉冲极性和脉冲对齐方式由通道 1, 3, 5, 7 的相应寄存器设置决定,通道 0, 2, 4, 6 的寄存器设置无效。合并后的通道计数器寄存器、周期寄存器和占空比寄存器均为 16 位,高字节为通道 0, 2, 4, 6 的相应寄存器,低字节为通道 1, 3, 5, 7 的相应寄存器。

7.3.5 用 PWM 实现 D/A 转换

1. 用 PWM 实现 D/A 转换的原理

由前面的介绍可知,通过改变 PWM 的占空比,可以输出不同电压值的波形,利用 PWM 的这一特性,可以模拟简单的 D/A 转换。D/A 转换的精度由 PWM 的位数决定。

这里通过一个简单的例子来说明如何利用 PWM 实现 D/A 转换。假设要输出的电平为 1V,电平的幅值为 5V。首先计算出 PWM 波的占空比为 $1\text{V} \div 5\text{V} \times 100\% = 20\%$ 。然后设置 PWM 波的周期和占空比,这里只要根据实际的要求设置合适的周期,一般来说,周期越短,响应速度越快,再在这周期的基础上设置占空比为 20%即可输出 1V 的电压,实现简单的 D/A 转换。也可以在输出引脚上加上适当的滤波电路,使输出的电平平稳。

2. 用 PWM 实现 16 位 D/A 转换的编程实例

本实例的主要功能是:通过设置不同的占空比,可以在 PWM 输出引脚上用万用表量出相应的电压,以模拟 DA 转换的功能。具体的程序与上节所介绍的 PWM 程序一致,只是在初始化设置时要将两个 8 位的 PWM 通道合并为 1 个 16 位的 PWM 通道即可。具体的程序参见光盘 “...\Chap07\07_PWM_DA(16)”。

第8章 队列式串行外设接口QSPI

本章主要介绍队列式串行外设接口 QSPI 模块。该模块具有与外部设备进行全双工同步通信的功能。Freescall 公司低端的 MCU 基本都集成了 SPI 模块。而在 SPI 协议的基础上，Freescall 公司又对 SPI 的功能进行了加强，增加了队列传输机制，推出了队列串行外围接口协议，即 QSPI 协议。使用该接口，用户可以一次性传输包含多达 16 个 8 到 16 位数据队列。一旦启动传输，直到传输结束都无需 CPU 干预，极大地提高了传输效率。该协议在 ColdFire 系列 MCU 中得到了广泛应用。本章给出 MCF52233 的 QSPI 模块的具体工作原理、寄存器介绍以及相应的编程方法，最后给出了构件化编程实例。

8.1 SPI的基本工作原理

8.1.1 SPI基本概念

串行外设接口（SPI，Serial Peripheral Interface）是 Freescall 公司推出的一种同步串行通讯接口，用于微处理器和外围扩展芯片之间的串行连接，现已发展成为一种工业标准。目前，各半导体公司推出了大量带有 SPI 接口的芯片，如 RAM、EEPROM、A/D 转换器、D/A 转换器、LED/LCD 显示驱动器、I/O 接口芯片、实时时钟、UART 收发器等，为用户的外围扩展提供了灵活而廉价的选择。SPI 一般使用 4 条线：串行时钟线 SPSCCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和从机选择线 $\overline{SS}$ 。在阐述 SPI 的特性之前，我们先来了解几个概念。

1. 主机-从机（Master-Slave）

SPI 系统是典型的“主机-从机”系统。一个 SPI 系统，由一个主机和一个或多个从机构成，主机启动一个与从机的同步通讯，从而完成数据的交换。提供 SPI 串行时钟的 SPI 设备称为 SPI 主机或主设备（Master），其它设备则称为 SPI 从机或从设备（Slave）。在 MCU 扩展外设结构中，仍使用主机-从机（Master-Slave）概念，此时 MCU 必须工作于主机方式，外设工作于从机方式。

2. 从机选择引脚 $\overline{SS}$ （Slave Select）

从机选择引脚 $\overline{SS}$ 也称为片选引脚。若一个 MCU 的 SPI 工作于主机方式，则置该 MCU 的 $\overline{SS}$ 为高电平。若一个 MCU 的 SPI 工作于从机方式，当 $\overline{SS}=0$ 时表示主机选中了该从机，反之则未选中该从机。对于单主单从（One master and one slave）系统，可以采用图 8-1 的接法。对于一个主 MCU 带多个从 MCU 的系统，主机 MCU 的 $\overline{SS}$ 接高电平，每一个从机 MCU 的 $\overline{SS}$ 接主机 I/O 输出线，由主机控制其电平高低，以便主机选中该从机。

3. 主出从入引脚MOSI（Master Out/Slave In）

主出从入引脚 MOSI 是主机输出、从机输入数据线。对于 MCU 被设置为主机方式，主机通向从机的数据从该引脚输出。对于 MCU 被设置为从机方式，来自主机的数据从该引脚输入。

4. 主入从出引脚MISO（Master In/Slave Out）

主入从出引脚 MISO 是主机输入、从机输出数据线。对于 MCU 被设置为主机方式，来自从机的数据从该引脚输入主机。对于 MCU 被设置为从机方式，通向主机的数据从该引脚输出。

5. SPI串行时钟引脚SPSCK（SPI Serial Clock）

SPI 串行时钟引脚 SPSCK 用于控制主机与从机之间的数据传输。串行时钟信号由主机的内部总线时钟分频获得，主机的 SPSCK 引脚输出给从机的 SPSCK 引脚，控制整个数据的传输速度。在主机启动一次传送的过程中，从 SPSCK 引脚输出自动产生的 8 个时钟周期信号，SPSCK 信号的一个跳变进行一位数据移位传输。

6. 时钟极性

表示时钟信号在空闲时是高电平还是低电平。

7. 时钟相位

决定数据是在 SPSCK 的上升沿采样还是在 SPSCK 的下降沿采样。

8.1.2 SPI的数据传输

图 8-1 是 SPI 的主-从连接示意图，图中的移位寄存器为 8 位，所以每一工作过程相互传送 8 位数据，工作从主机 CPU 发出启动传输信号开始，此时要传送的数据装入 8 位移位寄存器，同时产生 8 个时钟信号从 SPSCK 引脚依次送出，在 SPSCK 信号的控制下，主机中 8 位移位寄存器中的数据依次从 MOSI 引脚送出，到从机的 MOSI 引脚后送入它的 8 位移位寄存器；在此过程中，从机的数据也可通过 MISO 引脚传送到主机中。所以，我们称之为全双工主-从连接（Full-Duplex Master-Slave Connections）。其数据的传输格式是高位（MSB）在前，低位（LSB）在后。

图 8-1 是一个主 MCU 和一个从 MCU 的连接，也可以一个主 MCU 与多个从 MCU 进行连接形成一个主机多个从机的系统；还可以多个 MCU 互联构成多主机系统；另外也可以一个 MCU 挂接多个从属外设。但是，SPI 系统最常见的应用是利用一个 MCU 作为主机，其它处于从机地位，这样，主机的程序启动并控制数据的传送和流向，在主机的控制下，从属设备从主机读取数据或向主机发送数据。

至于传送速度、何时数据移入移出、一次移动完成是否中断、如何定义主机从机等问题，可通过对寄存器编程来解决，随后会阐述这些问题。

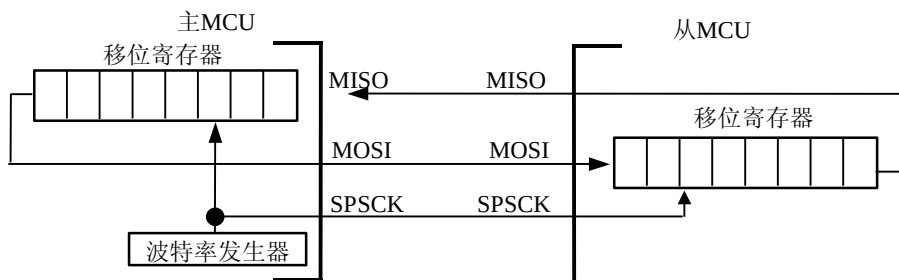


图8-1 SPI全双工主-从连接

8.1.3 SPI模块的时序

SPI 的数据传输是在时钟信号 SPSCK（同步信号）的控制下完成的。数据传输过程涉及到时钟相位与时钟极性两个概念。所谓时钟相位是指空闲时钟是高电平还是低电平，所谓时钟极性是指接收方从数据线上取数的时刻是在上升沿还是在下降沿。这样就有四种可能。以下讲解使用 CPHA 表达时钟相位，CPOL 表达时钟极性。四种可能情况分别如图 8-2、图 8-3、图 8-4 和图 8-5 所示。从时序图上可以看出，CPOL 为 1 或 0 时，时钟信号 SPSCK 的出现正好是反相关系，CPHA 用来选择两种不同的定时协议中的一种。主机和从机必须使用同样的时序模式，才能正常通信。总体要求是：确保发送数据在一周期开始的时刻上线，接收方在 1/2 周期的时刻从线上取数，这样是最稳定的通信方式。

从图 8-2 可以看出，当 CPHA=0，CPOL=0 时，MISO 引脚上的数据在第一个 SPSCK 沿跳变之前已经上线了，而为了保证正确传输，MOSI 引脚的 MSB 位必须与 SPSCK 的第一个边沿同步，在 SPI 传输过程中，首先将数据上线，然后在同步时钟信号的上升沿时，SPI 的接收方捕捉位信号，在时钟信号的一个周期结束时（下降沿），下一位数据信号上线，再重复上述过程，直到一个字节的 8 位信号传输结束。

图 8-3 是当 CPHA=0，CPOL=1 时的数据线和时钟线的时序，与图 8-2 唯一不同之处只是在同步时钟信号的下降沿时捕捉位信号，上升沿时下一位数据上线。

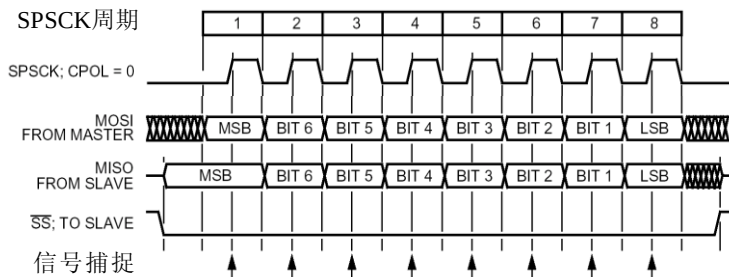


图8-2 CPHA=0，CPOL=0时的数据/时钟时序图

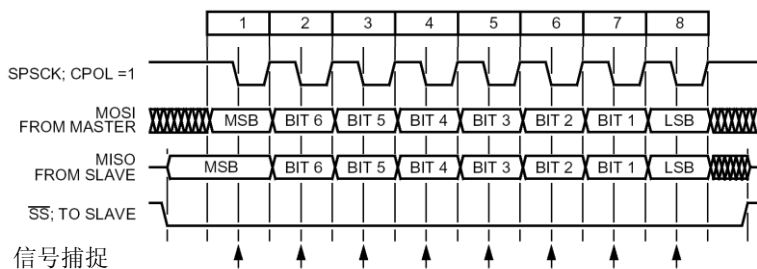


图8-3 CPHA=0, CPOL=1时的数据/时钟时序图

从图 8-4 可以看出，在 CPHA=1, CPOL=0 时，MISO 引脚和 MOSI 引脚上的数据的 MSB 位必须与 SPSCK 的第一个边沿同步，在 SPI 传输过程中，在同步时钟信号周期开始时（上升沿）数据上线，然后在同步时钟信号的下降沿时，SPI 的接收方捕捉位信号，在时钟信号的一个周期结束时（上升沿），下一位数据信号上线，再重复上述过程，直到一个字节的 8 位信号传输结束。

图 8-5 是当 CPHA=1, CPOL=1 时的数据线和时钟线的时序，与图 8-4 唯一不同之处只是在同步时钟信号的上升沿时捕捉位信号，下降沿时下一位数据上线。

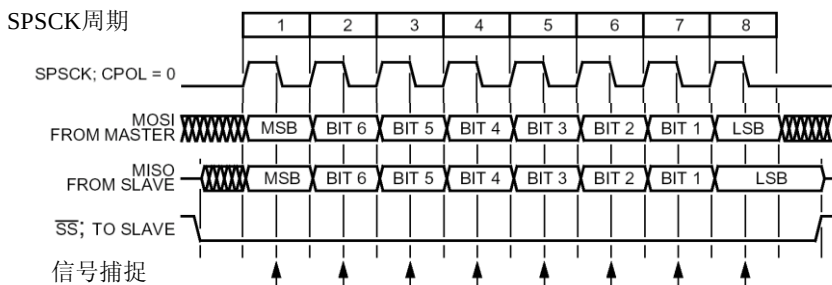


图8-4 CPHA=1, CPOL=0时的数据/时钟时序图

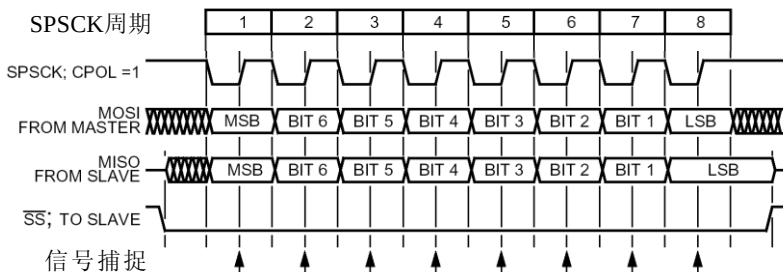


图8-5 CPHA=1, CPOL=1时的数据/时钟时序图

注意：SPI 主设备的时钟极性和相位是以从设备的时钟极性和相位为基准的，因此在配置 SPI 接口时钟时，必须先了解从设备的时钟要求。从设备何时接收数据，是在时钟的上升沿还是下降沿？何时输出数据，是在时钟的下降沿还是上升沿？由于主设备的接收引

脚与从设备的发送引脚相连接，主设备的发送引脚与从设备的接收引脚相连接，即从设备接收的数据是主设备的发送引脚发出的，主设备接收的数据是从设备发送引脚发出的，因此主设备接收数据的极性跟从设备接收数据的极性相反，跟从设备发送数据的极性相同。

举例说明：如果某从设备在时钟的上升沿输入（即接收）数据，在下降沿输出（即发送）数据。那么，主机的 SPI 时钟配置应为在时钟的上升沿输出数据，下降沿输入数据。因此，主设备 SPI 时钟极性应该配置为下降沿有效。只有正确地配置时钟极性和相位，数据才能够被准确的发送和接收。因此必须严格对照从设备的 SPI 接口时序来正确配置主设备时钟。

8.1.4 模拟SPI

对于不带 SPI 串行总线接口的 MCU 来说，可以使用软件来模拟 SPI 的操作。举例说明，我们可以使用三个普通 I/O 口，分别定义为 pin_SPSCCK、pin_MISO、pin_MOSI 来模拟 SPI 器件的 SPSCCK、MISO、MOSI。对于不同的串行接口外围芯片，它们的时钟时序可能有所不同。我们假设某 SPI 外围芯片在 SPSCCK 的上升沿输入数据和在下降沿输出数据。则我们初始化 pin_SPSCCK=1，在 MCU 正常工作后，置 pin_SPSCCK=0。这样，MCU 在输出 1 位 SPSCCK 时钟的同时，SPI 外围芯片串行左移一位，从而输出 1 位数据至 MCU 的 pin_MISO 线。然后再置 SPSCCK 为 1，则 MCU 的 pin_MOSI 线输出 1 位数据（先高位，后低位）到 SPI 外围芯片。至此，模拟 1 位数据输入输出便完成了。此后再置 SPSCCK=0，模拟下 1 位数据的输入输出……，依此循环 8 次，即可完成 1 次通过 SPI 总线传输 8 位数据的操作。对于在 SPSCCK 的下降沿输入数据和上升沿输出数据的器件，则应初始化 pin_SPSCCK=0，在 MCU 正常工作后，置 SPSCCK=1，使 SPI 器件输出 1 位数据（MCU 接收 1 位数据），之后再置 SPSCCK=1，使 SPI 外围芯片接收 1 位数据（MCU 发送 1 位数据），从而完成 1 位数据的传送。

8.2 MCF52233队列串行外设接口QSPI模块编程结构

8.2.1 QSPI概述

在 SPI 协议的基础上，Motorola 公司对 SPI 的功能进行了增强，增加了队列传输机制，推出了队列串行外围接口协议，即 QSPI 协议。与 SPI 相比，QSPI 结构最大的特点是以 80 字节的静态 RAM 取代了 SPI 的发送和接收数据寄存器。80 字节的静态 RAM 分成 3 部分：32 字节的发送 RAM，32 字节的接收 RAM 和 16 字节的命令 RAM。这 3 部分形成了具有 16 个 QSPI 传输控制组的传输队列，每个 QSPI 传输控制组由 1 个命令 RAM、1 个发送 RAM 和 1 个接收 RAM 组成。每个 QSPI 传输的数据长度、片选等信息可由该 QSPI 传输控制组的命令 RAM 单独决定。SPI 只有 1 个 8 位的发送数据寄存器，所以 CPU 每次最多只能发送一个字节数据。而 QSPI 拥有具有 16 个传输控制组的传输队列，所以 CPU 每次最多可以发送 16 个数据，并且可以通过命令 RAM 设置每个待发送数据的长度，每

个数据的长度为 8~16 位。一旦传输启动，直到传输结束都无需 CPU 干预，极大地提高了传输效率。

QSPI 的主要特点如下：

- (1) 无需用户干预，可传输多达 16 个队列
- (2) 支持 1 位增量的 8~16 位帧长度
- (3) 四条可控片选线，通过外接译码器可控制多达 15 个设备（注意：所有的片选都选中，即同时选中 16 个设备时，可能会导致控制错误）。
- (4) 在总线频率为 60MHz 的情况下，波特率范围为 117.6Kbps 到 15Mbps
- (5) 传输前和传输后的延时可以编程控制
- (6) 时钟相位和极性可以编程控制
- (7) 对于连续的传输，支持环绕模式

8.2.2 QSPI工作原理

QSPI 模块的结构如图 8-6 所示。

80 字节的 RAM 不出现在设备内存映像中，因为它不能被直接访问，只能通过读写 QSPI 地址寄存器（QAR）和数据寄存器（QDR）来间接访问。写 QDR 则数据将被写入到 QAR[ADDR]指定的内存地址中，此时 QAR 增加。相应地，读 QDR 将返回 QAR[ADDR]指定的内存地址处的值，此时 QAR 也会增加。发送 RAM 和命令 RAM 是只写的。接收 RAM 是只读的。

在 QSPI 模块的环绕寄存器（QWR）中，NEWQP 和 ENDQP 域分别决定了传输队列的起始点和结束点。起始点和结束点可以是 16 个 QSPI 传输控制组中的任意一个。当 QSPI 传输启动时，QSPI 模块将从起始点开始依次发送准备好的数据直到结束点，整个过程无需 CPU 干预。典型的 QSPI 传输流程如图 8-7 所示，其中 QP 为队列内部指针，指向当前执行的命令。

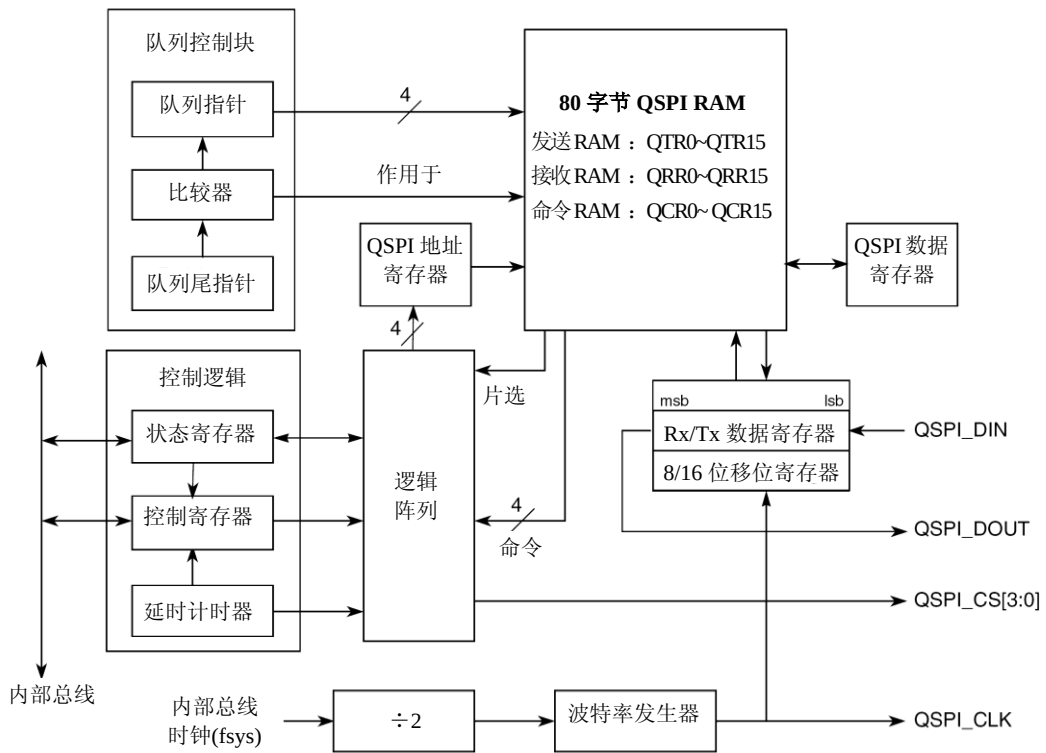


图8-6 QSPI模块的结构

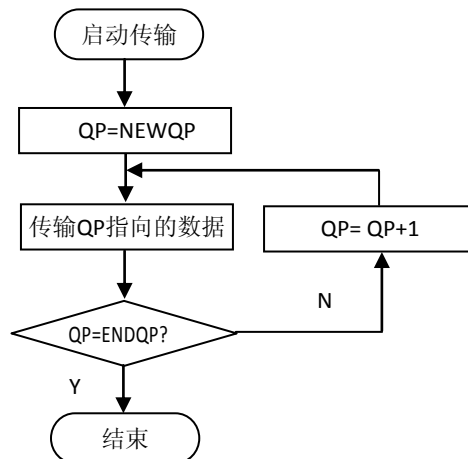


图8-7 QSPI传输流程

QSPI 模块有 7 个引脚（与 SPI 有所不同）：QSPI_DIN 为串行数据输入引脚；QSPI_DOUT 为串行数据输出引脚；QSPI_CLK 为串行时钟输出；QSPI_CS[3:0]为片选信号。

号，通过外接译码器可以选择多达 16 个设备。

QSPI 模块和其他 SPI 外围设备之间的接口实例如图 8-8 所示：

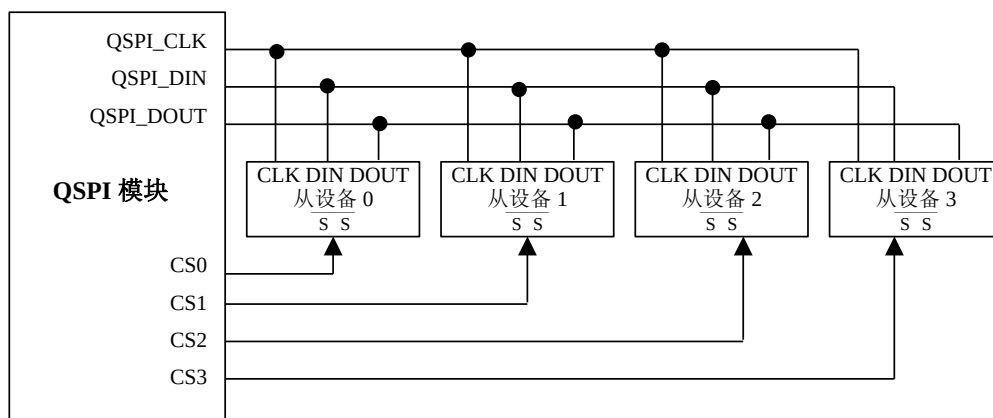


图8-8 QSPI一主三从连接

8.2.3 QSPI功能描述

QSPI 共使用 4 个队列指针：

- (1) 新队列指针（QWR[NEWQPI]）——指向队列中的第一条命令
- (2) 队列内部指针——指向当前执行的命令
- (3) 完成队列指针（QWR[CPTQP]）——指向已被执行的最近的一条命令
- (4) 结束队列指针（QWR[ENDQP]）——指向队列中的最后一条命令

用户可以通过 QSPI 环绕寄存器（QWR）来访问其中三个。初始化时队列内部指针与 QWR[NEWQPI]的值相同。QSPI 正常工作时，将重复以下操作：

- (1) 执行队列内部指针指向的命令
- (2) 将队列内部指针的值复制到 QWR[CPTQP]中
- (3) 队列内部指针加 1

只要 QWR[NEWQPI]的值不改变，QSPI 就连续执行队列内部指针地址中的命令。每执行完一条命令，比较 QWR[ENDQP]和 QWR[CPTQP]的值，如果这两个指针相同，则 QSPI 中断寄存器(QIR)的 SPIF 被置位，此时除非使能了环绕模式，否则 QSPI 将停止。置位 QWR 的 WREN 将使能环绕模式。QWR[NEWQPI]复位时会被清零。当 QSPI 使能时，除非 QWR[NEWQPI]被写入了其他的值，否则命令将从地址 0x0 处开始执行。QWR[ENDQP]复位时会被清零，但是在 QSPI 使能前它将被改变用来显示最近使用的一个队列节点。在任意时刻，都可以对 QWR[NEWQPI]和 QWR[ENDQP]进行写操作。当 QWR[NEWQPI]的值改变时，内部指针的值也跟着改变，除非传输正在进行，不过即使是传输正在进行也可以正常完成。如果 QSPI 使能，若将 QWR[NEWQPI]和 QWR[ENDQP]都置为 0x0 则将产生一次传输。通过设置 QMR[CPHA, CPOL]可以产生四种相位和极性组合中的任意一种，用于与传输数据相关的 QSPI_CLK。数据从高位（MSB）开始发送。默认传输位数为 8

位，但是可以通过命令寄存器（QCR）中的 BITSE 位，设置传输位数为 8~16 之间的任意值。通过设置 QSPI 模式寄存器（QMR）中的 4 个 BITS 位可以设置每一帧的位数。

8.2.4 QSPI 模块寄存器

从程序员的角度看，QSPI 模块有 6 个可以直接访问的 16 位寄存器，另外还有 16 个可间接访问的命令寄存器。表 8-1 给出了这些寄存器的基本信息。只要理解和掌握这些寄存器的用法，就可以进行 QSPI 编程了。除命令寄存器为只写外，其他寄存器都可进行读写。

表 8-1 QSPI 寄存器简介

偏移地址	寄存器名称与缩写	基本功能
0x00_0340	QSPI 模式寄存器(QMR)	配置 QSPI 的基本操作模式(时钟相位、时钟极性 etc)
0x00_0344	QSPI 延时寄存器(QDLYR)	初始化主机模式传输并设置各种延时参数
0x00_0348	QSPI 环绕寄存器(QWR)	控制放弃传输模式，环绕设置，设置队列指针等
0x00_034C	QSPI 中断寄存器(QIR)	控制 QSPI 中断和状态标志
0x00_0350	QSPI 地址寄存器(QAR)	指定 QSPI RAM 读写操作的作用范围
0x00_0354	QSPI 数据寄存器(QDR)	CPU 通过该寄存器读写 QSPI RAM
QAR[ADDR]	命令 RAM 寄存器 (QCR0~QCR15)	通过命令 RAM 设置操作模式

1. QSPI 模式寄存器（QMR）

该寄存器决定 QSPI 模块的基本操作模式。QSPI_CLK 时钟相位和极性，波特率，主机操作模式，传输长度等都由该寄存器决定。QMR 的地址为 0x00_0340。各位定义如下。

数据位	D15	D14	D13-D10				D9	D8	D7-D0							
定义	MSTR	DOHIE	BITS				CPOL	CPHA	BAUD							
复位	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0

注意：由于 QSPI 不能工作在从机模式，所以为了使 QSPI 正常工作，主机模式使能位 MSTR 必须被置位。

D15—MSTR，主机模式使能位。0：未定义；1：QSPI 配置为主机模式。要想正确操作必须配置 QSPI 模块为主机模式。

D14—DOHIE，数据输出高阻态使能位。该位选择 QSPI_DOUT 操作模式。0：QSPI_DOUT 在传输期间有效；1：QSPI_DOUT 在传输期间为高阻态。

D13~D10—BITS，传输位数。定义队列中的每个节点的数据传输位数。0000：16 位传输；0001~0111：保留；1000~1111：分别代表 8~15 位传输。

D9—CPOL，时钟极性位。该位定义 QSPI_CLK 的时钟极性。0：QSPI_CLK 空闲电平为逻辑 0 电平；1：QSPI_CLK 空闲电平为逻辑 1 电平。

D8—CPHA，时钟相位位。该位定义 QSPI_CLK 的时钟相位。0：数据在 QSPI_CLK 的第一个沿捕获数据并在下一个沿改变数据；1：数据在 QSPI_CLK 的第一个沿改变数据

并在下一个沿捕获数据。

D7~D0—BAUD，波特率分频因子。通过在该域写一个 2~95 的值来设置波特率。写 0 禁止 QSPI。写 1 为无效设置。期望的 QSPI_CLK 波特率与内部总线时钟有关，QMR[BAUD] 的值由下式决定：

$$\text{QMR[BAUD]} = f_{\text{sys}} / (2 \times [\text{期望的 QSPI_CLK 波特率}])$$

2. QSPI 延时寄存器（QDLYR）

QDLYR 用于初始化主机模式传输并设置各种延时参数。QDLYR 的地址为 0x00_0344。各位定义如下。

数据位	D15	D14-D8						D7-D0							
定义	SPE	QCD						DTL							
复位	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

D15—SPE，QSPI 使能位。当置位时，QSPI 通过执行命令 RAM 中的命令初始化主机模式传输。当一次传输结束时自动清该位。当 QSPI 中断寄存器（QIR）的 ABRTL 位被置位时，用户可以清该位来放弃传输。但推荐使用置位 QSPI 环绕寄存器（QWR）的 HALT 位的方法来放弃传输。

D14~D8—QCD，QSPI_CLK 延时。当命令 RAM 寄存器（QCR）中的 DSCK 位被置位时，该域决定从片选有效到 QSPI_CLK 有效之间的延时时间，也称“传输前延时”。

D7~D0—DTL，传输后延时。当命令 RAM 寄存器（QCR）中的 DT 位被置位时，该域决定串行传输之后的延时时间。

3. QSPI 环绕寄存器（QWR）

该寄存器控制放弃传输模式、环绕设置、队列指针位置等。QWR 的地址为 0x00_0348。各位定义如下。该寄存器复位全为 0。

数据位	D15	D14	D13	D12	D11-D8	D7-D4	D3-D0
定义	HALT	WREN	WRTO	CSIV	ENDQP	CPTQP(只读)	NEWQP

D15—HALT，终止传输位。置该位将使 QSPI 在当前命令执行完以后停止执行后续命令。

D14—WREN，环绕模式使能位。0：在执行完 QWR[ENDQP]所指的命令之后停止命令的执行。1：在执行完 QWR[ENDQP]所指的命令之后，返回到节点 0 或者 QWR[NEWQP]指向的节点，继续执行命令。

D13—WRTO，环绕位置位。该位决定环绕模式下 QSPI 从何处环绕，即执行完 QWR[ENDQP]所指的命令以后，从何处开始重新开始执行命令。0：环绕到 RAM 节点 0。1：环绕到 QWR[NEWQP]所指的 RAM 节点。

D12—CSIV，QSPI_CS 空闲时的电平状态位。0：当传输过程中当前命令 RAM 节点不被驱动时，QSPI 片选输出归零。（也就是说，空闲状态为 0，片选信号高有效）。1：当传输过程中当前命令 RAM 节点不被驱动时，QSPI 片选输出为 1。（也就是说，空闲状态为 1，片选信号低有效）。

D11~D8—ENDQP 队列尾指针。该指针指向队列中最后一条命令。

D7~D4—CPTQP 完成队列指针。该指针指向已执行队列中的最近一条命令。

D3~D0—NEWQP 队列起始指针。该指针指向 QSPI 将要传输的 RAM 中的第一个队列节点。

4. QSPI中断寄存器（QIR）

该寄存器控制传输中断使能和状态标志。QWR 的地址为 0x00_034C。复位全为 0。

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7-D4	D3	D2	D1	D0
定义	WCEFB	ABRTB		ABRTL	WCEFE	ABRTE		SPIFE		WCEF	ABRT		SPIF

D15—WCEFB，写冲突错误请求使能位。在数据传输期间，CPU 试图写一个正在被 QSPI 执行的节点的数据寄存器（QDR）时，将发生写冲突错误。当该位被置位时，写 QDR 将发生访问错误。

D14—ABRTB，异常访问错误使能。在传输期间，若 QDLYR 的 SPE 位被清零将发生异常。当该位被置位时，如果试图在传输期间清 QDLYR 的 SPE 位将发生访问错误。

D13，D9，D7~D4，D1—未定义。

D12—ABRTL，异常锁定位。当该位被置位时，QDLYR 的 SPE 位不能通过写 QDLYR 清零。QDLYR 的 SPE 位只能在传输结束时由 QSPI 来清零。

D11—WCEFE，写冲突（WCEF）中断使能位。0：禁止写冲突（WCEF）中断；1：允许写冲突（WCEF）中断。

D10—ABRTE，异常（ABRT）中断使能位。0：禁止异常中断；1：允许异常中断。

D8—SPIFE，QSPI 完成中断（SPIF）使能位。0：禁止 SPIF 中断；1：允许 SPIF 中断。

D3—WCEF，写冲突错误标志位。写冲突错误发生表明 CPU 试图访问一个正在被 QSPI 执行的节点。写 1 将清该位，写 0 没有意义。

D2—ABRT，异常标志位。异常发生表明用户通过将 QDLYR 中的 SPE 位清零来终止 QSPI 传输，而不是等待 QSPI 命令队列执行完成才终止 QSPI 传输。写 1 将清该位，写 0 没有意义。

D0—SPIF，QSPI 结束标志。当 QSPI 命令队列中所有的命令都执行完毕时该位被置位。当执行完 QWR[ENDQP]所指向的命令时，或 QWR 的 HALT 位被置位以后当前命令执行完毕时，该位被置位。环绕模式下，每次执行完 QWR[ENDQP]所指向的命令以后该位被置位。写 1 将清该位，写 0 没有意义。

5. QSPI地址寄存器（QAR）

该寄存器指定 QSPI RAM 读写操作的作用范围。发送 RAM 的地址范围为 0x0 到 0xF，接收 RAM 的地址范围为 0x10 到 0x1F，命令 RAM 的地址范围为 0x20 到 0x2F（这些地址是相对于 QSPI RAM 地址空间而言的，而不是内存映像）。

注意：对 QSPI RAM 读或写操作将致使 QAR 增加。但是，QAR 在每一个 RAM 区域内不环绕。应用软件必须谨防地址域错误，也就是说应用软件指定的地址范围不应超过 QAR 的有效地址范围。

QAR 的地址为 0x00_0350。D15~D6 未定义。D5~D0 为 ADDR 域，该地址域为读/写

QSPI RAM 的地址。0x00~0x0F 为发送 RAM，0x10~0x1F 为接收 RAM，0x20~0x2F 为命令 RAM，0x30~0x3F 保留。

6. QSPI数据寄存器（QDR）

QDR 用于间接访问 QSPI RAM。CPU 通过该寄存器读写 QSPI RAM。写 QDR 则数据将被写入到 QAR[ADDR]指定的内存地址中，此时 QAR 增加。相应的，读 QDR 将返回 QAR[ADDR]指定的内存地址处的值，此时 QAR 也会增加。

QDR 的地址为 0x00_0354。D15~D0 为 DATA 域。写该域将使数据被写入到 QAR[ADDR]所指定的 QSPI RAM 中。相似的，读该域将返回 QSPI RAM 中 QAR[ADDR]指定地址处的数据。在命令（QAR[ADDR]=0x20~0x2F）访问过程中，只使用该域的高字节（LMB）字节。

7. 命令RAM寄存器（QCR0~QCR15）

通过访问 QDR 的高字节可以访问命令 RAM。QSPI 不能修改命令 RAM 中的信息。QSPI 有 16 字节的命令 RAM。每一个字节被分成 2 部分：片选域和传输命令域。片选域使能进行传输的外围设备。传输命令域选择传输属性。

注意：命令 RAM 的访问只使用 QDR 的高（LMB）字节，间接访问的地址基址为 QAR[ADDR]。

QCR 的地址为 QAR[ADDR]。各位定义如下。

数据位	D15	D14	D13	D12	D11-D8	D7-D0
定义	CONT	BITSE	DT	DSCK	QSPI_CS	

D15—CONT，连续传输位。0：每个字传输完毕后片选信号恢复到由 QWR 的 CSIV 位定义的空闲电平状态；1：只有当整个队列节点（最多 16 个字）全部传输完毕后片选信号才恢复到由 QWR 的 CSIV 位定义的空闲电平状态。

注意：如果传输长度大于 16 个字，若要保持片选有效，则 QWR 的 CSIV 位必须被置成特定的电平状态，即第一次传输结束后片选返回到的电平状态。

D14—BITSE，每次传输使能位。0：传输数据位为 8 位；1：传输数据位为 QMR 的 BITS 中设置的位。

D13—DT，传输后延时使能位。0：复位默认值；1：QSPI 允许在紧接着的帧传输之间插入延时。延时周期由 QDLYR 的 DTL 位域决定。传输后延时可以提供取消外围设备选择的时间间隔。

D12—DSCK，从片选有效到有效的 QSPI_CLK 跳变之间的延时。0：从片选有效到有效的 QSPI_CLK 跳变之间延时为半个 QSPI_CLK 周期；1：从片选有效到有效的 QSPI_CLK 跳变之间延时为 QDLYR 的 QCD 位域指定的值。

D11~D8—QSPI_CS，片选位。该位用于选择要进行数据传输的外设。允许多个片选同时有效，且多个外设可以连接在同一个片选引脚上。D11~D8 位将直接映射到相应的 QSPI_CS 引脚上。如果需要 4 个以上的片选，可以在 QSPI_CS_n 引脚处外接译码器。

D7~D0：未定义。

8.2.5 发送延时

QSPI 支持两种延时，并可独立编程设置。这两种延时是：

- (1) 从 QSPI_CS 有效到第一个有效的 QSPI 时钟沿之间的延时，也称“传输前延时”。
- (2) 连续两次发送间的延时，也称“传输后延时”。

每个命令入口的传输前延时控制位 DSCK，控制从 QSPI_CS 有效到第一个 QSPI_CLK 上升沿之间的延时。置位 DSCK，则插入一个用户指定的较长延时。此时，延时长度由 QDLYR 的 QCD 指定（延时长度=QCD/fsys，QCD 的范围为 1~127）。当 QDLYR 的 QCD 或 QCR 的 DSCK 等于 0 时，延时长度为一个标准的 1/2QSPI 时钟。

传输后延时使能位 DT，控制从 QSPI_CS 信号无效到下一次传输启动之间的延时。DT 为取消外围设备选择提供时间。在连续发送之间（比如串行 AD 转换）可以加一个延时。用户可以选择在传输后延时一个标准周期，也可以从一开始就定义一个较长的延时周期。若 DT=1，则传输后延时由 QDLYR 的 DTL 指定（传输后延时=32 × DTL/fsys，DTL 的取值范围为 1~255）。

为了满足特殊的 SPI 需求，可以在任何一个命令入口中将 DT 和 DSCK 同时置 1，以允许传输前和传输后延时。若 DTL 为 0，则可以产生一个 8192/fsys 的传输后延时。

注意：在传输较长的数据流时，必须指定合适的延时。因为 QSPI 模块需要时间来加载发送 RAM 节点。

8.3 MCF52233的串行外设接口QSPI模块编程实例

在进行 QSPI 编程时，程序初始化部分必须进行 QSPI 模块初始化。QSPI 的初始化，主要是对 QSPI 模式寄存器 QMR 进行写入，定义其基本操作模式（时钟相位、时钟极性 etc）。同时，还要对延时寄存器、环绕寄存器、中断寄存器、地址寄存器、数据寄存器等写入，以便配置是否启用环绕模式、中断方式等。本节以 MCF52233 MCU 为例，按照构件的设计原则，给出 QSPI 构件的头文件以及若干通信函数的设计。

1. QSPI通信头文件：QSPI.h

```
//-----*
// 文件名： QSPI.h (QSPI驱动程序头文件)          *
//-----*
#ifndef QSPI_H          //防止重复定义
#define QSPI_H
//1. 头文件
#include "Type.h"
#include "mcf52233.h"
#include "sysinit.h"
//QSPI RAM 空间
#define QSPI_TX_RAM 0x00    //发送RAM起始地址0x00
#define QSPI_RX_RAM 0x10    //接收RAM起始地址0x10
```

```

#define QSPI_COMMAND_RAM 0x20 //命令RAM起始地址0x20
//命令RAM的控制字
#define QSPI_COM_CONT 0x5100
//QCR 15 14 13 12 11-8 7-0
//      |  |  |  |  |  |  |  | 保留位为0
//      |  |  |  |  |  |  |  |  QSPI_CS=1111(高/低电平有效由QWR的CSIV位决定)
//      |  |  |  |  |  |  |  |  DSCK=1使能从片选有效到有效的QSPI_CLK之间的延时
//      |  |  |  |  |  |  |  |  DT=0 一个标准的1/2 QSPI时钟
//      |  |  |  |  |  |  |  |  BITSE=1 传输QMR[BITS]中设置的位
//      |  |  |  |  |  |  |  |  CONT=0 不连续发送

//QSPI初始化配置参数:
typedef struct QSPI_InitPara
{
    uint8 CPHA;    //CPHA:时钟相位标志. 0: 在QSPI_CLK上升沿接收数据, 下降
                  //沿发送数据. 1: 在QSPI_CLK上升沿发送数据, 下降沿接收数据

    uint8 CPOL;    //CPOL: 时钟极性标志. 0: QSPI_CLK空闲电平为0电平。
                  // 1: QSPI_CLK空闲电平为1电平。

    uint8 fsys;    //内部时钟时钟. fsys范围因芯片类型而异. 单位MHz。
    uint16 Baud;   //传输波特率. 内部时钟为60MHz时, 范围为:117.6~15000 Kbps
                  //内部总线为66MHz时, 范围为:129.4~16600 Kbps。

    uint8 bitcnt;  //QSPI传输数据的位数, QSPI支持8~16位数据传输

    uint8 QCD;    //QCD:QSPI_CS有效到第一个QSPI_CLK上升沿之间的延时长度, 也
                  //称传输前延时。
                  //当命令RAM中DSCK=0或QCD=0时, 延时长度为一个标准的1/2
                  //QSPI时钟. 当命令RAM中DSCK=1时, QCD=1~127时, 延时计算
                  //公式为: 延时时长度=QCD/fsys
    uint8 DTL;    //DTL:从QSPI_CS信号无效到下一次传输启动之间的延时, 也称传
                  //输后延时。
                  //当命令RAM中DT=0时, 则延时长度为一个标准的1/2 QSPI 时钟
                  //当命令RAM中DT=1, DTL=1~255时, 传输后延时=32 × DTL/fsys
                  //当命令RAM中DT=1, DTL=0时, 传输后延时=8192/fsys。

    uint8 csiv;    //QSPI_CS空闲电平约定. 0:空闲电平为0; 1: 空闲电平为1。

    uint8 NEWQP;   //NEWQP: QSPI将要传输的RAM中的第一个队列节点号。(0~15)

    uint8 WREN;    //WREN, 环绕模式标志. WREN =0, 在执行完所有命令以后停止命
                  //令的执行. WREN =1, 在执行完所有命令以后返回到节点0, 或
                  //QWR[NEWQP]指向的节点继续执行命令。
    uint8 WRT0;    //WRT0, 环绕位. 该位决定环绕模式下QSPI从何处环绕, 即执行
                  //完命令以后从何处开始重新开始执行命令)当WREN=0时, 该位忽
                  //略. 当WREN=1时, 若WRT0=0, 环绕到RAM节点0执行. 若WRT0=1,
                  //则环绕到QWR[NEWQP]所指的RAM节点。

```

```

uint8 SPIFE; //SPIFE发送完成中断使能位。1:使能发送完成中断 0: 禁止发送
//完成中断
}QSPI_InitPara;

//QSPI通信相关函数声明
void QSPI_init(QSPI_InitPara Para); //MCF5223x的QSPI模块初始化函数
void QSPI_WriteToRAMN(uint8 address, uint16 data[], uint8 N); //写N个数据
uint16 QSPI_ReadFromRAMN(uint8 address, uint16 data[], uint8 N); //读N个数据
uint8 QSPI_StartTrans(uint8 bytes); //启动QSPI发送函数
#endif

```

2. QSPI初始化函数QSPI_init

```

//-----*
//函数名: QSPI_init *
//功 能: 初始化MCF5223x的QSPI模块 *
//返 回: 无 *
//说 明: QSPI不能工作在从机模式下 *
//-----*
void QSPI_init(QSPI_InitPara QSPIpara)
{
    //1. 主机模式、传输位数以及波特率配置
    MCF_QSPI_QMR = (0
                    |MCF_QSPI_QMR_MSTR
                    |MCF_QSPI_QMR_BITS(QSPIpara.bitcnt)
                    |MCF_QSPI_QMR_BAUD(QSPIpara.fsys*1000/(2*QSPIpara.Baud)));
    //2. 时钟相位和极性配置
    if(QSPIpara.CPHA==1)
        MCF_QSPI_QMR |=MCF_QSPI_QMR_CPHA ;
    if(QSPIpara.CPOL==1)
        MCF_QSPI_QMR |=MCF_QSPI_QMR_CPOL ;
    MCF_QSPI_QDLYR = (0
                      |MCF_QSPI_QDLYR_DTL(QSPIpara.QCD)
                      |MCF_QSPI_QDLYR_QCD(QSPIpara.DTL));
    //3. 片选信号空闲电平配置
    if(QSPIpara.csiv==1)
        MCF_QSPI_QWR=0|MCF_QSPI_QWR_CSIV|MCF_QSPI_QWR_NEWQP(QSPIpara.NEWQP);
    else
        MCF_QSPI_QWR =0 | MCF_QSPI_QWR_NEWQP(QSPIpara.NEWQP);
    //4. 环绕模式配置
    if(QSPIpara.WREN==1)
    {
        if(QSPIpara.WRTO==1)
            MCF_QSPI_QWR = MCF_QSPI_QWR_WRTO
                          |MCF_QSPI_QWR_WREN;
        else
            MCF_QSPI_QWR = MCF_QSPI_QWR_WRTO;
    }
    //5. 中断配置

```

```

    if(QSPIpara.SPIFE==1)
        MCF_QSPI_QIR =0|MCF_QSPI_QIR_SPIFE;
//6. QSPI模块引脚配置:
//QSPI共使用7个引脚:QSPI_DOUT, QSPI_DIN, QSPI_CLK, CS0, CS1, CS2, CS3
//引脚分配: PQSPAR0-QSPI_DOUT    PQSPAR1-QSPI_DIN    PQSPAR2-QSPI_CLK
//           PQSPAR3-QSPI_CS0     PQSPAR4-QSPI_CS1    PQSPAR5-QSPI_CS2
//           PQSPAR6-QSPI_CS3
//PQSPAR:端口QSPI引脚分配寄存器    0x01:主功能
MCF_GPIO_PQSPAR =(0
                |MCF_GPIO_PQSPAR_PQSPAR0(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR1(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR2(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR3(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR4(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR5(0x1)
                |MCF_GPIO_PQSPAR_PQSPAR6(0x1));
}

```

3. 通过QSPI发送1个数据

```

//-----*
//函数名: QSPI_WriteToRAM1                                *
//功 能: 写一个数据到发送RAM                              *
//参 数: address:发送RAM起始地址                          *
//           data :要写入发送RAM的数据                    *
//返 回: 无                                                *
//说 明:                                                  *
//QSPI模块RAM(包括发送接收RAM和命令RAM)可以通过QAR 和 QDR间接访问。QAR决定 *
//写RAM的起始地址。QDR为数据存放寄存器, 写入到QDR寄存器中的数据将按照QAR指 *
//定的地址依次放后位置。                                  *
//-----*
void QSPI_WriteToRAM1( uint8 address, uint16 data )
{
    MCF_QSPI_QAR = address;
    MCF_QSPI_QDR = data;
}

```

4. 通过QSPI发送N个数据

```

//-----*
//函数名: QSPI_WriteToRAMN                                *
//功 能: 写N个数据到发送RAM                              *
//参 数: address:发送RAM起始地址                          *
//           data:要写入发送RAM的数组                    *
//           N:写入数据长度                                *
//返 回: 无                                                *
//-----*
void QSPI_WriteToRAMN(uint8 address, uint16 data[], uint8 N)
{

```

```

uint8 i=0;
for(;i<N;i++)
{
    QSPI_WriteToRAM1(address + i,data[i]);
} //发送数据
}

```

5. 通过QSPI接收1个数据

```

//-----*
//函数名: QSPI_ReadFromRAM1 *
//功 能: 接收一个数据 *
//参 数: address:接收RAM起始地址 *
//返 回: 接收到的数据 *
//说 明: *
//QSPI模块RAM(包括发送接收RAM和命令RAM)可以通过QAR 和 QDR间接访问。QAR决定 *
//读RAM的起始地址。读QDR寄存器中可以得到相应RAM区的数据 *
//-----*
uint16 QSPI_ReadFromRAM1( uint8 address )
{
    MCF_QSPI_QAR = address;
    return( MCF_QSPI_QDR );
}

```

6. 通过QSPI接收N个数据

```

//-----*
//函数名: QSPI_ReadFromRAMN *
//功 能: 接收N个数据 *
//参 数: address:接收RAM起始地址 *
//      data[]:接收数据存放位置 *
//      N:数据个数 *
//返 回: 接收到的数据 *
//说 明: *
//QSPI模块RAM(包括发送接收RAM和命令RAM)可以通过QAR 和 QDR间接访问。QAR决定 *
//读RAM的起始地址。读QDR寄存器中可以得到相应RAM区的数据 *
//-----*
uint16 QSPI_ReadFromRAMN(uint8 address,uint16 data[],uint8 N)
{
    uint8 i=0;
    for(;i<N;i++)
    {
        data[i]= QSPI_ReadFromRAM1(address);
    }
}

```

7. QSPI发送/接收配置函数

```

//-----*
//函数名: QSPI_StartTrans *

```

```

//功 能：该函数启动QSPI传输,并轮询SPIF标志以判断传输是否结束      *
//参 数：bytes:QSPI传输数据长度                                      *
//返 回：发送是否成功标志 0：失败 1：成功                            *
//-----*
uint8 QSPI_StartTrans(uint8 bytes)
{
    uint16 i=0;
    MCF_QSPI_QIR |= MCF_QSPI_QIR_SPIF;          //清发送完成中断
    MCF_QSPI_QWR |=MCF_QSPI_QWR_ENDQP(bytes-1);
    MCF_QSPI_QDLYR |= MCF_QSPI_QDLYR_SPE;       //启动SPI传输
    for(;i<0xFFFFE;i++)                        //限时等待
    {
        if(MCF_QSPI_QIR & MCF_QSPI_QIR_SPIF)    //轮询QSPI传输结束使能位SPIF
            //同步接收数据 接收数据代码
            return (1);
    }
    return (0) ;
}

```

在随书光盘的“···\Chap08”文件夹中包含带有 QSPI 接口的液晶显示模块的编程实例，有兴趣的读者可以按照光盘上的内容进行实验。

第9章 I2C模块及应用实例

I2C 模块具有与外部设备进行半双工同步通信的功能。本章首先给出了该模块的具体工作原理、寄存器介绍以及相关的编程方法，随后给出了编程实例。

多主设备 I2C 接口是两线双向串行总线接口，是不同设备间传输数据的简单而有效的方式。该接口是为 MCU 和其他支持 I2C 协议的设备进行通信而设计的，由硬件产生开始（START）和停止（STOP）信号，每传输 1 字节均可产生中断。该总线适用于在多个设备间进行频繁的短程通信，使用灵活，向总线上添加新的设备极其容易。

9.1 I2C模块

I2C（Inter-Integrated Circuit）总线，有些文献记作 IIC、I2C 或 I²C，本书统一使用 I2C，主要用于同一电路板内各集成电路模块（Inter-Integrated，IC）之间的连接。I2C 采用双向 2 线制串行数据传输方式，支持所有 IC 制造工艺，简化 IC 间的通信连接。I2C 是 PHILIPS 公司于 20 世纪 80 年代初提出，其后 PHILIPS 和其他厂商提供了种类丰富的 I2C 兼容芯片。目前 I2C 总线标准已经成为世界性的工业标准。

9.1.1 I2C总线特点

SPI 采用的是数据输入、数据输出、时钟三线加片选线形式，对于少数外围器件的扩展极为方便。但当外围器件较多时，就要有相当多的片选线，而且由这种总线组成的多机系统中只能有一个主机。相比之下 I2C 总线克服了 SPI 的不足。在硬件结构上，它采用数据（SDA）和时钟（SCL）两根线来完成数据的传输及外围器件的扩展，数据 SDA 和时钟 SCL 都是开漏的，通过一个上拉电阻接到正电源，因此在不需要的时候仍保持高电平。任何具有 I2C 总线接口的外围器件，不论其功能差别有多大，都具有相同的电气接口，因此都可以挂接在总线上，甚至可在总线工作状态下撤除或挂上，使其连接方式变得十分简单。对各器件的寻址是软寻址方式，因此节点上没有必须的片选线，器件地址给定完全取决于器件类型与单元结构，这也简化了 I2C 系统的硬件连接。另外 I2C 总线能在总线竞争过程中进行总线控制权的仲裁和时钟同步，不会造成数据丢失，因此由 I2C 总线连接的多机系统可以是一个多主机系统。

I2C 主要特点总结如下。

（1）在硬件上，二线制的 I2C 串行总线使得各 IC 只需最简单的连接，而且总线接口都集成在 IC 中，不需另加总线接口电路。电路的简化省去了电路板上的大量走线，减少了电路板的面积，提高了可靠性，降低了成本。在 I2C 总线上，各 IC 除了个别中断引线外，相互之间没有其他连线，用户常用的 IC 基本上与系统电路无关，故极易形成用户自己的标准化、模块化设计。

(2) I2C 总线还支持多主控 (multi-mastering)，如果两个或更多主机同时初始化数据传输，可以通过冲突检测和仲裁防止数据被破坏。其中任何能够进行发送和接收的设备都可以成为主机。一个主机能够控制信号的传输和时钟频率。当然在任何时间点上只能有一个主机。

(3) 串行的 8 位双向数据传输位速率在标准模式下可达 100kbps，快速模式下可达 400kbps，高速模式下可达 3.4Mbps。

(4) 连接到相同总线的 IC 数量只受到总线最大电容 (400pf) 的限制。但如果在总线中加上 82B715 总线远程驱动器可以把总线电容限制扩展十倍，传输距离可增加到 15m。

9.1.2 I2C总线标准的发展历史

从飞利浦公司提出 I2C 总线标准到现在，I2C 已经发展了二十几个年头。

1992 年 PHILIPS 首次发布 I2C 总线规范 Version1.0，并取得专利。

1998 年 PHILIPS 发布 I2C 总线规范 Version2.0，至此标准模式和快速模式的 I2C 总线已经获得了广泛应用，标准模式传输速率为 100kbps，快速模式 400kbps。同时，I2C 总线也由 7 位寻址发展到 10 位寻址，满足了更大寻址空间的需求。

随着数据传输速率和应用功能的迅速增加，2001 年 PHILIPS 又发布了 I2C 总线规范 Version2.1，完善和扩展了 I2C 总线的功能，并提出了传输速率可达 3.4Mbps 的高速模式，这使得 I2C 总线能够支持现有及将来的高速串行传输，如 EEPROM 和 Flash 存储器等。

目前 I2C 总线已经被大多数的芯片厂家所采用，较为著名的有 ST Microelectronics、Texas Instruments、Xicor、Intel、Maxim、Atmel、Analog Devices 和 Infineon Technologies 等，I2C 总线标准已经属于世界性的工业标准。I2C 总线始终和先进技术保持同步，但仍然保持向下兼容。相信随着技术的进一步成熟，I2C 总线将会有更广泛的应用。

9.1.3 I2C总线的相关术语

本章在介绍 I2C 总线过程中涉及到以下术语：

(1) **主机（主控器）**：在 I2C 总线中，提供时钟信号，对总线时序进行控制的器件。主机负责总线上各个设备信息的传输控制，检测并协调数据的发送和接收。主机对整个数据传输具有绝对的控制权，其它设备只对主机发送的控制信息作出响应。如果在 I2C 系统中只有一个 MCU，那么通常由 MCU 担任主机。

(2) **从机（被控器）**：在 I2C 系统中，除主机外的其它设备均为从机。主机通过从机地址访问从机，对应的从机作出响应，与主机通信。从机之间无法通信，任何数据传输都必须通过主机进行。

(3) **地址**：每个 I2C 器件都有自己的地址，以供自身在从机模式下使用。在标准的 I2C 中，从机地址被定义成 7 位（扩展 I2C 允许 10 位地址）。地址 0000000 一般用于发出通用呼叫或总线广播。

(4) **发送器**：发送数据到总线的器件。

(5) **接收器**：从总线接收数据的器件。

(6) SDA (Serial DATA): 串行数据线。

(7) SCL (Serial CLock): 串行时钟线。

9.2 I2C总线工作原理

图9-1 给出一个由MCU作为主机,通过I2C总线带3个从机的单主机I2C总线系统。这是最常用、最典型的I2C总线连接方式。

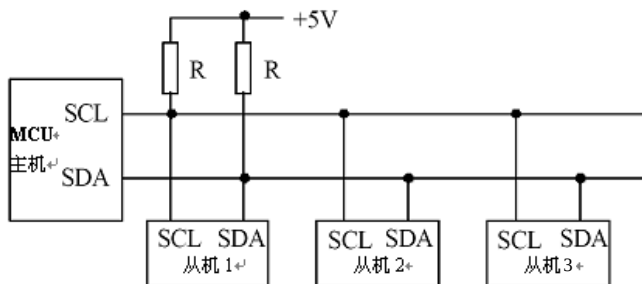


图9-1 I2C总线的典型连接

在物理结构上，I2C系统由一条串行数据线SDA和一条串行时钟线SCL组成。主机按一定的通信协议向从机寻址和进行信息传输。在数据传输时，由主机初始化一次数据传输，主机使数据在SDA线上传输的同时还通过SCL线传输时钟。信息传输的对象和方向以及信息传输的开始和终止均由主机决定。

每个器件都有唯一的地址，且可以是单接收的器件（例如LCD驱动器），或者是可以接收也可以发送的器件（例如存储器）。发送器或接收器可在主或从模式下操作，这取决于芯片是否必须启动数据的传输还是仅仅被寻址。

9.2.1 总线上数据的有效性

I2C总线以串行方式传输数据，从数据字节的最高位开始传送，每个数据位在SCL上都有一个时钟脉冲相对应。在一个时钟周期内，当时钟线高电平时，数据线上必须保持稳定的逻辑电平状态，高电平为数据1，低电平为数据0。当时钟信号为低电平时，才允许数据线上的电平状态变化，如图9-2所示。

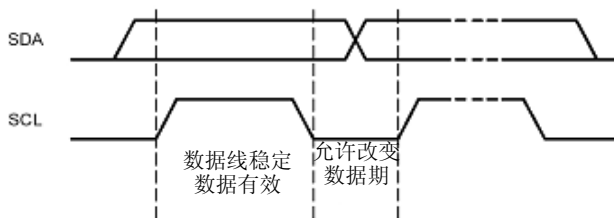


图9-2 I2C总线上数据的有效性

9.2.2 总线上的信号

I2C 总线在传送数据过程中共有四种类型信号，分别是开始信号、停止信号、重新开始信号和应答信号。

开始信号 (START): 如图 9-3 所示。当 SCL 为高电平时，SDA 由高电平向低电平跳变，产生开始信号。当总线空闲的时候（例如没有主动设备在使用总线，即 SDA 和 SCL 都处于高电平），主机通过发送开始信号 (START) 建立通信。

停止信号 (STOP): 如图 9-3 所示。当 SCL 为高电平时，SDA 由低电平向高电平的跳变，产生停止信号。主机通过发送停止信号，结束时钟信号和数据通信。SDA 和 SCL 都将被复位为高电平状态。

重新开始信号 (Repeated START): 在 I2C 总线上，由主机发送一个开始信号启动一次通信后，在首次发送停止信号之前，主机通过发送重新开始信号，可以转换与当前从机的通信模式，或是切换到与另一个从机通信。如图 9-3 所示，当 SCL 为高电平时，SDA 由高电平向低电平跳变，产生重新开始信号，它的本质就是一个开始信号。

应答信号 (A): 接收数据的 IC 在接收到 8 位数据后，向发送数据的 IC 发出的特定的低电平脉冲。每一个数据字节后面都要跟一位应答信号，表示已收到数据。应答信号在

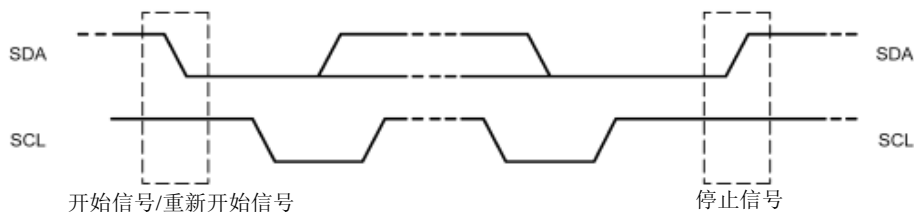


图9-3 开始、重新开始和停止信号

第 9 个时钟周期出现，这时发送器必须在这一个时钟位上释放数据线，由接收设备拉低 SDA 电平来产生应答信号，由接收设备保持 SDA 的高电平来产生非应答信号，如图 9-4 所示。所以一个完整的字节数据传输需要 9 个时钟脉冲。如果从机作为接收方向主机发送非应答信号，这样主机方就认为此次数据传输失败；如果是主机作为接收方，在从机发送器发送完一个字节数据后，发送了非应答信号，从机就认为数据传输结束，并释放 SDA 线。不论是以上哪种情况都会终止数据传输，这时主机或是产生停止信号释放总线，或是产生重新开始信号，从而开始一次新的通信。

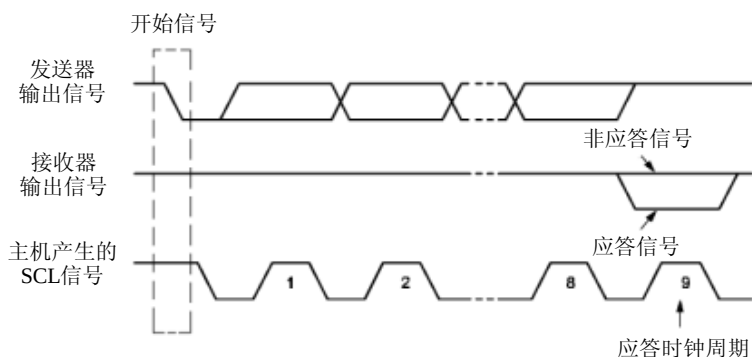


图9-4 I2C总线的应答信号

开始、重新开始和停止信号都是由主控制器产生，应答信号由接收器产生，总线上带有 I2C 总线接口的器件很容易检测到这些信号。但是对于不具备这些硬件接口的 MCU 来说，为了能准确地检测到这些信号，必须保证在 I2C 总线的时钟周期内对数据线至少进行两次采样。

9.2.3 总线上数据传输格式

一般情况下，一个标准的 I2C 通信由四部分组成：开始信号、从机地址传输、数据传输和结束信号。

由主机发送一个开始信号，启动一次 I2C 通信；在主机对从机寻址后，再在总线上传输数据。I2C 总线上传送的每一个字节均为 8 位，首先发送的数据位为最高位，每传送一个字节后都必须跟随一个应答位，每次通信的数据字节数是没有限制的；在全部数据传输结束后，由主机发送停止信号，结束通信。

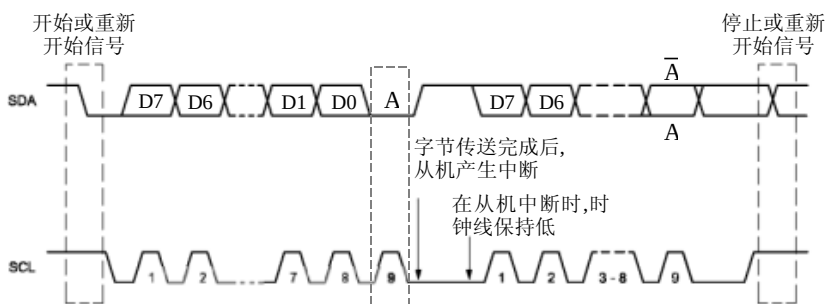


图9-5 I2C总线的数据传输格式

如图 9-5 所示，时钟线为低电平时，数据传输将停止进行。这种情况可以用于当接收器接收到一个字节数据后要进行一些其它工作而无法立即接收下个数据时，迫使总线进入等待状态，直到接收器准备好接收新数据时，接收器再释放时钟线使数据传输得以继续正常进行。例如，当接收器接收完主控制器的一个字节数据后，产生中断信号并进行中断处

理，中断处理完毕才能接收下一个字节数据，这时，接收器在中断处理时将钳住 SCL 为低电平，直到中断处理完毕才释放 SCL。

9.2.4 I2C总线寻址约定

为了消除 I2C 总线系统中主控器与被控器的地址选择线，最大限度地简化总线连接线，I2C 总线采用了独特的寻址约定，规定了起始信号后的第一个字节为寻址字节，用来寻址被控器件，并规定数据传送方向。

在 I2C 总线系统中，寻址字节由被控器的七位地址位（D7~D1 位）和一位方向位（D0 位）组成。方向位为 0 时，表示主控器将数据写入被控器，为 1 时表示主控器从被控器读取数据。主控器发送起始信号后，立即发送寻址字节，这时总线上的所有器件都将寻址字节中的 7 位地址与自己器件地址比较。如果两者相同，则该器件认为被主控器寻址，并发送应答信号，被控器根据数据方向位（R/W）确定自身是作为发送器还是接收器。

主器件作为被控器时，其 7 位从地址在 I2C 总线地址寄存器中设定。而非 MCU 类型的外围器件地址完全由器件类型与引脚电平给定。I2C 总线系统中，没有两个从机的地址是相同的。主控器不应该传输一个和它本身的从地址相同的地址。

9.2.5 主机向从机读写1个字节数据的过程

主机要向从机写 1 个字节数据时，主机首先产生 START 信号，然后紧跟着发送一个从机地址，这个地址共有 7 位，紧接着的第 8 位是数据方向位（R/W），0 表示主机发送数据（写），1 表示主机接收数据（读），这时候主机等待从机的应答信号（ACK），当主机收到应答信号时，发送要访问的地址，继续等待从机的响应信号，当主机收到响应信号时，发送 1 个字节的数据，继续等待从机的响应信号，当主机收到响应信号时，产生停止信号，结束传送过程。如图 9-6 所示。

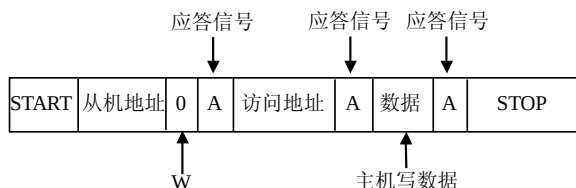


图9-6 主机向从机写数据

当主机要从从机读 1 个字节数据时，主机首先产生 START 信号，然后紧跟着发送一个从机地址，注意此时该地址的第 8 位为 0，表明是向从机写命令，这时候主机等待从机的应答信号（ACK），当主机收到应答信号时，发送要访问的地址，继续等待从机的应答信号，当主机收到应答信号后，主机要改变通信模式（主机将由发送变为接收，从机将由接收变为发送），所以主机发送重新开始信号，然后紧跟着发送一个从机地址，注意此时该地址的第 8 位为 1，表明将主机设置成接收模式开始读取数据，这时主机等待从机的应答信号，当主机收到应答信号时，就可以接收 1 个字节的数据，当接收完成后，主机发送非应答信号，表示不再接收数据，主机进而产生停止信号，结束传送过程。如图 9-7 所示。

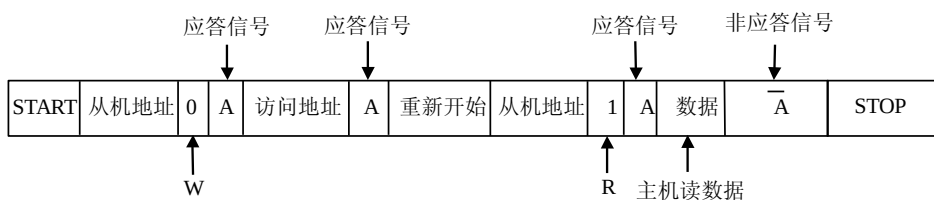


图9-7 主机从机读数据

9.3 I2C模块的编程基础

9.3.1 MCF52233的I2C模块

MCF52233 芯片具有 1 个 I2C 模块接口，I2C 总线以物理方式连接 2 条激活线和 1 条地线。激活线（称为 SDA 和 SDL）是双向的，SDA 是串行数据线，SCL 是串行时钟线。为了不影响 I2C 的灵活性，所有连接到这 2 条信号线的设备必须是漏极开路或集成电路开路输出，这些输出需带有外接上拉电阻。

MCF52233 芯片的 I2C 模块具有以下特点：

- (1) 兼容 I2C 总线标准 2.1 版本；
- (2) 支持 3.3V 设备；
- (3) 允许多主机模式；
- (4) 可软件编程选择 50 个串行时钟频率；
- (5) 软件编程实现 ACK 信号；
- (6) 位与位之间的中断传输方式；
- (7) 仲裁丢失中断的同时，自动从主机模式切换到从机模式；
- (8) 具有从地址识别中断；
- (9) 开始信号和停止信号的产生和检测，并可以重复产生开始信号；
- (10) ACK 信号的产生和检测；
- (11) 总线繁忙检测。

9.3.2 MCF52233的I2C模块寄存器介绍

从程序员的角度看，I2C 模块有 5 个 8 位寄存器，表 9-1 给出了这些寄存器的基本介绍。只要理解和掌握这 5 个寄存器的用法，了解 9.2 节给出的 I2C 总线协议，就可以进行 I2C 模块的编程。

表 9-1 MCF52233 的 I2C 模块相关寄存器

寄存器名称	缩写	IPSBAR偏移	访问权限	基本功能
地址寄存器	I2ADR	0x00_0300	读/写	设置从机地址
分频寄存器	I2FDR	0x00_0304	读/写	设置I2C模块的工作频率等

控制寄存器	I2CR	0x00_0308	读/写	设置传输格式、中断使能
状态寄存器	I2SR	0x00_030C	读/写	表明I2C模块的工作状态
数据I/O寄存器	I2DR	0x00_0310	读/写	收发数据

1. I2C地址寄存器（I2C Address Register, I2ADR）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	

该寄存器任何时候均可读和写,它包括了 I2C 总线将要做出响应的地址,复位后全 0。

D7~D1: ADR7~ADR1, 从机地址。

D0: 保留位。该位是为了将来的兼容性而设置的保留位,读出总是 0。

2. I2C分频寄存器（I2C Frequency Divider Register, I2FDR）

I2C 总线波特率由系统总线频率分频得来,该分频寄存器用于产生相应的分频因子。编程时应根据要与 MCU 通信的从机设备的 I2C 总线频率范围来确定具体的波特率值,并设置该分频寄存器。一般 I2C 总线频率设定在 100kbps 左右,可以通过减少 I2C 总线上的负载,使其总线最大波特率达到 CPU 总线频率的二十分之一。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IC							

I2C 分频寄存器的高 2 位保留,必须清零。I2C 的低 6 位 (IC) 用来定义分频因子 b,从而确定 I2C 时钟速率,复位后全 0。IC 中不同的内容对应着不同的分频因子值,具体的对应关系如表 9-2 所示。

表 9-2 分频因子

IC(Hex)	分频因子	IC(Hex)	分频因子
00	28	20	20
01	30	21	22
02	34	22	24
03	40	23	26
04	44	24	28
05	48	25	32
06	56	26	36
07	68	27	40
08	80	28	48
09	88	29	56
0A	104	2A	64
0B	128	2B	72
0C	144	2C	80
0D	160	2D	96
0E	192	2E	112
0F	240	2F	128
10	288	30	160
11	320	31	192
12	384	32	224
13	480	33	256
14	576	34	320
15	640	35	384

16	768		36	448
17	960		37	512
18	1152		38	640
19	1280		39	768
1A	1536		3A	896
1B	1920		3B	1024
1C	2304		3C	1280
1D	2560		3D	1536
1E	3072		3E	1792
1F	3840		3F	2048

下面举例说明 I2C 总线波特率计算方法。设 MCU 总线频率为 60MHz，要想得到 100kbps 的 I2C 总线波特率，计算过程如下：

先求出分频因子 b，I2C 总线波特率=MCU 总线频率（Hz）/b，将值代入得：

$$100000=60000000/b$$

$$b=600$$

通过查表 9-2 可知，IC=0x15 时，对应的分频因子为 640，与 600 最接近，所以选择 640 作为分频因子，也就是说 IC 设置为 0x15。

3. I2C 控制寄存器（I2C Control Register, I2CR）

I2C 控制寄存器使能 I2C 模块和 I2C 中断，也包含设置为主机或从机的操作位，复位后全 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IEN	IEN	MSTA	MTX	TXAK	RSTA		

D7—IEN 位，I2C 总线使能位。该位控制着整个 I2C 总线模块的软件复位。IEN=0，表示模块被复位，并且不可使用。这是上电复位状态。复位时低端接口被禁止，但是寄存器可以被访问。IEN=1，表示 I2C 总线模块可使用，该位置位后 I2CR 寄存器的其他位才会生效。

D6—IEN 位，I2C 总线中断使能位。IEN=0，表示来自 I2C 总线模块的中断不可使用，注意这并不清除当前的中断标识；IEN=1，表示来自 I2C 总线模块的中断可以使用。如果状态寄存器 I2SR（下面将会给予介绍）中的 IIF 位也设置了，则中断就可以产生。

D5—MSTA 位，主动/从机模式选择位。上拉复位时，该位清 0。如果 IEN 标志置位，将产生一个停止信号。当主机丧失仲裁时，MSTA 将清 0，而且不会产生停止信号。MSTA=0 表示从机模式，当该位由 1 变为 0 时，在总线上将产生一个停止信号，同时工作模式由主动变为被动；MSTA=1 表示主机模式，当该位由 0 变为 1 时，在总线上产生一个开始信号，而且主机模式被选择。

D4—MTX 位，传输/接收模式选择位。该位选择主动或者被动传输方向。如果设备被设置为被动的时候，根据状态寄存器 I2SR 的 SRW 位，可以对该位进行软件设置。在主机模式下，该位应该根据要求的传输模式来设置。因此在作为从机设备寻址过程中，该位将被置 1。MTX=0 表示接收；MTX=1 表示传输。

D3—TXAK 位，传输确认。在数据确认周期内，该位能识别数据线 SDA 的值。只有当 I2C 总线是一个接收者时，写 TXAK 才会有用。TXAK=0 表示在接收完一个字节的数

据后的第9个时钟周期，设备将会向总线发送一个确认信号；TXAK=1表示设备不会向总线发送确认信号。

D2—RSTA位，重开始位。在不丢失仲裁的情况下，试图重新开始。RSTA=0表示没有重新开始信号；RSTA=1表示产生一个重新开始条件。

D1~D0—保留位。该位用以扩展，读总是为0。

4. I2C状态寄存器（I2C Status Register, I2SR）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TCF	IAAS	IBB	IAL		SRW	IIF	RXAK
复位	1	0	0	0	0	0	0	1

D7—TCF位，数据传输位。当一个字节的数据正在传输，该位被置0。在传输1个字节后，总线上产生第9个时钟周期的下降沿，此时该位被置1。TCF=0表示正在传输；TCF=1表示传输完成。

D6—IAAS位，从机标志位。假如IEN被置位，CPU将被中断。然后CPU需要检查SRW位，并设置相应的Tx/Rx模式。当写I2CR的时候将清除该位。IAAS=0表示未编址；IAAS=1则自身编址为从机。

D5—IBB位，总线忙标志位。IBB=0表示总线进入空闲状态；IBB=1表示总线忙。当检测到STOP时，IBB清零；当检测到START时，IBB置位。

D4—IAL位，仲裁丧失位。

仲裁使能丧失位（IBAL）由硬件来置位。在下面的情况下，仲裁使能可能丧失：

- （1）在地址或数据传输周期中，当主机向SDA驱动高电平时，SDA处于低电平。
- （2）在数据接收确认周期中，当主机向SDA驱动高电平时，SDA处于低电平。
- （3）当总线忙时，发起一个开始周期。
- （4）在从机模式下，发起一个重新开始周期。
- （5）当主机不要求停止信号时，出现了一个停止信号。

该位应该通过写1由软件清除，对该位写0没什么意义。

D3—保留位，以备将来使用。对该位的读操作将返回0。

D2—SRW位，从机读/写位。当IAAS被置位，该位表明了主机发送的地址寄存器中的R/W位的对应值。该位只有当I2C总线是从机模式时才有效。SRW=0表示被动方接收，主机写数据到从机方；SRW=1表示被动方发送，主机从从机方读数据。

D1—IIF位，I2C总线中断。在中断处理程序中，通过软件写0清零该位。IIF=0表示没有I2C中断正在挂起；IIF=1表示有中断挂起。

当下面的情况发生时，IIF位置位：

- （1）1字节传输完成（在第9个时钟的下降沿置位）；
- （2）在从机模式下接收到的地址和它自己的地址匹配；
- （3）仲裁丧失。

D0—RXAK位，接收应答位，即在应答周期内的SDA值。RXAK=0表示在完整的8位数据传输完毕后接收应答信号；RXAK=1在第9个时钟周期非应答信号被检测到。

5. I2C I/O数据寄存器（I2C Data I/O Register, I2DR）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	DATA							

该寄存器复位后全 0。在主动发送模式下，向该寄存器写数据将启动数据传输，高字节首先发送。在主动接收模式下，读该寄存器将启动下一个字节数据的接收。在从机模式下，当地址匹配后，可以达到相同的作用。

在主动发送模式下，第一个字节的数据被写到 I2DR，由 I2CR 的 MSTA 位来确定用于地址转换，并且是调用地址（位于 D7~D1）连接要求的 R/W 位（位于 D0）组成。

9.3.3 MCF52233的I2C模块编程

按照构件的设计思想，I2C 总线与从机通信程序主要有以下几个函数：

对外接口函数：

- (1) I2C 模块初始化子函数

```
void I2C_init(uint8 addr)
```

- (2) 从从机读 N 个字节数据函数

```
uint8 I2C_readN(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data[], uint8 N)
```

- (3) 向从机写 N 个字节数据函数

```
uint8 I2C_writeN(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data[], uint8 N)
```

内部函数：

- (1) 从从机读 1 个字节数据函数

```
static uint8 I2C_read1(uint8 DeviceAddr, uint8 AccessAddr, uint8 *Data)
```

- (2) 向从机写 1 个字节数据函数

```
static uint8 I2C_write1(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data)
```

- (3) 设置波特率

```
static uint8 I2C_set_bps(unsigned int bps)
```

1. I2C模块初始化

I2C 相关寄存器地址宏定义在 MCU 头文件 mcf52233.h 中给出，MCF_I2C_I2AR、MCF_I2C_I2CR、MCF_I2C_I2SR 等寄存器名可以直接使用。

对 I2C 进行初始化，一般有以下几个步骤：

- (1) 设置 SDA 和 SCL 引脚作为 I2C 功能。
- (2) 调用 I2C_set_bps 函数设置 I2C 分频寄存器 I2FDR 的 IC，定义 I2C 总线波特率。
- (3) 设置 I2C 地址寄存器 I2ADR，定义 MCU 作为从机时的地址。
- (4) 置位 I2C 控制寄存器 I2CR 的 IEN 位，使能 I2C 总线接口。
- (5) 如果总线正在忙，则发送停止信号给从机。

(6) 设置 I2C 控制寄存器 I2CR，定义主机或从机模式、发送或接收模式、允许或禁止 I2C 中断等。

```

//I2Cinit:I2C模块初始化-----*
//功 能:对I2C模块进行初始化,设置SDA和SCL引脚作为I2C功能, *
//      I2C总线频率:100KHz,使能I2C模块并检查忙标志。 *
//      如果标志为忙,则释放模块。 *
//参 数:addr:从机地址 *
//返 回:无 *
//-----*
void I2C_init(uint8 addr)
{
    uint8 temp;
    MCF_GPIO_DDRAS = 0;
    MCF_GPIO_PASPAR |= ( MCF_GPIO_PASPAR_SDA_SDA
        | MCF_GPIO_PASPAR_SCL_SCL); //使能I2C信号引脚
    I2C_set_bps(100000u1); //设置速度为100k
    MCF_I2C_I2AR = MCF_I2C_I2AR_ADR(addr); //设置地址
    MCF_I2C_I2CR = MCF_I2C_I2CR_IEN | 0; //使能I2C模块
    //如果此位忙,发送STOP信号给从模块
    if( MCF_I2C_I2SR & MCF_I2C_I2SR_IBB)
    //当允许I2C总线模块时,如果I2SR的IBB位为1,
    //则在处理正常的初始化代码前,执行下面{}内的程序,这将确保给从机STOP命令,
    //使其处于空闲状态,好像它重启了。
    {
        MCF_I2C_I2CR = 0; //清控制寄存器
        MCF_I2C_I2CR = MCF_I2C_I2CR_IEN | //使能模块
            MCF_I2C_I2CR_MSTA; //发送start信号
        temp = MCF_I2C_I2DR; //空读
        MCF_I2C_I2SR = 0; //清状态寄存器
        MCF_I2C_I2CR = 0; //清控制寄存器
        MCF_I2C_I2CR = MCF_I2C_I2CR_IEN | 0; //重新使能模块
    }
    MCF_I2C_I2CR = 0b10001000; //不能去掉
        //|||||___不用
        //|||||
        //|||||___RSTA位
        //|||||___不发送应答信号
        //||||___接收模式
        //|||___从机模式
        //||___禁止I2C中断
        //||___使能I2C模块;

    return;
}

```

2. 接收一个字节数据

接收一个字节数据的步骤是:

- (1)置 I2C 控制寄存器 I2CR 的发送模式选择位 MTX 为 1,将 MCU 设为发送模式。
- (2)MCU 初始化之后设为从机模式,这时将 I2C 控制寄存器的从机模式选择位 MSTA 由 0 变为 1,MCU 被设为从机模式并由此产生开始信号。

(3) 发送与 MCU 通信的从机地址（发送字节的高 7 位），并通知从机接收数据（发送字节的最低位清 0）。只需将要发送的字节置于 I2DR，就初始化了此次发送。接着就等待这个字节发送完成。发送成功后，还需等待从机的应答信号。

(4) 发送要访问的具体从机中的寄存器地址，并等待发送完成。发送成功后，需等待从机的应答信号。

(5) 在改变通信模式前，要置 I2C 控制寄存器的 RSTA 位产生重新开始信号。接下来，MCU 要由先前的发送数据（包括控制信号和从机地址等）变为接收数据，从机也要由接收数据变为发送数据。所以此时 MCU 要发送一个重新开始信号。

(6) 发送从机地址字节，此时最低位为 1，通知从机准备发送数据。等待该字节发送完成以及从机的应答信号。

(7) 清 I2C 控制寄存器的 MTX 位为 0，MCU 被设置为接收模式。

(8) 读 I2DR，清除该寄存器，准备接收数据。等待从机发送数据完成，从机发完数据后，MCU 将 I2C 控制寄存器的从机模式选择位 MST 由 1 变为 0，MCU 被设为从机模式并由此产生停止信号，结束一次数据接收过程。

(9) 读 I2DR 寄存器，MCU 最终得到从机发来的一个字节数据。

```
//-----*
//函数名:I2C_read1                                     *
//功能:从从机读1个字节数据                             *
//参数:                                                 *
//    (1) DeviceAddr:设备地址                           *
//    (2) AccessAddr:访问地址                           *
//    (3) Data:带回收到的一个字节数据                   *
//返回:为0:成功读一个字节;为1:读一个字节失败           *
//内部调用:I2C_delay                                     *
//-----*
static uint8 I2C_read1(uint8 DeviceAddr, uint8 AccessAddr, uint8 *Data)
{
    while(MCF_I2C_I2SR & MCF_I2C_I2SR_IBB); //忙则等待
    MCF_I2C_I2CR |= MCF_I2C_I2CR_MTX;      //设置为发送模式
    MCF_I2C_I2CR |= MCF_I2C_I2CR_MSTA;     //产生START信号
    MCF_I2C_I2DR = (uint8)DeviceAddr & 0xfe; //设置设备地址
    I2C_delay();                             //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF)); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2DR = (uint8)AccessAddr;      //发送访问地址
    I2C_delay();                             //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF)); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2CR |= MCF_I2C_I2CR_RSTA; //主机模式下,RSTA位置1,产生重复开始信号
    MCF_I2C_I2DR = DeviceAddr | 0x01;      //通知从机改为发送数据
    I2C_delay();
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF)); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2CR &= ~MCF_I2C_I2CR_MTX;     //TX = 0, MCU设置为接收模式
```

```

    *Data = MCF_I2C_I2DR;           //空读I2DR
    I2C_delay();                     //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF )); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF; //清完成标志位
    MCF_I2C_I2CR &= ~MCF_I2C_I2CR_MSTA; //产生STOP信号
    *Data = MCF_I2C_I2DR;           //读I2DR
    I2C_delay();                     //延时
    return 0;
}

