

飞思卡尔大学计划用书

# 嵌入式系统设计实战

——基于飞思卡尔S12X微控制器

王宜怀 曹金华 编著



北京航空航天大学出版社  
BEIHANG UNIVERSITY PRESS

飞思卡尔大学计划用书

# 嵌入式系统设计实战 ——基于飞思卡尔 S12X 微控制器

王宜怀 曹金华 编著

北京航空航天大学出版社

## 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)16位S12X系列微控制器中MC9S12XS128为蓝本阐述嵌入式系统的软件与硬件设计。全书共11章,其中第1章阐述嵌入式系统的知识体系、学习误区与学习建议。第2章给出XS128硬件最小系统,并简要介绍S12XCPU(CPU12X)。第3章给出第一个样例程序及CodeWarrior工程组织,完成第一个S12X工程的入门。第4章给出基于硬件构件的嵌入式系统开发方法。第5章阐述串行通信接口SCI,并给出第一个带中断的实例。1~5章介绍了学习一个新MCU完整要素(知识点)的入门。6~12章分别介绍GPIO的应用(键盘、LED及LCD)、定时器(含PWM)、串行外设接口SPI、Flash存储器在线编程、CAN总线、A/D转换及S12XS128其他模块等。附录给出相关资料。

本书涉及的实例源程序、辅助资料、相关芯片资料及常用软件工具,可在北航出版社下载中心或苏州大学飞思卡尔嵌入式系统研发中心网站(sumcu.suda.edu.cn)下载。

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。

### 图书在版编目(CIP)数据

嵌入式系统设计实战:基于飞思卡尔S12X微控制器  
/王宜怀,曹金华编著.--北京:北京航空航天大学出版社,2011.5

ISBN 978-7-5124-0423-6

I. ①嵌… II. ①王… ②曹… III. ①微型计算机—系统设计 IV. ①TP360.21

中国版本图书馆CIP数据核字(2011)第074722号

版权所有,侵权必究。

### 嵌入式系统设计实战 ——基于飞思卡尔S12X微控制器

王宜怀 曹金华 编著

责任编辑 董立娟

\*

北京航空航天大学出版社出版发行

北京市海淀区学院路37号(邮编100191) <http://www.buaapress.com.cn>

发行部电话:(010)82317024 传真:(010)82328026

读者信箱:emsbook@gmail.com 邮购电话:(010)82316936

有限公司印装 各地书店经销

\*

开本:787×960 1/16 印张:27.5 字数:616千字

2011年5月第1版 2011年5月第1次印刷 印数:4 000册

ISBN 978-7-5124-0423-6 定价:49.00元

嵌入式计算机系统简称为嵌入式系统,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的影响。在后 PC 时代,嵌入式系统的广泛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(如工业控制、现代农业、家用电器、汽车电子、测控系统、数据采集、传感网应用等);另一类是计算机应用的延伸(如手机、电子图书、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的关注,嵌入式软硬件构件化开发方法逐步被业界所重视。

## 本书基本思想

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 16 位 S12X 系列微控制器中 MC9S12XS128 为蓝本阐述嵌入式系统的软件与硬件设计。教育部支持的高校科技竞赛之一“全国大学生飞思卡尔杯智能汽车竞赛”,从第四届(2009 年)开始,使用 S12XS128 芯片作为主控制器,一些学校也以此为蓝本进行嵌入式系统及应用教学,本书的主要目的是配合这两项工作。

我是从 1991 年开始从事单片机与嵌入式系统科研与教学工作的。1991—1999 年,使用 MCS-51 系列 MCU。2000 年至现在,一直使用飞思卡尔的 MCU。十多年来,陆续以飞思卡尔的 HC08/S08(8 位)、S12/S12X(16 位)、ColdFire(32 位)、M\*Core(32 位,该内核转给中国后称为 C\*Core)进行科研开发与教学工作,并以这些 MCU 为蓝本先后写了一些嵌入式应用技术入门方面的书,得到了大多数读者的肯定,深受感动。2010—2011 年,苏州大学嵌入式团队的工作重点是进行 ARM Cortex-M4 核 Kinetis 系列 MCU(K60)、新型 Zigbee 芯片 MC1323x、DSC 芯片 MC56F825x 等方面的工作,这些工作成果也将会逐步与读者分享。在写书方面,多年来一直在探索如何能够使读者不误入歧途,如何能够快速入门,如何能够规范编程,如何能够由浅入深、循序渐进,如何能够使读者打好嵌入式硬件与软件基础。为此从以下



几点把握写作：

① 把与芯片无关的通用知识分离出来,从涉及底层编程角度对基本原理进行简明扼要的阐述,分别放入相应章节的前面或网上光盘中。这些知识主要包括通用 I/O、串行通信、键盘编码原理、LED 扫描原理、SPI、PWM、USB、I<sup>2</sup>C、CAN、A/D、D/A、嵌入式以太网等。并在各书中基本保持不变。这一点是接受了飞思卡尔全球大学计划负责人 Andy Mastronardi 先生的建议,经过几年不断修改完善,可把通用部分斟酌得更好一些,也使得 8 位、16 位、32 位的书风格保持一致。新的芯片出来后,书的修改只要更新与芯片的相关部分。

② 硬件相关的部分,采用了硬件构件思想,制定了一些基本规范,对底层驱动进行构件化封装,提高了可复用性与可移植性。使程序结构更加清晰,初学者可以“先使用、后理解”。

③ 不论是 8 位、16 位、32 位,也不论是哪种芯片,从编程角度把与硬件相关的共性和与硬件无关的共性分别抽象出来,力求做到,硬件相关部分风格一致,硬件无关部分程序一致。这样便于融会贯通,不再纠结芯片位数、操作系统等问题。

## 关于飞思卡尔微控制器

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的一致赞誉。该公司的微控制器产品系列齐全,由不同位数(如 8 位、16 位、32 位等)、不同封装形式(如 DIP、SOIC、QFP、LQFP、BGA 等)、不同温度范围(0~70℃、-40~85℃、-40~105℃、-40~125℃等)、所含模块不同等构成了庞大的产品系列。飞思卡尔的 S08(8 位)、S12/S12X(16 位)、ColdFire(32 位)、ARM Cortex(32 位)等系列 MCU 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,为嵌入式系统各种应用提供了选择与解决方案,使得用户可以各取所需。

## 本书特点

2009 年,我编写了《基于 32 位 ColdFire 构建嵌入式系统》,2010 年编写了《嵌入式技术基础与实践(第 2 版)》。两书中系统阐述和应用了嵌入式构件开发思想,本书秉承这些工作,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。特点如下:

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,并保留在本书的网上光盘中,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,同时网上光盘还包含芯片参考手册、写入器安装与使用方法、工具软件(如开发环境、程序写入与读出软件、串口调试工具等)、有关硬件原理图及其他技术资料。

⑥ 提供硬件评估版、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。

## 本书主要内容

本书以飞思卡尔半导体 16 位 S12X 系列微控制器中 MC9S12XS128 为蓝本阐述嵌入式系统的软件与硬件设计。全书共 12 章,其中第 1 章阐述嵌入式系统的知识体系、学习误区与学习建议。第 2 章给出 XS128 硬件最小系统,并简要介绍 S12XCPU(CPU12X)。第 3 章给出第一个样例程序及 CodeWarrior 工程组织,完成第一个 S12X 工程的入门。第 4 章给出基于硬件构件的嵌入式系统开发方法。第 5 章阐述串行通信接口 SCI,并给出第一个带中断的实例。6~12 章分别介绍 GPIO 的应用(键盘、LED 及 LCD)、定时器(含 PWM)、A/D 转换及串行外设接口 SPI、Flash 存储器在线编程、CAN 总线、S12XS128 其他模块等。本书提供网上光盘,网上光盘中提供了本书所有实例源程序、辅助资料、相关芯片资料、常用软件工具及智能汽车竞赛参考资料,可在北航出版社下载中心或苏州大学飞思卡尔嵌入式系统研发中心网站(sumcu.suda.edu.cn)下载。

## 致 谢

本书除封面署名作者外,还得益于苏州大学计算机科学与技术学院嵌入式应用方向研究生姚丹丹、李翠霞、朱乐乐、冯上栋、石晶、苏勇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,为本书的编写提供了硬件及软件支持,并提出了许多宝贵建议。飞思卡尔半导体公司的许多技术人员提供了技术支持。北京航空航天大学出版社为本书的出版付出了大量细致的工作。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。有兴趣的读者可以发送邮件到:yihuaiw@suda.edu.cn,与作者进一步沟通,也可以发送邮件到:xdhydc5@sina.com,与本书策划编辑联系。

王宜怀

2011 年 3 月于苏州大学

# 网上光盘资料目录结构

---

- [-] 文件夹 SD-WYH-S12XS128BOOK-CD (V1.0-2011)
  - [-] 文件夹 01-整体资料
    - [+] 文件夹 0101-开发环境CodelWarrior for S12X (V5.0)
    - [+] 文件夹 0102- (S08-S12-ColdFire BDM) (写入器)安装与使用
    - 文件夹 0103-仪器用户手册
    - 文件夹 0104-CPU12X及XS128参考手册
    - 文件夹 0105-SD-WYH-S12XS128BOOK (课件V1.0)
    - 文件夹 0106-简介、前言及目录
  - [-] 文件夹 02-分章MCU源程序
    - [-] 文件夹 SD-WYH-S12XS128BOOK-Program (V1.0)-2011
      - [+] 文件夹 ASM
      - [+] 文件夹 Ch03-GPIO (Light)\_C
      - [+] 文件夹 Ch05-SCI\_C
      - [+] 文件夹 Ch06-KBI-LED-LCD\_C
      - [+] 文件夹 Ch07-Timer\_C
      - [+] 文件夹 Ch08-AD-SPI\_C
      - [+] 文件夹 Ch09-Flash
      - [+] 文件夹 Ch10-CAN\_C
      - [+] 文件夹 Ch11-Other\_C
  - [-] 文件夹 03-PC机源程序
    - [+] 文件夹 ADC-C#
    - [+] 文件夹 SCI-C#
    - [+] 文件夹 Timer-C#
  - [-] 文件夹 04-PC机免费工具
    - 文件夹 pcb\_hanzi
    - 文件夹 USB辅助工具
    - 文件夹 USB转串口驱动
    - 文件夹 USB-转串口驱动CH341SER
  - [-] 文件夹 05-分章阅读材料
    - 文件夹 第01章 (概述)阅读资料
    - [+] 文件夹 第02章 MCU阅读资料
    - 文件夹 第03章 (第一个程序)阅读资料
    - 文件夹 第04章 (构件开发方法)阅读资料
    - 文件夹 第05章 (SCI)阅读资料
    - 文件夹 第06章 (键盘、LED与LCD)阅读资料
    - 文件夹 第08章 AD转换模块阅读资料
    - 文件夹 第11章 其他阅读资料
  - [+] 文件夹 06-全国大学生飞思卡尔杯智能汽车竞赛-参考资料
  - [+] 文件夹 07-其他材料



# 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	2
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	10
1.3.1 与硬件相关的术语	10
1.3.2 与通信相关的术语	11
1.3.3 与功能模块及软件相关的术语	12
1.4 嵌入式系统常用的 C 语言基本语法	13
第 2 章 S12X 系列 MCU 硬件最小系统及 CPU12X	26
2.1 S12X 系列 MCU 概述及型号标识	26
2.1.1 S12X 系列 MCU 概述	26
2.1.2 S12X 系列 MCU 型号标识	28
2.2 S12X 系列 MCU 的功能及存储器映像	29
2.2.1 S12X 系列 MCU 的功能	30
2.2.2 S12X 系列 MCU 的存储器映像及特点	31
2.3 XS128 的引脚功能及硬件最小系统	36
2.3.1 XS128(80 引脚 QFP 封装)的引脚功能	37
2.3.2 XS128 的硬件最小系统	40





2.3.3 硬件最小系统的焊接与测试步骤	43
2.4 CPU12X 的内部寄存器	44
2.5 CPU12X 的寻址方式	47
2.6 CPU12X 指令系统概要	51
2.6.1 数据传送类指令	53
2.6.2 算术运算类指令	56
2.6.3 逻辑运算类与位操作类指令	60
2.6.4 程序控制类指令	63
2.6.5 其他类指令	71
2.7 CPU12X 汇编语言基础	72
2.7.1 S12X 汇编源程序格式	72
2.7.2 S12X 汇编语言伪指令	74
<b>第3章 第一个样例程序及 CodeWarrior 工程组织</b>	<b>77</b>
3.1 通用 I/O 接口基本概念及连接方法	77
3.2 XS128 的 GPIO 寄存器与 GPIO 构件封装	79
3.2.1 XS128 的 GPIO 寄存器	79
3.2.2 GPIO 的简单编程方法	83
3.3 CodeWarrior 开发环境与 S08/S12/ColdFire 三合一写入器	84
3.3.1 CodeWarrior 开发环境简介与基本使用方法	85
3.3.2 S08/ S12/ ColdFire 三合一写入器	86
3.3.3 MC9S12XS128 硬件评估板	87
3.4 CW 环境 C 语言工程文件的组织	87
3.4.1 工程文件的逻辑组织结构	88
3.4.2 工程文件的物理组织结构	90
3.4.3 系统启动及初始化相关文件	91
3.4.4 芯片初始化、主程序、中断程序及其他文件	98
3.4.5 机器码文件(.s19 文件)的简明解释	101
3.4.6 lst 文件与 map 文件	103
3.4.7 如何在 CW 环境下新建一个 S12 工程	105
3.5 第一个 C 语言工程:控制小灯闪烁	105
3.5.1 GPIO 构件设计	106
3.5.2 Light 构件设计	113
3.5.3 Light 测试工程主程序	115

3.5.4 理解第一个 C 工程的执行过程 .....	116
3.6 第一个汇编语言工程:控制小灯闪烁 .....	117
3.6.1 汇编工程文件的组织 .....	118
3.6.2 Light 构件汇编程序 .....	122
3.6.3 Light 测试工程主程序 .....	124
3.6.4 理解第一个汇编工程的执行过程 .....	126
<b>第 4 章 基于硬件构件的嵌入式系统开发方法</b> .....	<b>129</b>
4.1 嵌入式系统开发所遇到的若干问题 .....	129
4.2 嵌入式硬件构件的基本思想与应用方法 .....	130
4.3 基于硬件构件的嵌入式系统硬件电路设计 .....	131
4.3.1 设计时需要考虑的基本问题 .....	131
4.3.2 硬件构件化电路原理图绘制的简明规则 .....	133
4.3.3 实验 PCB 板设计的简明规则 .....	135
4.4 基于硬件构件的嵌入式底层软件构件的编程方法 .....	139
4.4.1 嵌入式硬件构件和软件构件的层次模型 .....	139
4.4.2 底层构件的实现方法与编程思想 .....	140
4.4.3 硬件构件及底层软件构件的重用与移植方法 .....	141
<b>第 5 章 串行通信接口 SCI</b> .....	<b>144</b>
5.1 异步串行通信的通用基础知识 .....	144
5.1.1 串行通信的基本概念 .....	145
5.1.2 RS-232 总线标准 .....	146
5.1.3 TTL 电平到 RS-232 电平转换电路 .....	148
5.1.4 串行通信编程模型 .....	149
5.2 SCI 模块的编程寄存器 .....	150
5.3 SCI 编程实例 .....	155
5.3.1 SCI 初始化与收发编程的基本方法 .....	156
5.3.2 SCI 构件设计与测试实例 .....	157
5.4 XS128 的中断源与第一个带有中断的编程实例 .....	166
5.4.1 中断与异常的通用知识 .....	166
5.4.2 XS128 的中断机制 .....	166
5.4.3 XS128 的中断编程方法 .....	171
5.4.4 XS128 的中断优先级编程实例 .....	173



第 6 章 GPIO 的应用实例:键盘、LED 与 LCD .....	175
6.1 键盘技术概述 .....	175
6.1.1 键盘模型及接口 .....	175
6.1.2 键盘编程的基本问题 .....	177
6.1.3 键盘构件设计与测试实例 .....	178
6.2 LED 技术概述 .....	184
6.2.1 扫描法 LED 显示编程原理 .....	184
6.2.2 LED 构件设计与测试实例 .....	186
6.3 LCD 技术概述 .....	191
6.3.1 LCD 的特点和分类 .....	191
6.3.2 点阵字符型液晶显示模块 .....	193
6.3.3 HD44780 .....	193
6.3.4 LCD 构件设计与测试实例 .....	199
第 7 章 定时器相关模块 .....	207
7.1 计数/定时器的基本工作原理 .....	207
7.2 定时器模块的基本编程方法与实例 .....	208
7.2.1 定时器模块计时功能的基本寄存器 .....	210
7.2.2 定时器构件设计与测试实例 .....	212
7.3 定时器模块输入捕捉功能的编程方法与实例 .....	216
7.3.1 输入捕捉的基本含义 .....	216
7.3.2 输入捕捉的寄存器 .....	217
7.3.3 输入捕捉构件设计与测试实例 .....	218
7.4 定时器模块输出比较功能的编程方法与实例 .....	221
7.4.1 输出比较的基本知识 .....	222
7.4.2 用于输出比较功能的相关寄存器 .....	222
7.4.3 输出比较构件设计与测试实例 .....	224
7.5 定时器模块脉冲累加功能的编程方法与实例 .....	226
7.5.1 脉冲累加的基本知识 .....	226
7.5.2 脉冲累加功能的相关寄存器 .....	227
7.5.3 脉冲累加器构件设计 .....	228
7.6 脉宽调制模块 .....	231
7.6.1 PWM 工作原理 .....	231

7.6.2 XS128 的 PWM 的特点及模块框图 .....	232
7.6.3 脉宽调制模块 PWM 相关寄存器 .....	233
7.6.4 PWM 构件设计及测试实例 .....	236
7.7 周期中断定时器模块 PIT .....	243
7.7.1 PIT 模块功能描述 .....	243
7.7.2 PIT 模块的编程寄存器 .....	245
7.7.3 PIT 构件设计与测试实例 .....	248
<b>第 8 章 A/D 与 SPI .....</b>	<b>252</b>
8.1 A/D 通用知识 .....	252
8.1.1 A/D 的基本问题 .....	252
8.1.2 A/D 转换器 .....	253
8.1.3 A/D 转换常用传感器简介 .....	254
8.1.4 电阻型传感器采样电路设计 .....	255
8.2 A/D 模块的编程寄存器 .....	257
8.3 A/D 模块编程方法与实例 .....	264
8.3.1 A/D 模块基本编程方法 .....	264
8.3.2 A/D 构件设计与测试实例 .....	265
8.4 SPI 的基本工作原理 .....	270
8.4.1 SPI 基本概念 .....	270
8.4.2 SPI 的数据传输 .....	272
8.4.3 SPI 模块的时序 .....	272
8.4.4 模拟 SPI .....	276
8.5 SPI 模块的编程寄存器 .....	276
8.6 SPI 构件设计与测试实例 .....	282
<b>第 9 章 Flash 存储器在线编程 .....</b>	<b>289</b>
9.1 S12X 系列 MCU 的 Flash 存储器的特点及分页机制 .....	289
9.1.1 S12X 系列 MCU 的 Flash 存储器的特点 .....	290
9.1.2 XS128 的 Flash 存储器分页机制 .....	290
9.2 Flash 存储器编程方法 .....	295
9.2.1 Flash 存储器编程的基本概念 .....	295
9.2.2 Flash 存储器的编程寄存器 .....	296
9.2.3 FCCOB-NVM 命令模式 .....	300



9.2.4	Flash 存储器的编程步骤 .....	301
9.3	D-Flash 在线编程 .....	303
9.4	P-Flash 在线编程 .....	308
9.5	Flash 存储器的保护特性和安全性 .....	313
9.5.1	Flash 存储器的配置区域 .....	313
9.5.2	Flash 存储器的保护特性 .....	314
9.5.3	Flash 存储器的安全性 .....	317
<b>第 10 章</b>	<b>CAN 总线 .....</b>	<b>321</b>
10.1	CAN 总线通用知识 .....	321
10.1.1	CAN 总线协议的历史概况 .....	321
10.1.2	CAN 硬件系统的典型电路 .....	321
10.1.3	CAN 总线的有关基本概念 .....	324
10.1.4	帧结构 .....	327
10.1.5	位时间 .....	331
10.2	MSCAN 模块简介 .....	332
10.2.1	MSCAN 特性 .....	333
10.2.2	报文存储结构、标识符验收过滤与时钟系统 .....	334
10.2.3	CAN 模块的主要运行模式、低功耗选项、中断与响应 .....	341
10.3	MSCAN 模块的内存映射及寄存器定义 .....	345
10.3.1	MSCAN 模块内存映射 .....	345
10.3.2	MSCAN 模块寄存器 .....	346
10.4	MSCAN 模块双机通信测试实例 .....	360
10.4.1	测试模型 .....	360
10.4.2	编程要点 .....	360
10.4.3	CAN 模块底层构件设计 .....	361
10.4.4	测试操作要点 .....	374
10.5	MSCAN 模块的自环通信实例 .....	374
10.5.1	测试模型 .....	374
10.5.2	编程要点及设计代码 .....	374
<b>第 11 章</b>	<b>系统时钟与其他功能模块 .....</b>	<b>378</b>
11.1	时钟与复位产生模块概述 .....	378
11.1.1	锁相环技术 .....	378

11.1.2	CRG 模块框图 .....	380
11.1.3	CRG 模块的工作模式 .....	381
11.1.4	XS128 内部锁相环结构 .....	383
11.2	XS128 的 CRG 模块的初始化 .....	384
11.2.1	XS128 的 CRG 模块寄存器 .....	384
11.2.2	初始化编程方法与实例 .....	389
11.3	CRG 模块的其他功能 .....	392
11.3.1	CRG 产生复位信号 .....	392
11.3.2	中 断 .....	397
11.4	XS128 的 $\overline{\text{IRQ}}$ 、 $\overline{\text{XIRQ}}$ 引脚、RTI、BRK 及 SWI 中断 .....	398
11.4.1	$\overline{\text{IRQ}}$ 与 $\overline{\text{XIRQ}}$ 引脚中断 .....	398
11.4.2	实时中断 .....	398
11.4.3	调试模块 DBG 与软件中断 SWI 指令 .....	399
附录 A	XS128 的映像寄存器 .....	400
附录 B	S08/S12/ColdFire BDM 简明使用方法 .....	410
附录 C	常见实践问题集锦 .....	414
附录 D	XS128 的 C 语言函数库 .....	417
附录 E	XS128 的中断源与中断向量表 .....	421
参考文献	.....	424

# 第 1 章

## 概 述

作为全书导引,本章主要知识点有:①简要给出嵌入式系统定义、由来及特点;②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;③归纳嵌入式系统的常用术语;④给出嵌入式系统常用的 C 的基本语法概要。网上光盘的【第 01 章(概述)阅读资料】中还补充给出了嵌入式 C 语言工程简明规范与嵌入式 C++ 语言的基本语法概要。

### 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》:

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,用作通用计算平台。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,嵌入式系统定义的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——MCU(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. MCU(微控制器)的基本含义

MCU 是单片微型计算机(单片机)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(Microcontroller)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”。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。

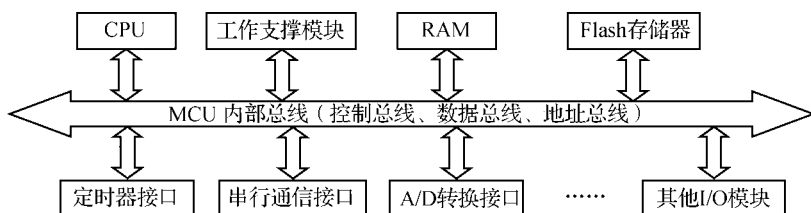


图 1-1 一个典型的 MCU 内部框图

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 5 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 20 世纪 70 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 MCU 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

何立民教授说:“有些人搞了十多年的 MCU 应用,不知道 MCU 就是一个最典型的嵌入式系统”。实际上,MCU 是在通用 CPU 基础上发展起来的,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,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,开发者可以根据具体要求与应用场合、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。其他电子线路成为 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,其他部分依此而展开。

### 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的角度谈嵌入式系统的特点。

① 嵌入式系统属于计算机系统,但不单独以通用计算机的面目出现。

嵌入式系统的本名叫嵌入式计算机系统(Embedded Computer System),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。

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的程序固化到非易失性存储器中。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如下:

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。它既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,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、写入调试器接口等);
- 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 I<sup>2</sup>C、CAN、USB、嵌入式以太网、无线传感器网络等);
- ⑥ 显示(LED、LCD、触摸屏等);
- 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嵌入式应用产品,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。有点像“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,再根据实际应用需要,选择一种实时操作系统(RTOS)进行实践。要记住:RTOS 是开发某些嵌入式产品的辅助工具,是手段,不是目的。况且许多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。

另外,即使学习 RTOS,也要认识到 RTOS 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%左右,80%左右是通用知识。但是这 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,嵌入式系统均含有硬件与软件两大部分,它们之间的关系如何呢?

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,没有像现在这样 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(MCU)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硬件设计方法发生了根本变化。特别是因软件危机而发展起来的软件工程学科对嵌入式系统软件的发展也产生重要影响,产生了嵌入式系统软件工程。

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者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者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,一些功能软件开发者名义上是在做嵌入式软件,但仅仅是使用嵌入式编辑、编译环境与下载工具而已,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。实际上,嵌入式系统设计是一个软件、硬件协同设计工程,不能像通用计算机那样,软件、硬件完全分开来看,要在一个大的框架内协调工作。在一些小型公司,需求分析、硬件设计、底层驱动、软件设计、产品测试等过程可能是由同一个团队完成,这就需要团队成员,对软件、硬件及产品需求有充分认识,才能协作开发好,甚至许多实际情况是在一些小公司这个“团队”可能就是一个人。

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。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,可能存在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,根据自身已有的知识结构,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,要注意知识学习的先后顺序与时间点的把握。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。如开发一个遥控器,一般不需要使用一个 32 位 MCU,原因是可能一个 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。由于这十多年来,我陆续以飞思卡尔 8 位(HC08、S08)、16 位(HCS12、S12X)、32 位(Coldfire)系列微控制器为蓝本写了嵌入式系统方面的书,其 C 语言编程实例按照构件封装原则把硬件相关的部分封装底层构件,统一接口,高层程序与芯片无关,可以在各种芯片应用系统移植与复用。所以,不论选择何种芯片学习,关键是要掌握嵌入式系统设计的基本要素与基本方法。

关于芯片选择的另一个误区是认为有“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。

关于学习芯片的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知识,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。

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,也会使你嵌入对嵌入式系统感到畏惧。等软件/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,一点一点地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,其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。

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的基础上理解硬件构件与软件构件等基本概念。

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。从学习规律角度讲,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

集成电路的封装(Package)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(Small Outline Package, SOP)、紧缩小外形封装(Shrink Small Outline Package, SSOP)、四方扁平封装(Quad-Flat Package, QFP)、塑料薄方封装(Plastic-Low-profile Quad-Flat Package, LQFP)、塑料扁平组件式封装(Plastic Flat Package, PFP)、带载封装(Tape Carrier Package, TCP)、插针网格阵列封装(Ceramic Pin Grid Array Package, PGA)、球栅阵列封装(Ball Grid Array Package, BGA)等。

#### (2) 印刷电路板

印刷电路板 PCB(Printed Circuit Board)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)与静态可读写随机存储器(SRAM)

动态可读写随机存储器 DRAM(Dynamic Random Access Memory)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,但控制繁琐,接口复杂。

静态可读/写随机存储器 SRAM(Static Random Access Memory),一般由 4 个或者 6 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。

#### (4) 只读存储器

只读存储器 ROM(Read Only Memory),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) 3 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。

### (5) 闪速存储器

闪速存储器(Flash Memory)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。闪存相对于传统的 EPROM 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。闪存的结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。

### (6) 模拟量与开关量

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。

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。下面介绍的 SPI、I<sup>2</sup>C、USB 等通信方式也属于串行通信,但由于历史发展和应用领域的不同,它们分别使用不同的专用名词来命名。

### (3) 串行外设接口

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

I<sup>2</sup>C(另一种简写为 I2C, Inter-Integrated Circuit),是一种由 NXP 公司开发的两线式串行



总线,有的书籍也记为 IIC,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

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。USB 不仅成为通用计算机上最重要通信接口,也是手机、家电等嵌入式产品的重要通信接口。

### (6) 控制器局域网

控制器局域网(CAN,Controller 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

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

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,这里不再进一步介绍。

## 1.3.3 与功能模块及软件相关的术语

### (1) 通用输入/输出

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

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除看门狗定时器,看门狗定时器会自动溢出,使系统程序复位。

### (5) 液晶显示

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3类。

### (6) 发光二极管

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

实时操作系统(RTOS,Real Time Operating System)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。

## 1.4 嵌入式系统常用的C语言基本语法

C语言是在20世纪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。

网上光盘的【第01章(概述)阅读资料】中还补充给出了嵌入式C语言工程简明规范与嵌入式C++语言的基本语法概要。

### (1) 基本数据类型

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所列。



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	-32 768~+32 767
	unsigned short	无符号短整型	16	2	0~65 535
	signed int	有符号短整型	16	2	-32 768~+32 767
	unsigned int	无符号短整型	16	2	0~65 535
	signed long	有符号长整型	32	4	-2 147 483 648~+2 147 483 647
	unsigned long	无符号长整型	32	4	0~4 294 967 295
实型	float	浮点型	32	4	约 $\pm 3.4 \times (10^{-38} \sim 10^{+38})$
	double	双精度型	64	8	约 $\pm 1.7 \times (10^{-308} \sim 10^{+308})$

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
	%	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

续表 1-2

运算类型	运算符	简明含义	举 例
位运算	~	按位取反	$A=0b00001111, B=\sim A, B=0b11110000$
	<<	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	$A=*P$
	&	取地址	$A=\&P$
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 3 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。

### (a) 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,则执行 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。

### (b) 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。

(a) 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: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”。

(b) 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。

(c) 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 语句在循环中的应用

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执行,进行一次新的循环(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,并返回一个确定值给调用程序。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) 数 组

在 C 语言中,数组是一个构造类型的数据,是由基本类型数据按照一定的规则组成的。构造类型还包括结构体类型及共用体类型。数组是有序数据的集合,数组中的每一个元素都属于同一个数据类型。用一个统一的数组名和下标唯一地确定数组中的元素。

##### 1) 一维数组的定义和引用

定义方式为:类型说明符 数组名[常量表达式];

其中,数组名的命名规则和变量相同。定义数组的时候,需要指定数组中元素的个数,即常量表达式需要明确设定,不可以包含变量。

**【例如】:**

```
int a[10];    //定义了一个整型数组,数组名为 a,有 10 个元素,下标 0-9
```

数组必须先定义,然后才能使用。而且只能通过下标一个一个的访问。形如:数组名[下标]。

##### 2) 二维数组的定义和引用

定义方式为:类型说明符 数组名[常量表达式][常量表达式]

**【例如】:**

```
float a[3][4];    //定义 3 行 4 列的数组 a,下标 0-2,0-3
```

其实,二维数组可以看成是两个一维数组。可以把 `a` 看作是一个一维数组,它有 3 个元素:`a[0]`、`a[1]`、`a[2]`,而每个元素又是一个包含 4 个元素的一维数组。

二维数组的表示形式为:数组名[下标][下标]。

##### 3) 字符数组

用于存放字符数据(char 类型)的数组是字符数组。字符数组中的一个元素存放一个字符。

**【例如】:**

```
char c[5];  
c[0] = 't'; c[1] = 'a'; c[2] = 'b'; c[3] = 'l'; c[4] = 'e';  
//字符数组 c[5]中存放的就是字符串“table”。
```

C语言中是将字符串作为字符数组来处理的。但是,在实际应用中,关于字符串的实际长度,C语言规定了一个“字符串结束标志”,以字符'\0'作为标志(实际值 0x00)。即如果有一个字符串,前面  $n-1$  个字符都不是空字符(即'\0'),而第  $n$  个字符是'\0',则此字符的有效字符为  $n-1$  个。

#### 4) 动态数组

动态数组是相对于静态数组而言。静态数组的长度是预先定义好的,在整个程序中,一旦给定大小后就无法改变。而动态数组则不然,它可以随程序需要而重新指定大小。动态数组的内存空间是从堆(heap)上分配(即动态分配)的,是通过执行代码而为其分配存储空间。当程序执行到这些语句时,才为其分配。程序员自己负责释放内存。

在C语言中,可以通过 malloc、calloc 函数进行内存空间的动态分配,从而实现数组的动态化,以满足实际需求。

#### 5) 数组如何模拟指针的效果

其实,数组名就是一个地址,一个指向这个数组元素集合的首地址。可以通过数组加位置的方式进行数组元素的引用。

**【例如】:**

a[5];            方式一: a[2]                      方式二: \*(a+2)

这两种方式都可以访问到数组 a 的第 3 个元素。关键是数组的名称本身就可以当作地址看待。

#### (6) 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。但是使用指针要特别细心,计算得当,避免指向不适当区域。

指针是一种特殊的数据类型,在其他语言中一般没有。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

(a) 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。

(b) 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。

**【例如】:**

```
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。

(c) 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 是指向数组 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 0~4),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,对指向其他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,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 void \* 则不同,任何类型的指针都可以直接赋值给它,无须进行强制类型转换:

```
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,“void \*”也可以无须强制类型转换地赋给其他类型的指针,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“int \*”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”。

### (7) 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

**【例如】**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
```

**【例如】**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像其他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,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,好像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:

```
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。



## (8) 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

**【例如】**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 位二进位。

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位分配的。

## 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,这样产生的代码紧凑、高效。

## (9) 编译预处理

C 语言提供编译预处理的功能,这是 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
#endif 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”
```

### (10) 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,然后在需要用到它们时,用 #include 命令把该文件包含进来。

⑤ 使用 typedef 有利于程序的通用与移植。

# 第 2 章

## S12X 系列 MCU 硬件最小系统及 CPU12X

飞思卡尔的 S12X 系列 MCU 是 S12 系列 MCU 的升级版本。本书以 MC9S12XS128 为蓝本阐述嵌入式应用。本章主要知识点有:①简要概述 Freescale 的 16 位 S12X 系列 MCU 及型号标识;②给出 MC9S12XS128 微控制器功能概述、存储器映像、引脚功能与硬件最小系统;③概要介绍 S12XCPU(CPU12X)的特点及内部寄存器、寻址方式、指令系统、CPU12X 汇编语言基础。本章中存储器映像、引脚功能与硬件最小系统是重点掌握的内容。硬件最小系统是芯片运行的基本条件,应该对此有清晰的理解。

### 2.1 S12X 系列 MCU 概述及型号标识

#### 2.1.1 S12X 系列 MCU 概述

在 2005 年左右,飞思卡尔公司在 16 位 S12 系列 MCU 基础上,开始推出 S12X 系列 MCU。它是新一代的双核微控制器,拥有卓越的性能,堪比 32 位微控制器。S12X 除了拥有主 CPU 外,还拥有—个平行处理器 XGATE 模块;该模块是一个智能的、可编程的直接存储器存取模块,可以进行中断处理及通信和数据预处理,并为其他任务释放一部分 CPU 空间,从而提高了芯片的整体运行性能。随着每年—亿多件的销量,S12/S12X 系列 MCU 已成为汽车行业领先的 16 位解决方案。在实现这一销量里程碑的同时,飞思卡尔维持了不到百万分之一的低缺陷率。

S12X 系列 MCU 的处理器使用 CPU12X(有时也称 S12XCPU)。CPU12X 从 CPU12(用于 S12 系列 MCU)发展而来。CPU12 的版本有 CPU12V0、CPU12V1 等,CPU12X 的版本有 CPU12XV0、CPU12XV1、CPU12XV2。CPU12X 核(CPU12XV0、CPU12XV1、CPU12XV2)都支持单周期的乘法操作和中断优先级,CPU12XV1 和 CPU12XV2 支持 SYS 指令,而 CPU12XV2 还支持用户状态。

从制造工艺来看,S12X 系列微控制器已经从 S12 系列微控制器的  $0.25\mu\text{m}$  制造工艺发展到  $0.18\mu\text{m}$  制造工艺,总线频率也从 25 MHz 发展到了 40 MHz。目前,S12X 的一个子系列 S12XG 已经发展到了 90 nm 的制作工艺,70 MHz 的总线频率。

S12X 已经出品了多个子系列,图 2-1 给出了 S12X 产品发展路线图,从中可以了解

## 16-bit Body Electronics MCU Roadmap

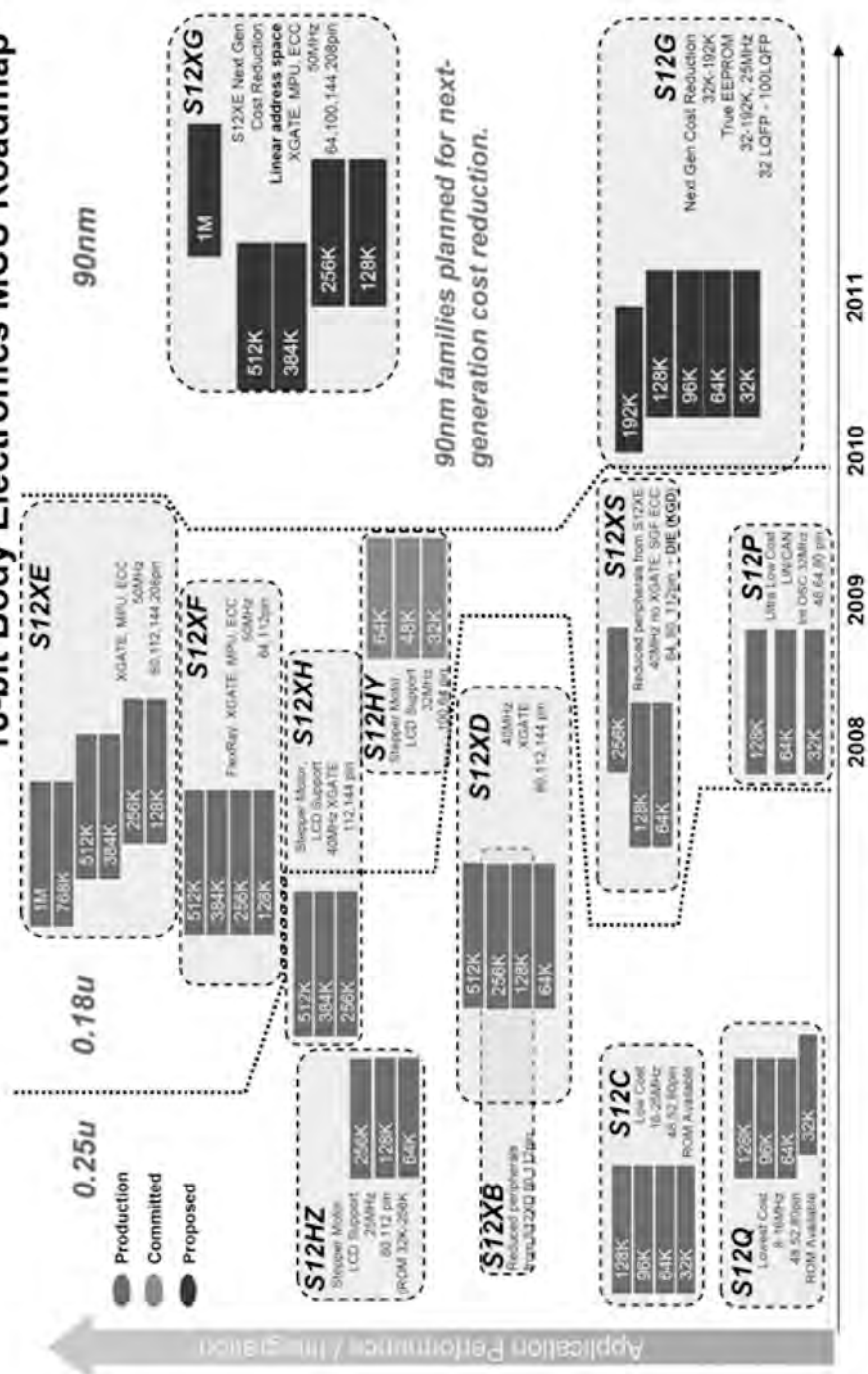


图 2-1 S12X 系列 MCU 发展路线图

S12X 的产品系列概况。本书以 S12X 的一个子系列 S12XS 系列 MCU 为蓝本阐述嵌入式应用。S12X 系列 MCU 使用 CPU12XV2 版本,未使用协处理器 XGATE,从而为 S12X 系列增加了一个可供选择的、经济的、高性能及高可靠的解决方案;内部总线频率可达 40 MHz,含有错误代码纠正 ECC(Error Code Correction)模块,提高了系统稳定性。

S12XS 系列微控制器主要应用领域是汽车电子,包括座位控制器、暖通空调控制、方向盘控制及电动汽车的电池管理与均衡系统等。在工业控制、通信等领域,S12XS 系列微控制器也大有用武之地。

Freescale 的 S12X 系列 MCU 有许多型号,不同型号的 MCU 中资源各不相同,即使是同一种型号的 MCU,也有多种封装形式(如目前的 XS 系列有 64、80、112 引脚 3 种封装形式),其 I/O 引脚数目也不相同。这里只简要介绍 S12X 子系列 S12XS 的资源配置情况,其他型号的资源情况可以到飞思卡尔官方网站(<http://www.freescale.com>)上查找。

表 2-1 给出了 S12XS 子系列 MCU 的引脚数量、封装形式及主要资源。

表 2-1 S12XS 系列 MCU 的引脚数量、封装形式及主要资源

分 类	引脚/封装	P-Flash	RAM	D-Flash	CAN	SPI	SCI	TIM	PIT	A/D	PWM
9S12XS256	112/LQFP	256 KB	12 KB	8 KB	1	1	2	8 通道	4 通道	16 通道	8 通道
	80/QFP				1	1	2	8 通道	4 通道	8 通道	8 通道
	64/LQFP				1	1	2	8 通道	4 通道	8 通道	8 通道
9S12XS128	112/LQFP	128 KB	8 KB	8 KB	1	1	2	8 通道	4 通道	16 通道	8 通道
	80/QFP				1	1	2	8 通道	4 通道	8 通道	8 通道
	64/LQFP				1	1	2	8 通道	4 通道	8 通道	8 通道
9S12XS64	112/LQFP	64 KB	4 KB	4 KB	1	1	2	8 通道	4 通道	16 通道	8 通道
	80/QFP				1	1	2	8 通道	4 通道	8 通道	8 通道
	64/LQFP				1	1	2	8 通道	4 通道	8 通道	8 通道

下面的章节将以 S12X 系列中 80 引脚 QFP 封装 MC9S12XS128 作为蓝本深入阐述嵌入式开发所必备的硬件知识、软件知识及相关技术。

### 2.1.2 S12X 系列 MCU 型号标识

Freescale S12X 系列 MCU 的型号众多,但同一子系列的 CPU 核是相同的,多种型号只是为了适用于不同的场合。为了方便实际应用时选型或者订购,需要了解 Freescale MCU 芯片名称的含义,下面以 MC9S12XS128MAA 芯片进行说明:

MC

9

S12X

S

128

M

AA

①

②

③

④

⑤

⑥

⑦

① 产品状态。MC—Fully Qualified(合格);PC—Product Engineering(工程测试品)。在

实际应用中,通常都是选用 MC 类型的产品,如 MC9S12XS128MAA 等。

② 存储器类型标志。9 表示片内带闪存 Flash。

③ 微控制器家族。S12X 使用内核 CPU12X。

④ 子系列型号标志。S12XS 是 S12X 的一个子系列,故子系列名即为 S,一般把该子系列简称为 XS。

⑤ MCU 内 Flash 存储器大小(单位 KB)。128 表示内含 128 KB 的 Flash 存储器。

⑥ 工作温度范围标志。C 表示  $-40\sim 85^{\circ}\text{C}$ ;V 表示  $-40\sim 105^{\circ}\text{C}$ ;M 表示  $-40\sim 125^{\circ}\text{C}$ 。

⑦ 引脚个数与封装标志。AE = 64 LQFP(64 引脚 LQFP 封装);AA = 80 QFP(80 引脚 QFP 封装);AL = 112 LQFP(112 引脚 LQFP 封装)。选用某款芯片制作电路板时要特别注意封装形式。QFP 是四方扁平封装(Quad - Flat Package)、LQFP 是薄方封装(Plastic - Low - profile Quad - Flat Package)。

## 2.2 S12X 系列 MCU 的功能及存储器映像

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 了解芯片的基本性能、内部主要功能模块及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 了解芯片引脚的总体布局情况,掌握硬件最小系统电路。

⑤ 理解 MCU 第一个 C 语言工程的结构,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、机器码文件等工程文件。

⑦ 一定要有硬件评估环境,这是学习一款 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。

⑨ 理解第一个带有中断过程的 C 语言工程结构,理解主循环与中断两条程序执行路线各自的作用。

至此,以上学习过程已经覆盖学习一个新 MCU 硬件设计与软件编程的基本要素。完成了以上学习步骤,就完成了“基本入门”过程。随后,可以在此框架下,结合嵌入式构件方法,逐



个模块学习就方便了。

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。这两章完成了“基本入门”的前⑧步。“基本入门”的第⑨步(第一个中断例程)将在第 5 章阐述。为了规范编程,符合嵌入式软件工程的基本要求,提高硬件及底层驱动软件的可复用与可移植性,第 4 章阐述了基于硬件构件的嵌入式系统开发方法。

## 2.2.1 S12X 系列 MCU 的功能

S12X 系列 MCU 的内部结构框图如图 2-2 所示,主要特点简述如下:

① S12X 系列单片机的中央处理器 CPU12X 是 16 位 MCU,它的指令系统与 S12 兼容。CPU 工作频率最高可达 80 MHz。

② 使用范围为 0.5~16 MHz 的外部晶振,产生更高的单片机内部总线时钟,最高可达 40 MHz。外部时钟缺失时,内部提供自时钟方式,直到外部时钟恢复为止。

③ 64 KB、128 KB 或者 256 KB 的 Flash(也称 P-Flash)或者 ROM,4 KB 或 8 KB 的数据 Flash(也称 D-Flash 或 EEPROM),4 KB、8 KB 或者 12 KB 的 RAM。

④ 16 通道的高达 12 位精度的 A/D 采集模块,支持 8 位、10 位、12 位多种精度选择。支持 CAN2.0A、B 两种协议的控制器局域网 CAN,又叫 MSCAN(Motorola Scalable Controller Area Network),通信速率可达 1 Mbps。标准定时器模块 TIM(Standard Timer Module),8 个 16 位通道的输入捕捉和输出比较,1 个带着 8 位精度的 16 位计数器和 1 个 16 位的脉冲累加器。周期中断定时器 PIT(Periodic Interrupt Timer),多达 4 个带有溢出周期的独立的定时器,溢出周期可以在  $1 \sim 2^{24}$  个总线周期之间。多达 8 通道 8 位或 4 通道 16 位的脉冲宽度调制 PWM。2 个串行外设接口模块 SPI(Serial Peripheral Interface),可配置为 8 位或者 10 位数据大小,支持主机、从机两种模式。2 个串行通信接口 SCI(Serial Communication Interface)支持全双工或者半双工操作模式,可选用普通非归零码或者 IrDA 1.4 归零码。低功耗唤醒定时器,定时溢出周期从 0.2 ms~13 s,每两个可选周期之间间隔为 0.2 ms。

⑤ INT/XINT,中断模块,7 级中断嵌套,支持 7 个等级的中断优先级,用户可以编程设置中断的优先等级。

⑥ 单线后台调试模式接口(BKGD),增强的断点能力,允许单一的断点设置在线调试(在片内调试模块加了多于两个的断点)。

⑦ CRG(Clock and Reset Generation)模块,包括 COP 看门狗、实时中断以及时钟监视器等。

⑧ 片内电压调节器。包含带低电压中断方式的低电压检测、上电复位电路以及低电压复位。含有带隙(bandgap)参考电压,提高了系统的温度适应性。

⑨ 存储器映像控制(Memory Mapping Control)模块,实现 8 MB 存储空间连续寻址。

2 LIN/SCI	4~12 KB RAM		PWM 8B 8-ch.
MSCAN	64~256 KB Flash ECC		Timer 16B 8-ch.
SPI	4~8 KB DataFlash ECC		
GPIO	INT	CRG	4-ch. Periodic Interrupt Timer
	DBG		ATD 12B 8-16-ch.
FMPLL	S12X CPU		64/112 LQFP 80 QFP

图 2-2 S12X 系列 MCU 功能框图

## 2.2.2 S12X 系列 MCU 的存储器映像及特点

### 1. S12X 系列 MCU 存储器的分页机制

以 S12X 系列中 XS128 为例,从表 2-1 可以看出,XS128 的 RAM 为 8 KB、D-Flash 为 8 KB、P-Flash 为 128 KB。但 S12X 系列 CPU 的基本地址线是 16 位,决定其寻址范围为  $0x0000 \sim 0xFFFF$ ,寻址空间为  $2^{16}B = 64\text{ KB}$ ,即在大多数的情况下,CPU 运行只能“看到”这 64 KB 的存储空间。

如图 2-3 所示,S12X 系列 MCU 的 64 KB 地址空间被分为 I/O 寄存器、数据 Flash 存储器(D-Flash,也称为 EEPROM)、RAM 及程序 Flash 存储器(P-Flash,一般直接称为 Flash)4 个部分。其中  $0x0000 \sim 0x07FF$  是 2 KB 的 I/O 寄存器区域, $0x0800 \sim 0x0FFF$  是 2 KB 的 D-Flash 区域, $0x1000 \sim 0x3FFF$  是 12 KB 的 RAM 区域, $0x4000 \sim 0xFFFF$  是 48 KB 的 Flash 区域。

16 位地址线的优势在于程序短,指针变量使用 2 B,和 32 位机相比,代码效率高,节省内存空间;缺点是寻址空间只有 64 KB,这 64 KB 寻址空间又要分配给 I/O 设备、数据 Flash、RAM 和程序 Flash。所以在使用 16 位地址线又能扩展存储器空间的情况下,S12X 系列 MCU 中集成了 MMC(Memory Mapping Control,存储器映像控制)模块,采用分页管理机制,将寻址空间由 64 KB(16 位)扩展到 8 MB(23 位)。

MMC 模块的主要功能是:地址映射、控制 MCU 操作模式、多主体(MCU 和 BDM)优先级访问、内部资源的选择、控制内部总线(包括存储空间和外围资源)等。主体的局部地址空间(16 位)被转换成全局(物理)存储器地址空间(23 位)。

MMC 中有全局页面索引寄存器 GPAGE(Global Page Index Register),该寄存器最高位固定为 0,实际成为 7 位寄存器。MCU 通过 GPAGE 寄存器,使得 16 位地址扩展成 23 位。



23 位的全局地址由 7 位 GPAGE 值[22 : 16]和 CPU 局部地址[15 : 0]组成。同时,CPU 的指令系统中增加了专用的 23 位地址存/取指令。仅当 CPU 执行全局指令时才使用 GPAGE 寄存器。GPAGE 提供了使用 23 位全局地址访问 8MB 空间的一种方法。此时 8MB 的连续地址分配如图 2-4 所示,具体分配如下:

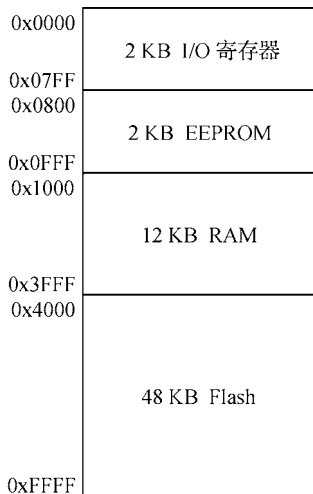


图 2-3 S12X 的 64 KB 地址空间分布



图 2-4 S12X 的 8 MB 扩展地址空间分布

0x00\_0000~0x00\_07FF

0x00\_0800~0x0F\_FFFF

0x10\_0000~0x13\_FFFF

0x14\_0000~0x3F\_FFFF

0x40\_0000~0x7F\_FFFF

2 KB I/O 寄存器地址空间

$64\text{ KB} \times 16 - 2\text{ KB} = 1\text{ MB} - 2\text{ KB}$  RAM 空间

$64\text{ KB} \times 4 = 256\text{ KB}$  D-Flash 空间

$64\text{ KB} \times 44 = 2\,816\text{ KB}$  未用空间

$64\text{ KB} \times 64 = 4\text{ MB}$  Flash 空间

同时为了使 D-Flash、RAM、P-Flash 便于管理及用户使用,MMC 增加了 3 个存储器页面寄存器,分别是数据闪存页面索引寄存器 EPAGE(Data FLASH Page Index Register)、RAM 页面索引寄存器 RPAGE(RAM Page Index Register)、程序页面索引寄存器 PPAGE(Program Page Index Register)来分别寻址对应的扩展区域。CPU 在 64 KB 寻址空间中开出几个窗口,利用上述几个页面寄存器,使得 CPU 可以随时将 64 KB 以外的存储空间映射到 64 KB 中窗口内,同时将暂时不用的那一块换出去,以此方法扩展 CPU12X 的寻址空间。同样,也可以利用这些页面寄存器来进行全局地址的访问。

下面以 RAM 部分为例阐述分页机制,关于 D-Flash 和 P-Flash 的分页机制见第 9 章 Flash 存储器在线编程。

如图 2-5 所示,地址范围为 0x1000~0x3FFF 的 RAM 被分为两个部分,其中 0x1000~

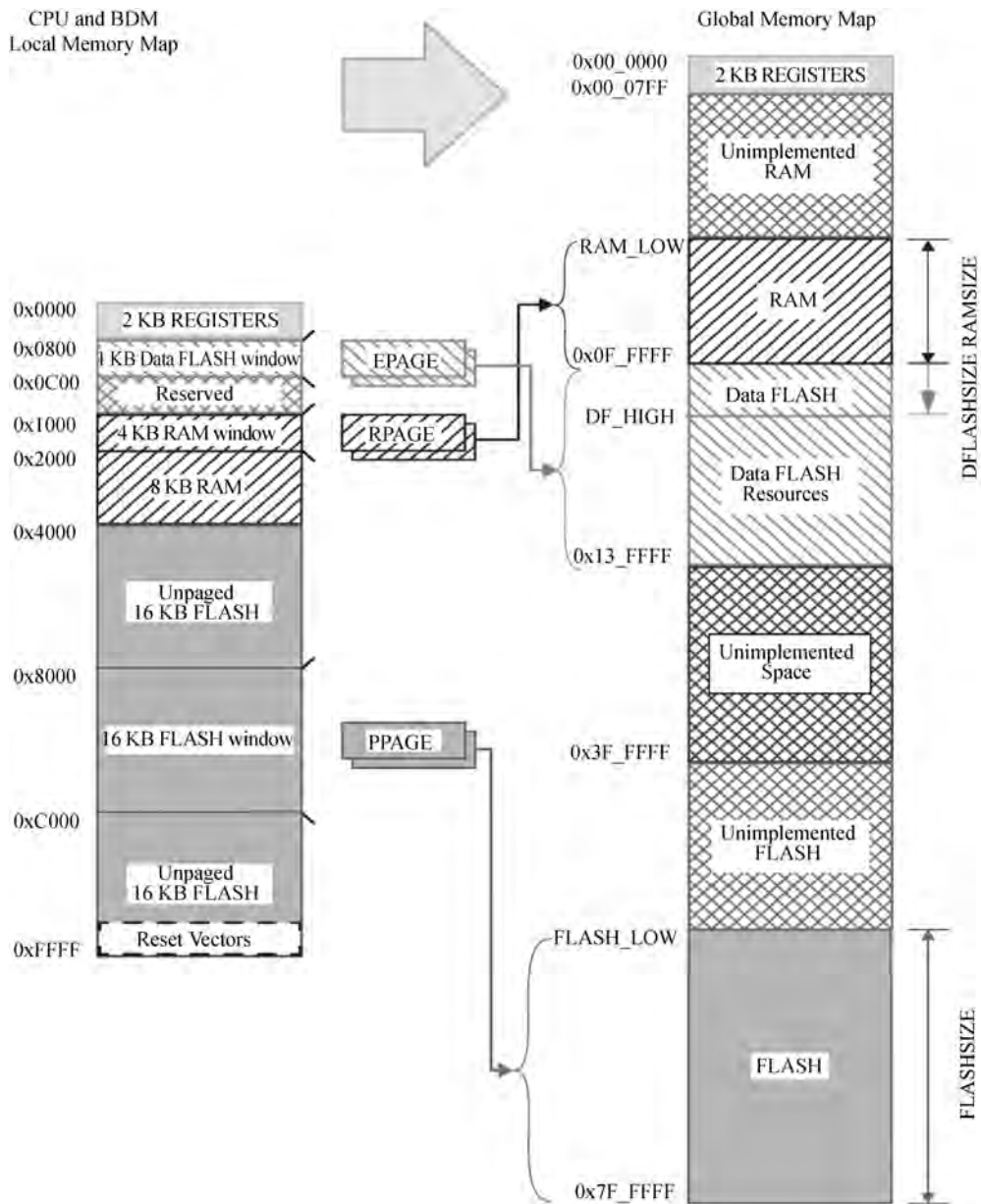


图 2-5 S12X MCU 寻址空间的扩展



0x1FFF 为 RAM 的分页窗口,大小为 4 KB,0x2000~0x3FFF 为 RAM 未分页区域,大小为 8 KB。在扩展的 8 MB 寻址空间中,RAM 区每一页的大小均为 4 KB,大小与 RAM 的分页窗口一样。

MMC 中的 RPAGE 寄存器是一个 8 位寄存器,允许通过这 8 个索引位,将总共  $2^8 = 256$  个 4 KB 页映射到位于 64 KB 寻址空间中的 RAM 窗口(0x1000~0x1FFF)中。所以扩展的 RAM 区大小为  $4 \text{ KB} \times 256 = 1 \text{ MB}$ 。由 RPAGE 寄存器值得到 23 位全局地址的方法如图 2-6 所示。23 位地址中的高 3 位固定为“000”,接下来的 8 位[19:12]为 RPAGE 的值,低 12 位[11:0]为 RAM 局部地址范围(0x000~0xFFF,4 KB)。

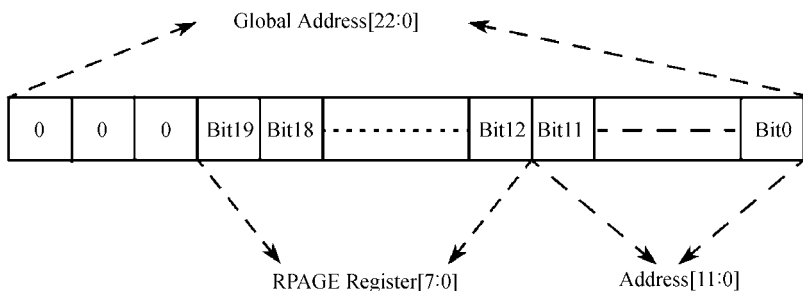


图 2-6 由 RPAGE 组成全局地址示意

但是,从图 2-5 看出,当 RPAGE 的值为 0x00 时,由 RPAGE 形成的地址中,0x00\_0000~0x00\_07FF 这 2 KB 被 I/O 寄存器占用了,所以不能作为 RAM 的扩展地址,实际可扩展的 RAM 大小为 1 MB-2 KB。

## 2. XS128 的分页存储器映像

对具体芯片来说,不是所有地址空间均有实际的物理存储器,例如,XS128 的 RAM 是 8 KB、D-Flash 也是 8 KB、P-Flash 是 128 KB。这些实际的物理存储器放在哪些地址上,在芯片设计时就确定了。

XS128 芯片 8 KB 的 RAM,使用全局地址范围是 0x0F\_E000~0x0F\_FFFF,RPAGE=0xFE~0xFF。芯片复位时,RPAGE 寄存器中的默认值为 0xFD,是个无效值,即访问地址 0x1000~0x1FFF 时会出错,产生非法地址中断。可以在单片机初始化时,将 RPAGE 寄存器值初始化为 0xFE,这样直接访问 0x2000~0x2FFF 和使用全局地址 0x0F\_E000~0x0F\_EFFF 访问是一样的。8 KB 的 D-Flash 使用全局地址范围是 0x10\_0000~0x10\_1FFF,EPAGE=0x00~0x07。128 KB 的 P-Flash 使用全局地址范围是 0x7E\_0000~0x7F\_FFFF,PPAGE=0xF8~0xFF。XS128 的分页存储器映像总结在表 2-2 中。实际编程时只能使用这些存储器地址资源,对这些地址之外的存储器地址操作没有意义,因为那些地址根本没有实际的物理存储器。

表 2-2 XS128 的分页存储器映像

存储器类型	大 小	GPAGE	全局页首地址	页面寄存器	每页大小
RAM (0x0F_E000~0x0F_FFFF)	8 KB	0x0F	0x0F_E000	RPAGE=0xFE	4 KB
			0x0F_F000	RPAGE=0xFF	
D-Flash (0x10_0000~0x10_1FFF)	8 KB	0x10	0x10_0000	EPAGE=0x00	1 KB
			0x10_0400	EPAGE=0x01	
			0x10_0800	EPAGE=0x02	
			0x10_0C00	EPAGE=0x03	
			0x10_1000	EPAGE=0x04	
			0x10_1400	EPAGE=0x05	
			0x10_1800	EPAGE=0x06	
			0x10_1C00	EPAGE=0x07	
P-Flash (0x7E_0000~0x7F_FFFF)	128 KB	0x7E	0x7E_0000	PPAGE=0xF8	16 KB
			0x7E_4000	PPAGE=0xF9	
			0x7E_8000	PPAGE=0xFA	
			0x7E_C000	PPAGE=0xFB	
		0x7F	0x7F_0000	PPAGE=0xFC	
			0x7F_4000	PPAGE=0xFD	
			0x7F_8000	PPAGE=0xFE	
			0x7F_C000	PPAGE=0xFF	

### 3. 逻辑地址与全局地址的转换

除了 16 位的局部地址和 23 位的全局地址,还会使用到逻辑地址来访问存储空间。XS128 的链接参数文件(Project.prm)中使用到的地址就是逻辑地址。逻辑地址由相应的存储器分页寄存器值[23:16]和 16 位的相应窗口地址[15:0]组成,例如 0xFC\_8000 为逻辑地址,此时 PPAGE 的值为 0xFC,0x8000 是 16 位 P-Flash 窗口地址范围 0x8000\_0xBFFF 的首地址。

P-Flash 的部分逻辑地址和全局地址的转换示意如图 2-7 所示。

由图 2-7 可以看出,逻辑地址范围为 0xFC\_8000~0xFC\_BFFF 的地址空间,对应的 PPAGE 的值为 0xFC。转化为相应的全局地址时,23 位地址的最高位[22]固定为 1,接下的 8 位为[21:14]为 PPAGE 的值,即为 0xFC。低 14 位为局部地址,范围为 0x0000~0x3FFF,故相应的全局地址范围为 0x7F\_0000~0x7F\_3FFF。

反过来,全局地址范围为 0x7F\_0000~0x7F\_3FFF 的地址空间,对应的 PPAGE 值为

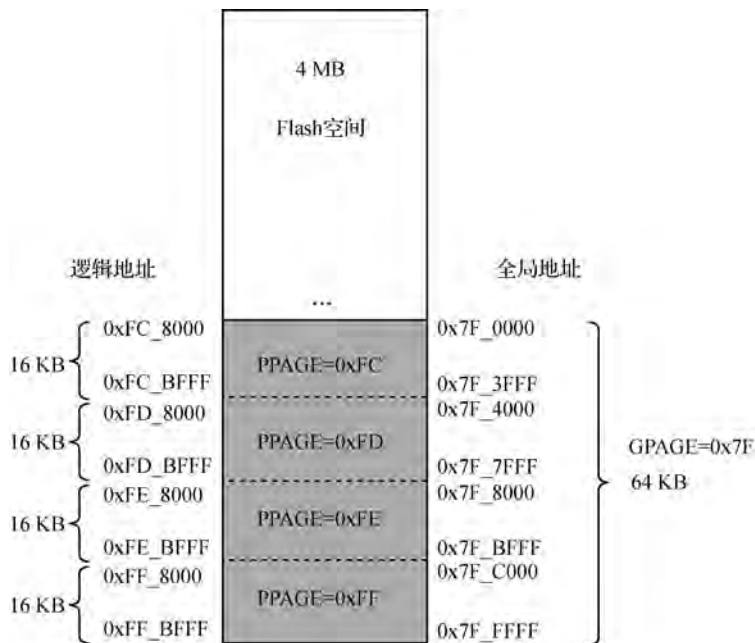


图 2-7 逻辑地址和全局地址的转换示意

[21:14]值,即 0xFC。因为[22]位为 1,故这是一个 P-Flash 页的全局地址,所以逻辑地址的低 16 位的变化范围为 P-Flash 窗口地址范围,即 0x8000~0xBFBB。故相应的逻辑地址范围 0xFC\_8000~0xFC\_BFFF。

关于 P-Flash 和 D-Flash 的详细地址说明,详见第 9 章 Flash 存储器在线编程。

#### 4. I/O 映像寄存器

所谓 I/O 映像寄存器,简称 I/O 寄存器,是指那些通过存储器地址访问的寄存器,它们不像寄存器 A、B、X、Y、SP、PC、CCRW 那样直接通过其“名”来使用它,而是通过其对应的地址来使用。要通过其“名”来使用,就必须用伪指令定义它们所占用的实际地址与“名”对应。附录 A 中给出了 XS128 的 I/O 映像寄存器简明列表。每个工程中 MC9S12XS128.h 文件含有 I/O 映像寄存器名与地址的对应关系,为了方便位操作,还给出了位定义。

## 2.3 XS128 的引脚功能及硬件最小系统

了解引脚功能并设计最小硬件系统是学习一个芯片的基本环节之一,本节给出 XS128 芯片 80 引脚 QFP 封装的引脚功能及硬件最小系统。



### 2.3.1 XS128(80 引脚 QFP 封装)的引脚功能

图 2-8 是 80 引脚 QFP 封装的 XS128 的引脚图。每个引脚都可能多个复用功能,有的引脚有两个复用功能,最多的有 5 个复用功能。系统设计时必须注意,一般情况下只能使用其中的一个功能。下面按为芯片服务与芯片提供服务的方式对引脚进行粗略分类。表 2-3 给出了需要服务的引脚,主要是芯片硬件最小系统引脚。芯片硬件最小系统的概念及电路图见 2.3.2 小节。

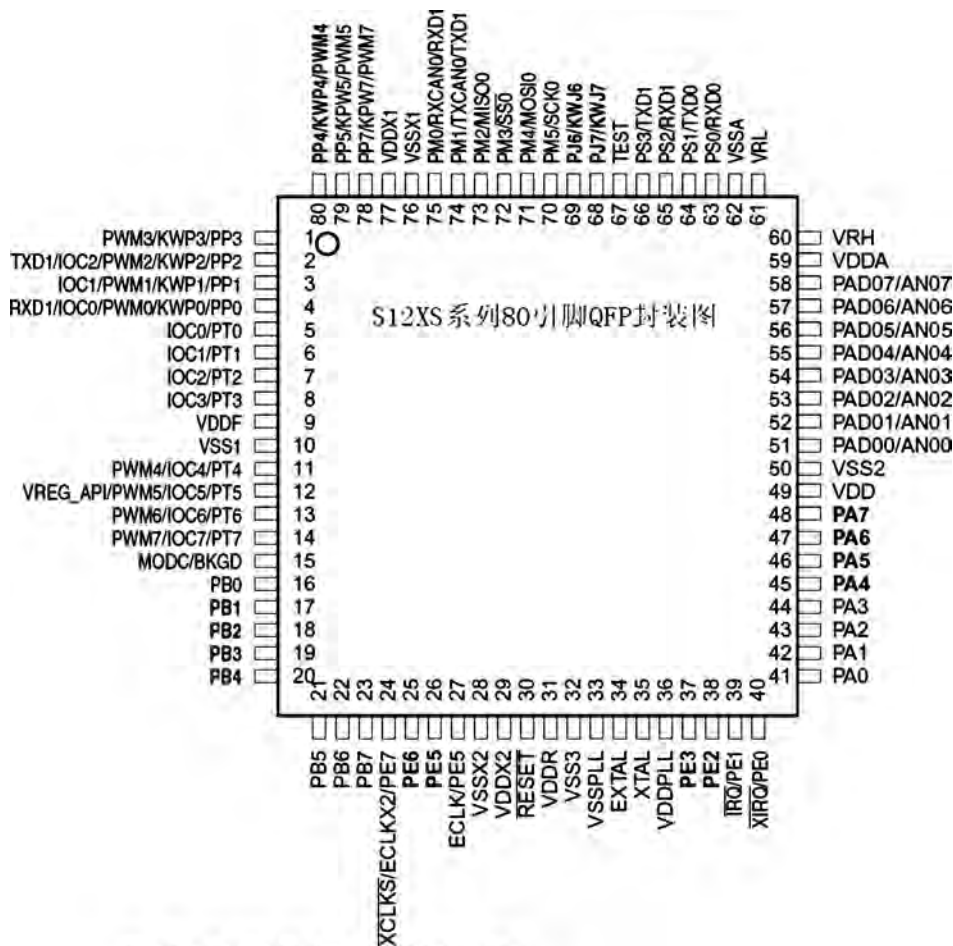


图 2-8 XS12880 引脚 QFP 封装图



## 1. 硬件最小系统引脚

XS128 芯片电源类引脚共有 16 个。芯片使用多组电源引脚分别为内部电压调节器、I/O 引脚驱动、A/D 转换电路等电路供电,内部电压调节器为内核、振荡器及锁相环 PLL 电路等供电。为了电源稳定,MCU 内部包含多组电源电路,同时给出多处电源引出脚,便于外接滤波电容。为了电源平衡,MCU 提供了内部相连的地的一处引出脚,供电路设计使用,如表 2-3 所列。

表 2-3 MCU 工作支撑引脚表

分 类	引脚名		引脚号	典型值/V	功能描述	备 注
电源类	外部输入	VDDR	31	5.0	为内部电压调节器供电	两个电源正极在芯片内部相连,两个电源负极在芯片内部相连
		VDDX1、VDDX2	77、29	5.0	为 I/O 引脚驱动供电	
		VSSX1、SSX2	76、28	0		
		VDDA	59	5.0	为 A/D 转换电路及内部电压调节器的部分电路供电	
		VSSA	62	0		
		VRH	60	5.0	A/D 转换参考电压	
		VRL	61	0		
	电压调节器输出	VDD	49	1.8	内核 CPU 电源	① VDD、VDDF、VDDPLL 为由 VDDR 经内电压调节器产生,引出目的外接滤波电容;② VSS1、VSS2、VSS3 在芯片内部相连
		VSS2、VSS3	50、32	0		
		VDDF	9	2.8	为 MCU 内部 Flash 供电	
		VSS1	10	0		
		VDDPLL	36	1.8	MCU 内部锁相环 PLL 电路及振荡器电源	
		VSSPLL	33	0		
复位	$\overline{\text{RESET}}$		30	复位引脚,是双向引脚。作为输入引脚,拉低可使芯片复位。作为输出引脚,上电复位期间有低脉冲输出,表示芯片已经复位完成		
晶振	EXTAL		34	晶振或时钟输入引脚		
	XTAL		35	振荡器输出引脚		
写入器	BKGD/MODC		15	BDM 通信引脚,写入调试器使用。 (该引脚有内部上拉电阻)		MODC=1,普通单片模式(一般运行使用);MODC=0,特殊单片模式(写入器使用)
工厂测试	TEST		67	测试引脚		为工厂测试保留
引脚个数统计			21			

复位引脚 RESET 是一个专用引脚,内部含有上拉电阻。空闲状态为高电平,低电平迫使芯片复位。在写入器电路中,该引脚被连接到标准的 6 芯 BDM 接口,以便写入器可以使

MCU 复位。

背景调试引脚 MODC/BKGD,用于连接调试器。实际系统运行时,此引脚为运行模式选择引脚,复位默认为高电平,进入普通单片模式。硬件设计时,可以使其有上拉电阻。

## 2. I/O 端口资源类引脚

除去需要服务的引脚外,其他引脚可以为实际系统提供 I/O 服务。芯片提供服务的引脚也可称为 I/O 端口资源类引脚。表 2-4 给出了 XS128(80 引脚 QFP 封装)59 个 I/O 引脚名、引脚号及功能描述。许多引脚具有复用功能。这些引脚在复位后,立即被配置为高阻状态,且为通用输入引脚,没有内部上拉电阻。需要注意的是,为了避免来自浮空输入引脚额外的漏电流,应用程序中的复位初始化例程需尽快使能上拉或下拉,也可改变不常用引脚的方向为输出,以使该引脚不再浮空。

比较特别的引脚:①PE1/IRQ 引脚是 IRQ 中断的输入源。如果外部可屏蔽中断 IRQ 的功能没有确立,该引脚不执行任何功能。②PE0/XIRQ 引脚是外部不可屏蔽中断 XIRQ 中断的输入引脚。③PE7/XCLKS 引脚,用于设置所采用的晶振工作模式。

表 2-4 I/O 端口资源表

端口名	引脚数	引脚名	引脚号	功能描述				
				第一	第二	第三	第四	第五
A	8	PA[7 : 0]	48~41	GPIO	—	—	—	—
B	8	PB[7 : 0]	23~16	GPIO	—	—	—	—
E	8	PE[7]	24	GPIO	$\overline{\text{XCLKS}}$	ECLK	—	—
		PE[6 : 5]	25,26	GPIO	—	—	—	—
		PE[4]	27	GPIO	ECLK	—	—	—
		PE[3 : 2]	37,38	GPIO	—	—	—	—
		PE[1]	39	GPIO	$\overline{\text{IRQ}}$	—	—	—
		PE[0]	40	GPIO	$\overline{\text{XIRQ}}$	—	—	—
T	8	PT7	14	GPIO	IOC7	PWM7	—	—
		PT6	13	GPIO	IOC6	PWM6	—	—
		PT5	12	GPIO	IOC5	PWM5	VREG_API	—
		PT4	11	GPIO	IOC4	PWM4	—	—
		PT[3 : 0]	8~5	GPIO	IOC[3 : 0]	—	—	—



续表 2-4

端口名	引脚数	引脚名	引脚号	功能描述				
				第一	第二	第三	第四	第五
S	4	PS3	66	GPIO	TXD1	—	—	—
		PS2	65	GPIO	RXD1	—	—	—
		PS1	64	GPIO	TXD0	—	—	—
		PS0	63	GPIO	RXD0	—	—	—
M	6	PM5	70	GPIO	SCK0	—	—	—
		PM4	71	GPIO	MOSI0	—	—	—
		PM3	72	GPIO	SS0	—	—	—
		PM2	73	GPIO	MISO0	—	—	—
		PM1	74	GPIO	TXCAN0	TXD1	—	—
		PM0	75	GPIO	RXCAN0	RXD1	—	—
P	7	PP7	78	GPIO	KWP7	PWM7	—	—
		PP[5 : 3]	79, 80, 1	GPIO	KWP[5 : 3]	PWM[5 : 3]	—	—
		PP2	2	GPIO	KWP2	PWM2	IOC2	TXD1
		PP1	3	GPIO	KWP1	PWM1	IOC1	—
		PP0	4	GPIO	KWP0	PWM0	IOC0	RXD1
J	2	PJ[7 : 6]	68, 69	GPIO	KWJ[7 : 6]	—	—	—
AD	8	PAD0[7 : 0]	58~51	GPIO	AN0 [7 : 0]	—	—	—
总数	59							

### 2.3.2 XS128 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,XS128 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、写入器接口电路。下面分别对这些电路给出简明分析。

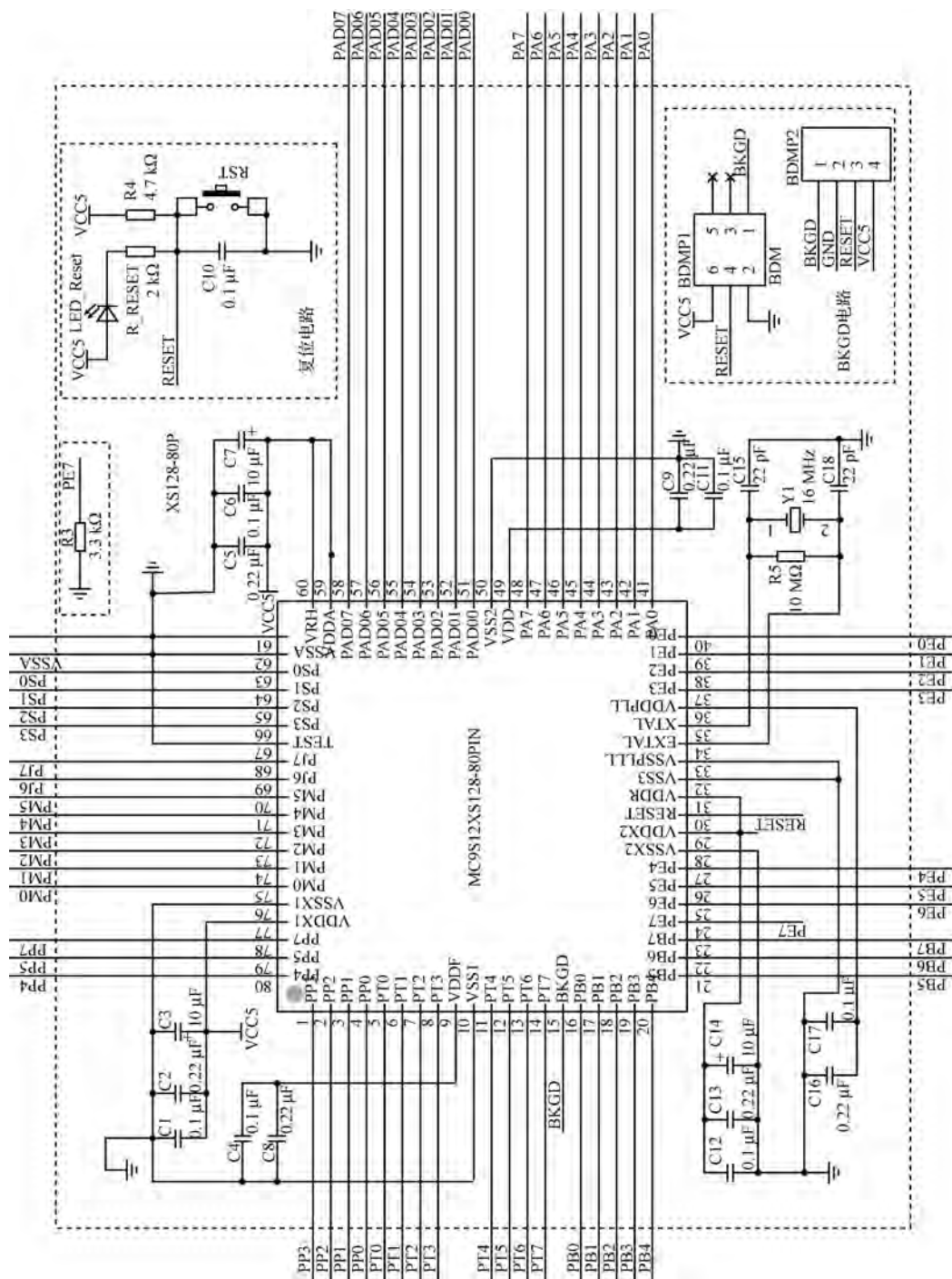


图 2-9 XS128 的硬件最小系统



绘制硬件最小系统原理图时,可以使用引脚的第一功能名称命名引脚的网标对 I/O 类功能引脚;若引脚具有 GPIO 功能,可以使用 GPIO 功能名命名网标。利用最小系统进行实际嵌入式系统功能原理图设计时,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。图 2-9 给出了 XS128(80 引脚 QFP 封装)的硬件最小系统。电路图在网上光盘中。凡是利用 XS128(80 引脚 QFP 封装)设计实际应用系统,该图一般不再更动。引出的网标供绘制其他功能构件使用。

### (1) 电源及其滤波电路

电路中需要大量的电源类引脚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声。

有关电源引脚的功能在 2.3.1 小节中已经介绍。有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。只有部分电源引脚是为了电源输入。其他的一些外接电容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。这些电容起到了滤波稳定电压的作用,一般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。

### (2) 复位电路

XS128 硬件最小系统的复位电路。复位意味着 MCU 一切重新开始。若复位引脚  $\overline{\text{RESET}}$  信号有效(低电平), $\overline{\text{RESET}}$  会产生一个低电平脉冲,MCU 复位。肉眼观察,复位电路的发光二极管会闪一下,这是正常复位情况。复位电路原理如下:正常工作时复位输入引脚  $\overline{\text{RESET}}$  通过 10 k $\Omega$  电阻接到电源正极,所以应为高电平。若按下复位按钮,则  $\overline{\text{RESET}}$  引脚接地,为低电平,导致芯片复位。需要注意的是,如果  $\overline{\text{RESET}}$  引脚被一直拉低,MCU 将不能正常工作。

### (3) BDM 接口电路

背景调试模式 BDM(Background Debug Mode)是由 Freescale 半导体公司自定义的片上调试规范,为开发人员提供了底层的调试手段。开发人员可以通过它初次向目标板下载程序,同时也可以通过 BDM 调试器对目标板 MCU 的 Flash 进行写入、擦除等操作。用户也可以通过它进行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。BDM 硬件调试插头的设计也非常简单,关键是要满足通信时序关系和电平转换要求。图 2-10 给出了一个常用的标准 BDM 调试插头(6 针)。由于早期的 USB 接口比较少见,所以大部分的 BDM 调试头采用了 6 针的封装。各个引脚信号的定义如表 2-5 所列。



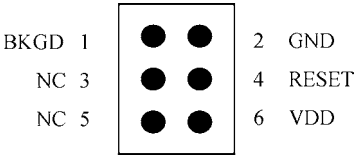


图 2-10 Freescale 的 BDM 调试头引脚定义

表 2-5 BDM 调试端口信号含义

BKGD	单线背景调试模式引脚
GND	接地
VDD	电源
RESET	目标机复位引脚

(4) 晶振电路及其工作模式选择电路

S12X 系列芯片的晶振可以工作在两个模式：一个是循环可控制的皮尔兹<sup>注1</sup> (Loop controlled Pierce, 简称 LCP) 模式；另一种是全振动皮尔兹 (full swing Pierce, 简称 FWP) 模式。这两种模式通过 XCLKS/PE7 引脚进行片选。该引脚的 XCLKS 功能可用于晶振模式的选择。当该引脚上拉，即 XCLKS=1 时，选择 LCP 模式，如图 2-11(a) 所示；反之，当该引脚下拉接地，即 XCLKS=0 时，选择 FWP 模式如图 2-11(b) 所示。但是，在 XCLKS=0 时，如果不外接晶振，EXTAL 可以直接外接频率信号，XTAL 脚不接。图 2-11 展示了前两种模式。图 2-9 给出的硬件最小系统电路，使用 XCLKS/PE7 引脚=0，采用图 2-11(b) 晶振接法。注意：对于图 2-11(b) 中的电阻 RS\*，当使用高频晶振时，可以换为 0Ω，即可以直接短接。

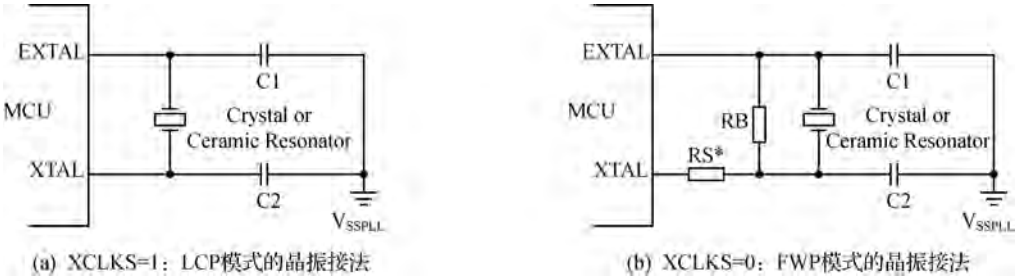


图 2-11 XCLKS 引脚设置与晶振接法

2.3.3 硬件最小系统的焊接与测试步骤

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随书网上光盘提供的资料)。根据原理图进行印刷电路板 PCB 的绘制，注意硬件最小系统的滤波电容尽量靠近芯片引脚，晶振 Q1 下方最好不要走线，这样系统的稳定性、抗干扰能力都会有所提高。铺地<sup>注2</sup>方式及各元器件参数可按照飞思卡尔官方参考手册中的推荐(见

注 1：皮尔兹晶振是一种简单的石英晶体振荡器，其中的晶体直接连接于有源器件(通常为双极性晶体管或场效应管)输入端与输出端之间。

注 2：“铺地”，PCB 电路板制图中的一个术语，为了提高系统的稳定性或者是增强电路板的韧性，常会推荐画完图后进行“铺地”操作。



附录 C)。

第一次绘制的电路板(硬件最小系统核心板)交付工厂制作完毕后,建议按照以下步骤进行硬件电路板的焊接和测试:

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 V5.0,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。

## 2.4 CPU12X 的内部寄存器

MC9S12 系列的各种型号 MCU 均使用 CPU12 家族 CPU。下面主要对 CPU12 中的寄存器功能进行介绍。CPU12 中有 6 个寄存器:16 位累加器 D(或 8 位累加器 A、B)、16 位变址寄存器 X、16 位变址寄存器 Y、16 位堆栈指针 SP、16 位程序计数器 PC 和 8 位条件码寄存器 CCR。这 6 个寄存器与端口部件的寄存器不同,它们可由指令直接访问或被控制器直接控制,而不需要像端口部件寄存器那样通过存储器映射方式来进行读/写。而 CPU12X 多出一个高 8 位的条件码寄存器 CCRH。6 个寄存器的结构示意图如图 2-12 所示。

### (1) 累加器 A、B(Accumulator)

在计算机中,对数据的处理被称为“操作”。操作通常有数学运算、逻辑运算、数据移动、跳转、数据的输入/输出等类型。操作所处理的数据被称为“操作数”。累加器 A、B 是 8 位通用寄存器,用来存放操作数和运算结果。数据读取时,累加器 A、B 用于存放从存储器读出的数据;数据写入时,累加器 A、B 用于存放准备写入存储器的数据。在执行算术、逻辑操作时,累加器首先存放一个操作数,执行完毕时累加器存放操作结果。累加器 A、B 是指令系统中最灵活的寄存器,各种寻址方式均可对之寻址。复位时,累加器的内容不受影响。有些指令结合了 A、B 两个累加器作为一个 16 位累加器(D)。绝大多数指令都可以交换使用累加器 A 和 B,也有一些特殊的指令不能交换(如 ABA、SBA 和 CBA),必须使用指定的累加器。

### (2) 变址寄存器 X、Y(Index Register)

16 位变址寄存器 X、Y 用做变址寻址。在变址寻址方式下,变址寄存器的内容加上 5 位、9 位或者 16 位的值,或者加上一个累加器中的内容得到指令操作的有效地址。变址寄存器 Y



图 2-12 CPU12X 寄存器结构示意图

在两个不同来源数据间的运算有特殊的用处。此外 CPU12X 还允许对 X、Y 进行完整的算术和逻辑操作。

### (3) 堆栈指针 SP(Stack Pointer)

CPU12X 提供自动堆栈程序,在调用子程序和中断发生的时候,栈用来保存系统内容,此外也可用来临时存储数据。堆栈可以寻址 64 KB 地址空间的任何地方。

SP 保存着堆栈最后使用的 16 位地址,通常 SP 初始化为应用程序的第一个指令。堆栈指针 SP 是指向下一个栈地址的 16 位寄存器,进栈时 SP 减 1,出栈时 SP 加 1。

当子程序被调用时,调用指令的地址经自动计算后放进栈中,通常子程序调用结束后程序返回指令执行。程序计数器保有先前栈值返回地址,调用程序返回后将在那个地址继续执行程序。

当发生中断时,当前指令执行完成,下一条指令的地址经计算后放进栈中,CPU 所有寄存器被放进栈中,然后 CPU 取得中断向量中的中断程序入口地址给程序计数器 PC,PC 由此地址执行中断服务例程。SP 寄存器就是一个中断栈结构。这个结构与 M68HC11 是一样的,而 CPU12X 增加了一个字节。

### (4) 程序计数器 PC(Program Counter)

程序计数器 PC 也是 16 位的,可寻址范围达 64 KB。PC 用于存放下条指令的地址,在执行转移指令时存放转移地址,在执行中断指令时存放中断子程序入口地址。复位入口地址也称复位向量地址或复位矢量地址(Reset vector address),意味着复位状态过后,PC 指向该处,从复位入口地址处执行程序。

### (5) 条件码寄存器 CCR(Condition Code Register)

CPU12 中条件码寄存器 CCR 是 8 位寄存器(见图 2-12 中 CCR 的低 8 位),其中有 5 个



状态指示符,两个中断控制码,一个 STOP 指令控制码。在 CPU12X 中条件码寄存器扩展为一个 16 寄存器 CCRW,其中低 8 位与 CPU12 相同,低 8 位中的前 3 位显示目前程序执行情况,这些位允许了中断嵌套,低级中断限制功用。分别介绍如下:

D7 停止模式(STOP)禁止位 S。清零 S 位将使能 STOP 指令,STOP 指令能够使振荡器停止。若在  $S=1$  时运行 STOP 指令,不会使振荡器停止,而实际效果则相当于空操作指令(NOP)。复位将设置 S 位为 1。

D6 非屏蔽中断允许位 X。非屏蔽中断用来处理主要的系统异常,比如电源掉电等。然而在系统上电和初始化后,使能非屏蔽中断却可以产生伪中断。X 位使得系统稳定后使能非屏蔽中断成为可能。复位时,一般将置 X 位为 1。只要 X 位为 1,通过 XIRQ 的中断请求就不能被识别和响应。清零 X 位将使能非屏蔽中断。只要 X 位被清为 0,软件就不能通过写 CCR 重设该位。当 XIRQ 中断在非屏蔽中断位设置为 0 后发生,X 位和 I 位都会自动设置来阻止其他中断。

D5 半进位标志位 H。执行加法指令(ADD)和带进位加法指令(ADC)时,如果相加的结果的低 4 位向上产生进位,即累加器中有 D3 向 D4 的进位,则 CPU 将半进位标志 H 置“1”。该标志对于一般运算是没有用的,但在二进制编码的十进制(BCD)数的运算中则很有用,由于 BCD 码是以 4 位二进制数来表示一位十进制数,所以在 BCD 码算术运算中,半进位标志 H 记录的是一位十进制数的进位。做十进制调整(DAA)时,要用到 H 和 C 的状态来判断是否需要调整。

D4 可屏蔽中断使能位 I。 $I=1$  表示禁止可屏蔽中断; $I=0$  表示允许可屏蔽中断。复位时,该位被置“1”,可用 CLI 指令开中断。如果允许中断,当中断产生后,I 位被自动置“1”以保证在该中断服务例程中防止其他中断,置“1”的执行发生在寄存器入栈之后中断服务例程的第一条指令之前。一般来说,RTI 指令在中断服务例程的最后,当 RTI 执行后 I 位被自动清“0”以开放可屏蔽中断。进入中断服务例程后可以将 I 位清“0”来允许可屏蔽中断,但是这样容易导致中断嵌套,需小心使用。

D3 负标志位 N。CPU 在运算过程中,如果产生负结果,则将负标志 N 位置“1”。该位用于带符号位的运算,数据位的最高位 D7 作为符号位,如果  $D7=1$ ,说明数据为负;如果  $D7=0$ ,说明数据为正。如果寄存器或存储单元的 D7 为 1,那么把寄存器或存储单元的内容送入累加器 A 时,就会使 N 置“1”。如果寄存器或存储单元的 D7 为 0,那么把寄存器或存储单元的内容送入累加器 A 时,就会使 N 清 0。所以,负标志 N 也可用于检查有关寄存器或存储单元 D7 位的状态。

D2 零标志位 Z。在 CPU 运算过程中,如果数据或运算结果为 0,则零标志位 Z 置 1;否则清 0。比较指令执行了内部减法,而条件码显示减法的结果,变址寄存器(X,Y)影响着 Z 位。而这些操作仅能决定等于或不等于。

D1 溢出标志位 V。二进制补码溢出时置位。有符号跳转指令 BGT、BGE、BLE 和

BLT 使用该标志。

D0 进位/借位标志 C。当进行加法运算时,在最高位 D7 上有进位,或在进行减法运算时 D7 需要向更高位借位,CPU 会将进位/借位标志 C 置 1,否则清 0。一些指令如位测试、跳转、移位等指令也会影响该标志位。

CPU12X 中还有 IPL[2:0]位,此位允许中断嵌套,但限制同级或低级嵌套,IPL 位通过标准中断栈处理程序放进栈中,新的 IPL 位通过高优先权的最高级别中断请求通道,复制到 CCR 中。当接到中断矢量时开始复制,通过执行 RTI 指令先前的 IPL 被自动重新保存。

## 2.5 CPU12X 的寻址方式

寻址方式确定了 CPU12 家族如何访问内存位置。这部分将讨论多种寻址方式及如何运用它们。寻址方式是 CPU12 家族指令中一个隐式的部分。CPU12 与 CPU12X 的变址方式没有什么区别。

除了内在寻址方式以外每个寻址方式都产生一个 16 位有效的地址,当存储器引用部分指令的时候使用地址。有效地址的计算不需要额外的执行周期。

### (1) 隐含寻址方式(Inherent Addressing Mode)

隐含寻址指令中只有操作码,没有操作数。很明显该指令本身已经隐含了操作数所在地址的寻址方式。这类指令一般是单字节指令。

例如:

```
NOP      ;空操作
INX      ;操作数是 CPU 寄存器 X 的内容
```

### (2) 立即寻址方式(Immediate Addressing Mode)

立即寻址方式是指由指令直接给出操作数的寻址方式。立即寻址指令通常是对立即数和累加器内容或者变址寄存器内容进行操作。下列指令属于立即寻址方式:

```
LDAA     # $ 55      ;把 $ 55 放入累加器 A 中
LDX      # $ 1234    ;把 $ 1234 放入变址寄存器 X 中
LDY      # $ 67      ;把 $ 0067 放入变址寄存器 Y 中
```

特别说明:在 Freescale MCU 的指令系统中,规定在数字前加“\$”表示十六进制数,加“%”表示二进制数,无前缀表示十进制数。另外,指令中的数据前加“#”表示立即数。

### (3) 直接寻址方式(Direct Addressing Mode)

直接寻址方式中操作数为单字节地址,高位地址默认为 \$00,因此,只能用来对 \$00~\$FF 这 256 个存储单元进行寻址。S12X 单片机默认内存分配中,\$00~\$FF 这一段是 I/O 寄存器地址,所以同 I/O 寄存器打交道时可以直接使用直接寻址。



#### (4) 扩展寻址方式(Extended Addressing Mode)

扩展寻址指令可访问存储器中的任何地址。它是相对于直接寻址方式而言,其寻址大小为 64 KB,比直接寻址范围大得多。下面的指令属于扩展寻址方式:

LDAA            \$ F03B    ;把存储单元 \$ F03B 地址中的值被放入累加器 A 中

特别说明:实际编程时,程序员不必考虑是直接寻址还是扩展寻址,汇编会自动识别,其主要区别在于汇编产生的指令长度不一样。

#### (5) 相对寻址方式(Relative Addressing Mode)

相对寻址方式只用于转移指令。短和长的条件转移指令只使用相对寻址方式,但转移指令中的位操作指令(置位(BRSET)转移和清除位(BRCLR)转移)使用多种寻址方式,包括相对寻址方式。

短转移指令包含了一个 8 位操作码和一个有符号的 8 位偏移。长转移指令包含了一个前 8 位字节、一个 8 位的操作码和一个有符号 16 位偏移。

每个条件转移指令都要测试一下条件码寄存器中的某些状态位。如果位在某一状态区域,则要在偏移形成一个有效的地址后,偏移才被放到下一个内存位置,并且一直执行进行到那个地址。如果位不在某一状态区域,紧跟转移指令的指令将继续执行下去。

位状态转移会测试记忆字节中的位是否在一个特定的状态。各种寻址方式可用于访问内存位置。有一个 8 位伪操作数用来测试位。如果在内存中符合 a1 的每个位要么是置位(BRSET)要么是清除位(BRCLR),则偏移形成有效的地址,并且继续执行进行到那个地址,然后将有一个 8 位偏移加到内存中的下一个地址。如果内存中的所有符合 a1 的位都不处于指定的状态,紧跟转移指令的指令将继续执行下去。

8 位、9 位和 16 位偏移被记为补码来支持内存中的向上转移和向下转移。短转移的数值范围是 \$ 80(-127)~\$ 7F(128)。循环控制指令支持 9 位偏移,它允许的地址范围为 \$ 100(-256)~\$ 0FF(255)。这长转移偏移值的数据范围是 \$ 8000(-32768)~\$ 7FFF(32767)。如果偏移是 0,不管是否包含了测试程序,CPU12 会立刻跟着转移指令执行。

因为偏移是在转移指令的结尾,用一个负偏移值就能使程序计数器(PC)指向操作码并开始循环。例如,一个转移(BRA)指令总是包含两个字节,所以用偏移 \$ FE 会进入无限循环。同理,长转移(LBRA)指令总是用 \$ FFFC 的偏移量也会进入无限循环。

指向一个操作码的偏移会引起一个位条件转移的重复执行,直到某一特定的位条件被满足。因为依据在内存中访问字节的寻址方式不同,位条件转移可以包括 4、5 或 6 字节,所以引起循环的的偏移量可以变化。

#### (6) 变址寻址方式(Indexed Addressing Mode)

变址寻址方式在指令操作码后可以加 0~2 个字节作为额外字节。后字节和扩展要做以下的任务:



- ① 指出哪个变址寄存器被使用；
- ② 决定累加器中的价值是否被用作偏移；
- ③ 启用自动的前置或后置的递增或递减；
- ④ 指定增加或递减的大小；
- ⑤ 列举 5 位、9 位或 16 位有符号偏移量的使用。

这种方法消除了 X 和 Y 寄存器的差别，大大提高了变址寻址方式的执行能力。

CPU12 家族中变址寻址方式的主要优点是：

- ① 在所有变址操作中，堆栈指针可以用来作为变址寄存器。
- ② 除了自增自减以外的所有方式中，程序计数器可以用作变址寄存器。
- ③ A、B 或 D 累加器可用于累加偏移量。
- ④ 自动前置或后置的由  $-8 \sim 8$  递增或递减。
- ⑤ 5 位、9 位或 16 位的选择不断地改变偏移。
- ⑥ 使用两个新变址寻址方式，即 16 位偏移间接变址方式及累加器 D 偏移间接变址方式。

所有的变址寻址方式使用 16 位 CPU 寄存器和额外的信息来创建一个有效地址。在大多数情况下，有效地址指明受到操作的内存位置。在一些变址寻址方式的变化中，有效地址列举了操作中指向内存位置的值。

变址寻址方式指令使用后字节来列举变址寄存器(X 和 Y)、堆栈指针(SP)或程序计数器(PC)来作为基址寄存器，进一步给有效地址的形成方式分类。有一组特殊的指令可使这个有效地址传送到变址寄存器中，以支持未来的计算：传送有效地址到堆栈指针(LEAS)、传送有效地址到 X 寄存器(LEAX)、传送有效地址到 Y 寄存器(LEAY)。

### 1) 5 位偏移量变址方式(5 - Bit Constant Offset Indexed Addressing)

该变址寻址方式用的是在指令后字节中的 5 位有符号偏移。这种短偏移被加到基址寄存器(X、Y、SP 或 PC)形成指令中实际有效地址。这给基址寄存器  $-16 \sim +15$  范围。尽管其他变址寻址方式允许 9 位或 16 位的偏移，这些方式也因为这额外的指令信息而需要额外的扩展字节。大多数的变址在实际的程序中使用适合最短的 5 位变址寻址的偏移量。

例如：

```
LDAA      0,X;将地址 $ 1000 中的数据读取到累加器 A 中,偏移为 0
STAB      -8,Y;将累加器 B 中的数据存储到地址 $ 1FFB($ 2000 - $ 8)
```

对于上述例子，假设执行前 X 的值为 \$ 1000，Y 的值为 \$ 2000。5 位偏移量变址方式不改变变址寄存器的值，因此执行指令后 X 的值仍然是 \$ 1000 且 Y 的值仍然是 \$ 2000。在第一个例子中，地址 \$ 1000 的数据将被读取到累加器 A 中。在第二个的例子中，累加器 B 中的数据被储存到地址 \$ 1FF8(\$ 2000 - \$ 8)。



## 2) 9 位偏移量变址方式 (9 - Bit Constant Offset Indexed Addressing)

变址寻址在指令中使用 9 位有符号偏移, 偏移范围为  $-256 \sim +255$ 。这种短偏移加到基址寄存器(X、Y、SP 或 PC)将形成指令中实际有效地址。

假设 X 中的值为 \$1000, Y 中的值为 \$2000, 则

LDAA    \$FF, X            ; 将变址寄存器 \$10FF ( $X + \$FF$ ) 地址内容取到累加器 A 中  
LDAB    - 20, Y           ; 将变址寄存器 \$1FEC ( $Y - 20$ ) 地址内容取到累加器 B 中

## 3) 16 位偏移量变址方式 (16 - Bit Constant Offset Indexed Addressing)

变址寻址在指令中使用 16 位有符号偏移。寻址范围可以达到 64KB 即整个 16 位的寻址空间。这种长偏移加到基址寄存器(X、Y、SP 或 PC)以形成指令中实际有效地址。

## 4) 16 位间接变址方式 (16 - Bit Constant Indirect Indexed Addressing mode)

该变址寻址方式将 16 位偏移量加到基址寄存器从而形成内存地址, 并从此内存取出新的地址(间接地址), 以便进行相关操作。例如:

LDAA    [10, X]

该指令把 10 和 X 相加, 结果是操作数的地址指针。例如,  $X = \$1000$ , \$2000 被保存在 \$100A 和 \$100B 中。指令先把 10 加上 X, 赋给 \$100A, 然后地址指针 \$2000 读取 \$100A 的内存。最后, \$2000 的值被读取并保存在累加器 A 中。

## 5) 自增自减变址寻址方式 (Auto Pre/Post Decrement/Increment Indexed Addressing)

该变址寻址模式提供了 4 种方式自动改变基址寄存器值, 变址寄存器可以先自增或自减, 也可以后自增或自减, 基址寄存器可以是 X、Y 和 SP。例如:

STAA    1, - SP           ; 相当于先执行  $SP - 1 \rightarrow SP$  再执行 PSHA  
STX     2, - SP           ; 相当于先执行  $SP - 2 \rightarrow SP$  再执行 PSHX  
LDX     2, SP +           ; 相当于先执行 PULX 再执行  $SP + 2 \rightarrow SP$   
LDAA    1, SP +           ; 相当于先执行 PULX 再执行  $SP + 1 \rightarrow SP$

## 6) 累加器偏移变址寻址方式 (Accumulator Offset Indexed Addressing mode)

该变址寻址方式的有效地址是由基址地址加上在累加器中的无符号偏移量形成。基址寄存器中的内容是一般不变, 可以是 X、Y、SP 和 PC, 累加器可以是 8 位的 A 或 B, 也可以是 16 位的 D。例如:

LDAA    B, X            ; 将地址  $B + X$  中的值读取到累加器 A 中

## 7) 累加器 D 间接寻址 (Accumulator D Indirect Indexed Addressing mode)

该寻址方式将累加器 D 中的值加到基址寄存器中以形成有效地址, 该有效地址并不是实际操作数的地址, 而是指向内存单元的指针。例如:



```

.....
JMP      [D,PC]
GO1:     PLACE1
.....
GO2:     PLACE2

```

其中, PLACE1 和 PLACE2 表示相关代码段。

程序在执行到 JMP [D,PC] 指令时, 会把 D 和 PC 相加形成跳转地址, 假设此时 D 中的值为 \$ 0002, 那么计算出来的跳转地址将是  $PC + \$ 0002$ ; 而  $PC + \$ 0002$  正好是标号 GO1 的地址, 这样 JMP [D,PC] 指令执行后, 程序将会跳转到 GO1 执行 PLACE1 处的代码。请注意: D 中具体值是由先前的程序计算得出。如果先前计算出的 D 值为 \$ 01B2, 而标号 GO2 相对于 JMP [D,PC] 指令的偏移也正好是 \$ 01B2, 那么程序将跳转到 GO2, 去执行 PLACE2 处的代码段。

### (7) 全局寻址方式

CPU12 核心的物理地址空间限制了 64 KB [15 : 0] 的物理地址。带有全局页索引寄存器的 CPU12X 核心结构能够用 7 个全局页索引位将 addr[22 : 0] 的 8 MB 地址集成到 64 KB 的内存映像 addr[22 : 0] 中, 这是 GPAGE 和 addr[15 : 0] 之间的结果。

以 G 开始的新指令如 (GLDAA, GSTAA...) 就用于此。

GLDAA: (G(M)  $\Rightarrow$  A)      从全局内存中读取数据到累加器 A

GLDAA 的寻址方式和 LDAA 相同, 唯一不同的是存储器地址 (64 KB) 都换成了全局内存地址 (8 MB)。这条原则适用于所有的全局指令。

## 2.6 CPU12X 指令系统概要

本节把 CPU12X 指令系统概括为 5 大类, 分别是数据传送类、算术运算类、逻辑运算与位操作类、程序控制类及其他类, 如表 2-6 所列。随后各小节给出指令系统简明功能概要, 供读者基本浏览。

CPU12X 指令是 CPU12 指令集的超集。为 CPU12 编写的代码可以被重新编译并且不需要变化在 CPU12X 上运行。该 CPU12X 提供了扩展功能, 提高代码效率。有两种 CPU12 的基本实现, 原始的 CPU12 和较新的 CPU12X。CPU12X 具有更高级的指令集, 对于现有的指令在逐周期访问的细节中有小的差异 (一些总线周期的顺序为适应实现指令队列方式的差异而改变)。这些细微的差别对于大多数用户是透明的。

在 CPU12 家族架构中, 所有内存和输入/输出 (I/O) 都映射在一个共同的 64 KB 地址空间 (内存映射 I/O), 这使得同一组指令集用于访问内存、I/O 和控制寄存器。通用存取、传输、交换和转移指令促进内存和外围设备的数据交换。



CPU12 家族有完整的 8 位和 16 位数学指令,包括与 8 位、16 位和一些较大的操作数相关的有符号和无符号运算的指令、除法指令和乘法指令。特殊算术和逻辑指令辅助堆栈操作、变址、二进制与十进制编码(BCD)的计算和条件码寄存器的操作。也有专门用于乘法累加运算、表插补指令和特别涉及数学计算的模糊逻辑运算指令。

表 2-6 CPU12X 指令集

类 型	保留字	含 义
数据 传送类	GLDAA, GLDAB, GLDD, GLDS, GLDX, GLDY	全局取数到 A、B、D、SP、X、Y
	LDAA, LDAB, LDD, LDS, LDX, LDY	取数到 A、B、D、SP、X、Y
	LEAS, LEAX, LEAY	取有效地址到 SP、X、Y
	GSTAA, GSTAB, GSTD, GSTS, GSTX, GSTY STAA	存 A、B、D、SP、X、Y 到全局内存
	STAB, STD, STS, STX, STY	存 A、B、D、SP、X、Y 到内存
	PSHA, PSHB, PSHC, PSHCW, PSHD, PSHX PSHY, PULA, PULB, PULC, PULCW, PULD PULX, PULY	进栈、出栈
	EXG, XGDX, XGDY, DAA	交换、转化为十进制数
	TAB, TAP, TBA, TFR, TFR, TPA, TSX, TSY	寄存器之间传送
	MOVB, MOVW	存储器单元之间传送
算术 运算类	ABA, ABX, ABY, ADDB, ADCA, ADCB, ADED, ADEX, ADEY, ADDA, ADDB, ADDX, ADDY	加法
	SBA, SUBA, SUBB, SUBD, SUBX, SUBY, SBCA, SBCB, SBED, SBEX, SBEY,	减法
	EMUL, EMULS, MUL, EDIV, EDIVS, FDIV, IDIV, IDIVS	乘法/除法
	INC, INCA, INCB, INCW, INCX, INCY, INS INX, INY, DEC, DECA, DECB, DECW, DECX, DECY, DES, DEX, DEY	加 1/减 1
	COM, COMA, COMB, COMW, COMX, COMY, NEG, NEGA, NEGB, NEGW, NEGX, NEGY	求反
	CBA, CMPA, CMPB, CPD, CPS, CPX, CPY, CPED, CPES, CPEX, CPEY	比较
	CLR, CLRA, CLRB, CLRW, CLRX, CLRY	清 0
	TST, TSTA, TSTB, TSTW, TSTX, TSTY, TXS, TYS	测试是否为 0

续表 2-6

类 型	保留字	含 义
逻辑运算与位操作类	ANDA, ANDB, ANDX, ANDY, ANDCC, ORAA, ORAB, ORCC, ORX, ORY, EORA, EORB, EORX, EORY	相、与、异或
	CLC, SEC, CLI, SEI, CLV, SEV, BCLR, BSET	清位或置位
	BITA, BITB, BITX, BITY	位测试
	ASL, ASLA, ASLB, ASLD, ASLW, ASLX, ASLY	算术左移
	LSL, LSLA, LSLB, LSLD, LSLW, LSLX, LSLY	逻辑左移
	ASR, ASRA, ASRB, ASRW, ASRX, ASRY	算术右移
	LSR, LSRA, LSRB, LSRD, LSRW, LSRX, LSRY	逻辑右移
	ROL, ROLA, ROLB, ROLW, ROLX, ROLY	循环左移
	ROR, RORA, RORB, RORW, RORX, RORY	循环右移
程序控制类	BCC, BCS, BVC, BVS, LBCC, LBCCS, LBEQ, LBGE, LBGT, LBHI, LBHS, LBLE, LBLO, LBLB, LBLT, LBMI, LBNE, LBPL, LBVC, LBVS	长、短标志位测试转移
	BEQ, BHI, BHS, BLO, BLS, BNE, BGE, BGT, BLE, BLT, BMI, BPL	有、无符号比较后转移
	BRCLR, BRSET, RTI, SYS, SWI, TRAP	位测试转移、中断指令
	BRA, LBRA, JMP	无条件相对、绝对地址转移
	IBEQ, IBNE, DBEQ, DBNE, TBEQ, TBNE	运算、测试条件转移
	BSR, CALL, JSR, RTC, RTS	跳转与子程序调用
	STOP, WAI, BGND, BRN, LBRN, NOP	停止与等待及后台模式与空操作
其他指令	BTAS, EMIND, EMINM, MINA, MINM, EMAXD, EMAXM, MAXA, MAXM, EMACS, ETBL, TBL, SEX, MEM, REV, REVW, WAV	

## 2.6.1 数据传送类指令

### 1. 存取数据指令

取数指令将存储器内容复制到累加器或寄存器。存储器的内容不会因为操作而改变。取数指令(但不是 LEA\_指令)影响条件码位,所以不需要单独的测试指令来检查被取的数值处于某种状态、为负或为 0 等。

存数据指令功能为复制 CPU 寄存器的内容到存储器。寄存器和累加器的内容不会因为操作而改变。存数据指令自动更新 N 和 Z 条件码位。表 2-7 是取数据和存数据指令摘要。



表 2-7 存取数据指令

指 令	功 能	操 作
取 数 指 令	GLDAA	取全局地址中的值到寄存器 A $(M) = > A$
	GLDAB	取全局地址中的值到寄存器 B $(M) = > B$
	GLDD	取全局地址中的值到寄存器 D $(M : M + 1) = > (A : B)$
	GLDS	取全局地址中的值到寄存器 SP $(M : M + 1) = > SPH : SPL$
	GLDX	取全局地址中的值到寄存器 X $(M : M + 1) = > XH : XL$
	GLDY	取全局地址中的值到寄存器 Y $(M : M + 1) = > YH : YL$
	LDAA	取局部地址中的值到寄存器 A $(M) = > A$
	LDAB	取局部地址中的值到寄存器 B $(M) = > B$
	LDD	取局部地址中的值到寄存器 D $(M : M + 1) = > (A : B)$
	LDS	取局部地址中的值到寄存器 SP $(M : M + 1) = > SPH : SPL$
	LDX	取局部地址中的值到寄存器 X $(M : M + 1) = > XH : XL$
	LDY	取局部地址中的值到寄存器 Y $(M : M + 1) = > YH : YL$
	LEAS	取有效地址到 SP 有效地址 $= > SP$
	LEAX	取有效地址到 X 有效地址 $= > X$
	LEAY	取有效地址到 Y 有效地址 $= > Y$
存 数 指 令	GSTAA	将 A 的值存入全局地址中 $(A) = > M$
	GSTAB	将 B 的值存入全局地址中 $(B) = > M$
	GSTD	将 D 的值存入全局地址中 $(A) = > M, (B) = > M + 1$
	GSTS	将 SP 的值存入全局地址中 $(SPH : SPL) = > M : M + 1$
	GSTX	将 X 的值存入全局地址中 $(XH : XL) = > M : M + 1$
	GSTY	将 Y 的值存入全局地址中 $(YH : YL) = > M : M + 1$
	STAA	将 A 的值存入局部地址中 $(A) = > M$
	STAB	将 B 的值存入局部地址中 $(B) = > M$
	STD	将 D 的值存入局部地址中 $(A) = > M, (B) = > M + 1$
	STS	将 SP 的值存入局部地址中 $(SPH : SPL) = > M : M + 1$
	STX	将 X 的值存入局部地址中 $(XH : XL) = > M : M + 1$
	STY	将 Y 的值存入局部地址中 $(YH : YL) = > M : M + 1$

## 2. 传送和数据交换指令

数据传送指令功能为复制一个寄存器或累加器的内容到另一个寄存器或累加器中。从一

个寄存器传送到另一个寄存器(TFR)是一种通用的传送数据指令,但其他助记符只能被兼容的 CPU11 接受。从 A 到 B 的传送(TAB)和从 B 到 A 的传送(TBA)的指令以与 CPU11 指令相同的方式影响 N、Z 和 V 条件码位。TFR 指令不影响条件码位。

符号扩展的 8 位操作数(SEX)指令是通用传送指令中的一种特殊情况,用于符号扩展的 8 位二进制补码,使这些数可以在 16 位操作中使用。8 位操作数是从累加器 A、累加器 B 或条件码寄存器中被复制到累加器 D、变址寄存器 X、变址寄存器 Y 或栈指针中。所有的 16 位结果的高字节位被赋予了 8 位数的最高有效位(MSB)的值。

数据交换指令交换一对寄存器或累加器的内容。当在一个 EXG 指令中第一个操作数是 8 位并且第二个操作数是 16 位时,零扩展操作是在 8 位寄存器上进行的,因为它被复制到 16 位的寄存器中。表 2-8 是传送和交换数据指令摘要。

表 2-8 传送和数据交换指令

指 令		功 能	操 作
传 送 指 令	TAB	将 A 的值传送到 B 中	$(A) = > B$
	TAP	将 A 的值传送到 CCR 中	$(A) = > CCR$
	TBA	将 B 的值传送到 A 中	$(B) = > A$
	TFR	将寄存器值传送到寄存器中	$(A, B, CCR, D, X, Y \text{ 或者 } SP) = > (A, B, CCR, D, X, Y \text{ 或者 } SP)$
	TPA	将 CCR 值传送到 A 中	$(CCR) = > A$
	TSX	将 SP 的值传送到 X 中	$(SP) = > X$
	TSY	将 SP 的值传送到 Y 中	$(SP) = > Y$
	TXS	将 X 的值传送到 SP 中	$(X) = > SP$
	TYS	将 Y 的值传送到 SP 中	$(Y) = > SP$
交 换 指 令	EXG	交换寄存器和寄存器的值	$(A, B, CCR, D, X, Y, \text{ or } SP) = > (A, B, CCR, D, X, Y, \text{ or } SP)$
	XGDY	交换 D 和 X 的值	$(D) < = > (X)$
	XGDY	交换 D 和 Y 的值	$(D) < = > (Y)$

3. MOVE 指令

MOVE 指令将存储器中的数据(字或字节)从源地址(M1 或  $M : M + 1_1$ )传送(复制)到目的地址(M2 或  $M : M + 1_2$ )。表 2-9 显示字节和字 MOVE 指令。

表 2-9 MOVE 指令

指 令	功 能	操 作
MOVB	移动字节(8 位)	$(M1) = > M2$
MOVW	移动字(16 位)	$(M1 : M1 + 1) = > M2 : M2 + 1$



## 2.6.2 算术运算类指令

### 1. 加法和减法指令

有符号和无符号的 8 位和 16 位加法和减法可以在寄存器之间或寄存器和内存之间进行。特殊指令支持指数计算。在条件码寄存器(CCR)中添加了进位的指令方便多倍精度计算。

加法和减法指令见表 2-10。

取地址指令(LEAS、LEAX 和 LEAY)指令也可以看成是特殊的加减指令。

表 2-10 加法和减法指令

指 令	功 能	操 作
加 法 指 令	ABA	A 的值和 B 的值相加并存入 A 中
	ABX	B 的值和 X 低 8 位值相加并存入 X 中
	ABY	B 的值和 Y 低 8 位值相加并存入 Y 中
	ADCA	带有进位将 A 和存储器地址中的值相加存到 A 中
	ADCB	带有进位将 B 和存储器地址中的值相加存到 B 中
	ADDA	不带进位将 A 和存储器地址中的值相加存到 A 中
	ADDB	不带进位将 B 和存储器地址中的值相加存到 B 中
	ADDD	D 和存储器地址中的值相加存到 D 中
	ADDX	X 和存储器地址中的值相加存到 X 中
	ADDY	Y 和存储器地址中的值相加存到 Y 中
	ADED	带有进位将 D 和存储器地址中的值相加存到 D 中
	ADEX	带有进位将 X 和存储器地址中的值相加存到 X 中
	ADEY	带有进位将 Y 和存储器地址中的值相加存到 Y 中
减 法 指 令	SBA	A 的值减去 B 的值存到 A 中
	SBCA	带借位从 A 中减去存储器地址中的值存到 A 中
	SBCB	带借位从 B 中减去存储器地址中的值存到 B 中
	SBED	带借位从 D 中减去存储器地址中的值存到 D 中
	SBEX	带借位从 X 中减去存储器地址中的值存到 X 中
	SBEY	带借位从 Y 中减去存储器地址中的值存到 Y 中
	SUBA	从 A 中减去存储器地址中的值存到 A 中
	SUBB	从 B 中减去存储器地址中的值存到 B 中
	SUBD	从 D 中减去存储器地址中的值存到 D 中
	SUBX	从 X 中减去存储器地址中的值存到 X 中
	SUBY	从 Y 中减去存储器地址中的值存到 Y 中

## 2. BCD 码指令

要添加 BCD 操作数,先使用在 CCR 中设置半进位的加法指令,然后使用十进制调整指令调整结果。表 2-11 是一个执行 BCD 码指令的概要。

表 2-11 BCD 码指令

指 令	功 能	操 作
ABA	A 与 B 相加并存到 A 中	$(A) + (B) = > A$
ADCA	带进位 A 与存储器地址中的值相加存到 A 中	$(A) + (M) + C = > A$
ADCB	带进位 B 与存储器地址中的值相加存到 B 中	$(B) + (M) + C = > B$
ADDA	A 与存储器地址中的值相加存到 A 中	$(A) + (M) = > A$
ADDB	B 与存储器地址中的值相加存到 B 中	$(B) + (M) = > B$
DAA	A 转化为十进制	$(A)10$

## 3. 自增自减指令

自增自减指令优化 8 位和 16 位的加法和减法运算。因为它们不影响 CCR 的进位,所以适合在多精度计算程序中作为循环计数器用。表 2-12 为自增自减指令摘要。

表 2-12 自增自减指令

指 令	功 能	操 作
DEC	存储器地址中的值减 1 存到存储器地址中	$(M) - \$01 = > M$
DECA	A 值减 1 存到 A 中	$(A) - \$01 = > A$
DECB	B 值减 1 存到 B 中	$(B) - \$01 = > B$
DECW	存储器地址中的值减 1 存到存储器地址中	$(M : M+1) - \$0001 = > M : M+1$
DECX	X 值减 1 存到 X 中(CCR 寄存器中的 N、Z、V 位受影响)	$(X) - \$0001 = > X$
DECY	Y 值减 1 存到 Y 中(CCR 寄存器中的 N、Z、V 位受影响)	$(Y) - \$0001 = > Y$
DES	SP 值减 1 存到 SP 中	$(SP) - \$0001 = > SP$
DEX	X 值减 1 存到 X 中(CCR 寄存器中的 Z 位受影响)	$(X) - \$0001 = > X$
DEY	Y 值减 1 存到 Y 中(CCR 寄存器中的 Z 位受影响)	$(Y) - \$0001 = > Y$





续表 2-12

指 令		功 能	操 作
自 增 指 令	INC	存储器地址中的值自增 1	$(M) + \$01 \Rightarrow M$
	INCA	A 值自增 1	$(A) + \$01 \Rightarrow A$
	INCB	B 值自增 1	$(B) + \$01 \Rightarrow B$
	INCW	存储器地址中的值自增 1	$(M : M+1) + \$0001 \Rightarrow M : M+1$
	INCX	X 值自增 1(CCR 寄存器中的 N、Z、V 位受影响)	$(X) + \$0001 \Rightarrow X$
	INCY	Y 值自增 1(CCR 寄存器中的 N、Z、V 位受影响)	$(Y) + \$0001 \Rightarrow Y$
	INS	SP 值自增 1	$(SP) + \$0001 \Rightarrow SP$
	INX	X 值自增 1(CCR 寄存器中的 Z 位受影响)	$(X) + \$0001 \Rightarrow X$
	INY	Y 值自增 1(CCR 寄存器中的 Z 位受影响)	$(Y) + \$0001 \Rightarrow Y$

#### 4. 乘法和除法指令

乘法指令用于有符号和无符号 8 位和 16 位的乘法运算。扩展除法指令使用 32 位的被除数和 16 位除数产生 16 位商和 16 位余数。表 2-13 是乘法和除法指令摘要。

表 2-13 乘法和除法指令

指 令		功 能	操 作
乘 法 指 令	EMUL	两个 16 位无符号数相乘(两个操作数分别放在寄存器 D 和 Y 中,结果高 8 位在 Y 中,低 8 位在 D 中)	$(D) \times (Y) \Rightarrow Y : D$
	EMULS	两个 16 位有符号数相乘(两个操作数分别放在寄存器 D 和 Y 中,结果高 8 位在 Y 中,低 8 位在 D 中)	$(D) \times (Y) \Rightarrow Y : D$
	MUL	两个 8 位无符号数相乘(两个操作数分别放在寄存器 A 和 B 中,结果高 8 位在 A 中,低 8 位在 B 中)	$(A) \times (B) \Rightarrow A : B$
除 法 指 令	EDIV	32 位无符号数除以 16 位无符号数(两个操作数分别放在寄存器 Y : D 和 X 中,结果商在 Y 中,余数在 D 中)	$(Y : D) \div (X) \Rightarrow Y \text{ Remainder} \Rightarrow D$
	EDIVS	32 位有符号数除以 16 位无符号数(两个操作数分别放在寄存器 Y : D 和 X 中,结果商在 Y 中,余数在 D 中)	$(Y : D) \div (X) \Rightarrow Y \text{ Remainder} \Rightarrow D$
	FDIV	D 的值除以 X 的值(相当于 D 中的值左移 16 位,除以 X,商在 X 中,余数在 D 中,再得出商所对应的小数)	$(D) \div (X) \Rightarrow X \text{ Remainder} \Rightarrow D$
	IDIV	16 位无符号整数除以 16 位无符号整数(商在 X 中,余数在 D 中)	$(D) \div (X) \Rightarrow X \text{ Remainder} \Rightarrow D$
	IDIVS	16 位有符号整数除以 16 位有符号整数(商在 X 中,余数在 D 中)	$(D) \div (X) \Rightarrow X \text{ Remainder} \Rightarrow D$

## 5. 清零、求补、取反指令

清零、求补、取反指令在累加器或者存储器中执行二进制值的具体操作。清零指令将值清为 0,求补操作将值替换成相应的二进制补码,取反操作将值替换成它的二进制反码。表 2-14 是清除、求补、取反指令摘要。

表 2-14 清除、求补、取反指令

指 令	功 能	操 作
清 除 指 令	CLC	清 CCR 中 C 位为 0 $0 \Rightarrow C$
	CLI	清 CCR 中 I 位为 0 $0 \Rightarrow I$
	CLR	清存储器地址中的值为 0 $\$00 \Rightarrow M$
	CLRA	清 A 为 0 $\$00 \Rightarrow A$
	CLRB	清 B 为 0 $\$00 \Rightarrow B$
	CLRW	清存储器地址中的值为 0 $\$0000 \Rightarrow M : M + 1$
	CLR X	清 X 为 0 $\$0000 \Rightarrow X$
	CLRY	清 Y 为 0 $\$0000 \Rightarrow Y$
	CLV	清 CCR 中 V 位为 0 $0 \Rightarrow V$
取 反 指 令	COM	存储器地址中的值求反 $\$FF - (M) \Rightarrow M$ or $(\overline{M}) \Rightarrow M$
	COMA	A 求反 $\$FF - (A) \Rightarrow A$ or $(\overline{A}) \Rightarrow A$
	COMB	B 求反 $\$FF - (B) \Rightarrow B$ or $(\overline{B}) \Rightarrow B$
	COMW	存储器地址中的值求反 $\$FFFF - (M : M + 1) \Rightarrow M : M + 1$ or $(\overline{M} : \overline{M+1}) \Rightarrow M : M + 1$
	COMX	X 求反 $\$FFFF - (X) \Rightarrow X$ or $(\overline{X}) \Rightarrow X$
	COMY	Y 求反 $\$FFFF - (Y) \Rightarrow Y$ or $(\overline{Y}) \Rightarrow Y$
求 补 指 令	NEG	存储器地址中的值求补 $\$00 - (M) \Rightarrow M$ or $(\overline{M}) + 1 \Rightarrow M$
	NEGA	A 求补 $\$00 - (A) \Rightarrow A$ or $(\overline{A}) + 1 \Rightarrow A$
	NEGB	B 求补 $\$00 - (B) \Rightarrow B$ or $(\overline{B}) + 1 \Rightarrow B$
	NEGW	存储器地址中的值求补 $\$0000 - (M : M + 1) \Rightarrow M : M + 1$ or $(\overline{M} : \overline{M+1}) + 1 \Rightarrow M : M + 1$
	NEGX	X 求补 $\$0000 - (X) \Rightarrow X$ or $(\overline{X}) + 1 \Rightarrow X$
	NEGY	Y 求补 $\$0000 - (Y) \Rightarrow Y$ or $(\overline{Y}) + 1 \Rightarrow Y$



## 6. 比较和测试指令

比较和测试指令在一对寄存器之间或寄存器和存储器之间执行减法,所得结果不被存储,而是根据结果设置相应状态位。这些指令通常用来建立分支指令的条件。在这个架构中,大多数指令自动更新条件码位,因此它不需要包含单独的测试或比较指令。表 2-15 是比较和测试指令摘要。

表 2-15 比较和测试指令

指 令		功 能	操 作
比 较 指 令	CBA	比较 A 和 B	$(A) - (B)$
	CMPA	比较 A 和存储器地址中的值	$(A) - (M)$
	CMPB	比较 B 和存储器地址中的值	$(B) - (M)$
	CPD	比较 D 和存储器地址中的值	$(A : B) - (M : M + 1)$
	CPED	有借位的比较 D 和存储器地址中的值	$(A : B) - (M : M + 1) - C$
	CPES	有借位的比较 SP 和存储器地址中的值	$(SP) - (M : M + 1) - C$
	CPEX	有借位的比较 X 和存储器地址中的值	$(X) - (M : M + 1) - C$
	CPEY	有借位的比较 Y 和存储器地址中的值	$(Y) - (M : M + 1) - C$
	CPS	比较 SP 和存储器地址中的值	$(SP) - (M : M + 1)$
	CPX	比较 X 和存储器地址中的值	$(X) - (M : M + 1)$
	CPY	比较 Y 和存储器地址中的值	$(Y) - (M : M + 1)$
测 试 指 令	TST	测试存储器地址中的值是否为 0 或负	$(M) - \$00$
	TSTA	测试 A 为 0 或负	$(A) - \$00$
	TSTB	测试 B 为 0 或负	$(B) - \$00$
	TSTW	测试存储器地址中的值为 0 或负	$(M : M + 1) - \$0000$
	TSTX	测试 X 为 0 或负	$(X) - \$0000$
	TSTY	测试 Y 为 0 或负	$(Y) - \$0000$

### 2.6.3 逻辑运算类与位操作类指令

#### 1. 移位和循环指令

移位和循环指令(见表 2-16)可以用于所有的累加器和存储器字节。通过 C 状态位方便多字节操作。由于逻辑和算术左移是完全相同的,所以没有单独的逻辑左移操作。

表 2-16 移位和循环指令

指 令		功 能	操 作
逻辑移位指令	LSL LSLA LSLB	存储器地址中的值逻辑左移 A 逻辑左移 B 逻辑左移	
	LSLD LSLW LSLX LSLY	D 逻辑左移 存储器地址中的值逻辑左移 X 逻辑左移 Y 逻辑左移	
	LSR LSRA LSRB	存储器地址中的值逻辑右移 A 逻辑右移 B 逻辑右移	
	LSRD LSRW LSRX LSRY	D 逻辑右移 存储器地址中的值逻辑右移 X 逻辑右移 Y 逻辑右移	
	ASL ASLA ASLB	存储器地址中的值算术左移 A 算术左移 B 算术左移	
	ASLD ASLW ASLX ASLY	D 算术左移 存储器地址中的值算术左移 X 算术左移 Y 算术左移	
	ASR ASRA ASRB	存储器地址中的值算术右移 A 算术右移 B 算术右移	
	ASRW ASRX ASRY	存储器地址中的值算术右移 X 算术右移 Y 算术右移	
	ROL ROLA ROLB	存储器地址中的值进位循环左移 A 进位循环左移 B 进位循环左移	
	ROLW ROLX ROLY	存储器地址中的值进位循环左移 X 进位循环左移 Y 进位循环左移	
	ROR RORA RORB	存储器地址中的值进位循环右移 A 进位循环右移 B 进位循环右移	
	RORW RORX RORY	存储器地址中的值进位循环右移 X 进位循环右移 Y 进位循环右移	



## 2. 布尔逻辑指令

布尔逻辑指令执行 8 位累加器或条件代码寄存器和存储器之间的值的逻辑运算,支持与、或和异或操作。表 2-17 总结了逻辑指令。

表 2-17 布尔逻辑指令

指 令	功 能	操 作
AND A	A 和存储器地址中的值与存入 A 中	$(A) \cdot (M) \Rightarrow A$
AND B	B 和存储器地址中的值求与存入 B 中	$(B) \cdot (M) \Rightarrow B$
AND CCR	CCR 和存储器地址中的值与(清除 CCR 位)存入 CCR 中	$(CCR) \cdot (M) \Rightarrow CCR$
AND X	X 和存储器地址中的值与存入 X 中	$(X) \cdot (M : M + 1) \Rightarrow X$
AND Y	Y 和存储器地址中的值与存入 Y 中	$(Y) \cdot (M : M + 1) \Rightarrow Y$
EOR A	A 和存储器地址中的值异或存入 A 中	$(A) \oplus (M) \Rightarrow A$
EOR B	B 和存储器地址中的值异或存入 B 中	$(B) \oplus (M) \Rightarrow B$
EOR X	X 和存储器地址中的值异或存入 X 中	$(X) \oplus (M : M + 1) \Rightarrow X$
EOR Y	Y 和存储器地址中的值异或存入 Y 中	$(Y) \oplus (M : M + 1) \Rightarrow Y$
OR A	A 和存储器地址中的值或存入 A 中	$(A) + (M) \Rightarrow A$
OR B	B 和存储器地址中的值或存入 B 中	$(B) + (M) \Rightarrow B$
OR CCR	CCR 和存储器地址中的值求或(设置 CCR 位)存入 CCR 中	$(CCR) + (M) \Rightarrow CCR$
OR X	X 同存储器地址中的值或存入 X 中	$(X) + (M : M + 1) \Rightarrow X$
OR Y	Y 同存储器地址中的值或存入 Y 中	$(Y) + (M : M + 1) \Rightarrow Y$

## 3. 位检测和操作指令

位检测和处理操作使用掩码来测试或改变累加器或存储器中的位值。位检测 A(BITA)和位检测 B(BITB)提供简便的测试位的方法而不改变其中任何一个操作数的值。表 2-18 是位检测和操作指令总结。

表 2-18 位检测和操作指令

指 令	功 能	操 作
BCLR	寄存器地址中的值的掩码对应位清 0	$(M) \cdot (\overline{mm}) \Rightarrow M$
BITA	A 和寄存器地址中的值相应位与	$(A) \cdot (M)$
BITB	B 和寄存器地址中的值相应位与	$(B) \cdot (M)$
BITX	X 和寄存器地址中的值相应位与	$(X) \cdot (M : M + 1)$
BITY	Y 和寄存器地址中的值相应位与	$(Y) \cdot (M : M + 1)$
BSET	寄存器地址中的值的掩码对应位置 1	$(M) + (mm) \Rightarrow M$

## 2.6.4 程序控制类指令

当特定的条件存在时,转移指令可以引起序列的改变。CPU12 使用 3 种转移指令,分别是短转移指令、长转移指令和按位条件转移指令。

转移指令也可以按照转移操作必须满足的条件类型来进行分类。例如,一元转移指令总是执行;当条件码寄存器的特定位在作为一个以前的操作结果特定的状态时,采取简单的转移指令;当采取比较或在特定条件寄存器位下组合的无符号结果时,采取无符号的转移指令;当采取比较或在特定条件寄存器位下组合的有符号结果时,采取有符号的转移指令。

### 1. 转移指令

转移指令分为短转移指令、长转移指令、按位条件转移指令。

#### (1) 短转移指令

短转移指令的操作是这样的:当指定条件得到满足时,有符号的 8 位偏移量添加到程序计数器的值中。程序转到新的地址处继续执行。

短转移偏移值的数值范围是程序计数器 PC 中值的偏移 \$ 80(−128)~\$ 7F(127)处。

表 2-19 短转移指令

指 令		功 能	等式或操作
一元转移	BRA	无条件转移	$(PC) + \$ 0002 + Rel = > PC$ Rel:跳转指令中的相对偏移量
	BRN	从不转移(相当于需要一个周期执行的 2 字节 NOP 指令)	$(PC) + \$ 0002 = > PC$
简单转移	BCC	C=0 时转移	C = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BCS	C=1 时转移	C = 1 时, $(PC) + \$ 0002 + Rel = > PC$
	BEQ	Z = 1 时转移	Z = 1 时, $(PC) + \$ 0002 + Rel = > PC$
	BMI	负转移	N = 1 时, $(PC) + \$ 0002 + Rel = > PC$
	BNE	Z = 0 时转移	Z = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BPL	正转移	N = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BVC	V=0 时转移	V = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BVS	V=1 时转移	V = 1 时, $(PC) + \$ 0002 + Rel = > PC$
无符号转移	BHI	大于时转移	C + Z = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BHS	大于或等于时转移	C = 0 时, $(PC) + \$ 0002 + Rel = > PC$
	BLO	小于时转移	C = 1 时, $(PC) + \$ 0002 + Rel = > PC$
	BLS	小于或等于时转移	C + Z = 1 时, $(PC) + \$ 0002 + Rel = > PC$



续表 2-19

指 令		功 能	操 作
有 符 号 转 移	BGE	大于或等于时转移	$N \oplus V = 0$ 时, $(PC) + \$0002 + Rel = > PC$
	BGT	大于时转移	$Z + (N \oplus V) = 0$ 时, $(PC) + \$0002 + Rel = > PC$
	BLE	小于或等于时转移	$Z + (N \oplus V) = 1$ 时, $(PC) + \$0002 + Rel = > PC$
	BLT	小于时转移	$N \oplus V = 1$ 时, $(PC) + \$0002 + Rel = > PC$

## (2) 长转移指令

长转移指令(见表 2-20)的操作是这样的:当指定条件得到满足时,一个有符号的 16 位偏移量加到当前程序计数器值中。程序从新的地址处继续执行。当需要大位移时,使用长转移指令。

长转移偏移值的数字范围是从当前程序计数器值的  $\$8000(-32,768) \sim \$7FFF(32,767)$ 。这允许转移从当前 64 KB 地址的任何位置映射到 64 KB 其他任何位置。

表 2-20 长转移指令

指 令		功 能	等式或操作
一元 转移	LBRA	无条件转移	$(PC) + \$0004 + Rel = > PC$
	LBRN	不转移	$(PC) + \$0004 = > PC$
简 单 转 移	LBCC	$C=0$ 时长转移	$C = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBCS	$C=1$ 时长转移	$C = 1$ 时, $(PC) + \$0004 + Rel = > PC$
	LBEQ	$Z=1$ 时长转移	$Z = 1$ 时, $(PC) + \$0004 + Rel = > PC$
	LBMI	负转移	$N = 1$ 时, $(PC) + \$0004 + Rel = > PC$
	LBNE	$Z=0$ 时长转移	$Z = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBPL	正转移	$N = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBVC	$V=0$ 时长转移	$V = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBVS	$V=1$ 时长转移	$V = 1$ 时, $(PC) + \$0004 + Rel = > PC$
无符 号 转移	LBHI	大于时长转移	$C + Z = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBHS	大于等于时长转移	$C = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBLO	小于时长转移	$C = 1$ 时, $(PC) + \$0004 + Rel = > PC$
	LBLS	小于等于时长转移	$C + Z = 1$ 时, $(PC) + \$0004 + Rel = > PC$
有符 号 转移	LBGE	大于等于时长转移	$N \oplus V = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBGT	大于时长转移	$Z + (N \oplus V) = 0$ 时, $(PC) + \$0004 + Rel = > PC$
	LBLE	小于等于时长转移	$Z + (N \oplus V) = 1$ 时, $(PC) + \$0004 + Rel = > PC$
	LBLT	小于时长转移	$N \oplus V = 1$ 时, $(PC) + \$0004 + Rel = > PC$



(3) 按位条件转移指令

按位条件转移是当存储器字节中的位在一个特定的状态时采用的。伪操作数是用来测试位置的。如果在该位置对应的掩码中的所有位都被置位(BRSET)或清 0(BRCLR),则转移。8 位偏移量的数值范围是从当前程序计数器中的值的 \$ 80(−128)~\$ 7F(127)处。表 2-21 是按位条件转移指令总结。

表 2-21 按位条件转移指令

指 令	功 能	等式或操作
BRCLR	选择的位为 0 时转移	如果(M) · (mm) = 0,则转移
BRSET	选择的位为 1 时转移	如果( $\overline{M}$ ) · (mm) = 0,则转移

2. 循环控制指令

循环控制也可以看作计数器转移。指令在寄存器或累加器(A、B、D、X、Y 或 SP)中测试计数器值为零或非零作为一个转移条件。这些指令有前置递增、前置递减和只测试版本。

8 位偏移量的数值范围是从下当前程序计数器中的值的 \$ 80(−128)~\$ 7F(127)处。表 2-22 是循环控制指令总结。

表 2-22 循环控制指令

指 令	功 能	等式或操作
DBEQ	(count) − 1 = 0, 则转移 (count = A, B, D, X, Y, or SP)	(Counter) − 1 ⇒ Counter 如果(Counter) = 0, (PC) + \$ 0003 + Rel ⇒ PC
DBNE	(count) − 1! = 0, 则转移 (count = A, B, D, X, Y, or SP)	(Counter) − 1 ⇒ Counter 如果(Counter) ! = 0, (PC) + \$ 0003 + Rel ⇒ PC
IBEQ	(count) + 1 = 0, 则转移 (count = A, B, D, X, Y, or SP)	(Counter) + 1 ⇒ Counter 如果(Counter) = 0, (PC) + \$ 0003 + Rel ⇒ PC
IBNE	(count) + 1! = 0, 则转移 (count = A, B, D, X, Y, or SP)	(Counter) + 1 ⇒ Counter 如果(Counter) ! = 0, (PC) + \$ 0003 + Rel ⇒ PC
TBEQ	(count) = 0, 则转移 (count = A, B, D, X, Y, or SP)	如果(Counter) = 0, (PC) + \$ 0003 + Rel ⇒ PC

3. 跳转指令和子程序调用指令

跳转(JMP)指令直接导致序列的变化。JMP 指令加载一个 64 KB 的寄存器映射地址到 PC,并在该地址处继续执行程序。该地址可以提供一个完整的 16 位地址或通过各种形式的变址寻址地址。



子程序调用指令优化将控制转移到一个执行特定任务的代码段。短转移(BSR)跳转到子程序(JSR)或扩充内存调用(CALL),可用于初始化子程序。返回地址是堆栈,然后在子程序的地址开始处执行。正常的 64 字节地址空间的子程序可以由 RTS 的指令而终止。RTS 不将返回地址放入栈中,以便指令在 BSR 或 JSR 后执行恢复。

扩充内存调用子程序(CALL)指令的目的是为扩充内存的使用。在 PPAGE 寄存器中 CALL,将值放入堆栈并返回地址,然后向 PPAGE 写入一个新值来选择子程序所在的存储器页面。该页面的值在所有地址模块中除了间接变址寻址方式之外都是直接的操作数。在这些模式中,操作数指向存储器中新页面的值和子程序存储的地址。RTC 指令是用来终止扩充内存的子程序。RTC 不把 PPAGE 值放入栈中,返回地址以便在 CALL 下一条指令后执行恢复。因为软件的兼容性,CALL 和 RTC 可以在不具有扩展寻址能力的设备中正确执行。表 2-23 总结了跳转和子程序调用指令。

表 2-23 跳转和子程序调用指令

指 令	功 能	操 作
BSR	转到子程序	$SP - 2 \Rightarrow SP, RTNH : RTNL \Rightarrow M(SP) : M(SP+1)$ 子程序地址 $\Rightarrow PC$
CALL	调用子程序	$SP - 2 \Rightarrow SP, RTNH; RTNL \Rightarrow M(SP) : M(SP+1)$ $SP - 1 \Rightarrow SP, (PPAGE) \Rightarrow M(SP), Page \Rightarrow PPAGE$ 子程序地址 $\Rightarrow PC$
JMP	跳转	有效地址 $\Rightarrow PC$
JSR	跳转到子程序	$SP - 2 \Rightarrow SP, RTNH : RTNL \Rightarrow M(SP) : M(SP+1)$ Subroutine address $\Rightarrow PC$
RTC	访问返回	$M(SP) \Rightarrow PPAGE, SP + 1 \Rightarrow SP, M(SP) : M(SP+1) \Rightarrow PCH :$ $PCLSP + 2 \Rightarrow SP$
RTS	子程序返回	$M(SP) : M(SP+1) \Rightarrow PCH : PCLSP + 2 \Rightarrow SP$

## 4. 中断指令

中断指令处理控制执行一个任务的程序的转移。软件中断是一种异常。软件中断(SWI)指令启动同步异常处理。首先,将返回的 PC 值压入堆栈。经过 CPU 堆栈,通过 SWI 向量继续执行所指向的地址。SWI 指令的执行会导致没有中断服务请求中断。

RTI 指令用来终止所有的异常处理程序,包括中断服务例程。RTI 首先恢复 CCR、B、A、X、Y,并从堆栈返回地址。如果没有其他中断被挂起,恢复其正常执行之后的最后一条指令之前执行中断指令。表 2-24 是中断指令摘要。

表 2-24 中断指令

指 令	功 能	操 作
RTI	中断返回	$(M(SP) : M(SP+1)) = > CCRH : CCR; (SP) - \$0000 = > SP$ $(M(SP) : M(SP+1)) = > B : A; (SP) - \$0002 = > SP$ $(M(SP) : M(SP+1)) = > XH : XL; (SP) - \$0004 = > SP$ $(M(SP) : M(SP+1)) = > PCH : PCL; (SP) - \$0006 = > SP$ $(M(SP) : M(SP+1)) = > YH : YL; (SP) - \$0008 = > SP$
SWI	软件中断	$SP - 2 = > SP; RTNH : RTNL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; YH : YL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; XH : XL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; B : A = > M(SP) : M(SP+1)$ $SP - 2 = > SP; CCRH : CCRL = > (M(SP) : M(SP+1))$ $1 = > I; 0 = > U$ (软件中断向量) $= > PC$
SYS(仅限 CPU12XV1、 CPU12XV2)	系统调用中断	$SP - 2 = > SP; RTNH : RTNL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; YH : YL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; XH : XL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; B : A = > M(SP) : M(SP+1)$ $SP - 2 = > SP; CCRH : CCR = > M(SP) : M(SP+1)$ $1 \Rightarrow I; 0 \Rightarrow U$ (系统调用中断向量) $\Rightarrow PC$
TRAP	未实现的操作码中断	$SP - 2 = > SP; RTNH : RTNL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; YH : YL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; XH : XL = > M(SP) : M(SP+1)$ $SP - 2 = > SP; B : A = > M(SP) : M(SP+1)$ $SP - 2 = > SP; CCRH : CCRL = > (M(SP) : M(SP+1))$ $1 = > I; 0 = > U$ (Trap 向量) $= > PC$

## 5. 变址操作指令

变址操作指令在 3 个变址寄存器和累加器、其他寄存器或存储器中执行 8 位和 16 位操作,如表 2-25 所列。



表 2-25 变址操作指令

指 令		功 能	操 作
加法 指令	ABX	B 加 X 存入 X 中	$(B) + (X) \Rightarrow X$
	ABY	B 加 Y 存入 Y 中	$(B) + (Y) \Rightarrow Y$
比较 指令	CPES	带借位比较 SP 和存储器地址中的值	$(SP) - (M : M + 1) - C$
	CPEX	带借位比较 X 和存储器地址中的值	$(X) - (M : M + 1) - C$
	CPEY	带借位比较 Y 和存储器地址中的值	$(Y) - (M : M + 1) - C$
	CPS	比较 SP 和存储器地址中的值	$(SP) - (M : M + 1)$
	CPX	比较 X 和存储器地址中的值	$(X) - (M : M + 1)$
	CPY	比较 Y 和存储器地址中的值	$(Y) - (M : M + 1)$
取数 指令	GLDS1	将存储器地址中的值存入 SP 中	$M : M + 1 \Rightarrow SP$
	GLDX1	将存储器地址中的值存入 X 中	$(M : M + 1) \Rightarrow X$
	GLDY1	将存储器地址中的值存入给 Y 中	$(M : M + 1) \Rightarrow Y$
	LDS	将存储器地址中的值存入 SP 中	$(M : M + 1) \Rightarrow SP$
	LDX	将存储器地址中的值存入 X 中	$(M : M + 1) \Rightarrow X$
	LDY	将存储器地址中的值存入 Y 中	$(M : M + 1) \Rightarrow Y$
	LEAS	将有效地址存入 SP 中	Effective address $\Rightarrow SP$
	LEAX	将有效地址存入 X 中	Effective address $\Rightarrow X$
	LEAY	将有效地址存入 Y 中	Effective address $\Rightarrow Y$
存数 指令	GSTS	将 SP 存入存储器地址中	$(SP) \Rightarrow M : M + 1$
	GSTX	将 X 存入存储器地址中	$(X) \Rightarrow M : M + 1$
	GSTY	将 Y 存入存储器地址中	$(Y) \Rightarrow M : M + 1$
	STS	将 SP 存入存储器地址中	$(SP) \Rightarrow M : M + 1$
	STX	将 X 存入存储器地址中	$(X) \Rightarrow M : M + 1$
	STY	将 Y 存入存储器地址中	$(Y) \Rightarrow M : M + 1$
转移 指令	TFR	转移寄存器到寄存器	$(A, B, CCR, D, X, Y, \text{ or } SP) \Rightarrow$ $(A, B, CCR, D, X, Y, \text{ or } SP)$
	TSX	转移 SP 到 X	$(SP) \Rightarrow X$
	TSY	转移 SP 到 Y	$(SP) \Rightarrow Y$
	TXS	转移 X 到 SP	$(X) \Rightarrow SP$
	TYS	转移 Y 到 SP	$(Y) \Rightarrow SP$

续表 2-25

指 令		功 能	操 作
交换 指令	EXG	交换寄存器与寄存器	$(A, B, CCR, D, X, Y, \text{ or } SP)$ $\leqslant=> (A, B, CCR, D, X, Y, \text{ or } SP)$
	XGDX	交换 D 与 X	$(D) \leqslant=> (X)$
	XGDY	交换 D 与 Y	$(D) \leqslant=> (Y)$

6. 进栈与出栈指令

两种堆栈指令见表 2-26,使用数学的和数据传送指令的专门形式执行栈指针操作。堆栈操作指令用于从系统堆栈存储和恢复信息。

表 2-26 栈指令

指 令		功 能	操 作
进 栈 与 出 栈 指 令	PSHA	A 进栈	$(SP) - 1 => SP; (A) => M(SP)$
	PSHB	B 进栈	$(SP) - 1 => SP; (B) => M(SP)$
	PSHC	CCR 进栈	$(SP) - 1 => SP; (CCR) => M(SP)$
	PSHCW	CCRW 进栈	$(SP) - 2 => SP; (CCRH:CCR) => M(SP); M(SP+1)$
	PSHD	D 进栈	$(SP) - 2 => SP; (A : B) => M(SP); M(SP+1)$
	PSHX	X 进栈	$(SP) - 2 => SP; (X) => M(SP); M(SP+1)$
	PSHY	Y 进栈	$(SP) - 2 => SP; (Y) => M(SP); M(SP+1)$
	PULA	A 出栈	$(M(SP)) => A; (SP) + 1 => SP$
	PULB	B 出栈	$(M(SP)) => B; (SP) + 1 => SP$
	PULC	CCR 出栈	$(M(SP)) => CCR; (SP) + 1 => SP$
	PULCW	CCRW 出栈	$(M(SP); M(SP+1)) => CCRH:CCR; (SP) + 2 => SP$
	PULD	D 出栈	$(M(SP); M(SP+1)) => A : B; (SP) + 2 => SP$
	PULX	X 出栈	$(M(SP); M(SP+1)) => X; (SP) + 2 => SP$
	PULY	Y 出栈	$(M(SP); M(SP+1)) => Y; (SP) + 2 => SP$

7. 指针和变址计算指令

该指令集合是将 5 位、8 位、16 位常量,或者 8 位累加器 A 或 B、16 位累加器 D 中的数据加上 X、Y 变址寄存器或者堆栈 SP 中的内容,存入到 X、Y 或者 SP 中,见表 2-27。



表 2-27 指针和变址计算指令

指 令	功 能	操 作
LEAS	取变址的结果,将有效的地址计算结果放入 SP 中	$r \pm \text{constant} \Rightarrow \text{SP or}(r) + (\text{accumulator}) \Rightarrow \text{SP}$ $r = \text{X, Y, SP, or PC}$
LEAX	取变址的结果,将有效的地址计算结果放入 X 中	$r \pm \text{constant} \Rightarrow \text{X or}(r) + (\text{accumulator}) \Rightarrow \text{X}$ $r = \text{X, Y, SP, or PC}$
LEAY	取变址的结果,将有效的地址计算结果放入 Y 中	$r \pm \text{constant} \Rightarrow \text{Y or}(r) + (\text{accumulator}) \Rightarrow \text{Y}$ $r = \text{X, Y, SP, or PC}$

## 8. 条件码指令

条件码指令(见表 2-28)是特殊的数据访问指令,通常被用来改变条件码寄存器。

表 2-28 条件码指令

指 令	功 能	操 作
ANDCC	CCR 与存储器地址中的值逻辑与	$(\text{CCR}) \wedge (\text{M}) \Rightarrow \text{CCR}$
CLC	C 位清 0	$0 \Rightarrow \text{C}$
CLI	I 位清 0	$0 \Rightarrow \text{I}$
CLV	V 位清 0	$0 \Rightarrow \text{V}$
ORCC	CCR 与存储器地址中的值逻辑或	$(\text{CCR}) + (\text{M}) \Rightarrow \text{CCR}$
PSHC	CCR 进栈	$(\text{SP}) - 1 \Rightarrow \text{SP}; \text{CCR} \Rightarrow \text{M}(\text{SP})$
PSHCW	CCRW 进栈	$(\text{SP}) - 2 \Rightarrow \text{SP}; (\text{CCR} \text{H}; \text{CCR}) \Rightarrow \text{M}(\text{SP}); \text{M}(\text{SP}+1)$
PULC	CCR 出栈	$(\text{M}(\text{SP})) \Rightarrow \text{CCR}; (\text{SP}) + 1 \Rightarrow \text{SP}$
PULCW	CCRW 出栈	$(\text{M}(\text{SP}); \text{M}(\text{SP}+1)) \Rightarrow \text{CCR} \text{H}; \text{CC}; (\text{SP}) + 2 \Rightarrow \text{SP}$
SEC	C 位置 1	$1 \Rightarrow \text{C}$
SEI	S 位置 1	$1 \Rightarrow \text{I}$
SEV	V 位置 1	$1 \Rightarrow \text{V}$
TAP	A 值存入 CCR	$(\text{A}) \Rightarrow \text{CCR}$
TPA	CCR 值存入 A	$(\text{CCR}) \Rightarrow \text{A}$

## 9. 停止与等待模式指令

如表 2-29 所列,这两个指令可将 CPU12 设置为低功耗的非活动状态。

STOP 指令将返回地址、CPU 的寄存器和累加器内容压栈,然后将所有的时钟悬置。

WAI 指令将返回地址、CPU 的寄存器和累加器内容压栈,然后等待中断请求,但系统时钟还继续运行。要想恢复正常运行,STOP 和 WAIT 指令都需要一个中断或复位操作。

表 2-29 停止与等待模式指令

指 令	功 能	操 作
STOP	进入停止模式	$SP - 2 \Rightarrow SP; RTNH : RTNL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; YH : YL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; XH : XL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; CCRH : CCR \Rightarrow M(SP) : M(SP+1)$ 停止 CPU 时钟
WAI	进入等待模式	$SP - 2 \Rightarrow SP; RTNH : RTNL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; YH : YL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; XH : XL \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M(SP) : M(SP+1)$ $SP - 2 \Rightarrow SP; CCRH : CCR \Rightarrow M(SP) : M(SP+1)$

## 10. 后台模式与空操作指令

后台调试模式是 CPU12 的一种特殊操作模式,当 BDM 被使能,执行 BGND 可以进入 BDM,而空操作可以在软件调试中用来重置其他指令。例如,通过禁止分支来测试程序,而避免抵消数值,见表 2-30。

表 2-30 后台模式与空操作指令

指 令	功 能	操 作
BGND	进入背景调试模式	如果 BDM 可启用,输入 BDM;否则恢复正常的处理
BRN	不转移	不转移
LBRN	不转移	不转移
NOP	空操作	—

## 2.6.5 其他类指令

CPU12X 的指令系统中还有一些特殊指令,如模糊逻辑指令等,这里简要罗列。

- 最大值和最小值指令:MAX、MIN。
- 乘积累加指令:EMACS。
- 表插值指令:TBL、ETBL。
- 模糊逻辑相关指令(仅用于 CPU12V0、CPU12XV0):MEM、WAV、REV、REVW。





## 2.7 CPU12X 汇编语言基础

能够在 MCU 内直接执行的指令序列是机器语言,用助记符号来表示机器指令便于记忆,这就形成了汇编语言。因此,用汇编语言写成的程序不能直接放入 MCU 的程序存储器中执行,必须先转为机器语言。把汇编语言转变成机器语言的过程由编译器自动完成,本节介绍 S12X 汇编编译器所能识别的汇编语言源程序格式及伪操作指令。

### 2.7.1 S12X 汇编源程序格式

把汇编语言写成的源程序“翻译”成机器语言的工具叫汇编程序或编译器(Assembler),以下统一称作编译器。汇编语言源程序可以用通用的文本编辑软件书写编辑,以 ASCII 码形式存盘。具体的编译器对汇编语言源程序的格式有一定的要求,同时编译器除了识别 MCU 的指令系统外,为了能够正确地产生目标代码以及方便汇编语言的编写,编译器还提供了一些在汇编时使用的命令、操作符号;在编写汇编程序时,也必须正确使用它们。由于编译器提供的指令仅是为了更好地做好“翻译”工作,并不产生具体的机器指令,因此这些指令被称为伪指令(Pseudo Instruction)。如伪指令告诉编译器:从哪里开始编译,到何处结束,汇编后的程序如何放置等相关信息。当然,这些相关信息必须包含在汇编源程序中,否则编译器就难以编译好源程序,难以生成正确的目标代码。

汇编语言源程序以行为单位进行设计,每一行最多可以包含以下 4 个部分:

标号	操作码	操作数	注释
----	-----	-----	----

#### (1) 标号(Labels)

对于标号有下列要求及说明:

- ① 如果一个语句有标号,则标号必须从第一列开始书写。
- ② 可以组成标号的字符有:字母 A~Z、字母 a~z、数字 0~9、下划线“\_”、美元符号“\$”,但开头的第一个符号不能为数字和\$。
- ③ S12 编译器区分标号中字母的大小写,但指令和伪指令不区分大小。
- ④ 标号长度基本上不受限制,但实际使用时通常不要超过 20 个字符。若希望更多的编译器能够识别,建议标号(或变量名)的长度小于 8 个字符。
- ⑤ 标号后必须带冒号“:”或双冒号“::”,一个冒号表示局部符号,两个符号表示全局符号。模块外调用的标号需要用全局标号,模块内跳转的标号用局部标号。
- ⑥ 一个标号在一个程序中只能定义一次,否则是重复定义,不能通过编译。
- ⑦ 一行语句可以只有标号,编译器将把当前程序计数器的值赋给该标号。

## (2) 操作码(Opercodes)

操作码包括指令码以及后面即将介绍的 S12 编译器可以识别的伪指令码。对于有标号的行,必须用至少一个空格或制表符(TAB)将标号与操作码隔开。对于没有标号的行,不能从第一列开始写指令码,应以空格或制表符(TAB)开头。S12 编译器不区分操作码中字母的大小写。

## (3) 操作数(Operands)

操作数可以是地址、标号或指令码定义的常数,也可以是由伪运算符构成的表达式。若一条指令或伪指令有操作数,则操作数与操作码之间必须用空格隔开书写。操作数多于一个的,操作数之间用逗号“,”分隔。

### 1) 常数标识

S12 编译器识别的常数有十进制(默认,不需要前缀标识)、十六进制(用 \$ 或 0x 前缀标识)、二进制(用 % 前缀标识)。

### 2) “#”表示立即数

一个常数前添加“#”表示一个立即数,不加“#”时,表示一个地址。

特别说明:初学时常常会将立即数前面的“#”遗漏,如果该操作数只能是立即数时,编译器会提示错误。但有的指令操作数可以是立即数,也可以是地址单元,若把“MOV # 0x50, 0x40”误写为:“MOV 0x50, 0x40”,编译器当然不知道你有错,但这两句本身含义不同。前者表示将立即数 50(相当于十进制数 80)赋给 0x 40 单元,后者表示表示将 50 单元的内容赋给 0x 40 单元。

### 3) 伪运算符

S12X 编译器识别表 2-31 所列的伪运算符。

表 2-31 S12X 编译器识别的伪运算符

运算符	功 能	类 型	实 例
*	乘法	二元	MOVB # 5 * 4, \$ 40    等价于 MOVB # \$ 14, \$ 40
/	除法	二元	MOVB # 900/68, \$ 40    等价于 MOVB # \$ 0D, \$ 40
%	取模	二元	MOVB # 900%68, \$ 40    等价于 MOVB # \$ 10, \$ 40
<<	左移	二元	MOVB # 4<<2, \$ 40    等价于 MOVB # \$ 10, \$ 40
>>	右移	二元	MOVB # 4>>2, \$ 40    等价于 MOVB # \$ 01, \$ 40
^	按位异或	二元	MOVB # %10000010^%11111110, \$ 40 等价于 MOV # %01111100, \$ 40
&	按位与	二元	MOVB # %10000010&%11111110, \$ 40 等价于 MOV # %10000010, \$ 40
	按位或	二元	MOVB # %10000010  %11111110, \$ 40 等价于 MOV # %11111110, \$ 40
-	负号	一元	MOVB # -5, \$ 40    等价于 MOV # \$ FB, \$ 40



运算符	功 能	类 型	实 例
~	取反运算	一元	MOVB #~%10000001,\$40 等价于 MOV #%01111110,\$40
<	取低字节	一元	MOVB #<0x5678,\$40 等价于 MOVB #0x78,\$40
>	取高字节	一元	MOVB #>0x5678,\$40 等价于 MOVB #0x56,\$40

#### (4) 注释(Comments)

注释即是说明文字,用分号“;”。

### 2.7.2 S12X 汇编语言伪指令

S12X 汇编语言伪指令可以在 CodeWarrior 开发环境的帮助文件中找到。这里给出常用伪指令的简要说明。

#### (1) 变量定义(变量声明)

[<label>:] DS[.<size>] <n>

定义了一个存储区,预留  $\text{size} * n$  个字节,<label>为存储区的名称,也作为一个变量引用,<size>可以是 B、W、L 或者 F。通常 B 可以省略,B、W、L 定义整数,B 代表一个字节,W 代表一个字或两个字节,L 代表 4 个字节;F 定义浮点型数据,F 占用 4 个字节。

这里的 n 可以为数字,也可以是已经定义过的符号。一般用于在 RAM 区用标号定义数据变量或缓冲区,相当于高级语言的变量声明。例如:

```
LABEL1: DS.B   1      ;定义 1 个字节的存储区(即内存变量 LABEL1)
LABEL2: DS.W   5      ;定义 5 个字(10 字节)的存储区(即内存变量 LABEL2)
```

在汇编时,经常会使用内存块(缓冲区或数组),例如:

```
LABELBuf: DS.B   16    ;定义了以 LABELBuf 为起始地址的 16 字节数组
```

数组范围 LABELBuf~ LABELBuf+15,假如要对 LABELBuf 中的 12 号单元赋值,可以使用如下语句:

```
STAA    LABELBuf + 11    ;给 LABELBuf 的 12 单元赋值(注意从 0 开始编号)
[<label>:] RMB <n>
```

<label>、<n>作用同上,定义了一个存储区,预留  $1 * n$  个字节。

#### (2) 数字常数与字符串常数定义

[<label>:] DC[.<size>] <expression> [,<expression>]...

<label>、<size>作用同上,在程序区中(Flash 存储器区)定义一段常数区,可以是一个字节常数(Byte)、一个字常数(2Byte,双字节常数)或者若干字节等。字常数将被放入目标程

序的两个连续地址中,高字节在前,低字节在后。该伪指令<expression>可以有一个或多个用逗号“,”隔开的操作数,表达式<expression>的值可以为数字、标号、字符串。

例如:

```
C1: DC.B $ 36           ;C1 为一个字节的常数 C1 = 0x36
C2: DC.W $ 36           ;C2 为一个字的常数 C1 = 0x0036
C3: DC.B "ABCDE"        ;一个数组常数,C3 = "ABCDE" = 0x4142434445 (ASCII 码)
C4: DC.W "ABCDE"        ;一个数组常数,C4 = "ABCDE" = 0x004142434445 (ASCII 码)
C5: DC.L $ 36           ;C5 为 4 个字节的常数区,前面 3 个字节都为 $ 00,最后一个字节为 $ 36,即区域
                        中的定义的值为右对齐,数字不够,前面添 0
```

```
[<label>:] FCB <value>
```

```
[<label>:] FDB <expression>
```

```
[<label>:] FCC <expression>
```

<label>、<size>作用同上,在程序区中(Flash 存储器区)定义一段常数区,可以是一个字节常数(Byte)、一个字常数(2Byte,双字节常数),也可以是字符串常数。

```
STRING1: FCB $ 00        ;定义了一个字节(Byte)的值为 $ 00
STRING2: FDB STRING1     ;定义了两个字节(Double Byte),取标号 STRING1 的地址
STRING3: FCC "Hello World!";定义了字符串"Hello World!"
```

### (3) 常数赋值与文本替代符伪指令

```
<label>: EQU <expression>
```

定义符号<label>等于<expression>的值,该表达式必须是确定的值,不能含未知变量,<label>一旦定义,程序中不能更改。例如:PI: EQU 3

```
<label>: SET <expression>
```

该语法类似上一条伪指令,符号<label>的值被设置为<expression>的值,同样该表达式必须是确定的值,但是<label>的值是临时的,可以再更改。

```
例如:count: SET 3
      one:  DC.B count
      count: SET count - 1
            DC.B count
```

### (4) 存储定位伪指令

```
ORG<value>
```

用于定义程序或数据区的起始地址。该伪指令中的 value 为数字或标号,它告诉编译器,在指令汇编后,其后的指令和数据在存储器中将从地址 value 开始向地址增大方向存放。对于 CodeWarrior IDE 开发平台来说,创建汇编工程项目,选择绝对定位地址,每个区(数据区、代码区、中断向量区等)必须由 ORG 语句指定该区的首地址。一个完整的可在编译后放入单



片机执行的源程序,至少必须有一个 ORG 语句使程序能够放入 Flash 区。

## (5) 文件包含伪指令

INCLUDE <filename>

INCLUDE 是一个附加文件的链接指示命令,利用它可以把另一个文件插入当前的源文件一起汇编,成为一个完整的源程序。filename 是一个文件名,汇编时首先寻找当前文件,如果没有发现,则在 TBDML.ini 的[Environment Variables]下 GENPATH 所有指定文件夹查找。Filename 还可以包含文件的绝对路径或相当路径,但建议对于一个工程的源文件应放到同一个文件夹中,所以更多的时候应使用相对路径。

## 第一个样例程序及 CodeWarrior 工程组织

本章详细阐述第一个 C 语言程序和汇编语言程序的结构,完成第一个 S12X 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 S12X 工程组织、框架,阐述各个文件的功能,主要目的是让读者通过 GPIO 的编程例子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。本章主要知识点有:①通用 I/O 接口基本概念及引脚连接方法;②XS128 的 GPIO 寄存器与 GPIO 构件封装;③S12X 的 C 语言工程组织与框架,相关文件及第一个工程的执行过程;④工程编译、下载与调试;⑤第一个汇编工程。重点是掌握编程框架、透彻理解工程结构、相关文件及第一个工程的执行过程。

### 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(V<sub>cc</sub>)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的状态,获知该引脚是“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 3 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( $V_{CC}$ )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  $1\sim 10\text{ k}\Omega$  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 3 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: $R1$  的阻值为  $20\text{ k}\Omega$ , $R2$  的阻值为  $1\text{ k}\Omega$ , $R3$  的阻值为  $10\text{ k}\Omega$ , $R4$  的阻值为  $200\text{ k}\Omega$ 。

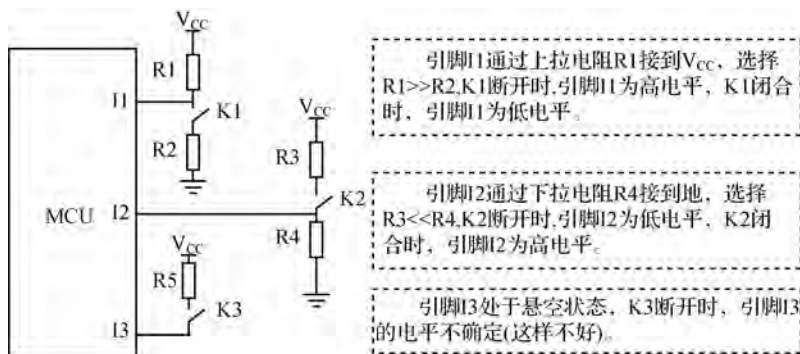


图 3-1 I/O 口输入电路

### (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。这种接法产生的驱动电流一般在  $2\sim 10\text{ mA}$ 。

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  $100\text{ mA}$  左右,而 O2 引脚控制电流可以在几个  $\text{mA}$  左右。

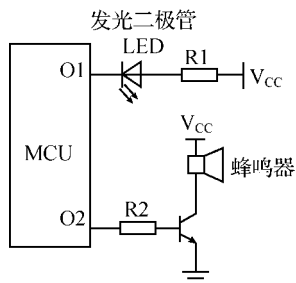


图 3-2 I/O 口输出电路

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



## 3.2 XS128 的 GPIO 寄存器与 GPIO 构件封装

### 3.2.1 XS128 的 GPIO 寄存器

80 引脚的 XS128 微控制器有 9 个通用 I/O 口,分别是 A 口、B 口、E 口、J 口、M 口、P 口、S 口、T 口、AD 口。这些引脚中的大部分具有多重功能(其中 A 口和 B 口只能作为普通 I/O 口使用),本节仅讨论它们作为普通 I/O 功能时的编程方法。该芯片有 59 个引脚具有通用 I/O 功能,其中 A、B、E、T、AD 都有 8 个引脚,而 J 口有 2 个引脚,M 口有 6 个引脚,S 口有 4 个引脚,P 口有 7 个引脚。

作为普通 I/O 口,它们的每一个引脚均可通过相应口的“数据方向寄存器”独立地设置为输入或输出(但 E 口引脚 1、引脚 0 仅能作 IRQ 和 XIRQ 输入)。对于被定义为输入的引脚,还可通过相应口的“上拉下拉电阻允许寄存器”独立地设置其有无内部上拉下拉电阻,但有的引脚只能设置上拉,有的端口各引脚只能一起设置上拉。被定义为输出的引脚,一律无上拉(或下拉)电阻,但可以根据功耗要求,设置成正常输出或低功耗输出。

作为通用 I/O,A、B、E 口除具有独立的数据方向寄存器与数据寄存器外,还共用上拉电阻控制寄存器(PUCR)与低功耗驱动寄存器(RDRIV)。而 T 口、S 口、M 口、P 口、J 口作为普通 I/O 口时,都有 6 个寄存器,它们是:数据方向寄存器、数据寄存器、输入寄存器、低功耗驱动寄存器、上拉下拉使能寄存器、上拉下拉选择寄存器。P 口、J 口还有两个寄存器,是中断使能寄存器和中断标志寄存器。S 口、M 口还有线或寄存器。

GPIO 基本功能与相关编程寄存器见表 3-1。下面做简要说明。

表 3-1 GPIO 寄存器描述

端口名	引脚数	GPIO 引脚名	寄存器名	缩写	地址	备注
A 口	8	PA[7 : 0]	数据方向寄存器	DDRA	\$ 0002	这 3 个口共用一个上拉控制寄存器 PUCR(\$ 000C),用于设置是否内部上拉。还共用一个低功耗驱动寄存器 RDRIV(\$ 000D),用于设置是否采用低功耗模式
			数据寄存器	PORTA	\$ 0000	
B 口	8	PB[7 : 0]	数据方向寄存器	DDRB	\$ 0003	
			数据寄存器	PORTB	\$ 0001	
E 口	8	PE[7 : 0]	数据方向寄存器	DDRE	\$ 0009	
			数据寄存器	PORTE	\$ 0008	



续表 3-1

端口名	引脚数	GPIO 引脚名	寄存器名	缩写	地址	备注
T 口	8	PT[7 : 0]	数据寄存器	PTT	\$ 0240	T 口具有接收输入捕捉功能, 每个引脚作为一个输入捕捉端口。在系统复位时, 引脚置位为高阻态。同时 PT[4 : 7]可复用为 PWM[4 : 7]
			输入寄存器	PTIT	\$ 0241	
			数据方向寄存器	DDRT	\$ 0242	
			低功耗驱动寄存器	RDRT	\$ 0243	
			上拉下拉使能寄存器	PERT	\$ 0244	
			上拉下拉选择寄存器	PPST	\$ 0245	
			保留		\$ 0246	
			引脚复用控制寄存器	PTTRR	\$ 0247	
S 口	4	PS[3 : 0]	数据寄存器	PTS	\$ 0248	S 口与 SCI 模块和 SPI 模块关联作为通信接口(即 PTS[0 : 1]、PTS[2 : 3]分别为 SCI0 和 SCI1 的通信端口, PTS[4 : 7]为 SPI0 的通信端)。在系统复位时, 引脚置位为带上拉电阻的输入状态
			输入寄存器	PTIS	\$ 0249	
			数据方向寄存器	DDRS	\$ 024A	
			低功耗驱动寄存器	RDRS	\$ 024B	
			上拉下拉使能寄存器	PERS	\$ 024C	
			上拉下拉选择寄存器	PPSS	\$ 024D	
			线或寄存器	WOMS	\$ 024E	
			保留		\$ 024F	
M 口	6	PM[5 : 0]	数据寄存器	PTM	\$ 0250	M 口 CAN 模块、SPI 模块和 SCI 模块关联作为通信接口, 即 CAN0 和 SCI1 共享 PTM[0 : 1], PTM[2 : 5]为 SPI0 的通信端。在系统复位时, 引脚置位为高阻态
			输入寄存器	PTIM	\$ 0251	
			数据方向寄存器	DDRM	\$ 0252	
			低功耗驱动寄存器	RDRM	\$ 0253	
			上拉下拉使能寄存器	PERM	\$ 0254	
			上拉下拉选择寄存器	PPSM	\$ 0255	
			线或寄存器	WOMM	\$ 0256	
			引脚复用控制寄存器	MODRR	\$ 0257	
P 口	7	PP7PP[5 : 0]	数据寄存器	PTP	\$ 0258	P 口与 PWM 模块和 SCI 模块关联作为通信接口, 即 PTP 每个引脚可作为一个 PWM 模块的输出引脚, PTP0 和 PTP2 可作为 SCI1 端口使用。在系统复位时, 引脚置位为高阻态输入
			输入寄存器	PTIP	\$ 0259	
			数据方向寄存器	DDRP	\$ 025A	
			低功耗驱动寄存器	RDRP	\$ 025B	
			上拉下拉使能寄存器	PERP	\$ 025C	
			上拉下拉选择寄存器	PPSP	\$ 025D	
			中断使能寄存器	PIEP	\$ 025E	
			中断标志寄存器	PIFP	\$ 025F	

续表 3-1

端口名	引脚数	GPIO 引脚名	寄存器名	缩写	地址	备注
J 口	2	PJ[7 : 6]	数据寄存器	PTJ	\$ 0268	J 口在作为输入输出端口时,提供中断功能;在系统复位时,引脚置位为带上拉电阻的输入状态
			输入寄存器	PTIJ	\$ 0269	
			数据方向寄存器	DDRJ	\$ 026A	
			低功耗驱动寄存器	RDRJ	\$ 026B	
			上拉下拉使能寄存器	PERJ	\$ 026C	
			上拉下拉选择寄存器	PPSJ	\$ 026D	
			中断使能寄存器	PIEJ	\$ 026E	
			中断标志寄存器	PIFJ	\$ 026F	
AD 口	8	PAD0[7 : 5]	数据方向寄存器	DDRAD0	\$ 0272	AD 口除了作为 AD 转换模块的模拟量输入口和外界触发脉冲的输入口,也可以作为普通输入输出
			数据寄存器	PT0AD0	\$ 0270	

1) 数据方向寄存器

数据方向寄存器的各位值决定了相对应引脚是输入还是输出。若数据方向寄存器某位为 0,则对应的引脚为输入;若为 1,则对应引脚为输出。数据方向寄存器复位值为 \$ 00(默认为输入)。

记忆要点:数据方向寄存器的一位:0—定义输入,1—定义输出

2) 数据寄存器与输入寄存器

当数据方向寄存器的某一位被置为 1 时(定义相应引脚为输出),这时可通过数据寄存器设置对应引脚为高电平(1)或低电平(0)。

记忆要点:

输出时:数据寄存器的一位:0—输出低电平,1—输出高电平

当数据方向寄存器的某一位被置为 0 时(定义相应引脚为输入),数据寄存器和输入寄存器功能一致,可通过读取数据寄存器获得对应引脚的状态,只是输入寄存器获取对应引脚的状态不受数据方向寄存器控制,输入寄存器可以随时获取对应引脚状态,用于检测输出引脚的超负荷或短路情况。

记忆要点:

输入时:数据寄存器和输入寄存器的一位:

0—代表外部输入低电平,1—代表外部输入高电平

3) 低功耗驱动寄存器

有些端口有低功耗驱动寄存器,以便实现低功耗输出。低功耗驱动寄存器的某位若为 1,



对应引脚的输出功耗为正常时的 1/5;若为 0,对应引脚输出正常功耗。

记忆要点:

输出时:低功耗驱动寄存器的一位:0—输出功耗正常,1—输出功耗为正常的 1/5。

#### 4) 上拉下拉电阻使能寄存器与上拉下拉选择寄存器

上拉下拉使能寄存器的某位如果为 0,表示禁用上拉或下拉电阻;如果为 1,则允许上拉或下拉电阻。

记忆要点:

上拉下拉使能寄存器的一位:0—无上拉或下拉,1—有上拉或下拉。

在 I/O 口的某位引脚被定义为输入且上拉下拉使能寄存器的对应位定义为允许上拉或下拉的情况下,上拉下拉选择寄存器的对应位的值若为 1,则该位设置为下拉;若为 0,则该位设置为上拉。

记忆要点:

输入和允许上拉下拉时,上拉下拉选择寄存器的一位:0—上拉 1—下拉。

#### 5) A、B、E 口共用的上拉电阻控制寄存器(PUCR)

A 口、B 口、E 口、BKGD 都有上拉电阻,它们共用一个控制寄存器(PUCR)。PUCR 寄存器的地址为 \$000C。PUCR 的第 0 位为 PUPAE(标识中“A”代表 A 口;B 口、E 口等亦如此),含义是 A 口的上拉电阻使能位。当 PUPAE=1 时,A 口的 8 个引脚中被定义为输入的引脚有内部上拉电阻。类似地,有设定 B 口、E 口上拉电阻的对应位:PUPBE、PUPEE,分别是 PUCR 的第 1、4 位。第 6 位定义为 BKGD 引脚上拉使能。PUCR 的其他位未定义。

记忆要点:

在引脚 PORTA 被定义成输入时,可通过上拉电阻允许寄存器 PUCR 中的 PUPAE 位来定义有无内部上拉电阻:0—没有内部上拉电阻 1—有内部上拉电阻。

#### 6) A、B、E 口共用的低功耗驱动寄存器(RDRIV)

A 口、B 口、E 口都有低功耗驱动功能,它们共用一个控制寄存器(RDRIV)。RDRIV 寄存器的地址为 \$000D。RDRIV 的第 0 位为 RDP A(标识中“A”代表 A 口;B 口、E 口等亦如此),是 A 口的低功耗使能位。当 RDP A=1 时,A 口的 8 个引脚中被定义为低功耗驱动。类似地,有设定 B 口、E 口低功耗驱动使能:RDPB、RDPE,分别是 RDRIV 的第 1、4 位。RDRIV 的其他位未定义。

记忆要点:

在引脚被定义为低功耗驱动时,可通过低功耗允许寄存器 RDRIV 中的相应位来定义是否低功耗驱动使能:0—不允许低功耗驱动使能 1—允许低功耗驱动使能。

#### 7) A/D 口也可以设置低功耗驱动功能

A/D 口也可以设置低功耗驱动功能,且各个引脚可以分别设置,控制寄存器是 RDR0AD0 和 RDR1AD0,地址为 \$0274、\$0275。

记忆要点：

在引脚被定义成低功耗驱动时,可通过低功耗驱动寄存器 RDR0AD0 和 RDR1AD0 中相应的位来定义相应引脚是否低功耗驱动: 0—不允许低功耗驱动使能 1—允许低功耗驱动使能。

### 8) 中断使能寄存器与中断标志寄存器

在 I/O 口的某位引脚被定义具有中断功能时,对应位定义为允许外界触发引起中断,中断使能寄存器的对应位若为 1,则设置中断相应使能;若为 0,则设置为禁止相应中断。

可由中断标志寄存器表征中断是否已经产生,若中断标志寄存器的某一位为 1,则表明相应中断已经产生;若为 0,则相应中断未产生。

P 口、J 口在作为输入端口时,提供了中断功能,可以和键盘相连;在系统复位时,引脚置位为带上拉电阻的输入状态。

### 9) 线或寄存器

在 I/O 口中 M 口、S 口的引脚具有输出的线或和推拉模式,该寄存器的对应位若为 1,则该位对应引脚为输出线或模式(即开漏);若为 0,则该位对应引脚为输出推拉模式。引脚设置为输入,该寄存器设置无效。

### 10) 有关口的复用功能简要说明

各端口除复位后默认作普通 I/O 口的功能外,还可以复用为其他功能。有关复用功能见表 3-1。T 口复用功能是输入捕捉与输出比较。S 口复用功能是 SCI 与 SPI 通信。M 口的复用功能是 CAN 通信、SPI 通信和 SCI 通信。P 口的复用功能是 SCI 通信与 PWM。

T 口、S 口、M 口、P 口引脚分别复用不同模块,其功能由 T 口复用功能寄存器(PTTTR)和 M 口复用功能寄存器(MODRR)设置。

### 11) 关于 AD 口的说明

AD 口除了作为普通输入输出,也可以用作数模转换模块的模拟量输入。AD 口有两个 8 引脚端口,分别关联数据寄存器 PT0AD0、PT1AD0 以及数据方向寄存器 DDR0AD0 和 DDR1AD0。针对 112 引脚的芯片,可以并在一起组成 16 位端口,即 PT01AD0[7:0]和 PT01AD0[15:8],以便进行字处理。

## 3.2.2 GPIO 的简单编程方法

### (1) 置位与清位的编程方法

下面是置位、清 0 等 3 个宏定义：

```
#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
```

其中,“<<”、“>>”、“|”、“&”、“~”等是位运算符,具体参考表 2-31。“1<<(bit)”中

符号前的 1 是被移动的数,符号“<<”右的 bit 是确定移动的位。3 个宏中表达式的功能说明如下:

置 1,如  $PA |= (1 << 3)$ ,其中“ $1 << 3$ ”的结果是“0b00001000”, $PA |= (1 << 3)$ 也就是  $PA = PA | 0b00001000$ ,任何数和 0 相或值不变,任何数和 1 相或值为 1,这样达到对 PA 寄存器的第 3 位置 1 的目的。

清 0,如  $PA \&= \sim(1 << 2)$ ,其中“ $\sim(1 << 2)$ ”的结果是“0b11111011”, $PA \&= \sim(1 << 2)$ 也就是  $PA = PA \& 0b11111011$ ,任何数和 1 相与值不变,任何数和 0 相与值为 0,这样达到对 PA 寄存器的第 2 位清 0 的目的。

得到某一位的状态,如  $(PA >> 4) \& 1$ ,是获得 PA 寄存器第 4 位的状态,“ $PA >> 4$ ”是将 PA 右移 4 位,将 PA 的第 4 位移至第 0 位,即最后 1 位,再和 1 相“与”,也就是和 0b00000001 相“与”,保留 PA 最后 1 位的值,以此得到第 4 位的状态值。

## (2) 开关量输出的编程方法

首先初始化端口引脚的数据方向为输出,然后运用该引脚的数据寄存器进行数据输出。

如:使 B 口的第 4 引脚输出高电平。

```
BSET(4, DDRB);    //B 口的第 4 引脚初始化为输出
BSET(4, PORTB);   //B 口的第 4 引脚输出高电平(1)
```

## (3) 开关量输入的编程方法

首先初始化端口的引脚数据方向为输入,然后运用该引脚将外界数据输入给对应数据寄存器中。

如:获取 B 口第 3 引脚的输入数据。

```
BCLR(3, DDRB);    //B 口的第 3 引脚初始化为输入
Data = BGET(3, PORTB); //获得 B 口第 3 引脚的输入数据赋给变量 Data
```

# 3.3 CodeWarrior 开发环境与 S08/S12/ColdFire 三合一写入器

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 XS128 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。下面将介绍 Freescale 公司的 CodeWarrior for S12 V5.0 集成开发环境(简称 CW 环境)、苏州大学的 XS128-80PIN-CoreBoard 硬件评估板以及 S08/S12/ColdFire 三合一写入器。S08/S12/ColdFire 三合一写入器的使用方法见附录 B。有关资料也存放在网上光盘中。在附录 C 中简要给出了硬件评估过程可能出现的常见问题。

### 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。在其环境下可编制并调试 XS128 MCU 的汇编语言、C 语言和 C++ 语言程序。其中特别版是免费的,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 12 KB,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 S08 V6.2、CodeWarrior for S12 V5.0、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 4 个主要阶段,其他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 XP 操作系统上,只要按照安装向导就可以完成。

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网上光盘,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 S12 V5.0\”。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。CW 环境的运行界面如图 3-3 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。方法如下:选择 Edit→Perferences 菜单项,则弹出 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,建议选中 Tab Inserts Spaces,使 Tab 键插入的是多个空格。

网上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



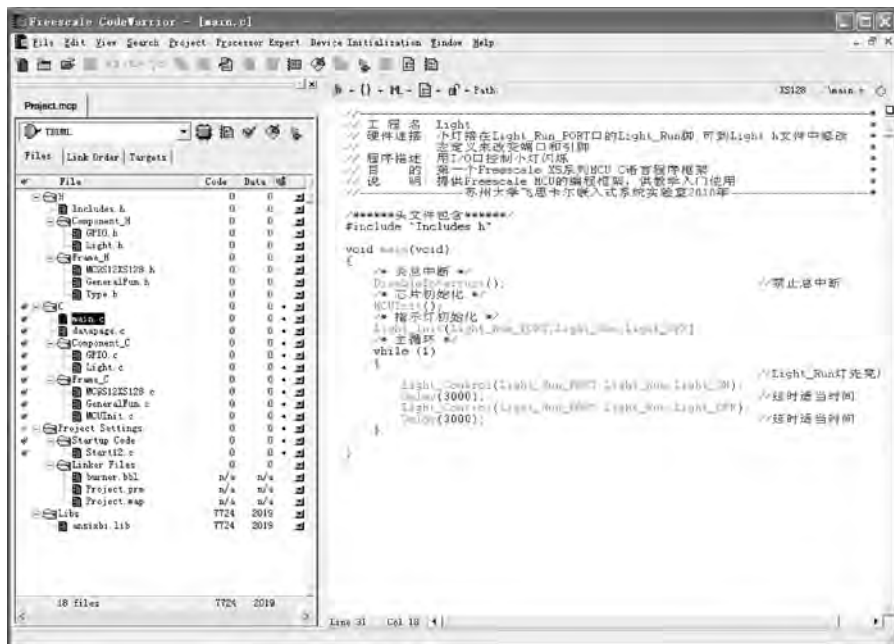


图 3-3 CW 环境运行界面

译、程序运行、调试等阐述。

### 3.3.2 S08/S12/ColdFire 三合一写入器

开发人员可以通过 S08/S12/ColdFire 三合一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 3-4 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 3-4 S08/S12/ColdFire 三合一写入器实物图

### 3.3.3 MC9S12XS128 硬件评估板

MC9S12XS128 硬件评估板如图 3-5 所示。该硬件评估板使用 80 引脚的 QFP 封装的 XS128 芯片,带 RS232 串口、一个复位按钮、一个 BDM 写入接口、引出所有 I/O 口、并带有与苏州大学飞思卡尔嵌入式系统研发中心 D 型扩展板的对接口。扩展板外接 12V 直流电源,转成 5V 供电。

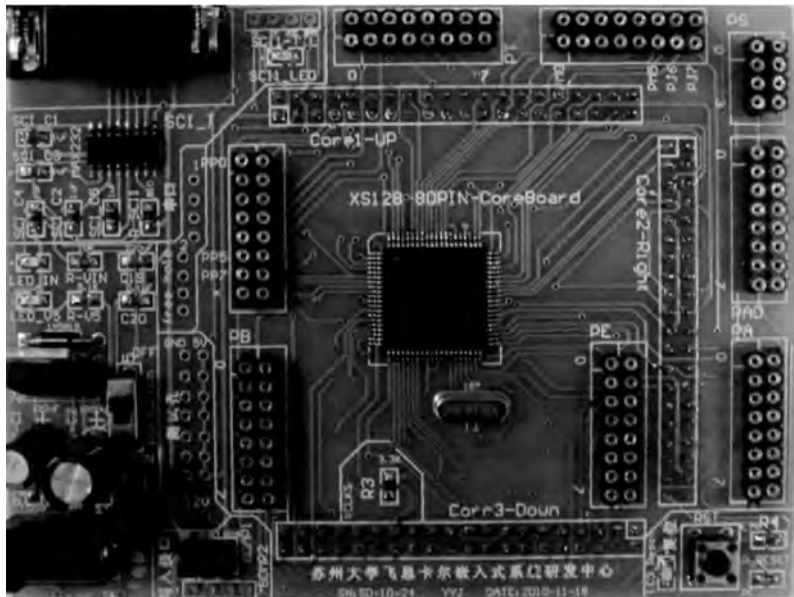


图 3-5 MC9S12XS128 硬件评估板

## 3.4 CW 环境 C 语言工程文件的组织

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尤为重要,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。

下面将从逻辑结构和物理结构两个层次为读者做一个简要的剖析。逻辑结构是相对于工程而言的,而物理结构即说明其在存储磁盘介质上的实际分布情况。这部分内容对于初学者而言有一定难度,但一定要有耐心,一时不完全理解也没关系,先建立一个初步印象,待学习过第 5、6 章后再回过头来体会就会发现良好且规范的工程组织对程序的可复用、可移植,减少重复编码,增强程序稳定性十分有益。

### 3.4.1 工程文件的逻辑组织结构

嵌入式系统工程的文件逻辑组织方法以硬件构件为核心展开,有关硬件构件的基本概念将在第4章阐述。

一个硬件构件的低层驱动软件由头文件和程序文件组成,在不引起二义性的前提下,也可直接称之为硬件构件。以硬件构件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。

下面以后面3.5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逻辑组织方法。图3-6给出了该工程相关源文件的树型结构,可分为:头文件夹<H>、C源



图 3-6 工程相关源文件的树型结构

程序文件夹<C>、库文件文件夹<Libs>、工程设置文件夹<Project Settings>及说明文件.txt共5个部分,其中,“头文件夹<H>”又包含“总体框架头文件夹<Frame\_H>”与“软件构件头文件<Component\_H>”;相应地,“C源程序文件夹<C>”包含“总体框架文件夹<Frame\_C>”与“软件构件文件夹<Component\_C>”。需要说明的是在CodeWarrior工程列表中看到的这些文件夹物理存储器里是不存在的,它只负责逻辑区分,便于工程管理,而不具有任何实际意义。

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,程序转到main.c文件中第一个可执行语句执行,随后到main.c文件中定义的主循环执行(理论上对于嵌入式系统而言,主程序一定是一个死循环);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,则返回主程序继续执行。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语言源程序文件分开管理,放在C源程序文件的根目录下。

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,以归类使其便于管理。Frame\_H里包含了type.h、MC9S12XS128.h、GeneralFun.h、MCUInit.h这4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,不必使用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。MC9S12XS128.h是MC9S12XS128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。GeneralFun.h与GeneralFun.c对应,存放与芯片无关的常用且基本的功能性子函数,如延时子函数等。MCUInit.h与Frame\_C子文件夹中的MCUInit.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MCUInit.c文件则包含实际初始化代码。

以上所述,通常是系统正常运行所必需的,它们实现了“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。按照软件构件化原则,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“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,它们分别被放置在这两个文件夹里。

另外,嵌入式系统工程框架中还必须包括部分与编译器相关的文件,如连接文件,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。另外,一个好的工程说明文件,对于一个工程的维护而言是相当重要的。

### 3.4.2 工程文件的物理组织结构

3.4.1 小节从逻辑层面上阐述了一些重要的内容,这一小节将带着读者去看看它们是如何分布在硬盘中的,即在硬盘中的目录分配情况。下面以小灯程序的文件架构作说明。

如图 3-7 所示工程文档结构图,bin 目录下的 .s19 文件为工程最后生成文件,可以通过写入工具写入到 MCU 芯片里运行,类似于 PC 机的 .exe 文件。有的厂家可能直接使用二进制存储,但那样不好直接使用一般文本编辑器阅读。而飞思卡尔公司则使用其定义的文本方式文件(.s19)文件存储机器码,优点是可直接使用通用的文件编辑器打开查看,缺点是下载到芯片之前,必须转为二进制机器码,当然这是下载工具的事情,一般读者不必关心。关于 .s19 文件的含义与组成方式见 3.4.5 小节。而 bin 目录下的 map 文件为代码在 MCU 存储器中的分配提供了证据,建议读者在完成第 5 章学习后,结合实际例子将 map 文件与 s19 对比分析一下,map 文件表明了源代码被编译链接后的机器码,到底被下载到 MCU 内存存储器中的什么地方,在高级调试时,可能需要用到这些知识。



图 3-7 小灯闪烁工程相关源文件的树型(物理)结构

<cmd>目录下为命令文件,可以用于配置编译器完成某些特别的功能,比如软件复位,芯片解密等。

<Project\_Data/Standard/ObjectCode>目录下存放了本工程各.c文件经过编译后产生的目标代码文件(.o文件)。这些文件经过“链接”后将生成可以下载到芯片中执行的机器码文件(.s19文件)。

<Prm>目录下放置 PRM 文件。关于 PRM 文件的含义将在 3.4.3 小节阐述。

<Sources>目录下放置了所以程序头文件和源程序文件,但是文件组织没有 CodeWarrior 中那样的逻辑组织,现在读者应该明白逻辑结构和物理结构的差异。

关于工程根目录下,读者只需要关注下 mcp 文件,这个是一个工程的总文件,只需要通过打开它来打开整个工程。一般的编译器都有这样的一个文件,只是后缀名不同而已。



关于工程中的其他文件,一般为 CodeWarrior 自动生成的,读者一般不需要关心,当然鼓励读者善用搜索引擎,刨根问底。

需要一个小小说明:在阐述工程的逻辑文件结构时使用“文件夹”一词,而在阐述工程的物理结构时使用“目录”一词,没有特别含义,只是为了阐述方便。

3.4.1 小节和 3.4.2 小节分别从逻辑与物理两个层次概述了工程文件的组织情况。下面 3.4.3~3.4.5 小节将对部分文件进行详细的说明,初学者对于这段内容可能感到比较困惑,甚至觉得有点复杂,但认真阅读这些内容并结合理解第一个工程,将有助于学习后面的内容,同时对一个项目有了一个比较全面的认识。

### 3.4.3 系统启动及初始化相关文件

系统启动及初始化相关文件主要指链接文件 Project.prm、启动文件 start08.c 及芯片映像寄存器头文件。这些文件在一个芯片的工程中一般不再改动。

#### 1. 链接文件 Project.prm

网上光盘中【第 03 章(第一个程序)阅读资料】有“关于 CodeWarrior 中 prrm 文件的详细说明”,可参考。这里给出简要说明。

prrm 文件主要实现了芯片的 RAM 和 ROM 的定义,初始化 RAM 中的变量、堆栈的大小;定义复位向量,即应用程序的默认入口;还包含了启动代码,即硬件复位后的函数入口。具体代码如下:

```
// ----- *
// 文件名: Project.prm *
// 说 明: MC9S12XS128 的链接文件 *
// ----- *

NAMES //通过命令行 CodeWarrior 会将所有需要的文件传给链接文件。
/* 但是你也需要在这里添加自己的文件 */

END

SEGMENTS //芯片的所有 RAM/ROM 地址在这里被列出。在下面的 PLACEMENT 中被使用
RAM = READ_WRITE DATA_NEAR 0x2000 TO 0x3FFF;

/* non-banked FLASH */
ROM_4000 = READ_ONLY DATA_NEAR IBCC_NEAR 0x4000 TO 0x7FFF;
ROM_C000 = READ_ONLY DATA_NEAR IBCC_NEAR 0xC000 TO 0xFEFF;

/* paged EEPROM 0x0800 TO 0x0BFF; addressed through EPAGE */
EEPROM_00 = READ_ONLY DATA_FAR IBCC_FAR 0x000800 TO 0x000BFF;
EEPROM_01 = READ_ONLY DATA_FAR IBCC_FAR 0x010800 TO 0x010BFF;
EEPROM_02 = READ_ONLY DATA_FAR IBCC_FAR 0x020800 TO 0x020BFF;
EEPROM_03 = READ_ONLY DATA_FAR IBCC_FAR 0x030800 TO 0x030BFF;
```



```

EEPROM_04    =    READ_ONLY    DATA_FAR IBCC_FAR    0x040800 TO 0x040BFF;
EEPROM_05    =    READ_ONLY    DATA_FAR IBCC_FAR    0x050800 TO 0x050BFF;
EEPROM_06    =    READ_ONLY    DATA_FAR IBCC_FAR    0x060800 TO 0x060BFF;
EEPROM_07    =    READ_ONLY    DATA_FAR IBCC_FAR    0x070800 TO 0x070BFF;
/ * paged FLASH: 0x8000 TO    0xBFFF; addressed through PPAGE * /
PAGE_F8      =    READ_ONLY    DATA_FAR IBCC_FAR    0xF88000 TO 0xF8BFFF;
PAGE_F9      =    READ_ONLY    DATA_FAR IBCC_FAR    0xF98000 TO 0xF9BFFF;
PAGE_FA      =    READ_ONLY    DATA_FAR IBCC_FAR    0xFA8000 TO 0xFABFFF;
PAGE_FB      =    READ_ONLY    DATA_FAR IBCC_FAR    0xFB8000 TO 0xFBFFFF;
PAGE_FC      =    READ_ONLY    DATA_FAR IBCC_FAR    0xFC8000 TO 0xFCBFFF;
PAGE_FE      =    READ_ONLY    DATA_FAR IBCC_FAR    0xFE8000 TO 0xFEBFFF;
END

PLACEMENT    //这里将所有预定义以及用户段都放置到上面定义的 SEGMENTS 中
_PRESTART,                                       //启动代码
STARTUP,                                         //启动数据结构
ROM_VAR,                                         //常量
STRINGS,                                         //串字面值
VIRTUAL_TABLE_SEGMENT,                         //C++有效表段
//.ostext,
NON_BANKED,
COPY                                              //复制信息:如何初始化变量
INTO ROM_C000/ * , ROM_4000 * /;
DEFAULT_ROM INTO PAGE_FE,PAGE_FC,PAGE_FB,PAGE_FA,PAGE_F9,PAGE_F8;
SSTACK,
PAGED_RAM,
DEFAULT_RAM                                     //非零页变量
INTO RAM;