```

3. 发送一个字节数据

发送一个字节数据的步骤是：

(1) - (4) 与接收一个字节数据的前四步一致。

(5) 将要发送给从机的一个字节数据写入 I2DR，初始化数据发送。接着就等待数据发送完成，等待从机的应答信号。收到应答后，表明数据已成功写入从机。

(6) 将 I2C 控制寄存器的从机模式选择位 MSTA 由 1 变为 0，MCU 被设为从机模式并由此产生停止信号，结束一次数据发送过程。

```

//-----*
//函数名:I2C_writel                                     *
//功    能:向从机写1个字节数据                         *
//参    数:                                             *
//    (1) DeviceAddr:设备地址                           *
//    (2) AccessAddr:访问地址                           *
//    (3) Data:要发给从机的1个字节数据                 *
//返    回:为0:成功写一个字节;为1:写一个字节失败     *
//内部调用:I2C_delay                                   *
//-----*
static uint8 I2C_writel(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data)
{
    while(MCF_I2C_I2SR & MCF_I2C_I2SR_IBB); //忙则等待
    MCF_I2C_I2CR |= MCF_I2C_I2CR_MTX;      //设置为发送模式
    MCF_I2C_I2CR |= MCF_I2C_I2CR_MSTA;     //产生START信号
    MCF_I2C_I2DR = (uint8)DeviceAddr;      //设置设备地址
    I2C_delay();                            //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF )); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2DR = (uint8)AccessAddr;      //发送访问地址
    I2C_delay();                            //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF )); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2DR = Data;                   //写数据
    I2C_delay();                            //延时
    while( !(MCF_I2C_I2SR & MCF_I2C_I2SR_IIF )); //直到产生中断才退出
    MCF_I2C_I2SR &= ~MCF_I2C_I2SR_IIF;     //清完成标志位
    MCF_I2C_I2CR &= ~MCF_I2C_I2CR_MSTA;    //产生STOP信号
    I2C_delay();                            //延时
}

```

```

    return 0;
}

```

4. 接收N个数据与发送N个数据

通过循环调用接收一个数据函数与发送一个数据函数，来分别实现接收与发送 N 个字节数据。值得注意的是每一次数据字节的传输之间要有适当的时间间隔。

```

//-----*
//函数名:I2C_readN                                *
//功能:从从机读N个字节数据                        *
//参数:                                             *
//    (1) DeviceAddr:设备地址                      *
//    (2) AccessAddr:访问地址                      *
//    (3) Data:读出数据的缓冲区                    *
//    (4) N:从从机读的字节个数                    *
//返回:为0: 成功读N个字节;为1: 读N个字节失败      *
//内部调用:I2C_read1                               *
//-----*
uint8 I2C_readN(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data[], uint8 N)
{
    uint8 i, j;
    for (i = 0; i < N; i++)
    {
        for(j = 0; j < 15; j++); //最小延时(发送的每个字节之间要有时间间隔)
        if (I2C_read1(DeviceAddr, AccessAddr + i, &Data[i]))
            return 1;           //其中一个字节没有接收到, 返回失败标志:1
    }
    if (i >= N)
        return 0;               //成功接收N个数据, 返回成功标志:0
}
//-----*
//函数名:I2C_writeN                                *
//功能:向从机写N个字节数据                        *
//参数:                                             *
//    (1) DeviceAddr:设备地址                      *
//    (2) AccessAddr:访问地址                      *
//    (3) Data:要写入的数据                        *
//    (4) N:写入数据个数                          *
//返回:为0: 成功写N个字节;为1: 写N个字节失败      *
//内部调用:I2C_writel                               *
//-----*
uint8 I2C_writeN(uint8 DeviceAddr, uint8 AccessAddr, uint8 Data[], uint8 N)
{
    uint8 i, j;
    for (i = 0; i < N; i++)
    {
        for(j = 0; j < 15; j++); //最小延时(发送的每个字节之间要有时间间隔)
        if (I2C_writel(DeviceAddr, AccessAddr + i, Data[i]))

```

```

        return 1;           //其中一个字节没有发送出去, 返回失败标志:1
    }
    if (i >= N)
        return 0;           //成功发送N个数据, 返回成功标志:0
}

```

MCF52233 首先向时钟芯片 PCF8563 写入时间和日期数据, 然后每隔约 10 秒, 读取 PCF8563 的时间并通过串行口发送给 PC 机。此实例中, MCF52233 作为主机, 初始化时, 将其地址设置为 1。PCF8563 作为从机, 其读写地址分别为 0xA3H 和 0xA2H。注意: 这里主机的地址是人为确定的, 只要不与从机地址相冲突即可, 而从机的地址是由器件本身决定的。

9.4 MCF52233的I2C模块的进一步讨论

9.4.1 仲裁程序

I2C 总线是多主机总线, 允许多个主机连接到该总线上。如果 2 个或者 2 个以上主机试图在同一时刻控制总线, 则时钟同步程序使用在每个主设备中集成的硬件定时器来确定总线时钟。因为 SCL 线上实现了线与, 故在 SCL 线上由高到低的跳变将影响所有连接到总线上的设备。设备开始对低电平计数, 一旦一个设备的时钟为低电平, 它将保持 SCL 线处于低电平状态, 直到时钟变为高电平状态。但是, 如果另一设备的时钟仍然为低电平, 则该设备由低变高的变化不能改变 SCL 线的状态。因此, 总线 SCL 电平的低状态由最长低电平周期的设备保持。具有较短低电平周期的设备在这段时间进入高等待状态, 如图 9-8 所示。当所有相关设备计数超过它们的低电平周期时, SCL 总线被释放, 并进入高电平状态。此时, 设备时钟和 SCL 线是同步的, 设备开始对高电平周期计数。第一个完成高电平周期计数的设备将把 SCL 线再次拉低。时钟信号低电平时间与主机中最长的低电平时间相等, 时钟信号高电平时间与主机中最短高电平时间相等。

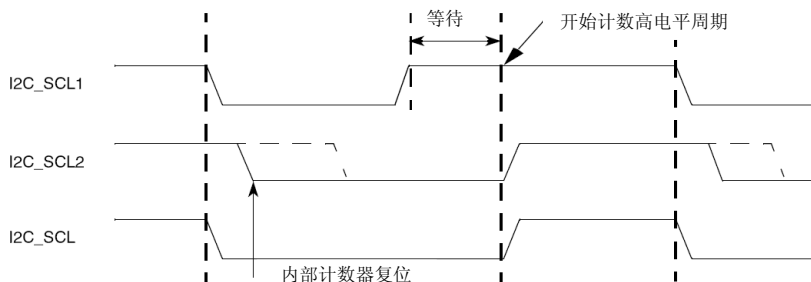


图9-8 时钟同步

由数据仲裁程序决定竞争的主设备的优先级。当另一个主设备发送逻辑 0, 而这个总线主设备发送 1 时, 该主设备将仲裁失败。仲裁失败的主设备立即转换为从设备接收模式, 并停止驱动 SDA 输出, 如图 9-9。在这种情况下, 由主设备到从设备的转变, 不产生 STOP 条件。在 ColdFire 设备中, 仲裁失败由置 I2SR 的 IAL 位来表示。

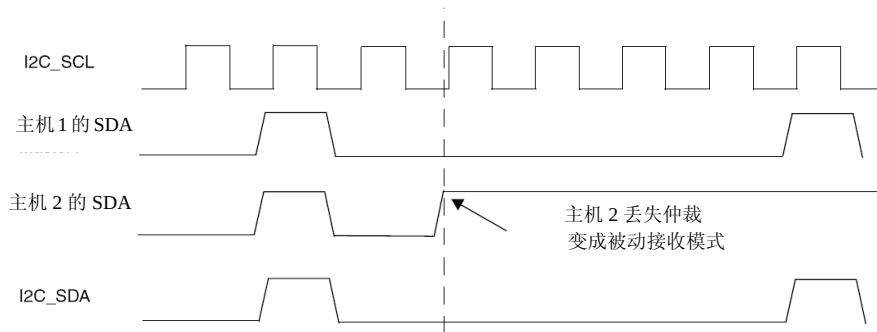


图9-9 仲裁程序

9.4.2 实现数据传输同步交换

时钟同步机制可用作数据传输的同步交换（Handshake）。在一个字节传输完成后，从设备能够保持 SCL 低电平状态。在这种情况下，时钟机制暂停总线时钟，并且强制主设备时钟进入等待状态，直到从设备释放 SCL。从设备可能会降低数据传输率。在主设备驱动 SCL 到低电平后，从设备能够将 SCL 的低电平状态保持一个需要的周期，然后释放它。如果从设备 SCL 低电平周期比主设备的低电平周期长，则 SCL 总线信号低电平周期将被拉长。

第10章 Flash在线编程

本章主要以 32 位微控制器 ColdFire 系列中的 MCF52233 为例阐述 Flash 存储器的编程方法，ColdFire 系列中的其它芯片的编程方法与此类似。同时讨论了 MCF52233 Flash 存储器的保护和加密特性，并给出实例。

10.1 Flash存储器概述

1. Flash存储器的特性

理想的存储器应该具备存取速度快、不易失、存储密度高（单位体积存储容量大）、价格低等特点。一般的存储器具有这些特点中的一个或几个。近几年 Flash 存储器技术趋于成熟，它是目前比较理想的存储器。闪速存储器具有电可擦除、无需后备电源来保护数据、可在线编程、存储密度高、低功耗、成本较低等特点。而这些特点，正是 MCU 所期望的，但早期引入片内 Flash 存储器技术的 MCU，在可靠性和稳定性方面仍有一些不足。随着 Flash 技术的成熟，目前，许多 MCU 内部集成了 Flash 存储器，它是 MCU 的重要组成部分。Flash 存储器的发展对单片机的发展起到了极大的推动作用。

Flash 存储器是一种高密度、真正不挥发的高性能读/写存储器，兼有功耗低、可靠性高等优点。与传统的固态存储器工艺相比，Flash 存储器的主要特点如下：

（1）固有不挥发性。这一特点与磁存储器相似，Flash 存储器不需要后备电源来保持数据。所以，它具有磁存储器无需电源保持数据的优点。

（2）易更新性。Flash 存储器具有电可擦除特点。相对于 EPROM（电可编程只读存储器）的紫外线擦除工艺，Flash 存储器的电擦除功能为开发者节省了时间，也为最终用户更新存储器内容提供了可能。

（3）成本低、密度高、可靠性好。Flash 存储器与传统的 EEPROM（电可擦除可编程只读存储器，也写为 E2PROM）相比较，Flash 存储器的成本更低、密度更高、可靠性更好。

2. Flash存储器编程的基本概念

在许多嵌入式产品开发中，需要使用掉电仍能保持数据的存储器来保存一些参数或重要数据，以前一般使用 EEPROM 来实现。ColdFire 系列单片机提供了对 Flash 存储器在用户模式下的在线编程功能，可以使用 Flash 存储器来实现 EEPROM 的功能，这样简化了电路设计，也节约了成本。

然而，对 Flash 存储器的读写不同于一般的 RAM 读写，需要专门的编程过程，Flash 编程的基本操作有两种：擦除（Erase）和写入（Program）。擦除操作的含义是将存储单元的内容由二进制的 0 变成 1，而写入操作的含义，是将存储单元的内容由二进制的 1 变成 0。擦除操作包括整体擦除和页擦除。而写入操作是以字为单位进行的。在擦除及写入

过程中一般需要高于电源的电压。ColdFire 系列单片机由片内电荷泵产生这个高压。另外,在执行写入操作之前,要确保该写入区在上一次擦除后没有被写过,即写入区是空白的(存储单元内容为\$FF)。所以,在写入之前一般都要先执行擦除操作。

10.2 ColdFire Flash 存储器编程方法

ColdFire Flash 模块 (ColdFire Flash Module, CFM) 由 4 个阵列组成,每个阵列由 32K*16 位组成,从而构成 256KB 的 Flash 存储区,如图 10-1。内部的 Flash 存储控制器需要两个周期访问 Flash,但它支持交叉访问,因此连续读 Flash 的效率更高,读每个字只要一个周期。

MCF52233 中的 256K 字节 Flash 空间,被分为 32 个 8K 大小的扇区 (Sector),每个扇区分为 4 个页 (Page),每页为 2K 字节。在对 Flash 进行编程的过程中,擦除以页为单位,也就是说,若要擦除则要至少擦除一页,写入是以双字 (4 个字节) 为单位。进行 Flash 程序编写必须注意这个特点。

因为 MCF52233 是 32 位的控制器,所以有 32 根地址线,可以寻址 4G 的地址空间。原则上,MCF52233 的 Flash 的起始地址是可以改变的,通过设置相应的寄存器可将 256K 字节的 32 位 Flash 定位到任意连续的空间。但是,实际使用时,一般将 Flash 的起始地址设置为 0x0000_0000,建议不要随意改动^①。

^① 由于中断向量表在 0x0000_0000 开始的地址,因此 Flash 的起始地址不宜定位在其它地址。

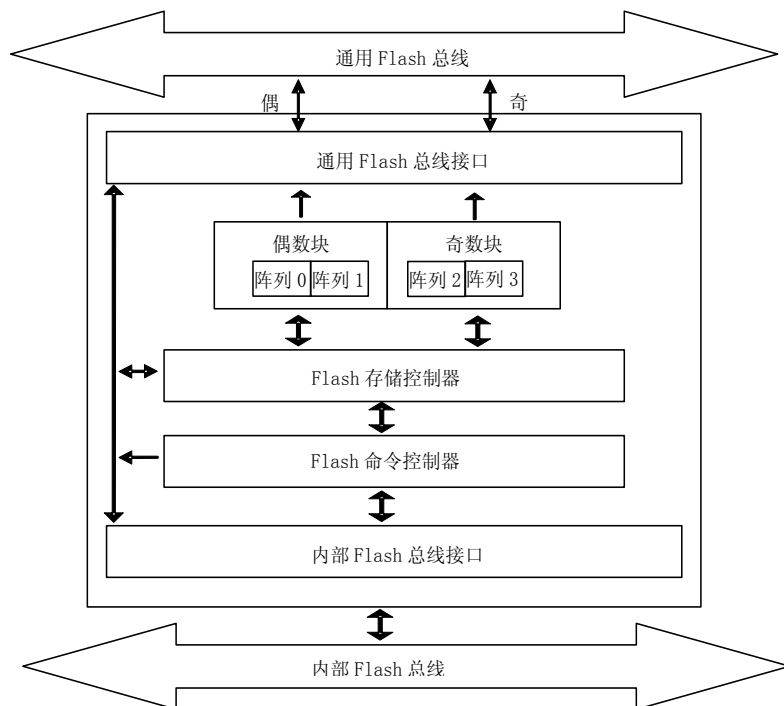


图10-1 ColdFire Flash 模块框架图

10.2.1 ColdFire Flash模块寄存器

擦除和写入操作主要用到的寄存器有：Flash 基址寄存器（FLASHBAR）、时钟分频寄存器（CFMCLKD）、模块配置寄存器（CFMMCRC）、保护寄存器（CFMPROT）、安全寄存器（CFMSEC）、用户状态寄存器（CFMUSTAT）和命令寄存器（CFMCMD）。

1. FLASH基址寄存器FLASHBAR（Flash Base Address Register）

FLASHBAR 定义 Flash 的基址、寄存器内容是否有效、权限访问。

FLASHBAR 的 19 -31 位定义 Flash 模块的基址，通过设置此寄存器可以把 Flash 存储器映射到地址空间任意的 256K 的空间。FLASHBAR 的 0 位定义 Flash 基址寄存器的内容是否有效，1 为有效 0 为无效。在 ColdFire 的初始化阶段，一般要设置此寄存器使得 Flash 存储器映射到某地址处。擦写 Flash 存储器时，擦除写入的地址要加上 0x44000000^①，这一点尤其要注意，通过后面的程序例子可以看到这一点。

2. CFM时钟分频寄存器CFMCLKD（CFM Clock Divider Register）

CFMCLKD 寄存器主要用于对擦除与写入时间的控制。在擦写之前，必须设置该寄存器，使得 Flash 内部工作时钟（F_{CLK}）在 150KHz 到 200KHz 之间，设置完成后，DIVLD

^① 这是技术手册要求的。

位自动置 1，若 DIVLD 位为 0 则不允许进行擦写操作。如果 Flash 内部工作时钟小于 150KHz 可能会由于所加的高压损坏 Flash 存储器，而大于 200KHz 可能导致擦写不完整。该寄存器复位全为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	DIVLD	PRDIV8	DIV					

D7—DIVLD: 时钟分频加载位 (Clock Divider Loaded)。DIVLD 为只读位。DIVLD=1，表示在复位后 CFMCLKD 已经被改写过，可以对 Flash 进行擦写操作；DIVLD=0，表示在复位后 CFMCLKD 没有被改写过，不可以对 Flash 进行擦写操作。

D6—PRDIV8: Flash 预分频使能位 (Enable Prescaler by 8)。PRDIV8=1，表示 Flash 分频器的时钟为内部 Flash 时钟的 1/8；PRDIV8=0，表示 Flash 分频器的时钟为内部 Flash 时钟。

D5~D0: 时钟分频位 (Clock Divider Bit)。Flash 时钟分频器的分频因子。Flash 的内部工作时钟 F_{CLK} 的计算方法为：

$$F_{CLK} = \text{Flash 分频器的时钟} / (DIV + 1), (150\text{KHz} < F_{CLK} \leq 200\text{KHz}).$$

注意以下几点：

- (1) 内部 Flash 时钟为总线时钟的一半。
- (2) 当输入时钟大于 12.8MHz 时，需将 PRDIV8 置 1，否则，将 PRDIV8 清 0。
- (3) Flash 分频器时钟的单位为 KHz，实际使用时应保证 F_{CLK} 在 180KHz 左右。

在对 Flash 进行编程操作时，Flash 的内部工作时钟必须降到 150KHz~200KHz，擦/写操作的脉冲是 Flash 的内部工作时钟的一个时钟周期，所以擦/写的时间相应地在 6.7 μ s~5 μ s。

3. CFM模块配置寄存器CFMMCR (CFM Module Configuration Register)

CFMMCR 寄存器用于配置和控制内部总线接口操作。复位全为 0。

数据位	D15~D11	D10	D9	D8	D7	D6	D5	D4~D0
定义		LOCK	PVIE	AEIE	CBEIE	CCIE	KEYACC	

D15~D11: 未定义。

D10—LOCK: 写锁定控制位 (Write Lock Control)。LOCK=1，保护寄存器 CFMPROT、管理员访问寄存器 CMFSACC 和数据访问寄存器 CFMDACC 均不可写；LOCK=0，上述寄存器可以写。

D9—PVIE: 保护违例中断使能位 (Protection Violation Interrupt Enable)。PVIE=1，当侵害保护标志位 PVIOL 置位的时候会发生中断；PVIE=0，不会发生中断。

D8—AEIE: 访问错误中断使能位 (Access Error Interrupt Enable)。AEIE=1，当访问出错标志位 ACCERR 置位的时候会发生中断；AEIE=0，不会发生中断。

D7—CBEIE: 命令缓冲区空中断允许位 (Command Buffer Empty Interrupt Enable)。CBEIE=1，当 Flash 命令缓冲区空中断标志位 CBEIF 为 1 时，则发生中断；CBEIE=0，命令缓冲区空中断禁止。

D6—CCIE: 命令完成中断允许位 (Command Complete Interrupt Enable)。CCIE=1，当 Flash 命令完成中断标志位 CCIF 为 1 时，则发生中断；CCIE=0，命令完成中断禁止。

D5—KEYACC: 密码写入允许位 (Enable Security Key Writing)。KEYACC=1, 允许密码匹配 (对 Flash 的写操作实际是进行密码匹配, 不改变 Flash 的内容); KEYACC=0, 禁止密码匹配 (在进行 Flash 写操作时, 须令 KEYACC=0)。

D4~D0: 未定义。

4. CFM保护寄存器CFMPROT (CFM Protection Register)

该 32 位的寄存器每一位控制着 MCF52233 的一个扇区, 此寄存器只有在 LOCK 位为零的时候才能够设置, 相应位为 1 表示相应的扇区被保护, 不可以擦除写入; 为 0 则表示相应的扇区未被保护。复位后该寄存器的内容从 Flash 地址 0x0408~0x040B 处加载。

5. CFM安全寄存器CFMSEC (CFM Security Register)

该 32 位的寄存器用于存储 Flash 的保护字和保护状态。定义为:

数据位	D31	D30	D29~D16	D15~D0
定义	KEYEN	SECSTAT		SEC

D31—KEYEN: 后门机制开启位 (Enable Backdoor Key access to unlock security)。KEYEN=1, 开启后门机制; KEYEN=0, 关闭后门机制。

D30—SECSTAT: Flash 安全状态。SECSTAT=1, 已使能安全机制; SECSTAT=0, 未使能安全机制。

D29~D16: 未定义。

D15~D0—SEC: Flash 安全字节。若 SEC=0x4AC8, Flash 已经加密; SEC=其他值, Flash 没有加密。

该寄存器是只读寄存器, 复位后它的内容从 Flash 地址 0x0414~0x0417 处加载。

6. CFM用户状态寄存器CFMUSTAT (CFM User Status Register)

CFMUSTAT 寄存器定义了 Flash 命令控制器、Flash 存储访问、保护及空白检测等状态。复位后 D7~D6 位为 1, 其余位为 0。定义为:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	CBEIF	CCIF	PVIOL	ACCERR		BLANK		

D7—CBEIF: Flash 命令缓冲区空中断标志位 (Flash Command Buffer Empty Interrupt Flag)。CBEIF=1, 表示命令缓冲区可以接收一个新的命令; CBEIF=0, 表示命令缓冲区非空, 不能接收新的命令。

D6—CCIF: Flash 命令完成中断标志位 (Flash Command Complete Interrupt Flag)。CCIF=1, 表示所有命令已经执行完毕; CCIF=0, 表示有命令正在执行中。

D5—PVIOL: 侵害保护标志位 (Protection Violation Flag)。当擦写命令试图访问保护区域时, PVIOL 位自动置 1, 写 1 清零该位。PVIOL=1, 检测到侵害; PVIOL=0, 表示未检测到侵害。

D3: 未定义。

D4—ACCERR: 访问出错标志位 (Access Error Flag)。当擦写操作序列没有正常完成时, 该位自动置 1。ACCERR=1, 表示一个访问错误已经发生。

D2—BLANK: Flash 空白标志位 (Flash Verified All Blank Flag)。如果 Flash 没有被写

入任何数据，在执行一个空白检测命令后，该标志位置 1。在执行一个空白检测命令后，BLANK=1，表示 Flash 是空的，即所有的字节为 0xFF；BLANK=0，表示 Flash 非空。

D1~D0：未定义。

7. Flash命令寄存器CFMCMD（CFM Command Register）

Flash 命令寄存器定义了 Flash 操作的相关命令，复位全为 0。定义为：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读								
	CMD							

D7：未定义。

D6~D0—CMD：存储擦除、写入等命令。向 CFMCMD 寄存器写入 0x05、0x06、0x20、0x40、0x41 分别表示空白检测命令、页擦除校验、写一个字命令、页擦除命令、整体擦除命令。如果向 CFMCMD 寄存器写入一个除上述 5 个数据以外的其他字节，将产生访问错误，此时会自动将 CFMUSTAT 寄存器的 ACCERR 置 1。

除了上述 7 个寄存器外，与 Flash 模块相关的寄存器还有保护寄存器 CFMPROT、管理员访问寄存器 CMFSACC 和数据访问寄存器 CFMDACC。若要了解它们的定义，可查阅数据手册。

10.2.2 ColdFire Flash存储器擦除与写入的步骤

对于 ColdFire 控制器来说，对 Flash 存储器的擦除操作可以进行整体擦除也可以仅擦除某一起始地址开始的一个页（2K）。也就是说，不能仅擦除某一字节或一次擦除小于 2K 字节。对 Flash 存储器的写入，必须将一组数据准备好后，放入 RAM 区，擦除 Flash 存储器的相应区域后再进行写入。我们要考虑到对 Flash 存储器的某一字节擦除与写入会影响其所在的一页，所以，在进行擦除与写入操作之前，通过 link 文件把与擦除区域相关的数据安排好十分必要，即不能擦除程序所在的页。

下面就详细分析 ColdFire Flash 擦除与写入的具体步骤，并给出相应的子程序实例。

1. 擦除和写入流程的一些公共操作

（1）判断是否设置时钟分频，若未分频，则先分频。时钟分频寄存器 CFMCLKD 的 DIVLD 位可以反映是否设置时钟分频，该位为 1，表示已分频；反之，则表示未分频。应该在擦除或写入前进行分频，否则擦除或写入操作将不被允许。

（2）清除用户状态寄存器 CFMUSTAT 的保护错误标志位 PVIOL 和访问错误标志位 ACCERR，以免前一次的操作结果干扰本次操作。Flash 擦/写初始化操作函数如下：

```
//-----
- - - - - *
//函数名: Flash_Init *
//功 能: Flash擦/写初始化操作, 设置分频寄存器、禁止保护、清除错误标志 *
//参 数: busclk: 总线频率(单位Khz) (要求>200Khz) *
//返 回: 无 *
//说 明: 请注意入口参数busclk(总线频率)的单位是Khz *
//-----
```



```

- - - - - *
static void Flash_Init(uint32 busclk) //Flash擦/写初始化操作
{
    uint8 div8;           //预分频因子变量
    uint32 fclk_val;       //Flash需要的时钟
    div8 = 0;
    fclk_val = busclk/2;   //得到Flash模块的内部时钟
    //1. 判断是否已经成功设置分频
    if(1 != BGET(DIVLD, MCF_CFM_CFMCLKD)) //未设置, 进行设置
    {
        //2. 初始化Flash分频寄存器, 计算的公式在手册中有描述
        if (fclk_val >= 12800)
        {
            fclk_val = fclk_val>>3;
            div8 = 1;
        }
        fclk_val = fclk_val/180;
        MCF_CFM_CFMCLKD |= fclk_val|(div8<<6);
    }
    MCF_CFM_CFMUSTAT |= (PVIOL|ACCERR); //清除错误标志
}
- - - - -

```

2. Flash的擦除

下面以擦除 Flash 的一个扇区为例, 分析 Flash 擦除操作的具体步骤。

(1) 设置时钟分频, 清相关错误标志位, 这些操作已作为擦除和写入流程开始部分的公共操作被封装成子程序 `Flash_Init`, 擦除时直接调用即可。

(2) 定位要擦除的页。向要擦除的页中任意地址写一任意值。

(3) 向命令寄存器 `CFMCMD` 写页擦除命令 `0x40` (Flash 操作的相关命令详见 10.2.1 节 `CFMCMD` 寄存器部分)。

(4) 向状态寄存器 `CFMUSTAT` 的命令缓冲区空中断标志位 `CBEIF` 写 1, 以清零该位来启动命令执行。

(5) 等待直到命令执行完成, 状态寄存器 `CFMUSTAT` 的命令完成中断标志位 `CCIF` 为 1 标志命令执行完成。

```

//-----
- - - - - *
//函数名: Flash_Page_Erase                                *
//功 能: 擦除指定的页                                    *
//参 数: page : 要擦除的页号(对MCF52233/MCF52223的256KB来说, page=0~127) *
//返 回: '1'=擦除成功, '2'=擦除失败                      *
//内部调用: Flash_Protect, Flash_Init                      *
//说 明: 在第8步骤前最好关总中断, 做完第9步骤后再开中断 *
//-----
- - - - - *

uint8 Flash_Page_Erase(uint8 page) //擦除指定的页
{
- - - - -

```

```

uint32 *addr;
uint16 i;

//1. 计算页号对应的地址
if(page>127) return '2';
addr = (uint32*)(page*2048+PROG_FLASH_OFFSET);
//2. 解除页所在扇区的保护
Flash_Protect(page, 0);
//3. 将flash总线时钟降至150khz~200khz之间, 实际得到178.6KHZ
Flash_Init(SYSTEM_CLOCK*1000); //总线时钟SYSTEM_CLOCK

在sysinit.h中定义
//4. 等待命令缓冲区空
while(!BGET(CBEIF, FUSTAT));
//5. 清除访问和保护违反标志
if(BGET(ACCERR, FUSTAT) || BGET(PVIOL, FUSTAT))
{
    BSET(ACCERR, FUSTAT);
    BSET(PVIOL, FUSTAT);
}
//以下6-9步, Flash被加高电压, 实践表明, 要擦除的页与该段程序不在同一
//逻辑页, 可以稳定擦除, 否则需要移入RAM中执行. 但此时不允许中断
//6. 向要擦除页首地址写入任意值
*addr = 0x00000000;
//7. 向命令寄存器写入页擦除命令
FCMD = PAGE_ERASE;
//8. 载入命令并执行
BSET(CBEIF, FUSTAT);
//9. 等待命令执行完成
while(!BGET(CCIF, FUSTAT));
//10. 对所有扇区加保护
MCF_CFM_CFMPROT = 0xffffffff;
//11. 判断是否全空
for(i=0; i<128; i++)
{
    if((*((uint32*)(page*2048+i*4))!=0xffffffff)
        return '2';
}
return '1';
}

```

3. Flash的写入

下面以向 Flash 一页写入若干字为例, 分析 Flash 写入操作的具体步骤。注意, 应该向干净的区域 (即没有被写过的区域) 写入。

(1) 设置时钟分频, 清相关错误标志位, 这些操作已作为擦除和写入流程开始部分的公共操作被封装成子程序 `Flash_Init`, 直接调用即可。

(2) 向指定地址写若干字数据。

首先, 将源地址的一个字写入目标地址。

其次，向命令寄存器 CFMCMD 载入命令 0x20（Flash 操作的相关命令详见 10.2.1 节 CFMCMD 寄存器部分）。

第三，向状态寄存器 CFMUSTAT 的命令缓冲区空中断标志位 CBEIF 写 1，以清零该位来启动命令执行。

第四，等待直到上一条命令执行完毕，即状态寄存器 CFMUSTAT 的 CBEIF 位为 1，这时就可以接收下一条命令。

最后，如果尚未写完数据，递增源地址和目标地址，转（2）。

注意：由于写入前一般要先进行擦除，而时钟分频寄存器 CFMCLKD 只需要设置一次，所以在写入操作时可以不进行时钟分频寄存器 CFMCLKD 的设置。另外，在擦写的时候，不可以对存放擦写代码的区域进行擦写。

```

//-----
-                                     -                                     *
/ / 函 数 名 :   F l a s h _ P a g e _ W r i t e                                     *
                                     *
//功 能: 从地址addr_source写入N个字节到第page页的地址pstart*4处
                                     *
/           /           参                                     数           :
                                     *
//      ( 1 )   p a g e       :   待 写 入 的 页 号 ( 有 效 范 围 0 ~ 1 2 7 )
                                     *
/ /      ( 2 )   p s t a r t   :   页 内 开 始 地 址
                                     *
/ /      ( 3 )   a d d r _ s o u r c e   :   源 地 址
                                     *
/ /      ( 4 )   N       :   写 入 的 字 节 的 长 度
                                     *
/ / 返      回   :           ' 1 ' 成 功 , ' 2 ' 失 败
                                     *
/ / 内 部 调 用   :   F l a s h _ I n i t
                                     *
//说 明: 由于ColdFire写入是以字(4个字节)为单位, 不是4对齐时, 程序自动填充
0                                     x                                     f                                     f                                     *
//-----
-                                     -                                     *

uint8 Flash_Page_Write(uint8 page,uint16 pstart,uint32 *addr_source,uint16 N)
{
    uint8 *tmpadd;
    uint32 *flashadd;
    uint32 add,*addr_destination,i,j;
    uint16 lenth;

    //1. 计算页号对应的地址
    if(page>127) return '2';
    add = page*2048+pstart*4;
    addr_destination = (uint32*)(add+PROG_FLASH_OFFSET);

```

```

//2. 解除页所在扇区的保护
Flash_Protect(page, 0);
//3. 将flash总线时钟降至150khz~200khz, 得到178.6KHZ
Flash_Init(SYSTEM_CLOCK*1000);
//4. 等待命令缓冲区空
while(!BGET(CBEIF, FUSTAT));
//5. 清除访问和保护违反标志
if (BGET(ACCERR, FUSTAT) || BGET(PVIOL, FUSTAT))
{
    BSET(ACCERR, FUSTAT);
                                                                    BSET(PVIOL, FUSTAT);
}
//6. 向目标地址写入数据
DisableInterrupts();
lenth = N/4;           //将待写入的数据按照4对齐
lenth = lenth*4;
tmpadd = (uint8*) (uint32)(addr_source+lenth/4);
if(lenth < N)
{
    for(i=0; i<lenth+4-N; i++) //填充0xff
        tmpadd[3-i] = 0xff;
    if(N > lenth)
        N = lenth + 4;
}

for(i=0; i<N/4; i++)
{
    //6.1 写数据
    *(addr_destination+i) = *(addr_source+i);
    //6.2 写命令寄存器
    FCMD = PROGRAM;
    //6.3 载入命令并执行
    BSET(CBEIF, FUSTAT);

    if (BGET(PVIOL, FUSTAT))
        BSET(PVIOL, FUSTAT);
    while(!BGET(CBEIF, FUSTAT));
}
while(!BGET(CCIF, FUSTAT));

EnableInterrupts();
//7. 对所有扇区加保护
MCF_CFM_CFMPROT = 0xffffffff;
//8. 校验
flashadd = (uint32 *)add;
for(i=0; i<N/4; i++)
{
    if(addr_source[i] != flashadd[i])
    {

```

```

        return '2';
    }
}
return '1';
}

```

4. Flash命令出错的几种可能情况

(1) 在向 Flash 地址写入数据前没有进行 CFMCLKD 寄存器设置; CBEIF 不为 1 时写 Flash; 写 Flash 块的数据长度不是 32 位; 写完 Flash 偶数块后不是向奇数块写数据而是向 Flash 的其他块写入数据; 向 CFMCMMD 中写入规定的 5 种命令以外的命令字; 向 Flash 任意地址写一个值后没有写 CFMCMMD 寄存器; 在执行完 CFMCMMD 命令前, 再次写 CFMCMMD; 写 CFMCMMD 后不将 CFMUSTAT 的 CBEIF 清零; 进入 STOP 模式时有命令未处理完。向 Flash 写入值后或向 CFMCMMD 写完命令但在载入命令前向 CBEIF 写 0 以取消写操作。

(2) 向 Flash 保护区进行擦除或写入; 有扇区保护时执行整体擦除都会将 PVIOL 位置 1。访问 Flash 逻辑块命令未完成时试图访问另一个逻辑块会将 CFMUSTAT 的 ACCERR 位置 1。

以上出错信息仅供参考, 在实际编写擦除和写入程序时, 为了简化编程, 我们可以只考虑两种情况: 成功和失败, 并不详细判定是哪种错误。

10.3 ColdFire Flash擦写函数的测试工程

网上光盘 Flash 测试工程的功能是: MCU 通过串行口接收 PC 机发送的预定格式的命令, 实现对 Flash 进行擦除、写入以及读取操作。PC 端软件可使用串口调试工具, 也可使用光盘中该章的自编测试程序。运行界面如图 10-2 所示。

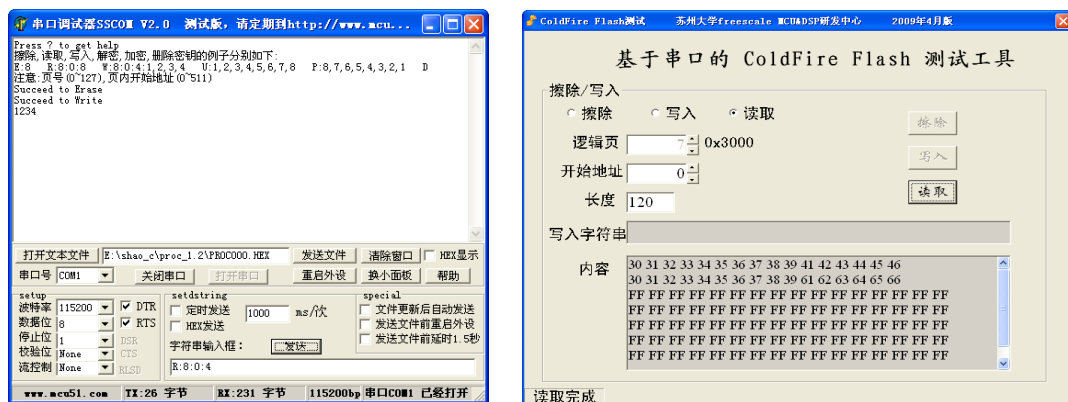


图 10-2 测试 Flash 的程序运行界面

下面给出使用串口调试工具操作 Flash 的命令格式:

命令
?

功能
MCU 向 PC 发送一些使用说明

E:8	擦除第 8 页
R:8:0:4	从第 8 页第 0 个字处读取 4 个字节的内容
W:8:0:4:A, C, B, D	向第 8 页第 0 个字位置写 4 个字节的数据 ACBD
P:8, 7, 6, 5, 4, 3, 2, 1	对 Flash 加密, 密码为字符串"87654321"
D	删除密码

上述测试例程只给出写入数据个数在一页（2048 字节）之内，网上光盘 Flash 目录中还给出了从 PC 机写超过一页数据的工程实例，以便实际应用中参考。

10.4 ColdFire Flash存储器的保护特性和安全性

10.4.1 ColdFire Flash存储器的保护特性

由于 Flash 是非易失性存储器，所以编程人员一般会将一些重要参数或数据存于 Flash。为了防止对这些重要数据区的误擦写，ColdFire 芯片提供了对 Flash 的保护机制。如果一个 Flash 区域被保护，那么是不能对这块区域进行任何擦写的。

与 ColdFire Flash 存储器保护特性相关的寄存器是 Flash 保护寄存器 CFMPROT，该寄存器在 10.2.1 介绍过，MCF52233 分为 32 个扇区，该 32 位寄存器的每一位控制这 32 个扇区的保护情况，如果为 1 相应的扇区就处于保护状态，擦除写入这个扇区会引起保护违例错误。在进行擦写前先要解除该扇区的保护，擦写完成后再恢复保护，在擦除子程序 Flash_Page_Erase 和写入子程序 Flash_Page_Write 中，语句 Flash_Protect(page,FALSE)解除了 page 所在扇区的保护，而语句 MCF_CFM_CFMROT = 0xffffffff 实现了恢复保护的功能。

10.4.2 ColdFire Flash存储器的安全性

ColdFire 的 0x0400~0x0417 为 Flash 的安全配置域，复位后，MCU 自动将该配置域的值载入到相应的寄存器。若 vectors.s 文件中 CFMSEC 标号处（地址为 0x0414~0x0417）的后 2 字节为 0x4ac8，表明芯片处于加密状态，从而防止随意读取 Flash 中的程序。CFMSEC 的高 2 字节为 0xc000，表明允许进行密码匹配。处于加密状态的芯片，通过《32 位 ColdFire 写入器》的 BDM 模式不能擦写。若将写入器设置为 JTAG 模式，可解除密码，即擦除第 0 页（程序将不能再运行），利用写入器的 BDM 模式可以重新进行擦写操作。

1. 设置MCU为保密状态

为了防止 Flash 中的程序被非法读出，就要将 MCU 设置为保密状态，下面提供两种加密 Flash 方法。

方法一：通过修改文件 vectors.s 文件中的安全配置域实现加密。例如，若要设置密码 0x3132333435363738（分两个字，地址 0x4000~0x4003 中为 0x31323334，地址

0x4004~0x4007 中为 0x35363738), 该密码即为字符串“12345678”。那么 vectors.s 文件中的安全配置域内容设置如下:

```
KEY_UPPER: .long 0x31323334 //Flash密码区, 密码高4字节 (KEY_UPPER地址0x0400)
KEY_LOWER: .long 0x35363738 //Flash密码区, 密码低4字节 (KEY_LOWER地址0x0404)
CFMPROT:   .long 0x00000000 //Flash扇区保护设置
CFMSACC:   .long 0x00000000 //Flash管理员访问字
CFMDACC:   .long 0x00000000 //Flash数据访问字
CFMSEC:    .long 0xc0004ac8 //Flash安全状态字, 低位为4ac8时芯片处于加密状态
```

这样, 当程序写入 Flash 并复位后, Flash 就处于加密状态了。如果无须加密 Flash, 则只需将 vectors.s 文件中 CFMSEC 的低 2 字节设置为除 0x4ac8 之外的任何值。

这里给出的加密方法是在程序编译写入之前设置完成, 一般在程序定版时进行。下面给出通过驻留程序提供在程序运行过程中进行动态加密的方法。

方法二: 在程序运行中, 通过调用自定义函数 Flash_Secure 修改 Flash 相关地址处的内容来加密 Flash。如 10.2 节所述, CFMSEC 寄存器的所有位都与设备的安全性相关, 并且这些位是只读的。复位时, 该寄存器从 Flash 的 0x0414 地址处自动载入值。当位 SECSTAT=1 时开启加密机制, 当 CFMSEC 寄存器的安全设置域 SEC 为 0x4ac8 时 Flash 处于加密状态。因此要将 Flash 加密, 需向地址 0x0414 处 (vectors.s 文件中的 CFMSEC 标号) 写入 0xc000 4ac8。加密子程序代码如下:

```
//-----
- - - - - *
//函数名: Flash_Secure *
//功 能: 加密Flash *
//参 数: key[2]:8个字节密码 *
//返 回: '1' 成功; '2' 失败 *
//内部调用: Page_Erase,Flash_Page_Write *
//-----
- - - - - *

uint8 Flash_Secure(uint32 key[2]) //加密Flash
{
    uint32
    vecdata[512], addr, data;

    uint16 i;
    uint8 state;
    //1. 首先对Flash第0页
    (存储安全信息) 解除保护
    Flash_Protect(0, 0);

    //Flash加/解保护

    //2. 复制中断向量表
    (Flash第0页) 至RAM的vecdata[512]

    addr = 0;
    for(i=0; i<512; i++)
    {
        vecdata[i] =
        *((uint32*)addr);
```

```

                                addr+=4;
                                }
                                //3. 修改密钥
                                vecdata[256] = key[0];

//Flash密钥区, 密码高4字节 (KEY_UPPER地址0x0400)
                                vecdata[257] = key[1];

//Flash密钥区, 密码低4字节 (KEY_LOWER地址0x0404)
                                //vecdata[258]-vecdata
                                [260]不变动
                                vecdata[261]          =

0xc0004ac8;    //安全配置域CFMSEC

                                //4. 重写Flash第0页
                                state = Flash_Page_Erase(0);    //擦除Flash第0页
                                if(state == '2') goto Flash_Secure_Exit;

                                BCLR(KEYACC, MCF_CFM_CF
MMCR); //禁止密码匹配, 使能Flash写操作
                                state = Flash_Page_Write(0, 0, vecdata, 2048); //写Flash的1页
                                Flash_Secure_Exit:

                                return state;
                                }

```

2. 解除MCU的保密状态

由于 ColdFire 系列芯片在加密后无法通过 BDM 整体擦除 Flash, 因此, 若芯片已经加密, 需要重新写入程序, 必须先解除密码。这里提供 2 种解除密码方法。

方法一: 通过将写入器设置为 JTAG 模式, 对已经加密的芯片进行整体擦除, 具体使用方法参见光盘中“《32 位 ColdFire 写入器》使用说明”。

方法二: 在芯片中驻留擦除密码或擦除 Flash 的操作接口, 通过调用 Flash_Delete_Key 函数实现。

```

//-----
- - - - - *
/ / 函 数 名 :   F l a s h _ D e l e t e _ K e y
                                *
//功 能: 删除密钥
                                *
/      /      参          数          :          无
                                *
//返 回: '1' 成功; '2' 失败

                                *
/ / 内 部 调 用 :   P a g e _ E r a s e , P a g e _ W r i t e
                                *
//-----
- - - - - *

uint8 Flash_Delete_Key()    //删除密钥

```

```

{
uint32 vecdata[280], addr, data;

//1. 解除页所在扇区的保护
Flash_Protect(0, 0); //Flash加/解保护

//2. 复制中断向量
for(i=0; i<280; i++)
{
    vecdata[i] =
        *((uint32*)addr);
    addr+=4;
}

//3. 删除密钥, 只要
修改CFMSEC的低2字节为0x4ac8以外的值
//vecdata[256] = key[0]; //KEY_UPPER
//vecdata[257] = key[1]; //KEY_LOWER
//vecdata[258] = 0x00000000; //CFMPROT
//vecdata[259] = 0x00000000; //CFMSACC
//vecdata[260] = 0x00000000; //CFMDACC
vecdata[261] = 0x00001111; //CFMSEC

//把存储加密信息的页直接擦除
state = Flash_Page_Erase(0);
if(state == '2') goto Flash_Delete_Key_Exit;
//延时
for(i=0; i<20000; i++);

//更新密钥
BCLR(KEYACC, MCF_CFM_CFMMCR); //禁止密码匹配, 使能Flash写操作
state = Flash_Page_Write(0, 0, vecdata, 2048); //写Flash的1页
Flash_Delete_Key_Exit:
return state;
}

```

第11章 MCF52235 FlexCAN模块编程方法

CAN 总线是一种应用广泛的串行通信协议之一，主要应用于对数据完整性有严格要求的汽车电子和工业控制领域。本章首先概要给出 CAN 总线的通用知识，在此基础上，讨论 ColdFire 系列中 MCF52235 的 FlexCAN 模块的编程要点，并给出编程实例。

11.1 CAN总线通用知识

11.1.1 CAN总线协议的历史概况

控制器局域网（Controller Area Network，CAN），最早出现于 20 世纪 80 年代末，是德国 Bosch 公司为简化汽车电子中信号传输方式并减少日益增加的信号线而提出的。CAN 总线是一个单一的网络总线，所有的外围器件可以挂接在该总线上。1991 年 9 月 Bosch 公司制定并发布了 CAN 技术规范 Version2.0。该技术规范包括 A 和 B 两部分，A 部分给出了曾在 CAN 技术规范 Version1.2 中定义的 CAN 报文格式，而 B 部分给出了标准的和扩展的两种报文格式。为促进 CAN 技术的发展，1992 在欧洲成立了 CiA（CAN in Automation）。在 CiA 的努力推广下，CAN 技术在汽车电子、电梯控制、安全监控、医疗仪器、船舶运输等方面均得到了广泛的应用，目前已经成为国际上应用最广泛的现场总线之一。

在 CAN 技术未得到广泛应用之前，在测控领域的通信方式选择中，大多设计者采用 RS-485 作为通信总线。但 RS-485 存在明显的缺点：一主多从，无冗余；数据通信为命令响应，传输率低；错误处理能力弱。而 CAN 总线技术可以克服这些缺点。CAN 网络上的任何一个节点均可作为主节点主动地与其它节点交换数据；CAN 网络节点的信息帧可以分出优先级，这对于有实时性要求的控制提供了方便；CAN 的物理层及数据链路层有独特的设计技术，使其在抗干扰以及错误检测等方面的性能大大提高。CAN 的上述特点使其成为诸多工业测控领域中首选的现场总线。

11.1.2 CAN硬件系统的典型电路

由于 CAN 控制器只是协议控制器，不能提供物理层驱动，所以在实际使用时每一个 CAN 节点物理上要通过一个收发器与 CAN 总线相连。每个 CAN 模块有发送 CAN_{TX} 和接收 CAN_{RX} 两个引脚。CAN_{TX} 发送串行数据到 CAN 总线收发器，同时 CAN_{RX} 从 CAN 总线收发器接收串行数据。常用的 CAN 收发器有 Philips 公司的 PCA82C250、TI 公司的 SN65HVD1050 等。

1. 最简明的CAN硬件连接方法

最简明的 CAN 硬件连接方法如图 11-1，把所有的 CAN_{TX} 线路经过快速二极管(如 1N4148 等)连接至数据线（以免输出引脚短路）， CAN_{RX} 输入直接连接到这条数据线，数据线由一个上拉电阻拉至+5V，以产生所需要的“1”电平。注意该电路中各节点的地是接在一起的。这个电路最大线长限制在 1m 左右。主要用于在电磁干扰较弱环境下的近距离通信。

进行 CAN 通信节点调试时，可以利用这个简单且易于实现的电路。另外，可以利用该电路理解 CAN 总线的通信机制。

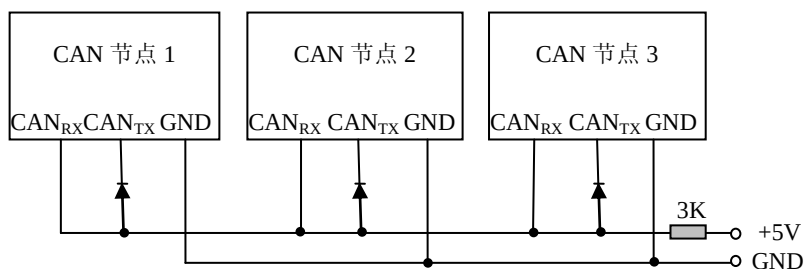


图 11-1 无需CAN收发器芯片的电路连接

2. 常用的CAN硬件系统的组成

常用的 CAN 硬件系统的组成如图 11-2 所示。

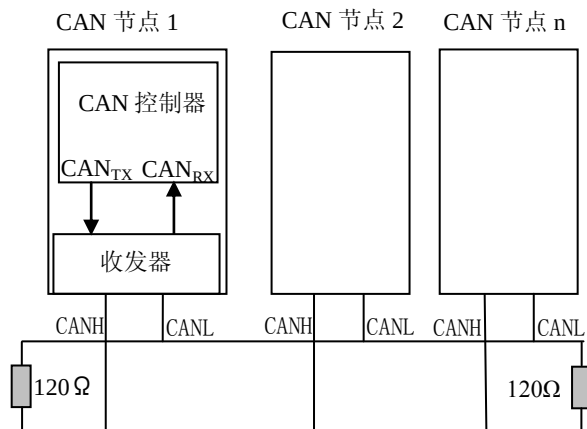


图11-2 常用的CAN硬件系统组成

注意：CAN 通信节点上一般需要添加 120Ω 终端电阻。每个 CAN 总线只需要两个终端电阻，分别在主干线的两个端点，支线上的节点不必添加。下面给出这个框图的实际电路。

3. 带隔离的典型CAN硬件系统电路

Philips 公司的 CAN 总线收发器 PCA82C250 能对 CAN 总线提供差动发送能力并对 CAN 控制器提供差动接收能力。在实际应用过程中，为了提高系统的抗干扰能力，CAN 控

制器引脚 CAN_{TX}、CAN_{RX} 和收发器 PCA82C250 并不是直接相连的，而是通过由高速光耦合器 6N137 构成的隔离电路后再与 PCA82C250 相连，这样可以很好地实现总线上各节点的电气隔离。一个带隔离的典型 CAN 硬件系统电路如图 11-3 所示。

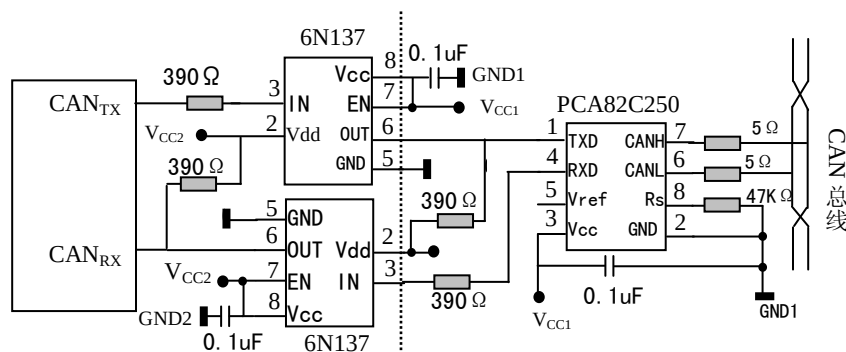


图11-3 带隔离的典型CAN硬件系统电路

该电路连接需要特别注意以下几个问题：

（1）6N137 部分的电路所采用的两个电源 VCC1 和 VCC2 须完全隔离，否则，光耦达不到完全隔离的效果。可以采用带多个 5V 输出的开关电源模块实现。

（2）PCA82C250 的 CANH 和 CANL 引脚通过一个 5Ω 的限流电阻与 CAN 总线相连，保护 PCA82C250 免受过流的冲击。PCA82C250 的电源管脚旁应有一个 0.1μF 的去耦电容。Rs 引脚为斜率电阻输入引脚，用于选择 PCA82C250 的工作模式（高速/斜率控制^①/待机）脚上接有一个下拉电阻，电阻的大小可根据总线速率适当的调整，其值一般在 16kΩ~140kΩ 之间，图 11-3 中选用 47kΩ。关于电路相连的更多细节请参见 6N137 手册以及 PCA82C250 手册。

^①在斜率控制模式中，由于CANL/CANH上的信号的单端转换速度和流出引脚Rs的电流IRs成比例关系（或称斜率关系），而电流IRs的大小主要由Rs阻值决定，因此Rs的阻值变化将引起转换速度的变化。在斜率控制模式下Rs阻值一般取在16.5K~140K之间。高速模式下，Rs阻值在0~1.8K之间。

4. 不带隔离的典型CAN硬件系统电路

在电磁干扰较弱的环境下，隔离电路可以省略，这样CAN控制器可直接与CAN收发器相连，如图11-4所示。

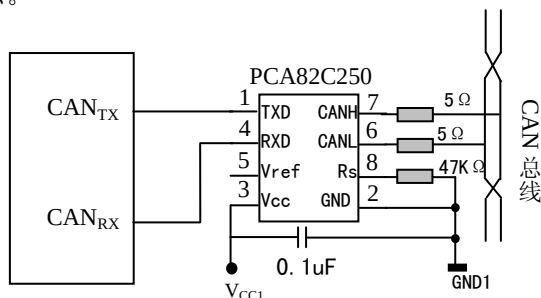


图11-4 不带隔离的典型CAN硬件系统电路

11.1.3 CAN总线的有关基本概念

CAN 通信协议主要描述设备之间的信息传递方式。CAN 各层的定义与开放系统互连模型 OSI 一致，每一层与另一设备上相同的那一层通信。实际的通信发生在每一设备上相邻的两层，而设备只通过物理层的物理介质互连。在 CAN 规范的 ISO 参考模型中，定义了模型的最下面两层：数据链路层和物理层，它们是设计 CAN 应用系统的基本依据。规范主要是针对 CAN 控制器的设计者而言，对于大多数应用开发者来说，只需对 CAN V2.0 版技术规范的基本结构、概念、规则作一般了解，知道一些基本参数和可访问的硬件即可。下面给出与 CAN 通信接口编程相关的部分术语。

1. CAN总线上的数据表示

CAN 总线由单一通道（Single Channel）组成，借助数据再同步实现信息传输。CAN 技术规范中没有规定物理通道的具体实现方法，物理层可以是单线（加地线）、两条差分线、光纤等。实际上大多数使用双绞线，利用差分方法进行信号表达，它是一种半双工通信方式。

CAN 总线上用显性（Dominant）和隐性（Recessive）分别表示逻辑 0 和逻辑 1。若不同控制器同时向总线发送逻辑 0 和逻辑 1 时，总线上出现逻辑 0（相当于逻辑与的关系）。物理上，现行的 CAN 总线大多使用二线制作为物理传输介质，使用差分电压表达逻辑 0 和逻辑 1。设两条信号线分别被称为 CAN_H 和 CAN_L，如图 11-5 所示。在隐性状态（即逻辑 1）时，CAN_H 和 CAN_L 被固定在平均电压 2.5V 左右，电压差($V_{diff}=V_{CAN_H}-V_{CAN_L}$)近似于 0。在显性状态（即逻辑 0）时，CAN_H 比 CAN_L 高，此时通常 CAN_H=3.5V，CAN_L=1.5V，电压差($V_{diff}=V_{CAN_H}-V_{CAN_L}$)在 2V 左右。在总线空闲或隐性位期间，发送隐性位。

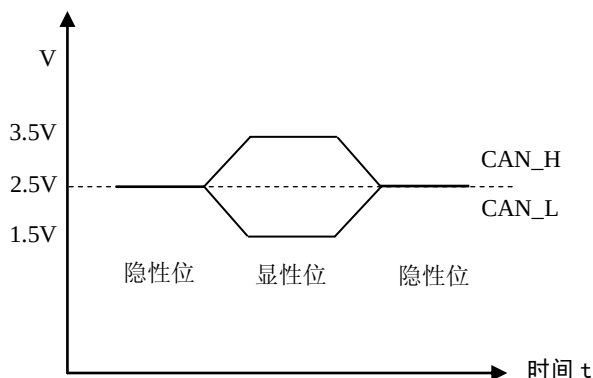


图11-5 总线数据表示

2. 报文、信息路由、位速率、位填充

报文 (Message): 是指在总线上传输的固定格式的信息，其长度是有限制的。当总线空闲时，总线上任何节点都可以发送新报文。报文被封装成帧 (Frame) 的形式在总线上传送，具体定义见 11.1.4 小节。

信息路由 (Information Routing): 在 CAN 系统中，CAN 不对通信节点分配地址，报文的寻址内容由报文的标识符 ID 指定。总线上所有节点可以通过报文过滤的方法来判断是否接收报文。

位速率 (Bit Rate): 是指 CAN 总线的传输速率。在给定的 CAN 系统中，位速率是固定唯一的。CAN 总线上任意两个节点之间的最大传输距离与位速率有关，表 11-1 列出了距离与位速率的对应关系。这里的最大距离是指在不使用中继器的情况下两个节点之间的距离。

表 11-1 CAN 总线上任意两节点最大距离及位速率对应表

位速率/kbps	1000	500	250	125	100	50	20	10	5
最大距离/m	40	130	270	530	620	1300	3300	6700	10000

位填充 (Bit Stuffing): 是为防止突发错误而设定的功能。当同样的电平持续 5 位时则添加一个位的反型数据，即连续出现5个“0”时，需要添加一个“1”。连续出现5个“1”时，需要添加一个“0”。

3. 多主机、标识符、优先权、仲裁

多主机 (Multimaster): CAN 总线是一个多主机 (Multimaster) 系统，总线空闲时，总线上任何节点都可以开始向总线上传送报文，但只有最高优先权报文的节点可获得总线访问权。CAN 通信链路是一条可连接多节点的总线。理论上，总线上节点数目是无限制的，实际上，节点数受限于延迟时间和总线的电气负载能力。例如，当使用 Philips P82C250 作为 CAN 收发器时，同一网络中一般最多允许挂接 110 个节点。

标识符 ID: CAN 节点的唯一标识。在实际应用时，应该给 CAN 总线上的每个节点按照一定规则分别一个唯一的 ID。每个节点发送数据时，发送的报文帧中含有发送节点

的 ID 信息。

在 CAN 通信网络中, CAN 报文以广播方式在 CAN 网络上发送, 所有节点都可以接收到报文, 节点通过判断接收到的标识符 ID 决定否接收该报文。报文标识符 ID 的分配规则一般在 CAN 应用层协议实现 (比较著名的 CAN 应用层协议为: CANopen 协议、DeviceNet 等)。由于 ID 决定报文发送的优先权, 因此 ID 的分配规则在实际应用中必须给予重视。一般可以用标识符的某几位代表发送节点的地址。接收到报文的节点可以通过解析接收报文的标识符 ID, 来判断该报文来自哪个节点, 属于何种类型的报文等。下面给出 CANopen 协议最小系统配置的一个 ID 分配方案, 供实际应用时参考。

D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
功能代码				节点地址						

该分配方案是一个面向设备的标识符分配方案, 该方案通过 4 位的功能代码区分 16 种不同类型的报文, 有 7 位节点地址, 可表达 128 个节点。但要注意到 CAN 协议中, 要求 ID 的高 7 位不能同时为 1。报文标识符 ID 的分配方法应遵循以下原则: 在同一系统中, 必须保证节点地址唯一, 这样每个报文的 ID 也就唯一了。

优先权 (Priorities): 在总线访问期间, 报文的标识符 ID 定义了一个静态的报文优先权。在 CAN 总线上发送的每一个报文都具有唯一的一个 11 位或 29 位的标识符 ID, 在总线仲裁时, 显性位 (逻辑 0) 的优先权高于隐性位 (逻辑 1), 从而标识符越小, 该报文拥有越高的优先权, 因此一个拥有全 0 标识符的报文具有总线上的最高级优先权。当有两个节点同时进行发送时, 必须通过“无损的逐位仲裁”^①方法来使得有最高优先权的报文优先发送。

仲裁 (Arbitration): 总线空闲时, 总线上任何节点都可以开始发送报文, 若同时有两个或两个以上节点开始发送, 总线访问冲突运用逐位仲裁规则, 借助于标识符 ID 解决。仲裁期间, 每一个发送器都对发送位电平与总线上检测到的电平进行比较, 若相同, 则该节点继续发送。当发送的是一“1”而监视到的是一“0”(见图 11-1), 则该节点失去仲裁, 退出发送状态。举例说明, 若某一时刻有两个 CAN 节点 A、B 同时向总线发送报文, A 发送报文的 ID 为 0b00010000000, B 发送报文的 ID 为 0b01110000000。由于节点 A、B 的 ID 的第 10 位都为“0”, 而 CAN 总线是逻辑与的, 因此总线状态为“0”, 此时两个节点检测到总线位和它们发送的位相同, 因此两个节点都认为是发送成功, 都继续发送下一位。发送第 9 位时, A 发送一个“0”, 而 B 发送一个“1”, 此时总线状态为“0”。此时 A 检测到总线状态“0”与其发送位相同, 因此 A 认为它发送成功, 并开始发送下一位。但此时 B 检测到总线状态“0”与其发送位不同, 它会退出发送状态并转为监听方式, 且直到 A 发送完毕, 总线再次空闲时, 它才试图重发报文。

4. 远程数据请求、应答

远程数据请求 (Remote Data Request): 当总线上某节点需要请求另一节点发送数据时, 这种情况, 在 CAN 总线协议术语中叫远程数据请求 (Remote Data Request)。需要远

^① “无损的逐位仲裁”: 当总线上出现报文冲突时, 仲裁机制逐位判断标识符, 实现高优先权的报文能够不受任何损坏地优先发送。

程数据请求时，可通过发送远程帧实现，有关帧内容见 11.1.4 小节。

应答 (Acknowledgment): 所有接收器对接收到的报文进行一致性 (Consistency) 检查。对于一致的报文，接收器给予应答；对于不一致的报文，接收器做出标志。

5. 故障界定、错误标定和恢复时间

故障界定 (Fault Confinement): CAN 节点能够把永久故障和短暂的干扰区别开来，故障节点会被关闭。

错误标定和恢复时间 (Error Signaling and Recovery Time): 任何检测到错误的节点会标志出已被损坏的报文。此报文会失效并自动重传。若不再出现错误，则从检测出错误到下一报文传送开始为止，恢复时间最多为 31 位的时间。

6. CAN 的分层结构

CAN 遵从 ISO/OSI 标准模型。按照该模型，CAN 结构划分为两层：物理层和数据链路层。在 CAN 技术规范 2.0B 版本中，数据链路层中的逻辑链路控制子层和介质访问控制子层分别对应于 2.0A 版本中的“对象层”和“传输层”。

物理层 (The Physical Layer): CAN 规范没有定义具体的物理层，允许用户根据具体需要定制物理层。物理层给出实际信号的传输方法，作用是在不同节点之间根据所有的电气属性进行位信息的实际传输。当然，在同一网络内，物理层对于所有的节点必须是相同的。尽管如此，在选择物理层实现方式还是很自由的。

数据链路层又分为逻辑链路控制子层和介质访问控制子层：

逻辑链路控制子层 (Logic Link Control, LLC): 负责报文滤波、过载通知和恢复管理。

介质访问控制子层 (Media Access Control, MAC): MAC 是 CAN 协议的核心。它把接收到的报文提供给 LLC，以及接收来自 LLC 的报文。MAC 负责位定时及同步、报文分帧、仲裁、应答、错误标定、故障界定等。

11.1.4 帧结构

CAN 总线协议中有四种报文帧 (Message Frame)，它们分别是数据帧、远程帧、错误帧、过载帧。其中数据帧和远程帧，与用户编程相关，错误帧和过载帧由 CAN 控制硬件处理，与用户编程无关。

1. 数据帧

在 CAN 节点之间的通信中，要将数据从一个节点发送器传输到另一个节点的接收器，必须发送数据帧。数据帧由 7 个不同的位场^①组成：帧起始 (Start Of Frame symbol, SOF)、仲裁场、控制场、数据场、CRC 场、应答场、帧结束 (End Of Frame, EOF)。数据帧组成如图 11-6 所示。

^① 这里用“场”，有的中文翻译用“域”，也有使用“字段”，均对应英文“Field”

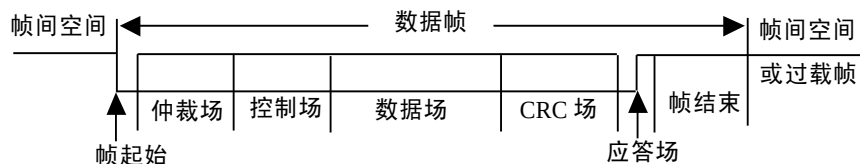


图11-6 数据帧组成

帧起始 SOF: 标志数据帧和远程帧的起始，仅由一个单独的“0”位组成。只有在总线空闲时，才允许节点开始发送报文。只要有一个节点发送帧起始 SOF，其它节点检测到该信号，与之同步。

仲裁场: 在 CAN2.0B 中定义标准帧与扩展帧两种帧格式。标准帧的标识符 ID 为 11 位，扩展帧的标识符 ID 为 29 位（11 位标准 ID+18 位扩展 ID）。

标准帧的仲裁场由 11 位标准 ID 和 1 位远程请求发送位(Remote Transmission Request, RTR)组成（图 11-7）。在数据帧里 RTR=0，在远程帧里 RTR=1。实际发送顺序是从 ID10 到 ID0。标准 ID 的高 7 位（ID10~ID4）不能全是 1（读者思考：为什么高 7 位不能全为 1？）。

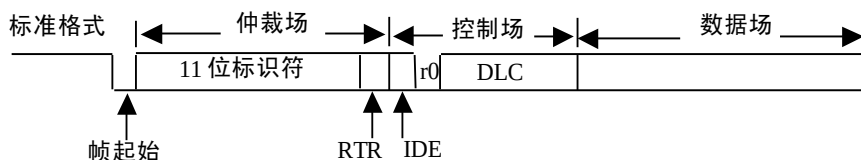


图11-7 数据帧标准格式中的仲裁场结构

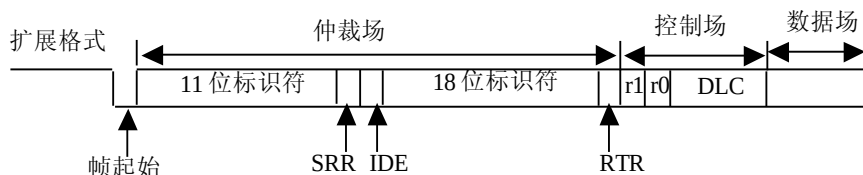


图11-8 数据帧扩展格式中的仲裁场结构

扩展帧的仲裁场由 11 位标准 ID、1 位远程发送请求替代位(Substitute Remote Request, SRR)、1 位标识符扩展位 (ID Extended bit, IDE)、18 位扩展 ID、1 位远程请求发送位 RTR 组成，如图 11-8 所示。在标准格式中 IDE=0，而扩展格式中 IDE=1。扩展帧中“替代远程请求位” SRR 位的实际位置是标准帧中 RTR 位的位置。当标准帧与扩展帧发生冲突且扩展帧的基本 ID 同标准帧的标识符一样时，标准帧优先于扩展帧。

6 位控制场: 标准帧中控制场包括：数据长度代码 DLC、IDE 位（为 0）、保留位 r0。扩展帧的控制场包括：数据长度代码 DLC、两个必须为 0 的保留位 r1 和 r0。

4 位数据长度代码 DLC: 指示了数据场中字节数，DLC=0000~1000（即十进制的 0~8，0000 代表空）表示数据场中字节数。若设置 DLC 大于 8，无效。

数据场：数据场为实际要发送的数据。字节数由 DLC 决定，在发送一个字节时，先发高位 MSB，最后发低位 LSB。

数据场之后跟随 16 位 CRC 场（15 位 CRC 校验位、1 位固定为“1”的 CRC 的界定符）、2 位应答 ACK 场（1 位应答间隙 ACK Slot、1 位应答界定符 ACK Delimiter）、帧结束 EOF（7 个“1”位）。在应答 ACK 场里，发送节点发送两个“1”位。当接收器正确地接收到有效的报文时，接收器就会在应答间隙期间向发送器发送一个“0”位以示应答。

2. 远程帧

远程帧跟数据帧非常相似，不同之处在于二者的远程发送请求位（Remote Transmission Request.）不同。数据帧的 RTR 位为“0”，远程帧的 RTR 位为“1”。需要特别注意的一点是远程帧没有数据场。总线上节点发送远程帧目的在于请求发送具有同一标识符的数据帧。作为数据接收的节点，可以借助于发送远程帧启动其资源节点传送数据。远程帧也有标准格式和扩展格式，而且都由 6 个不同的位场组成：帧起始、仲裁场、控制场、CRC 场、应答场、帧结束。远程帧的组成如图 11-9 所示。

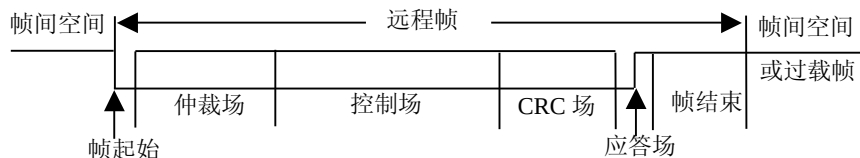


图11-9 远程帧的组成

为方便描述，下面我们自定义一种简单的标准 ID 分配方案，来阐述远程帧的使用方法。在实际应用过程中，用户可自行制定 ID 分配方案，或按照某种 CAN 高层协议来分配 ID。自定义 ID 分配方案如下：

D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
功能代码			源节点地址				目的节点地址			

假设现有节点 A 和 B，设置节点 A 的地址为 0，B 的地址为 1，A 节点需向 B 节点请求一温度数据。我们假定请求温度数据的功能代码为 0。则 A 需向 B 发送一个 ID 为 0x001 的远程帧，远程帧发送完毕后 A 将自动变为接收 ID 为 0x001 的数据帧的接收节点。当 B 检测到该远程帧时，将发送一个 ID 为 0x001 的数据帧作为回应，此时 A 将接收到 B 节点发来的数据帧。这样一次远程请求交互就完成了。

远程帧不是必须的，例如应用层协议 DeviceNet 中未用远程帧，但并未影响 DeviceNet 在可靠运行、通讯效率方面的性能，且 DeviceNet 也是国际流行的 CAN 应用层协议。

3. 错误帧

错误帧由 CAN 控制器的硬件进行处理，与用户编程无关。下面概要发送错误帧的工作机制。

CAN 节点通过发送引脚发送报文时，接收引脚也在同步接收报文，当发送报文的 ACK 场为“1”时，接收到的应答间隙（ACK Slot）一定要是“0”才代表发送成功。在 CAN

总线网络中只要有一个节点正确接收到了报文，并将发送节点的应答间隙写为“0”，则发送节点就认为发送数据成功。在报文的应答过程中，若某一节点检测到错误，则它会立刻发送错误帧，一般是发送连续的 6 个 0 或 1，由 CAN 的位填充原理可知，当有五个连续的 0 或 1 出现时，为了传送中的同步，必须插入一个反型位作为填充。因此如果连续出现 6 个或 6 个以上的 0 或 1，则此次传送错误，报文将被丢弃。此时当发送节点收到这个错误帧后，便知道发送出错，并试图重发报文。任何节点检测到总线错误都会发送错误帧。

错误帧由两个不同的场组成。第一个场是由不同节点提供的错误标志（FLAG）的叠加；第二个场是错误界定符。错误帧的组成如图 11-10 所示。



图 11-10 错误帧组成

错误标志有两种形式：主动错误（Error Active）^①标志和被动错误（Error Passive）^②标志。主动错误标志由 6 个连续的 0 位组成，而被动错误标志由 6 个连续的 1 位组成。

检测到错误条件的“主动错误”的节点通过发送主动错误标志指示错误。错误标志的形式破坏了从帧起始到 CRC 界定符的位填充的规则，或者破坏了 ACK 场或帧结束的固定形式。所有其他的节点由此检测到错误条件并与此同时开始发送错误标志。因此，6 个连续“0”的序列导致一个结果，这个结果就是把个别节点发送的不同的错误标志叠加在一起。这个序列的总长度最小为 6 个位，最大为 12 个位。

检测到错误条件的“错误被动”的节点试图通过发送被动错误标志指示错误。“被动错误”的节点等待 6 个相同极性的连续位（这 6 个位处于被动错误标志的开始）。当这 6 个相同的位被检测到时，被动错误标志的发送就完成了。

错误界定符包括 8 个“1”。错误标志发送以后，每一节点都发送“1”并一直监视总线直到检测出一个“1”为止，然后就开始发送其余 7 个“1”。

为了能正确地终止错误帧，“被动错误”的节点要求总线至少有 3 个位时间的总线空闲（如果“被动错误”的接收器有局部错误的话）。因此，总线的载荷不会达到 100%。

4. 过载帧

过载帧由 CAN 控制器的硬件进行处理，与用户编程无关。下面概要发送过载帧的工作机制。

过载帧用于在先行和后续数据帧（或远程帧）之间提供一附加的延时。过载帧包括两个位场：过载标志和过载界定符。过载帧的组成如图 11-11 所示。

^① Error Active：也称为“错误激活”。

^② Error Passive：也称为“错误认可”。



图11-11 过载帧的组成

有三种过载的情况会引发过载帧的传送：

- (1) 接收器的内部情况（此接收器对于下一数据帧或远程帧需要有一延时）。
- (2) 在间歇的第一和第二字节检测到一个“0”位。
- (3) 如果 CAN 节点在错误界定符或过载界定符的第 8 位（最后一位）采样到一个 0 位，节点会发送一个过载帧（不是错误帧）。

根据过载情况 1 而引发的过载帧只允许起始于所期望的间隙的第一个位时间，而根据情况 2 和情况 3 引发的过载帧应起始于所检测到“0”位之后的位。通常，为了延时下一个数据帧或远程帧，两种过载帧均可产生。

过载标志由 6 个 0 位组成。由于过载标志的格式破坏了间隙域的固定格式，因此，所有其他的节点都检测到过载条件，并与此同时发出过载标志。如果在间隙的第 3 个位期间检测到 0 位，则这个位将被解释为帧的起始。

过载界定符包括 8 个 1 位，过载标志被传送后，节点就一直监视总线，直到检测到一个从 0 位到 1 位的跳变为止。这时，总线上的每个节点完成了各自过程标志的发送，并开始同时发送其余 7 个 1 位。

11.1.5 位时间

所谓位时间，是指发送一位所需要的时间。实际工作过程的位时间与系统设定的位时间少有偏差，把理想情况下位时间称为标称位时间（Nominal Bit Time），相应的位速率（每秒发送的位数）称为标称位速率。标称位速率=1/标称位时间。

一个标称位时间分为 4 个时间段：同步段（SYNC_SEG）、传播段（PROP_SEG）、相位段 1（PHASE_SEG1）、相位段 2（PHASE_SEG2）。如图 11-12 所示。CAN 总线初始化时要通过相应的寄存器对传播段、相位段 1 及相位段 2 的时间长度进行设置。

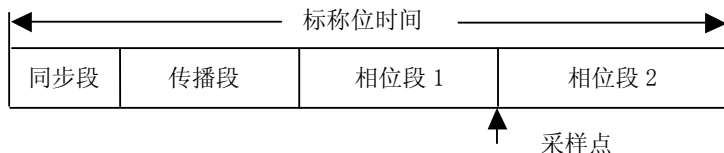


图11-12 标称位时间组成示意图

同步段（SYNC_SEG）：连接在总线上的多个 CAN 节点通过同步段实现时序调整，同步进行接收和发送工作。由 1 电平到 0 电平的跳变或由 0 电平到 1 电平的跳变最好出现

在该段中。

传播段 (PROG_SEG): 传播段用于补偿网络内的物理延时时间。它是总线上输入比较器延时和输出驱动器延时总和的 2 倍。

相位段 1 (PHASE_SEG1)、相位段 2 (PHASE_SEG2): 相位段用于补偿边沿阶段的误差。这两个段可以通过重新同步加长或缩短。

采样点 (Sample Point): 采样点是读总线电平并解释各位的值的一个时间点。采样点位于相位段 1 之后。

信息处理时间 (Information Processing Time): 信息处理时间是以一个采样点作为起始的时间段。采样点用于计算后续位的位电平。

最小时间份额 (Time Quanta, 简称 Tq): 最小时间份额是取自振荡器周期的固定时间单元, 也称为串行时钟 S_{clock} 周期。位时间与最小时间份额 Tq 的关系是:

$$\text{位时间} = m \times Tq$$

其中, m 为可编程的预比例因子, 其范围是 1~32 之间的整数。

M 的计算公式如下:

$$m = \text{同步段} + \text{传播段} + \text{相位段 1} + \text{相位段 2}$$

通常, 同步段为 1 个 Tq, 传播段可设置成 1、2、3...8 个 Tq, 相位段 1 可设置成 1、2、3...8 个 Tq, 相位段 2 为相位段 1 和信息处理时间之间的最大值, 信息处理时间少于或等于 2 个 Tq。一个位时间总的 Tq 值可以设置在 8~25 的范围。

在确定一个 CAN 总线的通信速率时, 主要根据上述参数确定。

11.2 MCF52235 FlexCAN 模块概述与编程结构

FlexCAN 模块是一种 CAN 协议的通信控制单元。FlexCAN 模块支持 CAN2.0B 协议中标准和扩展帧信息格式。虽然 CAN 协议起初是为汽车总线设计的, 但它并不专用于现代汽车总线。

11.2.1 FlexCAN 特性

FlexCAN 通信模块支持 CAN2.0B 协议, 支持标准和扩展帧格式, 有 16 个报文缓冲区 (MB0~MB15), 每个报文缓冲区内嵌 16 字节的 RAM, 可通过控制/状态字段部分的 4 位 CODE 域被独立配置成为发送/接收报文缓冲区, 而每个报文缓冲区都可以设置相应的发送/接收中断。其主要特点如下:

(1) 完全支持 CAN2.0B 协议: 标准数据帧 (可达 109 位) / 扩展数据帧 (可达 127 位) / 0~8 字节的数据长度 / 可编程控制的波特率设置, 数据传输速率可达 1Mbps / 与内容相关的寻址方式。

(2) 多达 16 个报文缓冲区, 每个报文缓冲区支持多达 8 字节数据传输。每个报文缓冲区都可以独立配置为发送缓冲区或接收缓冲区, 支持标准和扩展帧格式。

(3) 支持监听模式。

(4) 3 个可编程控制的报文过滤掩码寄存器：全局掩码寄存器 RXGMASK（作用于 MB0~MB13）、MB14 掩码寄存器 RX14MASK（作用于 MB14）、MB15 掩码寄存器 RX15MASK（作用于 MB15）。

(5) 可编程控制缓冲区报文发送次序：ID 最小的报文优先发送或最小的报文缓冲区内的报文优先发送。

(6) 基于 16 位自由运行定时器的时间戳^①机制。

(7) 全局网络时间，通过特殊报文进行同步。

(8) 中断掩码。

(9) 传输介质相互独立。

(10) 多主机总线。

(11) 强抗电磁干扰性。

(12) 对于高优先权信息具有短的响应时间。

11.2.2 操作模式

1. 正常模式

在正常模式下，CAN 模块收发数据帧、远程帧以及错误帧时，CAN 协议的所有功能全部是允许状态。对于一些控制比较严格的寄存器，在用户模式和管理员模式下访问是有区别的。

2. 冻结模式

冻结模式又称为调试模式，主要用于 FlexCAN 模块初始化设置。通过以下设置可以进入冻结模式：

(1) 配置寄存器 CANMCR 的 FRZ 位置 1

(2) 配置寄存器 CANMCR 的 HALT 位置 1 或 BKPT（BKPT，中断请求输入引脚。MCF5223x 系列芯片的第 2 脚）位置 1。

当请求进入冻结模式以后，FlexCAN 处于等待状态，直到 CAN 总线上出现空隙或空闲状态，或 FlexCAN 进入被动错误或总线关闭状态。只要有一个条件发生，则 FlexCAN 等待所有内部活动完成（如仲裁、匹配、移入和移出）后，将执行以下操作：

(1) FlexCAN 停止发送/接收帧。

(2) 禁止预分频，终止所有 CAN 总线通信。

(3) FlexCAN 忽略它的 CANRX 引脚并驱动它的 CANTX 引脚为 1 位。

(4) FlexCAN 丧失与 CAN 总线的同步且 CANMCR 中的 NOTRDY 和 FRZACK 被置位。

(5) CPU 允许读写错误计数器寄存器（其他模式下为只读）。

执行完上述操作之后，访问 FlexCAN 其他寄存器之前，用户必须等待 FRZACK 置位。只有当 FRZACK 被置位时，FlexCAN 才正式进入冻结模式。冻结模式下，所有的内存映

^① 时间戳：Time Stamp，用于发送/接收同步，详见 11.2.4 节。

像寄存器都是可访问的。如果要退出该模式，则必须拉低 BKPT 线，或清 CANMCR 的 HALT 位。在退出该模式后，FlexCAN 在进行 CAN 总线通信前，通过等待连续 11 个 1 位，重新同步 CAN 总线。

3. 禁止模式

该模式禁止 FlexCAN 模块。通过置位 CANMCR 的 MDIS 可以进入禁止模式。如果该模块在冻结模式下被禁止，则系统时钟将被关闭，LPMACK 被置位，FRZACK 位被清零。如果该模块在发送或接收过程中被禁止，则 FlexCAN 进行以下操作：

- (1) 等待空闲或总线关闭状态，或等待第 3 个空隙位并检查它是否为“1”。
- (2) 等待所有内部活动（如仲裁、匹配、移入和移出等）结束。
- (3) 忽略它的 Rx 引脚并驱动它的 Tx 引脚为“1”。
- (4) 关闭系统时钟。

此时，总线接口单元正常工作。CPU 可以访问除自由运行计数器、错误计数寄存器和报文缓冲区之外的内存映像寄存器。清 MDIS 位，可以退出该模式，恢复时钟并清 LPMACK 位。

4. 闭环模式

在系统调试时，往往让 CAN 控制器工作在闭环模式（也称为“自测模式”），用于检查 CAN 控制器工作是否正常。当控制寄存器 CANCTRL 的 LPB 位置位时，系统将进入闭环模式。在这种模式下，FlexCAN 发送器的位流输出回馈到内部的接收器输入。输入引脚 CANRX 将被忽略，且 CANTX 输出变为“1”。发送报文时，FlexCAN 模块像在正常模式一样，正常发送数据，而接收器则认为接收它自己的报文与接收远程节点的报文相同。FlexCAN 为保证能正确接收到自己发送的报文，将忽略应答间隙内的应答字段。报文发送和接收时，如果中断使能则 FlexCAN 将向 CPU 产生中断。FlexCAN 通过读取接收缓冲区的值可以判断 FlexCAN 工作是否正常。

5. 监听模式

在这种模式下，FlexCAN 模块禁止数据发送，所有的错误计数器都被冻结，且该模块工作在 CAN 被动错误模式。只有被其他 CAN 结点应答了的报文才可以被监听的结点接收。如果 FlexCAN 检测到一个还没有被应答的报文，则标记为一个 BIT0 错误，就好像它试图应答报文一样。由于 FlexCAN 模块在监听模式不影响 CAN 总线，所以它可以作为一个监视器使用，进行自动位速率检测。

11.2.3 FlexCAN模块的内存映像以及寄存器定义

FlexCAN 地址空间共有 384 字节，其中有 128 字节的基地址为 0x1C_0000（寄存器只使用了其中一部分地址空间，还有部分地址空间未使用），256 字节的基地址为 0x1C_0080（全部用于报文缓冲区）。除报文缓冲区 MB0~MB15 长度为 2048 位外，其他寄存器都是 32 位寄存器。除配置寄存器 CANMCR 只能在管理员模式下访问外，其他寄存器在管理员

/用户模式都可访问。所有寄存器均可读写。FlexCAN 的内存映像见表 11-2。

表 11-2 FlexCAN 内存映像

IPSBAR 偏移地址	寄存器	是否受硬件 复位影响	是否受软件 复位影响	复位值
FlexCAN				
0x1C_0000	FlexCAN 配置寄存器 (CANMCR)	Y	Y	0xD890_000F
0x1C_0004	FlexCAN 控制寄存器 (CANCTRL)	Y	N	0x0000_0000
0x1C_0008	自由运行定时器 (TIMER)	Y	Y	0x0000_0000
0x1C_0010	全局掩码寄存器 (RXGMASK)	Y	N	0x1FFF_FFFF
0x1C_0014	MB14 掩码寄存器 (RX14MASK)	Y	N	0x1FFF_FFFF
0x1C_0018	MB15 掩码寄存器 (RX15MASK)	Y	N	0x1FFF_FFFF
0x1C_001C	错误计数寄存器 (ERRCNT)	Y	Y	0x0000_0000
0x1C_0020	错误和状态寄存器 (ERRSTAT)	Y	Y	0x0000_0000
0x1C_0028	中断掩码寄存器 (IMASK)	Y	Y	0x0000_0000
0x1C_0030	中断标志寄存器 (IFLAG)	Y	Y	0x0000_0000
0x1C_0080	报文缓冲区 0~15 (MB0~15)	N	N	未定义

控制寄存器的设置是在初始化阶段完成的,此时 FlexCAN 模块与 CAN 总线尚未同步。而配置寄存器的设置可以在同步之后进行改变。要改变配置寄存器,首先要通过置位 CANMCR 的 HALT 位使 FlexCAN 进入冻结模式。

由于该模块寄存器内容较多,所以下文中以下划线标注初学者需重点学习的位。另外,错误计数寄存器 (ERRCNT) 和错误和状态寄存器 (ERRSTAT) 主要用于 CAN 总线错误处理,它的错误处理比较特殊,具体实现机制可参考 CAN2.0B 协议,初学者可不关心,所以放在了 11.5 节,有兴趣的读者可查看 11.5 节,在此不再赘述。

1. FlexCAN 配置寄存器 (CANMCR)

CANMCR 定义全局系统配置,如模块的操作模式和最大报文缓冲区数目等。除 MAXMB 字段只能在冻结模式下修改之外,该寄存器的其余字段可以在任意时刻被访问。CANMCR 偏移地址为 0x1C_0000。D29, D26, D22~D21, D19~D4: 未定义。

数据位	D31	D30	D28	D27	D25	D24	D23	D20	D3~D0
定义	MDIS	FRZ	HALT	NOTRDY	SOFTTRST	FRZACK	SUPV	LPMACK	MAXMB
复位	1	1	1	1	0	0	1	1	1

D31—MDIS 位, FlexCAN 模块禁止位。该位控制 FlexCAN 模块是否禁止。当禁止时, FlexCAN 模块关闭 CAN 驱动接口和报文缓冲区子模块的 FlexCAN 时钟。该只读位不受复位影响。MDIS=0, 使能 FlexCAN 模块, 使能时钟; MDIS=1, 禁 FlexCAN 模块和时钟。
D30—FRZ 位, 冻结模式使能位。该位控制 FlexCAN 模块是否进入冻结模式。该位置位时, 若 BKPT 线被拉高或者 HALT 被置位, 则 FlexCAN 进入冻结模式。清该位则 FlexCAN 退出冻结模式。FRZ=0, FlexCAN 模块忽略 BKPT 信号和 CANMCR 的 HALT 位; FRZ=1, FlexCAN 模块进入冻结模式。

D28—HALT 位, FlexCAN 停止位。置该位将使 FlexCAN 模块进入冻结模块。置该位

与拉高 BKPT 信号作用相同。复位后该位被置位，报文缓冲区和控制寄存器初始化以后该位会被清零。在该位被清零以前，FlexCAN 报文缓冲区接收和发送被禁止。当进入冻结模块时，CPU 对错误计数寄存器（ERRCNT）具有写权限，其他情况下只具有读权限。HALT=0，FlexCAN 正常操作；HALT=1，若 FRZ=1，则 FlexCAN 进入冻结模块。

D27—NOTRDY 位，FlexCAN 准备就绪位。该位标志 FlexCAN 是否处于禁止模式或冻结模块。该只读位在 FlexCAN 退出这些模式时被清零。NOTRDY=0，FlexCAN 处于正常模式、监听模式或闭环模式；NOTRDY=1，FlexCAN 处于禁止模式或冻结模块。

D25—SOFTTRST 位，FlexCAN 软复位位。当该位置位时，FlexCAN 复位它的内部状态机（序列器、错误计数器、错误标志以及自由运行的定时器）以及主机接口寄存器（CANMCR、TIMER、ERRCNT、ERRSTAT、IMASK 和 IFLAG）。注意：MDIS 位不受软复位影响。控制 CAN 总线接口的配置寄存器 CANCTRL、RXGMASK、RX14MASK 和 RX15MASK，以及报文缓冲区不受软复位影响。SOFTTRST 可以用于调试。由于复位是同步的且需要按照一定的时序执行一个请求/应答程序，因此它需要一定的时间来完成。当复位时，SOFTTRST 保持置位，复位完成时自动清零。用户可以轮询该位判断软复位是否结束。SOFTTRST=0，软复位周期结束；SOFTTRST=1 软复位尚在进行。

D24—FRZACK 位，FlexCAN 冻结应答位。该只读位标志 FlexCAN 模块是否进入冻结模式。用户可以在发出进入冻结模块请求以后轮询该位，依此来判断模块是否已经进入冻结模块。当退出冻结模式时，该位在 FlexCAN 预分频使能后被清零。FRZACK=0，FlexCAN 已退出冻结模块且使能了预分频；FRZACK=1，FlexCAN 进入冻结模块，且禁止了预分频。

D23—SUPV 位，管理员/用户数据空间。该位设置 FlexCAN 寄存器为管理员/用户数据空间。SUPV=0，由 SUPV 控制的寄存器在管理员模式和用户模式下都可以访问；SUPV=1，由 SUPV 控制的寄存器只能在管理员模式下访问。

D20—LPMACK 位，低功耗模式应答位。该只读位标志 FlexCAN 是否处于禁止模式。只有在当前发送或接收过程已经结束后才允许进入禁止模式，因此 CPU 可以轮询 LPMACK 位来判断 FlexCAN 是否已经进入低功耗模式。LPMACK=0，FlexCAN 未被禁止。LPMACK=1，FlexCAN 处于低功耗模式。

D3~D0—MAXMB 位，最大报文缓冲区数目位。该位域定义可以参与匹配和仲裁的最大报文缓冲区数目。复位值为 0xF，即最大报文缓冲区数目为 16。只有当 FlexCAN 模块进入冻结模块时才允许修改该字段。注意：允许使用的最大报文缓冲区数目为 MAXMB+1。

2. FlexCAN 控制寄存器（CANCTRL）

CANCTRL 用于指定 FlexCAN 模块与 CAN 总线相关的属性，比如位速率参数，Rx 位采样点，是否启动闭环模式，是否启动监听模式，总线关闭是否自动恢复，是否开启总线关闭/错误中断等。除了 BOFFMSK、ERRMSK 和 BOFFREC 位可以在任意时刻访问外，该寄存器的其他字段只能在禁止或冻结模式下修改。该寄存器中与位速率相关的参数 PRES DIV、RJW、PSEG1、PSEG2、PROPSEG、CLK_SRC 位需要重点掌握。

CANCTRL 偏移地址为 0x1C_0004。D11~D8—未定义。

数据位	D31~D24	D23~D22	D21~D19	D18~D16	D15	D14	D13
定义	PRESDIV	RJW	PSEG1	PSEG2	BOFFMSK	ERRMS	CLK_SRC

数据位	D12	D7	D6	D5	D4	D3	D2	D1	D0
定义	LPB	SMP	BOFFREC	TSYN	LBUF	LOM	PROPSEG		

D31~D24—PRESDIV 位，预分频因子。该位域定义时钟源频率（ f_{sys} 或 EXTAL）和串行时钟 S_{clock} 频率（即 f_{Tq} ，参见 11.1.5 节）之间的比例。该寄存器最大值为 0xFF，因此 S_{clock} 频率的最小值等于时钟源频率/256。 S_{clock} 频率的计算公式如下：

$$S_{\text{clock}} \text{ 频率} = (f_{\text{sys}} \text{ 或 } \text{EXTAL}) / (\text{PRESDIV} + 1)$$

D23~D22—RJW 位，再同步补偿宽度。该位定义在一次再同步过程中允许补偿误差的最大值。有效值为 0~3。再同步补偿宽度 = $(\text{RJW} + 1) T_{\text{q}}$ 。

D21~D19—PSEG1 位，相位段 1。该位定义相位段 1 位时间长度。有效值为 0~7。相位段 1 = $(\text{PSEG1} + 1) T_{\text{q}}$ 。

D18~D16—PSEG2 位，相位段 2。该位定义相位段 2 位时间长度。有效值为 0~7。相位段 2 = $(\text{PSEG2} + 1) T_{\text{q}}$ 。

D2~D0—PROPSEG 位，传播时间段，定义发送阶段位时间长度，有效值为 0~7。传播时间段 = $(\text{PROPSEG} + 1) T_{\text{q}}$ 。

D13—CLK_SRC，时钟源选择位。该位选择 CAN 模块时钟源，供预分频使用。该位只能在禁止模式下修改。CLK_SRC=0，选择 EXTAL 引脚的外部晶振输出作为时钟源；CLK_SRC=1，使用内部总线时钟 f_{sys} 作为时钟源。当 CAN 总线时序容错性要求比较严格时，应该选择外部晶振输出。晶体振荡器时钟比 PLL 时钟性能更好。CAN 时钟发生机制如图 11-13 所示。

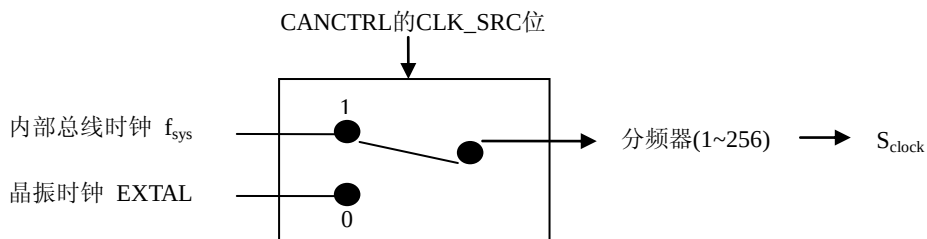


图11-13 CAN时钟发生机制

CAN 位速率计算公式如下：

$$\text{位速率} = f_{\text{Tq}} / (\text{Tq 总数})$$

T_{q} 总数=同步段+时间段 1（PROPSEG+PSEG1+2）+时间段 2（PSEG2+1）。其中，同步段固定为 1 个 T_{q} 。时间段 1 的取值范围为 4~16 个 T_{q} 。时间段 2 的取值范围为 2~8 个 T_{q} 。

D15—BOFFMSK 位，总线关闭中断掩码。该位控制总线关闭时是否允许中断。BOFFMSK=0，总线关闭禁止中断；BOFFMSK=1，总线关闭允许中断。

D14—ERRMSK 位, 错误中断掩码。该位控制发生错误时是否允许中断。ERRMSK=0, 禁止错误中断; ERRMSK=1, 允许错误中断。

D12—LPB, 闭环模式选择位。该位配置 FlexCAN 工作在闭环模式。闭环模式下, FlexCAN 执行内部循环用于自测。LPB=0, 禁止闭环模式。LPB=1, 使能闭环模式。

D7—SMP, 采样模式选择位。该位选择 FlexCAN 模块采样模式, 每个位时间采样一次还是采样三次来决定接收位的值。SMP=0, 在相位段 1 末尾采样一次, 用于决定接收位的值; SMP=1, 在相位段 1 末尾采样三次, 用于决定接收位的值。采样在 S_{clock} 的前两个周期内进行。

D6—BOFFREC 位, 总线关闭恢复模式。该位定义 FlexCAN 如何从总线关闭状态中返回。如果该位被清零, 则自动根据 CAN2.0B 协议规定从总线关闭状态中恢复。如果该位被置位, 总线关闭自动恢复被禁止。这种情况下, 总线关闭时, 除非通过用户复位, 否则该模块将保持总线关闭状态。如果在 CAN 总线上监测到 128 次连续 11 个 1 位之前该位被清零, 则总线关闭恢复, 就像 BOFFREC 从没有被置位过一样。如果该位在 CAN 总线上监测到 128 次连续 11 个 1 位之后被清零, 则 FlexCAN 通过等待 11 个连续 1 位来重新同步总线。该位清零后, 在总线关闭时该位可以被再次置位, 但是它只有在下一次进入总线关闭状态时才生效。如果在模块进入总线关闭状态时 BOFFREC 被清零, 在监测到 128 次出现 11 个 1 位之后, 允许的总线关闭恢复结点将回到主动错误状态。BOFFREC=0, 当总线关闭时, 自动根据 CAN2.0B 协议规定从总线关闭状态中恢复。BOFFREC=1, 禁止从总线关闭状态中自动恢复。

D5—TSYN 位, 计数器同步模式位。该位定义计数器的同步模式。使能 MB0 每接收到一个报文就复位自由运行计数器的机制。该机制为同步多个 FlexCAN 结点提供特定的 SYNC 报文 (全局网络时间)。TSYN=0, 禁止计数器同步; TSYN=1, 使能计数器同步。

注意: 使用这一机制, 在同一网络上的不同的 FlexCAN 模块每进行 4~5 个计数将会产生一位时钟偏移。

D4—LBUF 位, 报文缓冲区的发送次序位。LBUF=0, ID 最小的报文先发送; LBUF=1, 最小的报文缓冲区内的报文先发送。

D3—LOM 位, 监听模式位。该位配置 FlexCAN 工作在监听模式。LOM=0, FlexCAN 模块正常工作, 监听模式无效; LOM=1, FlexCAN 模块处于监听模式。

3. 接收掩码寄存器 (RXGMASK, RX14MASK, RX15MASK)

这 3 个寄存器用于过滤接收报文 ID。全局掩码寄存器 (RXGMASK), 用于过滤 MB0~MB13。MB14 掩码寄存器 (RX14MASK) 和 MB15 掩码寄存器 (RX15MASK), 分别用于过滤 MB14 和 MB15。掩码设置适用于标准帧和扩展帧。

全局掩码寄存器 (RXGMASK) 的偏移地址为 0x1C_0010, MB14 掩码寄存器 (RX14MASK) 的偏移地址为 0x1C_0014, MB 15 掩码寄存器 (RX15MASK) 的偏移地址为 0x1C_0018, 它们均是 32 位寄存器, D31~D29 均未定义。其他未定义如下:

D28~D18—标准 ID 掩码, 该位域用于设置是否对接收到报文的标准 ID 进行匹配, 复位全为 1。若某位为 1, 则需要对该位进行匹配, 否则忽略该位。

D17~D0—扩展 ID 掩码，该位域只适用于扩展帧格式，用于设置是否对接收到报文的扩展 ID 进行匹配，复位全为 1。若某位为 1，则需要对该位进行匹配，否则忽略该位。

注意：只能在冻结模式下修改掩码寄存器的值。否则，已通过验证但被锁存的帧被发送到相应的报文缓冲区时，与相应报文缓冲区的 ID 将不再匹配。下面给出一个报文匹配的例子来说明掩码的作用，RXGMASK、RX14MASK 和 RX15MASK 使用方法一致，只是作用的报文缓冲区有所不同。

假设 MB0、RXGMASK、总线上接收到报文帧 A 的设置如下：

MB0 ID	ID28.....ID18	ID17.....ID0
	1 1 1 1 1 1 1 1 0 0 0	
RXGMASK	1 1 1 1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1
报文帧A ID	1 1 1 1 1 1 1 1 0 0 1	

根据 RXGMASK 的设置可知：对于 MB0~13 而言，在对接收到的标准/扩展帧进行匹配时，由于 ID1~ID10、ID18 位均为 0，因此这些位不进行匹配；其他位为 1，因此其他位必须进行匹配；结合 MB0-ID 的设置可知，MB0 接收符合如下格式的帧：

ID设置	D28.....ID18	ID17.....ID0	说明
MB0接收帧ID	1 1 1 1 1 1 1 1 0 0 x		x=0

通过比较可知，报文帧 A 可被 MB0 接收。

4. 中断掩码寄存器（IMASK）/ 中断标志寄存器（IFLAG）

中断掩码寄存器 IMASK 包含每一个报文缓冲区的中断掩码。它控制每一个报文缓冲区在一次成功的发送/接收过程以后，是否允许产生中断。IMASK 的偏移地址为 0x1C_0028。D31~D16 位未定义。IMASK 其他位定义如下：

D15~D0—BUFnM，报文缓冲区中断掩码（n=0~15），使能/禁止 FlexCAN 报文缓冲区 n 发送/接收中断。BUFnM=0，禁止报文缓冲区 n 发送/接收中断。BUFnM=1，使能报文缓冲区 n 发送/接收中断。

中断标志寄存器 IFLAG 包含 16 个报文缓冲区的中断标志。每一次成功的发送/接收都置位 IFLAG 寄存器中相应位。如果相应的 IMASK 位被置位，将产生一个中断。写 1 清中断标志，写 0 无效。需要注意的是，如果当 IFLAG 的某位置位时，置位或清 IMASK 对应位可产生/消除中断请求。IFLAG 的基地址为 0x1C_0030。D32~D16 位未定义。IFLAG 其他位定义如下：

D15~D0—BUFnI，报文缓冲区 n 中断标志位（n=0~15），标志报文缓冲区 n 一次发送/接收成功。如果此时 IMASK 中的 BUFnM=1，则将产生相应的中断。用户必须写 1 来清中断标志。写 0 无效。BUFnI=0，报文缓冲区 n 没有接收/发送事件完成。BUFnI=1，报文缓冲区 n 成功进行了一次发送/接收。

注意：FlexCAN 模块共有 3 种中断源。16 个报文缓冲区分对应 16 个中断，它们通过逻辑或实现。因此，CPU 必须读 IFLAG 寄存器以决定具体是哪个报文缓冲区触发了中断。另外，FlexCAN 模块还有总线关闭中断和错误中断，它们也采样相同的机制处理，因此必须读 ERRSTAT 寄存器以决定具体发生了哪种类型的中断。

5. FlexCAN自由运行计数器寄存器（TIMER）

该寄存器内有一个 16 位自由运行的计数器，可以通过 CPU 进行读写。该寄存器复位值为 0x0000，计数到 0xFFFF 后回到 0x0000 重新开始计数。该计数器通过 FlexCAN 位时钟（位时钟决定 CAN 总线波特率）来计数。在报文发送/接收期间，每进行一次发送/接收，计数器加一。当总线上没有报文时，它按照编程设置的波特率来计数。在冻结模式下，计数器的值不改变。在 CAN 总线上，不管是收到什么类型的帧，该计数值总是在 ID 字段的起始处被捕获。捕获的值在报文被正确发送或接收后将被写入到的报文缓冲区的时间戳（Time Stamp）字段。写计数器是一个间接操作，数据首先被写入到一个辅助寄存器中，然后按照一定的时序执行一个内部请求/应答过程。因此需要一定的时间来将数据写入到寄存器中。除等待操作用户可见外，所有其他操作用户都是不可见的。如果需要，可以通过软件轮询该寄存器来检测数据何时被真正写入。TIMER 偏移地址为 0x1C_0008。16 位计数器使用的是低十六位（D15~D0），高 16 位未定义。

11.2.4 FlexCAN报文缓冲区（Message Buffer, MB）

FlexCAN 模块使用两个引脚 CANTX 和 CANRX 与其他 CAN 节点进行通信，其内部最重要的一个结构为报文缓冲区结构（Message Buffer Structure），FlexCAN 模块通过报文缓冲区进行报文帧的发送/接收。图 11-14 为 MCF52235 的 FlexCAN 模块结构框图。

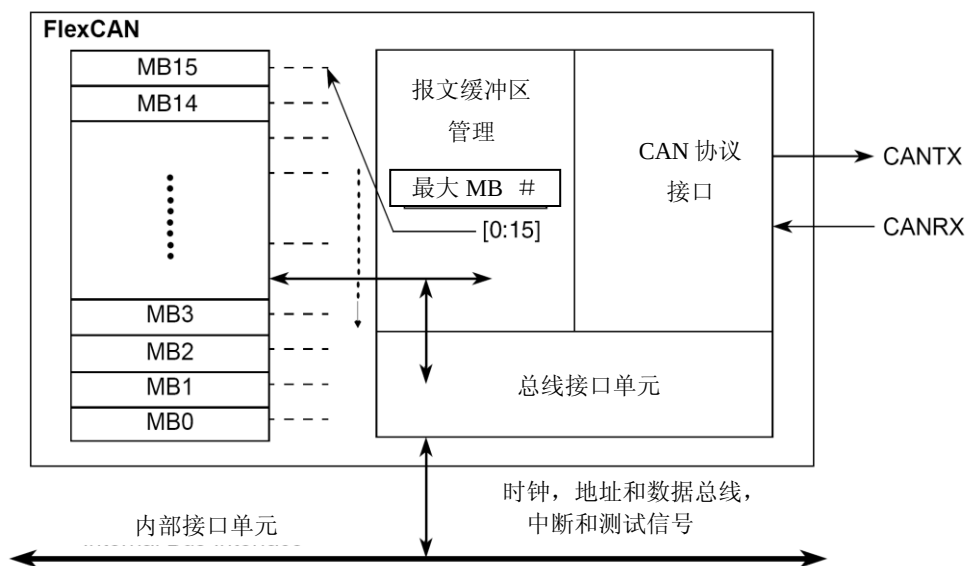


图11-14 MCF52235的FlexCAN模块结构框图

其中报文缓冲区(简称 MB)体系结构如图 11-15 所示：

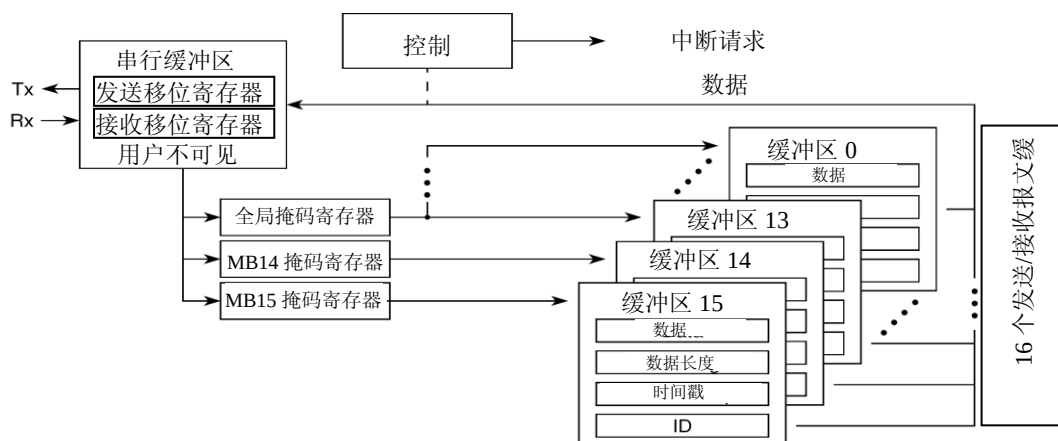


图11-15 FlexCAN报文缓冲区体系结构图

报文缓冲区 MB 内存映像为 FlexCAN 模块基地址 (0x1C_0000) + 0x80。16 个报文缓冲区共占用 256 字节，每一个报文缓冲区由 16 字节组成，其中包括 4 字节的控制/状态字段 (用于配置报文缓冲区)、4 字节的 ID 字段 (用于过滤接收报文) 以及 8 字节数据字段。报文缓冲区结构如图 11-16 所示。

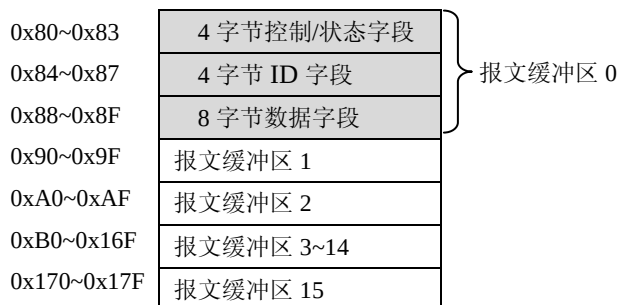


图11-16 报文缓冲区结构

CAN 2.0 B 定义了标准帧和扩展帧格式。标准帧 ID 有 11 位，扩展帧 ID 有 29 位，其中包括 11 位标准 ID 和 18 位扩展 ID。一个报文缓冲区格式如下：

D31~D29	D28	D27~D24	D23	D22	D21	D20	D19~D16	D15~D0
		CODE		SRR	IDE	RTR	LENGTH	Time Stamp
标准 ID[28:18]							扩展 ID[17:0]	
数据字节 0			数据字节 1			数据字节 2 数据字节 3		
数据字节 4			数据字节 5			数据字节 6 数据字节 7		

下面是各控制状态字段描述 (D31~D28, D23 未定义)：

D27~D24—CODE，报文缓冲区控制码。该域可以通过 CPU 或 FlexCAN 模块进行读写。通过设置该字段，每个 MB 都可以被独立配置成发送/接收报文缓冲区 (Tx/Rx MB)，并支持发送与接收标准/扩展数据帧或远程帧。通过 CODE 配置 MB 为 Tx/Rx MB 的方法

见表 11-3 和 11-4 所示。(表 11-5 中用到的 RTR 位,请见下文 D20—RTR 描述)

表 11-3 Tx MB CODE

MBn [RTR]	初 始 Tx CODE	成 功 发 送 后 CODE(由 CAN 更 新后)	描述
X	1000	—	空闲: 报文缓冲区尚未准备好进行数据发送并参与仲裁。
0	1100	1000	无条件发送数据帧一次。发送完成后, MB 自动回到空闲状态。
1	1100	0100	无条件发送一次远程帧。远程帧发送完以后, 发送报文缓冲区立即变为接收报文缓冲区, 接收具有相同 ID 的数据帧。
0	1010	1010	当接收到一个具有相同 ID 的远程请求帧时发送一个数据帧。同时, 该报文缓冲区参与匹配和仲裁处理。匹配处理将到来的远程请求帧的 ID 与 MB 的 ID 进行比较, 如果匹配, 则报文缓冲区允许参与当前仲裁处理且 CODE 域被自动更新为 1110, 从而允许 MB 参与进一步仲裁。当该帧最终被成功发送后, CODE 自动回到 1010 重新启动传输。
0	1110	1010	这是个中间代码, 它作为远程请求帧匹配的结果自动写入到报文缓冲区。无条件发送一次数据帧, 然后 CODE 自动回到 1010。CPU 写该码也具有相同效果。

表 11-4 Rx MB CODE

接收新帧 前 CODE	描述	接收新帧 后 CODE	描述
0000	空闲: MB 未被激活	—	MB 未参与匹配处理
0100	EMPTY: MB 被激活且 为空	0010	MB 参与匹配处理。当一个帧被成功接收时, CODE 自动更新为 FULL。
0010	FULL: MB 满	0010	解锁 MB 后, 读控制/状态字段不能使 CODE 回 到 EMPTY, 它依然是 FULL。在控制/状态字段 已读且 MB 被解锁以后, 如果一个新的帧被写 入到 MB, CODE 依然是 FULL。
		0110	如果 MB 是 FULL, 且一个新的帧需要在 CPU 读 已接收帧之前被写入到这个 MB, 则 MB 已接收 帧将被覆盖, 且 CODE 自动更新为 OVERRUN。
0110	OVERRUN: 一个帧被写 入到一个已满缓冲区中	0010	如果 CODE 显示为 OVERRUN, 但是 CPU 读控 制/状态字段后解锁 MB, 则当一个新帧被写入 到 MB 时, CODE 回到 FULL。
		0110	如果 CODE 显示为 OVERRUN, 且有另一个新的 帧必须被写入, 则 MB 将再次被重写, 且 CODE 将保持 OVERRUN。
0XY1	BUSY: FlexCAN 模块正 在用新接收到的帧来更 新 MB, 此时 CPU 不应该 试图访问 MB。	0010	一个 EMPTY 缓冲区被写入一个新帧 (XY 为 01)。
		0110	一个 FULL/OVERRUN 缓冲区被重写 (XY 为 11)。

D22—SRR, 远程发送请求替代位。该位总是 1, 且只适用于扩展帧格式。在扩展帧格式中 SRR=0 无效。在发送过程中, 发送报文缓冲区必须将该位置 1。在接收过程中, 接收报文缓冲区接收并保存该位的值。该位可以被接收为 0 或 1。如果 FlexCAN 接收到

该位为 0，则认为是仲裁失败。当一扩展帧和一标准帧同时出现在网络并且扩展帧的高 11 位与标准帧相同时，若该标准帧为数据帧，则 RTR 为 0 而对应的扩展帧中 SSR 位为 1，于是标准帧竞争仲裁获胜继续发送而扩展帧对应节点仲裁失败转为接收节点；若该标准帧为远程帧，RTR 位与 SRR 位同为 1，不存在冲突两帧继续发送 IDE 位，标准帧中 IDE 为 0 而扩展帧中 IDE 为 1，同样标准帧仲裁获胜继续占用总线而扩展帧对应的节点仲裁失败转为接收器。这也就是标准帧的优先级总高于扩展帧的原因。

D21—IDE，ID 扩展位。该位标志一个帧是标准帧还是扩展帧。IDE=0，标准帧格式。IDE=1，扩展帧格式。所有的报文缓冲区在进行报文过滤时都必须匹配 IDE 位。

D20—RTR，远程发送请求位。该位用于请求发送一个数据帧。如果 FlexCAN 发送该位为“1”，但是接收到为“0”，则认为是仲裁丢失。如果发送该位为“0”，但是接收到为“1”，则认为是位错误。如果该位发送的值与接收到的值相等，则认为是一次成功的发送。RTR=0，标志当前报文缓冲区有待发送的数据帧。RTR=1，标志当前报文缓冲区有待发送的远程帧。

D19~D16—LENGTH，数据长度位。该位标志接收/发送数据字节数。接收数据时，接收帧的数据长度码（DLC）被写入到该字段中。DLC 由 CAN 协议定义，它的值由帧的实际数据长度来决定。发送数据时，该位域由 CPU 写入，且作为待发送帧的数据长度码（DLC）被使用。当 RTR 置位时，待发送的帧是一个没有数据（DATA）字段的远程帧，这时 LENGTH 被忽略。

D15~D0—Time Stamp，时间戳。FlexCAN 使用时间戳来达到同步的目的。在接收报文时，当 CAN 总线上出现 ID 字段时，FlexCAN 模块保存自由运行定时器 TIMER 中的值。如果成功接收到一条报文，并将报文从串行缓冲区（见图 11-14）放入接收缓冲区，则保存的值将被写入到接收缓冲区中。当发送报文时，在报文被成功发送后，TIMER 的值将被写入到发送报文缓冲区的 Time Stamp 域中。在 MB0 接收到帧时允许将 TIMER 复位。这一特性可以用于完成网络时间同步。详情请参见 CANCTRL 的 TSYN 位。

下面是 ID 字段描述：

D31~D29 未定义。D28~D0—ID 字段。在标准帧格式中只有高 11 位（D28~D18）用于接收/发送匹配，低 18 位（D17~D0）被忽略。在扩展帧格式中，高 11 位和低 18 位都用于接收/发送匹配。D28~D18 用于标准帧 ID 匹配，D17~D0 用于扩展帧 ID 匹配。

下面是数据域字段描述：

D31~D0—DATA 数据域。一个数据帧可包含多达 8 个字节。接收帧时，当从 CAN 总线上接收到数据时数据被保存到该位域中。发送帧时，CPU 将该位域数据提供给发送帧进行发送。

11.3 MCF52235 FlexCAN 模块报文发送与接收函数设计

FlexCAN 模块共有 16 个报文缓冲区（MB），每个 MB 都可以被独立配置成发送/接收报文缓冲区，支持发送或接收标准/扩展数据帧和远程帧。每个 MB 可包含多达 8 字节数据，每个 MB 对应一个中断标志位用以标志该 MB 发送/接收是否完成。MB 发送报文的

优先级，由仲裁算法根据 MB 序号或报文 ID 来决定。报文匹配算法保证报文缓冲区只接收并保存与报文缓冲区 ID 相匹配的报文。掩码机制使得 MB 可以与 ID 在一定范围内的 CAN 帧相匹配，而不是只接收与 MB 的 ID 相等的报文。数据一致性机制用来保证 CPU 操作 MB 数据的完整性。

在进行功能描述之前，我们先来明确一个概念。如果某一报文缓冲区在某一时刻参与当时的匹配和仲裁算法则称为被激活。如果报文缓冲区的 CODE 域为 0000，那么该报文缓冲区被配置为接收缓冲区，且为空闲状态，没有被激活。相似地，如果报文缓冲区的 CODE 域为 1000，那么该报文缓冲区被配置为发送缓冲区，且为空闲状态，没有被激活。当 CPU 写 MB 的控制/状态字段时，没有被设置为 0000 或 1000 的 MB 是暂时停止的，不参与仲裁/匹配。由于 FlexCAN 模块发送/接收数据帧和远程帧的处理过程有所不同，所以下面分别来进行介绍。

11.3.1 数据帧发送/接收

1. 数据帧发送的过程

FlexCAN 模块 16 个报文缓冲区 (MB) 中的任意一个都可以被配置成发送缓冲区，且每个 MB 每次最多可以发送 8 字节数据。以第 0 个缓冲区 MB0 为例 (其他 MB 的配置与 MB0 配置方法相同)，将 MB0 配置为发送缓冲区并进行标准/扩展数据帧发送的步骤是：

(1) 写 MB0 控制/状态字段的 4 位 CODE 域=1000，配置 MB0 为发送报文缓冲区且为空闲状态；

(2) 如果要通过 MB0 发送一个扩展帧则需进行如下设置：写标准 ID 和扩展 ID 到 MB0 的 ID 字段；写待发送数据字节到 MB0 的数据字段；将 MB0 控制/状态字段的 IDE 位和 SRR 位置位；将待发送数据长度写入 MB0 控制/状态字段的 4 位 LENGTH 域。如果要通过 MB0 发送一个标准帧则需进行如下设置：写标准 ID 到 MB0 的标准 ID 字段；写 MB0 的扩展 ID 字段为 0；写待发送数据字节到 MB0 的数据字段；将待发送数据长度写入 MB0 控制/状态字段的 4 位 LENGTH 域。

(3) 写 MB0 控制/状态字段的 RTR 位=0，4 位 CODE 域=1100，激活报文缓冲区，启动数据帧发送。

(4) 由于发送过程需要一定时间，所以需要延时一定时间然后再判断发送是否完成。一旦 MB0 发送数据帧成功则中断标志寄存器 IFLAG 中的 BUF0I 位 (即 MB0 发送完成标志位) 将被置位。此时若允许中断 (中断掩码寄存器 IMASK 的 BUF0M 位被置位)，将触发 MB0 中断。

(5) 一次发送完成，需清中断标志寄存器 IFLAG 中 BUF0I 位。

2. 数据帧发送的函数设计

```
//CANSendData:发送一个数据帧-----*
//功 能:设置一个报文缓冲区为发送缓冲区并无条件发送一个数据帧      *
//参 数:                                                              *
//          (1)buffer:待配置报文缓冲区编号(0~15)                      *
```

```

//      (2)stdID: 待发送数据帧的标准ID(0~0x7EF)                                *
//      (3)extID: 待发送数据帧的扩展ID(0~0x3FFFF)                            *
//      (4)ide: 标准帧或者扩展帧(ide=0, 标准帧. ide=1, 扩展帧)                *
//      (5)length: 待发送数据长度(0~8, 0代表为空)                            *
//      (6)data: 待发送数据缓冲区首地址                                       *
// 返 回: 发送是否成功 0:失败 1: 成功                                          *
// 注 意: 11位标准ID的D10~D4不能全为1                                       *
//-----*
uint8 CANSendData(uint8 buffer, uint16 stdID, uint32 extID, uint8 ide, uint8 length, uint8*
data)
{
    uint32 i=0;
    //1 设置缓冲区为发送缓冲区且为非激活状态
    MCF_CAN_MB_CTRL(buffer) = MCF_CAN_MB_CTRL_CODE(8);
    //2. 数据帧设置
    //2.1扩展帧设置
    if(ide)
    {
        //ID设置
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                |MCF_CAN_MB_ID_EXTID(extID));
        CANLoadBufferData(buffer, length, data); //加载数据到数据域
        //控制状态字设置
        MCF_CAN_MB_CTRL(buffer) |= (MCF_CAN_MB_CTRL_SSR
                                    |MCF_CAN_MB_CTRL_IDE
                                    |MCF_CAN_MB_CTRL_LENGTH(length));
    }
    //2.2标准帧设置
    else
    {
        //ID设置
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                |MCF_CAN_MB_ID_EXTID(0));
        CANLoadBufferData(buffer, length, data); //加载数据到数据域
        //控制状态字设置
        MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_LENGTH(length);
    }
    //3. 使能远程帧发送请求, CODE=1100, RTR=0 无条件发送一次数据帧
    MCF_CAN_MB_CTRL(buffer) &= ~MCF_CAN_MB_CTRL_RTR;
    MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_CODE(12);
    //4. 限时等待发送完成(如果使用中断则限时等待语句可删除)
    while (!(MCF_CAN_IFLAG & (1<<buffer)))
    {
        if ((i++)==0x10000)
        {
            return 0;
        }
    }
}

```

```

//5. 清报文缓冲区中断标志
MCF_CAN_IFLAG = (1<<buffer);
return 1;
}

```

3. 数据帧接收的过程

MCF52235 FlexCAN 模块的 16 个报文缓冲区 (MB) 也可以独立配置成接收缓冲区, 并支持标准/扩展帧接收。

仍以 MB0 为例 (其他 MB 的配置与 MB0 配置方法相同), 将 MB0 配置为接收缓冲区并进行标准/扩展帧接收的步骤如下:

(1) 写该 MB0 的控制/状态字段的 4 位 CODE 域=0000, 配置 MB0 为接收缓冲区且为空闲状态;

(2) 如果要通过 MB0 接收一个扩展帧, 则首先写标准 ID 和扩展 ID 到 MB0 的 ID 字段, 然后将控制/状态字段的 IDE 位和 SSR 位置位。如果要发送一个标准帧, 则需写标准 ID 到 MB0 的标准 ID 字段, 并写 MB0 的扩展 ID 字段为 0;

注意: 接收报文缓冲区的 ID 字段将结合 RXGMASK 的设置, 及其控制/状态字段的 IDE 位来过滤报文。(ID、IDE 设置与报文过滤请参见 11.2.3 中“标准/扩展报文掩码实例”)。

(3) 写 MB0 的控制/状态字段的 4 位 CODE 域=0100, 激活接收缓冲区, 启动接收。

(4) 一旦 MB0 接收到数据, 则中断标志寄存器 IFLAG 中的 BUF0I 位将被置位, 此时若允许中断 (中断掩码寄存器 IMASK 的 BUF0M 位被置位) 将触发 MB0 中断, 用户可以通过查询相应标志位, 或通过中断读取接收报文缓冲区数据。

4. 数据帧接收的函数设计

```

//CANEnableRXBuffer:配置报文缓冲区为接收报文缓冲区-----*
//功 能:设置一个报文缓冲区为接收报文缓冲区                                     *
//参 数:                                                                           *
//      (1)buffer:待配置报文缓冲区编号(0~15)                                   *
//      (2)stdID: 待发送数据帧的标准ID(0~0x7EF)                               *
//      (3)extID: 待发送数据帧的扩展ID(0~0x3FFFF)                             *
//      (4)ide:标准帧或者扩展帧(ide=0, 标准帧. ide=1, 扩展帧)                 *
//返 回: 无                                                                           *
//-----*
void CANEnableRXBuffer(uint8 buffer, uint16 stdID, uint32 extID, uint8 ide)
{
    //1. 使报文缓冲区为接收缓冲区且处于非激活状态
    MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_CODE(0);
    //2. ID设置
    if (ide)
    {
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                |MCF_CAN_MB_ID_EXTID(extID));
        MCF_CAN_MB_CTRL(buffer) = (MCF_CAN_MB_CTRL_IDE
                                |MCF_CAN_MB_CTRL_SSR);
    }
}

```

```

    }
else
{
    MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID));
}
//3. 使能MB为RX缓冲区
MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_CODE(0x04);
}
//CANReadBufferData:读取接收报文缓冲区数据-----*
//功 能:读取接收报文缓冲区数据                      *
//参 数: (1)buffer:待读取数据的报文缓冲区            *
//        (2)data:存入数据的数据缓冲区首址          *
//        (3)length:接收到的数据长度                  *
//        (4)ide:接收到帧的类型 *ide=0, 接收到标准帧.*ide=1, 接收到扩展帧 *
//        (5)stdID:接收到的数据帧的标准ID             *
//        (6)extID:接收到的数据帧的扩展ID             *
//返 回: 接收数据是否成功 0:失败 1:成功              *
//-----*
uint8 CANReadBufferData(uint8 buffer, uint8* data, uint8 *length, uint8 *ide,
                        uint16 *stdID, uint32 *extID)
{
    uint8 u8Counter;
    uint32 i=0;
    //1. 限时等待完成(如果使用中断则限时等待语句可删除)
    while (!(MCF_CAN_IFLAG & (1<<buffer)))
    {
        if ((i++)==0x10000)
        {
            return 0;
        }
    }
    //2. 帧类型、帧ID以及数据
    (*ide)= (MCF_CAN_MB_CTRL(buffer) & MCF_CAN_MB_CTRL_IDE)>>21;
    (*length)=(MCF_CAN_MB_CTRL(buffer) & MCF_CAN_MB_CTRL_LENGTH(0xF))>>16;
    (*stdID)=(MCF_CAN_MB_ID(buffer) & MCF_CAN_MB_ID_STDID(0x7FF))>>18 ;
    (*extID)=(MCF_CAN_MB_ID(buffer) & MCF_CAN_MB_ID_EXTID(0x3FFFF));
    for(u8Counter=0;u8Counter<(*length);u8Counter++)
        data[u8Counter]=MCF_CAN_MB_DATA(buffer, u8Counter);
    //3. 清报文缓冲区中断标志
    MCF_CAN_IFLAG = (1<<buffer);
    return 1;
}

```

11.3.2 远程帧发送/接收

在 CAN 通信系统中，作为数据接收的 CAN 节点，可以借助于发送远程帧来启动其资源节点向它发送数据。MCF52235 FlexCAN 模块报文缓冲区远程帧发送/接收配置有以

下两种方式：

(1) 无条件发送一次远程帧。远程帧发送完以后，发送报文缓冲区立即变为接收报文缓冲区，接收具有相同 ID 的数据帧。

(2) 当接收到一个与其具有相同 ID 的远程请求帧时，发送一个帧作为应答。如果其控制/状态字段的 RTR=0，则发送一个数据帧作为应答。否则，发送一个远程帧作为应答。

当配置为 (1) 方式时，用户首先初始化一个报文缓冲区为发送报文缓冲区，然后置位 RTR 来标志发送一个远程帧。当远程帧发送成功后，发送报文缓冲区自动变为一个与被发送的远程帧具有相同 ID 的接收报文缓冲区。

当配置为 (2) 方式时，FlexCAN 模块接收到一个远程帧时，远程帧的 ID 与所有 CODE=1010 的报文缓冲区的 ID 相比较。如果 ID 匹配，则发送一个数据帧/远程帧（根据 RTR 位决定）作为应答。需要注意的是，FlexCAN 模块接收的远程帧不被保存在接收报文缓冲区中，它只用于自动触发帧发送。掩码寄存器不用于远程帧 ID 匹配。CAN 总线上到达的接收帧除 RTR 位之外的所有的仲裁位，都必须与触发它发送的远程帧匹配，以作为回应。相匹配的报文缓冲区立即进入内部仲裁处理，但是它被视为一个普通的发送报文缓冲区，并不具有更高的优先权。应答帧的数据长度取决于启动它发送的远程帧的请求数据长度。

1. 远程帧发送的过程

MCF52235 FlexCan 模块有 16 个报文缓冲区 (MB)，每个报文缓冲区都可以独立配置成发送缓冲区，可发送数据帧/远程帧，并支持发送标准帧/扩展帧。以 MB0 为例，其他 MB 的配置与 MB0 配置方法相同，将 MB0 配置为发送缓冲区并进行标准/扩展远程帧发送的步骤如下：

(1) 写 MB0 控制/状态字段的 4 位 CODE 域=1000，配置 MB0 为发送报文缓冲区且为空闲状态；

(2) 如果 MB0 发送一个扩展远程帧则需进行如下设置：写标准 ID 和扩展 ID 到 MB0 的 ID 字段；将 MB0 控制/状态字段的 IDE 位和 SRR 位置位；写远程请求数据字节长度写入 MB0 控制/状态字段的 4 位 LENGTH 域。

如果要发送一个标准远程帧则进行如下设置：写标准 ID 到 MB0 的标准 ID 字段；写 MB0 的扩展 ID 字段为 0；将远程请求数据字节长度写入 MB0 控制/状态字段的 4 位 LENGTH 域。

(3) 写控制/状态字段的 RTR=1，写 4 位 CODE 域=1010，激活报文缓冲区启动远程帧发送。

(4) 由于发送过程需要一定时间，所以需要延时一定时间后再判断发送是否完成。一旦 MB0 发送远程帧成功，则中断标志寄存器 IFLAG 中的 BUF0I 位（即 MB0 发送完成标志位）将被置位。此时若允许中断（即中断掩码寄存器 IMASK 的 BUF0M 位被置位），将触发 MB0 中断。一旦发送完毕，MB0 将变为接收具有相同 ID 的接收报文缓冲区。

(5) 一次发送完成，需清中断标志寄存器 IFLAG 中 BUF0I 位。

2. 远程帧发送的函数设计

```

//CANSendRemote:发送一个远程请求帧-----*
//功 能:设置一个报文缓冲区用于发送一个远程请求帧, 发送完成后该报文缓冲区立*
//      刻变为接收具有相同ID的接收报文缓冲区*
//参 数:                                     *
//      (1)buffer:待配置报文缓冲区编号(0~15)*
//      (2)stdID: 待发送数据帧的标准ID(0~0x7EF)*
//      (3)extID: 待发送数据帧的扩展ID(0~0x3FFFF)*
//      (4)length:请求的数据个数(1~8)*
//      (5)ide:标准帧或者扩展帧(ide=0, 标准帧; ide=1, 扩展帧)*
//返 回: 发送是否成功 0:失败 1:成功*
//-----*
uint8 CANSendRemote(uint8 buffer, uint32 stdID, uint32 extID, uint8 length, uint8 ide)
{
    uint32 i=0;
    //1. 设置缓冲区为发送缓冲区且为非激活状态
    MCF_CAN_MB_CTRL(buffer) = MCF_CAN_MB_CTRL_CODE(8);
    //2. 数据帧设置
    //2.1扩展帧配置
    if(ide)
    {
        //ID配置
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                |MCF_CAN_MB_ID_EXTID(extID));

        //控制状态字设置
        MCF_CAN_MB_CTRL(buffer) |= (MCF_CAN_MB_CTRL_SSR
                                |MCF_CAN_MB_CTRL_IDE
                                | MCF_CAN_MB_CTRL_LENGTH(length));
    }
    //2.2标准帧配置
else
    {
        //ID配置
        MCF_CAN_MB_ID(buffer) = MCF_CAN_MB_ID_STDID(stdID);
        //控制状态字设置
        MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_LENGTH(length) ;
    }
    //3. 使能远程帧发送请求, CODE=12, RTR=1代表无条件发送一次远程帧, 然后立刻
    // 变为接收具有相同ID的接收报文缓冲区
    MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_RTR;
    MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_CODE(0x0C);
    //4. 限时等待发送完成(如果使用中断则限时等待语句可删除)
    while (!(MCF_CAN_IFLAG & (1<<buffer)))
    {
        if ((i++)==0xFFFFF)
        {
            return 0;
        }
    }
}

```

```

    }
    //5. 清报文缓冲区中断标志
    MCF_CAN_IFLAG = (1<<buffer);           //清发送完成标志, 写1清标志位
    return 1;
}

```

3. 远程帧接收的过程

仍以 MB0 为例, 其他 MB 的配置与 MB0 配置方法相同, 将 MB0 配置为应答一个远程帧的报文缓冲区的步骤如下:

(1) 写 MB0 控制/状态字段的 4 位 CODE 域=1000, 配置 MB0 为发送报文缓冲区且为空闲状态。

(2) 如果设置应答帧为数据帧, 则将应答帧数据写入 MB0 的数据字段。

(3) 写允许应答的标准/扩展远程帧的 ID 到 MB0 的 ID 字段; 如果应答数据帧为扩展帧则将控制/状态字段的 IDE 位和 SSR 位置位。将应答数据帧长度写入 MB0 控制/状态字段的 4 位 LENGTH 域。

(4) 写控制/状态字段的 RTR=0 或 1, 写 4 位 CODE 域=1010, 激活报文缓冲区, 当收到 ID 匹配的远程帧时则启动发送数据帧/远程帧作为应答。应答帧将被发送远程帧的报文缓冲区所接收。

4. 远程帧接收的函数设计

```

//CANEnableRemoteBuffer: 设置一个报文缓冲区响应一个远程帧-----*
//功 能: 设置一个报文缓冲用于响应一个远程帧                                     *
//参 数:                                                                           *
//      (1)buffer: 待配置报文缓冲区编号 (0~15)                                   *
//      (2)stdID: 待发送数据帧的标准ID (0~0x7EF)                                *
//      (3)extID: 待发送数据帧的扩展ID (0~0x3FFFF)                             *
//      (4)length: 待发送数据长度                                                *
//      (5)ide: 标准帧或者扩展帧 (ide=0, 标准帧. ide=1, 扩展帧)                 *
//      (6)rtr: 发送一个数据帧/远程帧作为应答 (rtr=0, 发送一个数据帧作为应答*
//            rtr=1 发送一个远程帧作为应答)                                     *
//      (7)data: 数据缓冲区首址                                                 *
//-----*
void CANEnableRemoteBuffer(uint8 buffer, uint32 stdID, uint32 extID,
                           uint8 length, uint8 ide, uint8 rtr, uint8* data)
{
    //1. 使报文缓冲区为发送缓冲区且处于非激活状态
    MCF_CAN_MB_CTRL(buffer) = MCF_CAN_MB_CTRL_CODE(8);
    //2. 加载应答数据帧数据
    if (rtr==0)
        CANLoadBufferData(buffer, length, data);
    //3. ID配置
    if (ide)
    {
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                |MCF_CAN_MB_ID_EXTID(extID));
    }
}

```

```

        MCF_CAN_MB_CTRL(buffer) = (MCF_CAN_MB_CTRL_LENGTH(length)
                                     |MCF_CAN_MB_CTRL_IDE
                                     |MCF_CAN_MB_CTRL_SSR);
    }
    else {
        MCF_CAN_MB_ID(buffer) = (MCF_CAN_MB_ID_STDID(stdID)
                                   |MCF_CAN_MB_ID_EXTID(0));
        MCF_CAN_MB_CTRL(buffer) = (MCF_CAN_MB_CTRL_LENGTH(length));
    }
    //4. 使能缓冲区响应一个远程帧, CODE=0x0A, RTR=0, 当收到具有相同ID的远程报文时发送
    // 一个数据帧, RTR=1, 当收到具有相同ID的远程报文时发送一个远程帧
    if(rtr==0)
        MCF_CAN_MB_CTRL(buffer) &=~MCF_CAN_MB_CTRL_RTR;
    else
        MCF_CAN_MB_CTRL(buffer) |=MCF_CAN_MB_CTRL_RTR;
    MCF_CAN_MB_CTRL(buffer) |= MCF_CAN_MB_CTRL_CODE(0x0A);
}

```

11.3.3 仲裁处理、匹配处理及报文缓冲区管理

1. 报文缓冲区管理

报文缓冲区管理 (Message Buffer Managing, MBM) 是为了保证数据的一致性和 FlexCAN 操作的正确性。为对报文实现双缓冲, FlexCAN 设置了两个串行报文缓冲区 (Serial Message Buffer, SMB)。这两个缓冲区用于存放 FlexCAN 接收到的报文和待发送的报文。某一时刻只能有一个串行报文缓冲区处于激活状态。

1) 报文缓冲区失活

在当前匹配/仲裁期间, 任何 CPU 写 MB 的控制/状态字段都将使得 MB 从发送/接收过程中移除。这种机制称为 MB 失活。它是暂时性的, 只影响当前匹配/仲裁周期。失活的目的在于保证数据的一致性。在匹配/仲裁处理时扫描所有 MB 决定哪一个 MB 进行发送或接收。如果 CPU 在仲裁期更新 MB, 将造成 MB 的数据不一致, 因此 MB 需要失活。但是即使采用失活的机制, 不在冻结模式下时, 写已激活 MB 的控制/状态字段也可能导致意想不到的结果。因此如果需要改变某一个被激活的 MB 的控制/状态字段, 建议先使模块进入冻结模式后再进行修改。

2) 锁定和释放报文缓冲区

当 CPU 读一个已激活且不为空的接收报文缓冲区的控制/状态字段时, FlexCAN 假定 CPU 想要以一种原子操作读整个报文缓冲区, 然后置要读报文缓冲区的内部锁定标志, 这种操作称为锁定。CPU 读自由运行计数器 (全局解锁操作), 或读另一个报文缓冲区的控制/状态字段时将解除锁定, 这种操作称为释放。报文缓冲区的锁定是为了防止当 CPU 读某一个报文缓冲区时, 一个新的帧被写入到缓冲区中, 造成数据丢失。锁定机制只适用于接收报文缓冲区, 即 CODE 字段不是 0000 (MB 未被激活) 或 0100 (MB 被激活但为

空) 的报文缓冲区。

2. 仲裁处理

仲裁处理是报文缓冲区管理执行的一个算法, 它扫描整个 MB 内存, 寻找具有最高优先级的报文并发送。所有被设置为发送缓冲区的 MB 都会被扫描, 用于确定最小的 ID 或者最小的 MB 序号。具有最小 ID 的报文还是最小序号 MB 内的报文优先发送, 取决于 CANCTRL 的 LBUF 位。

仲裁处理发生在以下过程中: CAN 帧的 CRC 校验过程; CAN 帧的错误界定过程; 在仲裁中获胜的 MB 失活的间隙, 或没有 MB 需要发送, 但是在前一次仲裁结束后 CPU 写了某一个 MB 的控制/状态字段的间隙; 报文缓冲区管理 (MBM) 空闲或者总线关闭且 CPU 写了某一个 MB 的控制/状态字段时; 冻结模式退出过程。

当具有最高优先级的 MB 被选中后, 它被放入称为串行报文缓冲区 (SMB) 的临时存储空间中, 它与正常的报文缓冲区具有相同的结构, 但是对用户不可见。该操作称为移出。

3. 匹配处理

匹配处理是扫描整个 MB 内存, 寻找与 CAN 总线上接收到的帧的 ID 相匹配的接收缓冲区的一个算法。只有被设置为接收的 MB 才参与匹配。CAN 总线上的帧的 ID、DLC 和数据字段都被检索, 并暂时保存在串行报文缓冲区 SMB 中。匹配处理发生在 CRC 字段。如果找到一个匹配的 MB, 则在 CAN 协议的帧结束的第 6 个位时间, SMB 的内容被移到匹配的 MB 中。该操作称为移入。如果这期间检测到任何协议错误 (CRC 或 ACK 等), 移入操作都不会发生。具有匹配 ID 的 MB 在没有被锁定 (即 CODE=0100、0010、或 0110 的 MB, 或 CPU 读 MB 的控制/状态字段解锁了的 MB) 时, 还可以接收新的帧。

通过使用 ID 接收掩码, 可以匹配一系列 ID。FlexCAN 通过 3 个掩码寄存器 (RXGMASK、RX14MASK 和 RX15MASK) 来支持掩码机制。在匹配算法中, 如果掩码位被置位, 则相应 ID 位被验证。反之, 则相应 ID 位不进行验证。

11.4 MCF52235 FlexCAN 模块编程实例

11.4.1 初始化函数设计

1. FlexCAN 初始化过程

FlexCAN 的初始化包括复位以后报文缓冲区的初始化配置和 CAN 通信参数的配置, 以及在操作过程中可能需要再次配置的选项。FlexCAN 可通过以下三种方式来复位:

- (1) 设备级硬件复位—同步复位所有的内存映像寄存器
- (2) 设备级软件复位—同步复位部分内存映像寄存器
- (3) CANMCR 的 SOFTRST 位—与设备级软件复位等效

由于复位是同步的且需要按照一定的时序执行一个请求/应答程序, 因此它需要一定

的时间来完成。复位过程中 SOFTRST 位保持为 1，复位完成后自动清零。用户可以通过查询该位来判断软复位是否结束。注意，任意低功耗模式下的时钟关闭都将导致软复位失败，因此在执行软复位前必须退出低功耗模式且重启时钟。

当 FlexCAN 模块处于禁止模式时，通过设置 CANCTRL 的 CLK_SRC 选择时钟源、清零 CANMCR 的 MDIS 位使能 FlexCAN 以后，FlexCAN 自动进入冻结模式。在冻结模式下，FlexCAN 与 CAN 总线不同步。CANMCR 寄存器的 HALT 和 FRZ 位置位，内部状态机被禁止，且 CANMCR 寄存器的 FRZACK 和 NOTRDY 位被置位。CAN_{TX} 引脚变为 1 状态且 FlexCAN 不能进行任何 CAN 帧的发送/接收。由于复位不影响报文缓冲区，所以它不能被自动初始化。

对 FlexCAN 模块的任何配置改变或初始化操作，都必须在冻结模式下进行。下面是通用的初始化配置步骤：

(1) 初始化 CANCTRL 寄存器中的操作模式：①设置位时序参数：PROPSEG（传播时间段）、PSEG1（相位段 1）、PSEG2（相位段 2）、RJV（再同步补偿宽度）；②通过对 PRESDIV 域进行编程设置时钟；③通过 LBUF 位选择内部仲裁模式。

(2) 初始化报文缓冲区：①设置控制/状态字段，设置所有报文缓冲区为激活或非激活状态；②根据需要初始化每一个报文缓冲区的所有其他项。

(3) 根据接收掩码的需要初始化 RXGMASK（全局掩码寄存器，适用于缓冲区 0-13）、RX14MASK（缓冲区 14 独立掩码寄存器）和 RX15MASK（缓冲区 15 独立掩码寄存器）

(4) 初始化 FlexCAN 中断处理：①根据所需中断，初始化中断控制寄存器；②在 IMASK（中断掩码寄存器）中设置相应掩码位、相关报文缓冲区中断和 CANCTRL（总线开关和错误中断）

(5) 清 CANMCR.HALT（FlexCAN 配置寄存器的 HALT 位）。此时 FlexCAN 试图同步 CAN 总线。

2. FlexCAN 初始化函数设计

下面给出初始化配置函数的内容。限于篇幅，其他程序请参见随书光盘。

```
//CANInit:CAN通信初始化-----*
//功 能:CAN初始化                                     *
//参 数:MaxMBnum:允许使用的报文缓冲区数目(范围:0~15)   *
//返 回:无                                              *
//说 明:在25M 外部晶振频率下将总线速率设为100kbps      *
//-----*
void CANInit(uint8 MaxMBnum)
{
    uint8 u8Counter;
    //1. CAN模块引脚配置:MCF52235_pin_27为CANRX功能,MCF52235_pin_27为CANTX功能
    MCF_GPIO_PASPAR = MCF_GPIO_PASPAR_PASPAR2(2) | MCF_GPIO_PASPAR_PASPAR3(2);
    //2. 复位模块
    MCF_CAN_CANMCR = MCF_CAN_CANMCR_SOFTRST;
    //3. 等待复位完成
    while ((MCF_CAN_CANMCR & MCF_CAN_CANMCR_SOFTRST) != 0x00);
}
```

```

//4. 禁止 FlexCAN 时钟并设置缓冲区数目
MCF_CAN_CANMCR = MCF_CAN_CANMCR_MDIS | MCF_CAN_CANMCR_MAXMB(MaxMBnum);
//5. 等待禁止模式结束
while ((MCF_CAN_CANMCR & MCF_CAN_CANMCR_NOTRDY) != MCF_CAN_CANMCR_NOTRDY);
//6. 设置位速率 100K
MCF_CAN_CANCTRL &= ~(MCF_CAN_CANCTRL_CLKSRC); //选择60M内部时钟
MCF_CAN_CANCTRL = MCF_CAN_CANCTRL_PRESDIV(59) | //60 分 频
MCF_CAN_CANCTRL_RJW(0) | //再同步段1
MCF_CAN_CANCTRL_PSEG1(2) | //相位段1为3
MCF_CAN_CANCTRL_PSEG2(5) | //相位段2为6
MCF_CAN_CANCTRL_PROPSEG(5); //传播段为6
//7. 初始化报文缓冲区
for (u8Counter=0;u8Counter<MaxMBnum;u8Counter++)
MCF_CAN_MB_CTRL(u8Counter) = MCF_CAN_MB_CTRL_CODE(0);
//8. 设置接收ID掩码
MCF_CAN_RXGMASK = 0xFFFFFFFF;
//9. 掩码中断
MCF_CAN_IMASK=(MCF_CAN_IMASK_BUF0M | MCF_CAN_IMASK_BUF1M);
//10. 使能 FlexCAN 模块
MCF_CAN_CANMCR &= ~(MCF_CAN_CANMCR_MDIS);
//11. 等待直到正常模式
while ((MCF_CAN_CANMCR & MCF_CAN_CANMCR_NOTRDY) != 0x00);
//12. 使能 Flex CAN 寄存器自测模式, 如果不使用自测模式可将该句注释掉。
MCF_CAN_CANCTRL |= MCF_CAN_CANCTRL_LPB;//进入自测模式
return;
}

```

11.4.2 MCF52235 FlexCAN模块测试工程说明

本书的随书光盘附带了 2 个 FlexCAN 模块程序，分别是自测程序 SelfTest 和 CAN 通信程序 CANBus。

自测程序 SelfTest 主要用于测试芯片的 FlexCAN 模块是否正常，程序通过 MB0 发送一个数据帧，然后通过 MB1 进行接收，并将接收到的数据帧通过串口发送到 PC 机，用户可以通过串口调试工具观察到 MB1 接收到的数据。

CANBus 实现三个 CAN 节点进行通信。三个 CAN 节点分别标记为节点 A，节点 B 和节点 C。三个节点可使用 3 块带 FlexCAN 通信接口的 MCF52235 实验板来代替。

标志符的分配如下 (参考 CANopen 最小系统配置标识符分配方案进行分配。功能代码部分使用 0，定义为通用数据传输)：

节点 A：ID=0；节点 B：ID=1；节点 C：ID=2。

假设三个节点通信均传输标准数据帧，节点 A 为服务器节点，负责接收所有客户端节点发送的数据帧，并将接收到的数据通过串口发送到 PC 机。节点 B 每隔一段时间向节点 A 发送 ID 为 1，数据为“IamNodeB”的标准数据帧，节点 C 每隔一段时间向节点 A 发送 ID 为 2，数据为“IamNodeC”的标准数据帧。节点 A 若收到 B 节点发来的数据帧则向 B 发送一个数据为“NodeB OK”的标准数据帧；若收到 C 节点发来的数据帧则向 C 发

送一个数据为“NodeC OK”的标准数据帧。

所有节点均使用 MB0 作为发送缓冲区，MB1 作为接收缓冲区。

上述通信过程的实现步骤如下：

1. 节点 A 设置

(1) 设置 MB1 为接收缓冲区，接收任意 ID 的标准数据帧。MB1 的 ID 设置为 0，全局掩码寄存器 RXGMASK 设置为 0。设置语句如下：

```
CANEnableRXBuffer(1,0,0,0); //配置 MB1 为接收报文缓冲区
```

由于全局掩码寄存器 RXGMASK 只能在冻结模式下设置，所以需放在初始化函数中设置。

```
MCF_CAN_RXGMASK = 0x0; //设置接收掩码，接收任意 ID 的标准数据帧
```

(2) 设置 MB0 为标准数据帧发送报文缓冲区。当收到节点 B 发送的数据帧时，发送 ID 为 0，数据为“NodeB OK”的标准数据帧。当收到节点 C 发送的数据帧时，发送 ID 为 0，数据为“NodeC OK”的标准数据帧。设置语句如下：

```
uint8 received[8]={ 'A','B','C','D','E','F','G','H'}; //接收缓冲区
uint8 transmitB[8]={ 'N','o','d','e','B',' ','O','K'}; //发送缓冲区
uint8 transmitC[8]={ 'N','o','d','e','C',' ','O','K'}; //发送缓冲区

int main()
{
    .....(部分代码省略).....

    if (CANReadBufferData(1,received,&Length,&ide,&stdID,&extID)==1) //读取接收帧内容
    {
        if((ide==0)&&(Length==8))//接收到长度为 8 的标准数据帧
        {
            if((stdID==1)&&(extID==0)) //接收到结点 B 发来的数据帧
                CANSendData(0,0,0,0,8,transmitB);
            if((stdID==2)&&(extID==0)) //接收到结点 C 发来的数据帧
                CANSendData(0,0,0,0,8,transmitC);

            UartSendN(0,8,received); //发送接收到数据内容
        }
    }
}
```

2. 节点 B 设置

(1) 设置 MB0 为发送报文缓冲区，发送 ID 为 1，数据内容为“IamNodeB”的标准数据帧。

```
uint8 transmit[8]={ 'I','a','m','N','o','d','e','B'}; //发送缓冲区
```

```
CANSendData(0,1,0,0,8,transmit); //发送标准数据帧
```

(2) 设置 MB1 为接收 ID 为 0 的标准数据帧的报文缓冲区。全局掩码寄存器 RXGMASK 设置为 0xFFFFFFFF，只接收节点 A 发送的标准数据帧。

```
CANEnableRXBuffer(1,0,0,0); //配置 MB1 为接收报文缓冲区
```

```
MCF_CAN_RXGMASK = 0xFFFFFFFF; //设置接收掩码
```

3. 节点 C 设置:

(1) 设置 MB0 为发送报文缓冲区, 发送 ID 为 2, 数据内容为 “IamNodeC” 的标准数据帧。

```
uint8 transmit[8]={ 'I','a','m','N','o','d','e','C' }; //发送缓冲区
```

```
CANSendData(0,1,0,0,8,transmit); //发送标准数据帧
```

(2) 设置 MB1 为接收 ID 为 0 的标准数据帧的报文缓冲区。全局掩码寄存器 RXGMASK 设置为 0xFFFFFFFF, 只接收节点 A 发送的标准数据帧。

```
CANEnableRXBuffer(1,0,0,0); //配置 MB1 为接收报文缓冲区
```

通过以上设置即可完成上述通信功能。

SelfTest、CanBus 程序的文件结构图是相同的, 只是 main.c 文件内容有所不同, 三个程序的文件结构图如图 11-17。

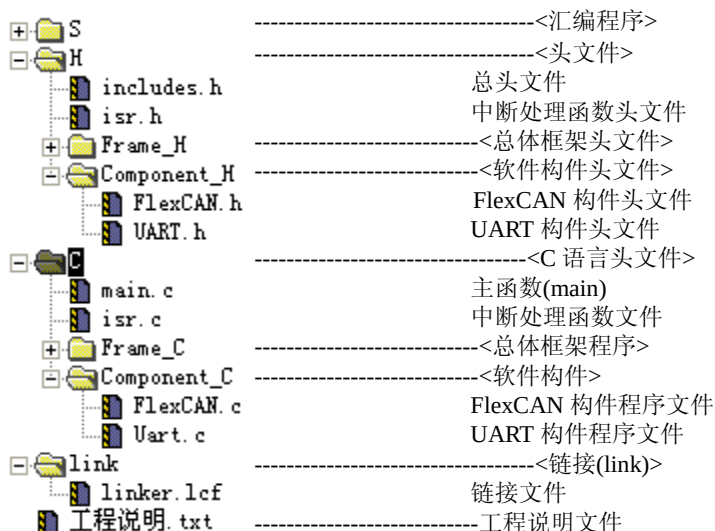


图11-17 FlexCAN模块程序的文件结构图

11.5 进一步讨论

为了保证数据的完整性和传输的可靠性, CAN 总线提供了错误标定、故障界定和错误恢复机制, 任何节点发生错误都会报错。CAN 总线中存在 5 种错误类型, 下面简要介绍一下它们产生的原因及处理方法。

(1) 位错误 (Bit Error): CAN 节点在向总线发送一位数据的同时也在同步检测总线状态, 当检测到总线电平与发送位电平不同时, 将视为出现位错误。

(2) 填充错误 (Stuff Error): 在使用位填充方法进行编码的报文中, 出现了第 6 个

连续相同的位电平时，视为出现填充错误。

(3) CRC 错误 (CRC Error): CRC 序列是由发送器 CRC 计算的结果组成的。接收器与发送器 CRC 计算方法相同。如果计算的结果与接收到的 CRC 序列不同，则视为出现 CRC 错误。

(4) 形式错误 (Form Error): 当固定形式的位域中出现一个或多个非法位时，则视为出现形式错误。

(5) 应答错误 (Acknowledgment Error): 在应答间隙，发送器未检测到“0”时，将视为出现应答错误。

检测到出错条件的节点通过发送错误标志进行错误标定。当任何节点检测出位错误、填充错误、形式错误或应答错误时，由该节点在下一位开始发送错误标志。

在 CAN 总线中，任何一个节点可能处于下列 3 种故障状态：主动错误状态、被动错误和总线关闭状态之一(请参见 11.1.4 节错误帧描述)。为了界定 CAN 节点处于何种故障，每个 CAN 节点中都设置了 2 个计数：发送出错计数和接收出错计数。下面结合 MCF52235 FlexCAN 错误/状态寄存器 ERRSTAT 和 FlexCAN 错误计数器寄存器 (ERRCNT) 来阐述 FlexCAN 模块的错误标定和故障界定的处理方法。

1. FlexCAN 错误/状态寄存器 (ERRSTAT)

ERRSTAT 显示各种错误条件以及 CAN 节点的一些状态信息。除 BOFFINT 和 ERRINT 之外，该寄存器的其余位均为只读位。该寄存器定义如下：

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5~D4	D2	D1
定义	BIT1 ERR	BIT0 ERR	ACK ERR	CRC ERR	FRM ERR	STF ERR	TX WRN	RX WRN	IDLE	TXRX	FLT CONF	BOFF INT	ERR INT

D32~D16、D0 位未定义。D15~D10 为已报告的错误条件位，该位域显示从 CPU 最近一次读该寄存器到当前时刻已报告的错误，读操作将清该位域。D9~D3 位为状态位。

D15—BIT1ERR, Bit1 错误。FlexCAN 模块在发送位的同时也会对总线进行监视。如果所发送的位值与所监视到的位不同，则在此位时间内将检测到一个位错误。若该位被置位则说明发送过程中位错误发生。BIT1ERR=0，无发送位错误。BIT1ERR=1，至少有一个发送的“1”被接收为“0”。

注意：在仲裁场的填充位流期间或报文的应答间隙发送一个“1”的情况例外。此时，当监视到一“0”时，不会视为发送位错误。当发送器发送一个被动错误标志但检测到“0”位时，也不视为发送位错误，此时发送错误位 BIT1ERR 不会被修改。

D14—BIT0ERR, Bit0 错误。该位标志报文中发送和接收的位不一致。BIT0ERR=0，无发送位错误。BIT0ERR=1，至少有一个发送的“0”被接收为“1”。

D13—ACKERR, 应答错误位。该位标志发送完一个报文时，是否接收到正确的应答信号。只要在应答间隙期间所监视的位不为“0”，则发送器就视为出现一个应答错误。ACKERR=0，从最近一次读该寄存器到当前时间没有发生应答错误。ACKERR=1，从最近一次读该寄存器到当前时间检测到应答错误。

D12—CRCERR 位，CRC 校验错误。该位标志接收器是否检测到 CRC 错误。CRC 序

列包括发送器的 CRC 计算结果,接收器计算 CRC 的方法与发送器相同。如果计算结果与接收到 CRC 序列的结果不相符,则检测到一个 CRC 错误。CRCERR=0,从最近一次读该寄存器到当前时间没有检测到 CRC 错误。CRCERR=1,从最近一次读该寄存器到当前时间检测到 CRC 错误。

D11—FRMERR,形式错误。该位标志接收节点检测到报文形式错误,即在固定格式的位场出现了至少一个非法位。FRMERR=0,从最近一次读该寄存器到当前时间没有检测到形式错误。FRMERR=1,从最近一次读该寄存器到当前时间检测到形式错误。

D10—STFERR,位填充错误。STFERR=0,从最近一次读该寄存器到当前时间没有检测到位填充错误。如果在使用位填充法进行编码的信息中,出现了 6 个连续相同的位电平时,将检测到一个填充错误。STFERR=0,从最近一次读该寄存器到当前时间没有检测到位填充错误。STFERR=1,从最近一次读该寄存器到当前时间检测到位填充错误。

D9—TXWRN,发送错误状态标志。该位反映 FlexCAN 发送错误计数器的状态。TXWRN=0,发送错误计数器<96。TXWRN=1,发送错误计数器≥96。

D8—RXWRN,接收错误状态标志。该位反映 FlexCAN 接收错误计数器的状态。RXWRN=0,接收错误计数器<96。RXWRN=1,接收错误计数器≥96。

D7—IDLE,空闲状态位。该位标志 CAN 总线是否有活动。IDLE=0,CAN 总线不空闲。IDLE=1,CAN 总线空闲。

D6—TXRX,发送/接收状态位。该位标志 FlexCAN 模块已成功发送/接收一个报文。当 IDLE=1 时 TXRX 无效。TXRX=0,若 IDLE=0,标志 FlexCAN 正在接收报文。TXRX=1,若 IDLE=0,标志 FlexCAN 正在发送报文。

D5~D4—FLTCONF,错误标定状态。该两位标志 FlexCAN 模块错误标定状态。如果寄存器 CANCTRL 的 LOM 被置位,则 FLTCONF 标定为“被动错误”。由于 CANCTRL 寄存器不受软复位影响,如果该寄存器的 LOM 被置位,那么 FLTCONF 字段不受软复位影响。FLTCONF=00,主动错误。FLTCONF=01,被动错误。FLTCONF=1x,总线关闭。

D2—BOFFINT,总线关闭中断位。该位用于当 FlexCAN 总线关闭时产生中断请求。用户必须写 1 清该位。写 0 无效。BOFFINT=0,不允许总线关闭中断请求。BOFFINT=1,当 FlexCAN 状态变为总线关闭时该位被置位。如果 CANCTRL 的 BOFFMSK 位被置位,则将产生一个中断。复位将清除中断请求。

D1—ERRINT,错误中断位。该位标志 ERRSTAT 的 D15~D10 位中至少有一个被置位。用户可以写 1 清该位。写 0 无效。ERRINT=0,没有错误中断请求。ERRINT=1,至少有一个错误位被置位。如果 CANCTRL 的 ERRMSK 位被置位,将产生一个中断请求。

2. FlexCAN 错误计数器寄存器 (ERRCNT)

该寄存器有两个 8 位的字段,用于显示 FlexCAN 接收错误计数器 (RXECTR) 和发送错误计数器 (TXECTR) 的值。FlexCAN 模块完全按照 CAN 协议标准执行加减计数。两个计数器只有在冻结模式下可写,在其他模式下仅可读。该寄存器复位为 0。D31~D16 未定义。其他位定义如下:

D15~D8—RXECTR,接收错误计数器,标志当前接收错误的数目。

D7~D0—TXECTR, 发送错误计数器, 标志当前发送错误的数目。

在冻结模式下, 对 ERRCNT 寄存器进行写操作是一个间接操作, 数据首先被写入到辅助寄存器中, 然后按照一定的时序执行一个请求/应答过程。因此需要一定的时间来将数据写入到寄存器中。除等待操作用户可见外, 所有其他操作对用户都是不可见的。如果需要, 可以通过软件轮询该寄存器来检测数据何时被真正写入。FlexCAN 按照协议对所有的总线状态(比如: 主动发送错误或被动错误)作出回应, 延时发送开始时(被动错误), 且在总线关闭时避免对总线干扰。FlexCAN 总线状态转变的基本规则如下:

(1) 如果 TXECTR 或 RXECTR 的值 ≥ 128 , 则出错和状态寄存器(ERRSTAT)中的 FLTCONF 字段将更新为“被动出错”状态。

(2) 如果 FlexCAN 为“被动出错”状态, 且 TXECTR 或者 RXECTR ≤ 127 , 当另一个节点也满足条件时, ERRSTAT 的 FLTCONF 字段更新为“主动出错”状态。

(3) 如果 TXECTR ≥ 255 , ERRSTAT 的 FLTCONF 字段更新为“总线关闭”状态, 这时若中断允许则将产生中断。TXECTR 的值复位为 0。

(4) 如果 FlexCAN 处于总线关闭状态, 然后 TXECTR 计数到总线上 128 次出现 11 个连续“1”位之后, 则 TXECTR 复位为 0, 且按照内部计数器的计数方式, 计数 11 个这样的位, 然后环绕计数。TXECTR=12 时, ERRSTAT 的 FLTCONF 字段更新为“主动出错”, 并且两个出错计数器都复位为 0。如果连续 1 位小于 11 个时, 后面出现了一个 0 位, 则内部计数器复位为 0, 但 TXECTR 不复位。

(5) 如果系统启动时, 只有一个结点在运行, 那么每发送一个报文, 若出现应答错误(由 ERRSTAT 的 ACKERR 标记), 则 TXECTR 加 1。在转变为“被动错误”状态后, 即使出现更多的应答错误 TXECTR 也不再增加。因此, 设备永远不会进入总线关闭状态。

(6) 如果 RXECTR ≥ 127 , 那么即使接收器检测到更多的错误它也不再增加。在成功接收到下一个报文时, 计数器被设置为 119 到 127 之间的一个值, 并变为“主动出错”状态。

关于错误计数的更多详细规则, 请参考 CAN2.0B 协议。

第12章 基于MCF52233的嵌入式以太网

本章按自底向上的顺序，逐层介绍使用 MCF52233 以太网模块来实现以太网技术的过程，其中链路层及其以下部分与芯片相关，涉及到对具体寄存器的操作；网络层及其以上部分则属软件层，与具体芯片型号无关。要通透理解嵌入式以太网技术，重点和难点在于对底层的学习和理解，因此，建议读者对 MCF52233 以太网模块重点学习。只有真正理解底层基本工作原理与过程，才能对嵌入式以太网技术有全面和深刻的理解。

本章内容对一般读者来说有一定难度，有些内容需反复阅读，并通过上下文联系才能理解，对未学过“计算机网络”的读者更是如此。对应用例程部分，需要耐心并反复实践与应用才能逐步掌握，千万不能急于求成。

12.1 嵌入式以太网相关知识

12.1.1 以太网的由来与协议模型

1. 以太网的由来与发展简史

计算机网络是指通过通信介质将地理上分散的计算机连接起来，以实现相互通信和资源共享（硬件和软件资源）。以太网（Ethernet）是现有计算机网中采用的最广泛的技术，最初是由 Bob Metcalfe 于 1973 年发明（他后来创建了著名的 3Com 网络公司）。Ether 一词在电磁学中曾表示充满宇宙的介质。Ethernet 寓意将遍布全球。

以太网最早是由 Xerox 研发，经 DEC 公司和 Intel 公司联合扩展，于 1982 年公布的基带局域网规范。以太网 IEEE802.3 规范就是以此为基础制定的。按照 ISO 开放互联参考模型的分层结构，以太网规范只包括该通信模型中的物理层和数据链路层。而俗称的以太网技术不仅包括上述以太网规范，而且还包括 TCP/IP 协议族，有时甚至把应用层的简单邮件传送协议 SMTP、域名服务 DNS 和文件传输协议 FTP 等都与以太网这个名词捆绑在一起。

早期以太网只有 10Mbps 的吞吐量，使用一种被称之为 CSMA/CD（带有碰撞检测的载波侦听多路访问）的访问控制方法，这种早期的以太网被称为标准以太网。随着网络技术的发展，1995 年 3 月 IEEE 宣布了 IEEE802.3 100BASE-T 快速以太网标准，标志快速以太网时代的开始，目前快速以太网正在继续发展。

2. 以太网拓扑结构

网络拓扑结构反映出网络中各实体的连接关系，它是实现各种网络协议的必然依托，对网络性能和系统可靠性都有重大影响。常用的结构有总线型拓扑结构和星型拓扑结构。

总线型拓扑结构：将网络中所有设备通过相应的硬件接口直接连接到公共总线上，结点之间按广播方式通信，某个结点发出的信息，其它所有结点均可“收听”到。它结构简

单、布线容易、易于扩充，是局域网中常用的拓扑结构。但是由于所有的数据都需经过总线传送，总线成为整个网络的瓶颈，带宽受限且出现故障时诊断较为困难。

星型拓扑结构：每个结点都有单独的通信线路与中心结点连结，中心节点采用专门的网络设备，如交换机。与总线式结构相比，每个节点发出的数据，在中心节点处转发时可能会受到某些限制，它更便于管理，且连接点的故障容易监测和排除，但同时中心节点要求更高，一旦中心节点出现故障，就很可能导致整个网络瘫痪，尽管如此，目前以太网使用星型拓扑结构组网的较多。

3. 网络协议模型

为了清晰地研究复杂的计算机网络，针对其所执行的各种功能，国际组织提出了“网络系统层次模型”的概念，下面将简要介绍基于该概念的两主要网络参考模型。

1) OSI参考模型

国际标准化组织（International Standards Organization, ISO）在 1983 年颁布的“开放系统互连（Open Systems Interconnection, OSI）”参考模型中，将整个网络功能划分为七个层次，从上至下依次是应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。该参考模型结构如图 12-1 所示，各层简明功能参见表 12-1。

国际电子电器工程师协会（The Institute of Electrical and Electronic Engineer, IEEE）定义的 IEEE802.3 协议，将图 12-1 中的数据链路层划分为逻辑链路控制（Logic Link Control, LLC）子层和介质访问控制（Medium Access Control, MAC）子层。

逻辑链路控制子层使得上层在使用网络服务时，不必去考虑其下层是如何设置的，它会启动控制信号的交互、组合数据的流通、解释命令、发出响应，并实现错误控制及恢复

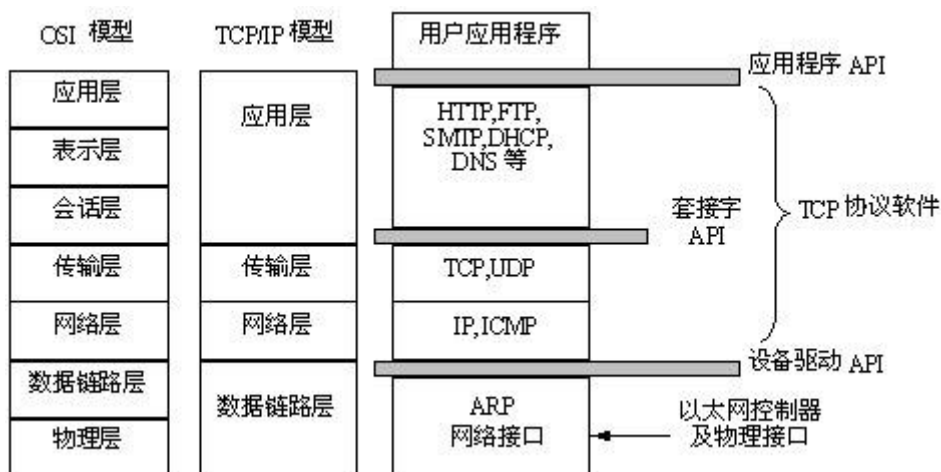


图12-1 网络参考模型及协议对应关系

等功能。而介质访问控制子层则负责定义各种不同的访问控制方法，以决定如何控制实际传输介质的使用。

2) TCP/IP协议模型

由图 12-1 可知，TCP/IP 协议模型的层次比 OSI 参考模型的七层要少，由应用层、传输层、网络层及数据链路层组成。与 OSI 参考模型相比，它的应用层对应于 OSI 参考模型的会话层、表示层和应用层；数据链路层对应于 OSI 的数据链路层和物理层；传输层和网络层则分别对应于 OSI 的传输层和网络层，各层的简明功能如表 12-1 所示。

表 12-1 网络模型的各层简明功能

OSI模型	TCP/IP模型	功能
应用层	应用层	实现网络虚拟终端的功能以及实现用户终端功能之间的映射，提供 FTP、HTTP、SMTP等功能。
表示层		用标准编码方式对数据进行编码，对该数据结构进行定义，并管理这些数据。
会话层		把要求建立会话的用户地址转换成相应的传送开始地址，以实现正确的传送连接。
传输层	传输层	接收从会话层发出的数据，根据需要把数据划分为许多很小的单元(即报文)，传送给网络层。
网络层	网络层	(1)处理来自传输层的分组发送请求，收到请求后，将分组装入IP数据报，填充报头，选择去往信宿机的路径，然后将数据报发往适当的网络接口； (2)处理输入数据报，检查其合法性，然后再进行寻径；处理ICMP报文，处理路径、流控、拥塞问题。
数据链路层	数据链路层	(1)将待发送的数据封装在多个数据帧里，并顺序地发送每一帧，同时处理接收方回送的确认帧； (2)该层通过在帧头和帧尾附加上特殊的二进制编码，产生和识别帧界。
物理层		按照传送介质的电气机械特性传送数据，传送单位主要以位(Bit)为单位，并将信息按位逐一从一个系统经物理通道送往另一个系统。

4. 协议的封装过程

各网络主机间的通信实际上可被视为各层的对等实体之间在进行通信，除了最底层的

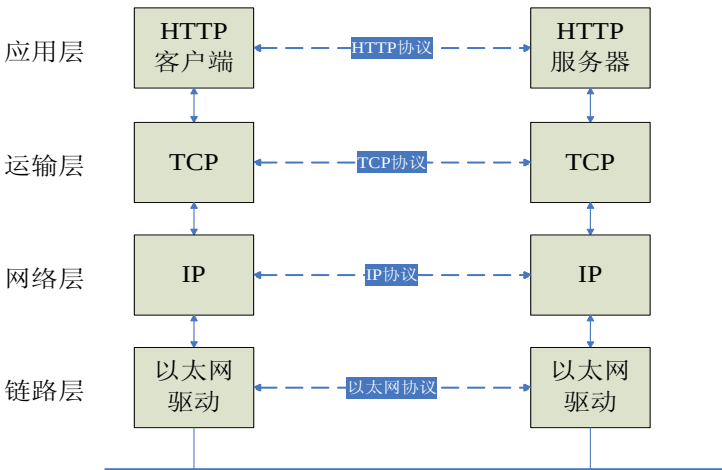


图12-2 以太网上运行HTTP协议的两台主机通信模型

对等实体间进行的是实际通信外，其他对等实体之间都是虚通信，即没有数据流从一台主机的某一层（除了最底层）直接流向另一台主机的相应层，它们之间的通信其实是通过下层提供的服务来完成的。

图 12-2 说明了网络上运行 HTTP 协议的两台主机的通信模型。实线表示真实通信通路，虚线表示逻辑通信过程。左边是 HTTP 客户端，右边是 HTTP 服务器。在该模型的每个层面上双方使用特定的协议进行通信。比如在链路层中使用的是以太网协议，即使用以太帧的格式来收发数据；而在应用层则使用了 HTTP 协议。在某些层中可以使用的协议有多个，比如运输层可以使用 UDP 协议，也可以使用 TCP 协议，这些协议的具体内容及使用方法，后面将会讨论。

当用户使用 TCP/IP 协议传输数据时，数据被送入协议栈中，然后逐个通过每层直到形成一串比特（Bit）流被送入网络。每层对收到的来自上层的数据都要增加首部信息，以打上本层的烙印，协议逐层封装过程如图 12-3 所示。其中 TCP 传给 IP 的数据单元称作 TCP 报文段（简称 TCP 段），IP 传给网络接口的数据单元称作 IP 数据报（简称 IP 段）。通过以太网传输的数据称作帧（Frame），有时也叫做以太帧（后面章节将作详细介绍）。