DISTRIBUTE          DISTRIBUTE_INT0
ROM_4000, PAGE_FE, PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8;
CONST_DISTRIBUTE    DISTRIBUTE_INT0
ROM_4000, PAGE_FE, PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8;
DATA_DISTRIBUTE     DISTRIBUTE_INT0
RAM;
END
ENTRIES
END
STACKSIZE 0x200
VECTOR 0 _Startup //复位向量:这是应用程序的默认入口

```



按所含的信息 prn 文件有 5 个组成部分构成：

### (1) NAMES ~ END 部分

用以指定在链接时加入除本项目文件列表之外的额外目标代码文件,这种用法不常用,因为在“Libs”栏目添加可实现同样功能。本书该部分为空。

### (2) SEGMENTS ~ END 部分

这部分很重要,它定义和划分了芯片所有可用的内存资源,包括程序空间和数据空间。一般将程序空间定义成 ROM,把数据空间划分成第 0 页的 Z\_RAM 和非 0 页的 RAM,这些名字都不是系统保留关键字,可以由用户随意修改。但修改后其他部分用到必须相应更改,因此,要慎重更改。

内存划分的具体方式如下：

由 SEGMENTS 开始到 END 为止,中间可以添加任意多行内存划分的定义,每一行用分号“;”结尾。定义行的语法格式为：

[块名] = [属性] [起始地址] TO [结束地址];

**块名:**其定义和 C 语言变量定义相同,是以英文字母和下划线开头的一个字符串。

**属性:**可以有 3 种不同的类型。对于只读的 Flash - ROM 区属性一定是 READ\_ONLY;对于可读/写的 RAM 区属性可以是 READ\_WRITE,也可以是 NO\_INIT,它们两者的关键区别是 ANSI - C 的初始化代码会把定位在 READ\_WRITE 块中的所有全局和静态变量自动清零,而 NO\_INIT 块中的变量将不会被自动清零。对于单片机系统,变量在复位时不被自动清零这一特性有时是很关键的。

**起始地址和结束地址:**决定了一内存块的物理位置,用 16 进制表示。

用 SEGMENTS 只是从 MCU 的物理内存这一角度对其进行空间划分。源程序本身并不知道物理内存被分割和属性定义的这些细节。它们两者之间必须通过下面的 PLACEMENT 建立联系。

### (3) PLACEMENT ~ END 部分

指派源程序中所定义的各种段,例如数据段 DATA\_SEG、CONST\_SEG 和代码段 CODE\_SEG 被具体放置到哪一个内存块中。它是将源程序中的定义描述和实际物理内存挂钩的桥梁。

PLACEMENT ~ END 内所描述的信息是告诉链接器源程序中所定义的各类段应该被具体放置到哪一个内存块中去。其语法格式为：

[段名 1], [段名 2], ... [段名 n] INTO [内存块名];

**段名:**就是在源程序中用 #pragma 声明的数据段、常数段或代码段的名称。如果用默认名 DEFAULT,则默认的数据段名为 DEFAULT\_RAM,代码段和常数段名为 DEFAULT\_ROM。若程序中定义的段名没有在 PLACEMENT 中提及,则将被视为 DEFAULT。几个相同性质但不同名字的段可以被放置到同一个内存块中,相互之间用逗号分隔。INTO 是系统保



留的关键词,在这里为放入的意思。内存块名就是前面介绍的用 SEGMENTS 划分好的不同的内存块名字。利用这样直观的定位描述文本可以方便灵活地将数据或代码定位到芯片内存任意可能的位置,实现某些特殊目的的应用。网上光盘中【第 03 章(第一个程序)阅读资料】的“关于 CodeWarrior 中 prrm 文件的详细说明”含有相关例子。

#### (4) STACKSIZE 部分

STACKSIZE 部分定义系统堆栈长度,其后给出的长度字节数可以根据实际应用需要进行修改。堆栈的实际定位取决于 RAM 内存的划分和使用情况。在常见的 RAM 线性划分变量连续分配的情况下,堆栈将紧挨在用户所定义的所有变量区域的高端。但如果将 RAM 区分成几个不同的块,必须确保其中至少有一个块能容纳已经定义的堆栈长度。

#### (5) VECTOR 部分

VECTOR 部分定义所有矢量入口地址。模板在生成 prrm 文件时已经定义了复位矢量的入口地址。对于各类中断矢量用户必须自己按矢量编号和中断服务函数名相关联。如果中断函数的定义是用“interrupt”加上矢量号,则无需在这里重复定义。

编写中断函数几乎是每一个单片机项目开发必需的一个内容。CW 针对 12 系列 MCU 的中断函数编写有 3 种方式可以实现,分别是:关键词 interrupt 和中断矢量编号;用关键词 interrupt 定义中断函数,中断矢量入口由 prrm 文件指定;用 #pragma TRAP\_PROC 定义中断函数,中断矢量入口由 prrm 文件指定。详见网上光盘中【第 03 章(第一个程序)阅读资料】的“关于 CodeWarrior 中 prrm 文件的详细说明”。

## 2. 关于 #pragma 指令的简要说明

上面对 prrm 文件的总体结构已经说明完毕,下面补充些 #pragma 指令的使用方法,它与 prrm 文件还是很有关联的。

#pragma 声明是基于单片机开发的特点面对标准 C 语法的一个扩充,对充分利用 MCU 内各类有限的资源起到不可或缺的关键作用。下面简单介绍几个最常用的 #pragma 声明。

#### (1) #pragma DATA\_SEG

定义变量所处的数据段。其语法形式为:

#pragma DATA\_SEG 名称

数据段名称可以自己任意命名,但习惯上有些约定的名称,其作用分别为:

DEFAULT—默认的数据段,在 12 系列单片机中的地址为 0x100 以上。一般的变量定义可以放在这一区域。

MY\_ZEROPAGE—特指第 0 页数据段,地址范围 0x00~0xff,但实际用户可用的空间不到 256 字节,因为前面的一些地址空间已经分配给了片内寄存器。需要频繁或快速存取的变量应该指定放在这一特殊区域,特别是位变量。

数据段名称必须和 prrm 文件中的数据段配置说明相关连才能真正发挥其定位作用。如

果自己命名的数据段在 prn 文件中没有特别说明,那此数据段的性质等同于 DEFAULT,否则可以使用自己在 prn 文件中定义的名称。

## (2) #pragma CONST\_SEG

定义一个常数数据段,必须和变量的 const 修饰关键词配合使用。其语法型式为:

```
#pragma CONST_SEG 名称
```

该数据段下定义的所有数据将被放置在程序只读的 ROM 区,也就是 MCU 内的 Flash 程序空间区。常数段名称可以用户自由定义,但一般用 DEFAULT,让链接器按可用的 ROM 区域自由分配常量位置。举例如下:

```
//这个 const 将会被放置在 flash 区中
#pragma CONST_SEG DEFAULT
const byte prjName = "This is a demo";
const word version = 0x0301;    //这个 const 将会被放置在 flash 区中
//没有 const 该变量将被放置在 RAM 区
#pragma CONST_SEG DEFAULT
word version = 0x0301;          //没有 const 该常量将被放置在 RAM 区
//尽管有 const 但该常量将被放置在 RAM 区
#pragma DATA_SEG DEFAULT
const word version = 0x0301;    //尽管有 const 但该常量将被放置在 RAM 区
```

在嵌入式系统中,程序与变量分别放在不同的物理介质中(程序在 Flash 中,变量在 RAM 中),常数最好也被设置放在 Flash 中。

## (3) #pragma CODE\_SEG

用以定义程序段并赋以特定的段名,语法型式如下:

```
#pragma CODE_SEG 名称
```

一般的程序设计是无需对代码段做特殊处理的。对于项目中所有的代码文件或库文件,链接器会在最后按程序模块出现的先后顺序挨个自动安排所有程序函数在内存中所处的实际位置,用户不必太关心某一个函数的具体位置。但某些特殊的设计需要将不同部分的程序分别定位到不同的地址空间,例如实现程序代码下载自动更新。这样的设计需要把负责应用程序下载更新的驱动代码固定放置在一个保留区域内,而把一般的应用程序放置在另外一个区域以便在需要时整体擦除后更新。这时就需要用 CODE\_SEG 来分别指明不同的程序段,还必须配合并利用 prn 文件对程序空间进行分配和指派。

## (4) #pragma TRAP\_PROC

用于定义一个函数为中断服务类型。此类型的函数编译器在将 C 代码编译成汇编指令时会在代码前后增加必要的现场保护和恢复汇编代码,同时函数的最后返回用汇编指令 RTI 而不是针对普通函数的 RTS。例如:



```
#pragma TRAP_PROC
void SCI1_Int(void)
{
...      //定义 SCI1 的中断服务程序
}
```

注意用 TRAP\_PROC 定义的中断服务函数的实际中断矢量地址必须通过 prn 文件指派。

### 3. start12.c 文件及启动过程

<Project Settings>(工程设置文件)目录下有个 start12.c 文件,其中有一段代码如下:

```
//-----*
//函数名: _Startup                                     *
//功 能: 1)初始化堆栈                                 *
//        2)调用 init()初始化 RAM,复制初始数据等      *
//        3)调用 main 函数                             *
//参 数: 无                                           *
//返 回: 无                                           *
//说 明: (1)链接文件(Project.prm)中的语句:VECTOR 0 _Startup将复位向量设 *
//        置为_Startup                                *
//        (2)从链接文件生成的代码_PRESTART处调用      *
//-----*
#pragma NO_EXIT

#if defined(__SET_RESET_VECTOR__)
__EXTERN_C void __interrupt 0 _Startup(void) {          // _Startup,启动入口
#else
__EXTERN_C void _Startup(void) {
#endif

    INIT_SP_FROM_STARTUP_DESC();                        //初始化堆栈
/* 省略其间的部分代码,详见网上光盘源代码 */
#ifndef __ONLY_INIT_SP
    Init();                                              //初始化函数
#endif

#ifdef _DO_ENABLE_COP_
    _ENABLE_COP(1);
#endif
```

```
main();           //调用主函数
}
```

这段程序包括设置堆栈、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。不建议修改该文件。CodeWarrior 启动模块程序实现步骤主要如下:

- ① 设置堆栈指针,将其映射到 RAM 空间。
- ② 初始化 RAM,复制初始数据。将初始化数据从 ROM 复制到 RAM。
- ③ 跳转到主函数 main() 执行。

start12.c 文件的开头部分有一个“#include <start12.h>”语句,所包含的 start12.h 头文件在安装目录的“..\CodeWarrior for S12(X) V5.0\lib\hc12c\include”文件夹中。用鼠标选取后,右击并选择 Find and Open Files,则可转到查看相应文件内容。

#### 4. XS128 映像寄存器头文件 MC9S12XS128.h

MC9S12XS128.h 中定义了编程时需要访问的外设寄存器,不修改该文件。例如,在 XS128.h 中定义 C 口的数据寄存器的定义形式如下:

```
/* * * PORTE - Port E Data Register; 0x00000008 * * */
typedef union {
    byte Byte;
    struct {
        byte PE0      :1;      /* Port E Bit 0 */
        byte PE1      :1;      /* Port E Bit 1 */
        byte PE2      :1;      /* Port E Bit 2 */
        byte PE3      :1;      /* Port E Bit 3 */
        byte PE4      :1;      /* Port E Bit 4 */
        byte PE5      :1;      /* Port E Bit 5 */
        byte PE6      :1;      /* Port E Bit 6 */
        byte PE7      :1;      /* Port E Bit 7 */
    } Bits;
} PORTESTR;

extern volatile PORTESTR _PORTE @(REG_BASE + 0x00000008);
#define PORTE                _PORTE.Byte
#define PORTE_PE0            _PORTE.Bits.PE0
#define PORTE_PE1            _PORTE.Bits.PE1
#define PORTE_PE2            _PORTE.Bits.PE2
#define PORTE_PE3            _PORTE.Bits.PE3
#define PORTE_PE4            _PORTE.Bits.PE4
#define PORTE_PE5            _PORTE.Bits.PE5
```



```

#define PORTE_PE6                _PORTE.Bits.PE6
#define PORTE_PE7                _PORTE.Bits.PE7

#define PORTE_PE0_MASK           1
#define PORTE_PE1_MASK           2
#define PORTE_PE2_MASK           4
#define PORTE_PE3_MASK           8
#define PORTE_PE4_MASK          16
#define PORTE_PE5_MASK          32
#define PORTE_PE6_MASK          64
#define PORTE_PE7_MASK         128

```

这个定义说明如下：

① 定义了一个联合体 PORTESTR, 其中包含一个字节定义 Bits, 将每个位分别进行定义。

② “extern volatile PORTESTR \_PORTE @(REG\_BASE + 0x00000008);”语句将该联合体定位到了地址 0x00000008。同时还将 PORTESTR 等效为 \_PORTE。

③ “#define PORTE \_PORTE.Byte”语句将联合体 \_PORTE(同 PORTESTR) 中的 Byte 定义为 PORTE。这样, 当引用到 PORTE 的时候, 使用的是地址 0x00000008 中的内容。

### 3.4.4 芯片初始化、主程序、中断程序及其他文件

#### (1) 系统初始化构件(MCUInit.h 与 MCUInit.c)

系统初始化操作是由 MCUInit.c 来实现的, 具体内容参见第 11 章。由于这部分内容对初学者来说较难理解, 可先使用后理解。MCUInit.c 所包含的头文件中给出了开关总中断的宏定义, 以便中断程序、主程序或其他程序中使用。

#### (2) 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。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。

#### (3) 中断文件 isr.c

中断文件 isr.c 给出中断程序框架, 具体使用方法见“5.4 节关于 XS128 的中断源与第一个带有中断的编程实例”的介绍。

中断定义文件 isr.c 的格式：

```

// ----- *
// 文件名: isr.c                                *
// 说 明: 中断处理函数, 本文件包含:          *

```

```
//          isrDummy: 中断处理程序                                *
// -----*

//总头文件
#include "include.h"
//此处为用户新定义中断处理函数的存放处
//未定义的中断处理函数,本函数不能删除
interrupt void isr_default (void)
{
}

//中断处理子程序类型定义
typedef void( * tIsrFunc)(void);

//中断矢量表,如果需要定义其他中断函数,请修改下表中的相应项目
const tIsrFunc _InterruptVectorTable[] @0xFF10 = {
    isr_default,      /* 0xFFD6      串口中断 */
    isr_default      /* 0xFFD8      SPI 中断 */
//RESET 是特殊中断,其向量由开发环境直接设置(在本软件系统的 Start12.o 文件中)
};
```

## (4) 芯片无关文件

### 1) 类型定义文件 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 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 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
//不优化变量类型

typedef volatile uint8  vuint8;   // 8 位无符号数
typedef volatile uint16 vuint16;  // 16 位无符号数
typedef volatile uint32 vuint32;  // 32 位无符号数
typedef volatile int8   vint8;    // 8 位有符号数
typedef volatile int16  vint16;   // 16 位有符号数
typedef volatile int32  vint32;   // 32 位有符号数
#endif
```

在定义不优化数据类型时,使用关键字 `volatile`。`volatile` 关键字用于通知编译器,对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。对 MCU 的映像寄存器的操作就不能优化,否则,对 I/O 口的写入可能被“优化”写入到 CPU 内部寄存器中,就会乱套。

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` 文件中可以定义经常使用的一些函数和宏,例如延时。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,也可设置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
```

(5) 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。对于一个大的工程项目而言,说明文件还是相当重要的,读者可以逐渐形成自己的工程风格,为自己积累项目文档规范。

3.4.5 机器码文件(.s19 文件)的简明解释

在编译过程中,CodeWarrior 会生成目标代码文件(.o 文件)。如上述工程的目标代码文件在工程目录下的“..\Project\_Data\Standard\ObjectCode”子目录中,打开这个目录,会看到 GeneralFun.o、GPIO.o、isr.o、Light.o、main.o、MC9S12XS128.o、MCUInit.o、Start12.o 等目标文件,这些文件是源文件经编译而生成的。目标代码文件经过“链接”,就生成最终文件“Project.abs.s19”,存放于 bin 目录下,这个文件可用 Windows 操作系统的附件程序中的记事本或写字板打开阅读。它是 Freescale MCU 的机器码文件,以文本文件的形式存储在磁盘上。利用 CodeWarrior 环境或写入器,可以将机器码下载到目标 MCU 内的 Flash 存储器,以便运行。

下面对 S19 文件格式进行必要的阐述。

S-record 格式文件是 Freescale CodeWarrior 编译器生成的后缀名为 .S19 的程序文件,是一段直接写进 MCU 的 ASCII 码,英文全称为 Motorola format for EEPROM programming。目标文件由若干行 S 记录构成,每行 S 记录用 CR/LF/NUL 结尾。一行 S 记录由下列 5 部分组成,每行最大是 78 个字节,如表 3-2 所列。

表 3-2 S 记录格式

类 型	记录长度	地 址	代码/数据	校验和
2 字节	2 字节	2、3 或 4 字节	0~n 字节	1 字节

(1) 类型(2 个字节)

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 3-2 给出了 S 记录的格式。

S0—地址位没有被用,用零置位(0x0000)。数据位中的信息被划分为以下 4 个子域:

- name(名称):20 个字符,用来编码单元名称;
- ver(版本):2 个字符,用来编码版本号;
- rev(修订版本):2 个字符,用来编码修订版本号;



description(描述):0~36 个字符,用来编码文本注释。

此行(S0 开头的记录)表示程序的开始,不需写入 memory。

S1—该记录包含代码/数据以及 2 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 3 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 4 个字节存储其代码/数据的存储器首地址。

S5—该记录包含前面 S1,S2,S3 记录的计数。

S7—对应 S3 记录的结束记录,4 个字节地址,不需要写入 MCU 中。

S8—对应 S2 记录的结束记录,3 个字节地址,不需要写入 MCU 中。

S9—对应 S1 记录的结束记录,2 个字节地址,不需要写入 MCU 中。

每个 S 记录块都使用唯一的终止记录。

## (2) 记录长度

表示该记录行中剩余字符对的数目,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 3 个部分所有字节之和的反码的低 8 位,用于校验。

下面是<Ch03 - GPIO(Light)\_C>工程<bin>子目录下的 Project.abs.s19 的部分内容:

```
S0590000443A5C533132582D424F4F4. .... 3A
...
S123C000CF2100C6055B134A8039FE4A8000FE0000C015C03B00000000972704580430FC10
...
S224FE8060F10A4A806DFE4D1E4079003C0A7900394C3A40CC81435B345A35790036A7A7FF1
...
S9030000FC
```

第一行为 S0 记录,表示文件名信息。S0 之后的 59 是十六进制数(十进制 89),表示后面有 89 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 3A 是本记录的校验和。

下一数据行: S123C000CF2100C6055B134A8039FE4A8000FE0000C015C03B00000000972704580430FC10, 前两个符号 S1 表示这一行是 S1 记录, 其后的“23”是十六进制数(十进制数的 35), 表示在此行其后有 35 个字节的数据, 包括 2 个字节的地址 C000、32 个字节的代码/数据, 最后 1 个字节是前面数据按照一定算法的出的校验和 10。该行记录所表示的实际代码/数据 CF2100C6055B134A8039FE4A8000FE0000C015C03B00000000972704580430FC 将被装入起始地址为 0xC000 的 MCU 存储器中。

再下一行: S224FE8060F10A4A806DFE4D1E4079003C0A7900394C3A40CC81435B345A35790036A7A74FF1, 前两个符号 S2 表示这一行是 S2 记录, 其后的“24”是十六进制数(十进制数的 36), 表示在此行其后有 36 个字节的数据, 包括 3 个字节的地址 FE8060、32 个字节的代码/数据, 最后 1 个字节是前面数据按照一定算法得出的校验和 F1。该行记录所表示的实际代码/数据 F10A4A806DFE4D1E4079003C0A7900394C3A40CC81435B345A35790036A7A74F 将被装入起始逻辑地址为 0xFE\_8060 的 MCU 存储器中。

最后一行是 S9 记录, S9 之后的 03 是十六进制 0x03, 表示其后有 3 个字节的/数据。0000 为 2 个字节的地址, FC 是校验和。实际上 S9 记录是 S1 记录的结束记录, 这里的地址字段只是用“0000”填充, 并没有实际意义。

### 3.4.6 lst 文件与 map 文件

编译链接还产生 lst 文件与 map 文件, 这里仅给出简要说明。

#### 1. lst 文件

打开工程后双击某一个源程序文件, 则相应文件内容显示在窗口中, 此时, 在源程序窗口右击, 从弹出菜单中选取 Disassemble 可查看 lst 文件(列表文件), 这是编译过程产生的文件。

列表文件给出了 C 语言编译后的机器码、偏移地址、对应的汇编语句信息, 是分析程序的工具之一。下面给出第一个工程的列表文件。

```
13:  #include "Includes.h"
14:
15:  void main(void)
16:  {
17:      /* 关总中断 */
18:      DisableInterrupt();                //禁止总中断
0000 1410          [1]      SEI
19:      /* 芯片初始化 */
20:      MCUInit(FBUS_32M);
0002 4a000000      [7]      CALL  MCUInit,PAGE(MCUInit)
```



```

21:      /* 指示灯初始化 */
22:      Light_Init(Light_Run_PORT,Light_Run,Light_OFF);
0006 c7          [1]      CLRB
0007 87          [1]      CLRA
0008 3b          [2]      PSHD
0009 52          [1]      INCB
000a 37          [2]      PSHB
000b c7          [1]      CLRB
000c 4a000000    [7]      CALL  Light_Init,PAGE(Light_Init)
0010 1b83        [2]      LEAS   3,SP

23:      /* 主循环 */
24:      while (1)
25:      {
26:                                     //Light_Run 灯先亮后暗
27:      Light_Control(Light_Run_PORT,Light_Run,Light_ON);
0012 c7          [1]      CLRB
0013 87          [1]      CLRA
0014 3b          [2]      PSHD
0015 52          [1]      INCB
0016 37          [2]      PSHB
0017 4a000000    [7]      CALL  Light_Control,PAGE(Light_Control)
001b 1b83        [2]      LEAS   3,SP

28:      Delay(3000);                  //延时适当时间
001d cc0bb8      [2]      LDD    # 3000
0020 4a000000    [7]      CALL  Delay,PAGE(Delay)

29:      Light_Control(Light_Run_PORT,Light_Run,Light_OFF);
0024 c7          [1]      CLRB
0025 87          [1]      CLRA
0026 3b          [2]      PSHD
0027 52          [1]      INCB
0028 37          [2]      PSHB
0029 c7          [1]      CLRB
002a 4a000000    [7]      CALL  Light_Control,PAGE(Light_Control)
002e 1b83        [2]      LEAS   3,SP

30:      Delay(3000);                  //延时适当时间
0030 cc0bb8      [2]      LDD    # 3000
0033 4a000000    [7]      CALL  Delay,PAGE(Delay)
0037 20d9        [3]      BRA    * - 37 ;abs = 0012

31:      }

```

```
32:
33: }
```

## 2. map 文件

打开工程后,展开“工程设置文件夹<Project Settings>”后,再展开 Linker Files 文件夹,可以看到“Project.map”文件,通常称之为工程的“映像文件”;这个文件告诉我们,源代码被编译链接后的机器码,到底被下载到 MCU 内存存储器中的什么地方,在高级调试时,可能需要用到这些知识。读者应仔细琢磨一下这个文件。

### 3.4.7 如何在 CW 环境下新建一个 S12 工程

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复制一份继续进行新的工程编程。

第一种方法的操作步骤如下:

选择 File→New Project 菜单项,则弹出新建对话框,选择 HCS12X→HCS12XS Family→MC9S12XS128,单击“下一步”,并选中“C”的选项,如果程序中有汇编代码则应该选中 Relocatable assembly,在右侧 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。这种方法适用于刚开始建立程序的模板,以后可以直接采用第二种方法,简单、方便、快捷。

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。

## 3.5 第一个 C 语言工程:控制小灯闪烁

本书用 XS128 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 XS128 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。XS128 的普通 I/O 口控制指示灯闪烁的 C 语言工程实例的文件在网上光盘内。

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 4 章阐述。



### 3.5.1 GPIO 构件设计

GPIO 引脚可以定义成输入、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。

说明:下面代码中将出现 `(* (vuint8 *))` 语法现象。这个看起来很复杂,其实可以分解开来,它的 C 语言定义为:`( * ( volatile unsigned char * ) )`,其中 `volatile` 是指告诉编译器,这个值与外界环境有关,不要对它优化,即不缓存它的值。`(volatile unsigned char * )` 是 C 语言中的强制类型转换,它的作用是把十六进制转换为一个地址指针,而接下来在它的外面的 `*`,就是取地址中的内容。

#### 1. GPIO 构件的头文件(GPIO.h)

```
// -----*
// 文件名: GPIO.h                                     *
// 说 明: GPIO 构件头文件                             *
// -----*

#ifndef GPIO_H
#define GPIO_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h" //包含总头文件

//XS128 端口名与地址的对应宏定义
#define PA      0x0000
#define PB      0x0001
#define PE      0x0008
#define PK      0x0032
#define PT      0x0240
#define PS      0x0248
#define PM      0x0250
#define PP      0x0258
```



```
#define PH      0x0260
#define PJ      0x0268
#define PAD0    0x0270
#define PAD1    0x0271
```

//端口的各个寄存器间偏移地址的对应关系

```
#define PRT    0      //数据寄存器
#define PTI    1      //输入寄存器
#define DDR    2      //方向寄存器
#define RDR    3      //低功耗驱动寄存器
#define PER    4      //上拉下拉使能寄存器
#define PPS    5      //上拉下拉极性选择寄存器
#define PIE    6      //引脚中断允许寄存器
#define PIF    7      //引脚中断标志寄存器
#define WOM    6      //引脚线或寄存器
#define PRR    7      //引脚功能选择寄存器
```

//构件函数声明区

```
void GPIO_Init(uint16 port,uint8 pin,uint8 direction,uint8 state);//初始化 GPIO
uint8 GPIO_Get(uint16 port,uint8 reg,uint8 pin);//获得引脚状态
void GPIO_Set(uint16 port,uint8 reg,uint8 pin,uint8 state);//设置引脚状态
uint8 GPPort_Get(uint16 port,uint8 reg);//获取端口各引脚状态
void GPPort_Set(uint16 port,uint8 reg,uint8 setFlag,uint8 bValue);//设置端口各引脚状态
```

#endif

## 2. GPIO 构件的程序文件(GPIO.c)

```
// ----- *
// 文件名: GPIO.c                                *
// 说 明: GPIO 驱动程序文件                      *
// ----- *
```

//头文件包含,及宏定义区

```
//头文件包含
#include "GPIO.h"
//2 个内部函数
uint8 ABEK_Bit(uint16 port);
uint8 ADEK_AddrChg(uint16 port,uint8 reg);
```

//构件函数实现



```
//-----*
//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(uint16 port,uint8 pin,uint8 direction,uint8 state)
{
    GPIO_Set(port,DDR,pin,direction);
    //1. 引脚方向为输出时,设置引脚状态
    if(direction == 1) GPIO_Set(port,PRT,pin,state);
}

//-----*
//函数名: GPIO_Get                                     *
//功 能: 获得引脚状态                                   *
//参 数: port: 端口名                                   *
//       reg: 指定端口寄存器                           *
//       pin: 指定端口引脚                             *
//返 回: state:得到引脚状态(ABEK 上拉或低功耗,效果 = 0xFF 或 = 0x00) *
//-----*

uint8 GPIO_Get(uint16 port,uint8 reg,uint8 pin)
{
    uint8 reval = 0,bit = 0;
    //获取 A、B、E、K 的上拉允许寄存器和低功耗驱动设置值
    if((reg == PER) || (reg == RDR))
    {
        bit = ABEK_Bit(port);
        if((reg == PER) && (bit != 8))
        {
            reval = BGET(bit,PUCR );
            return reval;
        }
        else if((reg == RDR) && (bit != 8))
        {
            reval = BGET(bit,RDRIV);
        }
    }
}
```

```

        return reval;
    }
}

//调整方向寄存器偏移量
reg = ADEK_AddrChg(port,reg);
//获取寄存器的设置值
reval = BGET(pin,(* (vuint8 *) (port + reg)));    //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                            *
//      reg  :指定端口寄存器                      *
//      pin  :指定端口引脚                        *
//      state:0 = 低电平,1 = 高电平(ABEK 上拉或低功耗,效果 = 0xFF 或 = 0x00)*
//返 回: 无                                        *
// ----- *

void GPIO_Set(uint16 port,uint8 reg,uint8 pin,uint8 state)
{
    uint8 bit = 0;
    //A、B、E、K 的上拉允许寄存器和低功耗驱动寄存器设置
    if((reg == PER) || (reg == RDR))
    {
        bit = ABEK_Bit(port);
        if((reg == PER) && (bit != 8))
        {
            if(state == 1) BSET(bit,PUCR);
            else BCLR(bit,PUCR);
            return;
        }
        else if((reg == RDR) && (bit != 8))
        {
            if(state == 1) BSET(bit,RDRIV);
            else BCLR(bit,RDRIV);
            return;
        }
    }
}

```



```
//调整方向寄存器偏移量
reg = ADEK_AddrChg(port,reg);
//设置寄存器的值
if(state == 1)
    BSET(pin,( * (vuint8 * )(port + reg)));    //输出高电平(1)
else
    BCLR(pin,( * (vuint8 * )(port + reg)));    //输出低电平(0)
}

//-----*
//函数名: GPPort_Get                                *
//功 能: 获得获取端口各引脚状态                    *
//参 数: port: 端口名                                *
//      reg: 指定端口寄存器                          *
//返 回: 得到端口状态(对于 ABEK 上拉或低功耗,效果 = 0xFF 或 = 0x00) *
//-----*
uint8 GPPort_Get(uint16 port,uint8 reg)
{
    uint8 reval = 0,bit = 0;
    //获取 A、B、E、K 的上拉允许寄存器和低功耗驱动设置值
    if((reg == PER) || (reg == RDR))
    {
        bit = ABEK_Bit(port);
        if((reg == PER)&&(bit != 8))
        {
            reval = BGET(bit,PUCR );
            if(reval == 1) reval = 0xFF;
            else reval = 0x00;
            return reval;
        }
        else if((reg == RDR)&&(bit != 8))
        {
            reval = BGET(bit,RDRIV);
            if(reval == 1) reval = 0xFF;
            else reval = 0x00;
            return reval;
        }
    }
}

//调整方向寄存器偏移量
```

```

reg = ADEK_AddrChg(port,reg);
//获取寄存器的设置值
reval = ( * (vuint8 *) (port + reg));    //得到引脚的状态
return reval;
}

// ----- *
//函数名: GPPort_Set *
//功 能: 设置端口各引脚状态 *
//参 数: port    : 端口名 *
//      reg     : 指定寄存器 *
//      setFlag: = 0 时, bValue 所有 = 0 的位, 寄存器对应位清 0; *
//              = 1 时, bValue 所有 = 1 的位, 寄存器对应位置 1; *
//              = 其他值, 寄存器 = bValue *
//      bValue: 指定值(对于 ABEK 上拉或低功耗, 效果 = 0xFF 或 = 0x00) *
//返 回: 无 *
// ----- *
void GPPort_Set(uint16 port,uint8 reg,uint8 setFlag,uint8 bValue)
{
    uint8 bit = 0;
    //A,B,E,K 的上拉允许寄存器和低功耗驱动寄存器设置
    if((reg == PER) || (reg == RDR))
    {
        bit = ABEK_Bit(port);
        if((reg == PER)&&(bit != 8))
        {
            if(bValue == 0xFF) BSET(bit,PUCR);
            else if(bValue == 0x00) BCLR(bit,PUCR);
            return;
        }
        else if((reg == RDR)&&(bit != 8))
        {
            if(bValue == 0xFF) BSET(bit,RDRIV);
            else if(bValue == 0x00) BCLR(bit,RDRIV);
            return;
        }
    }
}

//调整方向寄存器偏移量
reg = ADEK_AddrChg(port,reg);

```



```
//设置寄存器的值
if(setFlag==1)      ( *(vuint8 *)(port+reg)) |= bValue;
else if(setFlag==0) ( *(vuint8 *)(port+reg)) &= bValue;
else                ( *(vuint8 *)(port+reg))  = bValue;
}

//2 个内部函数的实现
//-----*
//函数名: ABEK_Bit                                     *
//功 能: A、B、E、K 的上拉允许寄存器和低功耗驱动寄存器的位确定 *
//参 数: port:端口名                                   *
//返 回: 寄存器对应的位值                               *
//-----*
uint8 ABEK_Bit(uint16 port)
{
    uint8 bit = 0;
    switch(port)
    {
        case PA:bit = 0;break;
        case PB:bit = 1;break;
        case PE:bit = 4;break;
        case PK:bit = 7;break;
        default:bit = 8;
    }
    return bit;
}

//-----*
//函数名: ADEK_AddrChg                                 *
//功 能: 调整方向寄存器偏移量                           *
//参 数: port:端口名                                   *
//      reg :指定端口寄存器                             *
//返 回: 寄存器地址偏移量修正值                         *
//-----*
uint8 ADEK_AddrChg(uint16 port,uint8 reg)
{
    if((port == PE) || (port == PK))
    {
        if(reg == DDR) reg = DDR - 1;
    }
}
```

```

    }
    else if((port == PAD0) || (port == PAD1))
    {
        if(reg == RDR) reg = RDR + 1;
        else if(reg == PER) reg = PER + 2;
    }
    return reg;
}

```

### 3.5.2 Light 构件设计

控制指示灯的亮或暗,通过调用 GPIO 构件完成。假设有 3 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 3 盏灯分别起个名字,分别叫 Light\_Run、Light\_Error、Light\_Link。它们所接在 MCU 的 GPIO 端口的名字分别叫 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,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 *
// 说 明: 指示灯驱动程序头文件 *
// ----- *

#ifndef Light_H    //避免重复相同的文件
#define Light_H

//头文件包含区
#include "GPIO.h" //包含 GPIO 构件

//宏定义区

//1. 灯状态宏定义(根据实际电路定义)
#define Light_ON      1    //灯亮(对应高电平)
#define Light_OFF     0    //灯暗(对应低电平)

//2. 灯控制引脚定义
#define Light_Run_PORT PA    //Run(运行指示)灯使用的端口

```





```
#define Light_Run      1      //Run(运行指示)灯使用的引脚

#define Light_Fun1_PORT PA    //Fun1 灯使用的端口
#define Light_Fun1     2      //Fun1 灯使用的引脚

//构件函数声明区
void Light_Init(uint16 port,uint8 name,uint8 state);//初始化指示灯状态
void Light_Control(uint16 port,uint8 name,uint8 state);//控制灯的亮和暗
void Light_Change(uint16 port,uint8 name);//状态切换
#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(uint16 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(uint16 port,uint8 name,uint8 state)
```

```
{
    GPIO_Set(port,PRT,name,state);    //控制引脚状态
}

// ----- *
//函数名: Light_Change                *
//功 能: 状态切换,原来[暗],则变[亮];原来[亮],则变[暗] *
//参 数: port:端口名;                *
//      name:指定端口引脚号;         *
//返 回: 无                          *
// ----- *

void Light_Change(uint16 port,uint8 name)
{
    if(GPIO_Get(port,PRT,name) == Light_ON)    //若原来为"亮",则变"暗"
        GPIO_Set(port,PRT,name,Light_OFF);
    else                                         //若原来为"暗",则变"亮"
        GPIO_Set(port,PRT,name,Light_ON);
}
```

### 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,较明显地看到指示灯闪烁的现象。

```
// ----- *
// 工 程 名: Light                *
// 硬件连接: 见 Light. h 文件      *
// 程序描述: 用 Light 构件控制小灯闪烁 *
// 目    的: 第一个 C 语言程序编程框架 *
// 说    明: 提供 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研发中心 2011 年 ----- *
//包含头文件
#include "Includes. h"    //包含总头文件
//在此添加全局变量定义
//主函数
void main(void)
{
    //0.1 主程序使用的变量定义
```



```
uint32 mRuncount = 0;    //运行计数器

//0.2 关总中断
DisableInterrupt();

//0.3 芯片初始化
MCUInit(FBUS_32M);

//0.4 模块初始化
Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
Light_Init(Light_Fun1_PORT,Light_Fun1,Light_OFF); //Fun1 灯初始化为暗

// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 2)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT,Light_Run);    //指示灯的亮、暗状态切换
    }

    // -----
    //2. Fun1 灯先亮后暗
    Light_Control(Light_Fun1_PORT,Light_Fun1,Light_ON);
    Delay(3000);    //延时适当时间
    Light_Control(Light_Fun1_PORT,Light_Fun1,Light_OFF);
    Delay(3000);    //延时适当时间
    // -----

} //for_end(主循环结束)
} //main_end
```

### 3.5.4 理解第一个 C 工程的执行过程

当 XS128 芯片上电复位(冷复位)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 Star12.c 文件中的\_Startup 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 \*.prm 文件中都可以找到复位中断向量,该向量的前面是堆栈初始化指针,分别如下:

```
STACKSIZE 0x100          //初始化堆栈大小
VECTOR 0 _Startup        // 复位向量
```

### (2) 执行 Start12.c 文件中的 \_Startup 函数

执行 Start12.c 文件中的 \_Startup 函数,并转向 main 函数。Start12.c 文件是系统启动文件,它的主体代码在 3.5.1 小节已经给出。具体包括以下过程:初始化堆栈;初始化 RAM,复制初始数据,初始化 ROM;系统模块初始化并跳转到 main 主程序。

若代码执行顺利,程序会执行到如下指令:

```
/* call main() */
main();
```

当执行到“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。这样整个程序的执行可被视为具有两条主线,一方面 main 主程序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 将转到中断向量表文件 isr.c 所指定的中断入口地址处开始执行中断服务程序(ISR, Interrupt Service Routine)。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。

## 3.6 第一个汇编语言工程:控制小灯闪烁

相对于其他的高级语言编程,汇编语言在编程的直观性、编程效率等方面有所欠缺,但针对资源相对较少的 MCU 以及时序要求严格的硬件接口编程,掌握汇编语言还是必不可少的。使用汇编语言编程是基本功,学习和掌握汇编语言编程可以增加编程者的“内力”,为使用高级



语言编程打下坚实的基础。有关 XS128 汇编语言的细节与规范请开发环境的帮助系统。

在本书教学资料中提供的 CodeWarrior 开发环境中,汇编程序是通过工程的方式组织起来的。汇编工程通常包含芯片相关的程序框架文件、软件构件文件、工程设置文件、主程序文件及中断处理程序文件等。下面将结合第一个 S12 汇编工程实例“Light.mcp”讲解上述的文件概念,并详细分析 12 汇编工程的组成、汇编程序文件的编写规范、软/硬件模块的合理划分等。读者若能认真分析与实践第一个汇编实例程序,可以达到由此入门的目的。

### 3.6.1 汇编工程文件的组织

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尤为重要,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。图 3-8 给出了小灯闪烁汇编工程相关源文件的树型结构,主要包括头文件、总体框架文件、软件构件文件、工程设置文件等。

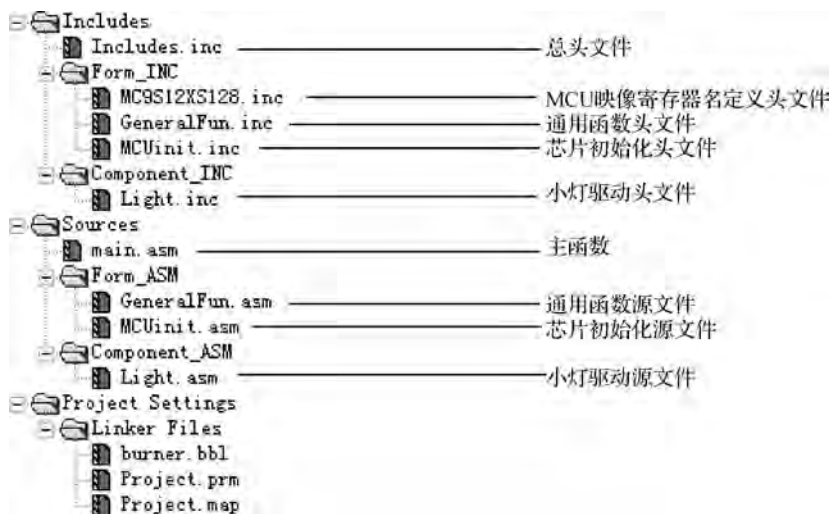


图 3-8 小灯闪烁汇编工程相关源文件的树型结构

#### 1. 主函数文件(main.asm)

一个汇编工程中包含一个汇编主程序文件,文件名固定为 main.asm。汇编主程序的主体是程序的主干流程,要尽可能简洁、清晰,流程中的各个环节由子程序去完成,主程序仅仅是完

成对子程序的调用。

主程序文件 main.asm, 包含有:

① 工程描述: 工程名、硬件连接索引、程序描述、目的、说明、注意、日期等。若调试过程有新的体会, 也可在此添加。目的是将来自己使用, 或同组开发提供必要的备忘信息。

工程名中每个意义单词(或单词缩写)的首字母大写, 后缀为.mcp。

硬件连接索引是工程所要控制的硬件对象索引, 详细描述在相应的硬件对象控制文件中给出。

② 总头文件。

③ 主程序: 主程序一般包括初始化与主循环两大部分。初始化包括堆栈初始化、系统初始化、内存变量初始化、I/O 端口初始化、中断初始化等。主循环是程序的工作循环, 根据实际需要安排程序段, 但一般不宜过长, 建议不要超过 200 行, 具体功能可通过调用子程序来实现, 或由中断程序实现。不带操作系统的 MCU 程序总有一个主循环, 表示程序周而复始地执行。

④ 内部直接调用子程序: 若有不单独存盘的子程序, 建议放在此处。这样在主程序总循环的最后一个语句就可以看到这些子程序。建议不要超过 3 个, 每个子程序不要超过 200 行。若有更多的子程序请单独存盘, 单独测试。