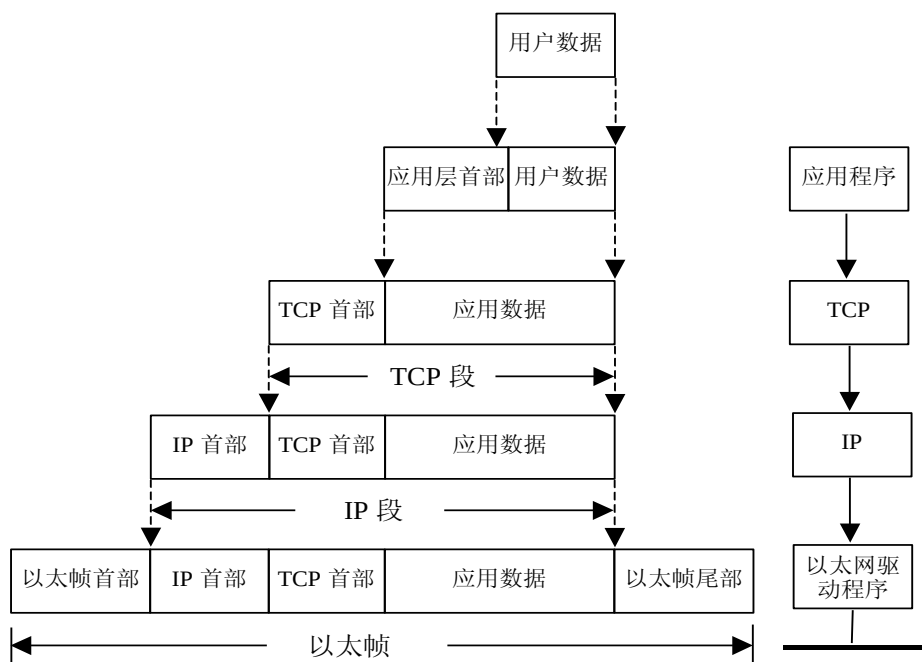


图12-3 协议逐层封装过程

12.1.2 以太网中主要物理设备

1. 网卡

网卡主要实现计算机与外界网络的连接。网卡又称为网络适配器（Adapter）或网络接口卡 NIC（Network Interface Card）等。它是工作在数据链路层的组件，是局域网中连

接计算机和传输介质的接口,不仅能实现与局域网传输介质之间的物理连接和电信号匹配,还涉及帧的发送与接收、帧的封装与拆封、介质访问控制、数据的编码与解码以及数据缓存等功能。具有以太网功能的嵌入式终端设备需要含有网卡的部分功能。

2. 网线

网线用于局域网连接,常见的网线主要有双绞线、同轴电缆和光缆三种。具有以太网功能的嵌入式终端设备常用双绞线与网络系统连接。双绞线是指互相绝缘的金属导线绞合形成的对线。它们的这种绞合方式可抵御很大一部分外界电磁干扰。实际使用的典型双绞线电缆有四对,放在一个电缆套管里。它可分为屏蔽双绞线(Shielded Twisted Pair, STP)和非屏蔽双绞线(Unshielded Twisted Pair, UTP)两种。屏蔽双绞线 STP 比非屏蔽双绞线 UTP 稳定性高,但价格也稍高一些。购买时,注意双绞线有五类、超五类、六类等,适用与不同的传输速率,价格差别较大。

常用网线的内部 8 根线按数字顺序其颜色分别为橙白、橙、绿白、蓝、蓝白、绿、棕白、棕。1 和 2 用于差分发送,3 和 6 则用于差分接收,4、5、7 和 8 暂不使用,1 和 2、3 和 6 必须双绞以最大限度的减少电磁干扰。

在实际使用时,接好网络接头的网线,注意有交叉线与直通线之分。两台网络终端设备直接互连时使用交叉线,其他情况使用直通线。

3. 交换机

交换机(Switch),或称交换式集线器,工作在数据链路层,它在内部维护一个 MAC 地址/端口映射表和缓存区域,实现依据 MAC 地址与端口映射关系来转发数据的功能。其简明工作过程如下:1) 交换机根据收到的数据帧中的源 MAC 地址建立该地址同交换机端口的映射,并将其写入 MAC 地址表中。2) 交换机将数据帧中的目的 MAC 地址同已建立的 MAC 地址表进行比较,以决定由哪个端口进行转发。3) 若数据帧中的目的 MAC 地址不在 MAC 地址表中,则向所有端口转发。这一过程称之为泛洪(flood)。但当交换机接收到广播帧和组播帧时,也向所有的端口转发。与集线器相比,交换机能够隔离冲突域,每一个端口是一个独立的冲突域,有效的抑制了广播风暴的产生。交换机支持不同速率的主机通信,为每一相连的网络设备提供专用带宽。交换机工作原理不要求自身具有 MAC 地址及 IP 地址,若有 MAC 地址及 IP 地址,那是为了其他工具对其配置使用。

以上例子中所说的集线器,是指早期的中继式集线器,它工作在“物理层”,其主要功能是对接收到的信号进行再生整形放大,仅通过将一个端口接收的信号重复分发给其他端口来扩展物理介质。所有连接到集线器的设备共享同一介质,所以它们也共享同一冲突域、广播域和带宽。由于其性能单一,目前一般不再使用。

4. 路由器

当需要在不同网段(后续将作介绍)间传输数据时就需要使用路由器这种网络互联设备。路由器(Router)工作在网络层,用于连接多个逻辑上分开的网络,它屏蔽了各子网使用的硬件设备的差异。其主要工作就是为经过路由器的每个数据帧寻找一条最佳传输路径,实现基于 IP 地址的数据转发。路由器中维护一张路由表(Routing Table),供路由选

择时使用。路由表中保存着目的网络地址、下一跳路由器的 IP 地址等。该表可由系统管理员手动设定为静态路由表，也可通过路由器中的路由协议获得动态路由表。

路由器工作原理本身要求其自身具有 MAC 地址及 IP 地址，路由器作为网关设备时，其 IP 地址，实质上就是网关地址。

12.1.3 相关名词解释

1. MAC地址

MAC (Media Access Control, 介质访问控制) 地址也叫硬件地址或物理地址，长度为 48 位 (6 个字节)，通常每个字节用二位十六进制数表达，每个字节用冒号隔开。如 08:00:20:0A:8C:6D 就是一个 MAC 地址，其中前三字节 08:00:20 代表网络硬件制造商的编号，它由 IEEE 分配，例如：3Com 公司生产的网络适配器的 MAC 地址的前三字节是 02:60:8C，而后三字节 0A:8C:6D 代表该制造商所制造的某个网络产品（如网络适配器）的系列号。每个网络制造商必须确保它所制造的每个以太网设备都具有相同的前三个字节以及不同的后三个字节，这样就可保证世界上每个以太网设备都具有唯一的 MAC 地址。

MAC 地址前三字节需要向国际组织申请，但是如果仅在自己局域网中使用，可以采用 22:22:22 开头 MAC 地址，这段 MAC 地址是内部保留地址，无需向 IEEE 申请，所以一般实验用嵌入式产品可以使用以 22:22:22 开头的 MAC 地址。

MAC 广播地址是全 1，即 FF-FF-FF-FF-FF-FF。MAC 地址中，第一字节最低位为 1 是多播地址，为 0 表示唯一地址；次低位为 1 表示局部唯一地址，为 0 表示全球唯一地址。

2. IP地址

Internet 上的每台主机都有一个唯一的 IP 地址。IP 地址由网络号 (Network ID) 和主机号 (Host ID) 两部分组成。网络号标识的是 Internet 上的一个子网，而主机号标识的是子网中的某台主机。

IP 地址长度为 32 位，分为 4 段，每段 8 位，如用十进制数字表示，则每段数字的范围为 1~254 (0 和 255 除外)，段与段之间用句点 “.” 隔开。例如 192.168.149.1。IP 地址就像是我们的家庭住址一样，如果你要写信给一个人，你就要知道他（她）的地址，这样邮递员才能把信送到，计算机发送信息就好比是邮递员，它必须知道唯一的“家庭地址”才能不至于把信送错人家。只不过我们的地址使用文字来表示，而计算机的地址用数字来表示。

表征网络号 (即网络地址) 和主机号 (即主机地址) 的 IP 地址可分为 A、B、C、D、E 共 5 类，常用的是 B 和 C 两类。

A 类 IP 地址：用 7 位来标识网络号，24 位标识主机号，最前面 1 位为 “0”，全世界总共只有 126 个^①A 类网络，每个 A 类网络最多可以连接 16777214 台^②主机。

B 类 IP 地址：用 14 位来标识网络号，16 位标识主机号，最前面 2 位是 “10”。B 类

^① $126=2^7-2$

^② $16777214=2^{24}-2$

地址的第 1 段取值介于 128~191 之间, 前 2 段合在一起表示网络号。B 类地址适用于中等规模的网络, 全世界大约有 16000 个 B 类网络, 每个 B 类网络最多可连接 65534 台主机。

C 类 IP 地址: 用 21 位来标识网络号, 8 位标识主机号, 最前面 3 位是“110”。C 类地址的第 1 段取值介于 192~223 之间, 前 3 段合在一起表示网络号。最后 1 段标识网络上的主机号。C 类地址适用于校园网等小型网络, 每个 C 类网络最多可以有 254 台主机。

网络地址是因特网协会的网址分配机构 ICANN (the Internet Corporation for Assigned Names and Numbers) 负责分配, 目的是为了保证网络地址的全球唯一性。主机地址是由各个网络的系统管理员分配。网络地址的唯一性与网络内主机地址的唯一性确保了 IP 地址的全球唯一性。

根据用途和安全性级别的不同, IP 地址还可分为两类: 公用地址和私有地址。公用地址在 Internet 中使用, 可以在 Internet 中较为随意访问。私有地址只能在内部网络中使用, 只有通过代理服务器才能与 Internet 通信, IP 地址中的私有地址有: 10.0.0.0~10.255.255.255、172.16.0.0~172.31.255.255、192.168.0.0~192.168.255.255。

4. 子网划分和子网掩码

子网掩码将某个 IP 地址划分成网络地址和主机地址两部分, 以便于 IP 地址的寻址操作。IP 地址中网络号部分全用“1”, 主机号部分全用“0”(二进制), 就是网络的子网掩码。比如 IP 地址“192.168.1.1”和子网掩码“255.255.255.0”。子网掩码“255.255.255.0”中有 24 个“1”, 代表与此相对应的 IP 地址左边 24 位是网络号; 有 8 个“0”, 代表与此相对应的 IP 地址右边 8 位是主机号。这样, 子网掩码就确定了一个 IP 地址的 32 位二进制数字中哪些是网络号、哪些是主机号。这对于采用 TCP/IP 协议的网络来说非常重要, 只有通过子网掩码, 才能表明一台主机所在的子网与其他子网的关系。子网掩码有数百种, 最常用的两种子网掩码是“255.255.255.0”和“255.255.0.0”。

3. 网段

网段即为一个网络的网络地址, 由主机 IP 地址和子网掩码“按位与”运算得到。不同的网络号为不同的网段。例如: 有两个网络地址分别是 192.168.0.0 和 192.168.1.0, 若两者的子网掩码均为 255.255.255.0, 那么它们属于不同的网段。但如果子网掩码是 255.255.254.0, 它们就处同一网段内。在以太网中, 只有在同一个网段内的计算机之间才能“直接”互通, 不同网段的计算机要通过网关 (Gateway, 通常为路由器) 才能互通。

5. 网关

网关 (Gateway) 是一种充当转换重任的计算机系统或设备, 又称网间连接器或协议转换器 (传输网关或应用网关)。作为前者, 它在传输层上实现网络的互连, 如局域网间的互连, 而作为后者, 它在使用不同的通信协议、数据格式或语言, 甚至体系结构完全不同的两种系统之间担任翻译器的角色, 它对收到的信息要重新打包, 以适应目的系统的需求, 常用于广域网的互连, 大多数运行在 OSI 七层协议的应用层。本书讨论的网关则指用于局域网互连的传输型网关。

从实际用户来看，网关就是一个网络通向其他网络的 IP 地址。比如有网络 A 和网络 B，网络 A 的 IP 地址范围为“192.168.1.1~192.168.1.254”，子网掩码为“255.255.255.0”；网络 B 的 IP 地址范围为“192.168.2.1~192.168.2.254”，子网掩码为“255.255.255.0”。在没有路由器的情况下，两个网络之间是不能进行 TCP/IP 通信的，即使是两个网络连接在同一台交换机上，TCP/IP 协议也会根据子网掩码（255.255.255.0）判定两个网络中的主机处在不同的网络里。而要实现这两个网络之间的通信，则必须通过网关。如果网络 A 中的主机发现数据包中的目的主机 IP 地址不在本地网络中，就把数据包转发给它自己的网关，再由网关转发给网络 B 的网关，网络 B 的网关再转发给网络 B 中的某个主机。所以说只有设置好网关的 IP 地址，TCP/IP 协议才能实现不同网络之间的相互通信。实际上具有路由功能的设备，如路由器、启用了路由协议的服务器（实质上相当于路由器）、代理服务器（也相当于路由器），都可充当网关的角色，它们的 IP 地址也就是网关的 IP 地址。

6. 以太网

以太网中数据以帧为单位进行传输，一帧数据又被称为以太网帧，其格式如图 12-4 所示。实际数据的传输必须经过以太网帧的封装和解封装。以太网帧中的前导位、帧起始位和 CRC 校验位由硬件自动添加和删除，与上层协议无关，因此以太网驱动程序只需要处理其他字段。

7字节	1字节	6字节	6字节	2字节	46~1500字节	4字节
前导位	帧起始位	目的MAC地址	源MAC地址	长度/类型	数据	CRC校验

图12-4 以太网帧格式

长度/类型字段：该字段值小于等于 1500 字节时，则表示“数据字段”中字节数（即长度）。若该字段值大于等于 1536 字节时，则表示数据字段中数据的类型，比如 0x0800 表示数据字段存放的是 IP 数据报，0x0806 则表示数据字段存放的 ARP 数据报。如果长度/类型字段的值大于 1500 而小于 1536，则此以太网帧无效。

数据字段：数据报的具体内容。最少 46 字节，不足 46 字节，需程序员填充到 46 字节。最大 1500 字节，若实际数据超过 1500 字节，需程序员分拆传输。

7. CSMA/CD（载波侦听多路访问/冲突检测）

在以太网中，共享一个网络段的所有计算机形成一个冲突域（collision domain），CSMA/CD（Carrier Sense Multiple Access/Collision Detect，载波侦听多路访问/冲突检测）是一种解决冲突的方法。在 CSMA/CD 中，每台计算机侦听网线的静默时间（以纳秒为单位）。当网线静默时，需要传输数据的计算机，将发送数据通过该段网线，如果其他计算机没有同时发送数据，则该计算机发送的数据顺利通过。当两台计算机试图同时在一条网线上传输数据时，会引起冲突（collision），一旦发生冲突，这两台计算机都会检测到冲突，并停止传输。继续进行侦听，各自随机等候一段时间，侦听到一个静默期后，重新传输。一般情况下，这样的方式可以解决冲突。有时，若有的计算机出现不遵守 CSMA/CD 情况，不断地发送数据，这被称为超时传输（jabber），通常是由硬件故障造成的。

8. 半双工与全双工、10BASE-T和100BASE-TX模式

半双工(Half Duplex): 指数据可以在一个信号载体的两个方向上传输,但是不能同时传输,即发送操作和接收操作不能并发执行。

全双工 (Full Duplex): 指在发送数据的同时也能够接收数据,两者同步进行,目前的网卡一般都支持全双工,全双工只存在于点对点连接。

10BASE-T 模式: 10 表示传输速率为 10Mbps, BASE 表示采用基带传输^①, T(Twisted) 为双绞线,其最大传输距离为 100m,实际传输距离受物理连线影响。物理上使用一种被称为曼彻斯特编码的方式,每位的中间有一个跳变,位中间的跳变既作时钟信号,又作数据信号;从高到低跳变表示“0”,从低到高跳变表示“1”。高电平为+0.85 伏,低电平为-0.85 伏。

100BASE-TX 模式: 一种快速以太网模式,100Mbps 的数据传输速率,在交换式以太网环境中可实现全双工通信。发送端在+1V 和-1V 之间共计有 3 种信号,每种信号之间的电压差为 0.6V;接收端在+0.26V 和-0.26V 之间共计有 3 种信号,每种信号之间的电压差为 0.17V。

9. 自动协商

自动协商用来确定连接对等方的通信能力。它是指本地设备向链路中的远端设备通告自己的工作方式和传输能力,并侦测远端的通告,以使得它们能自动配置在最优能力下工作。自动协商中,发送方每隔 16ms 发送快速链路脉冲 (Fast Link Pulse, FLP),接收方通过这个脉冲获得链路码字 (Link Code Word, LCW),选择双方所能支持的具有通信能力,并且通过 FLP 发送 ACK 来进行确认。**自动协商可以自动确定通信速度与模式。**自动协商完成之后按照协商的结果进行正常的数据通信,无需再进行自动协商,直到链路重新加电、重置或者强制重新协商后才重新开始自动协商过程。

12.2 MCF52233以太网模块外部连接及编程基础

MCF52233 以太网模块集成了快速以太网控制器 (Fast Ethernet Controller, FEC) 和以太网物理层收发器 (Ethernet Physical Transceiver, EPHY),实现以太帧的收发功能。MCF52233 以太网模块对外数据传输的引脚只有 4 个: 2 个差分输入 (RXP、RXN) 和 2 个差分输出 (TXP、TXN)。但是在具体使用该模块时,还需外接一些电阻电容元件和网络隔离部件等,以保证传输的稳定性。网络隔离部件通常采用 1:1 的变压器,它的主要功能是保护 EPHY 免于电气错误而导致损坏。

^①将数字信号 1 和 0 直接用两种不同的电压表示称为基带信号,在数字通信信道上直接传送基带信号的方法称为基带传输。

12.2.1 以太网模块引脚及硬件连接

1. MCF52233以太网外部实际电路

图 12-5 为 80 引脚 MCF52233 芯片的以太网外部引脚连接图，其中，标号右边的括号内为相应的引脚号，例如，PHY_TXP（50）表示它为以太网物理层发送引脚正端输出，且为 MCF52233 的第 50 引脚。下面简要介绍图中所出现的引脚和隔离芯片。

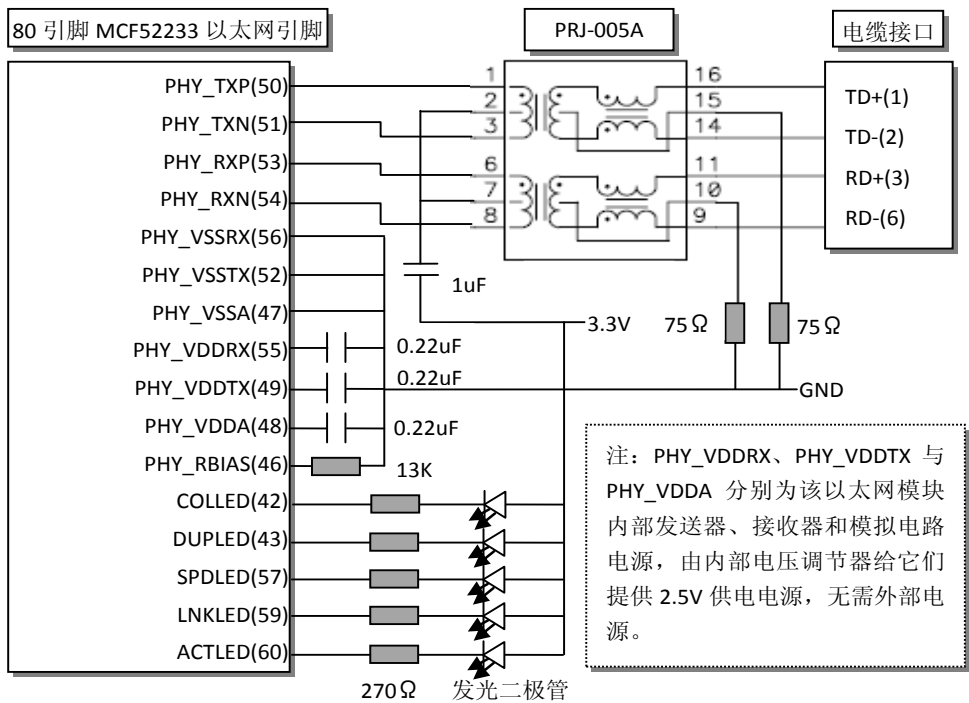


图12-5 MCF52233以太网模块外部连接图

2. MCF52233以太网模块外部引脚

EPHY 引脚包括电源、数据传输和连接网络状态指示灯的信号线等。

（1）数据传输引脚：PHY_TXP/ PHY_TXN 为双绞线输出 +/- 引脚，PHY_RXP /PHY_RXN 为双绞线输入 +/- 引脚。这些引脚复位后都处于高阻抗状态。上述四个引脚与隔离变压器 PRJ-005A 的连接关系见表 12-2。

表 12-2 MCF52233 网络传输引脚的外部接线

MCF52233引脚	PRJ-005A引脚号	电缆（双绞线中线号）
PHY_TXP	1	TD+（1）
PHY_TXN	3	TD-（2）
PHY_RXP	6	RD+（3）
PHY_RXN	8	RD-（6）

（2）EPHY 偏置控制引脚（PHY_RBIAS）：在该引脚与 PHY_VSSA 之间要外接 0.1%的

电阻 (RBIAS)，布板时该电阻要尽可能地靠近 PHY_RBIAS 引脚。寄生电容^①必须在 10pF 以下，超过 50pF 会使得通信不稳定。在 RBIAS 附近不允许有高速信号。

(3) 电源引脚：PHY_VDDRX/PHY_VSSRX 为 EPHY 接收器内部电源引脚；PHY_VDDTX/、PHY_VSSTX 为 EPHY 发送器内部电源引脚；PHY_VDDA/PHY_VSSA 为 EPHY 模拟电路内部电源引脚。引出的目的是为了外接滤波电容使用。这 3 对引脚不允许有静态负载，内部的电压调节器可产生 2.5V 电源提供给电源引脚正端，若将各自的正端接地，则内部电压调节器将关闭。

(4) 指示灯信号引脚：为了指示网络工作状态，MCF52233 提供了一些指示灯引脚，见表 12-3。当以太网物理收发器控制寄存器 0 (EPHYCTL0) 的 LEDEN=1，且硬件上按照图 12-5 的接法，这些指示灯将指示网络工作状态。

表 12-3 MCF52233 网络传输状态指示灯简明含义

指示灯	名称	现象
COLLED	网络冲突	冲突则闪烁（仅发生在半双工模式）
DUPLED	双工连接类型	指示灯亮为全双工，反之为半双工
SPDLED	连接速度	指示灯亮为100Mbps，反之为10Mbps
LNKLED	连接状态	指示灯亮则表明建立连接，反之未建立
ACTLED	接收/发送数据	指示灯亮则表明正在接收或发送数据
RXLED	接收数据	指示灯亮则表明正在接收数据
TXLED	发送数据	指示灯亮则表明正在发送数据

注： RXLED、TXLED在80引脚封装中没有，在112引脚封装中有

3. 网络隔离变压器

网络隔离变压器又称数据泵，是以太网物理层与物理传输介质相连应有的设备。它所起的作用主要有两个：一是传输数据，把差分信号用差模耦合线圈进行滤波以强化信号；二是隔离网线连接的不同网络设备间的不同电平，以防止不同电压通过网线传输损坏设备，实现阻抗匹配、波形修复、杂波抑制等作用。除此之外，它还能对设备起到一定的防雷保护作用。

目前市场上有单独的隔离变压器芯片和内置隔离变压器的 RJ45 接口出售，其工作原理都是一样的，都能够完成上述功能。为了简化电路，本书的实际开发板在元件选型时采用了内置以太网隔离变压器的 RJ45 接口 PRJ-005A。

12.2.2 以太网物理层收发器EPHY的编程基础

1. MCF52233以太网模块结构框图与EPHY模块功能概述

MCF52233 以太网模块由以太网物理层收发器 EPHY 和快速以太网控制器 FEC 两个

^①寄生电容是指：由于电路板布线结构的互容和互感，造成的寄生在布线之间的电容。它通常是指电感、电阻、布线和芯片引脚等在高频情况下表现出来的电容特性。

子模块构成,共同实现以太帧的收发功能。EPHY 与 FEC 之间通过介质无关接口 (Medium Independent Interface, MII) 进行通信。MII 定义了 FEC 与各种物理层之间的标准电气接口,通过它完成 EPHY 与 FEC 之间数据通信,实现 EPHY 的配置及通信状态的判断。图 12-6 给出了以太网模块 EPHY 与 FEC 连接关系。

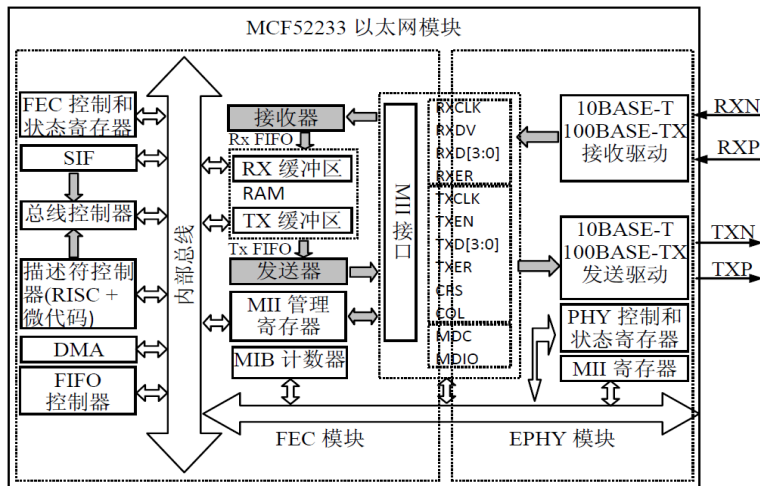


图12-6 以太网模块EPHY与FEC连接关系图

MCF52233 的 EPHY 是遵循 IEEE802.3 标准的 10BASE-T/100BASE-TX 以太网物理收发器,支持全/半双工模式,有自动协商功能,通过 MII 管理接口与 FEC 子模块进行通信。

2. EPHY模块编程寄存器

MCF52233 的 EPHY 提供了 3 个存储映像寄存器,即以太网物理收发器控制寄存器 0 (EPHYCTL0)、控制寄存器 1 (EPHYCTL1) 以及状态寄存器 (EPHYSR)。

1) 以太网物理收发器控制寄存器0 (Ethernet Physical Transceiver Control Register 0, EPHYCTL0)

地址为 0x1E_0000,主要用来设置 EPHY 使能、EPHY 层的自动协商、LED 信号使能以及 EPHY 的中断使能等。其中 D1 位为保留位,读为 0,写无意义。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	EPHYEN	ANDIS	DIS100	DIS10	LEDEN	EPHYWAI		EPHYIEN
复位	0	1	1	1	0	0	0	0

D7—EPHYEN, EPHY 使能位,可读写。EPHYEN =1,使能 EPHY 模块; EPHYEN =0,禁止 EPHY 模块。

D6—ANDIS, 自动协商禁止位,可读写。ANDIS =1,启动之后禁止自动协商,工作模式需要手动设置 MII 寄存器; ANDIS = 0,启动之后使能自动协商,工作模式自动实现。

D5—DIS100, 100BASE-TX 的 PLL 禁止位,可读写。DIS100=1,关闭 100BASE-TX 的 PLL; DIS100=0, EPHY 的操作模式决定 100BASE-TX 的 PLL 状态。

D4—DIS10, 10BASE-T 的 PLL 禁止位,可读写。DIS10=1,关闭 10BASE-T 的 PLL; DIS10=0, EPHY 的操作模式决定 10BASE-T 的 PLL 状态。

D3—LEDEN, LED 驱动使能位, 可读写。LEDEN =1, 使能 EPHY 的 LED 驱动信号; LEDEN =0, 禁止 EPHY 的 LED 驱动信号, 此时可以通过软件来驱动 LED 信号。

D2—EPHYWAI, WAIT 模式下 EPHY 模块停止位, 可读写。EPHYWAI =1, MCU 进入 WAIT 模式后禁止 EPHY 模块, EPHY 中断不能使 MCU 从 WAIT 模式跳出; EPHYWAI =0, WAIT 模式下允许 EPHY 模块继续工作。

D1—未使用。

D0—EPHYIEN, EPHY 中断使能位, EPHYIEN=1 允许中断, EPHYIEN=0 禁止中断。

2) 以太网物理收发器控制寄存器1 (Ethernet Physical Transceiver Control Register 1, EPHYCTL1)

地址为 0x1E_0001, 用来设置介质无关接口 (MII) 请求的 EPHY 地址 (PHYADD4 为最高位)。若使用内部 EPHY, 则该地址为 0。复位全为 0。

数据位	D7~D5	D4	D3	D2	D1	D0
定义		PHYADD4	PHYADD3	PHYADD2	PHYADD1	PHYADD0

3) 以太网物理收发器状态寄存器 (Ethernet Physical Transceiver Status Register, EPHYSR)

地址为 0x1E_0002, 用来存放 EPHY 模块状态, 包括网络通信速率及 EPHY 中断。

数据位	D7~D6	D5	D4	D3~D1	D0
定义		100DIS	10DIS		EPHYIF
复位	0	1	1	0	0

D7~D6, D3~D1—未使用。

D5—100DIS, EPHY 端口的 100BASE-TX 模式状态位, 只读。100DIS= 1, 已禁止; 100DIS= 0, 已使能。

D4—10DIS, EPHY 端口的 10BASE-T 模式状态位, 只读。10DIS =1, 已禁止; 10DIS =0, 已使能。

D0—EPHYIF, EPHY 中断标志位, 可读写。它表示是否发生了 EPHY 中断, 向该位写 1 可以清除中断标志, 写 0 无意义。具体发生何种中断需要通过 MII 管理接口读出中断控制寄存器的内容。EPHYIF =1, 发生了 EPHY 中断; EPHYIF =0, 未发生 EPHY 中断。

3. MII寄存器

除了上述 3 个存储映像寄存器外, MCF52233 的 EPHY 模块还提供了 19 个介质无关接口 (MII) 寄存器。表 12-4 给出了通过 MII 管理接口可以访问的以太网物理接口中的所有寄存器。这些寄存器不属于 MCU 的存储器映像空间, 所以不能直接操作, 只能通过 MII 接口进行配置和管理。其中, 常用的有 3 个 MII 寄存器, 分别为: EPHY 控制寄存器、EPHY 状态寄存器和中断控制寄存器, 对于其他寄存器请参见 MCF52233 的芯片手册。

表 12-4 MII 寄存器

编号	MI I 寄存器地址	寄存器名	访问权限
0~1	0x00/0x01	EPHY 控制/状态寄存器	读/写
2~3	0x02/0x03	EPHY 识别寄存器 1/2	读/写
4	0x04	自动协商发布寄存器	读/写
5	0x05	自动协商连接能力寄存器	读/写
6	0x06	自动协商扩展寄存器	读/写
7	0x07	自动协商下页发送	读/写
8~15	0x08~0x0F	保留	读/写
16	0x10	中断控制寄存器	读/写
17~18	0x11/012	专用状态/控制寄存器	读/写

1) EPHY控制寄存器（EPHY Control Register）

该 MII 寄存器的地址为 0x00，用来设置 EPHY 的自动协商使能、10/100Mbps 速度选择和全/半双工等工作模式。

数据位	D15	D14	D13	D12	D11	D10	D9	D8
定义	RESET	LOOPBACK	DATARATE	ANE	PDWN	ISOL	RAN	DPLX
复位	0	0	1	0/1	0	0	0	1
数据位	D7	D6~D0						
定义	COLTEST							
复位	0	0						

D15—RESET，EPHY 复位位。RESET=1，EPHY 将端口的状态和寄存器复位为默认值。复位之后，该位自动清 0。复位时间大约需要 1.3 毫秒；RESET=0，无影响。

D14—LOOPBACK，数字回环模式位。LOOPBACK=1，使能数字回环模式。在该模式下，TXD 数据直接发送到 RXD，EPHY 与传输介质独立；LOOPBACK=0，禁止数字自环模式。

D13—DATARATE，速度选择位。DATARATE=1，禁止自动协商时，选择 100Mbps 的速度；DATARATE=0，禁止自动协商时，选择 10Mbps 的速率。

D12—ANE，自动协商使能位。1-使能自动协商；0-禁止自动协商，此时由 DATARATE 和 DPLX 位定义连接配置。

D11—PDWN，低功耗位。1-端口处于低功耗模式；0-正常。

D10—ISOL，隔离位。1-将端口数据通道信号与 MII 隔离；0-正常。

D9—RAN，自动协商重启位。1-当允许自动协商（ANE=1）时，自动协商重新启动。0-正常模式。

D8—DPLX，双工模式位。当 LOOPBACK=1 时，该位对 EPHY 无影响。DPLX=1，表示全双工模式；DPLX=0，表示半双工模式。

D7—COLTEST，冲突检测位。该功能在 LOOPBACK=1 时才有效。COLTEST=1，强迫 EPHY 在 MII_TXEN（发送使能）信号有效后的 512 位时间内使 MII_COL（冲突检测）信号有效，以及在 MII_TXEN 信号无效后的 4 位时间内使 MII_COL 信号无效；COLTEST=0，正常模式。

2) EPHY状态寄存器 (EPHY Status Register)

该 MII 寄存器的地址为 0x01，用来指示 MII 所连接物理收发器的工作状态。其中，D10~D7 位未定义，读为 0，写无意义。

数据位	D15	D14	D13	D12	D11	D10	D9	D8
定义	100T4	100XFD	100XHD	10TFD	10THD			
复位	0	1	1	1	1	0	0	0
数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义		SUPPRE	ANCOMP	REMFLT	ANABL	LNKSTST	JABDT	EXCAP
复位	0	1	0	0	1	0	0	1

D15—100T4，EPHY 是否支持 100BASE-T4 传输。1-支持；0-不支持。注意，EPHY 模块未实现该功能。

D14—100XFD，EPHY 是否支持 100BASE-TX 全双工模式。1-支持；0-不支持。

D13—100XHD，EPHY 是否支持 100BASE-TX 半双工模式。1-支持；0-不支持。

D12—10TFD，EPHY 是否支持 10BASE-T 全双工模式。1-支持；0-不支持。

D11—10THD，EPHY 是否支持 10BASE-T 半双工模式。1-支持；0-不支持。

D6—SUPPRE，管理帧前导抑制位。1-管理帧不需前导位；0-需要前导位。

D5—ANCOMP，自动协商完成位。1-表明自动协商已处理，此时 MII 寄存器 4~7 有效；0-表明自动协商处理未完成。

D4—REMFLT，远程错误位。1-已检测到远程错误，读该寄存器或者 EPHY 复位后，该位被清 0；0-未检测到错误。

D3—ANABL，自动协商能力位。1-表示 EPHY 有自动协商能力；0-表示无该能力。

D2—LNKSTST，连接状态位。1-表示已建立有效连接；0-表示未连接。当连接失败时，该位被清 0。

D1—JABDT，超时传输 (Jabber) 检测位。对于 100BASE-TX 模式，该位被清 0。1-已检测到超时传输；0-未检测到超时传输。

D0—EXCAP，扩展能力位。1-EPHY 实现了扩展寄存器；0-未实现。

3) EPHY中断控制寄存器 (EPHY Interrupt Control Register)

该 MII 寄存器的地址为 0x10，用来设置 EPHY 中断使能和标识中断发生情况。该寄存器的 D15 和 D7 位为只读位，其他位可读写。复位全为 0。

数据位	D15	D14	D13	D12	D11	D10	D9	D8
定义		ACKIE	PRIE	LCIE	ANIE	PDFIE	RFIE	JABIE
数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义		ACKR	PGR	LKC	ANC	PDF	RMTF	JABI

D14—ACKIE，(从连接对象) 接收到响应位中断使能位。ACKIE=1，使能中断。

D13—PRIE，接收到新页中断使能位。PRIE=1，使能中断。

D12—LCIE，连接变化中断使能位。LCIE=1，使能中断。

D11—ANIE，自动协商变化中断使能位，表示自从最后一次访问该寄存器开始，自动协商的状态发生了变化，将产生中断。ANIE=1，使能中断。

D10—PDFIE，发生并行检测错误中断使能位。PDFIE=1，使能中断。

D9—RFIE，远程错误中断使能位。RFIE=1，使能中断。

D8—JABIE，检测到超时传输（Jabber）中断使能位。JABIE=1，使能中断。

D6~D0—分别对应上述中断使能的中断标志位，这些位为 1 时，表示发生了中断。

上述这些中断具有相同的系统中断向量（36 号中断），通过以太网物理收发器控制寄存器 0（EPHYCTL0）中的 EPHYIEN 位（第 0 位，EPHY 中断使能位）决定这些中断开启和关闭。若要清除中断标志需要先读中断控制寄存器后再将以太网物理收发器状态寄存器 EPHYSR 中的 EPHYIF 位写 1 即可。这 7 个中断里比较常用的是连接变化中断。需要注意的是 EPHY 模块的这三个寄存器不同于前面的以太网物理收发器的几个寄存器，它们不参与系统存储器统一编址，只能通过 MII 管理接口来访问。

12.2.3 第一个测试实例：网络连接

按照构件的设计原则，将 EPHY 模块的驱动程序封装成 EPHY 构件。

EPHY 构件对外接口函数：

- (1) uint8 EPHY_Init(uint16 sysClk, uint8 rateMode, uint8 DupMode, uint8 autoNeg);
//初始化以太网模块
- (2) uint8 EPHY_LinkState(void); //检测网络是否已连接

EPHY 构件内部函数：

- (1) void MII_init(int32); //MII 初始化
- (2) int32 MII_write(int32, int32, int32); //写 MII 寄存器
- (3) int32 MII_read(int32, int32, uint16*); //读 MII 寄存器

1. EPHY构件的初始化函数EPHY_Init

```
//-----*
//函数名: EPHY_Init *
//功 能: 初始化EPHY模块 *
//参 数: uint16 sysClk = 当前系统总线时钟(MHz) *
//      uint8 rateMode: 0:100M, 非0:10M *
//      uint8 DupMode: 0:半双工, 非0:全双工 *
//      uint8 autoNeg: 0:禁止自动协商, 非0:启用自动协商 *
//返 回: 1:成功 0:配置失败 *
//内部调用: MII_init() - MII初始化, MII_write(int32, int32, int32) - 写MII寄存器 *
//      MII_read(int32, int32, uint16*) - 读MII寄存器 *
//说 明: 若配置为自动协商功能则前面所设速率等参数就无效, 取决于协商结果 *
//注 意: 在通过MII接口写MII中断控制寄存器时应延时一段时间以等待MII接口稳定, *
//      否则可能导致写操作失败, 则无法产生EPHY中断, 由于编译器的优化功能未 *
//      声明函数可能也可以通过编译, 但实际没有执行, 编程时需注意 *
//-----*
uint8 EPHY_Init(uint16 sysClk, uint8 rateMode, uint8 DupMode, uint8 autoNeg)
{
    uint16 cfgData, regData, autoData;
    uint32 i;
```

```

//1. 初始化MII接口控制器, 根据总线频率配置EPHY与FEC模块的通信频率为2.5MHz
MII_init(sysClk);
//2. 配置EPHY地址, 使用内部EPHY(要求参数为0)
MCF_PHY_EPHYCTL1 = MCF_PHY_EPHYCTL1_PHYADDR(0x00);
//3. 由EPHY的操作模式决定100BASE-TX/10BASE-T的PLL状态(EPHYCTL0的D5, D4为0)
MCF_PHY_EPHYCTL0 = (uint8)(MCF_PHY_EPHYCTL0
    & ~(MCF_PHY_EPHYCTL0_DIS100|MCF_PHY_EPHYCTL0_DIS10));
//4. 启动时, 禁止自动协商(EPHYCTL0的D6为1)
MCF_PHY_EPHYCTL0 = (uint8)(MCF_PHY_EPHYCTL0 | MCF_PHY_EPHYCTL0_ANDIS);
//5. 使能EPHY模块和EPHY指示灯(EPHYCTL0的D7, D3位为1)
MCF_PHY_EPHYCTL0 = (uint8)(MCF_PHY_EPHYCTL0 | MCF_PHY_EPHYCTL0_EPHYEN
    | MCF_PHY_EPHYCTL0_LEDEN);
//6. 根据参数通过MII管理接口配置EPHY的MII寄存器, 如:100Mbps, 半双工, 禁止自
// 动协商(控制寄存器), 开启EPHY的各种中断(中断控制寄存器)
// 6.1 读内部PHY_REG_CR, 赋给regData;读内部PHY_REG_ANAR赋给autoData
while(!(MII_read(0x00, PHY_REG_CR, &regData)));
while(!(MII_read(0x00, PHY_REG_ANAR, &autoData)));
// 6.2 根据入口参数autoNeg改变临时变量regData、autoData
if(autoNeg==0) //若是禁止自动协商
{
    regData &= ~0x1000; // 用户模式, 禁止自动协商

if(rateMode==0)
    regData |=
0x2000; // 100Mbps
else
    regData &=
~0x2000; // 10Mbps
if(DupMode==0)
    regData &=
~0x0100; // 半双工
else
    regData |=
0x0100; // 全双工
}
else //若使能自动协商功能
{
    //启动自动协商(EPHYCTL0的D6为0)
    MCF_PHY_EPHYCTL0 = (uint8)(MCF_PHY_EPHYCTL0 & ~MCF_PHY_EPHYCTL0_ANDIS);
    regData |= 0x1000;
    autoData =
0x01E0;
}
//延时等待EPHY启动完成
for (i=10000; i >0; i--);
// 6.3 通过MII接口写配置数据到控制寄存器和自动协商发布寄存器
while(!(MII_write(0x00, PHY_REG_CR, regData)));
while(!(MII_write(0x00, PHY_REG_ANAR, autoData)));

```

```

        for(i=0;i<60000;i++); //延时等待MII接口稳定时再写中断配置数据
        //7. 配置EPHY模块的有关中断
        cfgData = PHY_R16_ACKIE | PHY_R16_PRIIE | PHY_R16_LCIE | PHY_R16_ANIE;
        cfgData = cfgData | PHY_R16_PDFIE | PHY_R16_RFIE | PHY_R16_JABIE;
        while(!(MII_write(0x00, PHY_REG_IR, cfgData)));
        MCF_INTC0_ICR36 = MCF_INTC_ICR_IL(3); //36号中断的level=3
        MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK36; //开放36号中断
        EPHYIntDisEnable(); //EPHY中断禁止
        //8. 延时, 等待初始化稳定

for( i=0;i<250000;
i++ )
{
    while(!(MII_read(0x00, PHY_REG_SR, &regData)));
    if( regData & 0x0004 )
    {
        return 1; //成功
    }
}
return 0; //不成功
}

```

2. 获取网络连接状态函数 EPHY_LinkState

```

//-----*
// 函数名: EPHY_LinkState *
// 功 能: 检测网络是否已连接 *
// 参 数: 无 *
// 返 回: 有连接返回1, 无连接返回0 *
//-----*
uint8 EPHY_LinkState(void)
{
    uint16 reg = 0;
    //1. 读取网络连接状态
    while(!(MII_read(0x00, PHY_REG_SR, &reg)));
    //2. 判断是否连接
    if( reg & 0x0004 )
        return 1;
    else
        return 0;
}

```

3. EPHY模块测试主函数

```

//-----*
// 工 程 名: EPHY *
// 硬件连接: (1) MCF52233的串口与PC机串口相连 *
//            (2) MCF52233的网口与PC机网口相连 *
// 功能描述: (1) 若PC机屏幕右下角“网络连接状态”图标显示已连上, *

```

```

//          表示MCF52233的EPHY模块初试化成功                                *
//          (2) 还可以通过串口调试工具查看连接状态                          *
//          收到“Link OK”表示已连接；否则收到“Link Fail”                    *
//          (3) 当连接建立则会产生连接变化中断，PC串口收到“EPHY_INT”        *
//-----苏州大学飞思卡尔嵌入式系统实验室2009年-----*
#include "includes.h" //头文件
void main(void)
{
    //1. 关总中断
    uint8 linkStatus, result;
    DisableInterrupts();
    //2. 模块初始化
    //小灯初始化
    Light_Init(Light_Link_PORT, Light_Link, Light_OFF);
    //UART初始化，使用UART0模块，使用系统时钟作为UART的时钟源
    //波特率为9600，初始化时禁止接收中断
    UartInit(0, SYSTEM_CLOCK, 9600, 0);
    //EPHY初始化，初始配置为100Mbps，半双工，禁止自动协商
    result = EPHY_Init(SYSTEM_CLOCK, 0, 0, 0);
    EPHYintEnable(); //EPHY中断使能
    //3. 开总中断
    EnableInterrupts();
    //4. 主循环
    while (1)
    {
        Light_Control(Light_Link_PORT, Light_Link, Light_ON);
        Delay(10);
        Light_Control(Light_Link_PORT, Light_Link, Light_OFF);
        Delay(10);
        //获取网络连接状态
        linkStatus = EPHY_LinkState();
        if(linkStatus)
            UartSendString(0, "Link OK.\r\n"); //已连接
        else
            UartSendString(0, "Link Fail.\r\n"); //未连接
    }
}

```

4. EPHY模块测试方法

为了加深对上述寄存器的理解，下面给出第一个测试实例：使用交叉网线将 MCF52233 设备与 PC 机相连接。

程序运行现象说明：

将灌入程序的目标板通过网线与 PC 直接相连后，对目标板上电使程序运行。

(1) 现象 1：若连接成功，则 PC 机右下脚显示标志，否则显示标志。当然 PC 机要启动本地连接，且设置允许网络连接状态显示（可设置“本地连接”属性对话框中“常规”选项卡下方“ 连接后在通知区域显示图标 (N)”及“ 此连接被限制或无连接时通知我 (M)”）。

(2) 现象 2: MCF52233 目标板上的连接状态指示灯 LNKLED 亮, 表明建立的连接有效。

(3) 现象 3: 若通过串行线将 MCF52233 目标板串口与 PC 机串口相连, 在 PC 机上运行串口调试工具软件, 可以看到目标板发出的字符串 “Link OK.”, 表示连接成功, 否则发出字符串 “Link Fail.”

(4) 现象 4: 样例程序开放 EPHY 中断, 当网络连接状态发生改变时 (LNKLED 由亮变暗或由暗变亮) 产生连接变化中断, 在中断处理函数 (isr.c) 中向 PC 机发送 “EPHY_INT”, 打开 PC 机上的串口调试工具可查看现象。读者可通过样例程序中的使能和禁止宏操作来使能或禁止 EPHY 中断。

工作原理说明:

PC 网卡 (一般都具有自动协商功能) 通过发送链路脉冲信号来协调与目标板兼容的速率和工作方式, 本书样例程序初始化 EPHY 为 100Mbps、半双工模式。所以最后双方的共同模式为 100Mbps、半双工, PC 方右下角显示连接速率为 100Mbps。

在实际使用中, 若自动协商不能确定通信对方的工作方式, 则可通过编程测试, 然后确定实际工作方式, 而不采用自动协商功能。注意连接双方最好是同时支持并开启自动协商功能, 否则可能会产生全半双工不匹配问题。

另外, 该样例程序不向 PC 方发送以太帧数据。读者可以通过查看 PC 方的本地连接属性看出, PC 只向外发送数据, 但并未收到目标板的返回数据; 也可以使用本章光盘资料中的网络捕包工具 Ethereal 或 CommView 来查看 PC 收到和发出的数据。PC 机发出的数据有 ARP 数据包、NBNS 协议数据包和 IGMP 数据包, 发送这些包的目的是为了获取连接对端及所在工作组的运行情况。

12.2.4 快速以太网控制器 FEC 的编程基础

1. FEC 模块功能概述

ColdFire 系列芯片集成了快速以太网控制器 FEC, 它将硬件和微程序相融, 实现了以太网介质访问控制 (Media Access Control, MAC) 功能。硬件上包括各种寄存器、发送和接收 FIFO、DMA 控制器和 RISC (精简指令集) 控制器等。微代码由嵌入在 FEC 模块内的 RISC 控制器执行, 负责解释发送和接收缓冲区描述符中的内容, 并联合 DMA 控制器, 将数据包从发送缓冲区发送出去或将收到的数据放入接收缓冲区中。这些缓冲区及其对应的缓冲区描述符都是位于 FEC 以外的系统存储区 (关于缓冲区描述符的功能将在后面详细介绍)。FEC 模块通过工业标准的 MII 接口与片内 EPHY 进行通信。

除了收发以太帧, MCF52233 的 FEC 模块还具有以下功能: 操作 MII 接口、地址识别、全双工流控制、冲突处理、内部和外部回环模式等。FEC 支持 10/100Mbps 以太网 MII 接口。地址识别是指 FEC 针对接收帧的目标地址域 (DA) 的过滤处理, 它实现单播、组播或广播帧的接收。全双工流控制允许用户软件发送并检测暂停帧。若使能了暂停帧检测, 当接收器检测到暂停帧后, 发送器将暂停, 根据暂停帧中的时间信息, 延时后再继续

发送。若在发送帧时，有冲突发生，则 FEC 很快将传输由 32 个“1”组成的堵塞信号，随机延时一段时间后再发送。FEC 支持内部和外部回环。内部回环模式用于自测，在该模式下，发送和接收 FIFO 都被使用，FEC 实际上操作于全双工方式。

2. FEC模块编程寄存器

MCF52233 的 FEC 模块共有 23 个 32 位寄存器，占用 92 字节存储空间。FEC 寄存器的存储映像如表 12-5 所示。

表 12-5 FEC 寄存器的存储映像

偏移地址	寄存器名	访问权限
0x1004	以太网中断事件寄存器（EIR）	读/写
0x1008	中断屏蔽寄存器（EIMR）	读/写
0x1010/0x1014	接收/发送描述符激活寄存器（RDAR/TDAR）	读/写
0x1024	以太网控制寄存器（ECR）	读/写
0x1040	MII管理帧寄存器（MMFR）	读/写
0x1044	MII速度控制寄存器（MSCR）	读/写
0x1064	MIB控制寄存器（MIBC）	读/写
0x1084/0x10C4	接收/发送控制寄存器（RCR/TCR）	读/写
0x10E4/0x10E8	物理地址低/高位寄存器（PALR/PAUR）	读/写
0x10EC	操作码/暂停时间寄存器（OPD）	读/写
0x1118/0x111C	单播哈希表高/低地址寄存器（IAUR/IALR）	读/写
0x1120/0x1124	组播哈希表高/低地址寄存器（GAUR/GALR）	读/写
0x1144	发送FIFO水位寄存器（TFWR）	读/写
0x114C/0x1150	接收FIFO界限/起始寄存器（FRBR/FRSR）	读/写
0x1180/0x1184	接收/发送描述符环开始寄存器（ERDSR/ETDSR）	读/写
0x1188	接收缓冲区大小寄存器（EMRBR）	读/写

本节仅介绍后面实例用到的部分 FEC 寄存器，其他寄存器请参考芯片手册。

1）以太网中断事件寄存器（Ethernet Interrupt Event Register，EIR）与中断屏蔽寄存器（Interrupt Mask Register，EIMR）

数据位	D31	D30	D29	D28	D27	D26	D25	D24
定义	HBERR	BABR	BABT	GRA	TXF	TXB	RXF	RXB
数据位	D23	D22	D21	D20	D19	D18~D0		
定义	MII	EBERR	LC	RL	UN			

以太网中断事件寄存器的地址为 0x1004，可读写，写 1 将清零中断标志，写 0 无影响。它包含了 FEC 提供的 13 种中断事件的标志位，通过这些标志可以判断发生了何种中断。D18~D0 为保留位。复位全为 0。

D31—HBERR，心跳信号错误标志位^①。如果发送控制寄存器 TCR 中的 HBC 被置位，且一帧传送完成后，FEC 在心跳窗口内（4 微秒内）没有检测到心跳条件，则心跳错误发生。当发生该错误时，FEC 关闭缓冲区，置 EIR 的 HBERR 位，若心跳错误中断被使能，则产生该中断。HBERR=1，产生心跳错误中断。

^①所谓心跳信号，是指某些收发器具有的自我测试特性。为实现该测试，以太网控制器完成帧传输的 4 微秒内，发送器释放一个冲突信号。该信号并不意味着网络上存在真正的冲突，但确是发送器看似运行正常的指示。这被称为心跳条件。

D30—BABR, 接收串扰错误位。BABR=1, 表示接收到的帧长度大于接收控制寄存器 RCR 的 MAX_FL 位域的值。该错误通常发生网络产生在故障或不规则的网络传输(例如有一台不间断的发送数据帧的网络终端设备)。

D29—BAPT, 发送串扰错误位。BAPT=1, 表示发送帧的长度大于接收控制寄存器 RCR 的 MAX_FL 位域的值。该错误通常发生在过长的帧被放入发送缓冲区。过长的帧并不被截断, 而是全部被发送出去。

D28—GRA, 优雅停止完成(Graceful stop complete)位。GRA=1, 表示完成当前帧发送后, 且没有发生错误, 发送器进入暂停状态。

D27—TXF, 发送帧中断位。TXF=1, 表示帧已发送完成且最后缓冲区描述符已更新。

D26—TXB, 发送缓冲区中断位。TXB=1, 表示已更新了发送缓冲区描述符。

D25—RXF, 接收帧中断位。RXF=1, 表示已接收到帧, 且最后缓冲区描述符已更新。

D24—RXB, 接收缓冲区中断位。RXB=1, 表示已更新了接收缓冲区描述符。

D23—MII, MII 中断位。MII=1, 表示 MII 已完成了所请求数据的传输。

D22—EBERR, 以太网总线错误位。EBERR=1, 表示在进行 DMA 传输时发生了以太网总线错误。

D21—LC, 滞后冲突位。LC=1, 即在半双工模式下, 冲突发生在 512 位时间后, FEC 停止传输。这时, 所有该帧的剩余缓冲区被清除并关闭, LC 位置位。若中断屏蔽寄存器 EIMR 寄存器 LC 中断使能, 则该中断发生。

D20—RL, 冲突重试限制位。RL=1, 表示在半双工模式下, 冲突检测次数超过了最大重发次数 16, 此时该帧被丢弃。

D19—UN, 发送 FIFO 提前清空位。UN=1, 表示在发送完成前, 发送 FIFO 已经为空。这时, 错误的 CRC 将被添加到碎帧(Fragment)之后, 且帧剩余部分将被丢弃。

以太网中断事件寄存器 EIR 中所有中断事件由中断屏蔽寄存器(Interrupt Mask Register, EIMR)允许中断或禁止中断。EIMR 的地址为 0x1008, 可读写。与 EIR 的每一位相对应, 1-表示允许相应中断, 0-表示禁止相应中断。

FEC 的 13 个中断拥有 13 个中断向量(23-35 号中断)。要开启这些中断, 需要设置以太网中断屏蔽寄存器 EIMR 中的相应位, 而以太网中断事件寄存器 EIR 则标志了各个中断发生的状态。当需要对相应中断进行处理时, 可以分别编写中断处理程序。读者若想了解具体中断的功能, 请参见芯片手册。在本书实例中, 主要使用了 2 个中断: 发送帧中断和接收帧中断。

2) 接收/发送描述符激活寄存器(Receive/ Transmit Descriptor Active Register)

接收描述符激活寄存器(RDAR)地址为 0x1010, 可读写, 用来指示接收描述符环已经更新, 即有空闲缓冲区可以供接收数据使用。向该寄存器写入任何值都会将 D24 位(R_DES_ACTIVE)置 1; 当接收环中没有空闲描述符时, FEC 将该位清 0。

发送描述符激活寄存器(TDAR)地址为 0x1014, 可读写, 用来指示发送描述符已经更新, 即发送缓冲区已准备就绪。向该寄存器写入任何值都会将 D24 位(X_DES_ACTIVE)置 1; 当发送环中没有空闲描述符时, FEC 将该位清 0。

3) 以太网控制寄存器 (Ethernet Control Register, ECR)

地址为 0x1024, 可读写, 通常由用户置 1, 硬件清 0。ECR 用于使能、禁止和复位 FEC 模块。D31~D2 位: 未定义。

D1—ETHER_EN, 以太网使能位。ETHER_EN=1, 使能 FEC。ETHER_EN=0, 立即停止接收操作, 发送操作在当前发送帧中加入 CRC 校验错误后停止。若软件将 ECR 的 RESET 位置 1 或错误条件导致 EIR 的 EBERR 位置 1, 硬件会自动清 ETHER_EN 位, 停止以太网接收操作, 所有其他的 FEC 寄存器被初始化为复位值。在复位期间由硬件自动清零该位。

D0—RESET, FEC 复位位。RESET=1, 执行 FEC 硬件复位, ETHER_EN 被清 0, 所有其它 FEC 寄存器被初始化为复位值。在复位期间由硬件自动清零该位, 复位完成需要至少 8 个系统时钟周期。

4) MII 管理帧寄存器与 MII 速度控制寄存器

MI 管理帧寄存器 (MI Management Frame Register, MMFR), 地址为 0x1040。MMFR 用来与所连接的 MII 兼容的 PHY 设备进行通信, 提供对那些 MII 寄存器的读写控制。通过 MMFR 访问 MII 寄存器之前, 软件应该先查询 EIR 的 MII 位, 以确保当前没有对 MII 的访问。

数据位	D31~D30	D29~D28	D27~D23	D22~D18	D17~D16	D15~D0
定义	ST	OP	PA	RA	TA	DATA

D31~D30—ST, 帧分隔符的开始位。对于有效的 MII 管理帧, 必须将该位编程为 01。

D29~D28—OP, 操作码位。为了产生有效的 MII 管理帧, 该字段必须设置为 10 (读) 或 01 (写); OP=11 产生读帧操作, OP=00 产生写帧操作, 但这些帧不遵循 MII 接口原则。

D27~D23—PA, 物理地址位。该位对应于 PHY 设备号, 可设置为 0~31。

D22~D18—RA, 寄存器地址位。该位对应于具体 PHY 设备内的 32 个寄存器号。

D17~D16—TA, 回转位。该位必须设置为 10, 以产生有效的 MII 管理帧。

D15~D0—DATA, 管理帧数据位。存放写入 PHY 寄存器或从 PHY 寄存器读出的数据。

MI 速度控制寄存器 (MI Speed Control Register, MSCR), 地址为 0x1044, 可读写。MSCR 提供对 MII 时钟 (EMDC 引脚) 频率的控制。

D7—DIS_PREAMBLE, 前导符禁止位。DIS_PREAMBLE=1, MII 管理帧未加前导符。只要所连的 PHY 不要求, MII 标准允许去掉前导符。

D6~1—MII_SPEED, MII 速度位。MII_SPEED 控制 MII 管理接口相对于系统时钟的频率。MII_SPEED=0, 关闭以太网模块时钟 (EMDC), 使其处于低电压状态。否则, EMDC 的频率为系统时钟频率的 $1/(MII_SPEED*2)$ 。

5) MIB 控制寄存器 (MIB Control Register, MIBC)

FEC 模块内部有个被称之为报文信息块 (Message Information Block, MIB) 的内部机构, 可以给出各种网络事件和统计值的计数。MIB 对 FEC 来说并不是必须的, 但它提供了有意义的网络管理计数。

MIB 控制寄存器的地址为 0x1064, 可读写。如果需要清空所有 MIB 计数器, 应该先

通过该寄存器禁止 MIB 块，然后清空所有 MIB 计数器，最后再使能 MIB 块。

D31—MIB_DISABLE，MIB 禁止位，该位可读写。MIB_DISABLE=1，MIB 逻辑停止，不更新任何 MIB 计数器。

D30—MIB_IDLE，MIB 空闲位，该位只读。MIB_IDLE=1，MIB 块当前没有更新 MIB 计数器。

6) 接收控制寄存器 (Receive Control Register, RCR)

地址为 0x1084，可读写。RCR 寄存器由用户编程，控制接收模块的操作模式，仅在 ECR 的 ETHER_EN 位=0（初始化期间）时才可修改。

数据	D26~D16	D15~D6	D5	D4	D3	D2	D1	D0
定义	MAX_FL		FCE	BC_REJ	PROM	MII_MODE	DRT	LOOP
复位	10111101110	0	0	0	0	0	0	1

D31~D27—未定义。

D26~D16—MAX_FL，最大帧长位。该长度为从目标地址开始到帧尾的 CRC 的长度。复位后为 1518。发送帧超过 MAX_FL 会产生发送错误位 (BABT) 中断；接收帧超过 MAX_FL 会产生接收错误位 (BABR) 中断。

D5—FCE，流量控制使能位。FCE=1，接收器检测 PAUSE（暂停）帧。

D4—BC_REJ，广播帧拒绝位。BC_REJ=1，如果 PROM=0，则拒绝接收目标地址为 FF_FF_FF_FF_FF_FF 的帧；如果 PROM=0，则接收广播帧，并设置接收缓冲区描述符的 M (MISS) 位为 1。

D3—PROM，混合模式。PROM=1，无论帧的目的地址是否与本机匹配，都被接收。

D2—MII_MODE，MII 模式位。用于选择外部接口模式。MII_MODE=1，选择 MII 模式。MII_MODE=0，选择 7 线模式（一般不用）。

D1—DRT，发送时接收禁止位。DRT=1，在发送时，禁止接收帧（通常用于半双工模式）。DRT=0，接收与发送操作相互独立（用于全双工模式）。

D0—LOOP，内部回环位。LOOP=1，发送帧被回送到设备内部，无输出信号。此时必须将 DRT 设置为 0。内部回环功能用于自测。

7) 发送控制寄存器 (Transmit Control Register, TCR)

地址为：0x10C4。TCR 寄存器由用户编程来配置发送块。系统复位时，该寄存器被清 0。第 1 位和第 2 位仅在 ECR[ETHER_EN]等于 0（初始化期间）时才可修改。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	—	—	—	RFC_PAUSE	TFC_PAUSE	FDEN	HBC	GTS
复位	0	0	0	0	0	0	0	0

D4—RFC_PAUSE，暂停帧接收控制位。只读状态位。RFC_PAUSE=1，表示已接收到全双工流量控制暂停帧，且发送器被暂停，持续时间为暂停帧中定义的持续时间。暂停时间结束后，RFC_PAUSE 自动清 0。

D3—TFC_PAUSE，暂停帧发送控制位。TFC_PAUSE=1，MAC 将当前数据帧发送结束后停止发送，然后发送一个 PAUSE 帧，最后将该位清 0 后，继续发送数据帧。

D2—FDEN，全双工使能位。该位必须在 ETHER_EN 之前进行设置（FDEN=1）。

D1—HBC，心跳控制位。该位必须在 ETHER_EN 前进行设置（HBC=1），发送结束后执行心跳检查。

D0—GTS，优雅发送停止（Graceful transmit stop）位。当该位置 1 时，在完成当前数据传输后 MAC 停止传输数据，同时以太网中断事件寄存器 EIR 的 GRA 位置 1。清零该位可重新开始传输数据。

8) 物理地址寄存器（Physical Address Register）

物理地址寄存器分为低位寄存器（PALR）和高位寄存器（PAUR）。

低位寄存器的地址为 0x10E4，用于设置 48 位 MAC 地址的低 32 位。

高位寄存器的地址为 0x10E8，D31~D16 用于设置 48 位 MAC 地址的高 16 位，D15~D0 为类型位，即 PAUSE 帧的类型字段，为常量 0x8808。这两个寄存器必须由用户初始化。

9) 单播哈希表地址寄存器（Descriptor Individual Address Register）

单播哈希表地址寄存器分为高位寄存器（IAUR）和低位寄存器（IALR）。

高位寄存器地址为 0x1118，用来设置 64 位单播地址哈希表的高 32 位。该寄存器的 D31 位对应的是哈希索引的第 63 位，D0 位对应的是哈希索引的第 32 位。

低位寄存器地址为 0x111C，用来设置 64 位单播地址哈希表的低 32 位。该寄存器的 D31 位对应的是哈希索引的第 31 位，D0 位对应的是哈希索引的第 0 位。

10) 组播哈希表地址寄存器（Descriptor Group Address Register）

组播哈希表地址寄存器分为高位寄存器（GAUR）和低位寄存器（GALR）。

高位寄存器地址为 0x1120，用来设置 64 位组播地址哈希表的高 32 位。该寄存器的 D31 位对应的是哈希索引的第 63 位，D0 位对应的是哈希索引的第 32 位。

低位寄存器地址为 0x1124，用来设置 64 位组播地址哈希表的低 32 位。该寄存器的 D31 位对应的是哈希索引的第 31 位，D0 位对应的是哈希索引的第 0 位。

11) 接收/发送描述符环开始寄存器（ERDSR/ETDSR）

接收描述符环开始寄存器（ERDSR）地址为 0x1180，用来配置接收缓冲区描述符队列的起始地址，该地址必须满足 32 位地址对齐。

发送描述符环开始寄存器（ETDSR）地址为 0x1184，用来配置发送缓冲区描述符队列的起始地址，该地址也必须满足 32 位地址对齐。

这两个寄存器不会被复位，必须由用户初始化。

12) 接收缓冲区大小寄存器（Receive Buffer Size Register, EMRBR）

该寄存器地址为 0x1188，用来设置接收缓冲区的大小。为了使每个缓冲区都能存放一个最大帧，EMRBR 必须设置为大于等于接收控制寄存器 RCR 的 MAX_FL 位。EMRBR 不会被复位，必须由用户初始化。

3. 发送先进先出缓冲区Tx FIFO和接收先进先出缓冲区Rx FIFO

在 MCF52233 的 FEC 模块内部有 512 字节先进先出缓冲区 FIFO, 其中包括发送 FIFO 和接收 FIFO, 分别记为 Tx FIFO 和 Rx FIFO。在使能了 FEC 后, 通过使能发送描述符激活寄存器, FEC 硬件自动将系统存储区中发送缓冲区的数据通过 Tx FIFO 传送到发送硬件。同样, 通过接收硬件可自动将 Rx FIFO 中的数据传送到系统存储区中的接收缓冲区。

TxFIFO 和 Rx FIFO 的存储空间是相连的, Tx FIFO 安排在 Rx FIFO 的前面, Tx FIFO 从 0 开始, 直到由 FIFO 接收起点寄存器 FRSR 的 R_FSTART 位域所指定数值, 而 Rx FIFO 则从该值开始, 直到由 FIFO 接收边界寄存器 FRBR 的 R_BOUND 位域所指定数值; 这两个位域都是 8 位的字段。另外, 在 FIFO 发送水位寄存器 TFWR 中, 还具有可编程的水位标志 X_WMRK, 该位域占 2 位, 用于设置当系统从存储区传送数据到 Tx FIFO 的字节数达到它所指定的值 (称作水位值) 时, 系统才启动硬件发送。水位值为 64 字节、128 字节、192 字节分别对应水位标志 X_WMRK=0x、10、11。

4. 以太网发送与接收缓冲区描述符 (TxBD/RxBD)

为了收发数据, FEC 的 DMA 需要将收到的数据放在接收缓冲区中, 或从发送缓冲区中获取需要发送的数据, 这些缓冲区都是位于 FEC 模块以外的系统存储空间内, 通常为片内 RAM, 如图 12-7。但也可以是其他任何 DMA 可以访问到的存储空间。每个缓冲区的位置和大小由相应的缓冲区描述符 (Buffer Descriptor, 以下简称 BD) 定义。

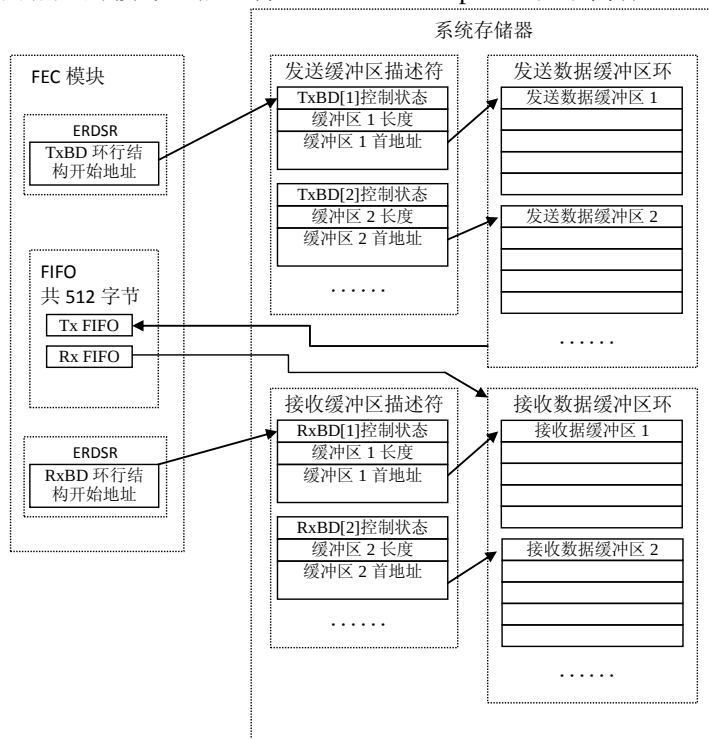


图12-7 缓冲区描述符和数据缓冲区

为了给用户提供最大限度的灵活性, 每个缓冲区描述符 BD 也是位于 FEC 以外的存

存储空间,由用户自己定义其位置。缓冲区描述符 BD 的大小是固定的,共 8 个字节,包括: 2 字节状态和控制信息、2 字节缓冲区大小、4 字节指向缓冲区起始地址的指针。

FEC 需要通过缓冲区描述符 BD 来管理与其对应的数据缓冲区。所有缓冲区描述符 BD 的大小相同,用户软件需将发送或接收缓冲区描述符 BD 存放在连续的存储空间,然后将第一个缓冲区描述符 BD 的首地址写入发送或接收描述符环开始寄存器(ETDSR/ERDSR)中,并设置相应环型标志,则这些连续的描述符序列就组成了一个“环”型结构供 FEC 使用。

若要将描述符序列设置成环型结构,则需把序列的最后描述符中控制/状态字节的 W 位置位。例如:若开辟两个发送缓冲区组成的“环”,就需要将第 2 个描述符中的 W 位置位,而第 1 个描述符中的 W 位清零,这样 FEC 就会将它们视为一个“描述符环”。另外需要注意的是,由于发送或接收描述符环开始寄存器(ETDSR/ERDSR)的低 2 位都为 0,所以第一个缓冲区描述符 BD 首地址必须按 4 字节对齐。

1) 以太网接收缓冲区描述符 (RxBD)

每个接收缓冲区描述符对应一个接收缓冲区,其格式如下:

	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
偏移+0	E	RO1	W	RO2	L	—	—	M	BC	MC	LG	NO	—	CR	OV	TR
偏移+2	数据长度															
偏移+4	Rx 数据缓冲区指针—A[31:16]															
偏移+6	Rx 数据缓冲区指针—A[15:0]															

E—空标志位,用户置 1, FEC 清 0。E=0,表示该 BD 的数据缓冲区中已经接收到数据。E=1,表示该 BD 所对应的数据缓冲区为空。

RO1—接收软件所有权 1 位。该位为软件保留,不会被硬件修改也不影响硬件。

W—环型控制位,由用户控制。W=0,表示下一个缓冲区描述符在紧跟的连续位置。W=1,表示下一个缓冲区描述符在 ERDSR 中定义的位置。

RO2—接收软件所有权 2 位。该位为软件保留,不会被硬件修改也不影响硬件。

L—帧结尾标志位,由 FEC 控制。L=0,该缓冲区不是接收帧的最后缓冲区。L=1,该缓冲区是接收帧的最后一个缓冲区。

M—忽略标志位,由 FEC 写入。在混合模式下,不是因为地址识别成功而接收到数据帧时,该位被置 1;若由于地址识别接收到数据帧,该位被置 0。用户可通过此位快速判定该数据帧是否是发送到本机。该位仅在 L 和接收控制寄存器(RCR)的 PROM 位为 1 时才有效。

BC—广播地址位。BC=1,表示目的地址为广播地址(即 FFFF-FFFF-FFFF)。

MC—多播地址位。MC=1,表示目的地址为多播地址,而不是广播地址。

LG—接收帧长度错误位,由 FEC 控制。LG=1,表示帧长度大于 RCR 的 MAX_FL 位域值。该位仅在 L 位置 1 时有效。

NO—接收到非 8 位对齐帧位,由 FEC 控制。表示接收到的数据帧的位数不能被 8 整除。如果该位置 1,下面的 CR 位就不会被置 1。

CR—接收 CRC 错误位,由 FEC 控制。该位仅在 L 位置 1 时才有效。

OV—溢出位,由 FEC 控制。表示在接收帧时出现接收 FIFO 超出范围错误。如果 OV=1,那么其他状态位如 M、LG、NO、CR 均为 0。该位仅在 L 位置 1 时才有效。

TR—截断位。TR=1,表示接收帧被截断(帧长>2047 字节)。此时,应该丢弃该帧,且忽略其他错误位。

数据长度—由 FEC 控制。如果 L=0,该数据长度为 FEC 写入到 BD 的数据缓冲区的字节数;如果 L=1,则该数据长度为帧的长度(包括 CRC 字段)。

A[31:16]—接收数据缓冲区指针的第 31 到 16 位。

A[15:0]—接收数据缓冲区指针的第 15 到 0 位。

2) 以太网发送缓冲区描述符(TxBD)

每个发送缓冲区对应一个发送缓冲区描述符,其格式如下:

	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
偏移+0	R	TO1	W	TO2	L	TC	ABC									
偏移+2	数据长度															
偏移+4	Tx 数据缓冲区指针 - A[31:16]															
偏移+6	Tx 数据缓冲区指针 - A[15:0]															

R—准备就绪位。该位由用户置 1, FEC 清 0。R=0,表示该 BD 的数据缓冲区还没有作好发送准备。R=1,表示该 BD 的数据缓冲区已经准备好发送。

TO1—发送软件所有权 1 位。该位为软件保留,不会被硬件修改也不影响硬件。

W—环型设置位,由用户控制。W=0,表示下一个缓冲区描述符在紧跟的连续的位置。W=1,表示下一个缓冲区描述符在 ETDSR 中定义的位置。

TO2—发送软件所有权 2 位。该位为软件保留,不会被硬件修改也不影响硬件。

L—帧的最后缓冲区位,由 FEC 控制。L=0,该缓冲区不是发送帧的最后一个缓冲区。L=1,该缓冲区是发送帧的最后一个缓冲区。

TC—发送 CRC 位。由用户控制,仅在 L=1 时才有效。TC=0,发送完最后一个数据字节后则立即停止。TC=1,发送完最后一个数据字节后发送 CRC 序列。

ABC—添加错误 CRC 序列。由用户控制,仅在 L=1 时才有效。ABC=0,无影响。ABC=1,发送完最后一个数据字节之后,发送错误的 CRC 序列(与 TC 取值无关)。

数据长度—由用户控制。该数据长度表示 FEC 应该从 BD 的数据缓冲区中发送的数据字节数,它不会被 FEC 修改。DMA 使用 D15~D5 位,D4~D0 位被忽略。

A[31:16]—发送数据缓冲区指针的[31:16]位。

A[15:0]—发送数据缓冲区指针的[15:0]位。

12.3 链路层编程

链路层的主要功能是:在帧头和帧尾附加特殊的二进制编码来产生和识别帧界;将待发送的数据封装在数据帧里,并顺序发送。基于 MCF52233 的以太网链路层程序主要涉及初始化、以太帧发送和以太帧接收函数。

12.3.1 FEC初始化函数FEC_Init

1. FEC模块初始化步骤

要使用 FEC 模块，用户软件必须先对若干 FEC 控制寄存器进行初始化，通过 MII 管理接口对 EPHY 某些属性有选择地初始化（前节已有讲述），再对发送和接收缓冲区描述符进行初始化，最后通过置位以太网控制寄存器 ECR 中使能位来启动 FEC 工作。在 FEC 使能前，若需发送数据，则还要将相应数据载入发送缓冲区。FEC 模块使能后，它的 RISC 控制器将开始执行微代码以完成如下工作：初始化退避随机数种子；激活接收器和发送器；清空接收和发送 FIFO；初始化接收和发送环形指针；初始化 FIFO 计数寄存器等。这些过程都无需用户软件的干预，由硬件自动完成。

在本书的实际编程中，用户软件对 FEC 模块初始化步骤大体如下：

（1）复位 FEC 模块。需要设置以太网控制寄存器 ECR 的 RESET 位为 1，置位后约 8 个系统总线周期 FEC 逻辑的复位序列执行完毕，此时 RESET 位由 FEC 逻辑自动清零。

（2）通过中断控制模块，设置 FEC 的 13 个中断级别。

（3）清空所有 MIB 计数器；解除所有 FEC 中断的屏蔽，清空所有中断事件标志。

（4）通过寄存器 PALR 和 PAUR，设置本地物理地址。低位寄存器 PALR 的地址用于设置 48 位 MAC 地址的低 32 位；而高位寄存器 PAUR 的 D31~D16 位用于设置 48 位 MAC 地址的高 16 位，D15~D0 为类型位，即 PAUSE 帧的类型字段，需要设为常量 0x8808。

（5）初始化哈希表单播寄存器和组播寄存器。

（6）配置以太网接收缓冲区大小寄存器（EMRBR），设置最大接收帧长度。该寄存器只有 D10 到 D4 位可被修改，低 4 位固定为 0，故该配置值应在 16~2047 字节范围内，且必须是 16 字节的整数倍。

（7）分别使用 RCR 和 TCR 寄存器来配置接收器和发送器的工作模式与属性，包括接收器接收的最大帧长、MII 模式、在发送时是否允许接收，以及是否使能 FEC 发送器的全双工模式。

（8）设置 MDC 时钟频率。MDC 频率为系统时钟频率的 1/（MII_SPEED*2），其中 MII_SPEED 速度选择因子由 MII 速度控制寄存器 MSCR 的 D6-D1 位指定。例如系统总线频率为 60MHz 时，要得到 2.5MHz 的 MDC 时钟频率，则 MII_SPEED=12 或 0x0C。

（9）建立缓冲区描述符环，初始化缓冲区描述符各字段，并分别将起始地址存放在接收描述符环开始寄存器 ERDSR 和发送描述符环开始寄存器 ETDSR 中。

（10）通过 MIB 控制寄存器 MIBC 使能 MIB 模块；使用以太网控制寄存器 ECR 的 ETHER_EN 位来使能 FEC 模块；并通过接收和发送描述符激活寄存器（RDAR/TDAR）来激活接收和发送描述符。

2. FEC初始化函数FEC_Init

```

//-----*
//函数名: FEC_Init *
//功 能: 初始化FEC模块 *
//参 数: uint8 mac_addr_fec[6]: 本机MAC地址 *
//返 回: 无 *
//内部调用: 1)clearMIB MIB硬件寄存器复位清零 *
//          2)BDinit缓冲区描述符初始化 *
//说 明: *
//-----*
void FEC_Init(uint8 mac_addr_fec[6])
{
    uint8 i;
    //1. 复位FEC(ECR.RESET=1)
    MCF_FEC_ECR = MCF_FEC_ECR_RESET;
    while (MCF_FEC_ECR & MCF_FEC_ECR_RESET) //复位完成后由硬件自动清0
    { ; } //大约8个时钟周期
    //2. 将13个FEC中断等级设置为6(23-35号中断)
    // 解除对发送帧、发送缓冲区、接收帧、MII中断的屏蔽, 使能FEC接收帧中断
    for (i = 23; i < 36; i++)
    {
        MCF_INTC0_ICR(i) = MCF_INTC_ICR_IL(6); //设置中断等级
    }
    //3. 清空所有的MIB计数器//屏蔽所有的FEC中断//清空所有的中断事件
    clearMIB();
    MCF_FEC_EIMR = MCF_FEC_EIMR_ALL_MASKS;
    MCF_FEC_EIR = MCF_FEC_EIR_ALL_EVENTS; //写1清空
    //4. 配置本地MAC地址
    MCF_FEC_PALR = (uint32)((0
        | (mac_addr_fec[0] <<24) | (mac_addr_fec[1] <<16)
        | (mac_addr_fec[2] <<8) | (mac_addr_fec[3] <<0)));
    MCF_FEC_PAUR = (uint32)((0
        | (mac_addr_fec[4] <<24) | (mac_addr_fec[5] <<16)
        | MCF_FEC_PAUR_TYPE(0x8808)));
    //5. 初始化哈希表寄存器, 为了允许接收组播帧, 将哈希表组寄存器全置1
    MCF_FEC_IALR = 0;
    MCF_FEC_IAUR = 0;
    MCF_FEC_GALR = 0xffffffff;
    MCF_FEC_GAUR = 0xffffffff;
    //6. 设置接收缓冲区大小: 接收缓冲区 = 最大帧长
    MCF_FEC_EMRBR = MAX_ETH_PKT; // >= 最大帧长度
    //7. 配置FEC收发器模式
    // 接收器模式: 最大帧长、MII模式、半双工
    MCF_FEC_RCR = (MCF_FEC_RCR_MAX_FL(MAX_ETH_PKT) |
        MCF_FEC_RCR_MII_MODE |
        MCF_FEC_RCR_DRT); //半双工
    // 配置FEC发送器模式: 全双工
    MCF_FEC_TCR = ( 0 | MCF_FEC_TCR_FDEN); // 全双工

```

```

//8. 配置MII接口速度, 必须 <= 2.5MHz.
//      60 / (0x0D*2) = 2.3 MHz <2.5MHz
MCF_FEC_MSCR = MCF_FEC_MSCR_MII_SPEED(0x0D);
//9. 建立环缓冲区
BDinit();
MCF_FEC_ERDSR = (uint32) & RxBDS[0];
MCF_FEC_ETDSR = (uint32) & TxBDS[0];
//10. 使能MIB模块, 使能FEC, 激活接收和发送描述符
MCF_FEC_MIBC &= ~MCF_FEC_MIBC_MIB_DISABLE;
MCF_FEC_ECR = MCF_FEC_ECR_ETHER_EN;
MCF_FEC_RDAR = MCF_FEC_RDAR_R_DES_ACTIVE;
MCF_FEC_TDAR = MCF_FEC_TDAR_X_DES_ACTIVE;
}

```

3. 缓冲区描述符初始化函数BDinit

实现以太网数据的收发, 缓冲区描述符的使用非常重要, 它是 FEC 实现数据收发的直接依托。

特别注意, 单个发送帧可以占用多个发送缓冲区及其描述符。例如, 某个待发送帧长度为 1500 字节, 而缓冲区大小为 400 字节, 这样, 就需要 4 个缓冲区来存放该帧, 这四个缓冲区分别由它们的描述符定义, 用户软件必须设置前 3 个描述符的 L 位为 0, 设置第 4 个描述符的 L 位为 1, 且前 3 个描述符的数据长度域为 400, 第 4 个描述符的数据长度域为 300。这样, FEC 就能够访问该发送帧的所有数据。其实, 实际使用并不需要这么复杂, 若把缓冲区大小开辟得足够大, 如 1520 字节 (常规帧大小为 1518 字节, 取 1520 是因为它是 16 的整数倍, MCF52233 的 FEC 推荐缓冲区使用 16 字节对齐, 主要是为了能把多个缓冲区开辟在连续的存储空间而不产生碎片), 则通常单帧长度就不可能超过缓冲区的大小, 因此, 存储 1 帧只需 1 个发送缓冲区就够了。在本书的工程实例中, 缓冲区大小就开辟为 1520 字节。

在本书编程实例中, 缓冲区描述符初始化过程如下。

```

缓冲区描述符结构定义
typedef struct
{
    vuint16 status; //控制和状态
    vuint16 length; //缓冲区长度
    vuint8 * dataAd; //缓冲区地址
} BD;
//FEC收发缓冲区描述符的个数
#define NUM_RXBDS 2
#define NUM_TXBDS 2
//FEC收发缓冲区大小(最好是16的整数倍)
#define RX_BUFF_SIZE 2048
#define TX_BUFF_SIZE 1520
//收发缓冲区描述符需要按16字节对齐
vuint8 rxbd[sizeof(BD) * NUM_RXBDS + 16];

```

```

uint8 txbd[sizeof(BD) * NUM_TXBDS + 16];
//收发缓冲区需要按16字节对齐
uint8 rxbuffer[RX_BUFF_SIZE * NUM_RXBDS + 16];
uint8 txbuffer[TX_BUFF_SIZE * NUM_TXBDS + 16];
BD * RxBDS;      // 接收缓冲区描述符
BD * TxBDS;      // 发送缓冲区描述符
uint8 * RxBuff;  // 接收缓冲区首地址
uint8 * TxBuff;  // 发送缓冲区首地址
//-----*
//函数名: BDinit                                *
//功 能: 缓冲区描述符初始化                      *
//参 数: 无                                      *
//返 回: 无                                      *
//-----*
void BDinit()
{
    uint8 i;
    //1. 缓冲区描述符按16字节对齐
    RxBDS = (BD *) ((uint32)(rxbd + 16) & 0xFFFFFFF0);
    TxBDS = (BD *) ((uint32)(txbd + 16) & 0xFFFFFFF0);
    //2. 缓冲区按16字节对齐
    RxBuff = (uint8 *) ((uint32)(rxbuffer + 16) & 0xFFFFFFF0);
    TxBuff = (uint8 *) ((uint32)(txbuffer + 16) & 0xFFFFFFF0);
    //3. 初始化接收描述符环
    for(i=0; i<NUM_RXBDS; i++)
    {
        RxBDS[i].status |= MCF_FEC_RxBD_E;
        RxBDS[i].length = 0;
        RxBDS[i].dataAd = & RxBuff[i * RX_BUFF_SIZE];
    }
    RxBDS[NUM_RXBDS-1].status |= MCF_FEC_RxBD_W;
    //4. 初始化发送描述符环
    for(i=0; i<NUM_TXBDS; i++)
    {
        TxBDS[i].status |= MCF_FEC_TxBD_L | MCF_FEC_TxBD_TC;
        TxBDS[i].length = 0;
        TxBDS[i].dataAd = & TxBuff[i * TX_BUFF_SIZE];
    }
    TxBDS[NUM_TXBDS-1].status |= MCF_FEC_TxBD_W;
}

```

12.3.2 以太帧发送

1. 以太帧的发送过程描述

发送一帧以太帧数据时，首先要获得一个空闲的发送缓冲区描述符，若没有则退出，若有则将待发送数据的首地址和数据长度赋给该缓冲区描述符相关字段，同时配置该描述符的 R、L 和 TC 位。然后使能发送缓冲区描述符激活寄存器来启动发送，DMA 控制器

将把发送缓冲区描述符所指向的缓冲区中的数据传送到 Tx FIFO 中。

当 Tx FIFO 被填充到水位（由发送水位寄存器 TFWR 定义）位置时，启动硬件发送器发送数据。在进行发送操作前，FEC 进行载波侦听，来判断网路是否可用；若检测到网路闲置 60 位时间，则再等待 36 位时间后开始数据帧的发送。一旦开始发送，FEC 首先将传输前导同步码（PA）序列和帧起始分隔符（SFD），或叫前导位域和帧起始位域，然后发送 FIFO 中的数据帧信息。数据发送过程中，FEC 会实时检测网络上是否产生冲突，若产生冲突则延时一段随机的时间再次重发，重发次数最多 16 次。在 FIFO 中所有帧数据都被发送后，如果允许（发送缓冲区描述符中的 TC 位置位），则帧校验序列 FCS（32 位的循环冗余校验 CRC）将被发送。相邻帧发送的时间间隔不小于 96 位时间。

发送缓冲区的一帧数据被全部发送完毕时，将产生发送帧中断（若初始化已使能）。发送缓冲区描述符中准备就绪位（R 位）就会被 FEC 清零，用以告知用户软件发送缓冲区已空，此时可以向其中写入数据；若用户软件写完数据，则需将该位置 1。

2. 以太帧发送函数 FEC_FrameSend

使用 MCF52233 发送以太帧，必须将帧的内容写入到 FEC 模块的发送缓冲区描述符所指定的内存缓冲区里，然后再通过发送缓冲区描述符激活寄存器 TDAR 将其发送出去，接下来的工作由底层硬件完成。放入内存缓冲区的内容包括目的 MAC 地址、源 MAC 地址、长度/类型字段、数据字段等，而以太帧的其他组成部分，如前导位、帧起始位和校验码，则由硬件自动添加。

在本书工程实例中，需要将以太帧中的数据字段内容预先存放在一维数组中，并通过传址和传值的方式将目的 MAC 地址、源 MAC 地址、长度/类型字段等相关信息传递给以太帧发送函数 FEC_FrameSend。具体来说，编程实现发送以太帧的一般步骤是：

（1）查找可用发送缓冲区描述符 BD，确定是否有空闲缓冲区可用，无则返回 0，有则获取空闲缓冲区的首地址。

（2）组建以太帧，并将它放入发送缓冲区。具体步骤：写入目的 MAC 地址，写入源 MAC 地址，写入长度类型字段，写入数据。

（3）配置发送缓冲区描述符长度字段。

（4）配置发送缓冲区描述符状态控制字段。

（5）使能发送缓冲区描述符激活寄存器。

以太帧发送函数 FEC_FrameSend 实现代码如下。

```
//-----*
//函数名: FEC_FrameSend                                *
//功 能: 发送单个以太帧                                *
//参 数: uint8 ch[]: 指向的内存中待发送的数据存放处    *
//      uint8 destAdd[6]: 6字节目的MAC地址                *
//      uint8 srcAdd[6]: 6字节源MAC地址                  *
//      uint16 lenType: 帧的长度/类型字段                 *
//      uint16 len: ch[]中的数据长度                     *
//返 回: 1: 发送成功, 0: 无可发送缓冲区                 *
```

```

//内部调用：内存拷贝函数memcpy *
//说 明： ch[] 传入的发送数据是指以太网格式中的数据字段 *
//-----*
int8 FEC_FrameSend(uint8 ch[], uint8 destAdd[6], uint8 srcAdd[6], \
                    uint16 lenType, uint16 len)
{
    uint8 i;
    vuint8 * TxdataAd;
    //1. 查找可用发送缓冲区
    for(i=0; i<NUM_RXBDS; i++) //顺序访问缓冲区描述符
    { //获得第1个空闲缓冲区描述符下标
        if((TxBDs[i].status & MCF_FEC_TxBD_R)==0) //发送准备就绪位=0
            break;
    }
    if(i==NUM_RXBDS) //无可发送缓冲区, 则返回0
        return 0;
    TxdataAd = TxBDs[i].dataAd; //获取空闲缓冲区的首地址
    //2. 组建以太网帧
    memcpy((void *)TxdataAd, (void *)destAdd, 6); //写入目的MAC地址
    TxdataAd += 6;
    memcpy((void *)TxdataAd, (void *)srcAdd, 6); //写入源MAC地址
    TxdataAd += 6;
    *TxdataAd = (lenType>>8); //写入长度类型字段
    TxdataAd ++;
    *TxdataAd = lenType;
    TxdataAd ++;
    memcpy((void *)TxdataAd, (void *)ch, len); //写入数据
    //3. 配置发送缓冲区描述符长度字段
    TxBDs[i].length = len + 14; //写入缓冲区长度
    //4. 配置发送缓冲区描述符状态控制字段
    TxBDs[i].status |= (0 | MCF_FEC_TxBD_L //设置为帧的最后缓冲区
                       | MCF_FEC_TxBD_TC //帧结尾由硬件发送CRC校验
                       | MCF_FEC_TxBD_R); //数据准备好(软件置1, FEC清零)
    //5. 使能发送缓冲区描述符激活寄存器
    MCF_FEC_TDAR = MCF_FEC_TDAR_X_DES_ACTIVE;
    return 1;
}

```

注意：待发送的数据使用参数 `ch[]` 传入。函数的主要功能是将待发送的以太网帧放入发送缓冲区，并启动发送，剩下的事情由硬件完成。发送完成后，相应发送准备就绪位被 FEC 清零（程序由此判断发送是否完成）；若发送帧中断使能，则产生相应中断。

12.3.3 以太网接收

1. 以太网接收过程

对于一帧以太网数据的接收来说，需要对相关寄存器进行设置并使能接收缓冲区描述符激活寄存器（在初始化已设置），当有以太网数据到来时，若 FEC 的地址识别结果是接

收该帧，则接收硬件会自动先把数据放入 Rx FIFO 中，之后 FEC 的 DMA 将 Rx FIFO 中数据传送到接收缓冲区描述符所定义的系统缓冲区，缓冲区通常被安置在 RAM 区。当整个帧都传送完毕后，缓冲区描述符相关状态和控制位将被 FEC 设置，同时产生接收帧中断（若初始化中已使能）。

当产生接收帧中断后，用户程序需要检查接收缓冲区描述符相关状态位，以确定该接收帧是否合法，若不合法则丢弃；若合法则将数据从接收缓冲区描述符所定义的系统缓冲区拷贝出来（如拷贝到用户缓冲区队列中）。然后重新配置缓冲区描述符相关状态控制位以及接收缓冲区描述符激活寄存器，以便接收下一帧数据。

帧最大长度和接收缓冲区大小都可以编程设定。接收缓冲区大小寄存器（EMRBR）用于定义缓冲区的大小，接收控制寄存器（RCR）的 MAX_FL 域则定义帧最大长度。缓冲区大小应在 16~2047 字节范围内，且是 16 字节的整数倍。缓冲区大小若大于或等于帧最大长度，则接收到的帧都可以放进该缓冲区中。但为了使得存储完整接收帧所需的缓冲区数量最小，推荐缓冲区大小应与实际使用的一致。帧长从接收帧目的地址开始，直到帧结束，包含 CRC 校验码字段。在本书工程实例中，初始化函数已经对帧最大长度和接收缓冲区大小进行了设定。

注意，当帧长度大于 RCR 的 MAX_FL 位域定义的值，但小于最大的 2047 字节时，帧仍可以被接收，且不会被截断。此时若 BABR 接收中断已被使能，则将产生该中断，同时帧最后缓冲区描述符状态字的 LG 位将被置“1”。若帧长度超过 2047 字节，则超出部分将被截去，而缓冲区描述符状态字的 TR 位被置“1”。所谓 BABR，就是指收发链路上的以太网帧实测长度大于定义长度的情况，该情况大多是由于网络故障或不规范的连续传送超过帧长限制造成，故而又叫“串扰”。

2. 以太帧接收函数FEC_ReceiveFrame

接收以太帧有两种方法：查询法和中断法。由于中断法有更好的执行效率，本书实例使用中断法接收以太帧。不管是查询法，还是中断法，接收完整一帧的一般步骤如下：

（1）查找各接收缓冲区是否有刚收到的数据。若所有接收缓冲区中都已无新数据，则退出；若有刚收到的数据，则获取其缓冲区首地址。

（2）从接收缓冲区描述符中获取长度字段。

（3）根据缓冲区首地址，从接收缓冲区中获取接收到的数据。

（4）配置接收缓冲区描述符状态控制字段，为该缓冲区的后续接收创造条件。如置位缓冲区描述符空标志位 E（软件置 1，FEC 清零），以声明该缓冲区已准备好接收数据。

（5）使能接收缓冲区描述符激活寄存器，等待接收下一帧。

具体函数实现如下：

```
//-----*
//函数名: FEC_ReceiveFrame                      *
//功 能: 接收单个以太网帧                      *
//参 数: uint8 ch[]: 指向接收数据所放内存的首地址 *
//      uint16 * len: 接收到的帧长度              *
```

```

//返 回：1：接收成功 0：无新数据                                     *
//说 明：建立连接或在建立连接后改变中断                             *
//-----*
int8 FEC_ReceiveFrame(uint8 ch[], uint16 * len)
{
    uint8 i;
    vuint8 * RxdataAd;
    //1. 查找各接收缓冲区是否有刚收到的数据
    for(i=0; i<NUM_RXBDS; i++) //顺序访问缓冲区描述符
    {
        //获得第1个空闲缓冲区描述符下标
        if((RxBDS[i].status & MCF_FEC_RxBD_E)==0)
            break;
    }
    if(i==NUM_RXBDS) //所有接收缓冲区中都已无新数据, 则返回0
        return 0;
    RxdataAd = RxBDS[i].dataAd; //获取有数据的缓冲区的首地址
    //2. 获取接收数据的长度
    *len = RxBDS[i].length; //从接收缓冲区描述符中获取长度字段
    //3. 获取接收数据
    memcpy((void *) ch, (void *) RxdataAd, *len); //写入数据
    //4. 配置接收缓冲区描述符状态控制字段
    RxBDS[i].status |= MCF_FEC_RxBD_E; //缓冲区准备好接收数据(软件置1, FEC清零)
    //5. 使能接收缓冲区描述符激活寄存器, 等待接收下一帧
    MCF_FEC_RDAR = MCF_FEC_RDAR_R_DES_ACTIVE;
    return 1;
}

```

事实上, `FEC_ReceiveFrame` 函数的功能就是将接收缓冲区的以太帧拷贝出来, 并为下一次接收准备条件, 而被拷贝的以太帧是由 `FEC` 硬件自动接收而来。

`FEC` 模块收到一个以太帧后, 会产生一个接收帧中断 (需在初始化时使能该中断)。在接收帧中断处理中进行以太帧的接收, 具体方法见测试实例。

12.3.4 第二个测试实例：以太帧的发送和接收

在第一个测试实例的基础上, 第二个测试实例新增了包含 `FEC` 初始化、以太帧发送和接收功能函数的 `FEC` 构件 (包括 `FEC.c` 和 `FEC.h`)。本测试实例采用了自回环模式来测试以太帧的收发过程。要设置成回环模式, 需要在初始化函数中稍做修改, 将初始化函数的以下语句中的 “`MCF_FEC_RCR_DRT`” 改为 “`MCF_FEC_RCR_LOOP`” 即可。前者所对应的掩码是半双工模式使能位, 而后者则是回环模式使能位, 两者不能同时开启。

```

//7. 配置FEC收发器模式
// 接收器模式：最大帧长、MII模式、半双工
MCF_FEC_RCR = (MCF_FEC_RCR_MAX_FL(MAX_ETH_PKT) |
               MCF_FEC_RCR_MII_MODE |
               MCF_FEC_RCR_DRT); //半双工

```

在 `FEC` 初始化 (已经为自回环模式) 之后, 主程序定时向外发送以太帧, 无需外接

网线，帧中断程序就会处理回环接收来的帧，并通过串口把它发送到 PC 机上。

1. 以太帧收发测试主函数

```

//-----*
// 工程名: FEC *
// 硬件连接: 无 *
// 功能描述: 采用自回环模式测试以太帧收发子程序 *
//          主程序定时向外发送以太帧, 帧中断程序则处理回环接收的帧, 并通过 *
//          串口把它发送到PC机 *
//-----苏州大学Freescale示范实验室2009-----*
#include "includes.h" //头文件
//全局变量定义
uint8 local_addr[6] = {0x00, 0xBA, 0xDB, 0xAD, 0x01, 0x02}; //本地MAC地址
uint8 dest_addr[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}; //目的MAC地址
uint8 ch[80]; // 存放待发数据
void main(void)
{
    uint8 i, j;
    //1. 关总中断
    DisableInterrupts();
    //2. 模块初始化
    //小灯初始化
    Light_Init(Light_Link_PORT, Light_Link, Light_OFF);
    //UART初始化, 使用UART0模块, 使用系统时钟作为UART的时钟源
    //波特率为9600, 初始化时禁止接收中断
    UartInit(0, SYSTEM_CLOCK, 9600, 0);
    //EPHY初始化, 初始配置为100Mbps, 半双工, 禁止自动协商
    EPHY_Init(SYSTEM_CLOCK, 0, 0, 0);
    //FEC初始化, 初始配置本地MAC地址为local_addr
    FEC_Init(local_addr);
    //3. 开总中断
    EnableInterrupts();
    j = 0;
    //4. 主循环
    while (1)
    {
        j++;
        //初始化传输数据(每次数据都不一样)
        for(i=0; i<80; i++)
            ch[i] = j;
        //待发数据首指针ch, 目的MAC地址destAdd, 源MAC地址srcAdd
        //帧的长度/类型字段0x0806, ch[]的数据长度80
        FEC_FrameSend(ch, dest_addr, local_addr, 0x0806, 80);
        //运行指示灯控制
        Light_Control(Light_Link_PORT, Light_Link, Light_ON);
        Delay(30);
        Light_Control(Light_Link_PORT, Light_Link, Light_OFF);
        Delay(30);
    }
}

```

```

    }
    return;
}

```

2. 以太网收发测试中断处理函数

```

uint8 ReceiveFrameFlag;    //接收到帧标志
uint8 ReceiveFrameData[1520]; //接收帧数据存放处
uint16 ReceiveFrameLenth; //接收帧的数据长度
//以太网接收中断处理函数
__interrupt__
void ReFrame_handler(void)
{
    uint8 status;
    DisableInterrupts(); //关总中断
    MCF_FEC_EIR = MCF_FEC_EIMR_RXF; //写1清空帧接收中断标志
    //将接收数据放入redata数组,数据长度为ReceiveFrameLenth
    status = FEC_ReceiveFrame(ReceiveFrameData, &ReceiveFrameLenth);
    UartSendN(0, 2, (uint8*)(&ReceiveFrameLenth)); //串口发送接收到的以太网帧的长度
    UartSendN(0, reLen, (uint8*) ReceiveFrameData); //串口发送接收到的以太网帧内容
    EnableInterrupts(); //开总中断
}

```

3. 以太网收发测试方法

接上串行通信线,运行本书给出的该测试实例后,打开 PC 机的串口调试器,可发现 MCU 不断发来的以太网帧数据,如图 12-8 所示,其中高亮显示的为完整一帧的内容。

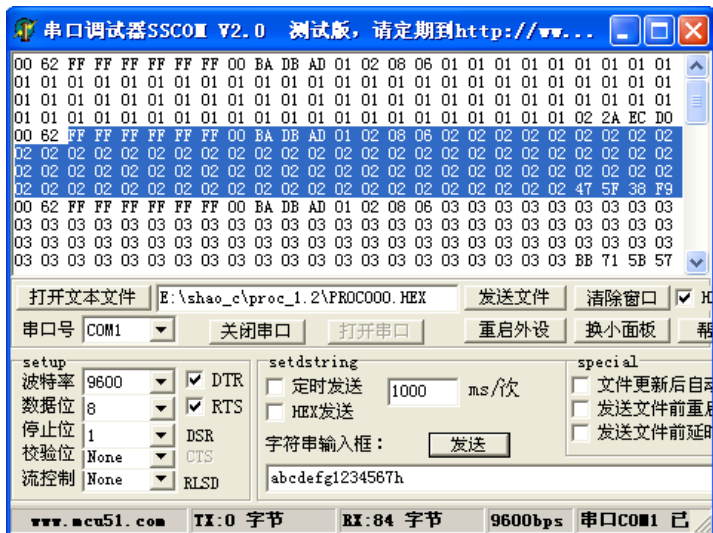


图12-8 以太网收发测试

12.3.5 第三个测试实例：链路层 ARP 协议编程

在以太网环境中，为了正确的向目的主机传送报文，必须把目的主机的 32 位 IP 地址转换成为 48 位以太网 MAC 地址，这样的话，源主机就需要向与之相连的主机广播 ARP 请求，以获得目的主机的 MAC 地址。ARP (Address Resolution Protocol) 是地址解析协议的简称，它属于链路层协议，与之配套的还有反地址解析协议 RARP。嵌入式以太网设备一般在应用中作为终端使用，所以本实例仅实现对 ARP 请求的应答。

1. ARP 帧格式

实现 ARP 协议的目的是将 IP 地址转换成物理 MAC 地址^①。图 12-9 给出了 ARP 协议的帧格式。

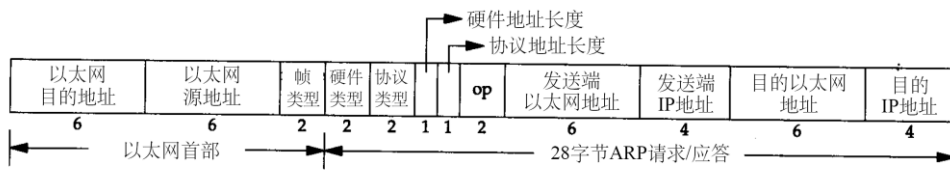


图12-9 ARP协议的帧格式

以太网中，在向目标主机发送一个 ARP 请求时，如果目标主机处于活动状态则会返回其 MAC 地址，这样，当只知道本网段中目标主机 IP 地址，就可由此协议获得目标主机的 MAC 地址。

2. ARP 请求及应答过程

ARP 协议是为了使通信双方获取对方 MAC 地址的通信协议，本书实例实现了 ARP 请求报文的接收和应答的处理功能。

当设备需要知道远程机器的 MAC 地址时，则生成一个 ARP 请求报文，再按以太帧格式在报文前添加目的 MAC 地址、源 MAC 地址和类型（0x0806）封装成以太帧，此时 ARP 请求报文以及以太帧中目的 MAC 地址应全部填 0xFF，最后由 MAC 层的以太帧收发驱动程序将该帧发送到局域网。

当远端设备接收到局域网中某台设备请求自己的 MAC 地址时，则向对方发送 ARP 响应报文，告诉对方自己的 MAC 地址。此时，只需要将以太帧头和 ARP 请求报文的源 IP 和目的 IP 地址、源 MAC 和目的 MAC 地址互换，ARP 报文中的操作字段改成 2，并将本机的 MAC 地址替换该报文的源 MAC 地址字段，最后将该以太帧发往局域网。

ARP 请求包格式具体如下：

以太帧目的 MAC 地址：0xFFFFFFFFFFFF（表示广播）；源地址：本机 MAC 地址；帧类型：0x0806 表示 ARP 帧；硬件类型：0x01（表示以太网）；协议类型：0x0800 表示

^①其详细过程可参考(美)Gary R. Wright, W.Richard Stevens 著，陆雪莹、蒋慧等译，TCP/IP 详解卷 1，机械工业出版社，2007 年。

IPv4 协议；硬件地址长度：0x06；协议地址长度：0x04；OP 操作选项：0x01 表示 ARP 请求；发送端以太网地址：本地 MAC 地址；发送端 IP 地址：本地主机 IP 地址；目的以太网地址：0x000000000000；目的 IP 地址：目标主机 IP 地址。

所谓广播，表示本网段中所有主机都能收到该数据包。在收到该包后，各主机判断目的 IP 地址是否与本主机 IP 地址匹配，若匹配则发回一个 ARP 应答包，否则丢弃该帧不作处理。

ARP 应答包的格式具体如下：

以太网目的地址：发出探测的主机的 MAC 地址；以太网源地址：本地 MAC 地址（这里本地指被探测主机）；帧类型：0x0806 表示 ARP 帧；硬件类型：0x01 表示以太网；协议类型：0x0800 表示 IPv4 协议；硬件地址长度：0x06；协议地址长度：0x04；OP 操作选项：0x02 表示 ARP 应答；发送端以太网地址：本地 MAC 地址（这里本地指被探测主机）；发送端 IP 地址：本机 IP 地址（这里本地指被探测主机）；目的以太网地址：发出探测的主机的 MAC 地址；目的 IP 地址：发出探测主机的 IP 地址。

3. ARP测试方法

在第二个测试实例的基础上，第三个测试实例新增了 ARP 构件（ARP.h 和 ARP.c），其实现的功能是：对于接收到的以太帧，若判断为 ARP 类型，则调用函数 ARP_Handle 对 ARP 包进行分析，如果是 ARP 请求包，则发送 ARP 响应。具体程序见本书网上光盘。

ARP 构件对外接口函数：

```
int ARP_Handle(PACKET p);           //处理 ARP 包
int ARP_Send(PACKET p);             //发送 ARP 包
```

该测试实例运行后，启动 PC 端的串口调试器，单击“开始”菜单下的“运行”，输入：cmd，进入 DOS 命令状态，运行界面如图 12-10。

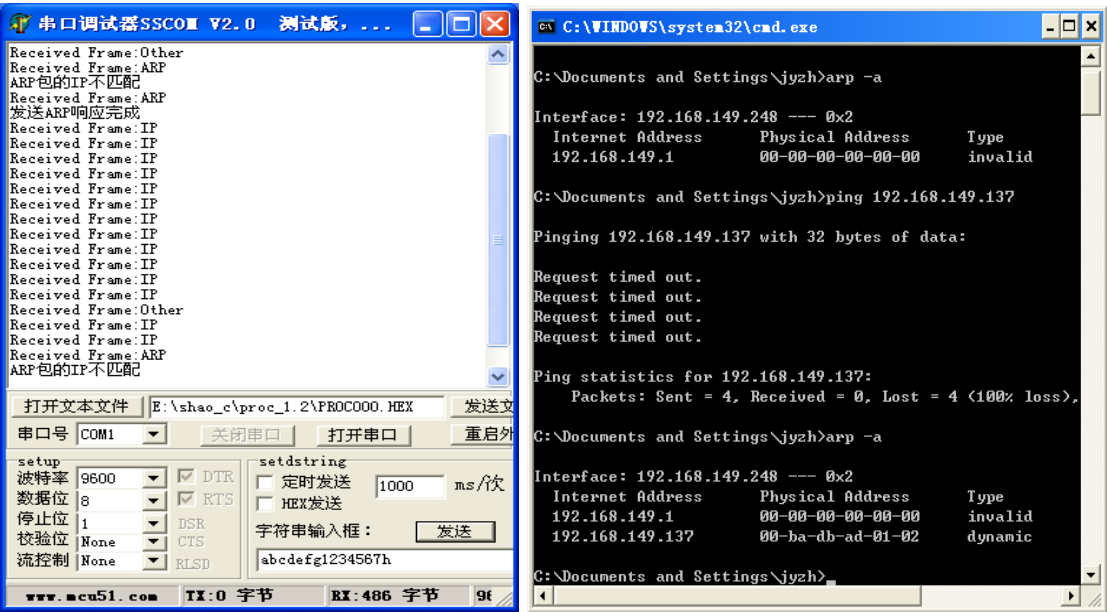


图12-10 测试ARP请求与应答

一开始，串口调试器窗口显示，MCF52233 设备不断收到各种以太网帧，对于 ARP 帧，则发现接收到的 ARP 包中目标 IP 地址与 MCU 的 IP 地址不匹配，这是因为 PC 机还没获取到设备的 MAC 地址。此时在 DOS 命令窗口输入命令：

```
arp -a
```

该命令的作用是查看当前 PC 机的 ARP 缓存表（IP 地址与 MAC 地址的对应表），结果显示只有网关（192.168.149.1）的选项，且 MAC 地址为 00-00-00-00-00-00。接着，在 DOS 命令窗口输入命令：

```
ping 192.168.149.137
```

该命令的作用是以广播方式在子网内发送 ARP 请求（详细内容见 12.4.2）。运行后，在串口调试器窗口中立即显示如下两行：

```
Received Frame:ARP
发送 ARP 响应完成
```

此时若在 DOS 命令窗口再次输入命令“arp -a”检查 ARP 缓存表时，发现已经多了一条“192.168.149.137 00-ba-db-ad-01-02”的记录，这就是 MCF52233 设备的 IP 地址和 MAC 地址，这样其它网络协议就可以通过查找 ARP 缓冲区来确定目的主机了。

值得注意的是：在此测试实例中，运行 DOS 命令“ping 192.168.149.137”后，发现连续显示了四个“Request timed out”，即“请求超时”，这是因为 MCF52233 程序仅发送了 ARP 响应，并未对随后接收到的 IP 包进行处理，这一问题将在下面的第四个测试实例中解决。

另外，ARP 协议只能在一个子网内的机器间运行，不同子网的机器之间是不能运行

ARP 请求或应答的，如果要进行网络层以上的通信，则要通过网关接口。

12.4 网络层的IP协议与ICMP协议编程

网络层是 TCP/IP 协议栈中最复杂的一层，主要解决主机到主机的通信问题。网间的数据报可以根据它携带的目的 IP 地址，通过路由器由一个网络传送到另一个网络。在嵌入式设计中，如果设备只是作为一个网络终端，而不需要具有网关或路由功能时，则只需要实现 IP 协议（数据传送）和 ICMP 协议（检测网络）。

12.4.1 IP协议简介

1. IP数据报格式

TCP/IP 协议定义了一个在因特网上传输的包，称为 IP 数据报（IP Datagram）。这是一个与硬件无关的虚拟包，由首部和数据两部分组成，其格式如图 12-11 所示。首部的前

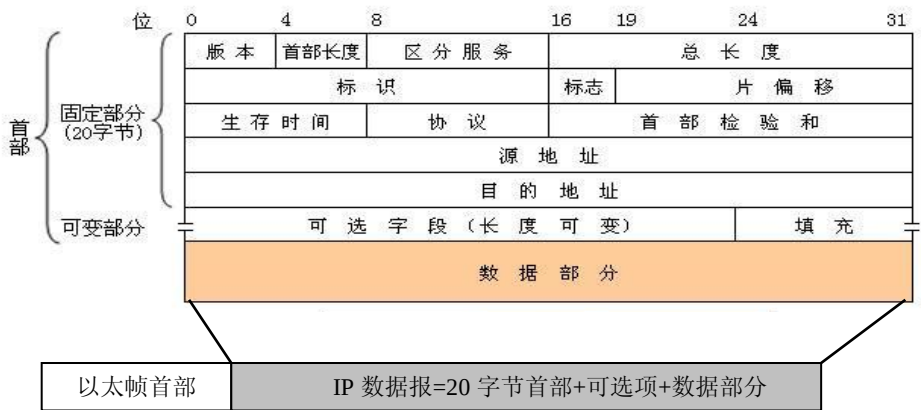


图12-11 IP数据报格式

一部分是固定长度，共 20 字节，是所有 IP 数据报必须具有的。在首部的固定部分的后面是一些可选字段，其长度是可变的。首部中的源地址和目的地址都是 IP 协议地址。本书的程序中不对可选字段做处理，并且不支持 IP 分片。

IP 数据报首部的固定部分中的各字段含义如下：

版本（占 4 位）：指 IP 协议的版本。通信双方使用的 IP 协议版本必须一致。目前广泛使用的 IP 协议版本号为 4（即 IPv4）。

首部长度（占 4 位）：可表示的最大十进制数值是 15。注意：该字段所表示数的单位是 32 位字（即 4 字节）。例如，当 IP 的首部长度为 1111（即十进制的 15）时，首部长度就达到 60 字节。当 IP 分组的首部长度不是 4 字节的整数倍时，必须利用最后的填充字段加以填充。因此数据部分永远在 4 字节的整数倍开始，这样在实现 IP 协议时较为方便。

区分服务（占 8 位）：用来获得更好的服务。该字段在旧标准中叫做服务类型，但实

际上一直没有被使用过。

总长度（占 16 位）：指首部和数据长度之和，单位为字节。总长度字段为 16 位，因此数据报的最大长度为 $2^{16}-1=65535$ 字节。

标识（占 16 位）：IP 软件在存储器中维持一个计数器，每产生一个数据报，计数器就加 1，并将此值赋给标识字段。但这个“标识”并不是序号，因为 IP 是无连接服务，数据报不存在按序接收的问题。当数据报由于长度超过网络的 MTU 而必须分片时，这个标识字段的值就被复制到所有的数据报的标识字段中。相同的标识字段的值使分片后的各数据报片最后能正确地重装成为原来的数据报。

标志（占 3 位）：IP 分片时才有用。本书不讨论。

片偏移（占 13 位）：IP 分片时才有用。本书不讨论。

生存时间（占 8 位）：生存时间字段常用的英文缩写是 TTL（Time To Live），表明是数据报在网络中的寿命。由发出数据报的源点设置该字段，其目的是防止无法交付的数据报无限制地在因特网中兜圈子，因而白白消耗网络资源。最初的设计是以秒为 TTL 的单位。每经过一个路由器时，就把 TTL 减去数据报在路由器消耗掉的一段时间。若数据报在路由器消耗的时间小于 1 秒，就把 TTL 值减 1。当 TTL 值为 0 时，就丢弃这个数据报。

协议（占 8 位）：该字段指出此数据报携带的数据是使用何种协议，以便使目的主机的 IP 层知道应将数据部分上交给哪个处理过程。0x01 表示 ICMP 协议，0x06 表示 TCP 协议，0x11 表示 UDP 协议。

首部检验和（占 16 位）：该字段只检验数据报的首部，不包括数据部分。这是因为数据报每经过一个路由器，路由器都要重新计算首部检验和。

源地址（占 32 位）：指源 IP 地址。

目的地址（占 32 位）：指目的 IP 地址。

2. IP 数据报接收

在接收 IP 数据报时，首先提取首部相关信息字段，在校验通过且相关设置符合要求（如 IP 不分片或数据报是发给本机的等）时，然后将其数据部分提供给更高层的协议处理，否则丢弃。由于嵌入式设备一般是作为终端设备，所以不需要实现转发功能。

3. IP 数据报发送

IP 协议在网络层提供调用接口给其他协议，因此发送程序首先根据调用方的协议类型作相应初始化；然后需要设定目的 MAC 地址，当目的主机与源主机在同一子网内，可以使用 ARP 协议直接匹配或发送 ARP 请求；但是如果目的主机在另一个子网中，则将目的 MAC 地址设置成默认网关的 MAC 地址，让路由器去转发；最后要做的就是根据图 12-11 封装 IP 数据报，再封装成以太网帧发送出去。

4. 组网注意事项

随着嵌入式以太网的广泛应用，实际应用中往往不止使用一台嵌入式以太网设备，有

时需要使用交换机、路由器等设备来组建一个局域网。在组网时需要注意以下事项：

(1) IP 地址和 MAC 地址的分配：在进行嵌入式以太网设备组网时（组建局域网）需要给每个设备分配局部唯一的 IP 地址和 MAC 地址，是由开发者自行分配的，只要确保在该局域网内是唯一的即可。

(2) 交换机的使用：当将嵌入式以太网设备连接到交换机时可能会出现无法通信，这是因为不同交换机支持的速率不一样，有的支持 100Mbps，有的支持 10Mbps。所以在编写以太网驱动时要考虑同时尝试这两种连接速率。关于该点读者可以参考网上光盘中第一个示例程序。

(3) 路由器的使用：将设备连接到路由器时，由于一般的路由器都启动 NAT^①（网络地址转化）功能，所以在实际应用中需要考虑到网内与网外之间通信地址转换问题。例如嵌入式设备本来的 IP 地址为 192.168.41.4，经过路由器 NAT 之后，出去的访问 IP 就会变成 192.168.1.176，访问的端口号也会改变，嵌入式设备就处于平时所说的网内，此时如果主机直接通过 UDP 协议访问嵌入式设备是访问不到的。

12.4.2 第四个测试实例：使用 ICMP 协议响应 Ping 请求

1. ICMP 报文格式

ICMP 协议（Internet Control Message Protocol，Internet 控制报文协议）是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息，如网络是否连通、主机是否可达、路由是否可用等。该协议属于网络层协议。ICMP 报文是在 IP 数据报内部被传输的，如图 12-12 所示。

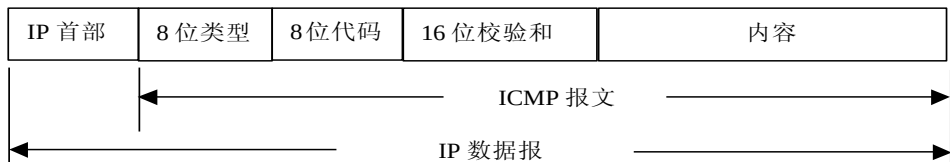


图12-12 封装在IP数据报内的ICMP报文

ICMP 报文的类型和代码字段各占 8 位，常用的类型字段值是“8”和“0”，“8”代表该 ICMP 报文为请求回应信息的报文，而“0”代表该 ICMP 报文为一个回应信息报文。

2. Ping 命令的使用

Ping 命令 是 Windows 操作系统自带的一个可执行命令，利用该命令可以检查网络是否能够连通。应用格式：**Ping IP 地址**。该命令还可加入各种参数使用。若直接键入“Ping”按回车即可看到详细说明。

Ping 命令通过发送一份“请求回应信息的 ICMP 报文”到目的主机，并等待接收一个“回应信息的 ICMP 报文”，由此来测试目的主机是否可达。因此，该命令的执行过程

^① NAT（网络地址转化）功能：指将内部网络的私有 IP 地址转换为公用 IP 地址

实际上就是 ICMP 协议工作的过程。在没有访问控制或防火墙的情况下, 执行 Ping 命令不能检测到某台主机, 那么就不能在高层的协议 (如 UDP、TCP 等) 上访问该主机。

当嵌入式设备和 PC 主机通过交叉线直连后, 只要 PC 的网卡显示已连通, 就可以通过 Ping 程序来测试是否可以与该设备通信。因为在网络通信中, 嵌入式设备大多是被动接收通信请求, 所以为了简化协议, 下面的测试实例只实现 ICMP 回显应答服务。

3. ICMP 测试方法

在第三个的基础上, 第四个测试实例新增了 IP 构件(IP.h、IP.c)和 ICMP 构件(ICMP.h、ICMP.c), 其实现的功能是: 对于接收到的以太帧, 判断是 ARP 包, 还是 IP 包。若为 ARP 包并且为 ARP 请求包, 则立即发送 ARP 响应; 若为 IP 包, 则分析其协议类型。若为请求回应信息的 ICMP 报文 (ICMP echo), 则立即发送回应报文 (ICMP echo Reply)。

IP 构件对外接口函数:

```
int IP_Handle(PACKET p);           //处理接收到的 IP 包
int IP_Send(PACKET p);             //发送 IP 包
```

ICMP 构件对外接口函数:

```
int ICMP_Handle(PACKET p);         //处理接收到的 ICMP 包
```

该测试实例运行后, 启动 PC 端的串口调试器, 单击“开始”菜单下的“运行”, 输入: cmd, 进入 DOS 命令状态。

首先在 DOS 命令窗口输入命令: arp -a, 查看当前 PC 机的 ARP 缓存表; 接着输入命令: ping 192.168.159.201, 可发现收到四个 ICMP 回应信息报文。

为了更详细地查看 ARP 包和 ICMP 包的内容, 可运行捕包工具进行查看。本书以常用的捕包工具软件 CommView (见随书网上光盘本章的辅助工具目录) 为例。该测试实例运行后, 启动 CommView, 则窗口中也显示出了相关状态信息, 如图 12-13。

Protocol	MAC Addresses	IP Addresses
1 ARP REQ	IntelCorpo:DA:76:1A => Broadcast	192.168.159.248 -> 192.168.159.201
2 ARP RESP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.201 -> 192.168.159.248
3 IP/ICMP	IntelCorpo:DA:76:1A => AA:BB:CC:00:00:01	192.168.159.248 => 192.168.159.201
4 IP/ICMP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.248 <= 192.168.159.201
5 IP/ICMP	IntelCorpo:DA:76:1A => AA:BB:CC:00:00:01	192.168.159.248 => 192.168.159.201
6 IP/ICMP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.248 <= 192.168.159.201
7 IP/ICMP	IntelCorpo:DA:76:1A => AA:BB:CC:00:00:01	192.168.159.248 => 192.168.159.201
8 IP/ICMP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.248 <= 192.168.159.201
9 IP/ICMP	IntelCorpo:DA:76:1A => AA:BB:CC:00:00:01	192.168.159.248 => 192.168.159.201
10 IP/ICMP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.248 <= 192.168.159.201

0x0000	AA BB CC 00 00 01 00 13-20 DA 76 1A 08 00 45 00	?... 趣...E.
0x0010	00 3C 43 9F 00 00 80 01-36 0F C0 A8 9F F8 C0 A8	.<C?.€.6.括填括
0x0020	9F C9 08 00 16 5C 02 00-35 00 61 62 63 64 65 66	焚...\.5.abcdef
0x0030	67 68 69 6A 6B 6C 6D 6E-6F 70 71 72 73 74 75 76	ghijklmnopqrstuv
0x0040	77 61 62 63 64 65 66 67-68 69	wabcdefghi

图12-13 使用ICMP协议响应Ping请求

12.5 运输层的UDP与TCP协议编程

运输层是 TCP/IP 协议栈的核心部分，提供端到端的数据传输服务。如果说网络层提供了主机之间的逻辑通信，那么运输层提供的是运行在不同主机上的进程间的逻辑通信。运输层传输的数据格式称为报文段（Segment）。UDP 和 TCP 是该层的主要协议。

12.5.1 UDP协议概述

UDP（User Datagram Protocol，用户数据报协议）是一种基本的通信协议，只在发送的报文中增加了端口寻址和可选的差错检测功能。它不是一种握手信息协议，不能确认接收到的数据或交换其它流量控制信息。UDP 是一种非连接协议，主机在使用 UDP 发送报文之前，不检测通信的对方是否已联网或指定的目的端口是否可用于通信。

正因为如此，将 UDP 又称为不可靠协议，即如果只使用 UDP 发送数据，则发送方不知道目的主机何时以及是否接收到报文。

发送主机将 UDP 报文段置于 IP 数据报的数据字段中。以太网的 IP 数据报存放在以太帧的数据字段中。接收到以太帧后，目的主机将 UDP 报文段的数据部分传递给报头指定的端口或进程。

虽然 UDP 在可靠性等方面不如 TCP，但是 UDP 实现简单，比较适合于特定的应用场合，而且 UDP 可将报文发送到多个目的主机，包括以广播方式发送到局域网内所有的 IP 地址，或者向指定的 IP 地址以组播方式发送。对于 TCP 而言，实现广播和组播都不现实，原因是：源主机必须与所有目的主机握手，当所发握手信号都回传时，网络冲突必然加剧，从而会降低网络运行性能。

12.5.2 第五个测试实例：UDP报文的发送和接收

1. UDP报文格式

UDP 报文由首部（共 4 个字段，8 个字节）和要传输的数据组成，如图 12-14 所示。

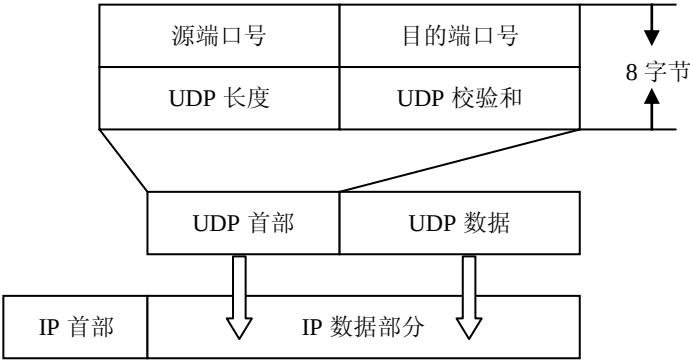


图12-14 UDP报文格式

源端口号：标识发送报文的主机端口或进程，长度为 2 字节，可选。如果接收进程不需要知道发送数据报的进程，则该字段可置为 0。

目的端口号：标识接收报文的目的主机端口或进程，长度为 2 字节。

UDP 长度：指整个数据报的长度，以字节为单位，包括报头，最大值为 65535。长度为 2 字节。

UDP 校验和：是根据 UDP 数据报和伪报头计算得到的差错检测值，可选。该字段为 2 个字节。关于伪报头，可参考计算机网络相关书籍。其实，仅在局域网内部传输报文不需要 UDP 校验和，因为以太帧的校验和已经提供了差错控制。

一个 UDP 数据报最大可达到 65535 个字节，报头占用 8 字节，因此一个数据报最多可以携带 65527 字节的数据。实际上，源主机常将数据报限制在较短的长度内。使用较短数据报的一个原因是，过大的数据报可能不适合目的主机的接收缓冲区大小；或因为接收数据的应用程序可能需要特定长度的报文。因此较短数据报也许更高效。当大的数据报经过不同的网络时，网络协议可能会将数据报拆分为片（Fragment），要求目的主机将这些片重组。但一般来说，在源主机处拆分数据，然后在目的主机处重组，会比依赖 IP 进行这项工作更高效。

2. UDP测试方法

在第四个测试实例的基础上，第五个测试实例新增了 UDP 构件（UDP.h、UDP.c），其实现的功能是：MCF52233 设备接收 PC 机通过网线发送的 UDP 报文段后，再回发给 PC 机。UDP 构件对外接口函数主要有：

```
uint8 UDP_Handle(PACKET p);           //处理接收到的 UDP 包
uint8 UDP_Send(uint16 port, PACKET p); //发送 UDP 包
```

函数 UDP_Receive 的设计思想是：如果 IP 数据报接收成功，则从 UDP 报文段中剥离报头。如果使用了 UDP 校验和，则计算其值，并与接收到的值相比较。根据目的端口号决定将接收到的数据转发到何处。

函数 UDP_Send 的设计思想是：发送 UDP 报文段时，在 UDP 报头的相应位置设置目

的端口号和数据长度。报头中的源端口号和校验和都是可选项。计算校验和需要知道源和目的 IP 地址。在报头后添加待发送的数据。在 IP 数据报的数据部分放置 UDP 报文段。IP 数据报需要源和目的 IP 地址，以及根据报头计算的校验和，将 IP 数据报传送给以太网控制器，以便向网络发送。

图 12-15 是运行在 PC 机（IP 地址为 192.168.159.248）上使用 UDP 协议与 MCF52233 设备（IP 地址为 192.168.159.201）通信的程序界面，若在发送数据文本框中输入“1234567890”，然后单击“UDP 发送”按钮，可以看到在接收数据文本框中也立即出现相同的数据。PC 端源程序代码（使用 VS2005 集成开发环境）请见网上光盘本章目录下的辅助工具文件夹。

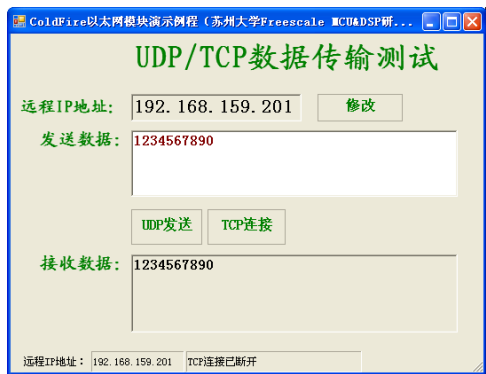


图12-15 使用UDP通信的PC程序界面

与此同时，在 CommView 界面中，也可清楚地看到数据报的发送过程：PC 向 MCU 发送 ARP 请求→MCU 向 PC 发送 ARP 应答→PC 向 MCU 发送 UDP 数据包→MCU 向 PC 发送 UDP 数据包。界面如图 12-16。

	Protocol	MAC Addresses	IP Addresses
1	ARP REQ	IntelCorpo:DA:76:1A => Broadcast	192.168.159.248 -?-> 192.168.159.201
2	ARP RESP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.201 -!-> 192.168.159.248
3	IP/UDP	IntelCorpo:DA:76:1A => AA:BB:CC:00:00:01	192.168.159.248 => 192.168.159.201
4	IP/UDP	IntelCorpo:DA:76:1A <= AA:BB:CC:00:00:01	192.168.159.248 <= 192.168.159.201
<			
0x0000	AA BB CC 00 00 01 00	13-20 DA 76 1A 08 00 45 00	?... 越...E.
0x0010	00 27 44 A1 00 00 80	11-35 12 C0 A8 9F F8 C0 A8	. 'D?.e. 5. 括填括
0x0020	9F C9 04 D1 04 D2 00	13-2B 0D 31 32 33 34 35 36	焚.??+. 123456
0x0030	37 38 39 30 00		7890.

图12-16 使用CommView查看UDP包

12.5.3 TCP协议概述

TCP 称为面向连接的协议，因为进程在交换数据之前必须先彼此建立通信连接。TCP 是一种可靠的协议，因为握手、校验以及序列号和确认号等使源主机可验证数据已正确地到达目的主机。

TCP 报文段由报头和可选的数据组成（也可能传输不含数据的报头，用于发送状态或

控制信息)。术语“报文段”表示单个 TCP 报文段只是完整 TCP 数据传输的一部分。实际上，每次成功的数据传输至少需要 2 个报文段。源主机发送一个或多个包含数据的报文段，目的主机也发送一个或多个报文段确认已接收到的数据。一次确认可确认多个报文段。与 UDP 相比，不同的是，每个 UDP 数据报都是独立的单元，不需要额外的通信。相同的是，两者都使用端口号表示源和目的主机的进程。

两个进程使用 TCP 发送和接收数据之前，首先要通过 3 次握手建立进程所在的计算机之间的连接。在完成握手时，每台计算机都要确认握手中指定的端口可用于接收来自另一台计算机指定端口的信息，然后双方才可以使用连接向对方计算机发送 TCP 报文段。

要关闭连接，双方都需要发送关闭连接的请求，并且等待对方对请求的确认。

12.5.4 第六个测试实例：TCP报文的发送和接收

1. TCP报文格式

图 12-17 给出了 TCP 报文格式。一个 TCP 报文由 TCP 首部和 TCP 数据部分组成。其中，TCP 首部以固定的 20 字节开头，后跟一些可选项。

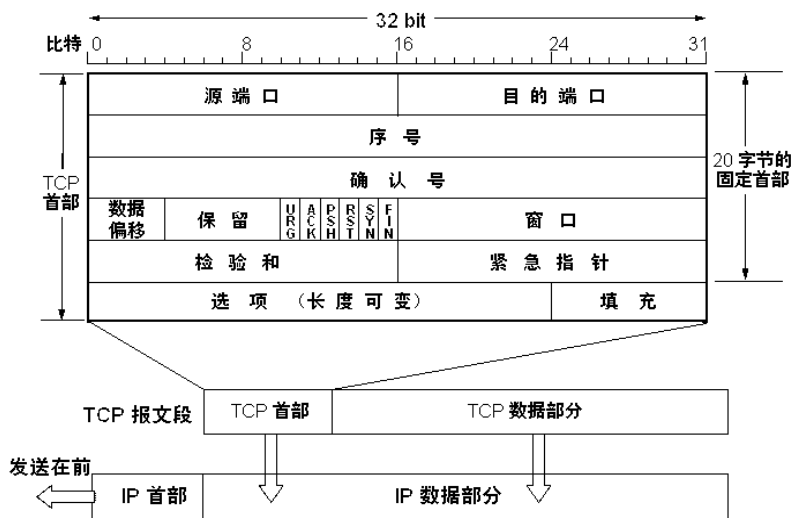


图12-17 TCP报文格式

源端口（16 位）：它（连同源主机 IP 地址）标识源主机的一个应用进程。

目的端口（16 位）：它（连同目的主机 IP 地址）标识目的主机的一个应用进程。这两个值加上 IP 报头中的源主机 IP 地址和目的主机 IP 地址唯一确定一个 TCP 连接。

序号（32 位）：用来标识从 TCP 源端向 TCP 目的端发送的数据字节流，它表示在这个报文段中的第一个数据字节的顺序号。如果将字节流看作在两个应用程序间的单向流动，则 TCP 用序号对每个字节进行计数。序号是 32 位的无符号数，序号到达 $2^{32}-1$ 后又从 0 开始。当建立一个新的连接时，SYN 标志变为 1，顺序号字段包含由这个主机选择的该连接的初始顺序号 ISN（Initial Sequence Number）。

确认号 (32 位): 包含发送确认的一端所期望收到的下一个顺序号。因此, 确认序号应当是上次已成功收到数据字节顺序号加 1。只有 ACK 标志为 1 时确认序号字段才有效。TCP 为应用层提供全双工服务, 这意味数据能在两个方向上独立地进行传输。因此, 连接的每一端必须保持每个方向上的传输数据顺序号。

数据偏移 (4 位): 该字段即为 TCP 报头长度, 以 32 比特为单位, 它实际上指明数据从哪里开始。需要这个值是因为任选字段的长度是可变的。这个字段占 4 位, 因此 TCP 最多有 60 字节的首部。然而, 没有任选字段, 正常的长度是 20 字节。

保留 (6 位): 保留给将来使用, 目前必须置 0。

URG: 为 1 表示紧急指针有效, 为 0 则忽略紧急指针值。

ACK: 为 1 表示确认号有效, 为 0 表示报文中不包含确认信息, 忽略确认号字段。

PSH: 为 1 表示是带有 PUSH 标志的数据, 指示接收方应该尽快将这个报文段交给应用层而不用等待缓冲区装满。

RST: 用于复位由于主机崩溃或其他原因而出现错误的连接。它还可以用于拒绝非法的报文段和拒绝连接请求。一般情况下, 如果收到一个 RST 为 1 的报文, 那么一定发生了某些问题。

SYN: 同步序号, 为 1 表示连接请求, 用于建立连接和使顺序号同步。

FIN: 用于释放连接, 为 1 表示发送方已经没有数据发送了, 即关闭本方数据流。

窗口大小 (16 位): 数据字节数, 表示从确认号开始, 本报文的源方可以接收的字节数, 即源方接收窗口大小。窗口大小是一个 16 位字段, 因而窗口大小最大为 65535 字节。

校验和 (16 位): 此校验和是对整个的 TCP 报文段, 包括 TCP 头部和 TCP 数据, 以 16 位字进行计算所得。这是一个强制性的字段, 是由发送端计算和存储, 并由接收端进行验证。

紧急指针 (16 位): 只有当 URG 标志置 1 时紧急指针才有效。紧急指针是一个正的偏移量, 和顺序号字段中的值相加表示紧急数据最后一个字节的序号。TCP 的紧急方式是发送端向另一端发送紧急数据的一种方式。

选项: 最常见的可选字段是最长报文大小, 又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段 (为建立连接而设置 SYN 标志的那个段) 中指明这个选项, 它指明本端所能接收的最大长度的报文段。选项长度不一定是 32 位字的整数倍, 所以要加填充位, 使得报头长度成为整字数。

TCP 数据: TCP 报文段中的数据部分是可选的。在一个连接建立和一个连接终止时, 双方交换的报文段仅有 TCP 首部。如果一方没有数据要发送, 就使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中, 也会发送不带任何数据的报文段。

2. TCP连接的建立、数据传输及关闭

1) 建立连接

TCP 连接需要三步完成, 也就是常说的“三次握手”。

第 1 步: 客户端向服务器发送一个 TCP 报头请求建立连接。报头中, 初始序列号是客户端随机产生的一个值, 确认号是 0; TCP 报头的 SYN 标志置位, 客户端告诉服务端

序列号区域合法，需要检查。

第 2 步：服务器收到这个同步请求数据包后，会对客户端进行一个同步确认。这个数据包中，序列号是服务器随机产生的一个值，确认号是客户端的初始序列号+1；同时，SYN 标志置位，ACK 标志置位，表示确认号有效。

第 3 步：客户端收到这个同步确认数据包后，再对服务器进行一个确认。该数据包中，序列号是上一个同步请求数据包中的确认号值，确认号是服务器的初始序列号+1；同时，ACK 标志置位，表示确认号有效。

2) 数据传输

在 TCP 建立连接后，就可以开始进行双向数据传输了。客户端向服务器发送数据包的步骤是：

首先，客户端向服务器发送一个带有数据的数据包，该数据包中的序列号和确认号与建立连接第三步的数据包中的序列号和确认号相同；ACK 标志置位，表示确认号有效。

然后，服务器收到该数据包后，向客户端发送一个确认包，在确认包中，序列号为接收到的数据包中的确认号值，而确认号为接收到的数据包中的序列号+数据包中所带数据的大小。服务器向客户端发送数据包的步骤与此类似。

3) 关闭连接

建立一个连接需要三个步骤，而关闭一个连接需要经过四个步骤。因为 TCP 连接是双向传输的工作模式，所以每个方向上需要单独关闭。在 TCP 关闭连接时，首先关闭的一方（即发送第一个终止数据包）将执行主动关闭，而另一方（收到这个终止数据包的）再执行被动关闭。关闭连接的步骤是：

第 1 步：服务器完成它的数据发送任务后，会主动向客户端发送一个终止数据包，以关闭在这个方向上的 TCP 连接。该数据包中，序列号为客户端发送的上一个数据包中的确认号值，而确认号为服务器发送的上一个数据包中的序列号；同时，FIN 标志置位，表示要关闭连接。

第 2 步：客户端收到服务器发送的终止数据包后，将对服务器发送确认信息，以关闭该方向上的 TCP 连接。这时的数据包中，序列号为第 1 步中的确认号值，而确认号为第 1 步的序列号+1；同时，ACK 标志置位，表示确认号有效。

第 3 步：同理，客户端完成它的数据发送任务后，也会向服务器发送一个终止数据包，以关闭在这个方向上的 TCP 连接。该数据包中，序列号和确认号与第 2 步相同，同时，FIN 标志置位，表示要关闭连接。

第 4 步：服务器收到客户端发送的终止数据包后，将对客户端发送确认信息，以关闭该方向上的 TCP 连接。这时在数据包中，序列号为第 3 步中的确认号值，而确认号为第 3 步中的序列号+1；同时，ACK 标志置位，表示确认号有效。

3. TCP 测试方法

在第五个测试实例的基础上，第六个测试实例新增了 TCP 构件（TCP.h、TCP.c），其实现的功能是：MCF52233 设备响应 PC 机发出的 TCP 连接请求和释放连接请求，接收

PC 机发出的数据包，并将数据包中的数据回发给 PC 机。

TCP 构件对外接口函数：

```
uint8 TCP_Handle(PACKET p);           //处理接收到的 TCP 包
uint8 TCP_Send(uint16 port, PACKET p) //发送 TCP 包
```

TCP 通信的过程是：建立连接→数据传输→关闭连接。单击图 12-18 中的“TCP 连接”按钮，将出现如图 12-15 所示界面。单击“TCP 发送”按钮，将发送数据文本框中的内容封装成 TCP 数据包发送给 MCF52233 设备，MCF52233 设备接收到数据包后进行解析，并再次封装成 TCP 数据包发送给 PC 机。单击“断开 TCP 连接”按钮，将释放 TCP 连接。



图12-18 使用TCP通信的PC程序界面

12.6 应用层的HTTP协议编程

应用层位于运输层之上，包含的协议有：超文本传输协议 HTTP（Hyper Text Transport Protocol）、文件传输协议 FTP（File Transport Protocol）和简单邮件传输协议 SMTP（Simple Mail Transport Protocol）等。

12.6.1 HTTP协议概述

HTTP 是超文本传输协议的缩写，它是属于应用层的面向对象的协议，主要特点有：

- （1）支持客户端/服务器模式。
- （2）客户端向服务器请求服务时，只需传送请求方法和路径。常用的请求方法有 GET、HEAD、POST。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
- （3）HTTP 允许传输任意类型的数据对象，正在传输的类型由 Content-Type 加以标记。
- （4）限制每次连接只处理一个请求。服务器处理完客户端的请求并收到应答后，即断开连接。采用这种方式可以节省传输时间。
- （5）HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力，意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。

在服务器不需要先前信息时它的应答就较快。

HTTP 协议由两部分程序实现：一个是客户端程序，一个是服务器程序，它们运行于不同的端系统中，通过交换 HTTP 报文进行会话。HTTP 消息分为请求消息和响应消息两类。要实现 Web 服务器，必须清楚这两部分的构成。

1. HTTP协议的运作方式

基于 HTTP 协议的客户端/服务器模式的信息交换分为以下四个步骤：

第 1 步：建立连接。连接的建立是通过申请套接字（Socket）实现的。客户端打开一个套接字并把它约束在一个端口上，如果成功，就相当于建立了一个虚拟文件。以后就可以在该虚拟文件上写数据并通过网络向外传送。

第 2 步：发送请求信息。打开一个连接后，客户端把请求信息发送到服务器的端口上，完成提出请求动作。

第 3 步：发送响应信息。服务器在处理完客户端的请求后，向客户端发送响应信息。

第 4 步：关闭连接。客户端和服务器双方通过关闭套接字来结束对话。

2. HTTP请求信息格式

客户端与服务器建立连接后，发送一个请求信息给服务器。HTTP 请求信息由请求行、请求报头、一个空行和请求正文组成，结构如图 12-19 所示。

请求方法	空格符	URL	空格符	协议版本号	回车符	换行符
域名：值					回车符	换行符
域名：值					回车符	换行符
.....					回车符	换行符
回车符 换行符						
请求正文						

图12-19 HTTP请求消息格式

请求行的格式为：请求方法、空格、URL、空格、协议版本号、回车符、换行符。其中，请求方法有 GET、POST、HEAD 等；URL 是一个统一资源标识符；协议版本号为 HTTP/1.0 或 HTTP/1.1。

请求报头由若干报头域组成，报头域之间用回车换行符隔开。每个报头域的格式为“域名：值”。

例如，一个 HTTP 请求信息内容可能为：

```
GET /form.html HTTP/1.1 回车换行
Accept:image/gif,image/jpeg, 回车换行
Accept-Language:zh-cn 回车换行
Accept-Encoding:gzip,deflate 回车换行
If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT 回车换行
User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) 回车换行
Host:www.suda.edu.cn 回车换行
Connection:Keep-Alive 回车换行
回车换行
```

.....（请求正文）

3. HTTP响应信息格式

服务器接到客户端的请求后，将向客户端发送相应的响应信息。HTTP 响应信息由状态行、响应报头、一个空行和响应正文组成，结构见图 12-20 所示。

版本号	空格符	状态码	空格符	原因短语	回车符	换行符
域名：值					回车符	换行符
域名：值					回车符	换行符
回车符 换行符						
响应正文						

图12-20 HTTP响应消息格式

状态行的格式为：协议版本号、空格、状态码、空格、原因短语、回车符、换行符。其中，状态码是服务器发回的响应状态代码，原因短语为状态代码的文本描述。常见状态码和原因短语有：

200 OK	客户端请求成功
400 Bad Request	客户端请求有语法错误，不能被服务器所理解
401 Unauthorized	请求未经授权
403 Forbidden	服务器收到请求，但是拒绝提供服务
404 Not Found	请求资源不存在
500 Internal Server Error	服务器发生不可预期的错误
503 Server Unavailable	服务器当前不能处理客户端请求

响应报头与请求报头一样，也是由若干报头域组成。

响应正文就是服务器返回的资源的内容。

例如，一个 HTTP 响应信息内容可能为：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu,08 Mar 2007 07:17:51 GMT
Connection: Keep-Alive
Content-Length: 23330
Content-Type: text/html
Expries: Thu,08 Mar 2007 07:16:51 GMT
Cache-control: private
回车换行
.....（响应正文）
```

12.6.2 第七个测试实例：HTTP协议静态页面的实现

在第六个测试实例的基础上，第七个测试实例新增了 HTTP 构件(HTTP.h、HTTP.c)，

其实现的功能是：启动 PC 端的 IE 浏览器，输入 MCF52233 设备的 IP 地址，如“http://192.168.159.201”，则 MCF52233 设备响应 PC 机的 HTTP 请求，将预先放置在 Flash 中的 HTML 静态页面内容发送给 PC 机，并在浏览器中显示出来，界面如图 12-21。

HTTP 构件对外接口函数：

```
uint8 HTTP_Init(); //初始化
uint8 HTTP_Connection(M SOCK s); //建立连接
uint8 HTTP_Send(uint8 session); //发送数据
void HTTP_Handle(uint8 session, PACKET p); //处理接收到的数据
```

限于篇幅，本书未给出各函数实现代码。读者可参阅随书网上光盘。

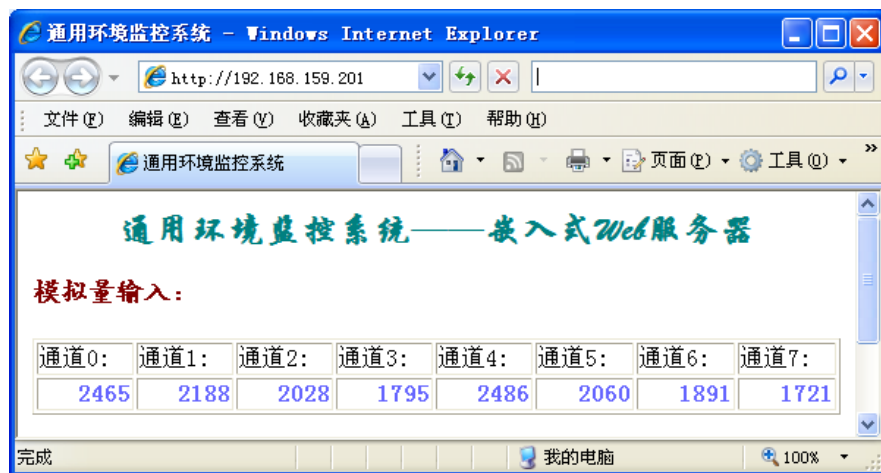


图12-21 PC端浏览器打开的HTTP静态页面

第13章 其它模块

前面章节讨论了 Cold Fire 系列芯片的大部分功能模块,本章讨论前面章节未涉及的功能模块。如芯片初始化设置所涉及相关模块等。主要包括时钟模块、芯片配置模块、系统控制模块、实时时钟模块、电源管理模块及复位控制模块等。

13.1 时钟模块

时钟模块产生系统工作时钟信号。时钟模块允许对处理器的时钟进行配置,内部包含锁相环(Phase-Locked Loop, PLL)时钟,通过内部晶体放大器,可接外部时钟或外部晶体。当然,PLL 也可以被禁用,处理器直接使用外部晶振作为设备时钟。当 PLL 被禁用,且选择外部时钟模式(这种模式也支持外部时钟源)时,设备在复位后开始运行。系统上电复位后的初试化程序应该决定是否使用 PLL,并决定选择 PLL 时钟作为系统时钟源还是直接外部晶振作为系统时钟源。一般情况下,使用 PLL 倍频机制产生内部总线时钟。

所谓锁相环 PLL,它是一种倍频技术。能够通过一定电路,将一个频率较低的时钟信号变换成频率较高的时钟信号。若 MCU 内含有 PLL 电路,可以在 MCU 外部晶振电路中使用较低的晶振频率(干扰较少),而在 MCU 内部,通过对 PLL 的编程,实现内部总线所需要的较高频率。

MCF52233 的时钟模块包含晶体放大器和振荡器(Oscillator, OSC)、锁相环 PLL、分频器(Reduced Frequency Divider, RFD)、状态和控制寄存器以及控制逻辑等。时钟模块框图见图 13-1。

时钟模块具有以下特点: 25MHz 晶振输入(外接晶振频率);为集成的 EPHY 模块提供时钟;片内 PLL 可生成最高 60MHz 的内核时钟频率 F_{sys} , 其计算公式:

$$F_{sys} = F_{ref} \times 2 \times (MFD + 2) / (2^{RFD})$$

其中,参考时钟频率 $F_{ref} = 25\text{MHz} / (CCHR + 1)$, CCHR 为 PLL 分频系数。例如, CCHR=4、MFD=4、RFD=0 时,则 $F_{ref} = 25 / (4 + 1) = 5\text{MHz}$, $F_{sys} = 5 \times 2 \times (4 + 2) / 1 = 60\text{MHz}$ 。

对时钟模块的编程,可以设置两种操作模式:普通 PLL 模式和外部时钟模式。在普通 PLL 模式下,PLL 是完全可编程的,可以将参考时钟倍频到 4 至 18 倍,PLL 参考时钟可以是晶体振荡器或外部时钟,同时有一个分频器 RFD,能够降低这种合成的频率,使锁相环更加稳定。该模式下默认的系统频率是复位后参考时钟频率的 2 倍。倍频后的频率不能超过芯片手册给出的范围(MCF52233 总线频率小于等于 60MHz);例如:若参考频率为 3MHz,那么经 PLL 倍频后可达到 12~54MHz。在外部时钟模式下,PLL 被禁止,外部时钟被应用于 EXTAL 引脚上作为系统时钟源。

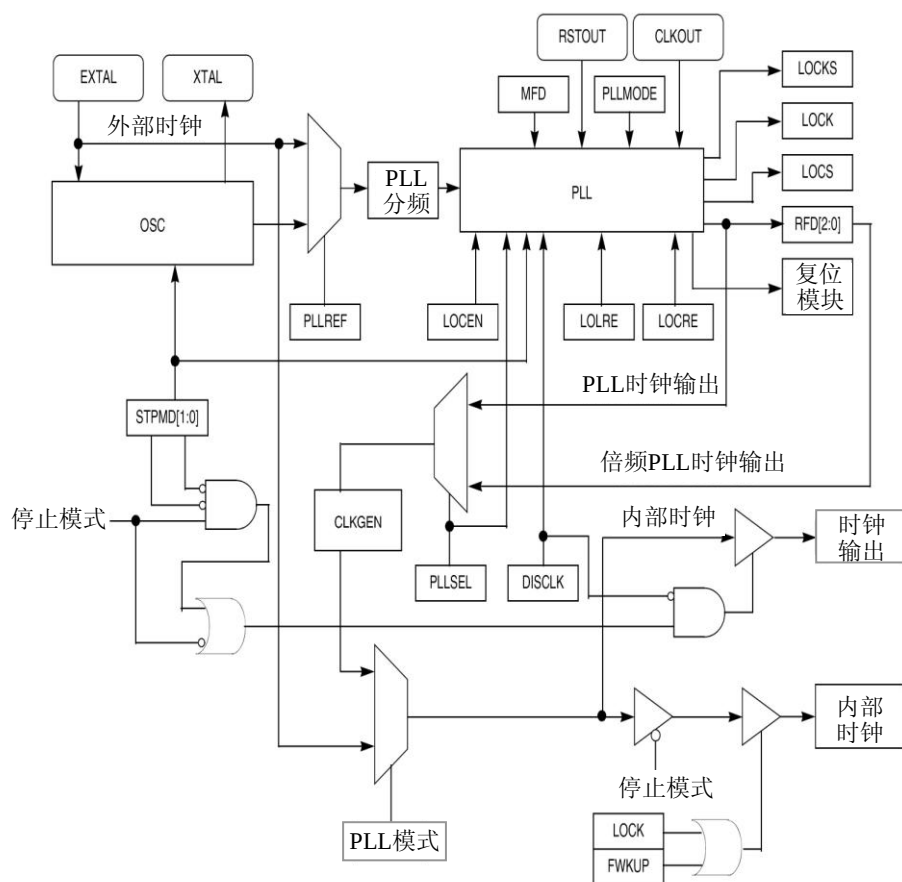


图13-1 时钟模块框图

表 13-1 列出时钟模块引脚功能以及简单说明，详情参考 MCF5223X 数据手册。

表 13-1 时钟模块信号引脚

引脚	功能
EXTAL	振荡器或时钟输入
XTAL	振荡器输出
TCLK/PSTCLK	系统时钟输出
RSTOUT (RSTO)	复位信号
VDDPLL、VSSPLL	PLL电源

13.2 芯片配置模块

芯片配置模块（Chip Configuration Module, CCM）用于控制芯片的配置和操作模式。图 13-2 为芯片配置模块结构图。

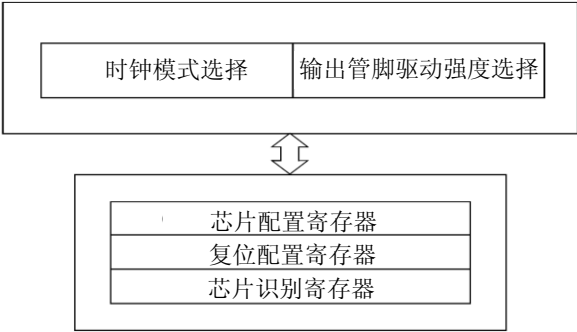


图13-2 芯片配置模块结构图

13.2.1 芯片配置模块简介

芯片配置模块的主要特性有：外部时钟模式或者带有内部或外部参考的锁相环（PLL）模式；输出管脚驱动能力可配置；低功耗配置；处理器状态（PST）和调试数据（DDATA）功能；BDM 或 JTAG 模式。

芯片配置模块具有复位配置、输出强度配置和时钟模式选择共 3 种功能。复位时，复位配置寄存器和外部复位配置信号决定系统功能的操作。如果在复位时产生复位配置信号，则系统将锁存外部复位配置引脚的值；如果在复位时未产生复位配置信号，则复位配置将利用复位配置寄存器来操作。只有在复位配置时，复位配置寄存器设定的功能才能重新配置。在复位配置时，特有功能的选择由芯片配置模块特点而定。

时钟选择允许使用片内系统时钟 PLL 或固定的输入时钟频率。输出管脚驱动强度选择模块可为外部总线接口选择合适的驱动电平；特别地，当使用带有 1~2 个存储设备的简单外部存储系统时，选择较低的驱动能力；当使用较复杂的外部存储设备时，选择较高的驱动能力。

13.2.2 芯片配置模块寄存器定义

芯片配置模块主要有以下寄存器：芯片配置寄存器（CCR）、复位配置寄存器（RCON）和芯片标识寄存器（CIR）。其中，芯片配置寄存器控制芯片的主要配置，复位配置寄存器给出芯片的默认配置，芯片标识寄存器包含一个用于识别芯片的唯一序列号。该模块的存储器映像如表 13-2 所示，这些寄存器仅限于管理员模式下访问。

表 13-2 芯片配置模块存储映像

IPSBAR 偏移量	寄存器	宽度 (位)	访问	复位值
0x11_0004	芯片配置寄存器(CCR)	16	读	0x0001
0x11_0008	复位配置寄存器(RCON)	16	读	0x0000
0x11_000A	芯片识别寄存器(CIR)	16	读	0x2000

1. 芯片配置寄存器（CCR）

芯片配置寄存器（CCR）的地址为 0x110004，它是 16 位寄存器，复位后，除 D0 外其余位为 0。

数据位	D15~D7	D6	D5	D4	D3	D2	D1	D0
定义		SZEN	PSTEN		BME	BMT		

D15~D7—未定义。

D6—SZEN，外部引脚的 TSIZ[1:0]使能位。SZEN=1，TSIZ[1:0]功能使能。

D5—PSTEN，外部引脚的 PST[3:0]/DDATA[3:0]使能位。PSTEN=1，使能处理器状态（PST）和调试数据（DDATA）功能。

D4—未定义。

D3—BME，总线监视器使能位。BME=1，使能总线监视器。

D2~D0—BMT，总线监视器定时选择位。这 3 位用于选取总线监视器的超时周期。其中，D2~D0 的取值范围从 000 到 111，分别对应数值 2^{16} （即 65536）到 2^9 （即 512）。

2. 复位配置寄存器（RCON）

复位配置寄存器（RCON）的地址为 0x110007，它是 16 位只读寄存器。复位后全为 0。

数据位	D15~D6	D5	D4	D3	D2	D1	D0
定义		RLOAD					MODE

D15~D6—未定义。

D5—RLOAD，驱动强度加载。反馈默认驱动强度配置。RLOAD=0，部分驱动强度（默认值）；RLOAD=1，全驱动强度。

D4~D1—未定义。

D0—MODE，芯片配置模式。MODE=0，单芯片模式，芯片从 Flash 存储器中取指令；MODE=1，主芯片模式（一般不用）。

3. 芯片标识寄存器（CIR）

芯片标识寄存器（CIR）的地址为 0x11000A，它是 16 位只读寄存器。其中，D15~D6 位是唯一的设备号，不同类型 MCU 的具体设备号如下：MCF52230：0x48，MCF52231：0x49，MCF52233：0x4A，MCF52234：0x4B，MCF52235：0x4C。D5~D0 位是原件修正号，该修正号为每个新增原件添加一个序列号，每次增 1，按时间顺序从 0 开始赋值。

13.3 系统控制模块

13.3.1 系统控制模块功能概述

系统控制模块（System Control Module，SCM）为各种不同功能提供了控制与状态参数，包括内部外设系统（IPS）以及内核存储空间的基址寻址和地址空间屏蔽。CPU 内核支持两块存储空间，一个是内部 Flash，另一个是内部 SRAM。

系统控制模块（SCM）包括有以下几个可编程寄存器：1）IPS 寄存器（IPSBAR）可以设置 1GB 外设空间的基地址和用户控制位；2）处理器存储器基址寄存器（RAMBAR）；3）系统控制寄存器包含内核复位状态寄存器（CRSR）显示上次复位类型、内核看门狗服务寄存器（CWSR）服务看门狗定时器、内核看门狗控制寄存器（CWCR）控制看门狗定时器；4）系统总线主控端可编程模块（MPARK）；5）系统存取控制单元（SACU）可编程模块包含主控端特权寄存器（MPR）、外设存取控制寄存器（PACR）、分组外设存取控制寄存器（GPACR0，GPACR1）。

系统存取控制单元（SACU）提供总线安全传输到系统地址空间的必要机制。总线仲裁编程模块驻留在 SCM 中，SCM 为总线主控端管理仲裁器提供必要的控制信号。

通过一种特殊的软件服务序列，内核看门狗定时器（CWT）可避免由于软件无限循环引起的系统死锁，如果周期性的软件服务没有发生，CWT 将超时并发出响应（系统复位或中断），以达到恢复或修正的目的。

13.3.2 系统控制模块的可编程寄存器

1. 内部外设系统基址寄存器（IPSBAR）

内部外设系统基址寄存器（IPSBAR）是 32 位寄存器，D31~D30 位指定了与内部外设相关的 1GB 存储空间的地基址，D29~D1 位保留，D0 位使能/禁止 IPS。复位时，基址从默认的 0x40000-0000 处加载且被标记为有效（D0=1）。如果需要，可在复位后将内部模块的地址空间更新为其他值。

2. 内存基址寄存器（RAMBAR）

内存基址寄存器（RAMBAR）是 32 位寄存器，D31~D16 位定义了 SRAM 的基址，D9 位禁止/使能其它模块访问 SRAM，D15~D10 和 D8~D0 位保留，默认值为 0。

静态 RAM（SRAM）可以被内核或其它总线主控端直接访问。可以在处理器的单指令周期内访问存储器，这样便能够使用双缓冲机制发挥最优的系统性能。例如：双缓冲应用中的 DMA 通道（也称乒乓机制）可以将数据载入双端口 SRAM 中的一个端口，同时处理器在 SRAM 中的另一个端口处理数据；一旦处理器完成数据计算，它便处理刚刚载入缓冲区中的数据，同时 DMA 将刚刚计算的数据从另一个缓冲区中移出，并将下一个数据块载入到空闲的存储区。处理器与 DMA 采用“乒乓机制”，在 SRAM 的双端口区域中交替运行。

ColdFire 系列 MCU 内部的 SRAM 模块是一个通用存储器块，处理器可在一个时钟周期内访问。存储器块的位置可指定为 4GB 存储空间中任一 32K 倍数的地址。SRAM 是存储临界区代码、数据结构和系统堆栈的理想区域，这是因为 SRAM 模块与处理器内部采用高速总线连接，能快速响应内核初始化或调试模式的访问命令。

3. 内核复位状态寄存器（CRSR）

内核复位状态寄存器（Reset Status Register, CRSR）是 8 位寄存器，D7 位可读写，其余位未定义。D7 位反映了最近一次的复位源。复位完成后，由控制逻辑自动更新该寄存器。若读 D7 位为 1，表明最近一次复位由 $\overline{\text{RSTI}}$ 引起。通过写 1 可以将该位清零，写 0 无效。

注意：复位状态寄存器（RSR）给出了除看门狗定时器复位以外的所有复位源，如软件复位、上电复位等。

4. 内核看门狗控制寄存器（CWCR）

内核看门狗控制寄存器（CWCR）是 8 位寄存器。复位全为 0。

数据位	D7	D6	D5~D3	D2	D1	D0
定义	CWE	CWRI	CWT[2:0]	CWTA	CWTAVAL	CWTIF

D7—CWE，看门狗使能位。CWE=0，禁止；CWE=1，允许。

D6—CWRI，看门狗中断选择位。CWRI=0，看门狗定时器溢出时，将产生中断；CWRI=1，无效。

D5~D3—CWT[2:0]，看门狗计数延迟。该位域取值决定了看门狗溢出周期。

D2—CWTA，看门狗确认使能位。CWTA=0，禁止；CWTA=1，允许。

D1—CWTAVAL，看门狗确认有效位。CWTAVAL =0，无效；CWTAVAL =1，有效，写 1 清零。

D0—CWTIF，看门狗定时器中断标志位。CWTIF =0，未产生中断；CWTIF =1，已经发生中断，写 1 清零。

5. 内核看门狗服务寄存器（CWSR）

COP 看门狗的主要用途是在程序非正常运行时，将程序进行热复位，以达到控制程序正常运行的功能。在使用 COP 看门狗时，涉及到以下几方面的考虑：如何开关看门狗、看门狗的定时溢出时间的设置以及如何复位看门狗定时器。

（1）COP 看门狗的设置方法

在使用 COP 看门狗之前，应当对 COP 看门狗进行设置，主要包括使能 COP 看门狗并设置 COP 看门狗的定时溢出时间，参考代码如下：

```
MCF_SCM_CWCR |= MCF_SCM_CWCR_CWT(0b110) | MCF_SCM_CWCR_CWE;
```

这里将看门狗的溢出时间设置为 2^{27} 个总线时钟周期，也就是说，如果在 2^{27} 个总线时钟周期这段时间内不复位看门狗定时器，那么系统将进行热复位，使系统回到一个可行的开始点。

（2）COP 看门狗有效性的程序验证方法

在设置完 COP 看门狗后，首先应当先验证 COP 看门狗是否已经正常工作，一个比较简单的验方法是：在主程序的主循环之前往串口发送一串信息，如“COP is running!”，如果发现程序运行时每隔一段时间便会发送该串信息，表明系统在不断复位，这样就可以证明 COP 看门狗在正常运行。这里要注意的是要保证程序在发送“COP is running!”这串信息之前，看门狗定时器未达到定时溢出时间。

（3）COP 看门狗的使用

在使用 COP 看门狗时，主要需要考虑如何复位 COP 看门狗定时器以及何时复位 COP 看门狗定时器这两个问题。通过向 CWSR 寄存器连续写入 0x55 和 0xAA 可以复位 COP 看门狗定时器。在使用时，建议只在主程序中复位 COP 看门狗定时器，而在中断程序和子程序中应当避免复位 COP 看门狗定时器。复位 COP 看门狗定时器的语句如下：

```
MCF_SCM_CWSR = 0x55;
```

```
MCF_SCM_CWSR = 0xAA;
```

6. 系统存取控制单元（SACU）

系统存取控制单元（SACU）提供必要机制保证到内部模块映射的地址空间安全的总线传输。SACU 支持 2 种特权等级模式：管理员模式和用户模式。典型地，管理员模式可以访问所有系统资源，而用户模式不可以访问系统控制和配置寄存器。在许多系统中，操作系统运行在管理员模式，应用软件运行在用户模式。

SACU 是系统控制模块（SCM）的一部分，而实际的访问控制逻辑是仲裁控制器的一部分。内部外设系统空间的每一次总线传输首先要检查是否有对应存储空间的访问权限，如果有访问权限，则总线访问继续；对于目标存储空间，如果没有足够的访问权限，则立即以异常模式终止传输，且目标模块不能被访问。

与 SACU 编程模块相关的寄存器主要有：主机特权寄存器（MPR）、外设访问控制寄存器

(PACR0~PACR8)、分组外设访问控制寄存器 (GPACR0 和 GPACR1)。

13.3.3 对系统初始化过程的理解

MCF52233 的系统初始化主要涉及到的文件是 sysinit.h 和 sysinit.c。在 sysinit.c 文件中的 mcf52233_init 函数主要完成了 MCF52233 的系统初始化的工作。MCF52233 的系统初始化主要包括以下几个方面：中断向量表的配置、数据的初始化、实时时钟频率、COP 看门狗、系统时钟以及 SRAM 的设置。下面将结合具体的初始化代码解析初始化的步骤。

1. 中断向量表的配置

中断向量表的配置主要是设置向量基地址寄存器的值，将程序中的中断向量表的首地址写入到向量基地址寄存器中，以便以后在中断处理时，可以根据该中断向量表的首地址找到相应的中断处理程序。具体的初始化代码如下：

```
mcf5xxx_wr_vbr((uint32)VECTOR_TABLE);    //配置中断向量表
```

2. 数据的初始化

数据的初始化操作主要包括将已经初始化的 ROM 中的数据移到 RAM 中并赋值的操作和未初始化的 RAM 中的数据的初始化操作。

对已经初始化的 ROM 中的数据，要将其移到 RAM 中，并且用 ROM 中的初始值对其进行赋值，具体的代码如下：

```
//将已初始化的数据从 ROM 移到 RAM
dp = (uint8 *)__DATA_RAM;
sp = (uint8 *)__DATA_ROM;
n = (uint32)(__DATA_END - __DATA_RAM);
while (n--)
    *dp++ = *sp++;
```

对于未初始化的 RAM 中的数据，本系统初始化程序均将其初始化为 0，具体的代码如下：

```
//将所有未初始化的数据初始化为 0
sp = (uint8 *)__BSS_START;
n = (uint32)(__BSS_END - __BSS_START);
while (n--)
    *sp++ = 0;
```

3. 实时时钟频率的设置

实时时钟是 MCF52233 内部集成的时钟模块，对其频率的设置需要注意的是要将该频率设置为 1Hz，以达到精确计时的目的，计算公式如下：

$$f_{\text{RTC}} = f_{\text{OSC}} \div (\text{RTCDF} + 1)$$

其中， f_{OSC} 为晶振频率， f_{RTC} 为实时时钟频率，RTCDF 为分频因子，对于本初始化程序，晶振频率为 25MHz，因此，设置的分频因子为 $25000000 - 1 = 24900000$ 。对实时时钟的分频因子寄存器的设置的具体代码如下：

```
MCF_CLOCK_RTCDR = 25000000;    //设置实时时钟频率在 1Hz
```

4. COP看门狗的设置

COP 看门狗的设置主要是开关看门狗以及看门狗的定时溢出时间的设置。若想禁用看门狗，则执行以下代码即可：

```
MCF_SCM_CWCR = 0;    //禁止看门狗
```

若想使用看门狗，执行下列代码，其中 MCF_SCM_CWCR_CWE 是使能 COP 看门狗的掩码，MCF_SCM_CWCR_CWT 用于设置看门狗的定时溢出时间。

```
MCF_SCM_CWCR |= MCF_SCM_CWCR_CWT(0b110) | MCF_SCM_CWCR_CWE;
```

5. 系统时钟的设置

系统时钟主要是从外部的晶振时钟产生内部的系统时钟，这里主要涉及到对 SYNCR 寄存器的设置，具体的设置代码如下：

```
//fsys 计数公式：fsys = fref × 2 × (MFD + 2) ÷ (2RFD);  
//该程序中，CCHR 是 4(复位后默认值),MFD=4,RFD=0。系统频率计算：  
//为 25MHz/(CCHR+1) × 2 × (MFD + 2) ÷ (2RFD) = 60Mhz.
```

```
MCF_CLOCK_SYNCR = (MCF_CLOCK_SYNCR_MFD(4)
```

```
    | MCF_CLOCK_SYNCR_CLKSRC  
    | MCF_CLOCK_SYNCR_PLLMODE  
    | MCF_CLOCK_SYNCR_PLEN);
```

```
//等待时钟稳定
```

```
while (!(MCF_CLOCK_SYNSR & MCF_CLOCK_SYNSR_LOCK))  
{  
}
```

6. SRAM的设置

SRAM 的设置主要包括 SRAM 基地址的设置和是否允许片上的模块访问内部的 SRAM。具体的设置代码如下：

```
MCF_SCM_RAMBAR = (0
```

```
    | MCF_SCM_RAMBAR_BA(SRAM_ADDRESS)    //设置 SRAM 的基地址  
    | MCF_SCM_RAMBAR_BDE);    //允许片上的 DMA 和 FEC 访问 SRAM
```

13.4 实时时钟模块

实时时钟（RTC）模块提供 10 个 32 位的寄存器，实现天、时、分和秒的计数、报警功能。这些寄存器分别为：

(1) 时分计数寄存器（HOURMIN）：D12~D8 设置“时”，D5~D0 设置“分”，其余位未定义。

(2) 秒计数寄存器（SECONDS）：D5~D0 设置“秒”，其余位未定义。

(3) 时分报警寄存器（ALRM_HM）：D12~D8 设置“时”报警值，D5~D0 设置“分”报警值，其余位未定义。

(4) 秒报警寄存器（ALRM_SEC）：D5~D0 设置“秒”报警值，其余位未定义。

(5) RTC 控制寄存器（RTCCTL）：仅两位有效，D7（EN）位为 0 禁止实时时钟，为 1 使能

实时时钟；D0（SWR）位为 0 无影响，为 1 复位实时时钟模块。

(6) RTC 中断使能寄存器（RTCIENR）：D31~D6 未定义，D5~D0 分别为时中断、秒中断、天中断、报警中断、分中断以及秒表中断的使能为，1 表示允许中断，0 表示禁止中断。

(7) RTC 中断状态寄存器（RTCISR）：D31~D6 未定义，D5~D0 表示是否发生了某些中断。1-发生了中断，0-未发生中断。注意：只有当中断使能寄存器 RTCIENR 中的相应中断允许时，中断状态寄存器 RTCISR 的 D5~D0 位才有意义。

(8) STPWCH（秒表寄存器）：D31~D6 未定义，D5~D0 位域值为秒表计数值。每当分加 1 的时，此寄存器的值会减 1，当值减到零时将产生秒表中断。

(9) 日计数寄存器（DAYS）：D31~D16 未定义，D15~D0 为当前的天、时、分和秒。

(10) 日报警寄存器（ALRM_DAY）：D31~D16 未定义，D15~D0 为报警时间。当该寄存器中的值和当前时间匹配时，将产生报警中断。

下面给出 RTC 实时时钟模块的一个示例程序，该程序的功能是模拟系统时钟，实现“时分秒”时钟的功能。在 PC 方有 MCU 当前时钟的显示界面，也可以通过该软件设置 MCU 的系统时间为 PC 的时间。该示例程序主要包括以下几部分：

1. RTC初始化程序

RTC 初始化程序主要是对 RTC 的时间进行设置。

```
//-----*
//函数名: RTCInit                                *
//功 能: 初始化RTC                                *
//参 数: nHour-时(0~23); nMinute-分(0~59); nSecond-秒(0~59) *
//返 回: 无                                        *
//说 明: 初始化结束后, RTC是禁止的, 与RTC相关的中断也是禁止的 *
//-----*
void RTCInit(uint8 nHour, uint8 nMinute, uint8 nSecond)
{
    //1、禁止RTC
    DisableRTC();
    //2、设置RTC时钟频率(RTC时钟频率=(晶振频率-1)/1Hz)
    MCF_CLOCK_RTCDR = 24900000;
    //3、设置RTC的时、分、秒计数器
    MCF_RTC_HOURLMIN = (uint32)((uint16)((uint16)(nHour<<8) & 0xFF00)
    | (uint16)(((uint16)nMinute) & 0x00FF));
    MCF_RTC_SECONDS = (uint32)nSecond;
    //4、禁止与RTC相关的所有中断
    MCF_RTC_IER = 0x00000000;
}
```

2. RTC中断程序

RTC 中断程序主要处理了秒增加的中断，当 RTC 得秒增加 1 以后，设置标志，以通知主程序当前 RTC 时间有变化，向 PC 发送当前的 RTC 时间。

```
//RTC_handler中断处理程序
__interrupt__
void RTC_handler(void)
{
    DisableInterrupts();    //关总中断
```

```

//RTC秒中断
if ((MCF_RTC_ISR & MCF_RTC_ISR_1HZ))
{
    g_nIsSendTime = 1;
    MCF_RTC_ISR |= MCF_RTC_ISR_1HZ;
}

EnableInterrupts();    //开总中断
}

```

3. RTC时间设置程序

该函数主要实现对 RTC 时间的设置功能。

```

//-----*
//函数名: RTCTimeSetting                                *
//功 能: 设置RTC当前的“时分秒”                        *
//参 数: RTCTime-保存“时分秒”的数组(1字节时+1字节分+1字节秒) *
//      len-数组的长度                                *
//返 回: 无                                            *
//-----*
void RTCTimeSetting(uint8 RTCTime[], uint8 len)
{
    //1、禁止RTC
    DisableRTC();
    //2、设置当前RTC时间
    MCF_RTC_HOURMIN = (uint32)((uint16)((uint16)(RTCTime[0]<<8) & 0xFF00)
                               | (uint16)((uint16)RTCTime[1] & 0x00FF));
    MCF_RTC_SECONDS = (uint32)RTCTime[2];
    //3、使能RTC
    EnableRTC();
}

```

4. 主程序

主程序循环查询当前的 RTC 时间是否有变化，如有变化则向 PC 发送当前的 RTC 时间。

```

while (1)
{
    if (g_nIsSendTime)
    {
        UartSendN(0, 3, g_Time);    //发送当前“时分秒”
        g_nIsSendTime = 0;
    }
}

```

13.5 电源管理模块

ColdFire 系列 MCU 可以根据需要工作在 RUN、WAIT、DOZE 和 STOP 等功耗模式。

13.5.1 低功耗模式

通过执行一条 STOP 指令, MCF5223X 系列芯片将会进入低功耗模式。低功耗模式有四种, 分别为 RUN、WAIT、DOZE 和 STOP。系统具体进入哪种模式由低功耗控制寄存器 LPCR 中 LPMD 位编程决定。在低功耗模式下, MCF5223X 系列芯片能够独立地关闭大多数外设和外部 CLKOUT 引脚。

RUN 模式是正常的系统运行模式。

WAIT 模式仅被用来停止 CPU 和内存时钟运行, 直到一个唤醒事件被检测到。在该模式下, 外设可通过编程使其继续运行并且可通过产生中断使 CPU 退出 WAIT 模式。

在 DOZE 模式下, 对每个外设可定义不同的运行特点, 外设继续运行并可产生中断, 使 CPU 退出 DOZE 模式返回 RUN 模式, 每个被停止的外设可重新按先前定义方式运行。

在 STOP 模式下, 系统的所有时钟和外设都停止工作。退出 STOP 模式后, 外围设备恢复到停止前的状态并正常运作。

13.5.2 低功耗模式下的外设行为

下面介绍几种外设 in 低功耗模式下的工作状态以及退出低功耗模式的方法。

(1) ColdFire 内核: 在任何的低功耗模式下都可以禁用 ColdFire 内核。当退出低功耗模式时不需要恢复时间。

(2) 静态随机访问存储器 (SRAM): SRAM 在低功耗模式下被关闭。当离开低功耗模式时不需要恢复时间。

(3) Flash: 如果没有被访问, Flash 模块处于低功耗状态。

(4) 系统控制模块 (SCM): SCM 的内核看门狗定时器能够使设备退出除 STOP 模式以外的所有低功耗模式。在 STOP 模式下, 所有的时钟停止, 且内核看门狗也停止运行。当内核看门狗中断使能时, 可通过看门狗中断使设备退出低功耗模式。

(5) DMA 控制器 (DMA0~DMA3): 在 WAIT 和 DOZE 模式下, DMA 控制器可以通过完成一次传输或一个错误条件产生中断, 使设备退出低功耗模式。

(6) UART 模块 (UART0、UART1 和 UART2): 在 WAIT 和 DOZE 模式下, UART 可产生一个中断来退出低功耗模式。首先, 清除发送使能位 (TE) 或接收使能位 (RE) 以禁止 UART 功能, 然后, UART 不受 WAIT 模式影响且可生成一个中断来退出此模式。在 STOP 模式下, UART 停止并冻结寄存器值、状态机和外部引脚, UART 时钟也被关闭。退出 STOP 模式后, UART 恢复到进入低功耗模式前的状态。

(7) IIC 模块: 当通过设置 I2CR 的 IEN 位使能 I2C 模块, 并且当设备不在 STOP 模式下时, IIC 模块可操作且可生成一个中断使设备退出低功耗模式。

其它模块在低功耗模式下的工作状态可参阅芯片数据手册。

13.5.3 电源管理模块寄存器定义

电源管理模块共涉及 6 个 8 位或 32 位编程寄存器。它们是外设电源管理高/低位寄存器 (PPMRH、PPMRL)、低功耗控制寄存器 (LPCR)、低功耗中断控制寄存器 (LPICR)、外设电源管理清除寄存器 (PPMRC) 以及外设电源管理设置寄存器 (PPMRS)。

- (1) 外设电源管理高/低寄存器 (PPMRH、PPMRL): 均为 32 位寄存器, 可通过设置相应位来使能或禁止各模块 (如 EPHY、FlexCAN 等) 的时钟, 具体设置需参考芯片手册。
- (2) 低功耗控制寄存器 (LPCR): 8 位寄存器, D7~D6 为低功耗模式选择位, 11: STOP, 10: WAIT, 01: DOZE, 00: RUN。D4~D3 控制 PLL 和 CLKOUT 时钟在低功耗 STOP 模式下的操作。D1 位用于决定 VREG 的等待模式。
- (3) 低功耗中断控制寄存器 (LPICR): 为 8 位寄存器, D7 (ENBSTOP) 位用于使能或禁止低功耗 STOP 模式, D6~D4 (XLPM_IPL) 位域设置退出低功耗模式所需的中断优先级。
- (4) 外设电源管理清除寄存器 (PPMRC): 8 位寄存器, 若向 PPMRC 寄存器写入 0~63 之间的数据值将使某些位清 0, 使能相应模块时钟; 若写入 64~127 之间任一数据值将使 D6~D0 位全部清 0, 以使能所有模块时钟。
- (5) 外设电源管理设置寄存器 (PPMRS): 8 位寄存器, 若向 PPMRS 寄存器写入 0~63 之间的数据值将使某些位置 1, 以禁止相应模块时钟; 若写入 64~127 之间任一数据值将使 D6~D0 位全部置 1, 以禁止所有模块时钟。
- (6) IPS 总线超时监视器 (IPSBMT): 32 位寄存器, D31~D4 未定义, D3 (BME) 用于使能或禁止总线超时监视器, 1: 使能, 0: 禁止。D2~D0 定义总线监视器超时时间, 000~111 分别代表 2^{10} ~ 2^3 系统总线时钟周期。

13.6 复位控制模块

复位控制模块有两个对外引脚: 复位输入引脚 ($\overline{RSTI}$) 和复位输出引脚 ($\overline{RSTO}$)。当 $\overline{RSTI}$ 持续 4 个 CLKOUT 时钟上升沿为低电平后, 将产生外部复位请求信号。当内部复位控制器模块检测到该信号后, 随即复位芯片, 复位过程中, $\overline{RSTO}$ 引脚为低电平, 复位结束后, $\overline{RSTO}$ 引脚恢复为高电平。

内部复位控制器能够检测 7 种系统复位源 (如图 13-3): 上电复位 (POR)、 $\overline{RSTI}$ 外部复位 (包括非 STOP 模式和 STOP 模式两种)、失锁复位、丢失时钟复位、软件复位以及低电压检测复位。

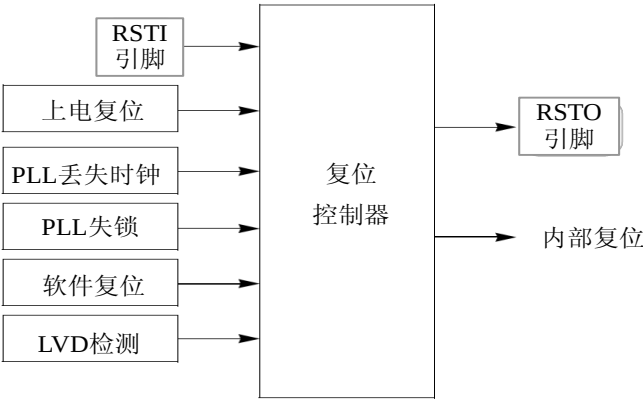


图13-3 复位控制器模块框图

复位控制模块编程涉及两个 8 位寄存器: 复位控制寄存器 (RCR) 和复位状态寄存器 (RSR)

1. 复位控制寄存器

复位控制寄存器 (RCR) 为可读写寄存器, 地址为 0x110000, 复位后值为 0x05。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SOFTRST	FRCRSTOUT		LVDF	LVDIE	LVDRE		LVDE

D7—SOFTRST，控制是否允许软件请求复位。0：不允许；1：允许。

D6—FRCRSTOUT，控制是否允许软件忽略外部复位。0：忽略；1：不忽略。

D5—未定义。

D4—LVDF，低电压检测标志。0：未检测到低电压；1：检测到低电压。

D3—LVDIE，低电压检测中断允许位。0：禁止；1：允许。

D2—LVDRE，低电压检测复位允许位。0：禁止；1：允许。

D1—未定义。

D0—LVDE，控制是否允许低电压检测。0：禁止；1：允许。

2. 复位状态寄存器

复位状态寄存器（RSR）为只读寄存器，地址为 0x110001。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义		LVD	SOFT		POR	EXT	LOC	LOL

D7—未定义。

D6—LVD，低电压检测复位标志。1 表示最近一次复位是低电压检测复位，否则不是。

D5—SOFT，软件复位标志。1 表示最近一次复位是软件复位，否则不是。

D4—未定义。

D3—POR，上电复位标志。1 表示最近一次复位是上电复位，否则不是。

D2—EXT，外部复位标志。1 表示最近一次复位是外部复位，否则不是。

D1—LOC，丢失时钟复位标志。1 表示最近一次复位是丢失时钟复位，否则不是。

D0—LOL，失锁复位标志。1 表示最近一次复位是失锁复位，否则不是。

第14章 基于MCF52223的USB 2.0编程

USB 通信接口具有易于使用、数据传输快速可靠、灵活、成本低和省电等优点，已成为 PC 机与外围设备最主要的数据通信方式。本章从应用角度阐述了 USB 的基本知识、USB2.0 协议基础以及 USB 系统的通用开发方法等内容。有关 USB 协议的详细内容，可访问站点 <http://www.usb.org>。

14.1 USB基本概念及硬件特性

14.1.1 USB概述

1. USB简介

通用串行总线（Universal Serial Bus，USB）是 2000 年以来普遍使用的连接外围设备和计算机的一种新型串行总线标准。与传统计算机接口相比，它克服了对硬件资源独占，限制对计算机资源扩充的缺点，并以较高的数据传输速率和即插即用等优势，逐步发展成为计算机与外设的标准连接方案。现在不但常用的计算机外设如鼠标、键盘、打印机、扫描仪、数码相机、U 盘、移动硬盘等使用 USB 接口，就连数据采集、信息家电、网络产品等领域也越来越多使用 USB 接口。到 2009 年为止，全球大约已有 20 多亿的 USB 设备。图 14-1 给出了通用 USB 标志和 PC 机上的 USB 口。



图14-1 USB标志和PC机的USB接口

USB 接口之所以被广泛应用，主要与 USB 的如下特点密切相关。

（1）支持即插即用（Plug-and-Play）。所谓即插即用，包括两方面的内容，一方面是热插拔，即在不需重启计算机或关闭外设的条件下，便可以实现外设与计算机的连接和断开，而不会损坏计算机和设备；另一方面是可以快速简易安装某硬件设备而无需安装设备驱动程序或重新配置系统。

（2）可以使用总线电源。USB 总线可以向外提供一定功率的电源，其输出电流的最小值为 100mA，最大为 500mA，输出电压为 5V，适合很多嵌入式系统。USB 协议中定义了完备的电源管理方式，用户可以选择是采用设备自供电还是从 USB 总线上获取电源。

（3）硬件接插口标准化、小巧化。USB 协议定义了标准的接插口：A 型和 B 型，这样就为种类繁多的 USB 设备提供了统一的硬件接插口。同时 USB 接口和老式的通信接口相比具有明显的体积优势，为计算机外设的小型化发展提供了可能。

（4）支持多种速度和操作模式。目前 USB 支持 3 种传输速度：低速 1.5Mbps、全速 12Mbps、高速 480Mbps，2009 年新推出的 USB3.0 芯片支持超速 5.0Gbps。同时 USB 还支持 4 种类型的传输模式：块传输、中断传输、同步传输和控制传输，这样可以满足不同外设的功能需求。

2. USB的历史与发展

USB 由 Intel，Compaq，Microsoft，Digital，IBM 以及 Northern Telecom 等公司共同提出。它

的最初用意是取代 PC 机上的众多连接器，同时力图简化通信设备的软件配置。第一台向用户提供了 USB 接口的计算机是 1998 年 5 月 6 日 Apple 公司生产的海蓝色 iMac G3 个人电脑。

从 USB 概念产生至今，其协议版本经过了多次升级更新。USB 协议的标准化由 USB 实施者论坛（USB Implementers Forum，缩写为 USB-IF）负责管理。1996 年 1 月，发表了 USB1.0 版本；1998 年 9 月，公布的 USB1.1 重新修订了 USB1.0，并新增一个新的传输类型（中断传输）；2000 年 4 月，发布了 USB2.0，新增了高速模式；2001 年 12 月 18 日 USB-IF 在 USB2.0 协议的基础上补充了 USB OTG 协议，主要应用于各种不同移动设备间的数据交换。2008 年 8 月，发布了最新的 USB3.0，支持超速模式，传输速度最高可达 5.0Gbps，NEC 公司已于 2009 年推出了带有 USB3.0 模块的芯片 μ PD720200。

目前，使用最为普及的是 USB1.1 和 USB2.0 协议规范。两种协议的比较如表 14-1 所示：

表 14-1 USB 1.1 和 USB 2.0 比较

比较项		USB 1.1	USB 2.0
最大速度		12Mbps	480Mbps
支持设备		低速、全速	低速、全速、高速
输出电流		最大输出电流 250mA，可通过外部接入直流电源来弥补其 USB1.1 版本接口的不足	输出电流达 500mA 以上，可以适合任何 USB 移动存储设备
兼容性		2003 年以前的电脑一般采用 1.1 版本接口	2003 年以后的电脑一般采用 2.0 版本接口
SOF 时间间隔		全速、低速下 $1.00\text{ms} \pm 0.0005\text{ms}$ 间隔发送 SOF	USB2.0 支持 1.1 中的 $1.00\text{ms} \pm 0.0005\text{ms}$ 间隔发送帧。高速模式下，支持微帧即 $125\mu\text{s} \pm 0.0625\mu\text{s}$ 间隔发送 SOF，每个微帧最多可以传输 13 个 512 字节的高速数据包
数据包长度（字节）	控制传输	8、16、32、64	64
	块传输	8、16、32、64	512
	中断传输	1~64	1024
	同步传输	1024	1024

【注】：表格中出现的一些 USB 的概念和术语将在随后章节详细阐述。

3. USB的典型连接

在普通用户看来，USB 系统就是外设通过一根 USB 电缆和 PC 机连接起来。通常把外设称为 USB 设备，把其所连接的 PC 机称为 USB 主机。一个 USB 系统中只能有一个主机。主机内设置了一个根集线器，提供了外设在主机上的初始附着点。包括根集线器上的一个 USB 端口在内，最多可以级联 127 个 USB 设备，层次最多 7 层。在 USB 系统中，将指向 USB 主机的数据传输方向称为上行通信，把指向 USB 设备的数据传输方向称为下行通信。一个典型的 USB 连接，如图 14-2 所示。

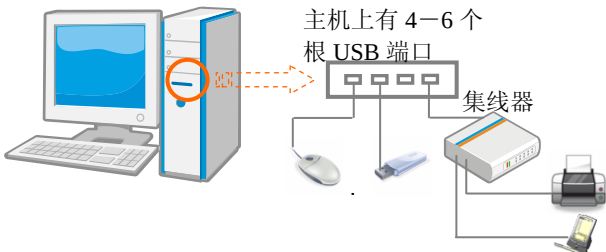


图14-2 USB的典型连接

14.1.2 USB 相关基本概念

1. USB 主机

USB 主机指的是包含 USB 主控制器，并且能够控制完成主机和 USB 设备之间数据传输的设备。广义地说，USB 主机包括计算机和具有 USB 主控芯片的设备。USB 的所有数据通信（不论是上行通信还是下行通信）都由 USB 主机发起，所以 USB 主机在整个数据传输过程中占据着主导地位。从开发人员的角度看，USB 主机可分为三个不同的功能模块：客户软件、USB 系统软件和 USB 总线接口。

1) 客户软件

客户软件负责和 USB 设备的功能单元进行通信，以实现其特定功能。一般由开发人员自行开发。客户软件不能直接访问 USB 设备，其与 USB 设备功能单元的通信必须经过 USB 系统软件和 USB 总线接口模块才能实现。客户软件一般包括 USB 设备驱动程序和界面应用程序两部分。

USB 设备驱动程序负责和 USB 系统软件进行通信。通常，它向 USB 总线驱动程序发出 I/O 请求包（IRP）以启动一次 USB 数据传输。此外，根据数据传输的方向，它还应提供一个数据缓冲区以存储这些数据。

界面应用程序负责和 USB 设备驱动程序进行通信，以控制 USB 设备。它是最上层的软件，只能看到向 USB 设备发送的原始数据和从 USB 设备接收的最终数据。

2) USB 系统软件

USB 系统软件负责和 USB 逻辑设备进行配置通信，并管理客户软件启动的数据传输。USB 逻辑设备是程序员与 USB 设备打交道的部分。USB 系统软件一般包括 USB 总线驱动程序和 USB 主控制器驱动程序这两部分。这些软件通常由操作系统提供，一般开发人员不必掌握。

3) USB 总线接口

USB 总线接口包括主控制器和根集线器两部分。根集线器为 USB 系统提供连接起点，用于给 USB 系统提供一个或多个连接点（端口）。主控制器负责完成主机和 USB 设备之间数据的实际传输，包括对传输的数据进行串行编解码、差错控制等。该部分与 USB 系统软件的接口依赖于主控制器的硬件实现，一般开发人员不必掌握。

2. USB 设备

USB 协议中将 USB 设备定义为具有某种功能的逻辑或物理实体。在最底层，设备指一个独立的硬件部件；在较高层设备可以是表现出一定功能的硬件部件的集合，如一个 USB 接口设备；在更高层次上，设备是指连接到 USB 总线上的那个实体所具有的功能，如一个数据/传真调制解调器。总之，设备的含义可以是物理的、电气的、可寻址的或逻辑的。

USB 对一些具有相似特点并提供相似功能的设备进行抽象，进而将 USB 设备分成多种标准类，包括音频、通信、人机接口设备 HID、显示、海量存储、电源、打印、集线器设备类等。设备类驱动程序通常由操作系统提供，开发人员可以直接使用，不必自己编写。设备描述符和接口描述符中的类代码、子类代码及协议代码指定了 USB 设备或其接口所属的设备类及相关信息，并定位合适的设备类驱动程序。表 14-2 给出了部分标准设备类的定义方法。想要获得全部的标准

USB 设备类，请查阅 USB 实施者论坛上的类别规范文件。

表 14-2 标准的 USB 设备类举例

编号	类名称	设备描述符 bDeviceClass 字段的值	设备描述符 bDeviceSubClass 字段的值	接口描述符 bInterfaceClass 字段的值	接口描述符 bInterfaceSubClass 字段的值
1	人机接口设备（HID）类	0x00	0x00	0x03	任意
2	打印机类	0x00	0x00	0x07	任意
3	大容量存储类	0x00	0x00	0x08	任意
4	集线器（Hub）类	0x09	任意	0x09	任意
5	芯片/智能卡类	0x00	0x00	0x0B	任意
6	厂商定义类	0xFF	0xFF	0xFF	0xFF

上述设备类中的集线器类，主要用于为 USB 系统提供额外的连接点，使得一个 USB 端口可以扩展连接多个设备，因此可单独作为一类，典型集线器的逻辑图和实物图，如图 14-3 所示；而其余设备类，由于它们一般可以设计为具有特定功能的独立的外部设备，用于扩展主机功能，所以统称为 USB 功能设备类。一般说 USB 设备，就是指 USB 功能设备。

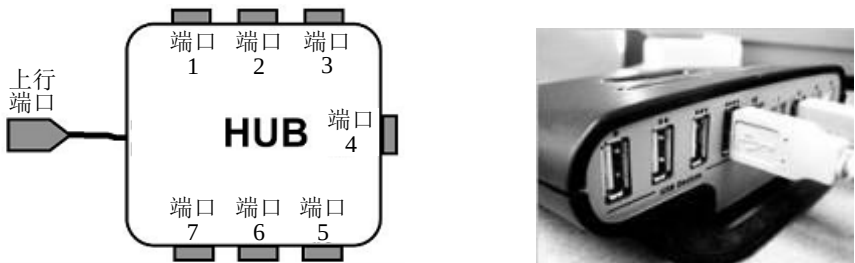


图14-3 典型集线器的逻辑图和实物图

3. USB设备的描述符

可以认为 USB 设备是由一些配置、接口和端点等组件构成的，即一个 USB 设备可以含有一个或多个配置，在每个配置中可含有一个或多个接口，在每个接口中可含有若干个端点。其中，配置和接口是对 USB 设备功能的抽象，实际的数据传输是由端点来完成。

USB 设备使用各种描述符来说明整个设备或设备中某个组件的信息。描述符是一种数据结构，通常被保存在 USB 设备的固件程序中，使主机了解设备的格式化信息。描述符包括以下几种类型：设备描述符、配置描述符、接口描述符和端点描述符，这 4 种描述符是必须具有的；而其他的描述符，如字符串描述符、设备限定描述符以及其他速率配置描述符则是可选的。

下面通过举例，说明各描述符的结构和各字段含义。

1) 设备描述符

设备描述符用于说明设备的总体信息，包括描述设备的类型、设备支持的协议类型、供应商 ID（VID）、产品 ID（PID）、设备版本号、供应商名称、产品名称及设备所支持的配置数等，目的是让主机获取插入的 USB 设备的属性，以便加载合适的驱动程序。一个 USB 设备只能有一个设备描述符，固定为 18 字节的长度，它是主机向设备请求的第一个描述符。

```
uint8 usb_device_descriptor[18] = { //设备描述符
0x12,      //bLength域，描述符的长度：18字节
```



```

0x01,      //bDescriptorType域, 描述符类型: 0x01表示本描述符为设备描述符)
0x00, 0x02, //bcdUSB域, USB规范版本号 (采用BCD码): 2.0
0x00,      //bDeviceClass域, 设备类代码
0x00,      //bDeviceSubClass域, 设备子类代码
0x00,      //bDeviceProtocol域, 设备协议代码 (0x00表示不使用任何设备类协议)
0x40,      //bMaxPacketSize0域, 端点0支持最大数据包的长度: 64字节
0x25, 0x04, //idVendor域, 供应商ID (VID)
0x0e, 0x05, //idProduct域, 产品ID (PID)
0x03, 0x00, //bcdDevice域, 设备版本号 (采用BCD码)
0x01,      //iManufacturer域, 供应商的字符串描述符索引: 1
0x02,      //iProduct域, 产品的字符串描述符索引: 2
0x03,      //iSerialNumber域, 设备序号的字符串描述符索引: 3
0x01      //bNumConfigurations域, 该USB设备支持的配置数目: 1个
};

```

USB 设备的类型 (如 HID, 打印机类, 大容量存储类等) 由 bDeviceClass 域指定, 须按照表 14-2 设置。

2) 配置描述符

配置描述符用于说明 USB 设备中各个配置的特性, 包括配置信息的总长度、配置所支持接口的个数、配置值、配置的属性及设备可以从总线提取的最大电流等。一个 USB 设备可以包含一个或多个配置 (如 USB 设备的低速模式和高速模式可分别对应一个配置)。每一个配置都对应一个配置描述符, 长度固定为 9 字节。

```

uint8 Configuration_Descriptor[9] = { //配置描述符
0x09,      //bLength域, 配置描述符的长度: 9字节
0x02,      //bDescriptorType域: 0x02表示本描述符为配置描述符
0x20, 0x00, //wTotalLength域, 配置信息的总长度 (包括配置、接口和端点): 32字节
0x01,      //bNumInterfaces域, 该配置所支持的接口数 (至少一个): 1
0x01,      //bConfigurationValue域, 配置值: 1
0x04,      //iConfiguration域, 配置字符串描述符索引: 4
0x80,      //bmAttributes域, 配置的属性 (具有总线供电、自供电及过程唤醒的特性)
//位7: 1-必须为1, 位6: 0-自供电, 位5: 0-不支持远程唤醒
0x32      //MaxPower域, 设备从总线提取的最大电流以2mA为单位: 50*2mA=100mA
};

```

在使用 USB 设备前, 主机必须为其选择一个合适的配置。主机根据设备描述符所支持的配置数按顺序查找所有的配置描述符, 继而查找接口描述符以及端点描述符, 直到查找到主机所支持的配置。

3) 接口描述符

接口描述符用于说明 USB 设备中各个接口的特性, 包括接口号、接口使用的端点数、所属的设备类及其子类等。一个配置可以包含一个或多个接口, 每个接口都必须有一个接口描述符。如对一个光驱来说, 当用于文件传输时使用其大容量存储接口; 而当用于播放 CD 时, 使用其音频接口。接口是端点的集合, 可以包含一个或多个可替换设置, 用户能够在 USB 处于配置状态时, 改变当前接口所含的个数和特性。

```

uint8 interface_descriptor[9] = { //接口描述符
0x09, //bLength域, 接口描述符长度: 9字节
0x04, //bDescriptorType域: 0x04表示本描述符为接口描述符

```

```

0x00, //bInterfaceNumber域, 接口号
0x00, //bAlternateSetting域, 接口的可替换设置值
0x02, //bNumEndpoints域, 接口使用的端点数(除端点0): 2
0xff, //bInterfaceClass域, 接口所属的USB设备类: 0xFF表示供应商自定义
0xff, //bInterfaceSubClass域, 接口所属的USB设备子类: 0xFF表示供应商自定义
0xff, //bInterfaceProtocol域, 接口采用的USB设备类协议: 0xFF表示供应商自定义
0x05 //iInterface域, 接口字符串描述符的索引: 5
};

```

4) 端点描述符

所有的传输都是传送到设备端点(endpoint), 或是从设备端点发出。这种端点实际上就是一个能够存储多个字节的缓冲器。端点通常是一个数据存储区, 或是控制器芯片中的一个寄存器, 端点所存储的数据可能是收到的数据, 或是等待发送的数据。端点所支持的传输类型和传输方向等信息, 都在端点描述符中定义。每个端点的描述符总是作为配置描述符的一部分返回, 端点 0 无描述符。端点描述符长度为 7 字节。

```

uint8 endpoint_descriptor2[7] = { //端点2描述符
0x07, //bLength域, 端点描述符长度: 7字节
0x05, //bDescriptorType域: 0x05表示本描述符为端点描述符
0x82, //bEndpointAddress域, 端点号和传输方向: 端点2、IN
0x02, //bmAttributes域, 端点特性: 数据端点、块传输
0x00, 0x02, //wMaxPacketSize域, 端点支持最大数据包长度: 512字节
0x00 //bInterval域, 轮询间隔, 以ms为单位。
};

uint8 endpoint_descriptor3[7] = { //端点3描述符
0x07, //bLength域, 端点描述符长度: 7字节
0x05, //bDescriptorType域: 0x05表示本描述符为端点描述符
0x03, //bEndpointAddress域, 端点号和传输方向: 端点3、OUT
0x02, //bmAttributes域, 端点特性: 数据端点、块传输
0x00, 0x02, //wMaxPacketSize域, 端点支持最大数据包长度: 512字节
0x00 //bInterval域, 轮询间隔, 以ms为单位。
};

```

主机针对收到的数据和要发出的数据也有缓冲器, 但是主机并没有端点, 而是与设备各个端点进行通信的出发点。在 USB2.0 协议中, 设备端点被定义为“USB 设备中的唯一可寻址的部分, 是主机和设备之间通信流的来源和去向”。这就说明了, 一个设备端点只能在单方向上传输数据。每个端点的地址是由端点号和方向组成。其中, 方向是基于主机角度定义的: IN 表示发送数据到主机, OUT 表示主机发送数据。主机在对 USB 设备配置时, 也会分配给相应的逻辑设备一个唯一的地址。由设备地址、端点号和端点方向就可以唯一指定一个端点。

端点的传输特性还决定了其与主机通信时所采用的传输类型, 如控制端点只能使用控制传输。根据端点的不同用途, 可将端点分为两类: 0 号端点和非 0 号端点。0 号端点比较特殊, 它有数据输入 IN 和数据输出 OUT 两个物理单元, 且只能支持控制传输。所有的 USB 设备都必须含有 0 号端点, 用作缺省控制管道。USB 系统软件就是使用该管道和 USB 逻辑设备进行配置通信的。0 号端点在 USB 设备上电以后就可以使用, 而非 0 号端点必须要在配置以后才可以使用。根据具体应用的需要, USB 设备还可以含有多个除 0 号端点以外的其他端点。对于低速设备, 其附加的端点数最多为 4 个, 端点号范围是 0~3; 对于全速/高速设备, 其附加的端点数最多为 16 个, 端点号范围是 0~15。

5) 字符串描述符

在 USB 设备中通常还含有字符串描述符，以说明一些专用信息，如制造商的名称、设备的序列号等。它的内容以 UNICODE 的形式给出，且可以被客户软件所读取。对 USB 设备来说，字符串描述符是可选的。

4. USB通信管道

USB 数据是通过管道传输的，在传输发生之前，主机和设备之间必须先建立一个管道(pipe)。USB 管道并不是一个实际对象，它只是设备端点和主机控制器软件之间的关联，代表了一种在两者之间移动数据的能力。主机是在设备刚刚连接后，在设备请求配置信息时建立管道的。设备被配置后，端点就可以使用了，管道也就存在了。如果设备从总线上移除，主机也就撤销这个不再使用的管道。每个设备都拥有一个使用端点 0 的默认控制管道。端点和各自的管道在每个方向上按照 0~15 编号，因此一个设备最多有 32 个活动管道，即 16 个 IN 管道和 16 个 OUT 管道。主机和设备之间的通信管道示意图如图 14-4。

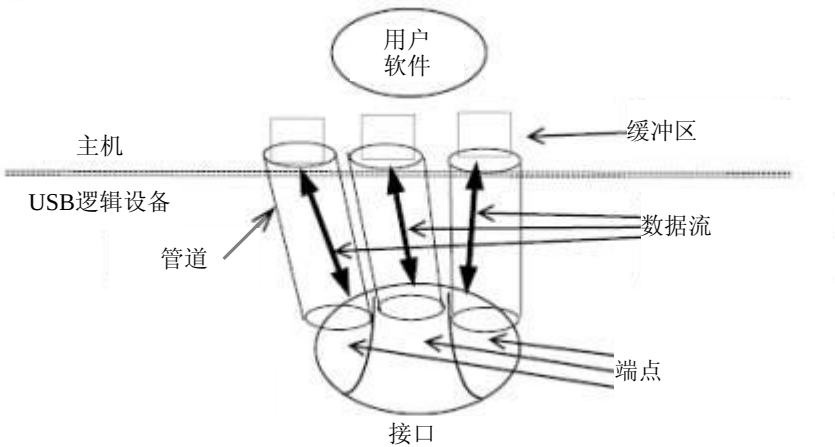


图14-4 管道

主机应用程序向 USB 设备传输数据，可总结为以下几步：首先，主机应用程序把要传输的数据放入数据缓冲区中并向 USB 总线驱动程序发送 I/O 请求包，请求数据传输。然后，USB 总线驱动程序响应主机应用程序的请求并将数据转化为一个或多个 USB 事务处理，并向下传递。接着，USB 主控制器驱动程序为这些事务建立一系列事务处理列表，由 USB 主控制器读取事务处理列表，并将其中的事务处理以信息包为单位发到 USB 总线上。最后，USB 设备收到信息包，由串行接口引擎（Serial Interface Engine, SIE）解包并将数据放入指定端点的接收缓冲区内，由芯片内的固件程序对其进行处理。

5. USB应用分类

根据 USB 模块所扮演的角色，可以将 USB 应用系统大致分为如下三类：

- (1) 待开发的 USB 设备作为从机，PC 作为主机；
- (2) 待开发的 USB 设备作为主机，其他设备作为从机；
- (3) 待开发的 USB 设备可以根据需要在主机身份和从机两种角色之间进行切换（OTG 技术的应用）的 USB 系统。

14.1.3 USB的物理特性

1. USB电缆和连接器

USB 使用一根屏蔽的 4 线电缆与其它设备进行互联。USB 协议规定，高速和全速传输需要使用外壳屏蔽、数据线双绞的 USB 电缆，而低速传输则没有这些要求。数据传输通过一对差分信号线进行，这两根线分别标为 D+ 和 D-，另外两根线是 V_{BUS} 和 GND，其中 V_{BUS} 向 USB 设备供电。使用 USB 电源的设备称为总线供电设备，而使用自己外部电源的设备叫做自供电设备。为了避免混淆，USB 电缆中的线都用不同的颜色标记，如图 14-5 所示。

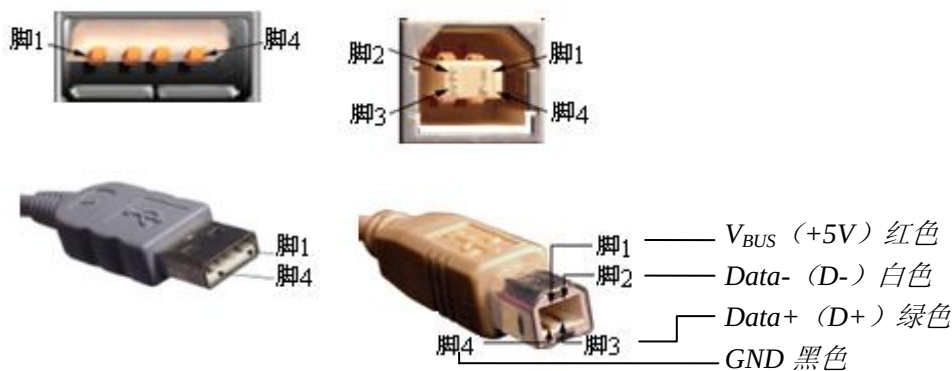


图14-5 USB连接器上的信号与颜色

USB 插头的 V_{BUS} 和 GND 线，要比其他两根线长，以保证在电缆连接时其电源线先接入，然后再接入信号线。

2. USB通信的差分信号

数据在 USB 总线上实际传输时，使用的是 NRZI（反向不归零）编码的差分信号，这种信号有利于保证数据的完整性和消除噪声干扰。

我们知道，传统的传输方式大多使用“正/负信号”技术，即用“正信号”或者“负信号”二进制表达机制，这些信号使用单线传输。用不同的信号电平范围来分别表示 1 和 0，它们之间有一个临界值，如果在数据传输过程中受到中低强度的干扰，高低电平不会突破临界值，那么信号传输可以正常进行。但如果遇到强干扰，高低电平突破临界值，由此将造成数据传输错误。一般说来，总线频率越高，线路间的电磁干扰就越厉害，数据传输失败的发生机率也就越高。因此这种信号表达技术无法应用于高速总线传输，而差分信号技术能有效克服这种缺点。

差分信号技术最大的特点是：必须使用两条线路才能表达一个比特位，用两条线路传输信号的压差作为判断 1 还是 0 的依据。这种做法的优点是具有极强的抗干扰性。倘若遭受外界强干扰，两条线路对应的电平会出现同样的大幅度提升或降低，但二者的电平改变方向和幅度几乎相同，电压差值就可始终保持相对稳定，因此数据的准确性并不会因干扰噪声而有所降低。当然，由于 1 个比特位需要两条线路，在总线宽度相等的条件下，差分技术需要的信号线条数就是“正/负信号”技术所需的信号线条数的两倍。

3. USB总线上的状态与设备速度模式检测

协议中，给 USB 总线定义了 4 种状态：SE0、SE1、J 和 K 状态。两根数据线都被拉低时的状态定义为 SE0 状态；两根数据线都被拉高时定义为 SE1 状态（该状态是非法状态）。当有设备连接到主机，使 D+或 D-被上拉，被上拉的线为高电平而另一根数据线的低电平时，这种状态称为 J 状态（该状态为空闲状态），包被传送之前和之后，数据线上就是该状态。而 K 状态是两根数据线上的极性都与 J 状态相反的状态（如 J 状态为 D+上拉，D-下拉，则 K 状态指 D+下拉，D-上拉），主机通过 K 和 J 状态测试设备是否支持高速通信。

当主控制器或集线器的下行端口上没有 USB 设备连接时，USB 总线处于 SE0 状态。当有设备连接以后，如图 14-6 所示，电流流过由集线器的下拉电阻和设备在 D+/D-的上拉电阻构成的分压器。由于下拉电阻的阻值是 $15\text{k}\Omega$ ，上拉电阻的阻值是 $1.5\text{k}\Omega$ ，所以在 D+/D-线上会出现大小为 $(V_{CC} \cdot 15 / (15 + 1.5))$ 的直流高电平电压。如果主机检测到 D+线上为高电压，说明连接上的是高速/全速设备；若检测到 D-线上为高电压，说明连接上的是低速设备。



【注】：（1）图中两个灰色上拉电阻的阻值均为 $1.5\text{k}\Omega$ ；两个白色下拉电阻的阻值均为 $15\text{k}\Omega$ 。
（2） V_{BUS} 为+5V， V_{CC} 为+3.3V。

图14-6 USB设备电缆和电阻的连接

高速设备在连接起始时需要以全速速率与主机进行通信，以完成其配置操作。在复位（主机将 D+和 D-都拉低，使数据线处于 SE0 状态，并保持至少 10ms。在 2.5us 后设备将识别复位条件。该复位要与微处理器的上电复位区别开来，它是 USB 协议复位，是为了使设备的 USB 信号开始时是一个已知状态）期间，支持高速的设备发送一个 K 状态（这是由集成在 USB 设备接口芯片的内部的软件控制开关完成的），具有高速能力的 HUB 检测到总线上的该状态并响应一个 K 和 J 的交替状态序列。以后设备就将全速提高为高速通信。如果 HUB 不响应设备的 K 状态，那么设备就用全速通信。

14.2 USB的通信协议

USB 事务处理是主机和设备之间数据传输的基本单位，由一系列具有特定格式的包组成。因此，要了解完整的 USB 通信协议，必须从 USB 的信息传输单元包及其数据域谈起。通过由下而上，从简单至复杂的通信协议单位组成各种复杂的通信协议，进而构建出完整的通信协议。

14.2.1 USB基本通信单元：包

USB 协议中，包（Packet）是 USB 系统中信息传输的基本单元，所有数据都是经过打包后在

总线上传输的。包由一些字段构成，不同功能的字段，按照特定的格式组合可以构成不同的包。组成包的字段主要有：同步字段、包标识符字段、地址字段、端点字段、帧号字段、数据字段、循环冗余校验字段和包结尾字段。包的一般格式为：

同步字段 (SYNC)	PID 字段	数据字段	CRC 字段	包结尾字段 (EOP)
-------------	--------	------	--------	-------------

在 USB 的数据传输中，所有的传输包都起始于 SYNC，接着是 PID，后面是包中所包含的数据信息，接下来是用来检测包中数据错误的循环冗余校验信息，最后以包结尾作为结束标志。下面将一一介绍这些字段。

1) 同步字段 (SYNC)

同步字段 (SYNC) 由 8 位组成，作为每个包的前导。顾名思义，它是起同步作用的，目的是使 USB 设备与总线的包传输率同步，它的数值固定为 00000001B。

2) 包标识符字段 (PID)

包标识符字段 (PID) 紧随在 SYNC 字段后面，用来表示包的类型。在 USB 协议中，根据 PID 的不同，USB 包具有不同的类型、格式以及含义。如下所示：

D7	D6	D5	D4	D3	D2	D1	D0
$\overline{\text{PID3}}$	$\overline{\text{PID2}}$	$\overline{\text{PID1}}$	$\overline{\text{PID0}}$	PID3	PID2	PID1	PID0

PID 长度为一个字节 (8 个数据位)，由 4 个位的包类型字段和 4 个位的校验字段构成。PID 是 USB 包类型的唯一标志，USB 主机和 USB 设备在接收到包后，首先必须对包标识符解码得到包的类型再决定下一步动作。PID 中的校验字段是通过将类型字段的每位取反产生的，它是用来对包类型字段进行错误检测用的，旨在保证对包的标识符译码的可靠性，如果 4 个检验位不是它们各自的类型位的反码，则说明标识符中的信息有错误。

表 14-3 中列出了包的类型，包括令牌、数据、握手或特殊四种包类型。

表 14-3 各种包的类型与规范

包类型	PID[3:0]	含义	格式				
令牌包	OUT	0001B	主机向设备发送数据				
	IN	1001B	主机向设备要数据				
	SETUP	1101B	主机到设备，用于控制传输				
	SOF	0101B	帧的起始标记，标志着一个时间片的开始，表明接下来安排数据传输				
数据包	DATA0	0011B	第偶数次发送的包				
	DATA1	1011B	第奇数次发送的包				
握手包	ACK	0010B	接收器收到无错误的包				
	NAK	1010B	接收器无法接收数据或发射器无法送出数据				
	STALL	1110B	端点产生停滞的状况				
特殊包	PRE	1100B	使能下游端口的 USB 总线的数据传输切换到低速的设备				

【注】：(1) 该表中的包格式只列出了包的中间一部分字段，前面还需加上同步字段和包结尾字段，才是完整的包格式。

(2) 该表中出现了一些术语和概念，由于篇幅所限，现在不能展开阐述，将在后文给以详细说明。

在 USB 系统中，只有主机才能发出令牌包。令牌包定义了数据传输的类型。令牌包中较为重要的是 SETUP、IN 和 OUT 这三种令牌包，它们用来在根集线器和设备端点之间建立数据传输。一个 IN 包用来建立一个从设备到根集线器的数据传送，一个 OUT 包用来建立从根集线器到设备的数据传输。IN 包和 OUT 包可以对任何设备上的任何端点寻址。一个 SETUP 包是一个 OUT 包的特殊情形，它是“高优先级的”，也就是说设备必须接受它，即使设备正在进行数据传输操作的过程中也要对其进行响应。SETUP 包总是指向端点 0 的。

SOF 令牌包包含 SOF 标记、帧序列号及 CRC5 校验码。在低速、全速模式下，主机每间隔 1ms（这个 1ms 称为一帧，允许误差 0.005ms）发送一个帧开始包 SOF（Start of Frame）。有时也可把帧理解为这个 1ms 时间段内传输的信息。在高速模式下，主机每间隔 125 μ s（即为一微帧，允许误差 0.0625 μ s）发送一个帧开始包 SOF。包括集线器的所有功能部件都可以收到 SOF 包，SOF 标记不会使得接收功能部件产生返回包，所以不能保证主机发送的 SOF 包都能被功能部件接收到。事实上，每两个 SOF 包之间的时间段为事务处理（Transaction）的时间（下面将具体阐述事物处理的概念），所以一次事务处理的时间是有限制的。

数据包有两种类型，根据不同的 PID，数据包分为 DATA0 和 DATA1，这样做是为了支持数据切换同步。主机总是通过配置事件初始化总线传送的首个数据包为 DATA0，下一个数据包使用 DATA1，并且以后的数据传送轮流切换，即 DATA0 $\rightarrow$ DATA1 $\rightarrow$ DATA0 $\rightarrow$ DATA1……。

3) 地址字段 (ADDR)

设备地址由 7 位组成，共有 128 个地址值。地址 0 作为缺省地址，不能分配给 USB 设备，因此只有 127 个可分配的地址值。在 USB 设备上电的时候，主机用缺省地址 0 与设备通信。当 USB 上电配置完成后，主机重新为 USB 设备分配一个 USB 地址。

4) 端点字段 (ENDP)

端点字段由 4 位组成，可寻址 16 个端点。ENDP 字段仅用在 IN、OUT 与 SETUP 令牌包中。对于低速设备可支持端点 0 以及端点 1 作为中断传输模式，而全速和高速设备则可以包含全部的 16 个端点。

5) 帧序列号字段

帧序列号长度为 11 位，从 0 开始，每发送一个 SOF，其值自动加 1，最大数值为 0x7FF（十进制值 2047），当超过最大值时自动从 0 开始循环。

6) 数据字段

数据字段的长度最大为 1024 字节，在数据传输的时候，首先传输低字节，然后传输高字节。对于每一个字节，先传输字节的低位，再传输字节的高位。实际的数据字段长度，需根据 USB 设备的传输速度（低速、全速或高速）以及传输类型（中断传输、批量传输、实传输）而定。

7) 循环冗余校验字段 (CRC)

循环冗余校验一般在发送方的位填充操作之前进行，这样可以检验包的错误，保证传输的可

靠性。在令牌包中，一般采用 5 位循环冗余校验；在数据包中，采用 16 位循环冗余校验。

8) 包结尾字段 (EOP)

包结束 EOP (End of Packet) 字段，在物理上表现为差分线路的两根数据线保持 2 比特低位时间和 1 比特空闲位时间。USB 主机根据 EOP 判断数据包的结束。

14.2.2 USB通信中的事务处理

USB 协议将事务 (Transaction) 定义为“将一个服务传送到一个端点”。这里的服务是指主机传送信息给设备，或者主机要求从设备获得信息，每个事务包含识别用途、错误检查、控制的信息以及要交换的数据。需注意的是，事务处理是不能被中断的。一个完整的事务处理包括令牌阶段、数据阶段和握手阶段 3 部分，其中令牌阶段是必须的。下面将详细阐述较有代表性的 SETUP 事务、IN 事务处理、OUT 事务处理过程。

1) SETUP事务处理

SETUP 事务处理是一种特殊的 USB 事务处理，它只在 USB 控制传输中使用，用于对 USB 设备进行配置。SETUP 的数据传输方向是主机到设备。一个完整的成功的 SETUP 事务处理流程如下所示：

1. 主机→设备 (令牌包)	SYNC	SETUP	ADDR	ENDP	CRC5	EOP
2. 主机→设备 (数据包)	SYNC	DATA0	数据		CRC16	EOP
3. 主机←设备 (握手包)	SYNC	ACK	EOP			

在实际传输过程中，难免出现各种错误，如果接收到的 SETUP 令牌包有错误，设备将忽略该包，不做任何响应。

2) IN事务处理

IN 事务用于实现 USB 主机向 USB 设备要数据，一个完整的成功的 IN 事务处理流程如下所示：

1. 主机→设备 (令牌包)	SYNC	IN	ADDR	ENDP	CRC5	EOP
2. 主机←设备 (数据包)	SYNC	DATA0/DATA1	数据		CRC16	EOP
3. 主机→设备 (握手包)	SYNC	ACK	EOP			

在实际数据传输过程中，可能会出现错误，设备和主机会根据具体情况，采用不同的处理方法。

情况 1：对于 USB 设备，当主机向设备发送的 IN 令牌包在传输过程中被损坏时，设备接收不到正确的 IN 令牌包，就会忽略该令牌包，不进行应答；

情况 2：对于 USB 设备，如果设备接收到正确的 IN 令牌包，但设备的 IN 端点被停止，无法向主机发送数据，此时设备向主机发送 STALL 握手包；

情况 3：对于 USB 设备，如果设备接收到正确的 IN 令牌包，但设备由于某种原因无法向主机提供数据，此时设备向主机发送 NAK 握手包。

情况 4：对于 USB 主机，当 USB 设备向主机发送的数据包在传输过程中被损坏，主机接收不到正确的数据包，主机将忽略该出错的数据包，不做响应。

3) OUT 事务处理

OUT 事务用于实现 USB 主机到设备的数据传输，一个完整的成功的 OUT 事务处理流程如下所示：

1. 主机→设备（令牌包）	SYNC	OUT	ADDR	ENDP	CRC5	EOP
2. 主机→设备（数据包）	SYNC	DATA0/DATA1	数据		CRC16	EOP
3. 主机←设备（握手包）	SYNC	ACK	EOP			

如果主机在向设备传送数据的过程中发生错误，设备会有如下响应：

情况 1：当 OUT 令牌包在传输过程中被损坏，设备接收不到正确的令牌包，设备会忽略该令牌包，不对其进行应答；

情况 2：如果设备接收到正确的 OUT 令牌包，但数据包在传输过程中被损坏，设备将忽略该数据包，不对其应答；

情况 3：如果设备接收到正确的 OUT 令牌包，但是设备的 OUT 端点被停止，无法接收主机发来的数据，那么设备向主机发送 STALL 握手包；

情况 4：如果设备接收到正确的 OUT 令牌包，但是设备由于某种原因无法接收主机发送的数据，那么设备将向主机发送 NAK 握手包；

情况 5：如果设备的数据触发位和接收到的数据包的触发位不一致，那么设备将丢弃该数据包，然后向主机发送 ACK 握手包。

14.2.3 从设备的枚举看 USB 数据传输

1. 控制传输

在 USB 传输中，制定了 4 种传输类型：批量传输、中断传输、实时传输以及控制传输。批量传输用于传输大量数据，要求传输不能出错，但对时间没有要求，适用于打印机、存储设备等。中断传输总是用于对设备的查询，以确定是否有数据需要传输。因此中断传输的方向总是从 USB 设备到主机，适用于 USB 鼠标、键盘等设备。实时传输要求数据以固定速率抵达或在指定时刻抵达，可以容忍偶尔错误的数据，一般用于麦克风、喇叭等设备。控制传输主要用于传输少量的数据，对传输时间和传输速率没有要求，它是 USB 传输中最重要的传输，唯有正确地执行完控制传输，才能进一步正确地执行其他传输模式，所以这里只详细介绍控制传输。

由于每个 USB 设备速度、传输的包的大小等信息有可能不同，因此每个 USB 设备固件内都由一些描述符记录着该设备的一些信息。当在主机上电检测到 USB 设备时，首先经历的是控制传输。在该过程中，主机要读取设备相关描述符，以确定该设备的类型和操作特性；另外，主机还对该设备进行相应的配置。每个 USB 设备都有一个缺省的控制端点，该端点总是 0 号端点。控制传输分为 2~3 个阶段：设置阶段、数据阶段（无数据控制没有此阶段）以及状态阶段，下面介绍各阶段的工作。

阶段 1：设置阶段

USB 设备在正常使用之前，必须先通过端点 0 进行配置。在本阶段，主机将会对 USB 设备的请求信息（如：读设备描述符）传送给 USB 设备，从 USB 设备获取配置信息后确定此设备有哪些功能。作为配置的一部分，主机还会设置设备的配置值。

设置阶段包含了 SETUP 令牌包、紧随其后的 DATA0 数据包（该包里的数据即为设备请求，固定为 8 字节）以及 ACK 握手包。该阶段定义了此次控制传输的内容。

阶段 2：数据传输阶段

数据传输阶段是用来传输主机与设备之间的数据。根据数据传输的方向，控制传输又可分为 3 种类型：控制读取（读取 USB 描述符）、控制写入（配置 USB 设备）以及无数据控制。

控制读取是将数据从设备读到主机，读取的数据包括 USB 设备描述符等内容。该过程先由主机向设备发送 IN 令牌包，表明接下来主机要从设备读取数据；然后设备向主机发送 DATA1 数据包；最后，主机将以下列的方式加以响应：若数据被正确接收，主机发送 ACK 握手包；若主机正在忙碌，则发送 NAK 握手包；若发生了传输错误，主机发送 STALL 握手包。

控制写入则是将数据从主机传到设备上，所传的数据即为对 USB 设备的配置信息，在该过程中，主机将会发送一个 OUT 令牌包，表明主机将要数据发送到设备；紧接着，主机将数据通过 DATA1 数据包传递至设备；最后，设备将以下列方式加以响应：若数据被正确接收，设备发送 ACK 握手包；若设备正在忙碌，则发送 NAK 握手包；若发生了传输错误，设备发送 STALL 握手包。

阶段 3：状态阶段

状态阶段用来表示整个控制传输的过程已经完全结束了。请注意，状态阶段传输的方向必须与数据阶段的方向相反。即：若数据阶段是 IN 令牌包，则状态阶段应为 OUT 令牌包；若数据阶段是 OUT 令牌包，则状态阶段应为 IN 令牌包。

对于控制读取而言，在此阶段，主机会发送 OUT 令牌包，其后再跟 1 个 0 长度的 DATA0 数据包。而此时，设备也会做出相对应的动作，向主机发送 ACK 握手包、NAK 握手包或 STALL 握手包。

相对地，对于控制写入而言，在此阶段，主机会发送 IN 令牌包，然后设备发送 1 个 0 长度的 DATA0 数据包，主机再做出相对应的动作：发送 ACK 握手包、NAK 握手包或 STALL 握手包。

2. 设备请求

在 USB 协议中，主机对 USB 设备的各种配置操作是通过设备请求来实现的。设备请求是在控制传输的 SETUP 阶段由主机发往设备的，通常在默认控制管道上传输，它的各个字段由主机定义，表达了每一次控制传输的目的。例如，主机想获得设备的配置描述符，那么在 SETUP 阶段，主机发往设备的数据包中就会包含 GET_CONFIGURATION（获得配置描述符）请求，在数据阶段将能够收到设备发来的配置描述符。

一个设备请求有 8 个字节，它的一般格式如表 14-4 所示：

表 14-4 数据请求的格式

位移量	字段值	大小	描述
0	bmRequestType	1字节	D7：数据传输方向。0：主机至设备，1：设备至主机 D[6:5]：类型。0：标准，1：类，2：供应商，3：保留 D[4:0]：接收端。0：设备，1：接口，2：端点， 3：其它，4~31-保留
1	bRequest	1字节	特定请求
2	wValue	2字节	字大小字段，根据请求的不同而不同
4	wIndex	2字节	字大小字段，根据请求的不同而不同，通常是传递索引和位移量
6	wLength	2字节	如果有数据阶段，该域表示所要传输的字节大小

bmRequestType 字段决定了特定请求的特征，该域的 D7 位表示在控制传输的数据阶段数据传

输的方向。如果 $wLength=0$ ，表示没有数据阶段，该位可以忽略； $D[6:5]$ 表示了该请求所属的类型，USB 标准中定义了所有的 USB 设备必须支持的一系列的请求，此外，类和供应商也可以定义一些其他的请求； $D[4:0]$ 表示接收端，请求可以针对设备、接口或设备的一个端点，当针对一个接口或端点时， $wIndex$ 字段决定了是哪个接口或端点。

$bRequest$ 字段表示特定请求。如果 $bmRequestType$ 的类型域为 0，结合 $bRequest$ 字段的值就可以确定是某个标准请求。例如： $bmRequestType$ 的类型域为 0， $bRequest$ 域的值分别为 1、5、6、9，分别对应标准请求：Clear_Feature（清除特性）、Set_Address（设置地址）、Get_Descriptor（获取描述符）、Set_Configuration（设置配置）。

$wValue$ 和 $wIndex$ 字段的值根据请求的不同而不同，前者用来传递一个参数给设备，后者用来指定一个接口或端点。

$wLength$ 字段表示控制传输的数据阶段中传输数据的字节大小，如果该域的值 0，表示没有数据阶段。下面结合实例分别介绍几个主要的设备请求。

1) 获取描述符请求

获取描述符请求（Get_Descriptor）用于取得 USB 设备中存在的特定的描述符。根据上述对请求格式的分析可知，获取描述符请求的 $bmRequestType$ 和 $bRequest$ 应分别设为 10000000B 和 06H。 $wValue$ 的高字节表示要取的描述符类型，低字节表示描述符的索引值，描述符的类型有：1 表示设备描述符，2 表示配置描述符，3 表示字符串描述符，4 表示接口描述符，5 表示端点描述符。 $wIndex$ 的值为 0 或字符串描述符的语言 ID； $wLength$ 指定了获取描述符请求要返回的数据长度。获取描述符请求有一次数据阶段，传输的是具体描述符信息。

2) 设置地址请求

设置地址请求（Set_Address）用于给 USB 设备设置地址，从而可以对该 USB 设备进行进一步的访问，该请求无数据阶段。设置地址请求的 $bmRequestType$ 和 $bRequest$ 的值分别为 00000000B 和 05H， $wValue$ 的值为新的设备地址。该请求与其他的请求有一个重要的不同点，即使在该请求下，USB 设备一直不改变它原来的地址，直到该请求的状态阶段被成功地完成，而其他请求的操作都是在状态阶段之前完成。 $wIndex$ 和 $wLength$ 的值一般为 0。若给定的设备地址大于 127，或者 $wIndex$ 或 $wLength$ 为非 0 值，那么该请求不执行。设置地址请求无数据阶段。

3) 设置配置请求

设置配置请求（Set_Configuration）用于为 USB 设备设置一个合适的配置值。该请求的 $bmRequestType$ 和 $bRequest$ 值分别为 00000000B 和 09H， $wValue$ 域的低字节表示设置的配置值，该值为 0 或与配置描述符中的配置值相匹配。如果设置值等于 0，表示设备在地址状态。 $wValue$ 域的高字节保留。 $wIndex$ 和 $wLength$ 的值一般为 0，如果这两个域有一个为非 0 值，那么该请求不执行。设置配置请求无数据阶段。

3. 设备枚举过程

当有 USB 设备连接到主机时，主机自动对设备进行枚举。下面将详细剖析该枚举过程，使读者能够更清楚地理解 USB 总线上的数据传输方式。枚举过程如下：

(1) 用户将一个 USB 设备连接到 USB 端口，或者系统带着一个 USB 设备上电启动。该 USB 端口可能是主机上的根 HUB 也可能是一个连接到主机下行端口上的 HUB。HUB 向主机汇报 HUB 状态改变，且会为端口提供电源，使设备得到稳定的 V_{BUS} ，而进入上电状态。此时该 HUB 端口

是未使能的。

(2) HUB 检测设备。HUB 检测它的每个端口的数据线上的电压。平时两根数据线都被拉低，处于 SE0 状态。当有设备连接上之后，设备会将数据线 D+ 或 D- 上的电压拉高（当连上全速或高速设备，D+ 被拉高，当连上低速设备，D- 被拉高）。这样 HUB 就能检测到有设备接入。检测到设备后，HUB 继续向设备提供电源，但不传输数据。

(3) 主机检测到新设备。HUB 会使用它的中断端点向主机报告，是否有一个端点发生事件。主机得知事件发生，会向 HUB 发送获取端口状态请求 `Get_Port_Status`，以获取更多信息。USB2.0 要求在 HUB 复位设备之前，检测该设备是低速、全速或高速设备。

(4) HUB 复位设备。当主机检测到一个新设备时，主机控制器向 HUB 发送一个获取端口特性请求 `Set_Port_Feature`，使得 HUB 能够复位该端口。HUB 使得设备的 USB 数据线保持复位条件（SE0 状态）至少 10ms。注意：HUB 只向新设备发送复位信号，不影响其它 HUB 和设备。

(5) 主机检测是否全速设备支持高速。在复位期间，支持高速的设备发送一个 K 状态，具有高速能力的 HUB 检测到该状态并响应一个 K 和 J 的交替状态序列。设备就将提高为全速，并且以后都用高速通信。如果 HUB 不响应设备的 K 状态，那么设备就用全速通信。

(6) HUB 为设备建立一条连向 USB 总线的通路。主机重复发送 `Get_Port_Status` 请求，直到设备在返回数据中的某一位向主机表明，设备已经退出复位状态。之后，设备进入默认状态，USB 寄存器也处在复位状态，可以用默认端点 0、默认地址 00H 与主机通信，同时从 V_{BUS} 汲取至少 100mA 的电能。

(7) 主机发送 `Get_Descriptor` 请求，以获得设备描述符（第 1 次），旨在得到端点 0 所支持的最大数据包长度（设备描述符的第 8 个字节）。这是一个 SETUP 事务→IN 事务→OUT 事务过程，OUT 事务一结束，主机要求 HUB 复位设备。

(8) 主机给从机分配地址。主控制器通过发送 `Set_Address` 请求，给设备分配一个唯一地址。设备使用默认地址完成状态阶段后，以后的所有通信都使用新地址。该新地址一直有效，直到设备移除，端口复位或系统重启。

(9) 主机获悉设备功能。主机发送 `Get_Descriptor` 到新地址，获取设备描述符（第 2 次）。这一次，主机将获取完整的设备描述符。此后，主机继续请求在设备描述符中指定的配置描述符。对于配置描述符的请求实际上是对配置描述符本身及其下层的描述符（包括接口描述符、端点描述符）的请求。主机先获得 9 个字节的配置描述符本身，其中包括了配置描述符和其下层的所有描述符的总长度。之后，将再次请求配置描述符，这一次将获得完整的配置、接口和端点描述符。

(10) 主机弹出消息，显示“产品字符串”，然后分配并加载设备驱动，相关内容，请参见本章 14.4.1 中“驱动程序的安装”一节。之后，驱动程序会要求设备重新发送描述符。

(11) 主机设备驱动为设备选择一个合适的配置值。在通过描述符获悉设备状况之后，设备驱动程序发送一个带有所需配置号的 `Set_Configuration` 请求。设备读到请求并使能所要求的配置。这时，设备就处于配置状态，并且设备接口被允许。

4. USB 总线上包的观测方法

在 USB 设备开发过程中，特别是在调试阶段，开发人员可能需要观测 USB 总线上的数据传输情况。这里推荐一款目前使用较多的设备调试工具 Bus Hound，它能够捕获计算机主机和 USB 外围设备之间的数据交换过程，协助开发人员轻松地进行调试工作。例如上述设备的枚举过程，就可用 Bus Hound 捕捉总线上的数据包进行观察。不仅如此，在 USB 设备的调试阶段，也可以用

该工具调试 USB 设备固件程序的运行情况。

下面介绍 Bus Hound 工具的使用方法。在启动 Bus Hound 后，首先选中【Device】标签，在设备资源列表中，选择要观测的设备驱动。再选中【Capture】标签，点击【Run】按钮，这时向主机的 USB 接口插入要观测的 USB 设备，【Capture】标签下的窗口就能显示截获的 USB 包。观察窗口界面如图 14-7 所示。其中，CTL 表示 USB 控制传输，DI 表示 PC 机收到的数据包，DO 表示主机发送的数据。

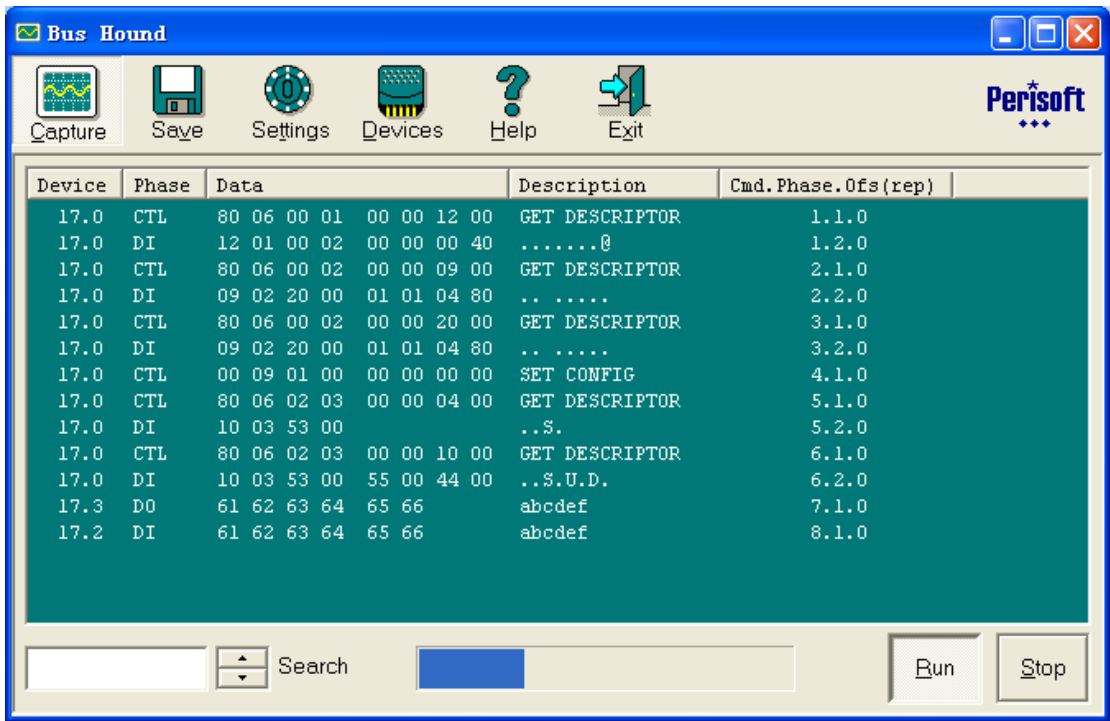


图14-7 Bus Hound截获的USB包

14.3 MCF52223硬件最小系统及USB模块功能简介

14.3.1 MCF52223硬件最小系统

1. MCF52223的主要特性

MCF52223 是基于 V2 ColdFire 的高集成度 32 位微处理器的代表，它主要用于通用目的的工业控制。MCF52223 有 100 脚 LQFP 和 81 脚 MAPBGA 两种封装形式，其主要具有如下功能特性：

- (1) 总线时钟可达 80MHz
- (2) 带有 MAC (Multiply ACcumulate) 乘法单元的 V2 ColdFire 核
- (3) 32K 字节的内部 SRAM
- (4) 256K 字节的片上 Flash 存储器
- (5) 1 个 USB OTG (Universal Serial Bus On-The-Go) 全速/低速主机和设备控制器

- (6) 3 个通用异步接收器/发送器 (Universal Asynchronous Receiver/Transmitter, UART)
- (7) 1 个内部集成电路控制器 (Inter-Integrated Circuit, I2C)
- (8) 1 个 12 位 AD 转换器 (Analog-to-Digital Converter, ADC)
- (9) 1 个实时时钟
- (10) 1 个队列串行外围设备接口模块 (Queued Serial Peripheral Interface, QSPI)
- (11) 1 个 4 通道 32 位直接内存访问控制器 (Direct Memory Access, DMA)
- (12) 4 个带有可选 DMA 功能的 32 位通用目的定时器
- (13) 2 个 16 位可编程中断定时器 PIT
- (14) 1 个可编程软件看门狗定时器
- (15) 可处理最多 63 个中断源的中断控制器
- (16) 带有 8MHz 片上振荡器的时钟模块和集成的锁相环 (PLL)

2. MCF52223 的最小系统

MCF52223 的硬件结构与 MCF52233 类似, 本节只给出 MCF52223 的最小系统设计。MCF52223 芯片的最小系统包括电源电路、PLL 电路、复位电路、晶振电路等, 如图 14-8 所示。

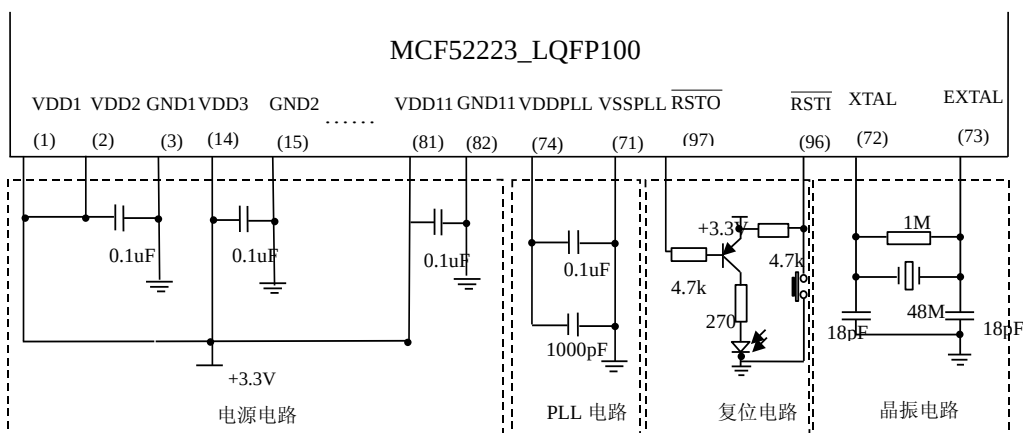


图14-8 MCF52223最小系统

14.3.2 MCF52223的USB模块功能简介

1. 引脚及外围硬件电路

该芯片的 USB 模块连接比较简单, 只要将 USB 模块的两个引脚 USB_DP 和 USB_DM 分别接 33 欧姆的电阻连接到 USB 接口的 D+和 D-即可。

2. USB模块主要特性

MCF52223 有一个 USB 双模式 (主机或从机模式) 控制器, 即: 使用 MCF52223 作为主控制器的 USB 设备既可被定义为单一的主机或从机模式, 也可以在主机和从机模式之间进行切换。该 USB 模块支持 USB2.0 协议中定义的全速和低速模式及 OTG (On-The-Go) 协议, 并通过状态控制寄存器及存储器中的数据结构与 ColdFire 处理器核进行通信。MCF52223 的 USB 模块具有如下

特性：

- (1) 支持 USB1.1 和 2.0 协议的全速设备控制器
- (2) 16 个双向端点
- (3) DMA 或 FIFO 数据流接口
- (4) 低功耗
- (5) 支持 OTG 协议

3. 缓冲区描述符表（BDT）

MCF52223 是通过缓冲区描述符表（Buffer Descriptor Table, BDT）来高效地管理 USB 端点通信的。每一个表项，即缓冲区描述符（Buffer Descriptor, BD）占 8 个字节（64 位），包括 32 位的控制/状态字和 32 位的缓冲区地址。BD 各字段分配如表 14-5 所示。每个端点的方向需要两个 BD（EVEN BD 和 ODD BD）。这样，当微处理器处理其中一个 BD 时，USB 模块就可以处理另一个，这种双缓冲方式能使数据传输达到最大吞吐量。MCF52223 有 16 个双向端点，因此其 BDT 表最多占据系统存储空间 512 字节。

表 14-5 缓冲区描述符（BD）的格式

31: 26	25: 16	15: 8	7	6	5	4	3	2	1: 0
	BC		OWN	DATA 0/1	KEEP / PID[3]	NINC / PID[2]	DTS / PID[1]	BDT_STALL / PID[0]	
缓冲区地址（32位）									

下面介绍表 14-5 中各字段的含义。

BC：接收到的数据字节数。

OWN：决定当前是 USB 模块还是微处理器拥有对缓冲区描述符表的操作权。OWN=1：USB 模块拥有。OWN=0，微处理器拥有。当 KEEP 位为 0 时，在完成了一个令牌后 SIE 向该位写 0。

DATA0/1：发送或接收了 DATA0（DATA0/1=0）或 DATA1（DATA0/1=1）数据包。USB 不改变该位。

KEEP/PID[3]：如果 KEEP 置 1，且 OWN 位置 1，OTG 模块将持久地对 BD 互斥访问。当令牌已经被处理时，KEEP 应清 0，这时 OTG 模块才释放 BD。一般地，对于实时端点（送给 FIFO），该位置 1。在这种情况下，NINC 通常置 1，禁止地址增加。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第 3 位被写回。

NINC/PID[2]：当 NINC 置 1，并访问数据缓冲区时，DMA 引擎将不会增加它的地址。这对于在一个存储器中存取数据的端点来说是有用的，如一个 FIFO。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第 2 位被写回。

DTS/PID[1]：如果 DTS 位置 1 将使能 OTG 模块同步触发。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第 1 位被写回。

BDT_STALL/PID[0]：如果将该位置 1，解释道一个 BD 处理的令牌，将使 OTG 模块发出一个 STALL 握手协议。在这种情况下，BD 的内容和它的 OWN 位不会改变。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第 0 位被写回。

缓冲区地址（32 位）：表示系统存储空间的 32 位地址。

14.3.4. MCF52223的USB模块主要寄存器介绍

MCF52223 的 USB 模块主要使用到表 14-6 中的寄存器，其它寄存器的详细信息可参考光盘
中本章的阅读材料

表 14-6 USB 模块寄存器列表

寄存器	缩写	描述
中断状态寄存器	INT_STAT	USB模块中断源
中断使能寄存器	INT_ENB	USB模块中断使能
状态寄存器	STAT	记录USB模块中事务状态
控制寄存器	CTL	提供USB模块各种控制和配置
地址寄存器	ADDR	根据主从不同而定
BDT寄存器1	BDT_PAGE_01	页寄存器1
Token 寄存器	TOKEN	主机模式下发起事务
BDT页寄存器2	BDT_PAGE_02	页寄存器2
BDT页寄存器3	BDT_PAGE_03	页寄存器3
端点控制寄存器0 端点控制寄存器15	ENDPT0 ENDPT15	对各个端点的控制

1. 中断状态寄存器INT_STAT（Interrupt Status Register）

中断状态寄存器为 8 位寄存器，与中断使能寄存器（INT_ENB）的 8 位一一对应。该寄存器的所有位与 OTG 中断状态寄存器（OTG_STAT）逻辑或以得到 ColdFire 核的一个中断源。中断位为 1 后，通过向中断位写 1 清零。复位后该寄存器的值为 0x00。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	STALL	ATTACH	RESUME	SLEEP	TOK_DNE	SOF_TOK	ERROR	USB_RST

在程序开发中主要考虑 D3 和 D0 位。

D3—TOK_DNE：令牌完成位。当处理完当前的令牌后，USB 模块自动将该位置 1。软件向该位写 1 将清 0。

D0—USB_RST：USB 复位位。USB 模块解析到有效的 USB 复位（2.5 毫秒）就将该位置 1，从而通知微处理器向地址寄存器中写入 0x00 并使能端点 0。

2. 中断使能寄存器INT_ENB（Interrupt Enable Register）

中断使能寄存器的每 1 位使能 USB 模块中的中断源。置 1 使能 INT_STAT 寄存器中相应的中断源。复位后该寄存器的值为 0x00。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	STALL _EN	ATTACH _EN	RESUME _EN	SLEEP _EN	TOK_DNE _EN	SOF_TOK _EN	ERROR _EN	USB_RST _EN

与中断状态寄存器一样，在程序开发中主要考虑 D3 和 D0 位。

D3—TOK_DNE_EN：令牌完成使能位。0：禁止，1：使能。

D0—USB_RST_EN：USB 复位使能位。0：禁止，1：使能。

3. 状态寄存器STAT（Status Register）

状态寄存器记录了 USB 模块中事务的状态。当产生令牌完成中断中断时，应先读取状态寄存

器以确定端点的通信状态。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	ENDP				TX	ODD		

D7~D4: 端点号。该 4 位值对应令牌完成中断对应的端点号，从而确定对应的 BDT 选项。

D3: 传输方向指示位。0: 上一个事务为接收操作；1: 上一个事务为发送操作。

D2: 如果最后更改的缓冲区描述符位于 BDT 的奇数行就将该位置 1。

4. 控制寄存器CTL（Control Register）

控制寄存器提供了 USB 模块的各种控制和配置信息。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	JSTATE	SE0	TXSUSPEND/ TOKENBUSY	RESET	HOST_ MODE_EN	RESUME	ODD_ _RST	USB_EN/ SOF_EN

D7—JSTATE: USB 差分接收 J 状态信号。该信号的极性受地址寄存器 ADDR 的 LS_EN 位状态的影响。

D6—SE0: USB 的 SE0 信号。

D5—TXSUSPEND/TOKENBUSY: 在主机模式时，将该位置 1 后，USB 模块将处理 USB 令牌，此时不能再向令牌寄存器写入令牌命令。从机模式时，SIE 禁止了包发送和接收时，将 TXSUSPEND 置 1。清 0 该位，SIE 能继续处理令牌。接收到 Setup 令牌后 SIE 将该位置 1。

D4—RESET: 将该位置 1 使能 USB 模块产生复位信号，从而使能 USB 模块复位 USB 外设。该信号只有在主机模式下（HOST_MODE_EN=1）才有效，在所需时间内，软件必须将 RESET 置 1 然后清 0。

D3—HOST_MODE_EN: 主机模式使能位，为 1 使能 USB 模块工作于主机模式。

D2—RESUME: 将该位设为 1 使能 USB 模块产生唤醒信号，从而使能 USB 模块远程唤醒。软件必须先将 RESUME 位置 1，等待规定的时间后再清 0。如果 HOST_MODE_EN 位为 1，当将 RESUME 位清 0 时，USB 模块在唤醒信号后添加一个低速结束包。

D1—ODD_RST: 将该位置 1 会将 BDT 的所有位复位为 0，同时指定到 BDT 的偶数行。

D0—USB_EN/ SOF_EN: USB 使能位。0: 禁止 USB 模块，1: 使能 USB 模块。

5. 地址寄存器ADDR（Address Register）

作为外设模式（HOST_MODE_EN=0），USB 模块解码时，地址寄存器有唯一值。而作为主机模式（HOST_MODE_EN=1），USB 模块使用令牌包传输地址，从而使能 USB 模块为 USB 外设指定唯一地址。无论哪种模式，控制寄存器的 USB_EN 位均为 1。复位或 USB 模块检测到复位信号时，地址寄存器复位为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	LS_EN	ADDR						

D7—LS_EN: 低速使能位。该位通知 USB 模块写入令牌寄存器的下一个令牌命令时，必须以低速执行。

D6~D0—ADDR: USB 地址。定义了在外设模式下，USB 模块解析的外设地址，或主机模式下传输的地址。

6. BDT页寄存器BDT Page Register1 ~ 3（BDT_PAGE_01 ~ BDT_PAGE_03）

缓冲区描述符表(BDT)页寄存器 1~3 用来计算当前缓冲区描述符表在系统存储空间中的地址。

BDT_PAGE_01 寄存器定义:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	BDT _BA15	BDT _BA14	BDT _BA13	BDT _BA12	BDT _BA11	BDT _BA10	BDT _BA9	

BDT_PAGE_02 寄存器定义:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	BDT _BA23	BDT _BA22	BDT _BA21	BDT _BA20	BDT _BA19	BDT _BA18	BDT _BA17	BDT _BA16

BDT_PAGE_03 寄存器定义:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	BDT _BA31	BDT _BA30	BDT _BA29	BDT _BA28	BDT _BA27	BDT _BA26	BDT _BA25	BDT _BA24

事实上,缓冲区描述符表 BDT 的 32 位地址通过如下方式拼接而成:

31: 24	23: 16	15: 9	8: 5	4	3	2: 0
BDT_PAGE_03	BDT_PAGE_02	BDT_PAGE_01[7: 1]	ENDP	TX	ODD	000

其中, ENDP、TX 以及 ODD 为状态寄存器 STAT 内容, 分别表示端点号、传输方向以及 DATA0 或 DATA1。

7. 令牌寄存器 TOKEN (Token Register)

令牌寄存器用于在主机模式下 (HOST_MODE_EN=1) 执行 USB 事务。当 ColdFire 核处理器要向外设发起一个事务时, 将令牌类型和端点写入该寄存器。写完后, USB 模块开始向地址寄存器中的地址发起指定的事务。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TOKEN_PID				TOKEN_ENDPT			

D7~D4—TOKEN_PID: USB 模块执行的令牌类型。有效的令牌类型有: 0001 (OUT 令牌)、1001 (IN 令牌)、1101 (SETUP 令牌) 等。

D3~D0—TOKEN_ENDPT: 令牌命令的端点地址。

8. 端点控制寄存器 ENDPT0~15 (Endpoint Control Registers)

USB 模块中有 16 个可用端点, 端点控制寄存器拥有每个端点的控制位。格式如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	HOST_WO _HUB	RETRY _DIS		EP_CTL _DIS	EP_RX _EN	EP_TX _EN	EP_STALL	EP_HSHK

D7—HOST_WO_HUB: 该位只能用于主机模式并且只在端点 0 (ENDPT0) 控制寄存器中有效。该位为 1 时允许主机直接和低速设备通信。为 0 时, 主机需要通过 HUB 与低速设备通信, 发送令牌给低速设备时, 主机提供 PRE_PID 然后切换到低速信号。

D6—RETRY_DIS: 该位只能用于主机模式并且只在端点 0 (ENDPT0) 控制寄存器中有效。将该位设为 1 时, 主机将重发 NAK (不响应) 事务。事务没有响应时, 用 NAK PID 更新 BDT PID, 并将 TOKEN_DNE 中断置 1。

D5—未定义。

D4—EP_CTL_DIS: 该位置 1 会禁止控制传输 (SETUP), 设为 0 时使能控制传输。只有当 EP_RX_EN 和 EP_TX_EN 位都为 1 时才有效。

D3—EP_RX_EN: 该位为 1 时, 使能端点为 RX 传输。

D2—EP_TX_EN: 该位为 1 时, 使能端点为 TX 传输。

D1—EP_STALL: 该位为 1 时表明使用了该端点, 该位的优先级比端点使能寄存器的其它位都高, 只有在 EP_TX_EN=1 或 EP_RX_EN=1 时才有效。访问该端点会使得 USB 模块返回一个 STALL 握手包。

D0—EP_HSHK: 该位设为 1 时, 该端点的事务期间能执行握手。

表 14-7 给出了端点寄存器的 EP_CTL_DIS 位、EP_RX_EN 位以及 EP_TX_EN 位的设置示例及其含义。

表 14-7 端点方向和控制

EPL_CTL_DIS	EP_RX_EN	EP_TX_EN	端点使能 / 方向控制
X	0	0	禁止端点
X	0	1	端点只允许 TX 传输
X	1	0	端点只允许 RX 传输
1	1	1	端点允许 TX 和 RX 传输
0	1	1	端点允许 TX 和 RX 传输及控制 (SETUP) 传输

14.4 MCF52223作为USB从机的开发方法

如果使用 PC 机作为主机, 那么 USB 应用系统的开发主要包括对 USB 驱动程序、PC 方应用程序和设备固件程序的编写。

14.4.1 PC方USB设备驱动程序的选择及基本原理

1. 驱动概述

设备驱动程序又叫功能驱动程序 (Function Driver), 它安装在计算机主机上, 负责管理一种具体的设备。在 WINDOWS 操作系统下, 应用程序不能直接对硬件接口操作, 对硬件资源的访问是交由驱动程序完成的。应用程序只要同设备驱动程序通信, 就能实现和设备的数据交换。

设备驱动程序只是支持计算机主机与设备进行通信的软件组中的一层。因为在 USB 通信中使用分层式的驱动程序模型, 其中最具有代表性的是 Microsoft 力推的全新驱动程序模式 WDM (Windows Driver Model), 即 Windows 驱动程序模型 (关于 WDM, 请参阅其他相关驱动开发参考文献, 这里不再详述)。不同层之间的驱动程序完成不同的操作, 且可相互调用。驱动程序分为以下几个层次: 总线驱动程序、总线过滤驱动程序、下层过滤驱动程序、功能驱动程序和上层过滤驱动程序。总线驱动程序负责管理逻辑的或物理的总线, 例如 PCMCIA、PCI、USB、IEEE1394 和 ISA, 检测并向 PnP (Plug and play, 即插即用) 管理器通知总线上的设备, 并且能够管理电源。过滤驱动程序 (Filter Driver) 与功能驱动程序协同工作, 用于增加或改变功能驱动程序的行为。一般操作系统已经提供了总线驱动和过滤驱动。

有时候用户 USB 设备的功能在 Windows 提供的驱动程序, 或者厂商提供的驱动程序中都没有, 这时候就需要自己编写驱动程序。

2. 驱动程序开发工具

要建立一个 WDM 驱动程序通常需要 Microsoft 的 Visual C++编译器 (其中包含完整的开发环

境和调试功能)、Windows DDK (Windows Device Developer's Kit) 即 Windows 设备开发包、MSDN (Microsoft's Developer's Network) 即 Windows 面向软件开发者的一种信息服务、驱动程序工具软件以及高级的调试器等。

其中, Windows DDK 包含以下内容:

(1) 范例代码和文件说明。与 USB 相关的文件包括 WDM 驱动程序的教学示范, 以及 USB 驱动程序源代码。

(2) 批量传输的源代码和编译码、文件说明以及 bulkusb.sys 驱动程序的范例应用程序。Bulkusb.sys 驱动程序可以使用在任何支持批量传输的 USB 芯片上, 应用程序使用 ReadFile 和 WriteFile 函数来读/写数据。

(3) 处理实时传输的 isousb.sys 驱动程序。在 USB Implementers Forum 网站上, 有使用这些驱动程序的技巧和修正说明。

(4) 一个过滤器驱动程序范例, 以及 usbview 工具软件。

3. 驱动标识 GUID

GUID (Globally Unique Identifier, 全球唯一标识符) 是一个 128 位的字母数字标识符, 用于指示产品的唯一性安装。在许多流行软件应用程序 (例如 Web 浏览器和媒体播放器) 中, 都使用 GUID。

GUID 的标准格式为 “xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx”, 将 GUID 分成 5 个类别的十六进制字符, 类别之间以连字号 “-” 隔开, 其中 x 是 0-9 或 A-F 范围内的一个十六进制的数字。例如: 6F9619FF-8B86-D011-B42D-00C04FC964FF 就是一个有效的 GUID 值。

世界上任何两台计算机都不会生成重复的 GUID 值。GUID 主要用于在拥有多个节点、多台计算机的网络或系统中, 分配具有唯一性的标识符。在 Windows 平台上, GUID 应用非常广泛: 注册表、类及接口标识、数据库、甚至自动生成的机器名、目录名等。

这里用 GUID 唯一标识驱动程序, 是主机应用程序和驱动程序的通信接口 (如主机应用程序可以通过驱动的 GUID 找到设备路径)。Windows 会替标准对象 (例如 HID 类别) 定义 GUID。其他设备可以使用 Visual Studio 编译环境内所附的 guidgen.exe 程序, 来取得一个 GUID。如在 VC.NET 2003 环境下, 选择菜单栏中【工具】下的【创建 GUID】选项, 就可以弹出如图 14-9 所示的创建 GUID 对话框, 以生成新的 GUID。GUID 包含在驱动程序的程序代码内。应用程序可以使用 API 来读取设备它。

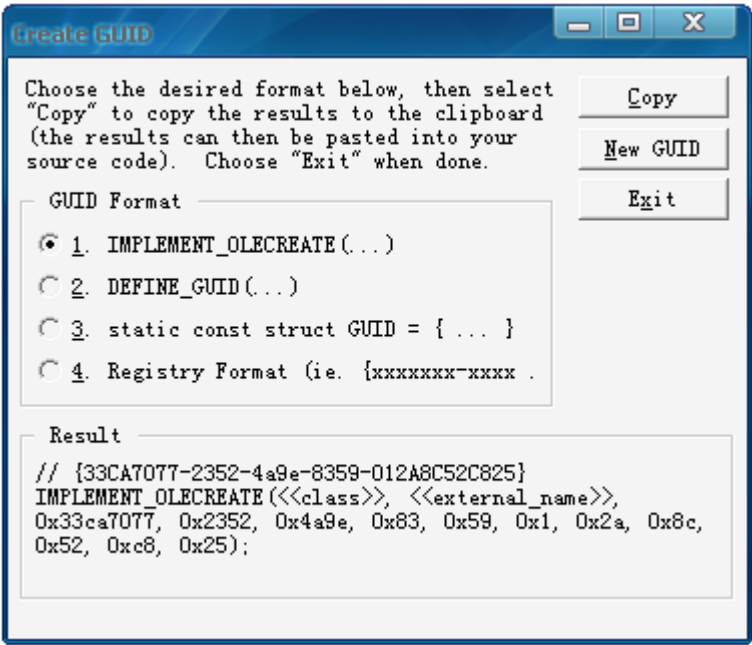


图14-9 创建GUID对话框

4. 驱动程序文件（.sys）和设备信息文件（.inf）

驱动程序文件是驱动程序的可执行代码，其扩展名为.sys。对于 PnP 设备，在设备插入后，.sys 文件会被 Windows 装载到内存中，系统线程调用.sys 中的函数来和设备进行通信。驱动程序编写好了之后，用 DDK 工具可以生成相应的.sys 文件。

INF（Device Information File）即设备信息文件，用来指示安装设备驱动程序（*.sys），驱动程序要配合 INF 才可以安装。INF 文件的后缀名为.inf，但实际上是一个文本文件，可以用记事本工具打开、查看和编辑。在 INF 文件中提供了设备的产品 ID、对应的.sys 文件名、驱动 class 名以及 class GUID；还指明了硬件驱动该如何安装到系统中、源文件在哪里、安装到哪一个文件夹中、怎样在注册表中加入自身相关信息等。INF 文件应该由驱动程序开发人员随驱动程序一起提供。用户可以按照一定的语法规则自行编写 INF 文件，但为了快速开发，也可在 Microsoft 提供的模板上修改，得到合适的 INF 文件。

INF 文件由许多按层次结构排列的节组成。INF 文件中常用的节如表 14-6 所示。

表 14-6 INF 文件中常用的节

节	说明
Version	INF 文件的开始，描述了版本信息，主要用于版本控制。
DestinationDirs	指明 INF 文件和驱动程序的目标目录。
SourceDisksNames	指明驱动程序所在的磁盘或 CD-ROM。
SourceDisksFiles	指明驱动程序的文件名。
Manufacturer	指明供应商及其对应 Models 节的名称。
DDInstall（Windows2000）	指明需复制的文件、向注册表中添加的内容等信息。
DDInstall.Services	指明驱动程序安装的详细信息。

在 Version 节，根据驱动使用的具体 GUID，对 ClassGUID 进行赋值。在 Manufacturer 节，指定了设备的 VID 和 PID，在设备固件的设备描述符中也定义了这两个值。当设备固件中的这两个值改变后，这里也要做相应修改。这样才能保证 PC 连接设备后，能正确加载合适的驱动。在 Strings

段定义了设备驱动的描述信息。

5. 驱动程序的安装

在设备的枚举过程中，当主机获得完整的配置描述符之后，并在主机对设备配置之前，主机弹出 Windows 消息，显示“产品字符串”，如图 14-10 (a)，产品字符串可以在固件程序的任意一个字符串描述符中定义，再将设备描述符的产品字符串索引域定义为相应的字符串索引即可。然后，主机开始分配并加载设备驱动。主机通过设备描述符获悉了设备后，会为该设备寻找一个最合适的驱动来管理与设备的通信。在选择一个驱动的时候，Windows 尝试将设备中的 VID、PID 和版本号（此项可选）与 PC 的 INF 文件中的相匹配，逐个寻找与之相符合的 INF 文件。过程如下：（1）系统从连接的 USB 设备中获得设备的 VID（设备描述符的供应商字段）和 PID（设备描述符的产品字段），从而得到设备的硬件 ID。（2）系统查找与该硬件 ID 相符的 INF 文件，如果找不到，系统将读取接口描述符，从中提取 USB 设备的兼容 ID，并查找与兼容 ID 相符合的 INF 文件。（3）如果仍未找到，系统提示用户自己安装该 USB 设备驱动程序。这时，会显示【找到新的硬件向导】对话框，引导用户安装相应的设备驱动程序。一般只要从指定位置安装.sys 后缀的驱动程序即可。

在操作系统成功安装驱动程序之后，系统会将驱动程序文件 *.sys 保存在 windows/system32/drivers 目录中，将相应的 *.INF 文件保存在 windows/inf 目录中（Windows 2000/XP 会将没有经过微软认证的驱动程序的 INF 文件改名为 OEMx.INF 和 OEMx.PNF 来存放，x 是从数字 0 开始的正整数，PNF 为预编译信息文件），并且系统会弹出消息，提示用户“新硬件已安装并可以使用了”，如图 14-10 (b)。同时 Windows “设备管理器”中会增加这个已安装的新设备驱动项，如图 14-10 (c) 中的“test Interact”。驱动程序安装完成后，系统的注册表也会储存所安装设备驱动的信息。Windows 通过注册表和 INF 文件来将设备和驱动程序之间的关系绑定。点击 Windows 桌面的开始菜单，选择【运行】，在运行对话框中输入 regedit，点确定后就可以查看和编辑注册表的内容了。图 14-10 (d)，就是一个 Windows 注册表。注册表编辑器使用树状结构来安排其内容，选择 HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\DeviceClasses 分支，在该分支下再选中设备驱动程序的 GUID，就可以显示 USB 设备的驱动程序的注册信息。

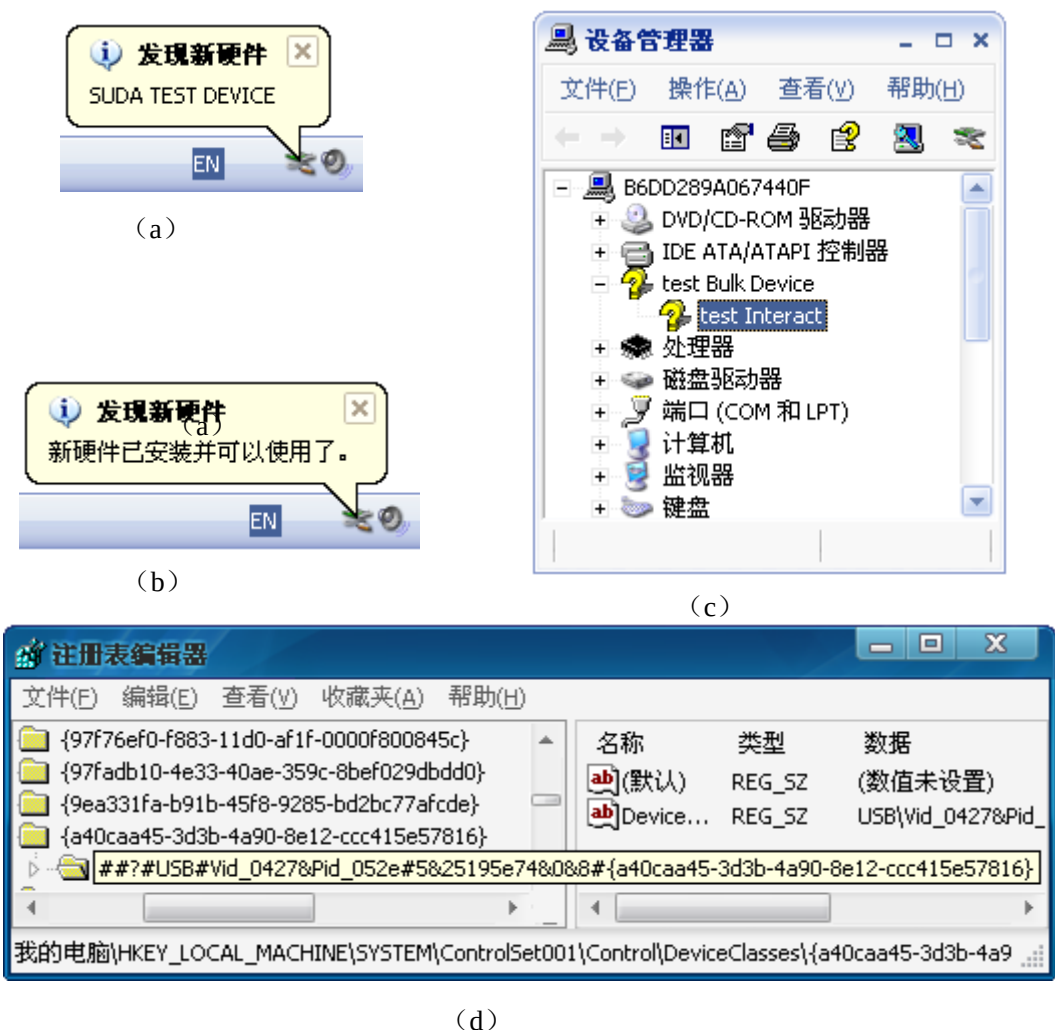


图14-10 WINDOWS中的设备信息

如果主机已经安装了设备使用的驱动程序,这时再接入相应的设备,会有一连串的配置过程,这个过程是由系统自带的驱动程序来完成的,对用户来说是透明的。在配置的过程当中,系统会根据 USB 设备返回来的信息(PID 和 VID)到注册表中寻找相应的驱动程序。具体步骤如下:

- (1) USB 总线报告设备改变;
- (2) 系统发送 I/O 请求包 (IRP), 获得子 DeviceObject, 得到硬件 ID、兼容 ID 等。
- (3) 为子 Device 设定 PhysicalDevice 信息。
- (4) 检查是否已经安装过驱动程序。如果已经安装过,就直接加载驱动;否则,检查驱动程序数据库,查看有没有与硬件 ID、兼容 ID 相同的驱动程序,如果有,则直接安装;否则报告“找到新硬件”。

14.4.2 PC机作为USB主机的程序设计

通常, PC 方应用程序作为主机程序要有数据接收、处理和发送能力,并能够提供友好的界面以便于用户操作和使用。

1. PC方测试程序功能

为了让读者掌握 USB 通信机理，本书配套的网络光盘中提供了一个 PC 方测试程序。测试程序采用 VB 语言编写，工程包括 1 个窗体模块（TestUSB20.frm）和 2 个标准模块（USB20.bas 和 GeneralFuc.bas）。

测试程序的运行界面如图 14-11 所示。该测试程序不仅提供了基本的 USB 数据发送和接收功能，还能实时显示程序运行时的状态、错误信息以及目标设备的相关描述符。用户可在“数据通信”的发送数据文本框中输入要发送的数据（该程序支持一次发送或接收 64 个英文字母和数字数据），再单击【发送】按钮，将数据通过 USB 总线发送给低端目标设备，此时程序使能【接收】按钮，单击该按钮后，高端程序将从低端收回上一次发送给低端设备的数据，并显示在接收数据文本框中。“程序运行信息提示”栏实时显示了程序运行时的状态、错误信息和 PC 机 USB 端口上的设备变化信息（如：提示有设备插入、移除及相关设备类型）。“设备信息”栏显示目标设备的相关描述符信息，可以选择显示设备、配置、接口或端点描述符等。



图14-11 PC方测试程序界面

2. 与USB通信相关的函数介绍

在测试工程的标准模块 USB20.bas 中，声明了 14 个 API 函数，自定义了 6 个用户函数，这些函数与 USB 通信密切相关。

1) API函数

表 14-8 给出了测试程序中用到的 API 函数的功能描述。函数的具体使用方法请参考网络光盘中的程序及 MSDN。

表14-8 API函数功能描述

函数名	功能描述
-----	------

RegisterDeviceNotification	登记Windows要接收信息的设备或设备类型
UnregisterDeviceNotification	注销指定的设备或设备类型
RtlMoveMemory	移动一块内存内容到另一处内存
CallWindowProc	将消息信息传送给指定的窗口过程
SetWindowLong	改变指定窗口的属性
SetupDiGetClassDevs	返回与所给参数匹配的已安装的所有设备的信息集的句柄
SetupDiEnumDeviceInterfaces	枚举包含在设备信息集中的设备接口
SetupDiGetDeviceInterfaceDetail	获得设备接口详细信息
CreateFile	创建、打开或截去一个文件。
DeviceIoControl	向指定的设备驱动程序发送控制码（如获取设备、配置、接口和字符串描述符等），使USB设备执行相关操作
CloseHandle	关闭一个内核对象，包括文件、文件映射、进程、线程、安全和同步对象等
WriteFile	写文件
ReadFile	读文件
GetLastError	获取错误代码

2) 用户自定义函数

(1) WindowProcDevice 函数：自定义的消息处理函数

当 USB 设备与 PC 机连接以后，将触发 WM_DEVICECHANGE 消息，程序调用自定义的消息处理函数 WindowProcDevice（工作流程如图 14-12）。在此函数中，首先判断连接上的 USB 设备的类型。当检测到设备时，程序会进一步读取 USB 设备字符串描述符，判断连接上的 USB 设备是否为指定的“SUDA TEST DEVICE”目标设备，若是，则程序进一步读取该设备的全部描述符信息，并允许高低端通过 USB 总线收发数据，否则就不能通信。注意：本程序只提供与使用苏大嵌入式实验室 USB2.0 驱动的设备通信的功能，对于 USB 存储设备和其他类型的 USB 设备，只通知用户设备类型和连接状态而不提供通信功能。

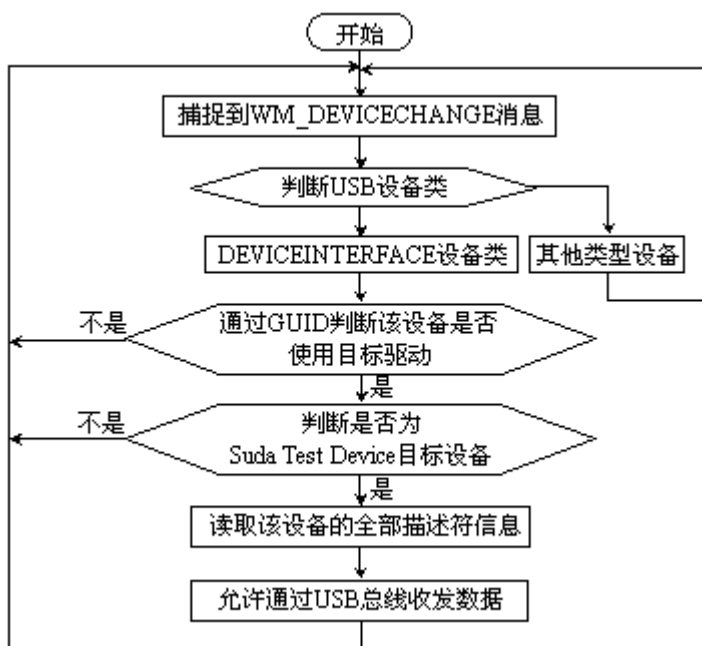


图14-12 自定义消息处理函数WindowProcDevice工作流程

(2) OpenUSB 函数：扫描目标设备并获得设备路径。

该函数主要是对当前连接到主机 USB 端口上的设备进行扫描，并获得某些指定设备（这些设备使用了由指定的 GUID 标识的驱动）的路径，为以后的数据通信做好准备。该函数模块先调用 API 函数 SetupDiGetClassDevs，得到指定设备的信息结构体数组地址，再通过 API 函数 SetupDiEnumDeviceInterfaces 枚举包含在该设备信息集（由上一个函数的返回值确定）中的设备接口。最后通过调用 API 函数 SetupDiGetDeviceInterfaceDetail 获得与所设参数匹配的设备接口的详细信息，这里主要是要得到设备路径。

(3) GetStringDescribe 函数：获取设备信息集中设备的配置、接口、端点以及字符串描述符。

(4) GetDeviceInfo 函数：根据设备路径，取得设备代号，并用此代号与设备交换数据。

(5) OpenWriteHandle 函数：调用 API 函数 CreateFile，实现数据发送功能。

(6) OpenReadHandle 函数：调用 API 函数 CreateFile，实现数据接收功能。

14.4.3 MCF52223作为USB从机的程序设计

1. 配置过程

USB 设备必须通过控制传输配置成功后才能与主机进行通信。下面从 MCF52223 的角度阐述 USB 设备从插入主机到通信时的配置过程。

插入主机 USB 口后，MCF52223 就开始工作，调用 usb_init 对 USB 模块初始化。当使能了 D+ 上的上拉电阻后，主机就识别到该 USB 设备，随即对设备进行枚举，具体过程可参照 14.2.3 节的设备枚举过程。主机会要求读取设备描述符，其中供应商 ID 和产品 ID 信息与驱动程序密切相关。USB 设备可支持多个配置，但某时刻只能有一个有效，用户可在初始化设备时指定。主机获得设备描述符后继续读取配置描述符，根据所支持的接口数读取接口描述符，根据接口描述符

中的端点数读取端点描述符并对其进行配置。这几个描述符密切相关，要同时修改相应的字段。在 14.3.2 节第 3 部分介绍了 MCF52223 的 USB 模块，由于其 USB 模块的数据通信是由缓冲区描述符表 BDT 控制的，因此必须在 RAM 中开辟连续的 512 字节作为 BDT（可根据实际使用的端点数修改 BDT 的大小），而 USB 模块是通过页寄存器来指向 BDT，所以在 USB 初始化过程中（使能 USB 模块前）必须设置页寄存器，其作用是定位到端点的缓冲区描述符表。在接收到数据或发送数据时，USB 模块就能根据该寄存器及状态寄存器找到该端点的缓冲区描述符表项，并将数据存放到对应的缓冲区或将缓冲区描述符表项指定的发送缓冲区的内容发送出去。配置完成后必须将寄存器 MCF_USB_OTG_CTRL 的 MCF_USB_OTG_CTRL_DP_HIGH 位上拉，这样才能将总线由 SE0 状态改变到 SE1 状态，从而使得主机识别到设备为全速，再对其进行配置。

MCF52223 的通信过程比较简单，当接收到数据后就会将数据存放到端点 BDT 所指定的缓冲区中，用户程序只要将缓冲区的内容读出即可。如果要发送数据，先要将数据放入缓冲区，将发送端点 BDT 的地址域指向发送数据所在的缓冲区，修改 BC、DTS 字段启动数据传输即可。

2. 固件程序设计

本书中的从机 USB 设备的端点类型包括支持控制传输的端点 0 以及支持批量传输的其它端点。自 USB 设备插入 PC 的 USB 端口后，首先要经历一个枚举过程（详见 14.2.3 节），在该过程中，USB 设备要复位其 USB 模块并通过端点 0 向 PC 主机报告自身的设备信息。枚举过程之后，设备就能用其他端点与 PC 进行批量传输数据了。下面从 USB 模块初始化、端点 0 的控制传输、数据接收过程以及数据发送过程这四方面讲述从机固件程序的设计。

1) USB模块初始化

在 MCF52223 中，端点的控制信息是存放在 RAM 的 BDT 中，MCF52223 是通过设置 BDT 来进行 USB 通信的，而 USB 模块是通过页寄存器定位到 BDT 的，所以必须在启动 USB 模块前设置好页寄存器。同时要在使能 USB 模块前将寄存器 MCF_USB_OTG_CTRL 的 MCF_USB_OTG_CTRL_DP_HIGH 位上拉，这样主机才能识别到设备并对其进行配置，这些工作是在 USB 模块初始化中完成的。

```
//-----*
//函数名: usb_init                                *
//功    能: USB模块初始化                          *
//参    数: ip: 中断优先级(0~7) 0最低, 7最高      *
//返    回: 成功返回0, 不成功返回非0              *
//内部调用: usb_default                            *
//说    明: 先将MCF_USB_OTG_CTRL的DP位拉低, 退出前再将该位拉高, 这样才能在 *
//          总线的D+和D-线上状态产生变化, 使得主机识别到该USB设备。      *
//-----*

void usb_init(uint8 ip)
{
    MCF_USB_CTRL = 0x01;    //退出suspend状态并设定时钟源为外部晶振
    MCF_USB_INT_ENB = 0x0;  //禁止所有USB中断
    MCF_USB_OTG_INT_EN = 0x0; //禁止所有OTG中断
    MCF_USB_ERR_ENB = 0x0;  //禁止所有USB错误中断

    MCF_USB_OTG_CTRL &= ~MCF_USB_OTG_CTRL_DP_HIGH;

    //配置USB为默认状态
```

```

usb_default();

MCF_USB_OTG_INT_STAT = 0xff; //清除OTG中断标志
MCF_USB_INT_STAT = 0xff; //清除USB中断标志
MCF_USB_ERR_STAT = 0xff; //清除USB Error中断标志

MCF_INTC0_ICR53 = 0x30 + ip; //设置中断优先级
//使能中断控制器中的USB中断
MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK53;
MCF_INTC0_IMRL &= ~MCF_INTC_IMRL_MASKALL;

//使能USB中断
MCF_USB_INT_ENB = MCF_USB_INT_ENB_SLEEP | MCF_USB_INT_ENB_TOK_DNE
                  | MCF_USB_INT_ENB_ERROR | MCF_USB_INT_ENB_USB_RST
                  | MCF_USB_INT_ENB_STALL;

//设定缓冲区描述符表BDT的地址
MCF_USB_BDT_PAGE_01 = (uint8)((uint32)BDT_BASE >> 8);
MCF_USB_BDT_PAGE_02 = (uint8)((uint32)BDT_BASE >> 16);
MCF_USB_BDT_PAGE_03 = (uint8)((uint32)BDT_BASE >> 24);

//使能USB模块
MCF_USB_CTL = MCF_USB_CTL_USB_EN_SOF_EN;

//使能D+和D-上的上拉电阻
MCF_USB_OTG_CTRL = MCF_USB_OTG_CTRL_DP_HIGH|MCF_USB_OTG_CTRL_OTG_EN;
}

```

2) 端点0的控制传输

前面提到, USB 设备必须通过控制传输配置成功后才能与主机进行通信。而配置过程是通过 USB 中断服务程序实现的。设备的相关信息作为固件的一部分, 在低端程序中是通过设备、配置、接口及端点等描述符来表达的。当发生 USB 中断时, 若检测到是 SETUP 中断, 程序会调用 SETUP 包解码函数 `usb_decode_setup` 来分析主机请求, 并做出相应的响应动作。例如: 若 SETUP 包是取设备描述符, 那么 USB 设备会调用 `_usb_send` 函数将设备描述符发送给主机。USB 中断服务程序流程如图 14-13 所示。

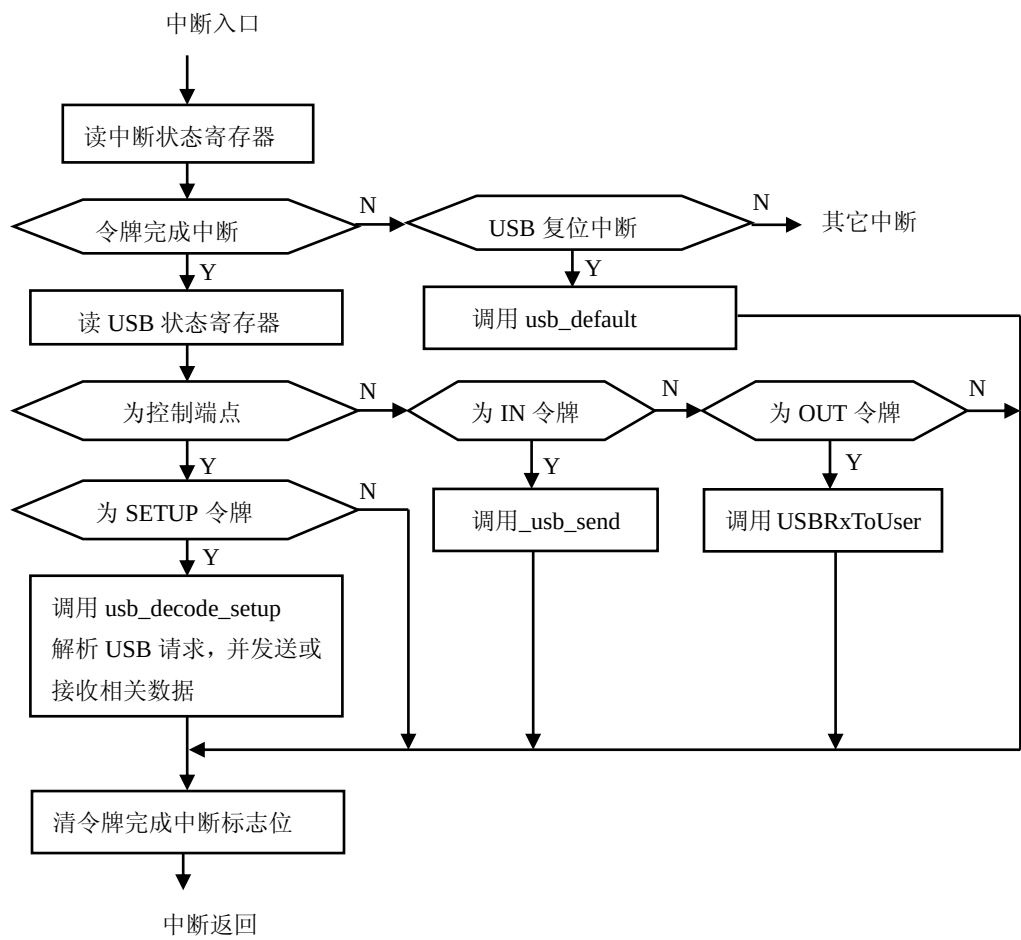


图14-13 USB中断服务程序流程图

3) 数据接收过程

USB 模块接收到数据后是自动存放在 BDT 所指定的缓冲区中，而端点支持双缓冲，因此在移交控制权之前需要选择数据接收缓冲区，该过程通过调用 USBReadyReceive 函数完成。接收的实际长度存放在 BDT 的 BC 域中，用户程序要及时从接收缓冲区中将数据取出。数据接收过程是通过 USBReceiveData 函数完成的。

```
//-----*
//函数名: USBReceiveData                                *
//功 能: USB接收函数                                    *
//参 数: ep:接收端点号;buff:数据存放的缓冲区首址      *
//返 回: 接收到的字节数                                *
//-----*
uint16 USBReceiveData(uint8 ep,uint8 *buff)
{
    uint16 len;
    ep_info[ep].data_func = 0;
    ep_info[ep].address   = buff;           //用户缓冲区地址
    ep_info[ep].state     = EPST_DATA_RX;   //初始状态
}
```

```

    ep_info[ep].error = USBEPERR_NONE;        //清除错误
    ep_info[ep].flags = 0;
    //将控制权交给USB-FS
    USBReadyReceive(ep, ep_info[ep].data0_rx);
    //等待接收到数据
    while(ep_info[ep].tlength == 0);
    len = ep_info[ep].tlength;
    ep_info[ep].tlength = 0;
    //返回接收到的字节数
    return len;
}

```

4) 数据发送过程

配置完成后, USB 模块就能进行数据发送了。发送的过程比较简单, 程序只要将发送缓冲区的地址及数据长度写到指定端点的 BDT 中, 调用 USBReadySend 函数启动数据发送。USB 模块根据三个页寄存器 (MCF_USB_BDT_PAGE_01、02、03) 和状态寄存器找到对应的 BDT, 并将缓冲区中的数据发送出去, 发送的长度为 BDT 中指定的数据长度 (最大为 1024)。发送过程是通过 USBSendData 函数完成的。

```

//-----*
//函数名: USBSendData                                *
//功 能: USB发送函数                                *
//参 数: ep:发送端点号;buff:数据缓冲区;length:待发送的数据长度 *
//返 回: 无                                           *
//-----*
void USBSendData(uint8 ep,uint8 *buff,uint16 length)
{
    uint8 buf;
    ep_info[ep].tlength = length;
    ep_info[ep].maxlength = ep_info[ep].tlength;
    ep_info[ep].data_func = 0;
    ep_info[ep].address = buff;
    ep_info[ep].flags = 0;
    ep_info[ep].error = USBEPERR_NONE;
    ep_info[ep].state = EPST_DATA_TX;
    //选择发送端点的缓冲区
    buf = ep_info[ep].next_tx;
    WR_LE32(&BDT_ADR_TX(ep, buf), (uint32)ep_info[ep].address);
    //设置tx包的长度
    WR_LE32(&BDT_CTL_TX(ep, buf), (uint32)(length<<16u));
    //将控制权交给USB-FS
    USBReadySend(ep);
    ep_info[ep].data0_tx ^= BDT_CTL_DATA;
    //若最后一个数据包的长度等于端点的最大数据包长度, 则设备要向主机
    //发送0长度的数据包以结束数据阶段
    if ((length == ep_info[ep].psize) && (ep_info[ep].flags & EPFL_ZPACKET))
        ep_info[ep].state = EPST_TX_STOP;
    else
        ep_info[ep].state = EPST_DATA_TX_LAST;
}

```

14.5 MCF52223作为USB主机的开发方法

将 MCF52223 配置为主机时, 必须将 MCF_USB_OTG_CTRL 寄存器的第 2、4、5 位设置为 1, 在 USB 控制寄存器 MCF_USB_CTL 中使能主机模式, 即将第 3 位置 1。当有设备接入到 USB 口后, MCF52223 就能检测到设备。

14.5.1 MCF52223作为USB主机的基本功能

作为 USB 主机的 MCF52223 的固件程序应具有以下几方面的功能。

1. 使能主机模式

(1) 使能主机模式 (CTL[HOST_MODE_EN]=1)。使能下拉电阻, 禁止上拉电阻。开始产生 SOF。向 SOF 计数器中写入 12, 000。向 USB 使能位写 0 (CTL[USB_EN]=0) 禁止产生开始帧包。

(2) 使能 ATTACH 中断 (INT_ENB[ATTACH]=1)。

(3) 等待 ATTACH 中断 (INT_STAT[ATTACH])。USB 目标设备上拉电阻改变 DPLUS 或 DMINUS 状态, 从 0 变为 1 (SE0 变位 J 或 K 状态)。

(4) 检测控制寄存器的 JSTATE 和 SE0 的状态。如果 JSTATE 位为 0 说明所连接的设备为低速设备。如果所连接的设备为低速设备则将地址寄存器的低速位 (ADDR[LS_EN]) 以及端点 0 控制寄存器的不带 HUB 的主机位 (EP_CTL0[HOST_WO_HUB]) 置 1。

(5) 使能 RESET (CTL[RESET]) 10ms。

(6) 使能 SOF 包以防止接入设备从运行转到挂起状态 (CTL[USB_EN]=1)。

2. 与接入设备完成控制传输

(1) 将端点控制寄存器设置为双向控制传输, 即 EP_CTL0[4:0]=0x0d。

(2) 将设备框架建立命令复制到存储器缓冲区。设备框架命令集见 USB2.0 协议。

(3) 初始化端点 0 的 BDT 以发送 8 字节的设备框架命令数据 (如: Get_Descriptor)。首先将 BDT 命令字设为 0x00080080 (字节数为 8, OWN 位为 1), 然后将 BDT 缓冲区地址域设为该 8 字节命令缓冲区的开始地址。

(4) 设置 USB 地址为地址寄存器中的值 (ADDR[6:0])。USB 总线复位后, USB 设备地址为 0。通过设置地址设备框架命令会将其设置为其他的值 (通常为 1)。

(5) 向端点 0 即目标设备的默认控制管道的令牌寄存器写入 SETUP (TOKEN=0xD0)。从而在总线上发送 SETUP 令牌, 并紧跟一个数据包。该数据包传输结束后, 在 BDT 的 PID 域返回一个握手包。向 BDT 写入后, 会产生令牌结束中断 (INT_STAT[TOK_DNE]), 结束 SETUP 事务的设置阶段。

(6) 开始 SETUP 事务的数据阶段, 在存储器中为发送建立缓冲区。

(7) 初始当前端点 0 的 BDT 用于发送数据。首先, 将 BDT 命令字设为 0x004000C0 (字节数为 64, OWN 位为 1, DATA 触发为 DATA1), 然后, 将 BDT 缓冲区地址域设置为数据缓冲区的开始地址。

(8) 向端点 0 写入 IN 或 OUT 令牌, 后跟数据包。数据包结束后, 写 BDT 并产生一个令牌

结束中断 (INT_STAT[TOK_DNE])。对于单个包数据阶段, 这就结束了 SETUP 事务的数据阶段。

(9) 开始 SETUP 事务的状态阶段, 在存储器中建立一个用于接收或发送 0 长度数据包的缓冲区。

(10) 初始化端点 0 的 BDT 用于发送状态数据。首先, 将 BDT 命令字设为 0x00000080 (字节数为 0, OWN 位为 1, DATA 触发为 Data0), 然后, 将 BDT 缓冲区地址域设置为数据缓冲区的开始地址。

(11) 向端点 0 写入 IN 或 OUT 令牌, 后跟一个 0 长度数据包。数据包结束后, 会将设备发出的握手包写入 BDT 并产生一个令牌结束中断 (INT_STAT[TOK_DNE])。这就结束了 SETUP 事务的状态阶段。

3. 向目标设备发送一个全速块数据

(1) 完成发现接入设备的所有步骤并配置接入的设备。向 ADDR 寄存器写入目标设备的地址。典型情况是 USB 总线上只有一个设备, 所以 ADDR 值为 0x01 并保持不变。

(2) 设置 ENDPT0 寄存器为 0x1D, 使能带握手的发送和接收。

(3) 设置缓冲区描述符表中端点 0 的偶数发送项为最多发送 64 字节。

(4) 设置目标 USB 设备地址为地址寄存器的值 (ADDR[6:0])。

(5) 将 TOKEN 寄存器设为 OUT。这将触发 OTG 模块发送令牌和数据。

(6) 设置缓冲区描述符表中端点 0 的奇数发送项为最多发送 64 字节。

(7) 跟第 4 步一样, 向 TOKEN 寄存器写入 OUT 令牌。

(8) 等待令牌完成中断。

(9) 产生令牌完成中断后, 可以检查 BDT 并且返回到第 2 步。

14.5.2 USB 主机与 CDC 类 USB 设备通信

下面通过查询方式详细描述 USB 主机与前面开发的 CDC 类设备通信的详细过程。首先进行主机初始化, 使能主机模块。当主机检测到有 USB 设备连接时, 按照以下步骤完成对设备的初始化:

(1) 检测确实有设备连接。初次检测到有 USB 设备连接后需要按照 USB 协议清除 ATTACH 标志并等待 100ms 再次检测总线上是否有 USB 设备连接。

(2) 在确认有 USB 设备连接后需复位总线, 这时 USB 设备可以使用默认地址与主机通过控制传输进行通信。

(3) 取得设备描述符。主机调用 USBGetDeviceDesc 函数发送 IN 包, 从机接收到该包后调用 usb_decode_setup 函数进行解包, 分析后知道是读取描述符请求的标准请求, 会将设备描述符发送给主机。在设备描述符中指明了从机设备有几个配置描述符。

(4) 取得配置描述符。主机调用 USBGetConfigDesc 函数发送 IN 包, 从机接收到该包后调用 usb_decode_setup 函数进行解包, 分析后知道是读取配置描述符请求的标准请求, 会将 usb_config_descriptor[] 的配置信息发送给主机。配置信息包括配置描述符、接口描述符以及端点描述符。在本例中主机会收到 32 字节的数据, 包括一个配置描述符, 一个接口描述符以及两个端点描述符。

(5) 查找接口描述符。由于配置描述符中可以有多个接口描述符, 需要从中选择主机支持的接口。

(6) 查找端点描述符。接口描述符中指定了所支持的端点数，需要取得分别支持 IN 和 OUT 的端点。

(7) 当上述的 (4) - (6) 均满足时就可以按照该配置信息对从机进行设置，如主机发送 USBSetConfig (1)，从机调用 `usb_decode_setup` 函数进行解包后会对其进行设置。如果不满足时需要取得另外的配置描述符再按照 (4) - (6) 进行查找。

(8) 完成了上面步骤后主机就知道从机的端点支持数据发送和接收的长度，以及如何向从机的端点发送数据，即可以根据配置信息与从机进行通信了。当从机的接收端点对应的 BD 交给 USB 模块后，主机通过调用 `USBSendData` 函数，从机就能接收数据，否则从机不能接收到数据。同样的道理，从机的发送端点对应的 BD 交给 USB 模块后，主机通过调用 `USBRevData` 函数，从机就能将缓冲区中的数据发送出去。

14.5.3 USB主机与MassStorage类USB设备通信

主机对 MassStorage 类设备的配置与 CDC 类似，也是要通过控制传输知道该如何与从机进行通信，包括发送端点、接收端点及端点支持的数据长度。下面以 U 盘为例描述其通信过程，该样例程序支持以下几个命令：

`help$` 命令：显示所有支持的命令；

`dir$` 命令：列出 U 盘中的所有文件信息；

`dump 文件名$` 命令：以十六进制形式显示文件的内容；

`type 文件名$` 命令：以文本形式显示文件内容；

`emgedit 文件名$` 命令：创建文件，输入文件内容以字符“#”结束。输入完成后会在串口调试工具中显示 File Closed，表示写入文件成功。

由于 MassStorage 类与 CDC 类不同，需要了解文件的结构等信息，在本书中不作详细介绍，详细信息可参照光盘中的代码。测试界面如图 14-14：

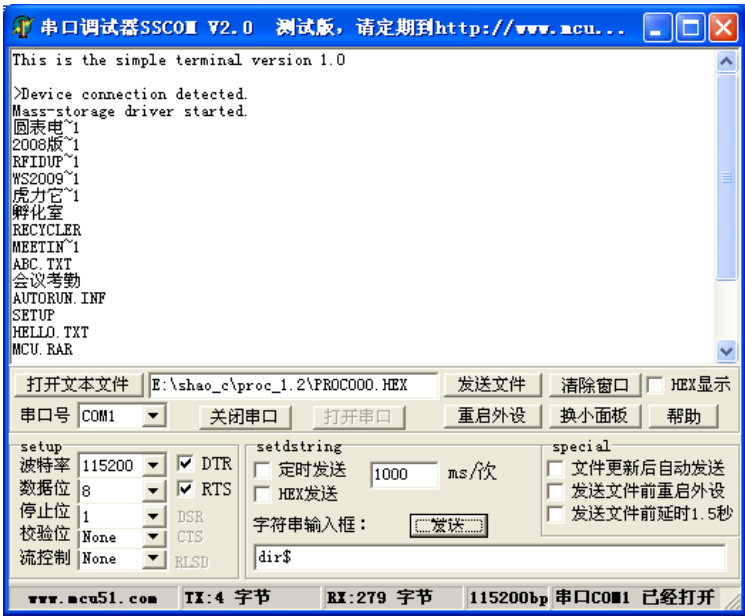


图14-14 U盘文件列举

14.6 采用OTG技术的USB系统开发方法

选用MCF52223作为主控芯片的USB设备,可以扮演主机或从机的双重角色,要做到这一点,只需更换接插在设备上的USB缆线插头类型,而无需更换芯片中的固件程序。具体地说,当芯片中固化了合适的程序,将A型或B型插头插入设备之后,设备检测到该插头类型,就会相应的切换到主机模式或从机模式。这样就可以做到即使没有PC机也能实现USB设备之间的互联,形成一个对等网。这种技术突破了使用标准USB协议的设备只能被定义为主机或从机而不能在这两者之间进行切换的限制,对该技术的详细说明就是OTG(On-The-Go)协议,它支持在没有PC的情况下USB设备间的数据传输,它是对标准USB协议的扩展和补充。

OTG的双重角色具备以下特征:

- (1) 有限的主机性能。
- (2) 像外围设备一样能全速通信。
- (3) 像主机一样支持全速通信。
- (4) 只有一个Mini-AB插座。
- (5) V_{BUS} 的输出电流最小是8mA。
- (6) 会话请求协议SRP(Session Request Protocol): 该协议允许B设备请求A设备接通 V_{BUS} 并启动会话,还允许由电池供电的A设备在没有总线活动时切断 V_{BUS} ,以节约电能。
- (7) 主机交换协议HNP(Host Negotiation Protocol): 允许在两个直接连接的双重角色设备间切换主从机功能,而无须用户切换电缆连接来改变设备间的通信控制。HNP只能通过设备的Mini-AB插座实现。

1. A设备操作的OTG双重角色

由插入的电缆类型确定设备是否为 A 设备。如果 USB A 插头或 USB Mini A 插头插入到设备则被认为是 A 设备。

A 设备的双重角色如图 14-15 和表 14-9 所示。

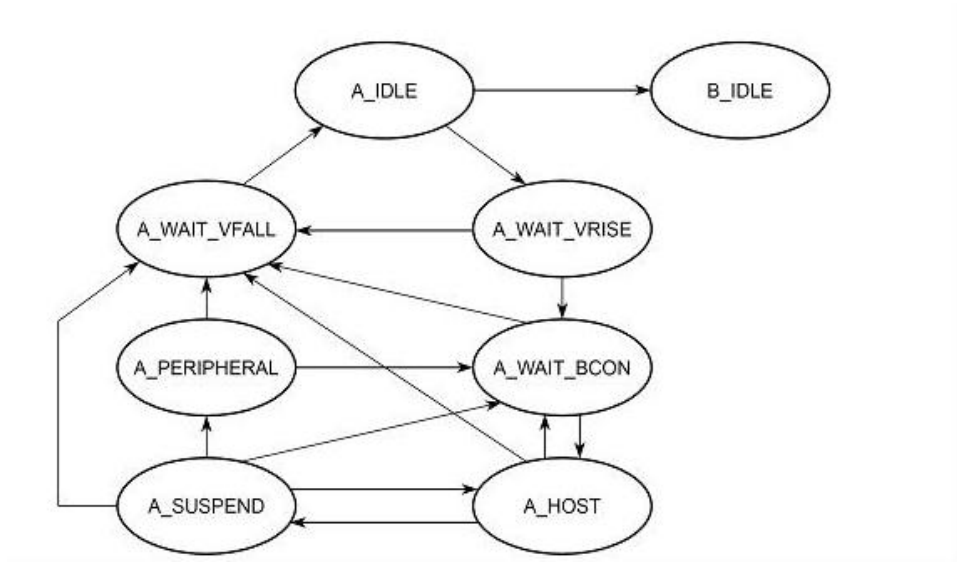


图14-15 A设备双重角色流程图

表14-9 A设备双重角色状态描述

状态	动作	应答
----	----	----

A_IDLE	如果产生ID中断。 电缆被拔出或插入了B类型的电缆。此时设备作为B设备。	跳转到B_IDLE
	如果A设备需要使用总线，或B设备请求SRP（A_SESS_VLD中断或插入或端口状态变化中断，检查数据线5-10毫秒脉冲）。	跳转到 A_WAIT_VRISE 打开DRV_VBUS
A_WAIT_VRISE	如果ID中断或100毫秒后A_VBUS_VLD为假。 电缆状态改变或A设备不支持B设备请求。	跳转到 A_WAIT_VFALL 关闭DRV_VBUS
	如果产生A_VBUS_VLD中断	跳转到 A_WAIT_BCON
A_WAIT_BCON	200毫秒内没有设备插入或ID中断。（如果需要可以一直等）	跳转到 A_WAIT_FALL 关闭DRV_VBUS
	A_VBUS_VLD中断或插入了B设备	跳转到A_HOST 打开Host Mode
A_HOST	列举设备以确定OTG支持	
	如果产生A_VBUS_VLD/中断或A设备结束或B设备被拔出。	跳转到 A_WAIT_VFALL 关闭Host Mode 关闭DRV_VBUS
	如果A设备会话结束或A设备允许B设备占有总线。	跳转到 A_SUSPEND
	ID中断或B设备被拔出	跳转到 A_WAIT_BCON
A_HOST	如果ID中断或150毫秒B设备断开时间到（超时的时间值可能更长）或产生A_VBUS_VLD中断。	跳转到 A_WAIT_VFALL 关闭DRV_VBUS
	如果使能HNP，并且在150毫秒内拔出B设备，然后B设备成为主机。	跳转到 A_PERIPHERAL 关闭Host Mode
	如果A设备要开始另一个会话	跳转到A_HOST
A_PERIPHERAL	如果产生ID中断或A_VBUS_VLD中断	跳转到 A_WAIT_VFALL 关闭DRV_VBUS
	如果总线空闲3-200毫秒	跳转到 A_WAIT_BCON 打开Host Mode
A_WAIT_VFALL	如果产生ID中断或（A_SESS_VLD/ & b_conn/）	跳转到A_IDLE

2. B设备操作的OTG双重角色

如果用 USB B 类型电缆或 Mini B 电缆接入到 USB 总线则被认为是 B 设备。B 设备的双重角色如图 14-16 和表 14-10 所示。

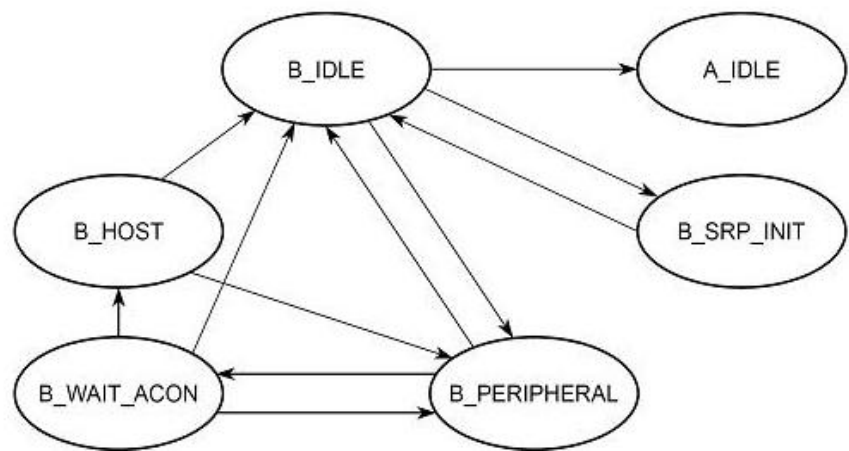


图14-16 B设备双重角色流程图

表14-10 B设备双重角色状态描述

状态	动作	应答
B_IDLE	如果产生ID中断 如果插入A型电缆并且设备需要作为A设备进行响应。	跳转到A_IDLE
	如果产生B_SESS_VLD中断。 如果A设备打开了V _{BUS} 并开始一个会话。	跳转到B_PERIPHERAL 打开DP_HIGH
	如果B设备需要使用总线并且总线空闲2ms， B_SESS_END位为1，B设备可请求SRP。	跳转到B_SRP_INIT Pulse CHRG_V _{BUS} Pulse DP_HIGH 5-10 ms
B_SRP_INIT	如果产生ID中断或SRP结束（SRP必须在100毫秒内结束）	跳转到B_IDLE
B_PERIPHERAL	如果使能了HNP，总线挂起并且B设备要求使用总线，这时B设备可成为主机。	跳转到B_WAIT_ACON 关闭DP_HIGH
B_WAIT_ACON	如果接入了A设备，即接收到了插入中断。	跳转到B_HOST 打开Host Mode
	如果产生ID中断或B_SESS_VLD中断 如果电缆改变或V _{BUS} 结束，主机不再支持。 跳转到B_IDLE	跳转到B_IDLE
	超过3.125ms或出现Resume。	跳转到B_PERIPHERAL
B_HOST	如果ID中断或B_SESS_VLD中断 如果电缆改变或V _{BUS} 结束，主机不再支持。	跳转到B_IDLE
	如果B设备应用结束或A断开。	跳转到B_PERIPHERAL

下面举例说明如何在 OTG 初始化中识别做为主机还是用作外设。为了支持 OTG 的 HNP 和 SRP 协议,测试用例中使用了 Max3353 芯片,在初始化 IIC 接口与 Max3353 通信(参见 Max3353Init 函数)后,通过调用 OTGGetRole 函数检测 USB 模块的工作模式,主函数根据该函数不同的返回

类型做不同的初始化。

```
//-----*
//函数名: OTGGetRole                                *
//功 能: 取得OTG设备的类型                          *
//参 数: 无                                          *
//返 回: USB模块工作类型                          *
//说 明:                                          *
//-----*
otg_role_t OTGGetRole(void)
{
    uint8 value;

    value = Max3353ReadByte(STAT_REG, MAX3353_I2C_ADDR);
    if (value & STAT_REG_ID_FLOAT)
        return(otgr_b);
    if (value & STAT_REG_ID_GND)
        return(otgr_a);
    return(otgr_a);
}
```

第15章 μ C/OS- II 在ColdFire上的移植与应用

嵌入式应用中使用实时操作系统（Real Time Operation System，RTOS），可以极大简化应用系统的软件设计，使嵌入式系统的稳定性和可靠性得到大幅度提高，最重要的是保证了控制系统的实时性，可以接近理想的实时。

本章主要以 32 位单片机 ColdFire 系列中的 MCF52233 为例，介绍嵌入式实时操作系统 μ C/OS-II 的移植和应用。首先概述了 RTOS 的概念和分类，在此基础上讨论了任务的含义和状态。然后给出了一个基于 μ C/OS-II 的应用实例。最后，介绍了如何在 MCF52233 上移植 μ C/OS-II。

15.1 实时操作系统概述

实时操作系统是计算机科学学科数十年发展的精美成果。使用实时操作系统可以提高开发效率、缩短开发周期、使程序更便于修改与维护。随着微电子技术的发展，微控制器的功能日益强大，加之嵌入式系统规模的增大，越来越多的嵌入式系统都使用了实时操作系统。嵌入式实时操作系统的使用使得实时应用程序的设计和扩展变得容易，无需大的改动就可以增加新的功能。

15.1.1 实时操作系统基本含义和要求

什么是实时？实时含有立即、及时之意。如果操作系统能使计算机及时响应外部事件的请求，并能及时控制所有实时设备与实时任务的协调运行，且能在一个规定的时间内完成对事件的处理，那么这种操作系统就叫实时操作系统（Real Time Operation System，RTOS）。对实时系统有两个基本要求：第一，实时系统的计算必须产生正确的结果，称之为逻辑或功能正确；第二，实时系统的计算必须在预定的周期内完成，称为时间正确。

嵌入式系统主要是对设备和装置进行控制的，在这些应用场合中，系统是否能及时快速地响应外部事件，常常是对系统的第一要求，因此嵌入式系统使用的操作系统大多是 RTOS。对于一个 RTOS 来说，就是要求它在现有的硬件条件下，在接收输入后要尽可能快地计算出输出结果，并且应用程序设计者在设计应用程序时，就能预先准确地计算出完成任务所需要的最大时间。为达到上述要求，RTOS 应满足三个条件：RTOS 必须是多任务系统；任务的切换时间应与系统中的任务数无关；中断延时的时间可预知并尽可能短。

15.1.2 任务和多任务

一个任务，也称为一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属于自己。通常一个系统被划分为多个任务（即“多任务”概念），每个任务都是整个应用的一部分，都被赋予一定的优先级，有自己的一套 CPU 寄存器和堆栈空间。

计算机在执行应用程序时，经常要用 I/O 设备进行数据的输入和输出，而 I/O 设备在工作时总是需要一段时间的，于是在 I/O 设备工作期间，如果 CPU 没有其他任务，那么就只能等待，这样就会使计算机运行一个应用程序所花的时间比较长，也就是说，这种系统的实时性比较差。如果把一个大的任务分解成多个可并行运行的小任务，那么在一个任务需要等待 I/O 时，就可以交出对 CPU 的使用权，让 CPU 去运行其他任务，这样就可以大大提高 CPU 的利用率，这种系统就是多任务系统。相对而言，多任务系统完成任务的时间会大大减少，从而提高了系统的实时性。除此之外，多任务系统还带来了另外一个优点，它可以让程序员把一个大的应用程序分成相对独立的多个任务来完成，从而给应用程序的设计和维护提供了极大的方便。鉴于多任务的诸多优点，现在应用于嵌入式系统的 RTOS 都是多任务系统。

多任务系统通过任务切换实现各个任务之间的调度运行。所谓任务切换是指当内核决定运行另外的任务时，它先把正在运行任务的当前状态（即 CPU 寄存器中的全部内容）保存在该任务自己的堆栈区中，然后把即将运行的任务的当前状况从该任务的堆栈区中装入 CPU 的寄存器并开始该任务的运行。

15.1.3 RTOS 的常用术语

RTOS 具备一些其它操作系统所不具备的基本概念。接下来介绍一些嵌入式实时操作系统的常用术语。

1. 代码临界段

代码临界段也称为临界区，指不可分割处理的代码。一旦这部分代码开始执行，则不允许任何中断打扰。为确保临界段代码的执行，在进入临界段之前要关中断，且临界段代码执行完以后要立即开中断。

2. 共享资源

任何被任务所占用的实体都可称之为资源，它可以是输入/输出设备，也可以是一个变量、结构体或一个数组等。可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时必须独占该资源。

3. 内核

多任务系统中，内核负责管理各个任务，或者说为各个任务分配 CPU 时间并负责任务间的通信。内核提供的基本服务是任务切换。

微内核是内核的一种精简形式。通常与内核集成在一起的系统服务层被分离出来，变成可以根据需求加入的选件，这样就可提供更好的可扩展性和更加有效的应用环境。对用微内核设计的系统进行升级，只要用新模块替换旧模块，不需改变整个操作系统。

4. 不可剥夺型内核/可剥夺型内核

不可剥夺型内核要求每个任务主动放弃 CPU 的使用权。不可剥夺型内核无需考虑函数可重入问题，不需使用信号量保护共享数据。

当系统响应时间很重要时，须使用可剥夺型内核。可剥夺型内核总是让就绪态中最高优先级的任务得到 CPU 的使用权，中断服务可以抢占 CPU。目前商用的嵌入式实时操作系统绝大多数采用的是可剥夺型内核。 $\mu\text{C}/\text{OS-II}$ 的内核就是可剥夺型内核。

5. 调度

系统内核根据任务状态、任务优先级等信息，决定将 CPU 资源分配给某个任务运行的过程称之为调度。

6. 死锁

死锁指两个或两个以上的任务无限期地互相等待对方释放其所占资源的情况。

7. 同步

在较为复杂的应用系统中通常都需要几个任务协同工作。而几个任务按照一定的逻辑顺序协同工作就称之为“同步”。通常使用信号量来解决任务之间的同步问题。

8. 任务间通信

任务间的中断服务程序与任务间的信息传递称为任务间通信。任务间信息传递有两个途径，通过全局变量传递，或发送消息给另一个任务。

9. 消息邮箱/消息队列

一个任务或中断服务程序通过内核服务将一条消息放进邮箱。另一个或多个任务则通过内核服务从邮箱中接收到该消息。

10. 信号量

信号量实际上是一种约定机制，在多任务内核中普遍使用。主要用于：

- (1) 控制共享资源的使用权；
- (2) 标志某事件的发送；
- (3) 使 2 个任务的行为同步。

信号量像一把钥匙，任务要运行下去就必须先拿到该钥匙。如果信号量被别的任务占用，那么该任务只得被挂起，直到信号量被当前使用者释放。

11. 时钟节拍

时钟节拍是特定的周期性中断。通过定时器产生周期性的中断供内核判断是否有更高优先级

的任务已进入就绪状态。

15.1.4 RTOS的现状

面对嵌入式系统的巨大应用领域和诱人的前景，世界上的各大软件开发公司、厂商甚至个人纷纷开发出各具特色的 RTOS。目前比较常见的有：WindRiver 公司的 VxWorks、pSOS、微软的 Windows CE、QNX 公司的 QNX，但是这些商用 RTOS 需要高昂的费用。针对这种情况，一些组织或个人也开发了一些免费的、源码开放的 RTOS 在互联网上发布，其中比较有名的是 μ CLinux、RTLlinux 和 μ C/OS-II。下面简单介绍几种具有代表性的 RTOS。

1. VxWorks

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种嵌入式实时操作系统，是 Tornado 嵌入式开发环境的关键组成部分。VxWorks 有良好的持续发展能力、高性能的内核以及友好的用户开发环境。VxWorks 的特点：可裁剪微内核结构；高效的任务管理；灵活的任务间通讯；微秒级的中断处理；支持 POSIX 1003.1b 实时扩展标准；支持多种物理介质及完整的 TCP/IP 网络协议等。

然而，由于 VxWorks 本身以及开发环境都是专有的，需要专门的技术人员掌握开发技术和维护，所以软件的开发和维护成本都非常高，支持的硬件数量有限。通常需花费 10 万元人民币以上才能建起一个可用的开发环境，对每一个应用一般还要另外收取版税。一般情况下，供应商不会提供 VxWorks 源代码，只提供二进制代码。

2. Windows CE

Windows CE 是一种主要针对小容量、移动式、智能化设备的实时嵌入式操作系统，为建立设备的动态应用程序和服务提供了功能丰富的操作平台。Windows CE 与 Windows 系列有较好的兼容性，是其推广的一大优势。它能在多种处理器体系结构上运行，通常适用于那些对内存占用空间具有一定限制的设备。Windows CE 从整体上为有限资源的平台设计提供了多线程、完整优先权、多任务等服务，它的模块化设计允许用户对其进行定制，定制范围可以从掌上电脑到专用工业控制器的用户电子设备。

从技术角度上讲，Windows CE 作为嵌入式操作系统有很多的缺陷：没有开放源代码，使应用开发人员很难实现产品的定制；在效率、功耗方面的表现并不出色；和 Windows 一样占用过多的系统内存，应用程序庞大；使用厂商需要支付版权许可费。

3. 嵌入式Linux

嵌入式 Linux 是嵌入式操作系统的一个新成员，其最大的特点是源代码公开并且遵循 GPL 协议。由于其源代码公开，人们可以任意修改嵌入式 Linux 源码以满足自己的应用。嵌入式 Linux 的特点：有大量的应用软件和开发工具可用，其中大部分都是遵从 GPL 的免费开放代码，用户无须为每例应用交纳许可证费，稍加修改后可以应用于自己的系统；在 Internet 时代尤为重要的优秀的网络功能；庞大的开发群体，无须专门的人才，只要懂 Unix/Linux 和 C 语言即可；稳定，内核精悍，运行所需资源少，十分适合嵌入式应用；和普通 Linux 一样，支持的硬件数量庞大。

运行嵌入式 Linux 的一个缺点是在嵌入式 Linux 体系上需要添加实时软件模块。而这些模块运行的内核空间正是操作系统实现调度策略、硬件中断异常和执行程序的部分。由于这些实时软件模块是在内核空间运行的，因此错误的代码可能会破坏操作系统从而影响整个系统的可靠性，

这对于实时应用将是一个致命的弱点。

4. μ C/OS- II

μ C/OS- II 是著名的源代码公开的实时内核，是专为嵌入式应用设计的，可用于 8 位、16 位、32 位单片机和数字信号处理器（DSP）。它在原版本 μ C/OS 的基础上做了重大改进与升级，经过近十年的使用和实践，已经有了许多成功应用该实时内核的实例。

由于 μ C/OS- II 仅是一个实时内核，这就意味着它不像其他实时操作系统那样提供给用户一些 API 函数接口，有很多工作需要用户自己去完成。

15.2 嵌入式实时操作系统 μ C/OS- II

15.2.1 μ C/OS- II 简介

μ C/OS- II 是 Jean J.Labrosse 于 1992 年编写的一个嵌入式多任务 RTOS。最早这个系统叫做 μ C/OS，后来经过近 10 年的应用和修改，在 1999 年 Jean J.Labrosse 推出了 μ C/OS- II，并在 2000 年得到了美国联邦航空管理局对用于商用飞机的、符合 RTCA DO-178B 标准的认证，从而证明 μ C/OS- II 具有足够的稳定性和安全性。

μ C/OS- II 是用 C 语言和汇编语言编写的。其中绝大部分代码都是用 C 语言编写，只有极少部分与处理器密切相关的代码是用汇编语言编写的，所以用户只要做很少的工作就可把它移植到各类 8 位、16 位和 32 位嵌入式处理器上。它的主要特点如下：

- 公开源代码：易于把操作系统移植到各个不同的硬件平台上。
- 可移植性：绝大部分源代码是用 C 语言写的，便于移植到其他微处理器上。
- 可裁剪性：有选择的使用需要的系统服务，以减少 μ C/OS- II 所需的存储空间。
- 抢占式实时内核：总是运行就绪状态下优先级最高的任务。
- 多任务：可管理 64 个任务，任务优先级必须是不同的，不支持时间片轮转调度法。
- 可确定性：函数调用与服务的执行时间可确定，不依赖于任务的多少。
- 实用性和可靠性：众多成功应用该内核的实例是其实用性和可靠性的最好证据。

μ C/OS- II 的构思巧妙，结构简洁精练，可读性很强，同时又具备了 RTOS 的全部内容，所以虽然它只是一个内核，但非常适合初次接触嵌入式 RTOS 的学生、嵌入式系统开发人员和爱好者学习，还可通过适当的扩展应用到实际系统中。

μ C/OS- II 一共有 16 个文件，如表 15-1 所示，如果读者希望了解关于 μ C/OS- II 的更多信息，可以访问 μ C/OS- II 的网站 <http://www.micrium.com/>。

表 15-1 μ C/OS-II 文件列表

分类	文件名	说明	备注	
与处理器无关的文件	OS_CORE.C	实时内核文件	μC/OS- II 的基本文件，一定要使用。	移植 μC/OS- II 时，这些文件无需修改
	OS_TIME.C	时钟节拍文件		
	OS_TASK.C	任务管理文件		
	OS_MEM.C	内存分配文件	内存动态分配和管理	
	OS_MBOX.C	消息邮箱文件	任务间通信使用	
	OS_SEM.C	信号量通信文件		
	OS_Q.C	消息队列文件		
	OS_FLAG.C	信号量集文件		

	OS_MUTEX.C	互斥信号量文件		
	uCOS_II.C	μC/OS-II 总文件，包含以上文件	μC/OS- II 总文件	
	uCOS_II.H	μC/OS- II 总头文件		
	OS_CFG.H	配置文件，可选择性的裁减功能	移植 μC/OS- II 时，须重新定义内核功能	
	INCLUDES.H	系统总头文件，包含C函数库等		
与处理器相关的文件	OS_CPU.H	处理器相关数据类型和宏定义文件	移植 μC/OS- II 时，需要修改为与特定处理器相关的代码	
	OS_CPU_A.S	处理器相关汇编文件		
	OS_CPU_C.C	处理器相关C文件		

15.2.2 μC/OS-II 与几种RTOS的比较

Windows CE 继承了传统 Windows 的特性（特别是图形界面），而且在其平台上可以使用 Windows 的编程工具（如 Visual Basic，Visual C++等）、同样的函数和同样的界面风格。绝大多数的应用软件只需简单的修改和移植就可以在 Windows CE 平台上继续使用，这使得 Windows CE 在推广上具有极大的优势。嵌入式 Linux 以其优越的性能和极强的性价比，在市场上赢得了众多用户的青睐。而 μC/OS-II 结构简洁精练，可读性很强，同时又具备了 RTOS 的全部内容。表 15-2 给出了这三种 RTOS 基本内容的比较情况。

表 15-2 三种 RTOS 基本内容的比较

RTOS名称	嵌入式Linux	Windows CE	μC/OS-II
大小	内核从几十KB到500KB，整个嵌入式环境最小才100KB左右，并且以后还将越来越小	核心一般占500KB的ROM和250KB的RAM。整个Windows CE包括硬件抽象层、Kernel、GDI、User、文件系统和数据库，大约共1.5MB	只需要几KB的RAM和ROM
可开发定制	用户可以方便的开发定制，可以自由卸载用户模块，不受限制	用户开发定制不方便，受Microsoft公司限制较多	用户的开发定制更灵活，不受任何限制
互操作性	互操作很强	比较强，.Wwdmvs CE可通过OEM的许可协议使用于其他设备	互操作很强
通用性	不仅适应于x86芯片，并且可以支持30多种CPU和多种硬件平台，开发和使用都很容易	适用于多种CPU和硬件平台	μC/OS-II已经移植到了几乎所有知名的CPU上
实时性	很好	比较好	很好
适用领域	由于Linux内核结构及功能上的优势，嵌入式Linux应用领域非常广泛，特别适用于进行信息家电的开发	应用领域较广，Windows CE是为新一代非传统的PC设备而设计的，这些设备包括掌上电脑、手持电脑以及车载电脑等	由于μC/OS-II是一个非常容易学习、结构简单、功能完备和实时性很强的RTOS，它不仅适合于广大嵌入式系统开发人员的学习和大专院校的教学科研，而且适用于开发那些对系统要求不是很苛刻的，而RAM和ROM有限的小型嵌入式系统设备

15.2.3 $\mu\text{C}/\text{OS-II}$ 中的任务

$\mu\text{C}/\text{OS-II}$ 的任务是一个无限的循环，一个任务可以有返回类型，有形式参数变量，但是任务是绝不会返回的。当任务完成以后，任务可以自我“删除”，即 $\mu\text{C}/\text{OS-II}$ 不理睬这个任务了，这个任务的代码也不会再运行。从存储结构来看，任务有三个组成部分：任务程序代码、任务堆栈和任务控制块。其中，任务控制块用来保存任务属性；任务堆栈用来保存任务工作环境；任务程序代码是任务的执行部分。 $\mu\text{C}/\text{OS-II}$ 可以管理多达 64 个任务，其中空任务（IDLE）和统计任务（STATISTICS）被系统占用。任务在建立时必须被赋予不同的优先级，优先级的数值越小，则任务的优先级越高，而 $\mu\text{C}/\text{OS-II}$ 总是会运行进入就绪状态中优先级最高的任务。就绪状态是任务的五种状态之一。根据是否占用 CPU，以及是否处于被中断、等待等情况，任务在 $\mu\text{C}/\text{OS-II}$ 中可能处于表 15-3 所示的五种状态的任一种。

表 15-3 任务的 5 种状态

任务的状态	说明
睡眠状态	任务只是以代码的形式驻留在程序空间，还没有交给 $\mu\text{C}/\text{OS-II}$ 管理时的情况叫做睡眠状态，即任务在没有任务控制块时的状态叫任务的睡眠状态。
就绪状态	如果系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，则任务就具备了运行的充分条件，这时任务的状态叫做就绪状态。
运行状态	处于就绪状态的任务如果经调度器判断获得了 CPU 的使用权，则任务就进入运行状态。任何时刻只能有一个任务处于运行状态，就绪的任务只有当所有优先级高于本任务的任务都转入等待状态时，才能进入运行状态。
等待状态	正在运行的任务需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给其他任务，而使任务进入等待状态。
中断服务状态	一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态。

图 15-1 给出了任务状态的转换过程。在任一时刻，任务的状态一定是这五种状态之一。

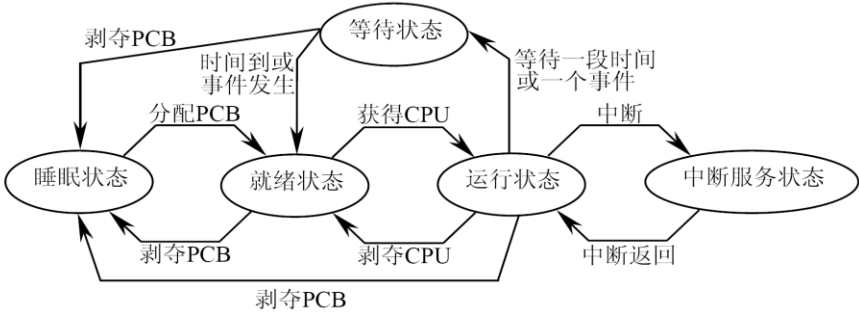


图15-1 任务状态转换图

15.2.4 $\mu\text{C}/\text{OS-II}$ 用户应用程序的一般结构

从程序代码来看，用户任务似乎就是一个 C 语言函数，但是这个函数又不是一般的 C 函数，它是一个线程。因此，它不是被主函数或其他函数调用的，而由 $\mu\text{C}/\text{OS-II}$ 负责来调度运行它们，主函数 main() 只负责创建和启动它们。从代码上来看，用户应用程序的代码大体结构如下：

```

//定义用户任务1
void MyTask1(void * pdata)
{
    进行准备工作的代码;
    for(;;)                                //无限循环
    {
        任务实体代码;
        调用系统延时函数;                //调用OSTimeDly()
    }
}
//定义用户任务2
void MyTask2(void * pdata)
{
    进行准备工作的代码;
    for(;;)                                //无限循环
    {
        任务实体代码;
        调用系统延时函数;                //调用OSTimeDly()
    }
}
//定义主函数
void main()
{
    .....
    OSInit();                            //初始化μC/OS-II
    .....
    OSTaskCreate(MyTask1, .....);        //创建用户任务1
    OSTaskCreate(MyTask2, .....);        //创建用户任务2
    .....
    OSStart();                            //启动任务
    .....
}

```

15.3 μC/OS-II 的任务划分

设计一个嵌入式应用系统都是从功能的需求分析开始, 然后进行系统设计, 即硬件系统设计和软件系统设计。在进行软件系统设计时, 是否采用 RTOS 将导致完全不同的设计方式。一个嵌入式应用系统的软件可以分为多个“程序块”。当不使用 RTOS 时, 将这些“程序块”进行分类组合, 就可以得到一个个“功能模块”, 其中有一个“监控模块”, 其他“功能模块”在“监控模块”的管理下运行。当使用 RTOS 时, 将这些“程序块”进行分类组合, 就可以得到一个个“任务”, 这些“任务”在 RTOS 的调度下运行。对一个基于 RTOS 的嵌入式应用系统进行“任务划分”, 是基于 RTOS 的软件系统设计的关键。任务划分是否合理将直接影响软件设计的质量。若任务划分合理, 软件设计将比较简洁、高效; 否则将可能比较繁杂, 甚至失败。

15.3.1 任务划分的目标

对具体的基于 RTOS 的嵌入式应用系统进行任务划分, 有很多不同的划分方案。首先要明确任务划分的目标, 才能对不同的方案进行比较, 选择最能实现目标的方案。一般基于下述目标划分任务。

首要目标是满足“实时性”要求。即使在最坏的情况下, 系统中所有对实时性有要求的功能

都能够正常实现。

任务数目合理：对于同一个应用系统，任务划分的数目较多时，每个任务需要实现的功能就相对简单，任务的设计也简单一些，但任务的调度与任务之间的通信就会比较频繁，使系统运行效率较低，资源开销增大。任务划分的数目较少时，每个任务需要实现的功能就相对复杂，但可免除不少通信工作，减少共享资源的数量，减轻 RTOS 的负担，减少资源开销。所以，如何合理的安排任务数目，充分发挥 RTOS 的优势，就显得尤为重要。

简化软件系统：一个任务要实现其功能，除了需要 RTOS 的调度功能支持外，还需要 RTOS 的其他服务功能支持，如时间管理功能、任务之间的同步功能、任务之间的通信功能、内存管理功能等。合理划分任务，可以减少对 RTOS 的服务要求，使 RTOS 的功能得到剪裁，简化软件系统，减少软件代码的规模。

降低资源需求：合理划分任务，减少或简化任务之间的同步和通信需求，就可以减少相应数据结构的内存规模，从而降低对系统资源的需求。

15.3.2 任务划分的方法

1. 硬件相关任务的划分

嵌入式系统的硬件系统一般由 MCU 加上各种硬件功能模块组成。有些硬件功能模块已经集成到 MCU 芯片内部，还有些硬件功能模块通过接口电路最终以输入/输出设备的形式与 MCU 打交道。各种输入/输出设备在 MCU 的控制下协调工作，而这种控制是通过软件来实现的，这些软件就是若干个硬件相关的任务。

与 MCU 执行指令的速度相比，输入/输出设备的运行速度都比较低，这导致相关的控制程序（即相关的任务程序）为了配合相关设备只能走走停停。这种运行状态正是 RTOS 进行任务调度所需要的。通过任务调度，使得各种设备可以并发运行。并发性是任务的基本特性，而硬件驱动程序具有“先天”的并发性。所以，把各个硬件功能模块的驱动程序分别封装为不同的任务是合理的。这样，我们就可以划分出第一批任务，如键盘任务、显示任务、输出控制任务和通信任务等。

2. 关键任务的划分

关键性是指某种功能在嵌入式应用系统中的重要性。如果这种功能不能正常实现，将造成重大影响，甚至引起灾难性的后果。实现关键功能的任务称为关键任务，关键任务必须得到运行机会，即使遗漏执行一次也不行。

如果将关键性操作与其他操作打包成一个任务，可能因为其他操作出现问题而导致整个任务的挂起，这样关键性操作就得不到执行，进而导致系统瘫痪。因此我们得到一条原则：对于关键功能，必须尽可能与其他功能割离，独立成为一个任务，通过通信方式再触发其他任务，完成其后续操作。

如果关键功能可以由中断触发，则可将该功能直接放入中断处理程序中完成。相对于普通的任务，中断处理程序可以看成具有超优先级的任务。为了提高系统的实时性，中断处理程序必须尽可能简短快速，以免影响其他关键任务的实时响应。如果关键任务有严格的实时性要求，则必须赋予它足够高的优先级，以便及时获得运行权；否则，没有必要为其分配高优先级。

3. 紧迫任务的划分

紧迫性是指某种功能必须在规定的时间内得以运行（及时运行），并在规定的时刻前执行完毕（按时完成），可见这类功能有严格的实时性要求。将关键任务划分出去以后，剩下的对实时性有严格要求的功能作为紧迫任务来处理。

大多数紧迫任务是由异步事件触发的，这些异步事件一般能够引发某种中断。在这种情况下，将紧迫任务安排在相应的中断处理程序中是最有效的方法。如果紧迫任务不能安排在中断处理程序中，则为它安排尽可能高的优先级是解决实时性的有效方法。

要达到“按时完成”的目的，必须使紧迫任务需要的执行时间尽可能短。其办法是对紧迫任务进行简化，尽可能剥离不太紧迫的操作，只剩下必须立刻做的操作，将被剥离的不太紧迫的操作另外封装为一个任务。

4. 数据处理任务的划分

用户应用程序中占用处理器时间最多的是各种数据处理程序，这种程序通常不只一个，它们分别为不同的功能服务，应该将这些程序划分出来，分别包装成不同的任务。由于这类任务需要占用较多的处理器时间，所以它们的优先级必须安排的比较低，以便可以使用其他任务剩余的处理器时间。

5. 功能聚合任务的划分

将关系密切的若干功能组合成为一个任务，达到功能聚合的效果。关系密切有两个含义：数据关系紧密和时序关联紧密。如果将关系密切的若干功能分别用不同的任务来实现，则需要大量的数据通信和同步通信，这对系统是一个很大的负担。而将关系密切的若干功能组合成为一个任务，相关数据为各个功能共享，同步机制通过合理安排各个功能在任务中的时序来实现。

6. 运行周期相同任务的划分

绝大多数系统功能都需要循环执行，如果是按固定的时间间隔循环执行，则认为这个功能具有周期性。将周期相同的功能组合在一起封装为一个任务，就可以避免一个时间事件触发多个任务，省去任务间的通信。

7. 顺序操作任务的划分

如果多个任务按固定顺序执行，相互之间完全没有并发性，则应该将这些功能组合为一个任务。如果将它们分别用不同的任务封装，则必须在这些任务之间进行频繁的同步通信，以完成规定的任务顺序操作。

8. 任务划分小结

通过以上分析，可以对任务划分小结如下：

首先，以 MCU 为中心，将各种硬件功能相关的驱动程序分别划分为独立的任务。

对于关键功能，将其最重要的部分剥离出来，用一个独立的任务或中断处理程序完成，剩余部分用另一个任务实现。

对于紧迫功能，将其最重要的部分剥离出来，用一个独立的高优先级任务或中断处理程序完成，剩余部分用另一个任务实现。

对于既属于关键功能又属于紧迫功能的模块，按紧迫功能处理。

将占用处理器时间较多的数据处理程序划分出来，封装为低优先级任务。

将关系密切的若干功能组合成一个任务，达到功能聚合的效果。

将运行周期相同的功能组合成一个任务，从而简化任务间的通信。

将若干按固定顺序执行功能组合成一个任务，从而简化任务间同步通信。

15.4 μ C/OS-II 在 MCF52233 上的应用实例

15.4.1 基于 μ C/OS-II 的应用实例

通过上一节的描述，读者对于如何划分任务有了一定的了解，下面通过一个基于 μ C/OS-II 的应用实例来具体阐述任务划分的方法，为学习 μ C/OS-II 系统开发的读者提供参考。

该实例完成的功能是读取 PC 机发送来的数据，根据该数值控制小灯闪烁次数。MCU 从串口接收 PC 机发送的一个字节的数据，控制小灯闪烁相应次数，向 PC 机回送该数据并在 PC 机上显示出来。在这个实例中，首先要保证 μ C/OS-II 在 MCF52233 正确移植，有关移植相关问题请参考下一节。

根据与硬件相关任务划分的原则，我们把串口发送和控制小灯闪烁定义为两个独立的任务，利用串行接收中断来接收 PC 机发送的一个字节数据。由于这里只接收一个字节数据，过程比较简单，可以把该过程安排在串行接收中断处理程序中完成。如果接收的数据较多，接收到数据后需要对数据进行处理，根据“中断处理过程尽可能简洁”的原则，最好不要将全部内容安排在串行接收中断处理程序中完成。可以安排一个与串行接收中断处理相关联的任务来完成接收到数据后的一些工作。在串行接收中断接收到数据之后，我们将数据放入全局变量中。串行发送任务将该变量发送给 PC 机，小灯闪烁任务要读取该数据来控制小灯闪烁次数。

串行接收中断和串行发送任务、小灯闪烁任务间存在资源同步问题，采用了信号量机制。使用信号量 UartSem 来同步接收中断与串行发送任务，使用信号量 RandomSem 来同步接收中断与小灯闪烁任务间的操作。相关任务和串行接收中断程序片段如下，详细代码可参见网络光盘。

```
//-----*
//函数名: UART0_handler                      *
//功 能: UART0中断处理函数                  *
//参 数: 无                                *
//返 回: 无                                *
//调用函数: DisableInterrupts关总中断        *
//          EnableInterrupts开总中断          *
//          UartReN串口接收N个字节            *
//          OSSemPost释放信号量              *
//-----*
__interrupt__
void UART0_handler(void)
{
    //关总中断
    DisableInterrupts();
    //将收到的字节存在Times中
    if (UartReN(1,&Times)==0)
    {
        OSSemPost(RandomSem);    //释放信号量RandomSem
        OSSemPost(UartSem);      //释放信号量UartSem
    }
    //开总中断
}
```

```

    EnableInterrupts();
}

//-----*
//任务名:UART                                *
//功 能: 串口发送任务子程序, 使用串口发送接收到的数据        *
//参 数: 无                                *
//返 回: 无                                *
//调用函数:OSSemPend等待信号量                                *
//          UartSend1串行发送1个字节                            *
//          OSTimeDly任务延时函数                              *
//-----*
void UART(void *data)
{
    INT8U err;
    data = data;
    while(1)
    {
        //等待信号量UartSem
        OSSemPend(UartSem, 0, &err);
        //通过串口发送数据Times至PC
        UartSend1(Times);
        OSTimeDly(5); // 延迟5个时钟节拍
    }
}

//-----*
//任务名:Light                                *
//功 能: 小灯任务子程序, 根据串口任务接收到的数据控制小灯闪烁次数 *
//参 数: 无                                *
//返 回: 无                                *
//调用函数: EnableTIN0Int                                *
//          OSSemPend请求信号量                            *
//          LightControl控制小灯状态函数                    *
//          OSTimeDly任务延时函数                              *
//-----*
void Light(void *data)
{
    INT8U i;
    INT8U err;

    data = data;
    while (1)
    {
        //等待信号量RandomSem
        OSSemPend(RandomSem, 0, &err);
        //通过调用LightControl函数控制小灯闪烁Times次
        for(i=0;i<Times-'0';i++)
        {
            LightControl(0x08); //小灯亮
            OSTimeDly(5); // 延迟5个时钟节拍
            LightControl(0x00); //小灯暗
            OSTimeDly(5); // 延迟5个时钟节拍
        }
    }
}

```

15.4.2 与无RTOS实例的比较

RTOS 给设计者提供一个操作的平台,用户只需根据实际的需要定制和划分一些实际的任务,就可以完成嵌入式应用系统的设计,简化了整个设计的过程。由于采用了 RTOS,系统的实时性、稳定性和可靠性都得到了很好的保证,需要设计者考虑的问题就会大大地减少,开发周期也会相应的缩短。

但是,RTOS 并不是一把万能钥匙,有了 RTOS,一切都迎刃而解的想法是错误的。设想一下,实现同样的功能用 RTOS 和不用 RTOS,对 CPU 来说意味着什么?无疑是增大了开销,对同一个 MCU 来说,干的事情越多就越容易出错,这是永远不变的道理。因此,在使用 RTOS 的时候要慎重,如果一些低档的微处理器用了 RTOS,就会使本来简单的问题复杂化。RTOS 适用于较高性能的微处理器和嵌入式应用系统比较复杂的情况,它在用户与 MCU 之间建立了桥梁。用户只需要知道一些接口函数的使用方法,通过构建自己的应用任务,就可以将对任务的管理交由 RTOS 完成。对于要深入理解和应用 RTOS 的读者而言,则有必要了解 RTOS 移植的过程和方法。

15.5 μ C/OS-II 在 MCF52233 上的移植

移植就是指使某个操作系统可以在某个微处理器或者微控制器上运行。虽然 μ C/OS-II 的大部分源代码是用 C 语言写成的,但一些与处理器相关的代码仍是用 C 语言和汇编语言完成的。 μ C/OS-II 在最初设计的时候就充分考虑到系统的可移植性,90%的代码是用 C 语言编写的,只要有相应的 C 编译器就可以将这些代码移植到目标处理器上,因此移植的绝大部分工作集中在多任务切换的实现上。这部分主要是进行处理器状态的保存、切换、恢复,与具体芯片的体系结构直接相关,因此代码主要用处理器的汇编语言来实现。

在本书中,采用 μ C/OS-II V286 版本的内核,在移植过程中无需修改的文件为:UCOS_II.H、UCOS_II.C、OS_CORE.C、OS_TASK.C、OS_TIME.C、OS_SEM.C、OS_MBOX.C、OS_MEM.C、OS_MUTEX.C、OS_Q.C、OS_FLAG.C。需要修改的文件为:OS_CFG.H、INCLUDES.H、OS_CPU.H、OS_CPU_C.C、OS_CPU_A.S。其中 OS_CFG.H 文件用于配置 μ C/OS-II 内核的属性;而 OS_CPU_A.S 为使用汇编语言编写的与硬件相关的代码,在这里不做详细说明。下面具体讨论 μ C/OS-II 在 MCF52233 上的移植的过程,具体代码可参见网络光盘。在其他 ColdFire 系列芯片上的移植方法与此类似。

15.5.1 定义内核的大小和功能

1. 修改 includes.h 文件

该文件定义为整个系统的总头文件,可以包含 C 语言库的头文件、 μ C/OS-II 内核的头文件、用户定义的头文件。

具体的修改代码如下:

```
//1、包含通用头文件
#include "mcf52233.h" //CodeFire 52233 MCU映像寄存器名定义
#include "type.h" //类型定义头文件
#include "config.h" //配置头文件
#include "isr.h" //中断处理头文件
```

```
//2、包含面向硬件对象头文件
#include "Light.h"      //开关量头文件
#include "UART.h"      //UART头文件
#include "PIT.h"

#include "os_cpu.h"     //与CPU相关的头文件
#include "os_cfg.h"     //μC/OS-II配置文件
#include "ucos_ii.h"    //μC/OS-II总头文件

INT8U Times;
```

2. 修改OS_CPU.h文件

(1) 数据类型定义

由于不同的处理器有不同的字长，μC/OS-II 的移植需要重新定义一系列的数据结构，保证其可移植性。μC/OS-II 的代码不直接使用 C 语言的 short, int 和 long 等数据类型，因为它们是与编译器相关的，不可移植。

由于处理器在任务切换时，所有现场寄存器都将保存在当前运行任务的堆栈中，因此 OS_STK 数据类型应该与处理器的寄存器长度一致（ColdFire 系列为 32 位），且所有与任务相关的堆栈都必须用 OS_STK 声明。针对 MCF52233 芯片，状态寄存器长度应设置为 16 位。

```
.....
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned short INT16U;
typedef signed   short INT16S;
typedef unsigned int   INT32U;
typedef signed   int   INT32S;
typedef float        FP32;
typedef double       FP64;

typedef unsigned int   OS_STK;    //ColdFireCPU中的堆栈入口数据长度32位
typedef unsigned short OS_CPU_SR; //定义了ColdFire CPU中的SR寄存器16位
.....
```

(2) 代码临界区

与所有实时内核一样，μC/OS-II 需要先禁止中断再访问代码的临界区，并在访问完毕后重新允许中断。这就使得 μC/OS-II 能够保护临界区代码免受多任务或中断服务例程的破坏，OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()就是用于实现开关中断的。

具体的代码如下：

```
.....
//定义进入临界区代码的方法
#define OS_CRITICAL_METHOD    2

#if OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL()  asm(move.w #0x2700, sr) //禁止中断
#define OS_EXIT_CRITICAL()   asm(move.w #0x2000, sr) //使能中断
#endif
.....
```

(3) 定义任务切换指令

如果使用的 CPU 有软中断指令，那么在这个宏中封装一个软中断指令即可；如果 CPU 没有提供软中断指令，那么就可以试一试在 OS_TASK_SW() 中封装其他使 PC 压栈的指令，如调用指令 Trap，在使用前需要清除 Trap 指令所对应硬件的操作：1→SR 的 S 位；SP-4→SP；nextPC→(SP)；SP-2→SP；SR→(SP)；SP-2→SP；Format/Offset→(SP)。而在调用 RTE 指令时执行的是与 Trap 相反的动作：(SP+2)→SR；(SP+4)→PC；SP+8→SP。

在任务切换的时候，往往要保存当前任务的堆栈指针，以便任务可以正常恢复运行。

```
.....
//定义任务切换的指令
#define OS_TASK_SW()      asm(TRAP #14;)    //使用Trap #14执行任务级切换
```

15.5.2 与硬件相关的代码

1. 修改 OS_CPU_C.C 文件

(1) 改写任务堆栈初始化函数 OSTaskStkInit

这个函数虽然是用 C 语言写的，但是这是一个与 CPU 硬件相关的函数。这个函数初始化任务的堆栈，由建立任务的函数 OSTaskCreate() 或扩展的建立任务函数 OSTaskCreateExt() 调用。

```
.....
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos,
                      INT16U opt)
{
    INT32U *pstk32;

    opt = opt;
    switch ((INT32U)ptos & 0x00000003) {
        case 0:
            pstk32 = (OS_STK *) ((INT32U)ptos + 0);
            break;
        case 1:
            pstk32 = (OS_STK *) ((INT32U)ptos - 1);
            break;
        case 2:
            pstk32 = (OS_STK *) ((INT32U)ptos - 2);
            break;
        case 3:
            pstk32 = (OS_STK *) ((INT32U)ptos - 3);
            break;
    }

    *pstk32 = 0;
    *--pstk32 = (INT32U)p_arg;           // p_arg
    *--pstk32 = (INT32U)task;           // 任务返回地址
    *--pstk32 = (INT32U)task;
    *--pstk32 = (INT32U)(0x40000000 | OS_INITIAL_SR); //格式和状态寄存器
                                           //保存寄存器
    *--pstk32 = (INT32U)0x00A600A6L;    //A6
    *--pstk32 = (INT32U)0x00A500A5L;    //A5
    *--pstk32 = (INT32U)0x00A400A4L;    //A4
    *--pstk32 = (INT32U)0x00A300A3L;    //A3
    *--pstk32 = (INT32U)0x00A200A2L;    //A2
    *--pstk32 = (INT32U)0x00A100A1L;    //A1
    *--pstk32 = (INT32U)p_arg;          //A0
}
```

```

    *--pstk32 = (INT32U) 0x00D700D7L;           //D7
    *--pstk32 = (INT32U) 0x00D600D6L;           //D6
    *--pstk32 = (INT32U) 0x00D500D5L;           //D5
    *--pstk32 = (INT32U) 0x00D400D4L;           //D4
    *--pstk32 = (INT32U) 0x00D300D3L;           //D3
    *--pstk32 = (INT32U) 0x00D200D2L;           //D2
    *--pstk32 = (INT32U) 0x00D100D1L;           //D1
    *--pstk32 = (INT32U) p_arg;                  //D0
    return ((OS_STK *)pstk32);                  //返回栈顶
}
.....

```

(2) 加入让优先级最高的任务运行的函数 OSStartHighRdy

这一段代码是用汇编写的，不难看懂。将 CPU 的堆栈指针 SP 的值，改为优先级最高的就绪态任务的堆栈指针的值，该值是其任务控制块的第一个参数；然后将该任务的状态字由非运行态“0”，改为运行态“1”；恢复任务代码所在的存储器页面的值以换入即将运行的任务代码，换出被挂起的任务的代码；然后执行中断返回指令 RTE 以开始运行这个任务。前面的接口函数，使用户可以插入一些自定义的代码。

```

_OSStartHighRdy:
    JSR        _OSTaskSwHook

    MOVEQ.L    #1, D4                          //OSRunning = TRUE;说明多任务运行
    MOVE.B     D4, _OSRunning

    MOVE.L     (_OSTCBHighRdy), A1              //取得最高优先级任务的TCB地址
    MOVE.L     (A1), A7                        //取得其堆栈地址

    MOVEM.L    (A7), D0-D7/A0-A6               //保存寄存器
    LEA        60(A7), A7                     //修改堆栈指针

    RTE

```

(3) 修改任务切换函数 OSCtxSw, OSIntCtxSw

中断级的任务切换和任务级的任务切换非常相似，在任务级做切换，用的是 Trap，在 Trap 中断中跳转到中断处理函数 OSCtxSw 中运行。中断级的任务切换是这样发生的，在每次中断服务结束时，中断服务子程序会调用 OSIntExit()，如果 OSIntExit()发现终端激活了更高优先级的任务，则调用中断级的任务切换函数。

```

_OSCtxSw:
    LEA        -60(A7), A7
    MOVEM.L    D0-D7/A0-A6, (A7)              //保存寄存器到当前任务的堆栈
    JSR        _OSTaskSwHook

    MOVE.L     (_OSTCBCur), A1                 //在当前任务TCB中保存栈顶指针
    MOVE.L     A7, (A1)

    MOVE.L     (_OSTCBHighRdy), A1            // OSTCBCur = OSTCBHighRdy
    MOVE.L     A1, (_OSTCBCur)
    MOVE.L     (A1), A7                      //取得将要运行任务的堆栈

    MOVE.B     (_OSPrioHighRdy), D0           // OSPrioCur = OSPrioHighRdy
    MOVE.B     D0, (_OSPrioCur)

```

	MOVEM. L	(A7), D0-D7/A0-A6	//恢复CPU寄存器
	LEA	60(A7), A7	
RTE			
_OSIntCtxSw:			
	JSR	_OSTaskSwHook	
	MOVE. B	(_OSPrioHighRdy), D0	//OSPrioCur = OSPrioHighRdy
	MOVE. B	D0, (_OSPrioCur)	
	MOVE. L	(_OSTCBHighRdy), A1	//OSTCBCur = OSTCBHighRdy
	MOVE. L	A1, (_OSTCBCur)	
	MOVE. L	(A1), A7	//SP =
		OSTCBHighRdy->OSTCBStkPtr	
	MOVEM. L	(A7), D0-D7/A0-A6	//恢复CPU寄存器
	LEA	60(A7), A7	
RTE			//运行任务

2. 配置系统中断

在这里我们把与系统中断相关的代码放在 `ISR.C` 文件中。该文件主要包含了系统的中断函数，即软中断和硬件中断。其中软中断主要用于任务级任务之间的切换；而硬件中断中的定时器中断主要用于产生系统的时钟节拍。`ISR.C` 的具体的代码如下：

```
//-----*
//函数名: PIT1_handler                                *
//功 能: PIT1中断处理程序                            *
//参 数: 无                                           *
//返 回: 无                                           *
//-----*

__interrupt__
void PIT0_handler(void)
{
    //关总中断
    DisableInterrupts();
    MCF_PIT0_PCSR |= MCF_PIT_PCSR_PIF;    //清零事件标志
    OSTimeTick();
    OSIntExit();
    //开总中断
    EnableInterrupts();
}
```

在为内核修改了上述代码以后，用户就可以为自己的项目编写实际的代码了。具体应用实例和任务划分方法可参考本章的 15.4 小节。

第16章 基于硬件构件的嵌入式系统开发方法

嵌入式系统开发包括硬件设计和软件设计两个过程。为了提高硬件电路和底层驱动程序的可移植性,减少软硬件设计工作中的重复,本章提出了基于硬件构件的嵌入式系统开发方法。先给出了硬件构件的概念,接着介绍了基于硬件构件的原理图设计的基本原则、PCB 设计的基本原则以及底层驱动程序的编写方法,最后给出应用实例。

16.1 嵌入式系统开发所遇到的若干问题

自从1974年第一款微处理器芯片问世以来,嵌入式系统应用已深入到军事、航空航天、通信、家电等各个领域。近年来,随着微控制器(MCU)内部Flash存储器可靠性提高及擦写方式的变化,内部RAM及Flash存储器容量的增大,以及外部模块内置化程度的提高,设计复杂性、设计规模及开发手段已经发生了根本变化。

在嵌入式系统发展的最初阶段,嵌入式系统开发(包括硬件和软件设计)通常是由一个工程师来承担,软件在整个工作中的比例很小。随着时间的推移,硬件设计变得越来越复杂,软件的份量也急剧增长,嵌入式开发人员也由一人发展为由若干人组成的开发团队。

目前,嵌入式系统开发主要存在以下两大问题:

问题 1: 硬件设计缺乏重用支持

导致硬件设计缺乏重用支持的主要原因是:目前缺少可供硬件设计工程师们共同遵守的设计规范。设计人员往往是凭借个人工作经验和习惯的积累进行系统硬件电路的设计。在开发完一个嵌入式应用系统再进行下一个应用开发时,硬件电路原理图往往需要从零开始,重新绘制;或者在一个类似的原理图上修改,但往往又很麻烦,容易出错。

问题 2: 驱动程序可移植性差

驱动程序的开发在嵌入式系统的开发中具有举足轻重的地位。驱动程序的好坏直接关系着整个嵌入式系统的稳定性和可靠性。然而,开发出完备、稳定的驱动程序并非易事。长期以来,开发人员在编写驱动程序时缺少软件工程思想的支撑,软、硬件设计过程孤立,造成与硬件密切相关的底层软件缺乏通用性,可移植性和可复用性较差,开发过程中缺少标准化、文档化的管理,给开发人员之间的交流以及日后系统的维护带来很大的困难。

上述两个问题导致的结果是系统开发周期长,效率低。下面提出的基于硬件构件的软硬件设计思想可在一定程度上解决这些问题。

16.2 嵌入式硬件构件的基本思想与应用方法

什么是嵌入式硬件构件?它与我们常说的硬件模块有什么不同?

众所周知,嵌入式硬件是任何嵌入式产品不可分割的重要组成部分,是整个嵌入式系统的构建基础,嵌入式应用程序和操作系统都运行在特定的硬件体系上。一个以MCU为核心的嵌入式系统通常包括以下硬件模块:电源、写入器接口电路、硬件支撑电路、UART、USB、Flash、A/D、D/A、LCD、键盘、传感器输入电路、通信电路、信号放大电路、驱动电路等模块。其中有些模

块集成在 MCU 内部，有的位于 MCU 之外。

与硬件模块的概念不同，嵌入式硬件构件是指将一个或多个硬件功能模块、支撑电路及其功能描述封装成一个可重用的硬件实体，并提供一系列规范的输入/输出接口。由定义可知，传统概念中的硬件模块是硬件构件的组成部分，一个硬件构件可能包含一个硬件功能模块，也有可能包含多个。

根据接口之间的生产消费关系，接口可分为提供接口和需求接口两类。根据所拥有接口类型的不同，硬件构件分为核心构件、中间构件和终端构件三种类型。核心构件只有提供接口，没有需求接口。也就是说，它只为其它硬件构件提供服务，而不接受服务。在以单 MCU 为核心的嵌入式系统中，MCU 的最小系统就是典型的核心构件。中间构件既有需求接口又有提供接口，即它不仅能够接受其它构件提供的服务，而且也能为其它构件提供服务。而终端构件只有需求接口，它只接受其它构件提供的服务。这三种类型构件的区别如表 16-1 所示。

表 16-1 核心构件、中间构件和终端构件的区别

类型	需求接口	提供接口	举例
核心构件	无	有	MC68HC908GP32最小系统、MCF52233最小系统
中间构件	有	有	电源控制构件、可变频率产生构件、电平转换构件
终端构件	有	无	LCD构件、LED构件、键盘构件

利用硬件构件进行嵌入式系统硬件设计之前，应该进行硬件构件的合理划分，按照一定规则，设计与系统目标功能无关的构件个体，然后进行“组装”，完成具体系统的硬件设计。这样，这些构件个体也可以被组装到其他嵌入式系统中。在硬件构件被应用到具体系统中后，设计人员需要做的仅仅是为需求接口添加接口网标。

16.3 基于硬件构件的嵌入式系统硬件电路设计原则

16.3.1 设计时需要考虑的基本问题

1. MCU 的选择

选择 MCU 时要考虑 MCU 所能够完成的功能、MCU 的价格、功耗、供电电压、I/O 口电平、管脚数目以及 MCU 的封装等因素。MCU 的功耗可以从其电气性能参数中查到。供电电压有 5V、3.3V 以及 1.8V 超低电压供电模式。为了能合理分配 MCU 的 I/O 资源，在 MCU 选型时可绘制一张引脚分配表。

2. 电源

(1) 考虑系统对电源的需求，例如系统需要几种电源，如 24V、12V、5V 或者 3.3V 等，估计各需要多少功率或最大电流 (mA)。在计算电源总功率时要考虑一定的余量，可按公式“电源总功率 = 2 × 器件总功率”来计算。

(2) 考虑芯片与器件对电源波动性的需求。一般允许电源波动幅度在 ±5% 以内。对于 A/D 转换芯片的参考电压一般要求 ±1% 以内。

(3) 考虑工作电源是使用电源模块还是使用外接电源。

3. 普通I/O口

(1) 上拉、下拉电阻：考虑用内部或者外部上/下拉电阻，内部上/下拉阻值一般在 $700\ \Omega$ 左右，低功耗模式不宜使用。外部上/下拉电阻根据需要可选 $10\text{K}\ \Omega \sim 1\text{M}\ \Omega$ 之间。

(2) 开关量输入：一定要保证高低电压分明。理想情况下高电平就是电源电压，低电平就是地的电平。如果外部电路无法正确区分高低电平，但高低仍有较大压差，可考虑用 A/D 采集的方式设计处理。对分压方式中的采样点，要考虑分压电阻的选择，使该点通过采样端口的电流不小于采样最小输入电流，否则无法进行采样。

(3) 开关量输出：基本原则是保证输出高电平接近电源电压，低电平接近地电平。I/O 口的吸纳电流一般大于放出电流。对小功率元器件控制最好是采用低电平控制的方式。一般情况下，若负载要求小于 10mA ，则可用芯片引脚直接控制；电流在 $10\sim 100\text{mA}$ 时可用三极管控制， $100\text{mA} \sim 1\text{A}$ 时用 IC 控制；更大的电流则适合用继电器控制，同时需要使用光电隔离芯片。

4. A/D电路与D/A电路

(1) A/D 电路：要清楚前端采样基本原理，对电阻型、电流型和电压型传感器采用不同的采集电路。如果采集的信号微弱，还要考虑如何进行信号放大。

(2) D/A 电路：考虑 MCU 的引脚通过何种输出电路控制实际对象。

16.3.2 硬件构件化电路原理图绘制规则

1. 硬件构件设计的通用规则

在设计硬件构件的电路原理图时，需遵循以下基本原则：

(1) 元器件命名格式：对于核心构件，其元器件直接编号命名，同种类型的元件命名时冠以相同的字母前缀。如电阻名称为 R1、R2 等，电容名称为 C1、C2 等，电感名称为 L1、L2 等，指示灯名称为 E1、E2 等，二极管名称为 D1、D2 等，三极管名称为 Q1、Q2 等，开关名称为 K1、K2 等。对于中间构件和终端构件，其元器件命名格式采用“构件名-标志字符?”。例如，LCD 构件中所有的电阻名称统一为“LCD-R?”，电容名称统一为“LCD-C?”。当构件原理图应用到具体系统中时，可借助理图编辑软件为其自动编号。

(2) 为硬件构件添加详细的文字描述，包括中文名称、英文名称、功能描述、接口描述、注意事项等，以增强原理图的可读性。中英文名称应简洁明了。

(3) 将前两步产生的内容封装在一个虚线框内，组成硬件构件的内部实体。

(4) 为该硬件构件添加与其它构件交互的输入/输出接口标识。接口标识有两种：接口注释和接口网标。它们的区别是：接口注释标于虚线框以内，是为构件接口所作的解释性文字，目的是帮助设计人员在使用该构件时理解该接口的含义和功能；而接口网标位于虚线框之外，且具有电气特性。为使原理图阅读者便于区分，接口注释采用斜体字。

在进行核心构件、中间构件和终端构件的设计时，除了要遵循上述的通用规则外，还要兼顾各自的接口特性、地位和作用。

2. 核心构件设计规则

设计核心构件时，需考虑的问题是：“核心构件能为其他构件提供哪些信号?”。核心构件其实就是某型号 MCU 的最小系统。核心构件设计的目标是：凡是使用该 MCU 进行硬件系统设计时，核心构件可以直接“组装”到系统中，无须任何改动。为了实现这一目标，在设计核心构件的实

体时必须考虑细致、周全，包括稳定性、扩展性等，封装要完整。核心构件的接口都是为其它构件提供服务的，因此接口标识均为接口网标。在进行接口设计时，需将所有可能使用到的引脚都标注上接口网标（不要考虑：核心构件将会用到怎样的系统中去）。若同一引脚具有不同功能，则接口网标依据第一功能选项命名。遵循上述规则设计核心构件的好处是：当使用核心构件和其它构件一起组装系统时，只要考虑其它构件将要连接到核心构件的哪个接口（不是考虑：核心构件将要连接到其它构件的哪个接口），这也符合设计人员的思维习惯。在第 2 章，图 2-9 是以 ColdFire 系列 80 引脚 LQFP 封装的 MCF52233 芯片为 MCU 的核心构件原理图。

3. 中间构件设计规则

设计中间构件时，需考虑的问题是：“中间构件需要接受哪些信号，以及提供哪些信号？”。中间构件是核心构件与终端构件之间通信的桥梁。在进行中间构件的实体封装时，实体的涉及范围应从构件功能和编程接口两方面考虑。一个中间构件应具有明确的且相对独立的功能，它既要有接受其它构件提供的服务的接口，即需求接口，又要有为其他构件提供服务的接口，即提供接口。描述需求接口采用接口注释，描述提供接口采用接口网标。当中间构件被作为一个“零件”组装到具体系统中时，设计人员只要考虑为构件提供服务的来源，为接口注释添加对应的应用网标即可，其它内容无须关心或改动。

中间构件的接口数目没有核心构件那样丰富。为直观起见，设计中间构件时，将构件的需求接口放置在构件实体的左侧，提供接口放置在右侧。接口网标的命名规则是：构件名称-引脚信号/功能名称。而接口注释名称前的构件名称可有可无，它的命名隐含了相应的引脚功能。

电源控制构件（如图 16-1 所示）、可变频率产生构件（如图 16-2 所示）是常用的中间构件。图 16-1 中的 *Power-IN* 和图 16-2 中的 *SDI*、*SCK* 和 *SEN* 均为接口注释，*Power-OUT* 和 *LTC6903-OUT* 为接口网标。

4. 终端构件设计规则

设计终端构件时，需考虑的问题是：“终端构件需要什么信号才能工作？”。终端构件是嵌入式系统中最常见的构件。终端构件没有提供接口，它仅有与上一级构件交互的需求接口，因而接口标识均为斜体标注的接口注释。LCD (YM1602C) 构件（如图 16-3 所示）、LED 构件、指示灯构件以及键盘构件（如图 16-4）等都是典型的终端构件。

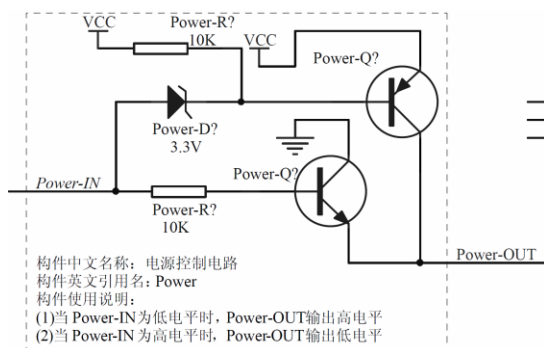


图16-1 电源控制构件

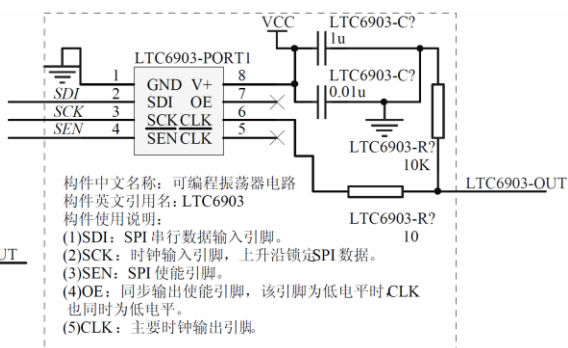
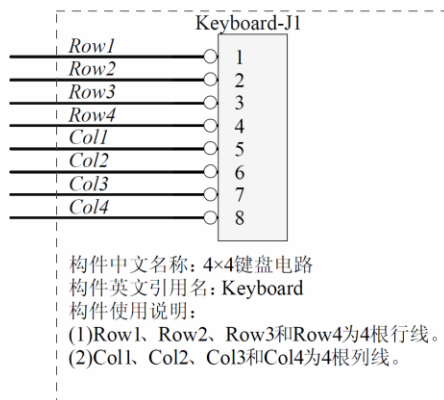
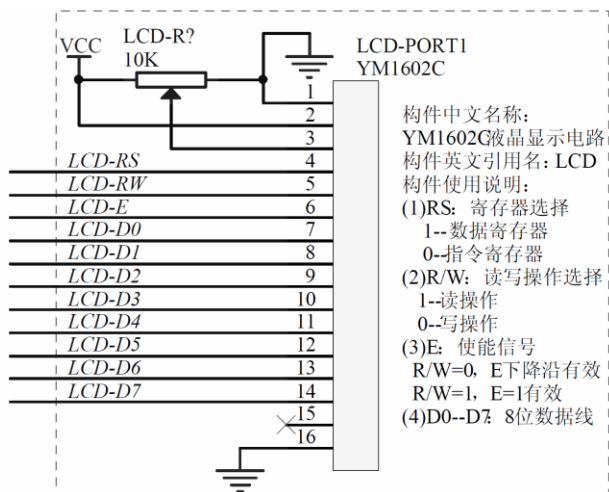


图16-2 可变频率产生构件



5. 使用硬件构件组装系统的方法

对于核心构件，在应用到具体的系统中时，不必作任何改动。具有相同 MCU 的应用系统，其核心构件完全相同。对于中间构件和终端构件，在应用到具体的系统中时，仅需为需求接口添加接口网标，在不同的系统中，接口网标名称不同，但构件实体内部完全相同。

使用硬件构件化思想设计嵌入式硬件系统的过程是：

- (1) 根据系统的功能划分出若干个硬件构件。
- (2) 将所有硬件构件原理图“组装”在一起。
- (3) 为中间构件和终端构件添加接口网标。

16.3.3 PCB设计基本原则

印刷电路板（Printed-Circuit Board, PCB）是由环氧树脂粘合而成的镀了铜的玻璃纤维板。蚀刻掉部分镀铜，只留下以构成电路互连通路的铜层走线。PCB 可以是单层、双层或者四层、六层、八层、十二层甚至更多。层数越多，连接的布线就越容易，但成本却越高，调试也越难。如果 PCB 有专用供电层和接地层，系统将会有更强的噪声抗扰度。

在使用电子设计自动化软件（如 Protel）设计 PCB 时需注意以下几点：

1. 布局

在正式走线之前要对 PCB 的大体格局进行规划，布局规划基本原则：

- (1) 在 PCB 布板之前首先要打印出相应的原理图，然后根据原理图确定整个 PCB 板的大体布局，即各个硬件构件的位置安排。
- (2) PCB 板的形状如无其他要求，一般为矩形，长宽比为 4:3 或 3:2。
- (3) 考虑面板上元件的放置要求。
- (4) 考虑边缘接口。

2. 元件放置

- (1) 芯片尽可能正放。
- (2) 元件放置要求整齐, 属于同一硬件构件内的元件尽可能排放在一起。
- (3) 电容的位置要特别注意, 其中电源模块的滤波电容要求靠近电源, 而 IC 的滤波电容要靠近 IC 的引脚。
- (4) 考虑元件间的距离, 防止元件之间出现重叠。
- (5) 晶体应平放, 而不要竖直放置。
- (6) PLL 滤波电路应尽量靠近 MCU。

3. 有关设定

在正式布线之前, 需要对软件环境的以下参数进行设置:

- (1) 线宽。导线尽可能宽, 这样既可以减小阻抗, 又可以防止由于制造工艺的原因导致导线断路。电源、地线的宽度要大于普通信号线。三者关系为: 地线宽度>电源线宽度>信号线宽度。
- (2) 间距。导线间距离以及导线与元件间距离要尽可能地大, 这样可以有效解决焊接时短路的问题。
- (3) 过孔大小。过孔大小设定要适中。

4. 布线

布线时, 应该首先对时钟和高速信号进行布线, 以确保它们的走线尽可能直接。石英晶振和对噪声特别敏感的器件下面不要走线。对总线进行布线时, 尽可能地保持信号线平行走线, 而时钟信号线则要避免平行。布板完成后一定要进行自动与人工检查。过孔数尽可能少。最小系统中未使用的 I/O 口, 可通过电阻接地。走线尽量少拐弯, 线宽不要突变, 导线拐角应 $\geq 90^\circ$, 力求线条简单明了。信号线的拐角应设计成钝角走向, 为圆形或圆弧形, 切忌画成 90° 或更小角度形状。

5. 测量点

考虑到硬件测试的方便, 在 PCB 布板时要留下一些测量点, 以便调试之用。测量点要根据原理图确定。以下几处需要留测量点:

- (1) 原理图中模块的输入输出引脚。
- (2) 最小系统模块中 MCU 的引脚。
- (3) 各硬件功能模块单元的输入、输出口;

画测量点的步骤是: 引出、打孔、标字。孔的大小以万用表头方便测量为原则, 不要在线上直接打孔。

6. 模块标示

由于在整体布局时, 已经将各个硬件构件的组成元件放在一起, 因而可在 PCB 板上用矩形框将各个硬件构件区分开, 并用汉字标出构件名(与原理图一致), 并注意字体字号。

7. 铺地

在布板的最后都要铺地, 目的是减小干扰, 提高 PCB 板的稳定性。铺地需注意:

- (1) 在铺地前, 要设定地与导线、地与引脚之间的距离, 并要求该距离尽可能大。
- (2) 铺地本应该双面铺, 作为实验用板, 为了方便检查, 可只铺反面地。
- (3) 如果电路板中有数字地和模拟地, 应将它们隔离开, 两者间使用磁珠相连。

8. 空余位置的利用

PCB 板的空余位置可用作：

- (1) 电源、地。空白处多留几排电源和地。
- (2) 双排孔。留出几排两孔相连的排孔，以用来扩展或试验时焊接其他元件。
- (3) 固定孔。在 PCB 上画固定板的固定孔，一般在板的四个角落。

9. 标注相关信息

在完成 PCB 板的铺地之后，要在板的正面适当位置标出以下信息：

- (1) 单位
- (2) 日期
- (3) 责任人
- (4) PCB 板的名称
- (5) 编号，如 2008-XX

16.4 基于硬件构件的嵌入式底层软件构件的编程方法

嵌入式系统是软件与硬件的综合体，硬件设计和软件设计相辅相成。嵌入式系统中的驱动程序是直接工作在各种硬件设备上的软件，是硬件和高层软件之间的桥梁。正是通过驱动程序，各种硬件设备才能正常运行，达到既定的工作效果。

16.4.1 嵌入式硬件构件和软件构件的层次模型

嵌入式软件构件 (Embedded Software Component, ESC) 是实现一定嵌入式系统功能的一组封装的、规范的、可重用的、具有嵌入特性的软件单元，是组织嵌入式系统的功能单位。

嵌入式软件构件分为高层软件构件和底层软件构件 (以下简称高层构件和底层构件)。高层构件与硬件无关。而底层构件与硬件密不可分，是硬件驱动程序的封装。前面提到，在硬件构件中，核心构件为 MCU 的最小系统。通常，MCU 内部包含有 GPIO (即通用 IO) 口和一些内置功能模块，我们将通用 I/O 口的驱动程序封装为 GPIO 构件，各内置功能模块的驱动程序封装为功能构件，如 MCF52233 的功能构件有 UART 构件、Flash 构件、USB 构件、IIC 构件等。

在硬件构件层中，相对于核心构件而言，中间构件和终端构件是核心构件的“外设”。由这些“外设”的驱动程序封装而成的软件构件称为底层外设构件。注意，并不是所有的中间构件和终端构件都可以作为编程对象。例如：键盘、LED、LCD 等硬件构件与编程有关，而电平转换硬件构件就与编程无关，因而不存在相应的底层驱动程序，当然也就没有相应的软件构件。嵌入式硬件构件与软件构件的层次模型如图 16-5 所示。

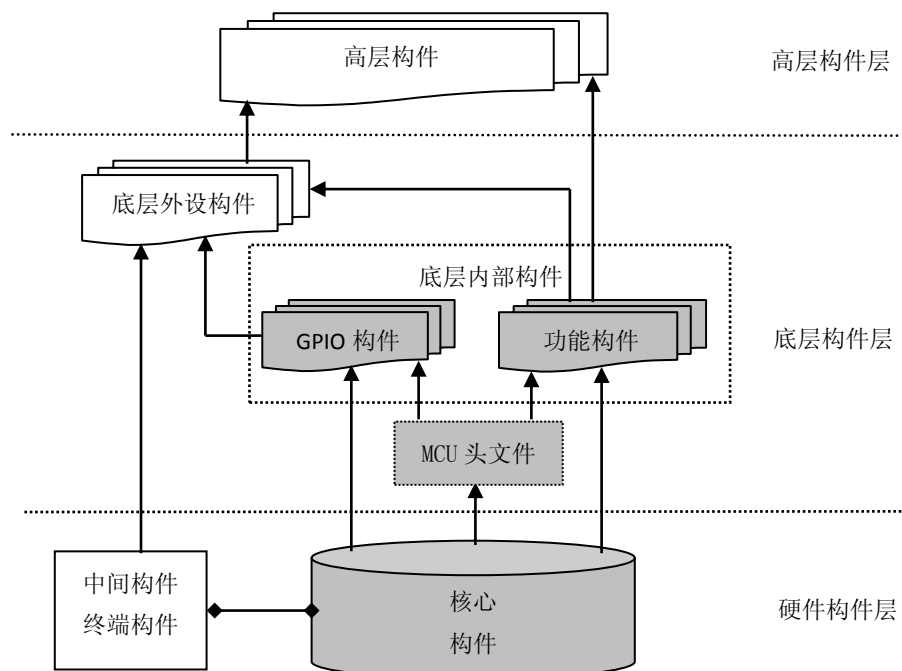


图16-5 嵌入式硬件构件与软件构件的层次模型

由图 16-5 可看出，底层外设构件可以调用底层内部构件，如 LCD 构件可以调用 GPIO 构件、PCF8563 构件（时钟构件）可以调用 IIC 构件等。而高层构件可以调用底层外设构件和底层内部构件中的功能构件，而不能直接调用 GPIO 构件。另外，考虑到几乎所有的底层内部构件都涉及到 MCU 各种寄存器的使用，因此将 MCU 的所有寄存器定义组织在一起，形成 MCU 头文件，以便其它构件头文件中包含该头文件。

16.4.2 底层构件的实现方法与编程思想

底层构件是与硬件直接打交道的软件，由头文件和源程序文件两部分组成。

头文件中的内容主要有：包含下层构件头文件的 `#include` 语句、用以描述构件属性的宏定义语句以及对外接口函数原型说明。在头文件中使用函数原型，对于建立代码模块和外部接口的规范，便于他人使用，都是很有帮助的。使用这些函数的用户，不需要查找源代码去了解参数的具体类型，直接查看函数原型即可。

源程序文件中存放构件的内部函数和外部函数的定义，即函数的实现代码，以完成函数所要实现的功能。

在对底层构件进行设计时，最关键的工作是要对构件的共性和个性进行分析，抽取出构件的属性和对外接口函数。尽可能做到：当一个底层构件应用到不同系统中时，仅需修改构件的头文件，对于构件的源程序文件则不必修改或改动很小。

例如，串行通信模块 SCI 是大多数 MCU 都具有的内部模块。仔细分析各种 MCU 串行通信程序发现：在查询方式下，各种 MCU 都是根据状态寄存器中的两个标志位来判断是否接收到数据和数据是否发送完毕，这就是 SCI 模块的共性。对于不同的 MCU，该状态寄存器的名称可能不同，这两个标志位的位号也有可能不同。此外，用以设置波特率、通信格式、是否校验、是否

允许中断等参数的寄存器也不同，这就是 SCI 模块的个性。分析出了共性和个性之后，就可以抽取出 SCI 构件的属性和操作，编制构件头文件和程序文件了。

以 MC68HC908GP32 的 SCI 模块为例，数据寄存器名称为 SCDR，空标志位（第 7 位）和满标志位（第 5 位）均位于状态寄存器 SCS1 中，SCI 构件头文件（SCI.h）设计如下：

```
#include "GP32.h"           //包含 MCU 头文件
#define SCI_ReSendDataR      SCDR    //数据寄存器
#define SCI_ReSendStatusR    SCS1    //空标志位和满标志位所在的状态寄存器
#define SCI_SendTestBit      7        //发送缓冲区空标志位
#define SCI_ReTestBit        5        //接收缓冲区满标志位
void SCI_Init(void);         //初始化 SCI
void SCI_Send1(unsigned char x);      //发送 1 个字节
void SCI_SendN(unsigned char n, unsigned char ch[]); //发送 n 个字节
unsigned char SCI_Receive1(unsigned char *p); //接收 1 个字节
unsigned char SCI_ReceiveN(unsigned char n, unsigned char ch[]); //接收 n 个字节
```

在编写构件时，主要注意以下几方面的内容：

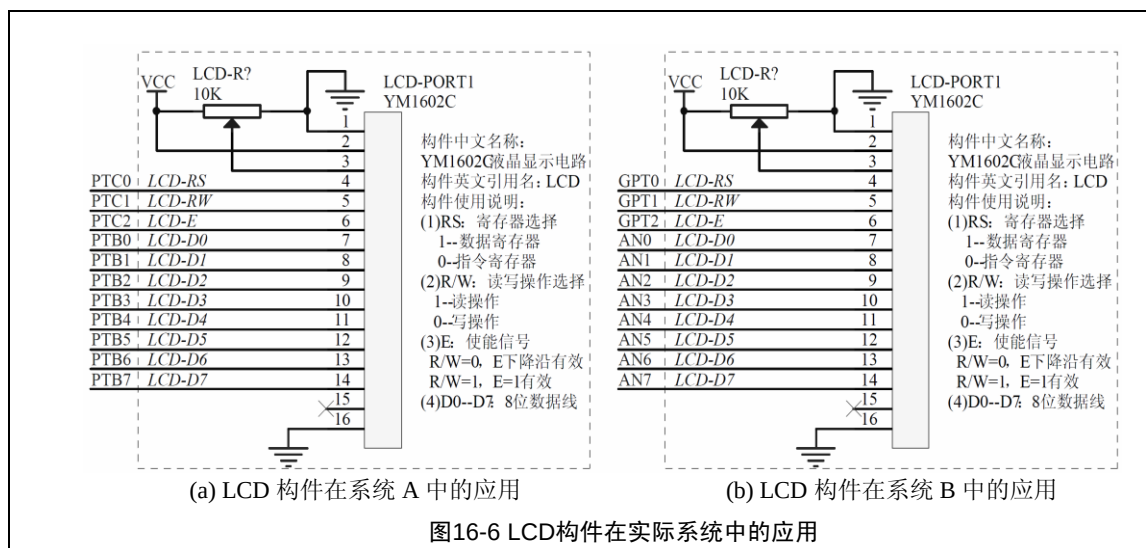
- （1）构件的头文件和源程序文件的主文件名一致，且为构件名。
- （2）属性和操作的命名统一以构件名开头。这样做的好处是：当使用底层构件组装软件系统时，避免构件之间出现同名现象。同时，名称要使人有“顾名思义”的效果。
- （3）对 MCU 内的模块寄存器名和端口名进行重定义，在其它的代码里面都将使用宏名对模块寄存器和端口进行操作。这样，当底层驱动程序移植到其它 MCU 时，只要修改重定义语句就可以了。
- （4）内部函数与外部函数要设计合理，函数参数个数及类型要考虑全面。内部函数仅提供给同一构件中的其它内部函数或外部函数调用，作用域仅限于定义该函数的文件。外部函数是对外接口函数，供上层应用程序调用。在定义外部函数时，应该对函数名、函数功能、入口参数、函数返回值、使用说明、函数适用范围等进行详细描述，以增强程序的可读性。上层应用程序不能直接对构件的属性进行读取或设置，必须借助于该构件提供的接口操作函数来实现。
- （5）应用程序在使用底层构件时，严格禁止通过全局变量来传递参数，所有的数据传递都要通过函数的形式参数来接收。这样做不但使得接口简洁，更加避免了全局变量可能引发的安全隐患。

16.4.3 硬件构件及底层软件构件的重用与移植方法

重用是指在一个系统中，同一构件可被重复使用多次。移植是指将一个系统中使用到的构件应用到另外一个系统中。

1. 硬件构件的重用与移植

对于以单 MCU 为核心的嵌入式应用系统而言，当用硬件构件“组装”硬件系统时，核心构件（即最小系统）有且只有一个，而中间构件和终端构件可有多个，并且相同类型的构件可出现多次。下面以终端构件 LCD 为例，介绍硬件构件的移植方法。



在应用系统 A 中，若 LCD 的数据线（LCD-D0~LCD-D7）与 M68HC908GP32（8 位 MCU）芯片的通用 IO 口的 B 口相连，C 口作为 LCD 的控制信号传送口，其中，LCD 寄存器选择信号 LCD-RS 与 C 口第 0 引脚连接，读写信号 LCD-RW 与 C 口第 1 引脚连接，使能信号 LCD_E 与 C 口第 2 引脚连接，则 LCD 硬件构件实例如图 16-6(a)所示。虚线框左边的文字（如 PTC0、PTC1 等）为接口网标，虚线框右边的文字（如 LCD-RS、LCD-RW 等）为接口注释。

在应用系统 B 中，若 LCD 的数据线（LCD-D0~LCD-D7）与 MCF52233（32 位 MCU）芯片的通用 IO 口的 AN 口相连，TA 口的第 0、1、2 引脚分别作为寄存器选择信号 LCD-RS、读写信号 LCD-RW、使能信号 LCD_E，则 LCD 硬件构件实例如图 16-6(b)所示。

2. 底层构件的移植

当一个已设计好的底层构件移植到另外一个嵌入式系统中时，其头文件和程序文件是否需要改动呢？这要视具体情况而定。例如：系统的核心构件发生改变（即 MCU 型号改变）时，底层内部构件头文件和某些对外接口函数也要随之改变，例如模块初始化函数。以 SCI 构件为例，由于不同 MCU 的 SCI 模块的数据寄存器名称不同、状态寄存器的名称及位定义不同，初始化工作所涉及的波特率、通信格式、是否校验等设置值也不同，因此当 MCU 改变时，构件头文件中相关宏定义值和 SCI_Init 的函数体实现代码也会有所不同。以 DG128 为例，其有两个串行口 SCI0 和 SCI1。若使用串行口 SCI0 进行收发数据，则数据寄存器名称为 SCI0DRL，空标志位（第 7 位）和满标志位（第 5 位）位于状态寄存器 SCI0SR1 中。因此需要将 GP32 的 SCI 构件头文件中前三行修改成如下内容：

```
#include "DG128.h"           //包含 MCU 头文件
#define SCI_ReceiveSendDataR  SCI0DRL //数据寄存器
#define SCI_ReceiveSendStatusR SCI0SR1 //空标志位和满标志位所在的状态寄存器
```

再以 LCD 构件为例，与图 16-6(a)相对应的底层构件头文件 LCD.h 可如下编写：

```
#include "GPIO.h"           //包含 GPIO 构件头文件
#define LCD_Data            GPIO_PORTB //显示数据传送口
#define LCD_Ctrl            GPIO_PORTC //控制信号传送口
#define LCD_RS              0          //寄存器选择信号
```



```
#define LCD_RW      1           //读写信号
#define LCD_E       2           //使能信号
void LCD_Init(void);           //液晶显示初始化
void LCD_Fill(INT8U a);        //填充 LCD，可实现清屏
void LCD_PutDot(INT8U x,INT8U y); //画点
void LCD_PutLine(INT8U x1,INT8U y1,INT8U x2,INT8U y2); //画线
void LCD_PutChar(INT8U a);     //显示一个字符
void LCD_PutString(INT8U str[]); //显示一串字符
```

当 LCD 硬件构件发生如图 16-6(b)所示的移植时，显示数据传送口和控制信号传送口发生了改变，此时，只要将上面的第 1 和第 2 条宏定义语句修改成：

```
#define LCD_Data     GPIO_PORTAN //显示数据传送口
#define LCD_Ctrl     GPIO_PORTTA //控制信号传送口
```

必须申明的是，本书提出构件化设计方法的目的是，在进行软硬件移植时，设计人员所做的改动要尽量小，而不是不作任何改动。事实上，不作任何改动是不现实的。

16.5 硬件构件化嵌入式开发技术的应用实例

为了让读者对硬件构件化嵌入式开发技术有更深入的理解，本节以“高温炉温度控制系统”为例，介绍该系统的硬件设计和软件设计方法。

16.5.1 系统功能

“高温炉温度控制系统”主要由 PC 机、集线器和若干个温度采集器组成。温度采集器通过传感器采集各点温度并在 LCD 显示屏上实时显示。按下温度采集器上的“开始”按钮，则检测在时间段(t_0)内，所测温度(T)与预先设定的基准值(T_0)的偏差是否超出了允许的波动值 ΔT ，若超出则输出报警信号。当接收到 PC 机的数据发送请求时，温度采集器通过网络将测得的温度值发送给 PC。通过 PC 端软件，可对温度值进行分析，并绘制处时间-温度变化曲线图。系统组成如图 16-7 所示。

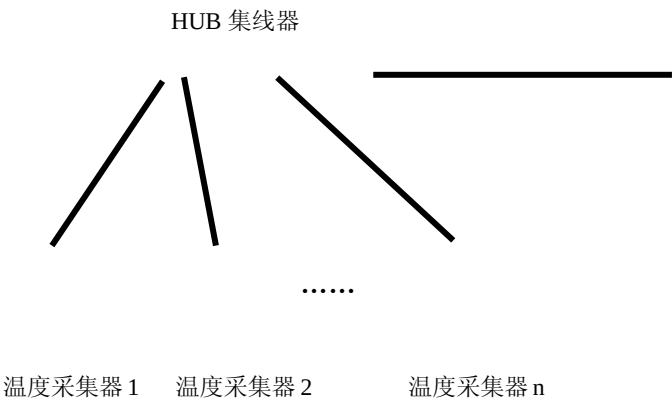


图16-7 高温炉温度控制系统的组成

16.5.2 硬件设计

在进行具体的硬件设计之前，必须根据系统所要实现的功能确定 MCU 的选型。考虑到，本系统需要借助温度传感器检测高温炉温度，并通过网络将相关数据传送给 PC 机，故选用了内含 A/D 转换模块和网络通信模块的 ColdFire 系列 MCF52233 作为系统的 MCU。

MCU 选型确定之后，接下来考虑电源，由于 MCF52233 MCU 需要 3.3V 供电，信号放大电路、继电器等需要 5V 供电，而外接电源为+24V，因此需要使用电源模块将+24V 转换为+5V 和 +3.3V。信号放大电路的作用是将温度传感器输入信号放大后，由 MCU 的 A/D 模块转换为数字信号，经过滤波、物理量回归转换成实际温度，并在 LCD 上显示。由于 MCF52233 MCU 的 GPIO 口资源有限，LCD 模块的数据线可采用串行输入模式。为方便用户设置一些工作参数，如温度基准值、允许最大偏差值等，可提供若干键盘按键。当温度波动超出一定范围时，通过继电器动作，输出报警信号。温度信息通过网络接口发送给 PC 机，本例中，将网络信号线与 24V 电源输入信号线合在一起，通过 9 芯串行口与外界连接。此外，为了便于测试系统当前运行状态，可设计冲突检测指示灯、网络连接指示灯、应答指示灯、运行指示灯以及出错指示灯等。

通过上面的分析，根据构件化思想，“高温炉温度控制系统”包含的硬件构件有：MCF52233 最小系统、电源构件、LCD 显示构件、指示灯（Light）构件、温度传感器信号放大构件、继电器构件、通信接口构件以及键盘构件。它们的功能和类型参见表 16-2。其中，MCF52233 最小系统构件原理图可参考本书第 2 章图 2-9。其它构件原理图如图 16-8 到如 16-14 所示。

表 16-2 高温炉温度控制系统的硬件构件划分

构件中文名称	构件英文名称	构件功能	类型
M52233最小系统	M52233-MinSys	MCF52233 MCU的最小系统，包含BDM电路。	核心构件
继电器	Relay	通过输入高低电平控制继电器的吸合。继电器闭合输出+24V。	中间构件
电源	Power	将+24V电压转换为+5V和+3.3V两种电压。	中间构件
温度传感器信号放大	TemEnLarge	将温度传感器两端电压差放大输出。	终端构件
液晶显示	LCD	采用串行输入模式显示数据。	终端构件
指示灯	Light	根据输入电平点亮或熄灭指示灯。	终端构件
键盘	KeyBoard	默认输出高点平，按下按钮输出低电平。	终端构件
网络接口	NETPORT	使用9芯的SCI接口作为网络通信接口。	终端构件

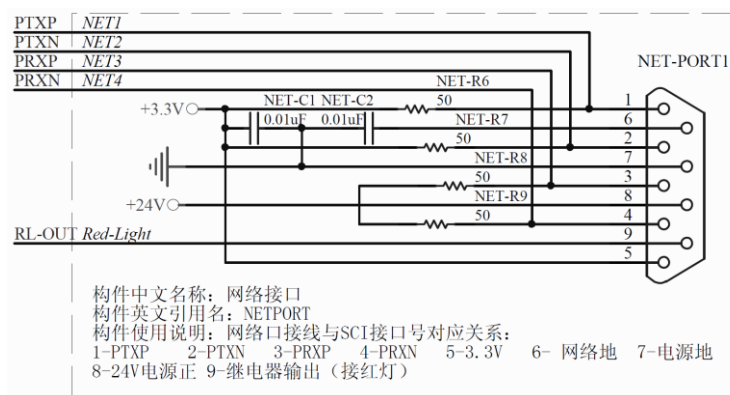


图16-8 网络接口构件

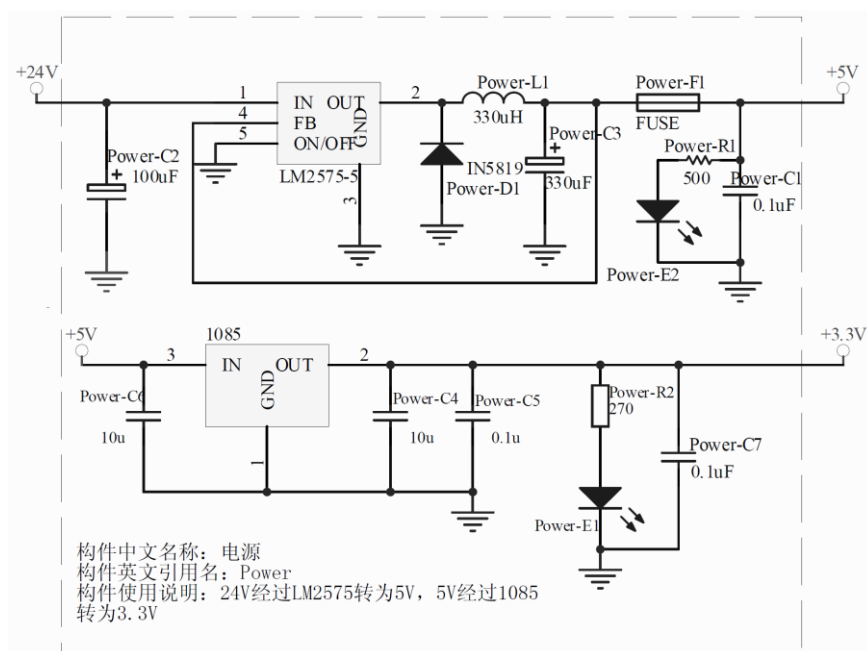


图16-9 电源构件原理图



图16-10 状态指示灯构件

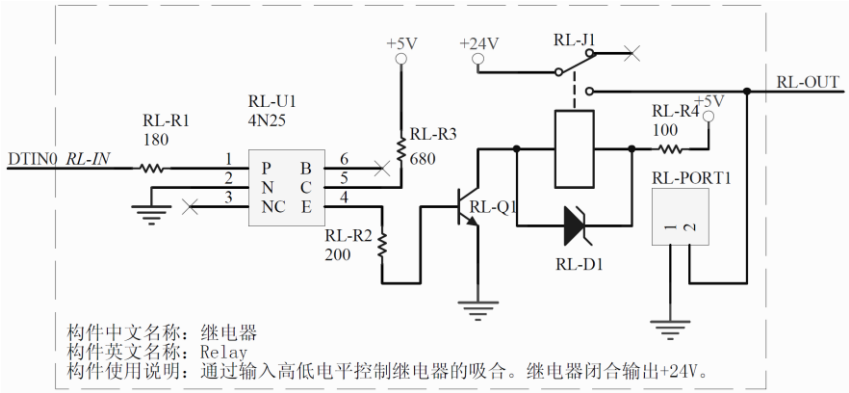


图16-11 继电器构件原理图

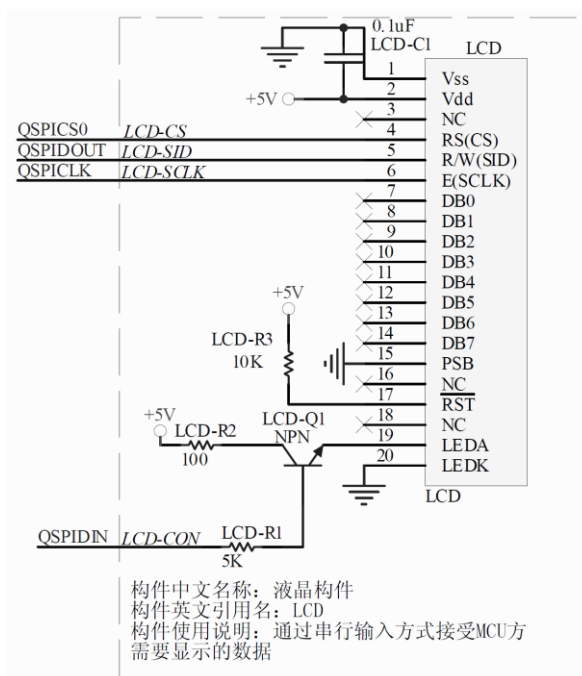


图16-12 液晶构件原理图

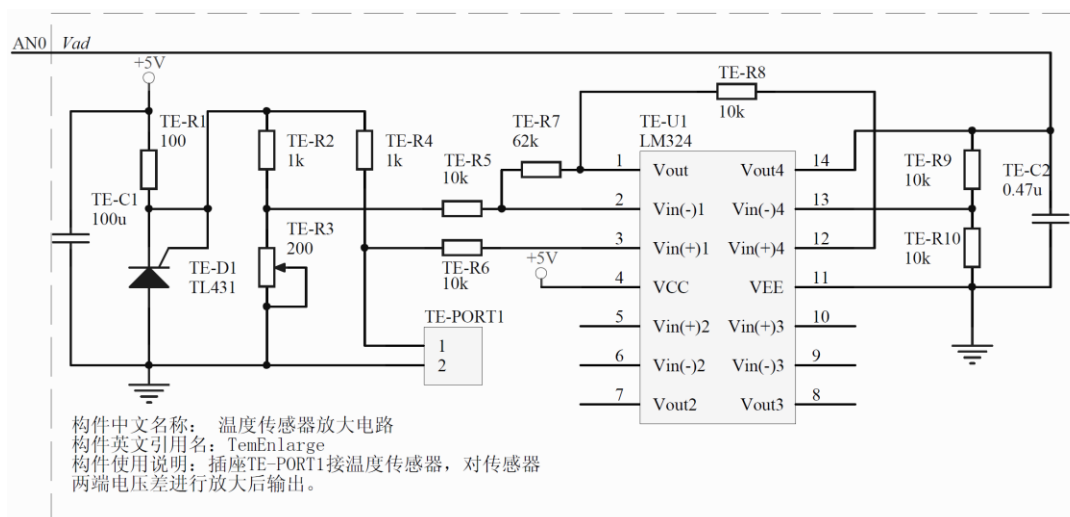


图16-13 温度传感器放大电路构件原理图

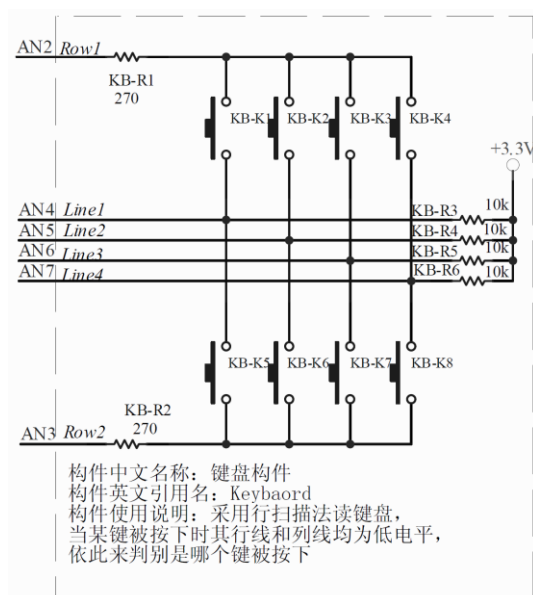


图16-14 键盘构件原理图

16.5.2 软件设计

硬件设计好之后，接下来的工作是进行软件设计。本案例的软件分为低端软件设计和高端软件设计两部分。

1. 底层构件设计

如前所述，底层构件是硬件的驱动程序，是构建低端软件的基石。在进行低端软件设计时，首先要结合软件所要实现的功能，按照构件化思想，设计并规划底层构件。本案例涉及到的底层构件有：

- (1) 通用 IO（GPIO）构件：初始化端口、设置端口状态以及获取端口状态等。
- (2) 指示灯（Light）构件：点亮或熄灭出错指示灯、运行指示灯等。
- (3) Flash 构件：将指定内容写入 Flash 的指定单元中，用于设置系统工作参数等。
- (4) 以太网物理层收发器（EPHY）构件：初始化 EPHY 模块、获取网络连接状态。
- (5) 快速以太网控制器（FEC）构件：初始化 FEC 模块、发送以太网帧、接收以太网帧。
- (6) 定时器（GPT）构件：初始化定时器。
- (7) 模数转换（AD）构件：初始化 A/D 转换模块、启动 A/D 转换、均值滤波等。
- (8) 键盘（KeyBoard）构件：扫描并获取键盘按键值。
- (9) 液晶显示（LCD）构件：初始化 LCD、显示字符数据等。

2. 主程序设计

低端软件的主程序负责获取 AD 采集数据，再将 AD 值转换为物理量，并在 LCD 显示屏上显示。当检测到用户按键时，根据不同的按键，完成不同的功能，流程如图 16-15。



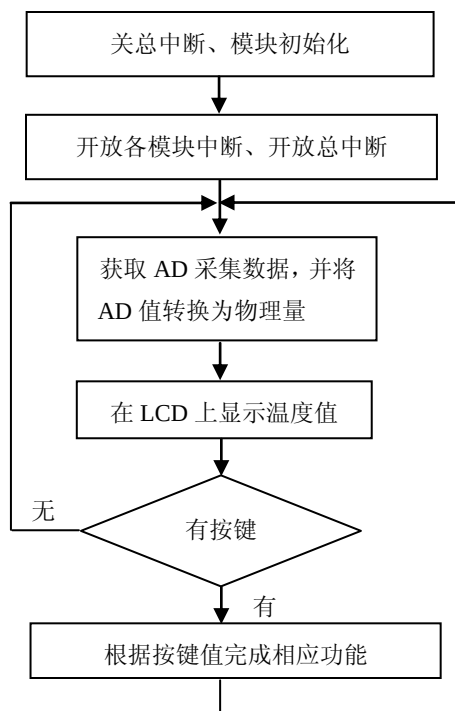


图16-15 低端软件主程序流程图

3. 中断程序设计

低端软件的中断程序主要响应 PC 机需要数据的请求。当接收到 PC 机发送的一条命令后，低端软件进入 FEC 接收中断。在中断服务子程序中，首先获取接收帧内容并分析，根据命令的不同，将相关数据组装成以太帧，并将帧内容写入到 FEC 模块的发送缓冲区中，然后再通过发送缓冲区描述符激活寄存器 TDAR 将其发送出去。



图16-16 高温炉温度控制系统工程相关源文件的树型结构

4. 文件组织结构

本案例中的低端软件开发环境采用 Freescale CodeWarrior, 其文件组织结构如图 16-16。main.c 为主程序文件, isr.h 和 isr.c 分别为中断处理函数的头文件和程序文件。与本书第三章“图 3-7 小灯闪烁工程相关源文件的树型结构”相比较, 不难看出, 两者之间唯一不同的是, 在 Component_H 和 Component_C 文件夹下, 增添了 Flash、EPHY、FEC、GPT、AD、KeyBoard 以及 LCD 构件的头文件和程序文件。

5. 高端软件设计

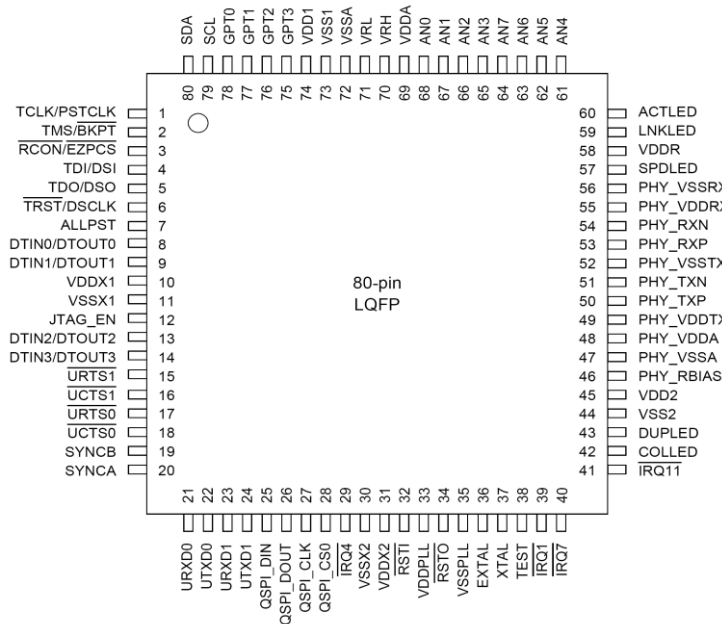
PC 机的高端软件需要实现的功能是：向低端程序提出温度数据请求并等待接收，对接收到的数据进行存储、显示和分析，并绘制出温度变化曲线等。考虑到，环境参数监测系统应用广泛，除了可应用于高温炉的温度监测，还可应用于畜禽养殖、生产厂区等场所的温度、照明度监测等，因此，本案例中的高端软件具有一定通用性，用户可根据实际情况设置软件运行参数，其开发环境使用 Microsoft Visual Studio 2005，编程语言使用 VB.NET，运行主界面如图 16-17 所示。



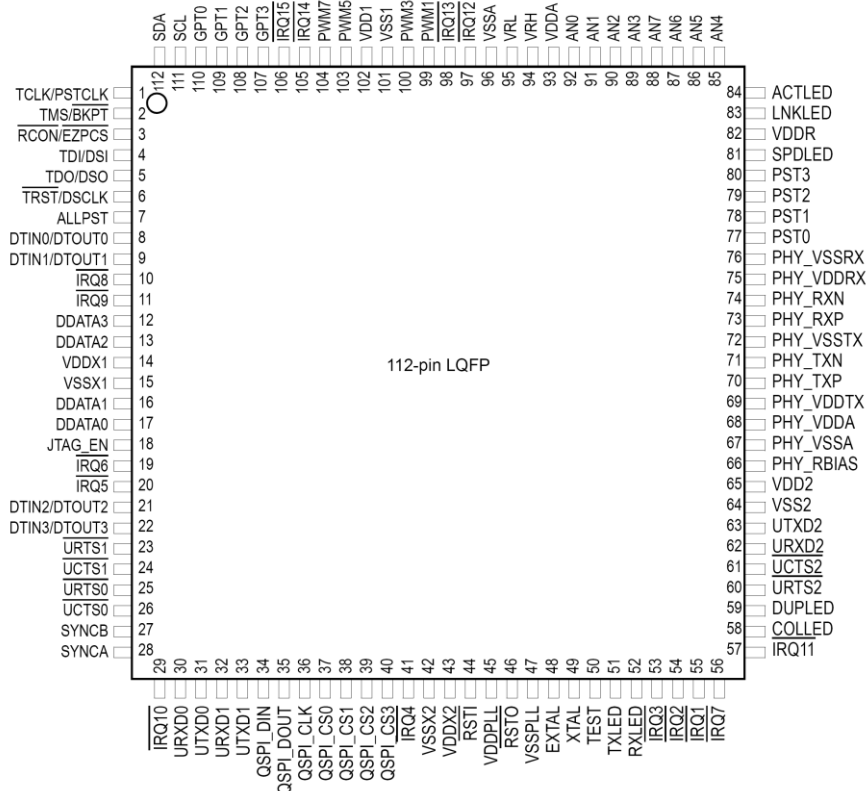
图16-17 通用型环境控制系统运行主界面

附录A 本书使用的ColdFire系列MCU芯片封装

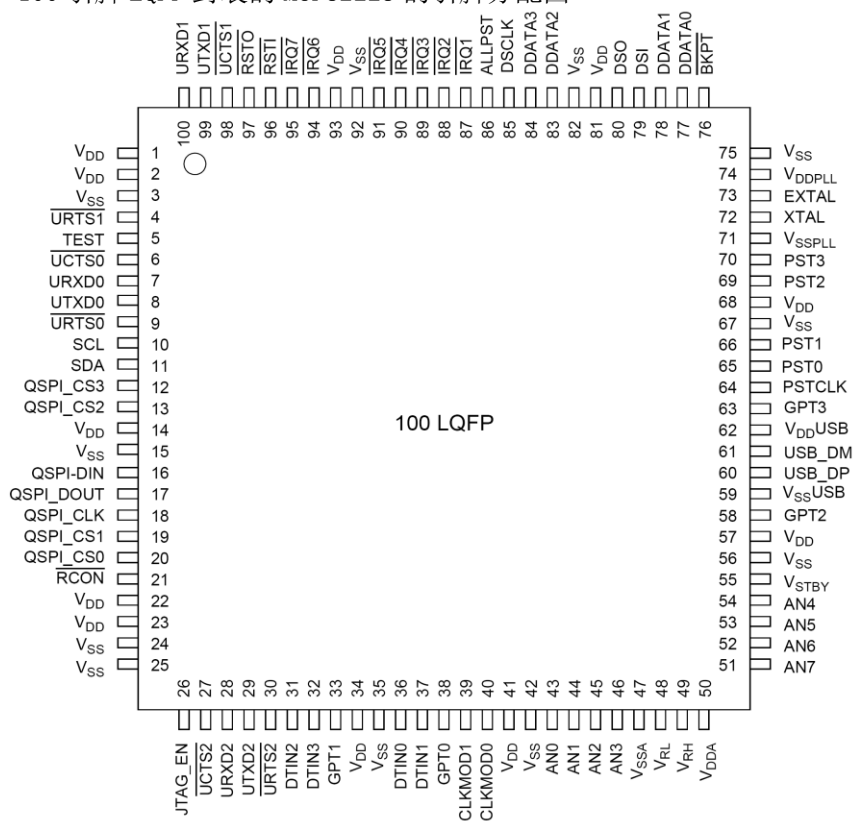
(1) 80 引脚 LQFP 封装的 MCF52233 的引脚分配图



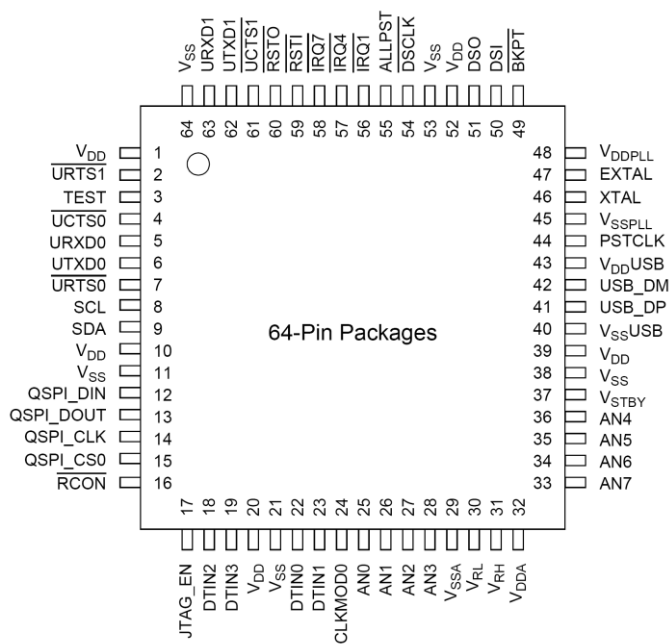
(2) 112 引脚 LQFP 封装的 MCF52233、MCF52235 的引脚分配图



(3) 100 引脚 LQFP 封装的 MCF52223 的引脚分配图



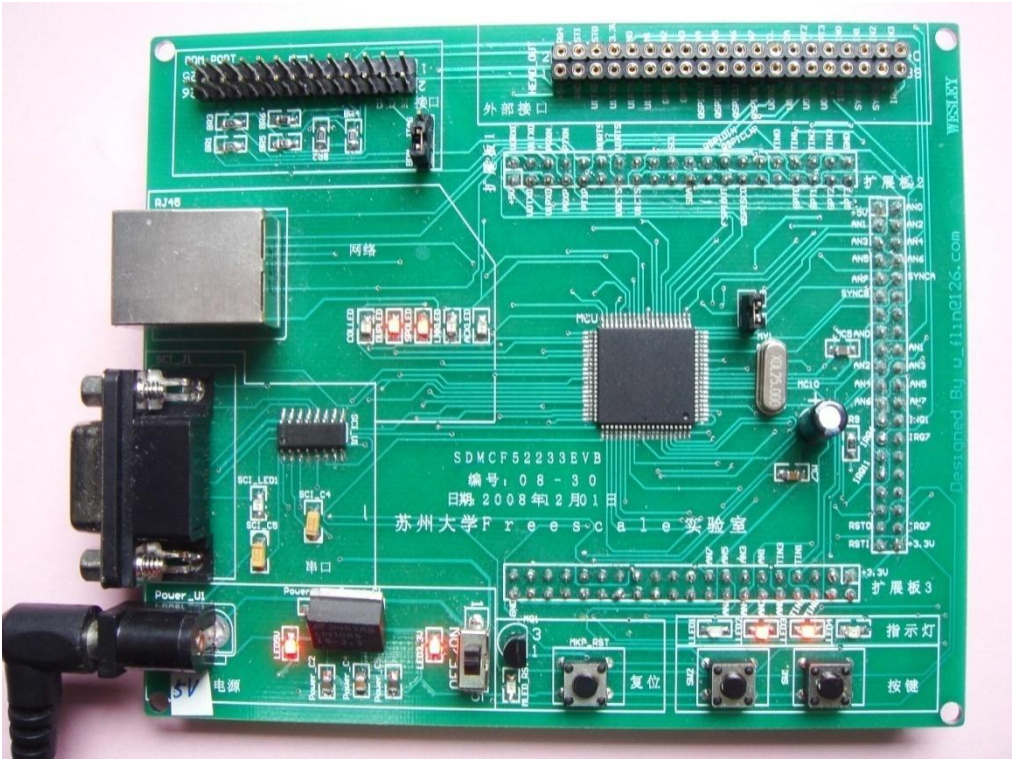
(4) 64 引脚 LQFP 和 QFN 封装的 MCF52223 的引脚分配图



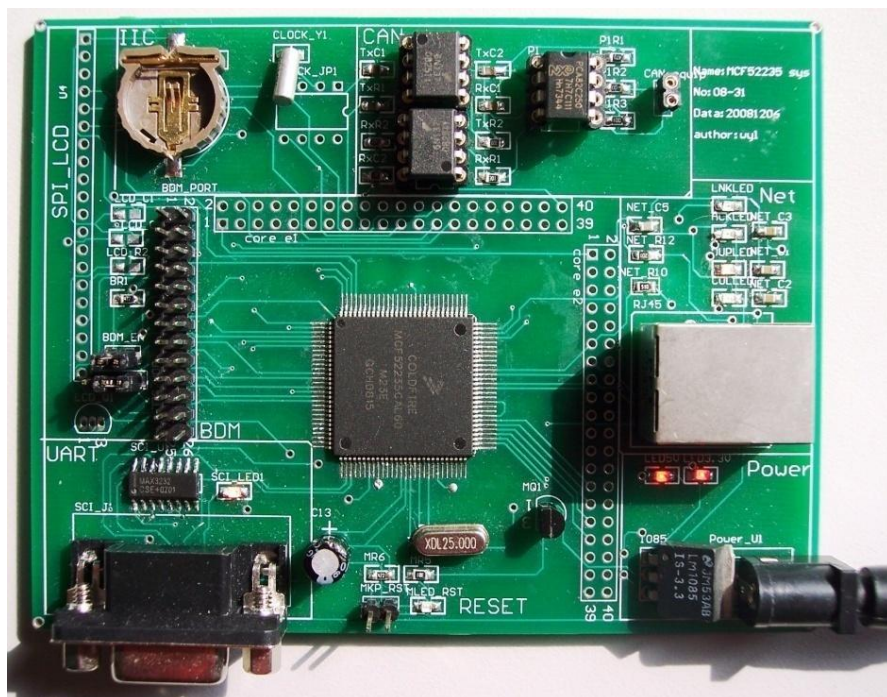
附录B 写入器、评估板及购买方法



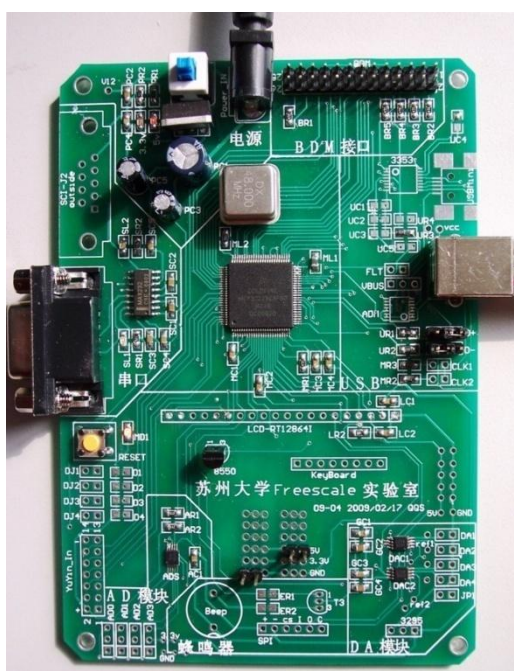
32 位 ColdFire 写入器



MCF52233 评估版（带写入器接口、串行口、以太网口）

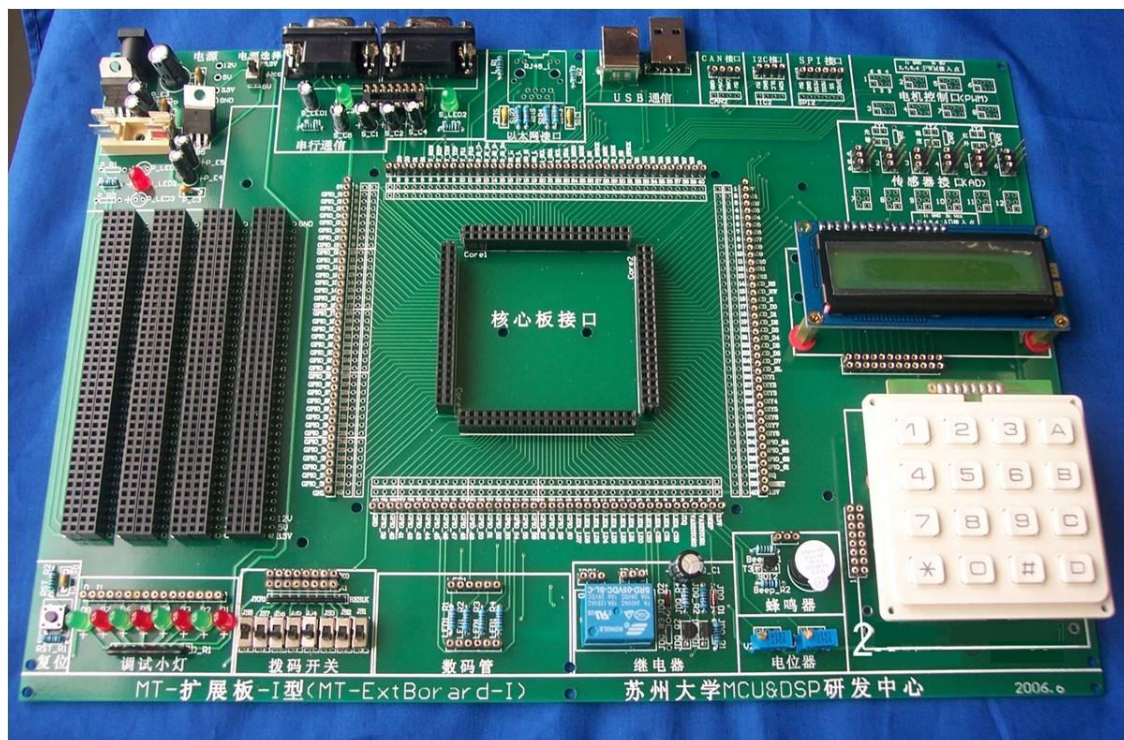


MCF52235 评估版（带写入器接口、串行口、以太网口）



MCF52223 评估版（带写入器接口、串行口、USB 口）

以上评估板可单独使用，也可插在如下图所示的扩展接口板上使用。



购买方法:

昆山鑫盛盟创科技有限公司:

网址: <http://www.xsmc.net>

地址: 昆山市经济开发区创业路1号(昆山留学生创业园205室)

邮编: 215300

电话: 0512-55213557 传真: 0512-55213556

EMAIL: xsmc@vip.sina.com, xsmc@xsmc.net

技术咨询:

苏州大学飞思卡尔嵌入式系统研发中心:

网址: <http://sumcu.suda.edu.cn>

电话: 0512-65214819 传真: 0512-65214819

附录C 常见实践问题解答

1 问：如果自己制作电路板，能否仅使用苏州大学飞思卡尔嵌入式系统研发中心提供有关评估板调试本书程序？

答：可以。对于 MCU 内部模块，如通用 I/O、定时器、I2C、串口通信、Flash、嵌入式以太网、USB 可以使用最小系统板；有些程序需要其它外接模块，如键盘、LED、LCD 等，此时可自己搭建电路，也可购买与扩展接口板结合进行硬件调试。

2 问：本书介绍的工程测试实例涉及哪些评估板？

答：MCF52233 评估板、MCF52235 评估板（CAN 模块实验用）和 MCF52223 评估板（USB 模块实验用）。

3 问：如何焊接和测试，以确保一个最小系统的硬件正常？

答：第 1 步：焊接电源、滤波、复位、晶振、PLL 以及写入器接口电路；第 2 步：在确保电源和地线未短路的情况下，接通电源。测量芯片电源电压是否正常，按下复位按钮是否能够使芯片复位，观察复位指示灯状态变化；第 3 步：连接写入器，对目标 MCU 进行擦除和写入。将具有简洁显示功能的程序（如小灯闪烁）写入芯片，看程序运行是否正常。如果成功，则表明最小系统工作正常。

4 问：将写入器与 PC 机的 USB 口连接后，为什么 PC 机的屏幕右下角显示“无法识别的 USB 设备”？

答：有两种可能。（1）没有正确安装写入器的 USB 驱动程序。（2）操作次序不合理。正确的操作方法是：先接通评估板电源插座，然后将写入器与 PC 机连接。

5 问：在 CodeWarrior 集成开发环境中使用 ColdFire 写入器对目标 MCU 进行擦除或写入时，出现错误提示：Error: Connect Failed. See Details for additional information. 为什么？

答：没有装入正确的 xml 文件或 Connection 项目选择错误。

（1）若是 MCF52233 和 MCF52235 评估板，单击“Load Settings...”按钮，选择“Flash Programmer\ColdFire”文件夹下的“CFM_MCF52233_25MHz.xml”文件，在“Target Configuration”栏目中，勾选“Use Custom Setting”，Connection 选中“PEMICRO_USB”。

（2）若是 MCF52223 评估板，单击“Load Settings...”按钮，选择“Flash Programmer\ColdFire”文件夹下的“CFM_MCF52223_48MHz.xml”文件，在“Target Configuration”栏目中，勾选“Use Custom Setting”，Connection 选中“TBLCF GDI”。

6 问：在 CodeWarrior 集成开发环境中使用 ColdFire 写入器对目标 MCU 进行擦除或

写入时，出现如下消息框，为什么？



答：“Target Configuration”栏目中，Connection 选项没有正确设置（参考上一个问答）。

7 问：在 CodeWarrior 集成开发环境中使用 ColdFire 写入器对目标 MCU 进行擦除或写入时，出现错误提示：Error: Erase failed. See Details for additional information. 为什么？

答：评估板要单独供电，不能仅靠 USB 供电。

8 问：MCF52233 评估板上的 LED1、LED2、LED3 以及 LED4 分别与哪个端口的引脚连接？

答：LED1、LED2、LED3 和 LED4 分别连接在 TC 端口的 0、1、2 和 3 引脚上。

9 问：在将 MCF52233 评估板插入扩展接口板上调试本书键盘按键程序时，需要人工连接哪些线？

答：若 4*4 键盘插在上排孔，则无需人工连线。若插在下排孔，则需将评估板上的 AN0~AN7 分别与扩展板上键盘区竖排插孔的 1~7 脚连接。

10 问：在将 MCF52233 评估板插入扩展接口板上调试本书四连排 LED 显示程序时，还需要人工连接哪些线？

答：无需连线。

11 问：在调试本书四连排 LED 显示程序时，LED 不显示或显示数据不符，为什么？

答：LED 数码管有共阴和共阳之分。扩展接口板适用于共阴类型的数码管。

12 问：如何通过万用表的测量判断一个四连排 LED 是共阴还是共阳？

答：使用万用表的电阻档（二极管），若正端接任一数据信号引脚，负端接任一片选信号引脚时为导通状态，反接为不导通状态，则为共阴数码管。若正端接任一片选信号引脚，负端接任一数据信号引脚时为导通状态，反接为不导通状态，则为共阳数码管。

13 问：在将 MCF52233 评估板插入扩展接口板上调试本书 LCD 显示程序时，还需要人工连接哪些线？

答：评估板上的 GPT1、GPT2、GPT3 分别与扩展板上的 LCD_E、LCD_RW、LCD_RS 连接。

14 问：扩展接口板上同时插上 MCF52233 评估板、键盘（上排）和 LCD 时，调试本书键盘按键程序时，发现按键不正常，为什么？

答：因为它们共用 MCF52233 的 AN 端口作为数据线了。

15 问：在调试本书串口通信程序时，PC 端的“串口调试器”窗口中出现乱码，为什么？

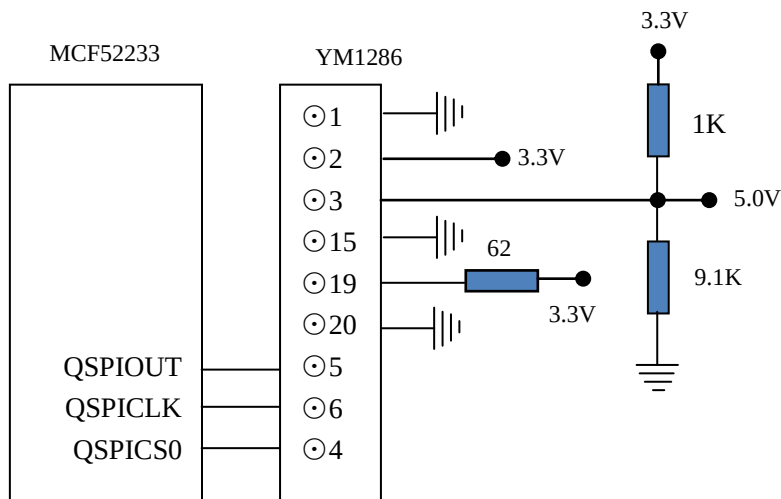
答：串口调试器中设置的波特率与 MCU 方设置的波特率不相同。

16 问：在调试本书 LCD 程序时，当串口发送 32 个字符给 LCD 显示后，再将这 32 个字符回发给 PC 机时，总是出现一段乱码，为什么？

答：这是由于写入器的干扰造成的。运行程序时，将写入器拔掉即可解决此问题。

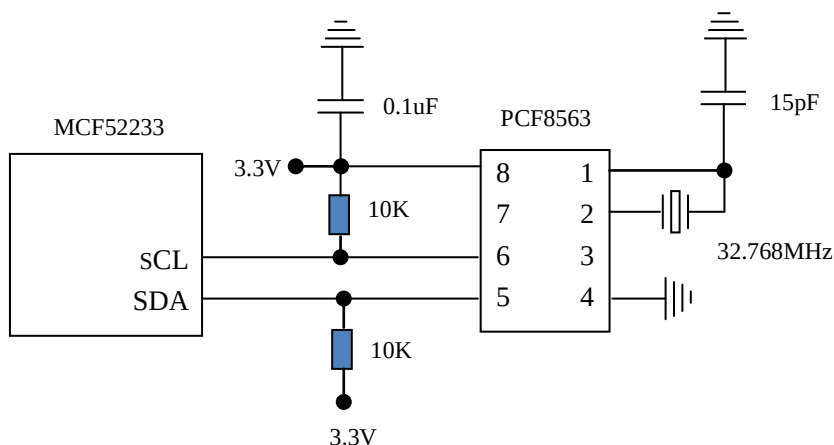
17 问：要进行本书介绍的 MCF52233 与 YM12864 液晶通过 QSPI 通信的实验，需要搭建怎样的电路？

答：电路如下图：



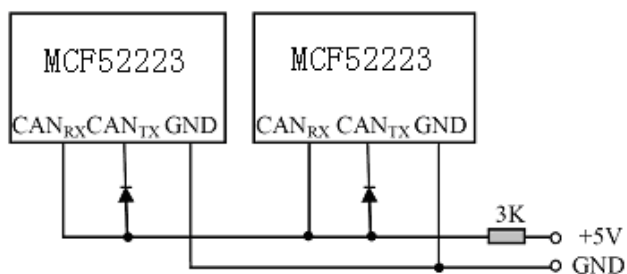
18 问：要进行本书介绍的 MCF52233 与 PCF8563 时钟芯片通过 IIC 通信的实验，需要搭建怎样的电路？

答：电路如下图：



19 问：若使用两块同样的 MCF52235 评估板进行 CAN 总线通信实验，最简单的方法需要搭建怎样的电路？

答：电路如下图，其中二极管可使用 1N4148。



20 问：在调试 Falsh 程序时，通过 PC 的串口向 MCU 发送擦除命令“E:页号”后，发现页号较小时，执行完擦除命令后，MCU 方程序就不能正常运行了，为什么？

答：E 命令中页号取值范围为 0~127。事实上，Flash 测试程序本身也要占用一定的 Flash 空间。当页号较小时，实际上将 Flash 测试程序擦除了，程序运行当然就不正常了。

21 问：在调试以太网通信程序时，当评估板与 PC 机通过网线连接后，PC 右下角仍显示未连接状态图标，为什么？

答：应该使用交叉网线。若手头上只有直通线，可将评估板通过以太网交换机与 PC 机通信。

主要参考书目

- [1] Freescale.MCF52235 ColdFire Integrated Microcontroller Reference Manual, Rev.5, 2007.
- [2] Freescale.MCF52235 ColdFire Microcontroller, Rev.3, 2007.
- [3] Freescale.MCF52223 ColdFire Integrated Microcontroller Reference Manual, Rev.3, 2007.
- [4] Freescale.MCF52223 ColdFire Microcontroller, Rev.2, 2007.
- [5] Freescale.ColdFire Family Programmer's Reference Manual, Rev.3, 2005
- [6] [美]Munir Bannoura, Rudan Bettelheim, Richard Soja 著, 李晶皎, 王爱侠, 高晓兴等译. ColdFire 微处理器与微控制器. 北京:电子工业出版社, 2008
- [7] [美]John Catsoulis 著, 徐君明, 许铁军, 黄年松等译. 嵌入式硬件设计. 北京:中国电力出版社, 2004
- [8] [美]Jack Ganssle, Michael Barr 著, 马广云等译. 英汉双解嵌入式系统词典. 北京:北京航空航天大学出版社, 2006
- [9] [美]Randall Hyde 著, 韩东海译. 编程卓越之道(第一卷 深入理解计算机). 北京:电子工业出版社, 2006
- [10] [美]Randall Hyde 著, 张菲译. 编程卓越之道(第二卷 运用底层语言思想编写高级语言代码). 北京:电子工业出版社, 2007
- [11] [印度]Raj Kamal 编著, 陈曙晖译. 嵌入式系统-体系结构、编程与设计. 清华大学出版社, 2005
- [12] [美]David E. Simon 著, 陈向群等译. 嵌入式系统软件教程. 北京:机械工业出版社, 2005
- [13] [加]Peter Spasov 著, 李小洪译. 微控制器原理与应用(第5版). 北京:清华大学出版社, 2006
- [14] [美]Jeremy Bentham. 嵌入式系统 Web 服务器. 北京:机械工业出版社, 2003
- [15] 谢希仁编著. 计算机网络(第5版). 北京:电子工业出版社, 2008
- [16] 边海龙、贾少华. USB 2.0 设备的设计与开发. 北京:人民邮电出版社, 2004
- [17] 任哲. 嵌入式实时操作系统 $\mu\text{C}/\text{OS-II}$ 原理及应用: 北京:北京航空航天大学出版社, 2005