⑤ 外部子程序: 若程序使用独立存盘的子程序, 可在此处使用“INCLUDE 子程序文件名”将其包含。注意, 独立存盘的子程序必须与主程序在同一个目录中。

## 2. 中断处理程序文件(isr.asm)

该文件包含了中断处理程序。中断处理程序是当中断发生后执行的子程序, 中断子程序在返回时需要使用 RTI 指令。中断向量表实质是在 MCU 的向量区写入一些常量值, 这些常量值就是相应的中断处理程序的入口地址。当系统发生中断时, 从中断向量表获得相应的中断处理子程序的入口地址, 使程序流转向中断服务例程, 执行相应的中断处理动作, 处理结束后从中断返回。如果中断向量都为 UNASSIGNED\_ISR 子程序的入口地址, UNASSIGNED\_ISR 子程序是一个空程序, 这表明当发生相应中断时不做任何处理。如果工程中还需要其他中断, 先要在 isr.asm 中定义相应的中断处理子程序, 再将该程序的入口地址置于中断向量表的对应位置即可。中断向量表存放在 isr.asm 文件中如下所示:

```
DC.W UNASSIGNED_ISR    ; 0x08    0xFF10    ivVsi
DC.W UNASSIGNED_ISR    ; 0x09    0xFF12    ivVsyscall
DC.W UNASSIGNED_ISR    ; 0x0A    0xFF14    ivVReserved118
.....                  ;略, 详见网上光盘源代码
DC.W UNASSIGNED_ISR    ; 0x7A    0xFFF4    ivVxirq
DC.W UNASSIGNED_ISR    ; 0x7B    0xFFF6    ivVswi
DC.W UNASSIGNED_ISR    ; 0x7C    0xFFF8    ivVtrap
```

### 3. 框架程序文件

框架程序文件是依赖于具体的芯片型号的,在实际编程过程中,这些文件内容较为固定,一般不改动,或只对个别条目进行修改。

Light.mcp 工程中芯片相关类的程序文件包括:XS128 映像寄存器名定义头文件(MC9S12XS128.inc)、芯片初始化文件(MCUinit.asm)。

#### (1) 芯片头文件 MC9S12XS128.inc

本文件为 MC9S12XS128 MCU 映像寄存器名定义头文件,它是 XS128 的汇编工程的映像寄存器名与其地址的对应表,具体内容参见附录 C.1。本文件和具体 MCU 相关,同时为了程序的通用性,本文件内容不宜更改。实际程序使用“INCLUDE ‘MC9S12XS128.inc’”语句将“MC9S12XS128.inc”文件包含到源程序中。例如,有了这个头文件,对“D 口数据寄存器”读出操作,可用“LDA PORTA”取代“LDA \$0000”,这样更容易理解。

#### (2) 芯片初始化文件 MCUinit.asm

芯片初始化文件主要包含与芯片有关的设置,如总线频率、看门狗模块的使用等。

本实例程序中包含的 MCUinit,是 XS128 系统初始化程序,它将对 SYNRR、REFDV 等寄存器进行设置,并对 PLL 进行编程,由外部晶振  $f = 16\text{ MHz}$ ,得到内部总线时钟  $f_{\text{BUS}} = 32\text{ MHz}$ 。这部分内容对初学者较难,可直接使用。详细技术资料可参看网上光盘之中提供的 XS128 数据手册。

### 4. 汇编软件构件文件

MCU 工程是面向硬件对象的,所以为了复用代码、提高编程效率和增强代码的可移植性,控制小灯闪烁的实例编程中使用了构件化的思想。将每个独立模块分离出来创建一个头文件(如 Light.inc)和一个程序文件(如 Light.asm)。在 Light.inc 头文件中定义小灯控制引脚定义和相关的宏定义,这体现了 MCU 编程的“面向硬件对象”的特性。Light.asm 文件中是小灯的控制实现程序。

### 5. 通用程序文件

通用程序文件中可以定义一些与硬件无关、可以在不同工程项目中直接引用的子程序,如内存数据移动、延时以及数据转换子程序等。它还可以是与这些通用子程序文件相对应的头文件。在工程 Light.mcp 中,有通用子程序文件 GeneralFun.asm,它定义了延时子程序 DelayHX,这个程序不依赖于任何硬件模块,只是完成一个延时动作。如果在另一个工程中需要延时操作时,只要包含 GeneralFun.asm,就可以直接使用已经定义好的延时子程序了。如果通用子程序文件中还需要包含一些头文件或者宏定义,就可以创建一个通用程序头文件 GeneralFun.inc,把这些内容放到这个头文件中(本工程并没有用到 GeneralFun.inc)。

工程 Light.mcp 中还有一个比较特殊的文件——总头文件(Includes.inc),它也属于通用



程序文件。Includes.inc 用于声明全局变量和包含主程序文件中需要的头文件、宏定义等。它使得主函数文件能够尽量避免改动,结构更加清晰。在 Includes.inc 中声明内存变量,通常称为“开辟内存变量”,内存变量的初始化在主程序开始部分完成。第一个内存变量需用“ORG”语句定位,随后,按地址从小到大顺序存放。每个内存变量都有固定的内存地址。借用 C 语言术语,这里所开辟的所有内存变量都是“全局变量”。在 Includes.inc 中也可以进行宏定义。例如,实际应用中,D 口是接一个液晶显示器 LCD 的数据端口,若在总头文件中使用常量定义“LCDDData: equ PTD”,之后 LCDDData 就与 PTD 完全等同,在编程时可以用 LCD-Data 代表 LCD 的数据端口。这样处理不仅符合嵌入式编程的面向对象思想,而且更有利于程序的移植。如想改用 A 口接 LCD 的数据端口,只要在 Includes.inc 中改变相应的常量定义为“LCDDData: equ PTA”就行了,而程序的其他部分无须任何改动。

### (1) 总头文件 Includes.inc

```

;-----*
; 文件名: Includes.inc *
; 说 明: 总头文件,本文件包含: *
;      主函数(main)文件中用到的头文件、外部函数声明 *
;-----*

IFDEF __Includes_INC
__Includes_INC: EQU 2

INCLUDE mc9s12xs128.inc ;MCU 映像寄存器名
INCLUDE MCUinit.inc ;芯片初始化头文件
INCLUDE GeneralFun.inc ;通用子程序头文件
INCLUDE Light.inc ;小灯驱动头文件

ENDIF ; IFDEF __Includes_INC
    
```

### (2) 通用函数文件 GeneralFun.asm

```

;-----*
; 文件名: GeneralFun.asm *
; 说 明: 通用函数文件,包含一些通用函数,可自行添加函数 *
;-----*

;声明外部函数
XDEF DelayHX ;延时子程序

;DelayHX:延时子程序-----*
    
```



```

;功能:用程序的方法,延时约为 HX * 1000 时钟周期(T)                                     *
;入口:HX(0 - 65535)                                                                    *
;出口:无                                                                                *
;堆栈深度:2 + 1 = 3                                                                    *
;说明:                                                                                  *
;    (1)忽略进入与退出部分指令的执行时间                                              *
;    (2)这种延时方法,实际延时的长短与总线周期有关                                  *
;-----*
DelayHX:
    PSHA                                ;[A 进栈](保护寄存器 A)
    CPX #0                             ;X 变址寄存器中的值是否为 0
    BEQ DelayHX_Exit
    ;延时约 HX * 1000(T) -----
DelayHX_1:
    ;延时约 200 * 5 = 1000(T) -----
    LDAA #200
DelayHX_2:
    NOP                                ;(1T)
    NOP                                ;(1T)
    DBNE A,DelayHX_2
    ;-----
    DBNE X,DelayHX_1
    ;-----
DelayHX_Exit:
    PULA                                ;[A 出栈](恢复寄存器 A)
    RTS

```

### 3.6.2 Light 构件汇编程序

Light 构件用于控制指示灯的亮或暗。包括小灯初始化程序 LightInit 和控制小灯亮暗程序 Light\_L 和 Light\_A。

#### (1) Light 构件的汇编头文件(Light.inc)

```

;-----*
; 文件名: Light.inc                                                                    *
; 说 明: 指示灯驱动程序头文件                                                        *
;-----*

IFDEF __Light_INC
__Light_INC: EQU 4

```

```
;声明外部函数
XREF LightInit      ;小灯初始化
XREF Light_L        ;驱动小灯"亮","暗"
XREF Light_A        ;驱动小灯"亮","暗"
```

```
ENDIF      ; IFNDEF __Light_INC
```

## (2) Light 构件的汇编程序文件(Light.asm)

```
;-----*
; 文件名:  Light.asm (小灯驱动函数文件)                                *
;本文件包含:                                                                *
;    (1)Lightinit;定义控制小灯的 MCU 的 I/O 引脚为输出                    *
;    (2)Light_L_A;驱动小灯"亮","暗"                                        *
;硬件连接:                                                                    *
;    (1)本处的小灯是一个发光二极管,由 MCU 的 I/O 引脚控制                *
;    (2)控制引脚为高电平时,小灯"暗";反之,小灯"亮"                        *
;-----*
;小灯驱动所需头文件
INCLUDE mc9s12xs128.inc
;小灯控制引脚宏定义
Light_P:  equ  PORTB      ;灯(Light)接在 PTB 口
Light_D:  equ  DDRB       ;相应的方向寄存器
Light_Pin: equ  1         ;所在的引脚
;声明外部函数
XDEF LightInit      ;小灯初始化
XDEF Light_L        ;驱动小灯"亮","暗"
XDEF Light_A        ;驱动小灯"亮","暗"
;Lightinit;定义控制小灯的 MCU 引脚为输出 -----*
;功能:定义控制小灯的 MCU 引脚为输出,并使小灯初始为暗                    *
;入口:无                                                                *
;出口:无                                                                *
;堆栈深度:2                                                            *
;-----*
LightInit:
    PSHA
    BSET  Light_D,Light_Pin      ;令小灯引脚为输出
    BCLR  Light_P,Light_Pin      ;初始时,小灯"暗"
    PULA
```



RTS

```

;Light_L:驱动小灯"亮"-----*
;功能:控制小灯的亮                                     *
;入口:无                                                 *
;出口:无                                                 *
;堆栈深度:2                                             *
;-----*

```

Light\_L:

PSHA

BSET Light\_P,Light\_Pin

PULA

RTS

```

;Light_A:驱动小灯"暗"-----*
;功能:控制小灯的暗                                     *
;入口:无                                                 *
;出口:无                                                 *
;堆栈深度:2                                             *
;-----*

```

Light\_A:

PSHA

BCLR Light\_P,Light\_Pin

PULA

RTS

### 3.6.3 Light 测试工程主程序

在 includes.inc 文件中需要包含 Light.inc, 这样在该工程中就可以调用 Light 构件的接口函数。首先调用 LightInit 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\_L 和 Light\_A 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, 较明显地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 XS128 系列 MCU 汇编语言程序框架 *
;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研发中心 2011 年-----*

```

；包含派生的特定申明

```
INCLUDE "Includes.inc"
;XREF MCU_init
```

；输出标记

```
XDEF Entry, _Startup, main
；我们用输出"entry"作为标记
；这允许我们以后在 linker.prm 或者 C/C++
；文件中来引用"Entry"
```

```
XREF __SEG_END_SSTACK      ；被连接器定义为栈的终止符
```

；变量/数据区

```
MY_EXTENDED_RAM: SECTION
；在这里插入你要定义的数据
```

；代码区

```
MyCode:      SECTION
main:
_Startup:
Entry:
```

```
SEI
LDS  #__SEG_END_SSTACK      ；初始化堆栈指针
JSR  MCU_init               ；芯片初始化
JSR  LightInit              ；I/O口小灯控制引脚初始化
```

```
mainLoop:
MOVB  #4>>2, $40
JSR  Light_L
LDX  # $0FFF                ；延时
JSR  DelayHX
JSR  Light_A
LDX  # $0FFF                ；延时
JSR  DelayHX
BRA  mainLoop
```

RTS;经过实践我发现子函数返回时的“RTS”前面需要加上空格(至少一个)



### 3.6.4 理解第一个汇编工程的执行过程

当 XS128 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 main.asm 文件中的 \_Startup 函数,进行系统的初始化,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 \*.prm 文件中都可以找到异常向量表,在该表的第一行是复位中断向量地址:

```
VECTOR 0 _Startup      // 复位向量
```

#### (2) 堆栈指针初始化及芯片初始化

执行 main.asm 文件中的 \_Startup 函数,初始化堆栈指针、芯片及所用到的模块。

#### (3) 中断程序的执行

当某个中断发生后,MCU 将转到中断向量表文件 isr.asm 所指定的中断入口地址处开始执行中断服务程序(ISR, Interrupt Service Routine)。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。

下面简单介绍一下 XS128 中断程序的汇编编写方法,这里以 SCI 中断为例,涉及其他模块的中断应用方法会在后续章节之中分别介绍。

在 XS128 之中,其中有些步骤可由芯片内部机制自行完成。当 SCI 采用中断方式收发数据时,需要编写中断处理程序。在 CW 环境下使用 XS128 芯片中断的步骤是:

- ① 在 main.asm 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.asm 文件中,编写中断服务程序,修改中断向量表。

XS128 MCU 开始运行后,要关闭总中断,它就相当于一个总闸门,如果总闸不开,所有中断都不可能发生。这需要汇编指令来实现:

```
SEI      //关总中断  
CLI      //开总中断
```

XS128 的中断编程可概括为下述 3 个步骤:

- ① 新建(或者复制)一个 isr.asm 文件,并加入工程中。
- ② 定义中断向量表(复制 isr.asm 的应修改中断向量表)。

在 XS128 的 Flash 地址空间中,有一段用来存储所有的中断矢量,通常放在最后的 Flash 页面上。该区域每两个字节存储的是一个中断处理函数的地址,各个中断处理函数的地址共

同组成一个逻辑上十分规则的区域——中断向量表。

中断向量表是一个指针数组,内容是中断函数的地址。

首先要定义该中断向量表的地址,XS128 的中断向量从 0xFF10 开始(不同的 MCU 中断矢量起始地址是不相同的,使用时需要查阅相关的技术手册),要使用预编译指令将中断向量表的首地址定义在 0xFF10。

中断向量表格式:

```
isr_default:
    ; 将你中断的代码写在这里
    RTI      ; 结束 isr_default
```

中断向量表内容是从中断向量表起始地址开始顺序增加,均与 Flash 的中断向量地址相对应;如果某个中断不需要使用,要将在数组对应的项中填入 isr\_default。isr\_default 是中断向量表中不需要使用的中断填入的函数,它是一个空函数。

③ 定义 ISR 并在中断向量表中填入相应 ISR 的名称,如中断处理函数文件(isr.asm)之中的函数 SCIO\_ISR 的定义。

通过上述 3 个步骤,就可以定义好所需要的中断了。在实际编程中,可以直接从给定的汇编工程框架中得到“isr.asm”文件;该文件中只定义了一个空中断处理函数 isr\_default 和由这个空函数名组成的中断向量表。用户只须定义所需的中断处理函数,并用该函数名代替向量表中相应位置上的 isr\_default 即可。详细信息见网上光盘提供的示例工程。

中断向量表修改如下:

```
DC.W    UNASSIGNED_ISR    ; 0x08    0xFF10    -    ivVsi
DC.W    UNASSIGNED_ISR    ; 0x09    0xFF12    -    ivVsyscall
DC.W    UNASSIGNED_ISR    ; 0x0A    0xFF14    1    ivVReserved118
.....

DC.W    UNASSIGNED_ISR    ; 0x68    0xFFD0    1    ivVReserved23
DC.W    UNASSIGNED_ISR    ; 0x69    0xFFD2    1    ivVatd0
DC.W    UNASSIGNED_ISR    ; 0x6A    0xFFD4    1    ivVscil
DC.W    SCIO_ISR           ; 0x6B    0xFFD6    1    ivVscio
DC.W    UNASSIGNED_ISR    ; 0x6C    0xFFD8    1    ivVspi0
DC.W    UNASSIGNED_ISR    ; 0x6D    0xFFDA    1    ivVtimpai
DC.W    UNASSIGNED_ISR    ; 0x6E    0xFFDC    1    ivVtimpaaovf
.....

DC.W    UNASSIGNED_ISR    ; 0x7A    0xFFF4    -    ivVxirq
DC.W    UNASSIGNED_ISR    ; 0x7B    0xFFF6    -    ivVswi
DC.W    UNASSIGNED_ISR    ; 0x7C    0xFFF8    -    ivVtrap
```





中断处理函数如下：

SCIO\_ISR:

SEI

PSHA

JSR SCIRel

JSR SCISend1

PULA

CLI

RTI

## 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重用性与可移植性,减少软硬件设计工作中的重复,本章主要知识点有:①讨论嵌入式开发中的硬件、软件的可复用与可移植性问题;②提出嵌入式硬件构件的基本思想,阐述基于硬件构件的嵌入式系统开发方法;③给出基于硬件构件的嵌入式系统硬件电路设计方法与嵌入式底层软件构件的编程方法。网上光盘【第 04 章(构件开发方法)阅读资料】中给出了一个设计参考实例。这一章从方法论角度讨论嵌入式系统设计,第 3 章的例子已经遵循这些基本原则,后续章节的例子将按构件化思想设计注重这些底层软件构件,以提高可复用性。

### 4.1 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到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。

## 4.2 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 3 种类型。核心构件只有提供接口,没有需求接口。也就是说,它只为其他硬件构件提供服务,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他构件提供的服务,而且也能为其他构件提供服务。而终端构件只有需求接口,它只接受其他构件提供的服务。这三种类型构件的区别如表 4-1 所列。

表 4-1 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	芯片的硬件最小系统
中间构件	有	有	电源控制构件、232 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。

## 4.3 基于硬件构件的嵌入式系统硬件电路设计

### 4.3.1 设计时需要考虑的基本问题

设计以 MCU 为核心的嵌入式系统硬件电路根据需求分析进行综合考虑,需要考虑的问题较多,这里给出几个特别要注意的问题。

#### (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、引脚数目以及 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 5 V、3.3 V 以及 1.8 V 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,供以后的设计使用。

#### (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 24 V、12 V、5 V 或者 3.3 V 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

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 Ω 左右,低功耗模式不宜使用。外部上/下拉电阻根据需要可在 10 kΩ~1 MΩ 之间。

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。

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 10 mA,则可用芯片引脚直接控制;电流在 10~100 mA 时可用三极管控制,在 100 mA~1 A 时用 IC 控制;更大的电流则适合用继电器控制,同时建议使用光电隔离芯片。



#### (4) A/D 电路与 D/A 电路

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。

D/A 电路:考虑 MCU 的引脚通过何种输出电路控制实际对象。

#### (5) 控制电路

对外控制电路要注意设计的冗余与反测,要有合适的信号隔离措施等。在评估设计的布板时,一定要在构件的输入输出端引出检测孔,以方便排查错误时测量。

#### (6) 考虑低功耗

低功耗设计并不仅仅是为了省电,更多的好处在于降低了电源模块及散热系统的成本。由于电流的减小也减少了电磁辐射和热噪声的干扰。随着设备温度的降低,器件寿命则相应延长,要做到低功耗一般需要注意以下几点:

① 并不是所有的总线信号都要上拉。上下拉电阻也有功耗问题需要考虑。上下拉电阻拉一个单纯的输入信号,电流也就几十微安以下。但拉一个被驱动了的信号,其电流将达毫安级。所以需要考虑上下拉电阻对系统总功耗的影响。

② 不用的 I/O 口不要悬空,如果悬空,则受外界的一点点干扰就可能成为反复振荡的输入信号,而 MOS 器件的功耗基本取决于门电路的翻转次数。

③ 对一些外围小芯片的功耗也需要考虑。对于内部不太复杂的芯片功耗是很难确定的,它主要由引脚上的电流确定。例如有的芯片引脚在没有负载时,耗电大概不到 1 mA,但负载增大以后功耗可能功耗很大。

#### (7) 考虑低成本

① 正确选择电阻值与电容值。比如一个上拉电阻,可以使用  $4.5\text{ k}\Omega\sim 5.3\text{ k}\Omega$  的电阻,就选个整数  $5\text{ k}\Omega$ ,但实际市场上不存在  $5\text{ k}\Omega$  的阻值,最接近的是  $4.99\text{ k}\Omega$ (精度 1%),其次是  $5.1\text{ k}\Omega$ (精度 5%),其成本分别比精度为 20% 的  $4.7\text{ k}\Omega$  高 4 倍和 2 倍。20% 精度的电阻阻值只有 1、1.5、2.2、3.3、4.7、6.8 几个类别(含 10 的整数倍);类似地,20% 精度的电容也只有以上几种值,如果选了其他的值就必须使用更高的精度,成本就翻了几倍,却不能带来任何好处。

② 指示灯的选择。面板上的指示灯选什么颜色呢?有些人按颜色选,比如自己喜欢蓝色就选蓝色。但是其他红绿黄橙等颜色的不管大小(5 mm 以下)封装如何,都已成熟了几十年,价格一般都在 5 毛钱以下,而蓝色却是近三四年才发明的,技术成熟度和供货稳定度都较差,价格却要贵四五倍。

③ 不要什么都选最好的。在一个高速系统中并不是每一部分都工作在高速状态,而器件速度每提高一个等级,价格差不多要翻倍,另外还给信号完整性问题带来极大的负面影响。

## 4.3.2 硬件构件化电路原理图绘制的简明规则

### (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他构件交互的输入/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,无需任何改动。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他构件提供服务的,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(不用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他构件一起组装系统时,只要考虑其他构件将要连接到核心构件的哪个接口(不用考虑核心构件将要连接到其他构件的哪个接口),这也符合设计人员的思维习惯。

### (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且相对独立的功能,它既要有接受其他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,其他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 4-1 所示)、可变频率产生构件(如图 4-2 所示)是常用的中间构件。图 4-1 中的 Power-IN 和图 4-2 中的 SDI、SCK 和 SEN 均为接口注释,Power-OUT 和 LTC6903-OUT 为接口网标。

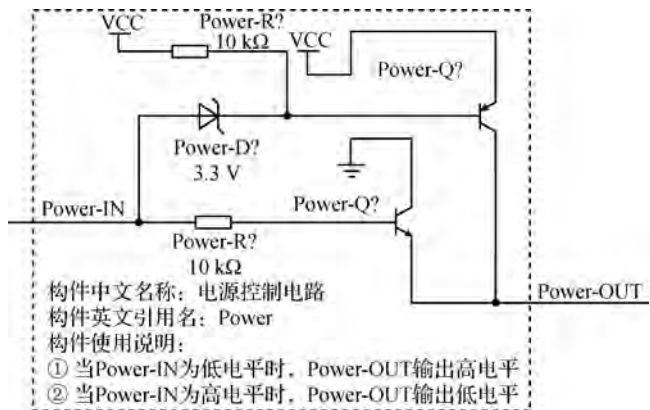


图 4-1 电源控制构件

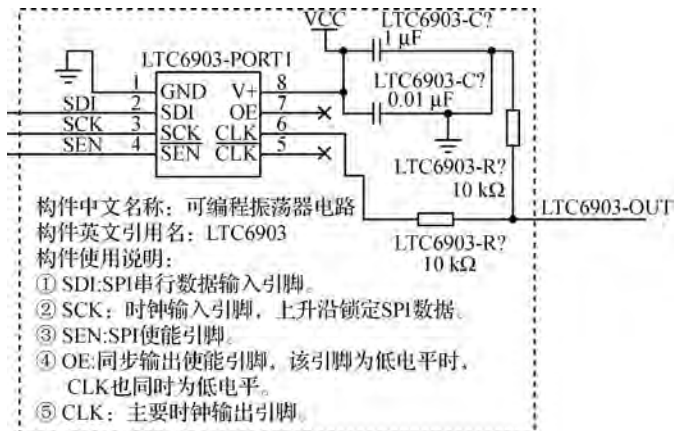


图 4-2 可变频率产生构件



#### (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 4-3 所示)、LED 构件、指示灯构件以及键盘构件(如图 4-4)等都是典型的终端构件。

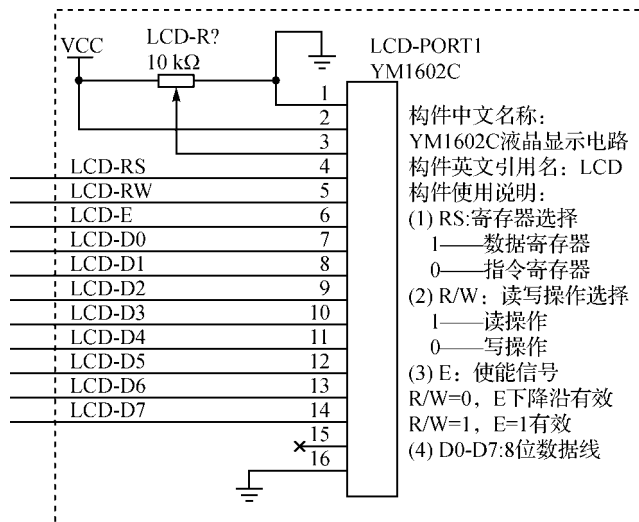


图 4-3 LCD 构件



图 4-4 键盘构件

#### (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,不必做任何改动。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与步骤是:

- 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。

### 4.3.3 实验 PCB 板设计的简明规则

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 4 层、6 层、8 层、12 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 DXP)设计 PCB 时需注意以下几个方面的问题:

### (1) PCB 板布局

在正式走线之前要对 PCB 的大体格局进行规划。布局规划应遵循下列基本原则:

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 元件放置要求整齐,尽可能正放,属于同一硬件构件内的元件尽可能排放在一起。排列方位尽可能与原理图一致,布线方向最好与电路图走线方向一致。元器件在 PCB 上的排列可采用不规则、规则和网格 3 种排列方式中的一种,也可同时采用多种。

不规则排列:元件轴线方向彼此不一致,这对印制导线布设是方便的,且平面利用率高,分布参数小,特别对高频电路有利。

规则排列:元器件轴线方向排列一致,布局美观整齐,但走线较长且复杂,适于低频电路。

网格排列:网格排列中的每一个安装孔均设计在正方形网格交点上。

布局的元器件应有利于发热元器件散热,高频时要考虑元器件之间的分布参数,高、低压之间要隔离,隔离距离与承受的耐压有关。

② 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。

③ 考虑元件间的距离,防止元件之间出现重叠。还要考虑元器件的引脚间距,元器件不同,其引脚间距也不相同,在 PCB 设计中必须弄清楚元器件的引脚间距,因为它决定着焊盘放置间距。对于非标准器件的引脚间距的确定最直接的方法就是:使用游标卡尺进行测量。

④ PCB 四周留有 5~10 mm 空隙不布器件。

⑤ 先放置占用面积较大的元器件,先集成后分立,先主后次,多块集成电路时先放置主电路。

⑥ 可调元件应放置在便于调节的地方,质量超过 15 g 的元器件应当用支架,热敏元件应远离发热元件。晶体应平放,而不要竖直放置。

⑦ 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。导线的最小宽度由导线与绝缘基板间的粘附强度和流过它们的电流值决定。走线时导线尽可能宽,最好不要小于 0.5 mm,手工制板应不小于 0.8 mm,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。电源线和接地线因电流量较大,宽度要大

于普通信号线,一般不要小于 1 mm。三者关系为:地线宽度>电源线宽度>信号线宽度。

② 间距。导线间的间距由它们之间的安全工作电压决定,相邻导线之间的峰值电压、基板的质量、表面涂层、电容耦合参数等都影响导线的安全工作电压,为满足电气安全要求,导线间距离以及导线与元件间距离要尽可能地大,一般不要小于 1 mm,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,且在导线之间最好加接地线。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 $90^\circ$ 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,一般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) 空余位置的利用并标注相关信息

PCB 板的空余位置可适度作如下用途:

- 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 4 个角落。



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:单位、日期、责任人、PCB 板的名称、编号等。

### (9) 抗干扰问题的特别考虑

PCB 设计中除了考虑以上问题外,还要考虑一些隐藏的问题,这些问题设计时不起眼,但是解决的时候却非常麻烦,这就是电路的干扰问题,为此,在 PCB 设计时还应解决如下问题:

#### 1) 热干扰及抑制

元器件在工作中都有一定程度的发热,尤其是功率较大的器件所发出的热量会对周边比较敏感的器件产生干扰,若热干扰得不到很好的抑制,整个电路的电性就会发生变化,最后造成短路。为了对热干扰进行抑制,可采取以下措施:

##### (a) 发热元件的放置

不要贴板放置,也可以单独设计为一个功能单元,放在靠近边缘容易散热的地方。另外,发热量大的器件与小热量的器件应分开放置。

##### (b) 大功率器件的放置

应尽量靠近边缘布置,在垂直方向时应尽量布置在板上方。

##### (c) 温度敏感器件的放置

对温度比较敏感的器件应安置在温度最低的区域,千万不要将它放在发热器件的正上方。

##### (d) 器件的排列与气流

非特定要求下,一般设备内部均以空气自由对流进行散热,故元器件应以纵式排列;若强制散热,元器件可横式排列。另外,为了改善散热效果,可添加与电路原理无关的零部件以引导热量对流。

#### 2) 共阻抗及抑制

共阻干扰是由 PCB 上大量的地线造成,当两个或两个以上的回路共用一段地线时,不同的回路电流在共用地线上产生一定压降,此压降经放大就会影响电路性能,当电流频率很高时,会产生很大的感抗而使电路受到干扰。为了抑制共阻抗,可采取以下措施:

##### (a) 一点接地

使同级单元电路的几个接地点尽量集中,适用于信号的工作频率小于 1 MHz 的低频电路,如果工作频率在 1~10 MHz 而采用一点接地时,其地线长度应不超过波长的 1/20。

##### (b) 就近多点接地

PCB 上有大量公共地线分布在板的边缘,且呈现半封闭回路(防磁场干扰),各级电路采取就近接地,以防地线太长。适用于信号的工作频率大于 10 MHz 的高频电路。

##### (c) 大面积接地

在高频电路中将 PCB 上所有不用面积均布设为地线,以减少地线中的感抗,从而削弱在地线上产生的高频信号,并对电场干扰起到屏蔽作用。

##### (d) 加粗接地线

若接地线很细,接地电位则随电流的变化而变化,致使电子设备的定时信号电平不稳,抗噪声性能变坏,其宽度至少应大于 3 mm。

(e) D/A(数/模)电路的地线分开

两种电路的地线各自独立,然后分别与电源端地线相连,以抑制它们相互干扰。

### 3) 电磁干扰及抑制

电磁干扰是由电磁效应而造成的干扰,由于 PCB 上的元器件及布线越来越密集,如果设计不当就会产生电磁干扰。针对由电源布线、信号布线产生的电磁干扰,可采取不同的措施。

(a) 电源布线引起的电磁干扰

电源布线可采用以下预防措施:

- 布线要宽。
- 加去耦电容。这种电容起到旁路滤波的作用。要在电源的输入端并联较大的和较小的滤波电容。
- 地线环绕。地线要靠近供电电源母线和信号线,因电流沿路径传输会产生回路电感,地线靠近,回路面积减小,电感量减小,回路阻抗减小,从而减小电磁干扰耦合。

(b) 信号布线引起的电磁干扰

信号布线可采用以下预防措施:

- 不同功能的单元电路(如数字电路与模拟电路,高频与低频)分开设置,布线图形应易于信号流通且使信号流向尽可能保持一致。
- 合理使用屏蔽和滤波技术,注意高低压之间的隔离。
- 尽量不选用比实际需要的速度更快的元件,在元件的位置安排上,易受电磁干扰的元器件不能相距太近,应大于信号波长的  $1/4$ ,输入器件与输出器件尽量远离。
- 做到安全接地。

## 4.4 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。

### 4.4.1 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 I/O)口和一些内置功能模块,可将通用 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,如芯片内含模块的功能构件有串行通信构件、Flash 构件、定时器构件等。

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 4-5 所示。

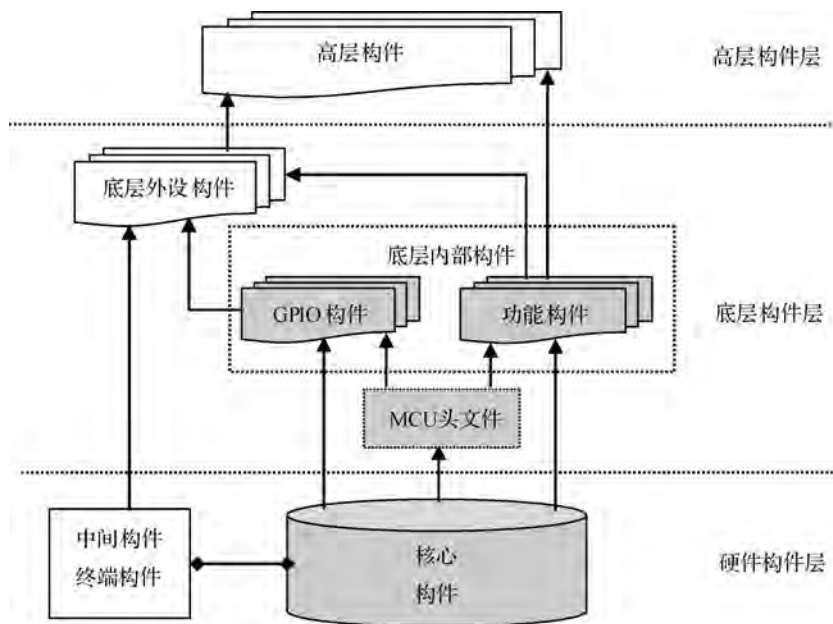


图 4-5 嵌入式硬件构件与软件构件的层次模型

由图 4-5 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 I2C 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 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,以便其他构件头文件中包含该头文件。

#### 4.4.2 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。**尽量做到:当一个底层构件应用到不同系统中时,仅须修改构件的头文件,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。有关内容参见第 5 章。

在编写构件时,一般应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,在其他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他 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他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。

#### 4.4.3 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,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)与芯片 AW60(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 4-6(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 4-6(b)所示。

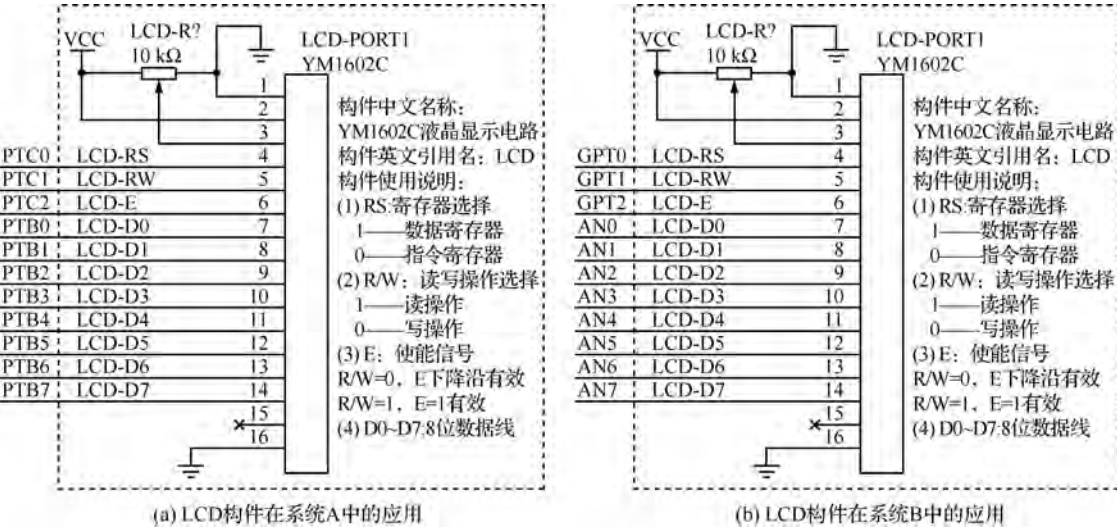


图 4-6 LCD 构件在实际系统中的应用

## 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。

而对于外接硬件构件,希望不改动程序文件,而只改动头文件,那么,头文件就必须充分设计。以 LCD 构件为例,与图 4-6(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 4-6(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,而不是不做任何改动。希望改动在头文件中进行,而不希望改动程序文件,事实上,不做任何改动是不现实的。

# 第 5 章

## 串行通信接口 SCI

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 和 COM2,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 3 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。

本章主要知识点有:①异步串行通信的通用基础知识;②XS128 的 SCI 模块的编程结构与 SCI 构件设计,并测试这个构件;③借助 XS128 的串行接收中断来阐述中断概念与编程方法,并给出编程实例。基于构件的编程思想,再次在设计 SCI 构件中加以阐述,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。通过本章学习中断编程后,以一个 MCU 为蓝本学习嵌入式系统硬件软件基础知识的基本要素已经完成,后续章节将进行知识的不断积累与扩展,学习新模块的工作原理与驱动软件设计,但基本思想与方法方法是一致的。本章重点是理解 SCI 构件设计方法、理解第一个中断例程的执行过程。

### 5.1 异步串行通信的通用基础知识

本节简要概述异步串行通信中常用的基本概念与硬件连接方法,为学习 MCU 的串行通信接口编程做准备。

### 5.1.1 串行通信的基本概念

“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,本小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 5-1 给出了 8 位数据、无校验情况的传送格式。

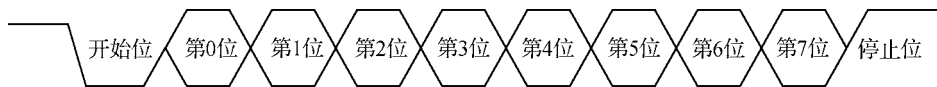


图 5-1 串行通信数据格式

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 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、1 200、1 800、2 400、4 800、9 600、19 200、38 400、57 600、



115 200、128 000 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 1 KB 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,以至于很容易受到电磁源的干扰,通信就不可靠。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。

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。

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。

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。

## 5.1.2 RS-232 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 2.4 V 和 0.4 V(目前使用 3 V 供电的 MCU 中,该特征值有所变动),即大于 2.4 V 则识别为“1”,小于 0.4 V 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 5 m 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,以下简称 RS-232。RS-232 采用负逻辑,-15~-3 V 为逻辑“1”,+3~+15 V 为逻辑“0”。RS-232 最大的传输距离是 30 m,通信速率一般低于 20 kbps。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 300 m 之外,这是很少见的,且稳定性很不好。

RS-232 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-232 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 1~2 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-232 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,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 5-2 给出了 9 芯串行接口的排列位置,相应引脚含义见表 5-1。



图 5-2 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(如请求发送 RTS 信号与允许发送 CTS 信号),初学时可以忽略这些信号的含义。

表 5-1 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)		



### 5.1.3 TTL 电平到 RS-232 电平转换电路

在 MCU 中,若用 RS-232 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-232 电平;在接收端,需要用接收电路将 RS-232 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。目前广泛使用 MAX232 芯片较多,该芯片使用单一的+5 V 电源供电实现电平转换。图 5-3 给出了 MAX232 的引脚说明。

引脚含义简要说明如下:

V<sub>CC</sub>(16 脚):正电源端,一般接+5 V。

GND(15 脚):地。

VS+(2 脚):VS+=2V<sub>CC</sub>-1.5 V=8.5 V。

VS-(6 脚):VS-=-2V<sub>CC</sub>-1.5 V=-11.5 V。

C2+、C2-(4、5 脚):一般接 1 μF 的电解电容

C1+、C1-(1、3 脚):一般接 1 μF 的电解电容。

输入输出引脚分两组,基本含义见表 5-2。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。

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

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 5-4 给出了基本串行通信接口的电平转换电路。

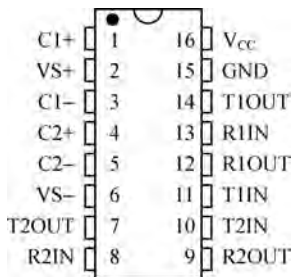


图 5-3 MAX232 引脚图

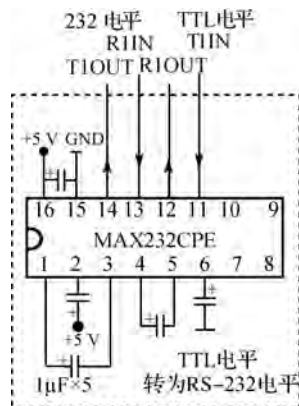


图 5-4 串行通信接口电平转换电路



表 5-2 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(T1IN)	输入	接 MCU 的 TXD	13	输入	接到 9 芯接口的 2 脚 RXD
	12(R1OUT)	输出	接 MCU 的 RXD	14	输出	接到 9 芯接口的 3 脚 TXD
2	10(T2IN)	输入	接 MCU 的 TXD	8	输入	接到 9 芯接口的 2 脚 RXD
	9(R2OUT)	输出	接 MCU 的 RXD	7	输出	接到 9 芯接口的 3 脚 TXD

MAX232 芯片进行电平转换基本原理是:

**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。

## 5.1.4 串行通信编程模型

从基本原理角度看,串行通信接口 SCI 的主要功能是:接收时,把外部的单线输入的数据变成一个字节的数据送入 MCU 内部;发送时,把需要发送的一个字节的数据转换为单线输出。图 5-5 给出了一般 MCU 的 SCI 模块的功能描述。为了设置波特率 SCI 应具有波特率寄存器。为了能够设置通信格式、是否校验、是否允许中断等,SCI 应具有控制寄存器。而要知道串口是否有数据可收、数据是否发送出去等,需要有 SCI 状态寄存器。当然,若一个寄存器不够用,控制与状态寄存器可能有多个。而 SCI 数据寄存器存放要发送的数据,也存放接收的数据,这并不冲突,因为发送与接收的实际工作是通过“发送移位寄存器”和“接收移位寄存器”完成的。编程时,程序员并不直接与“发送移位寄存器”和“接收移位寄存器”打交道,只与数据寄存器打交道,所以 MCU 中并没有设置“发送移位寄存器”和“接收移位寄存器”的映像地址。发送时,程序员通过判定状态寄存器的相应位,了解是否可以发送一个新的数据。若可以发送,则将待发送的数据放入“SCI 数据寄存器”中就可以了,剩下的工作由 MCU 自动完成:将数据从“SCI 数据寄存器”送到“发送移位寄存器”,硬件驱动将“发送移位寄存器”的数据一位一位地按照规定的波特率移到发送引脚 TxD,供对方接收。接收时,数据一位一位地从接收引脚 RxD 进入“接收移位寄存器”,当收到一个完整字节时,MCU 会自动将数据送入“SCI 数据寄存器”,并将状态寄存器的相应位改变,供程序员判定并取出数据。

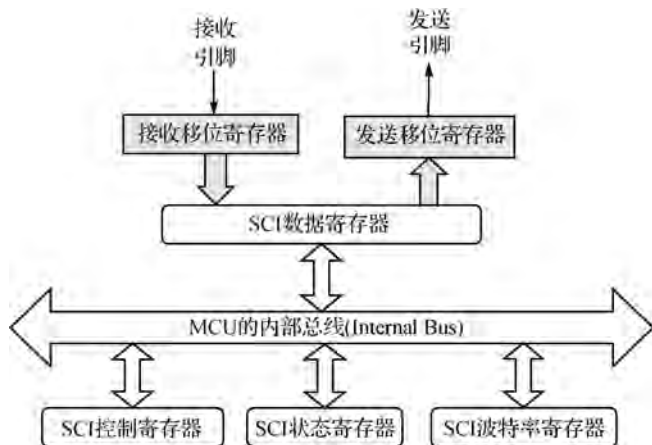


图 5-5 SCI 编程模型

## 5.2 SCI 模块的编程寄存器

本节从编程角度介绍 MC9S12XS128 MCU 的 SCI 模块。该芯片有两个串行口 SCI0 和 SCI1。从外部引脚来看,负责串行通信的是 PS0/RxD0、PS1/TxD0、PS2/RxD1、PS3/TxD1 这 4 个引脚。当允许 SCI 时,这些引脚作为串行通信引脚;每个串口模块包含 2 个引脚,分别称为串行发送引脚 TxD 和串行接收引脚 RxD。本章以 SCI0 为例来介绍 SCI 模块,SCI1 的使用方法与 SCI0 完全相同,只是相关寄存器映像地址不同,以下统称 SCI 模块,具体使用时在头文件中区分使用哪个串口。

从程序员角度看,SCI 模块有 11 个 8 位寄存器,表 5-3 列出了主要寄存器的基本信息。只要理解和掌握这 8 个寄存器的用法,就可以进行 SCI 编程。

下面从编程角度介绍 SCI 模块的寄存器(波特率寄存器、控制寄存器、状态寄存器和数据寄存器)。

表 5-3 XS128 的 SCI 寄存器

地 址		寄存器名称与缩写	访问权限	基本功能
SCI0	SCI1			
\$ 00C8	\$ 00D0	波特率寄存器(SCIxBDH、SCIxBDL)	读/写	设置波特率
\$ 00C9	\$ 00D1			
\$ 00CA	\$ 00D2	控制寄存器(SCIxCR1、SCIxCR2)	读/写	设置传输格式、中断使能
\$ 00CB	\$ 00D3			

续表 5-3

地 址		寄存器名称与缩写	访问权限	基本功能
SCI0	SCI1			
\$ 00CC	\$ 00D4	状态寄存器 SCIxSR1	只读	中断标志、发送与接收状态
\$ 00CD	\$ 00D5	状态寄存器 SCIxSR2	读/写	
\$ 00CE	\$ 00D6	数据寄存器(SCIxDRLH、SCIxDRL)	读/写	收发的数据
\$ 00CF	\$ 00D7			

注：由于有两个串行口，使用 SCIx 时，寄存器名称中 SCI 改为 SCIx，x=0、1。

### 1. SCI 波特率寄存器(SCI Baud Rate Register, SCIxBDH/L)

波特率寄存器[SCIxBDH:SCIxBDL]用来设置 SCI 模块的通信波特率，分为波特率寄存器的高 8 位与低 8 位，如下所示：

SCIxBDH：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IREN	TNP1	TNP0	SBR12	SBR11	SBR10	SBR9	SBR8
复位	0	0	0	0	0	0	0	0

SCIxBDL：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
复位	0	0	0	0	0	1	0	0

如果只写波特率的高位寄存器不写低位寄存器，波特率将不能被正确设置。波特率由 MCU 内部总线时钟 BUSCLK 分频而来，SCI 模块需要的工作时钟频率为波特率的 16 倍。波特率高位寄存器加低位寄存器共 16 位，只有低 13 位为波特率分频因子，这个 13 位数 SBR 可表示的值为 0~8191，其中 SBR[12：0]=0 表示 SCI 停止工作，以便节省电能，为非 0 值时是分频因子。IREN 的值影响波特率的计算，在 IREN 取不同值的情况下，可以分别用以下公式计算波特率的值：

$$\text{IREN}=0 \text{ 时,SCI 波特率}=\text{BUSCLK}/(16\times\text{BR}[12:0])$$

$$\text{IREN}=1 \text{ 时,SCI 波特率}=\text{BUSCLK}/(32\times\text{BR}[12:0])$$

一般情况下，内部总线频率 BUSCLK( $f_{\text{BUS}}$ )是已知的，SCI 波特率(Bt)是编程者希望设定的，然后根据 BUSCLK( $f_{\text{BUS}}$ )、SCI 波特率(Bt)波特率求出 BR[12：0]的值：

$$\text{BR}[12:0]=f_{\text{BUS}}/(16\times\text{Bt}), \quad \text{IREN}=0 \text{ 时}$$

$$\text{BR}[12:0]=f_{\text{BUS}}/(32\times\text{Bt}), \quad \text{IREN}=1 \text{ 时}$$



例如,当  $IREN=0$  时,  $f_{BUS}=19.6608\text{ MHz}=19660800\text{ Hz}$ , 希望设定波特率  $B_t=9600$ , 以求得  $BR=19660800/(16\times 9600)=128$ , 转为十六进制为  $0x80$ , 则  $SCIBRH=0x00$ ,  $SCID-BL=0x80$ 。要先写  $SCIBRH$  再写  $SCIBRL$ 。若是使用  $SCI0$ , 则程序如下:

```
SCI0BDH = 0x00;    //先给高位赋值
SCI0BDL = 0x80;    //后给低位赋值
```

**注意:**若  $f_{BUS}=25\text{ MHz}$ , 则  $BR=25\,000\,000/(16\times 9600)=162.76\approx 163$ , 由于不能整除, 所以有时存在一定误差, 这是允许的。

该寄存器的其他位定义如下:

**IREN:** 红外调制解调子模块使能位。S12XS 系列 MCU 的 SCI 模块支持红外调制、解调功能, 通过设置该位, 可以使能或禁止红外调制解调子模块。 $=1$  红外调制解调子模块使能;  $=0$ , 红外调制解调子模块禁止。

**TNP1、TNP0:** 这两位与红外功能有关, 用来选择 SCI 发送的窄脉冲的宽度, 红外调制解调时, 窄脉冲对应位为 0, 无窄脉冲对应位为 1。当  $TNP1$ 、 $TNP0=11$ 、 $10$ 、 $01$ 、 $00$  时, 窄脉冲的宽度分别为位宽度的  $1/4$ 、 $1/32$ 、 $1/16$ 、 $3/16$ 。

在使用 SCI 模块时, 一般不需要使用红外功能, 保持 IREN 位的默认值即可, 并按第一个公式计算波特率的数值。如果使用红外功能, 除了将 IREN 位置 1, 还要求 SCI 模块的外引脚  $TxD$ 、 $RxD$  接外部红外收发器。

## 2. SCI 控制寄存器

SCI 控制寄存器有两个, 分别是 SCI 控制寄存器 1、SCI 控制寄存器 2。

### (1) SCI 控制寄存器 1(SCI Control Register1, SCICR1)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	LOOPS	SCISWAI	RSRCPT	M	WAKE	ILT	PE	PT
复位	0	0	0	0	0	0	0	0

**D7—LOOPS 位,** 循环模式选择位 (Loop Select Bit)。LOOPS=1, 选择循环模式; LOOPS=0, 正常模式。只有当发送和接收同时允许时, 才能使用循环模式。在循环模式下,  $RxD$  引脚与 SCI 系统断开。发送器的输出直接接到接收器的输入。

**D6—SCISWAI 位,** 等待模式下, 如果该位置 1, 在等待模式下 SCI 模块禁止; 该位清 0, 在等待模式下 SCI 仍然正常工作。

**D5—RSRC 位,** 接收器资源位 (Receiver Source Bit)。该位仅在 LOOPS=1 的模式下有意义。RSRC=1,  $RxD$  和  $TxD$  引脚短接; RSRC=0, 内部短接, 不经过外部引脚。

**D4—M 位,** 字符长度选择位。定义收发数据格式,  $M=1$ , 9 位数据传送;  $M=0$ , 8 位数据传送。

D3—WAKE 位,唤醒条件选择位。WAKE=1,地址唤醒;WAKE=0,空闲线唤醒。

D2—ILT 位,空闲线方式选择位,ILTY=1,空闲字符位从“停止位”开始计数;ILTY=0,空闲字符位从“起始位”开始计数。

D1—PE 位,奇偶校验允许位。PE=1,允许奇偶校验;PE=0,不进行奇偶校验。

D0—PT 位,奇偶校验类型选择位。PT=1,奇校验;PT=0,偶校验。不进行奇偶校验时,该位无意义,一般写 0。

## (2) SCI 控制寄存器 2(SCI Control Register2, SCICR2)

该寄存器是 SCI 子系统的主控制寄存器。这个寄存器可以允许/禁止发送器或接收器,允许/禁止系统中断、唤醒功能和终止码功能。TIE、TCIE、RIE、ILIE 位是局部中断控制,决定 SCI 系统是否发出硬件中断请求。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
复位	0	0	0	0	0	0	0	0

D7—TIE 位,使能发送数据寄存器空标志(TDRE),产生中断请求。SCTIE=1,使能 TDRE 中断请求;SCTIE=0,禁用 TDRE 中断请求。

D6—TCIE 位,发送完成中断允许位。TCIE=1,允许发送完成产生中断;反之不允许。

D5—RIE 位,接收中断允许位。SCRIE=1,允许产生接收中断请求;反之不允许。

D4—ILIE 位,空闲线中断允许位。ILIE=1,允许产生空闲中断请求;反之不允许。

D3—TE 位,发送器允许位。TE=1,允许发送器发送;反之不允许发送器发送。

D2—RE 位,接收器允许位。RE=1,允许接收器接收;反之不允许接收器接收。

D1—RWU 位,接收器唤醒位。RWU=1,接收器处于等待状态,并关闭接收中断,正常情况下硬件自动清零该位而唤醒接收器;RWU=0,正常操作。

D0—SBK 位,发送终止位。SBK=1,发送终止码;SBK=0,发送非终止码。

在一般的系统中,TE 和 RE 写入 1 以允许 SCI 子系统的接收器和发送器工作。而 ILIE、RWU 和 SBK 不常用,写入 0 以禁止这些功能。如果不准备采用中断处理 SCI,应向 TIE、TCIE 和 RIE 写入 0,否则要写入 1。例如,一个系统不用中断,就要在初始化时向 SCIxCR2 装入 \$0C。

## 3. SCI 状态寄存器

SCI 状态寄存器有两个,分别是 SCI 状态寄存器 1、SCI 状态寄存器 2。

### (1) SCI 状态寄存器 1(SCI Status Register, SCISR1)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TDRE	TCPE	RDRF	IDLE	OR	NF	FE	PF
复位	1	1	0	0	00	0	0	0



TDRE 标志位与 RDRF 标志位比较常用,而 TC 位和 IDLE 位一般不用。是否使用 OR、FE 和 NF 位,应根据系统类型的不同要求来决定。

SCI 通信连接方式主要有两种。其中之一是直接连接,如 MCU 和 PC 之间的连接。在这种方式下,OR、NF 和 FE 之类的错误未必会真发生,因此典型做法是忽略掉。第二种类型是两个远地设备通过 MODEM 连接。在这种连接下,很有可能发生错误,所以有必要在两者之间采用某种协议,允许在发生错误时再发送一遍数据。

D7—TDRE 位,发送数据寄存器空标志位。该位为 1 时表明要发送的数据已经移入发送移位寄存器,数据寄存器为空。

D6—TC 位,发送完成标志位。该位为 1,表明发送已经完成;该位为 0,表明发送正在进行。

D5—RDRF 位,接收数据寄存器为满标志位。该位为 1,表明接收器已满,可以从 SCI 数据寄存器 SCIDRH/SCIDRL 中读取收到的数据;为 0,表示 SCI 数据寄存器中的数据无效。

D4—IDLE 位,空闲标志位。当接收 10 个连续的逻辑 1( $M=0$ )或接收 11 个连续的逻辑 1( $M=1$ )时,IDLE 位即被设置为 1。IDLE=1 表明接收器处于空闲状态。注意当接收唤醒位 RWU=1 时,空闲线的状态不能影响 IDLE 位。

D3—OR 位,接收器溢出标志位。该位为 1,表明接收器溢出。

D2—NF 位,接收器噪声标志位。该位为 1,表明接收器出现噪声错误。

D1—FE 位,接收器帧错误标志位。该位为 1,表明接收器出现帧错误。

D0—PF 位,接收器奇偶错误标志位。该位为 1,表明接收器出现奇偶校验错误。该位写无效。

## (2) SCI 状态寄存器 2(SCI Status Register, SCISR2)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	AMAP	0	0	TXPOL	RXPOL	BRK13	TXDIR	RAF
复位	0	0	0	0	0	0	0	0

D7—AMAP 位,寄存器访问选择位。S12X 系列 MCU 的 SCI 为增强型 SCI,增加了 SCIxASR1、SCIxACR1 和 SCIxACR2 这 3 个寄存器,这 3 个寄存器分别与 SCIxBDH、SCIxBDL 和 SCIxCR1 这 3 个寄存器共享相同的地址。这 3 个寄存器与 SCI 基本功能无关,很少被用到,这里不做介绍。当 AMAP 为默认值 0 时,可以访问 SCIxBDH、SCIxBDL 和 SCIxCR1 寄存器,当 AMAP 为默认值 1 时,可以访问 SCIxASR1、SCIxACR1 和 SCIxACR2 寄存器,SCIxBDH、SCIxBDL 和 SCIxCR1 则被隐藏。

D6~D5,未定义。

D4—TXPOL,发送极性选择位。TXPOL=0,正常极性。TXPOL=1,反向极性。在正常模式 NRZ 格式下,1 表示高,0 表示低。在反向极性下刚好相反。

D3—RXPOL,接收极性选择位。RXPOL=0,正常极性。RXPOL=0,反向极性。

D2—BRK13 位,终止码长度选择位。BK13=1,终止码长度为 13 或 14 位;BK13=0,终止码长度为 10 或 11 位。帧错误不受该位的影响。

D1—TXDIR 位,单线模式下发送引脚的数据传输方向位。在单线模式下,只使用 TxD 引脚发送接收数据。该位用来控制 TxD 外引脚用于发送数据还是接收数据。TXDIR=1, TxD 用于发送数据;TXDIR=0, TxD 用于接收数据。所谓单线模式就是 RxD 与 TxD 相接,只使用 TxD 发送和接收数据,并且由 TXDIR 位决定是发送还是接收。

D0—RAF 位,正在接收标志位。该位为 1,表明正在接收。

4. SCI 数据寄存器(SCI Data Register, SCIDRH /L)

SCI 数据寄存器由两个 8 位寄存器构成,分别为 SCIDRH 与 SCIDRL,如下:

SCIDRH:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读操作	R8	T8	0	0	0	0	0	0
写操作	—		—	—	—	—	—	—
复位	0	0	0	0	0	0	0	0

SCIDRL:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读操作	R7	R6	R5	R4	R3	R2	R1	R0
写操作	T7	T6	T5	T4	T3	T2	T1	T0
复位	0	0	0	0	0	0	0	0

当使用 8 位数据格式时,只使用 SCIDRL 寄存器。当使用 9 位数据格式时,两个寄存器都要用,发送时,先写 SCIDRH 寄存器,再写 SCIDRL 寄存器。

SCIDRH:只有 D7、D6 被定义,分别记为 R8、T8。当 SCI 配置为 9 位数据格式(M=1)时,R8 为收到的第 9 位数据,T8 则存放要发送的第 9 位数据。

SCIDRL:读出时,为接收的数据,记作 R7~R0;写入时,为要发送的数据,记作 T7~T0。

5.3 SCI 编程实例

无论用查询方式还是中断方式进行串行通信编程,在程序初始化时均必须对 SCI 进行初始化,主要进行波特率设置、通信格式的设置、发送接收数据方式的设置等。本节给出最基本的方法,作为 S12X 系列 MCU 的串行通信编程入门知识。下节将给出规范的串行通信编程子程序,读者可以直接将其应用于实际的嵌入式应用系统的开发中。





## 5.3.1 SCI 初始化与收发编程的基本方法

### 1. SCI 初始化基本方法

有关寄存器地址的定义已经在头文件 MC9S12XS128.h 中给出, SCIBDH、SCIBDL、SCI-CR1、SCICR2、SCISR1、SCISR2、SCISDRH、SCISDRL 等接口可以直接使用。

对 SCI 进行初始化,最少由以下 3 步构成(以 SCI0 为例):

① 定义波特率。一般选择内部总线时钟为串行通信的时钟源。设 MCU 系统初始化时已定义总线频率为  $f_{\text{BUS}}$ ,同时准备定义串行通信波特率记为  $B_t$ ,可通过设置 SCI 波特率寄存器 SCI0BD 的波特率选择位 SBR[12:0](13 位)以便选择合适的分频系数,在 5.2 节已叙述过,这里不再赘述。

② 写控制字到 SCI 控制寄存器 1(SCI0CR1)。设置是否允许 SCI、数据长度、输出格式、选择唤醒方法、是否校验等。如若设定允许 SCI、正常码输出、8 位数据、无校验,则 SCI0CR1 取值应为二进制“00000000”,程序如下。

```
//设置允许 SCI,正常码输出、8 位数据、无校验  
SCI0CR1 = 0x00;
```

实际上,这种情况只要取 SCI0CR1 各位的默认值即可。

③ 写控制字到 SCI 控制寄存器 2(SCI0CR2)。设置是否允许发送与接收、是中断接收还是查询接收等。如若设置允许发送( $\text{TE}=1$ )、允许接收( $\text{RE}=1$ )、查询方式收发,则 SCI0CR2 取值应为二进制“00001100”,程序如下:

```
//设置允许发送、允许接收,查询方式收发  
SCI0CR2 = 0x0C;
```

另外几个寄存器供后面编程使用,不需初始化。

### 2. 发送与接收一个字节数据的编程方法

一般情况下,对 SCI 的初始化仅在程序的初始化部分进行一次即可,而串行通信的基本用途就是编程来发送与接收数据。发送数据是通过判断状态寄存器 SCI0SR1 的第 7 位(TDRE)进行的,而接收数据是通过判断状态寄存器 SCI0SR1 的第 5 位(RDRF)进行的。不论是发送还是接收,均会使用 SCI 数据寄存器 SCI0DRL。发送时将要发送的数据送入 SCI0DRL 即可,接收时从 SCI0DRL 中取出的即是收到的数据。

例如,下面的程序将字节型变量 i 中的数从串行引脚 TxD 发送出去。通过对状态寄存器 SCI0SR1 的第 7 位(TDRE)判断是否可以向数据寄存器 SCI0DRL 送数,若  $\text{TDRE}=1$  可以送数,否则必须等待。

```
//判断 SCI0SR1 的第 7 位(TDRE)是否为 1,是 1 则可以发送数据 Data
while(1)
    if ((SCI0SR1 &(1<<7)) != 0) // SCI0SR1.7 是否为 0,为 0 则等待
    {
        //为 1,可以发送数据
        SCI0DRL = i;
        break;
    }
```

要以查询方式接收一个数据,首先通过状态寄存器 SCI0SR1 的第 5 位(RDRF)判断有没有数据可收,若 RDRF=1 则有数据可收,下面程序持续等待串行口(实际上是 RxD 引脚)接收一个字节数据。

```
//查询方式接收一个字节的的数据放入字节型变量 i 中:
while(1)
    if ((SCI0SR1 &(1<<5)) != 0) // SCI0SR1.5 是否为 0,为 0 则等待
    {
        //为 1,可以取出数据
        i = SCI0DRL;
        break;
    }
```

在实际编写串行通信接收子函数时,不采用永久循环形式,而改用测试一段时间,若无数据可接收,则带错误标志返回。

## 5.3.2 SCI 构件设计与测试实例

### 1. SCI 构件设计

SCI 具有初始化、接收和发送 3 种基本操作。按照构件的思想,可将它们封装成几个独立的功能函数,初始化函数完成对 SCI 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 SCI 模块进行编程,实际上已经涉及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 3 种基本操作所对应的功能函数共同放置在命名为 SCI.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 SCI.h 的头文件,用来定义模块的基本信息和对外接口。

#### (1) SCI 构件的头文件(SCI.h)

头文件 SCI.h 中的内容可分为两个主要的部分,它们分别是 6 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 SCI 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 SCI.h 含有串行通信寄存器、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。



```
// ----- *
// 文件名: SCI.h *
// 说 明: SCI 构件头文件 *
// ----- *

#ifndef SCI_H
#define SCI_H

//头文件包含,及宏定义区
//头文件包含
#include "Includes.h"
//开放或禁止 SCI 的接收中断的宏定义
#define EnableSCIReInt0          SCI0CR2 |= 0x20    //开放 SCI0 接收中断
#define DisableSCIReInt0        SCI0CR2 &= 0xDF    //禁止 SCI0 接收中断
#define EnableSCIReInt1          SCI1CR2 |= 0x20    //开放 SCI1 接收中断
#define DisableSCIReInt1        SCI1CR2 &= 0xDF    //禁止 SCI1 接收中断
//串行通信寄存器及标志位定义
#define ReSendStatusR0          SCI0SR1            //SCI0 状态寄存器
#define ReSendDataR0            SCI0DRL            //SCI0 数据寄存器
#define ReSendStatusR1          SCI1SR1            //SCI1 状态寄存器
#define ReSendDataR1            SCI1DRL            //数据寄存器
#define ReTestBit                5                  //接收缓冲区满标志位
#define SendTestBit              7                  //发送缓冲区空标志位
//构件函数声明区
void SCIInit(uint8 SCINo, uint8 sysclk, uint16 baud); //初始化 SCIx 模块。x 代表 0,1
void SCISend1(uint8 SCINo, uint8 ch); //串行发送 1 个字节
void SCISendN(uint8 SCINo, uint8 n, uint8 ch[]); //串行发送 N 个字节
uint8 SCIRe1(uint8 SCINo, uint8 * p); //从串口接收 1 个字节的数据
uint8 SCIReN(uint8 SCINo, uint8 n, uint8 ch[]); //从串口接收 N 个字节的数据
void SCISendString(uint8 SCINo, char * p); //串口传输字符串
#endif
```

以初始化函数 SCIInit 为例,通道号、当前的系统时钟、希望实现的通信波特率都被设计为函数参数。这样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。

## (2) SCI 构件的程序文件(SCI.c)

```
// ----- *
//文件名: SCI.c *
//说 明: SCI 构件函数源文件 *
// ----- *

//头文件包含,及宏定义区
```

```
//头文件包含
#include "SCI.h"
//构件函数实现
//-----*
//函数名: SCIInit *
//功 能: 初始化 SCIx 模块。x 代表 0,1 *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块。其中 SCINo 取值为 0,1 *
//          如果 SCINo 大于 1,则认为是 1 *
//          uint8 sysclk: 系统总线时钟,以 MHz 为单位 *
//          uint16 baud: 波特率,如 4800,9600,19200,3840 *
//返 回: 无 *
//说 明: SCINo = 0 表示使用 SCI0 模块,依此类推。 *
//-----*
void SCIInit(uint8 SCINo, uint8 sysclk, uint16 baud)
{
    uint8 t;
    uint16 ubgs = 0;
    if(SCINo > 1) SCINo = 1; //串口号错误输入处理
    //计算波特率并设置:ubgs = fsys/(波特率 * 16)(其中 fsys = sysclk * 1000000)
    ubgs = sysclk * (10000/(baud/100))/16; //理解参考上一行,此处是为了运算
    //初始化相应寄存器

    switch (SCINo)
    {
        case 0:
            //设置波特率
            SCI0BDH = (uint8)((ubgs & 0xFF00) >> 8); //给高 8 位赋值
            SCI0BDL = (uint8)(ubgs & 0x00FF); //给低 8 位赋值

            SCI0CR1 = 0x00; //设置允许 SCI,正常码输出,8 位数据,无校验
            t = SCI0DRL; //读数据寄存器(清 0)
            t = SCI0SR1; //读状态寄存器(清 0)
            SCI0CR2 = 0x0C; //允许 SCI0 接收和发送 查询方式
            break;

        case 1:
            //设置波特率

            SCI1BDH = (uint8)((ubgs & 0xFF00) >> 8); //给高 8 位赋值
            SCI1BDL = (uint8)(ubgs & 0x00FF); //给低 8 位赋值
    }
}
```



```

        SCI1CR1 = 0x00;          //设置允许 SCI,正常码输出,8 位数据,无校验
        t = SCI1DRL;              //读数据寄存器(清 0)
        t = SCI1SR1;              //读状态寄存器(清 0)
        SCI1CR2 = 0x0C;          //允许 SCI0 接收和发送 查询方式
        break;
    }
}

//-----*
//函数名: SCISend1                      *
//功 能: 串行发送 1 个字节                *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块,其中 SCINo 取值为 0,1      *
//      uint8 ch: 要发送的字节                *
//返 回: 无                                *
//说 明: SCINo = 0 表示使用 SCI0 模块,依此类推                *
//-----*
void SCISend1(uint8 SCINo, uint8 ch)
{
    uint16 k;
    if(SCINo > 1)
    {
        SCINo = 1;                //若传进的通道号大于 2,则按照 2 来接收
    }

    //根据通道号发送数据

    switch (SCINo)
    {
        case 0:
            for (k = 0; k < 0xfbbb; k++) //有时间限制
            {
                //判断发送缓冲区是否为空
                if ((ReSendStatusR0 & (1<<SendTestBit)) != 0)
                {
                    ReSendDataR0 = ch;
                    break;
                }
            }
            break;

        case 1:
            for (k = 0; k < 0xfbbb; k++) //有时间限制
            {

```

//判断发送缓冲区是否为空

```

if ((ReSendStatusR1 & (1<<SendTestBit)) != 0)
{
    ReSendDataR1 = ch;
    break;
}
}
break;
}
}

//-----*
//函数名: SCISendN                                     *
//功 能: 串行发送 N 个字节                             *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块,其中 SCINo 取值为 0,1 *
//      uint8 n:      发送的字节数                     *
//      uint8 ch[]:   待发送的数据                     *
//返 回: 无                                             *
//说 明: SCINo = 0 表示使用 SCI0 模块,依此类推        *
//      调用了 SCISend1 函数                           *
//-----*
void SCISendN(uint8 SCINo, uint8 n, uint8 ch[])
{
    uint8 i;
    for (i = 0; i < n; i++)
        SCISend1(SCINo, ch[i]);
}

//-----*
//函数名: SCIRel1                                     *
//功 能: 从串口接收 1 个字节的数据                     *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块,其中 SCINo 取值为 0,1 *
//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 *
//说 明: SCINo = 0 表示使用 SCI0 模块,依此类推        *
//-----*
uint8 SCIRel1(uint8 SCINo, uint8 *p)
{
    uint16 k;

```



```
uint8 i;

if(SCINo > 1)
{
    SCINo = 1;                //若传进的通道号大于 2,则按照 2 来接收
}

                                //根据通道号发送数据
switch (SCINo)
{
    case 0:
        for (k = 0; k < 0xfbbb; k++)
            //判断接收缓冲区是否满
            if ((ReSendStatusR0 & (1 << ReTestBit)) != 0)
            {
                i = ReSendDataR0;
                *p = 0x00;
                break;
            }

            //接收失败

            if (k >= 0xfbbb)
            {
                i = 0xff;
                *p = 0x01;
            }

            return i;          //返回接收到的数据
            break;
    case 1:
        for (k = 0; k < 0xfbbb; k++)
            //判断接收缓冲区是否满
            if ((ReSendStatusR1 & (1 << ReTestBit)) != 0)
            {
                i = ReSendDataR1;
                *p = 0x00;
                break;
            }

            //接收失败

            if (k >= 0xfbbb)
            {
                i = 0xff;
                *p = 0x01;
```



```

    }
    return i;                                //返回接收到的数据
    break;
}
}
//-----*
//函数名: SCIReN                                *
//功 能: 从串口接收 N 个字节的数据                *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块,其中 SCINo 取值为 0,1    *
//      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: SCINo = 0 表示使用 SCI1 模块,依此类推      *
//      调用了 SCIRe1 函数                            *
//-----*
uint8 SCIReN(uint8 SCINo, uint8 n, uint8 ch[])
{
    uint8 m;
    uint8 fp;
    m = 0;

    //接收 n 个数据

    while (m<n)
    {
        ch[m] = SCIRe1(SCINo,&fp);
        if (fp == 1) return 1;    //只要有 1 个字节数据没接收到就返回报错
        m++;
    }
    return 0;
}

//-----*
//函数名: SCISendString                        *
//功 能: 串口传输字符串                        *
//参 数: uint8 SCINo: 第 SCINo 个 SCI 模块,其中 SCINo 取值为 1,2    *
//      char * 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 SCISend1 函数                            *
//-----*

```



```

void SCISendString(uint8 SCINo, char * p)
{
    uint32 k;
    if(SCINo > 1)
    {
        SCINo = 1;                //若传进的通道号大于 2,则按照 2 来接收
    }
    if(p == 0) return;
    for(k = 0; p[k] != '\0'; ++k)
    {
        SCISend1(SCINo, p[k]);
    }
}

```

## 2. SCI 构件测试实例

SCI 构件测试的硬件连接,只需要用串口线见 PC 机与 MCU 评估板相连。

### (1) 查询方式的串行通信测试工程实例

```

// -----*
// 工 程 名: SCI 查询方式接收 *
// 硬件连接: 接 PC 机串口 *
// 程序描述: 使用 SCI0 和 PC 机通信 *
//          初始化发送"Hello! World! "到 PC 机 *
//          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: 第一个 Freescale S12X 系列 MCU C 语言程序框架 *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*

//包含头文件
#include "Includes.h"      //包含总头文件
//在此添加全局变量定义
uint8 msg[14] = "Hello! World!";
//主函数
void main(void)
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;    //运行计数器
    uint8 i;
    uint8 SerialBuff[1];    //存放接收数据的数组
    //0.2 关总中断

```

```

DisableInterrupt();
//0.3 芯片初始化
MCUInit(FBUS_32M);
//0.4 模块初始化
Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
SCIInit(0,FBUS_32M,9600); //串口 0 初始化
SCISendN(0,13,msg); //发送"Hello! World!"
// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 50000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT,Light_Run); //指示灯的亮、暗状态切换
    }
    // -----
    //2. 主循环中等待接收 PC 机发送的数据,当接收到了来自 PC 机的数据后立即回发
    i = SCIRen(0,1,SerialBuff); //等待接收 1 个数据
    if (i == 0) SCISendN(0,1,SerialBuff); //发送接收到的数据
    // -----

} //for_end(主循环结束)
} //main_end

```

SCI0 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(调用 SCIRen 函数接收数据),则立即将该数据回发给 PC 机(调用 SCISend1 函数发送数据),随后继续等待接收 1 个数据并回发,如此循环。

注:使用的波特率为 9600,并使用 SCI0 和 PC 机通信。

## (2) 中断方式的串行通信测试工程实例

具体实现:PC 机向 XS128 发送数据,XS128 收到后返回。该实例的详细说明参见 5.4 节的介绍。



## 5.4 XS128 的中断源与第一个带有中断的编程实例

### 5.4.1 中断与异常的通用知识

中断是 MCU 实时地处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时,中断系统将迫使 CPU 暂停正在执行的程序,转而去进行中断事件的处理;中断处理完毕后,又返回被中断的程序处,继续执行下去。实际上中断提供了一种方法来保存当前 CPU 状态和寄存器,转而执行中断服务子程序,然后恢复 CPU 状态,以便返回执行中断之前的处理状态。与只是程序指令的软件中断不同,中断由硬件事件触发。

中断的处理过程一般为关中断(在此中断处理完成前,不处理其他中断)、保护现场、执行中断服务程序、恢复现场、开中断等。

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、不对界错误、违反特权级和跟踪等。许多处理器把硬件复位和硬件中断都归类为异常,把硬件复位看作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指令后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。

### 5.4.2 XS128 的中断机制

#### 1. XS128 的 XINT 中断解析模块

在 S12X 系列的微控制器中都有一个 XINT 模块,该模块主要用于解析中断请求的优先级并且将中断向量交给 CPU(对于 S12XS 系列而言只有 CPU,其他如 XE 系列还存在 XGATE,可以完成中断的处理)执行。

XINT 相关配置寄存器如下:

### (1) 中断向量基址寄存器( Interrupt Vector Base Register, IVBR )

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IVB_ADDR[7 : 0]							
复位	1	1	1	1	1	1	1	1

该寄存器用于设定中断向量地址的高位的值,例如位于 0xFF10~0xFFFE 中断向量的高位地址为 0xFF,那么 IVBR 寄存器的值就为 0xFF。复位时为 0xFF,是为了和先前的 S12 微控制器兼容。一般情况下这个寄存器不需要设置,只有当中断向量被移动到其他地址的时候,这个寄存器才需要设值。

### (2) 中断请求配置地址寄存器( Interrupt Request Configuration Address Register,INT\_CFADDR)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	INT_CFADDR[7 : 4]				0	0	0	0
复位	0	0	0	1	0	0	0	0

这个寄存器和下面的 INT\_CFDATA0 : 7 这 8 个数据寄存器配合使用,用来索引默认的 128 个中断向量地址。具体的使用会在 5.4.4 小节的中断编程实例中说明。

### (3) 中断请求配置数据寄存器(Interrupt Request Configuration Data Registers,INT\_CFDATA 0 - 7)

INT\_CFDATA0 ~ INT\_CFDATA7:

数据位	7	6	5	4	3	2	1	0
定义	RQST	0	0	0	0	PRIOLVL[2 : 0]		
复位	0	0	0	0	0	0	0	1

这 8 个寄存器中的第 7 位 RQST 用于设定中断是由 CPU 处理还是由 XGATE 处理。RQST=0, CPU 处理中断; RQST = 1, XGATE 处理中断。对于 XS128 而言,因为没有 XGATE,所以该位不需要配置,默认为 0。

低 3 位 PRIOLVL[2 : 0]用于设定优先级,取值范围为 0~7。在 XS128 中存在有 8 个等级,其中等级 0 表示中断关闭,等级 1~7,优先级逐步提高,等级 7 的优先级是最高的。

INT\_CFDATA0 : 7 这 8 个寄存器一次可以设定 8 个中断向量优先等级,而中断请求配置地址寄存器 INT\_CFADDR 的高 4 位 INT\_CFADDR[7 : 4]可以设定 = 16 个地址,那么  $16 \times 8 = 128$ 。即 INT\_CFADDR 和 INT\_CFDATA0~7 进行组合可以设定 128 个向量的优先级,这个刚好和 S12X 系列的中断向量表是对应的。采用这样的一个机制设定优先级是为了和 S12 系列保持兼容。

## 2. XS128 的中断执行过程

XS128 与大多数其他 MCU 一样,如果使能的中断源发生一个事件,与之关联的只读中断

状态标志会被置位。只有程序开放 CPU 总中断(条件码寄存器 CCR 中的全局中断屏蔽 I 位为 0),中断才会被响应。复位后,I 被自动初始化为 1(禁止中断),屏蔽了所有可屏蔽中断源。用户程序在清除 I 位以允许 CPU 响应中断之前必须初始化堆栈指针和完成系统其他设置。

当 CPU 响应中断时,I 位自动置 1,以避免其他中断打断中断服务例程 ISR(Interrupt Service Routine,即产生中断嵌套现象)。通常,从中断服务例程 ISR 返回时,CCR 中的 I 位恢复成 0。当 I=0 时,若两个或更多中断等待时,高优先级的中断源首先被服务。在很少的情况下,若 I 位在 ISR 中被清除,这就开放了总中断,允许中断嵌套,这种编程方式,初学者一般不使用。这种方法容易导致难于调试的隐蔽错误,只有编程经验丰富的工程师在特别需要的情况下才使用。

在编程时,在中断服务例程 ISR 开始的地方最好有清除中断标志语句,这样该中断源若有新的一次中断请求,也不会被遗漏,但新的中断请求只有等到当前 ISR 完成之后才会得到响应。

中断发生时,CPU 内部寄存器被自动保存在堆栈中。在中断之前,堆栈指针 SP 指向了堆栈中下一个有效的字节位置。CPU 中寄存器 PC、Y、X、D、CCR 依次入栈,入栈之后,SP 指向堆栈中的下一个有效位置,这是一个比保存条件码寄存器 CCR 的地址小 1 的地址。入栈的 PC 值是主程序中在中断返回时将执行的下一条指令所在地址。中断服务例程以一条中断返回指令 RTI 结束。执行 RTI 指令时中断返回,堆栈中保存的值以相反的顺序从堆栈中恢复到 CCR、D、X、Y、PC 中。图 5-6 给出了中断过程 CPU 中寄存器进出栈情况(注:这里的 D 是指 A+B)。

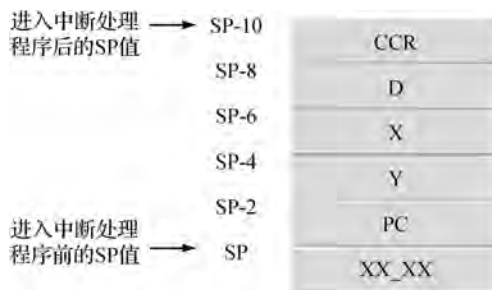


图 5-6 中断过程 CPU 寄存器进出栈情况

XS128 的中断过程详细描述如下:

CPU 每执行完一条指令,若程序有开放某些中断及总中断(使用 CLI 指令),则 CPU 按照优先级次序查询所有中断标志位,若某个中断已发生,则响应该中断请求。当 CPU 收到一个许可的中断请求,它在相应中断之前要完成当前指令。中断顺序遵守与 SWI 指令相同的循环顺序,包括:

- ① 在堆栈中保存 CPU 寄存器——CPU 内的寄存器 PC、Y、X、D、CCR 依次进栈。
- ② CCR 中的 I 位置 1，即自动关总中断(即相当于自动执行 SEI 指令)，防止其他中断进入。
- ③ 在目前等待的中断中取出最高优先级中断的中断向量，从相应的中断向量地址取出中断向量(即中断服务例程的入口地址)送给 PC。
- ④ 执行中断服务例程，直到中断返回指令 RTI。RTI 指令从堆栈中依次弹出 CCR、D、X、Y、PC，CPU 返回原来中断处继续执行。
- ⑤ 若中断过程也允许响应新的中断，可在中断服务程序中用 CLI 指令开放中断。一般不建议这样做，可用其他编程技巧处理相关问题。

### 3. XS128 的中断优先级

在 XS128 中存在有 8 个等级。其中第 0 等级表示中断关闭，等级 1~7，优先级逐步提高。而设定优先级是通过 INT\_CFADDR 和 INT\_CFDATA0 : 7 这两个寄存器进行设定的。下面结合中断代码，解释如何配合使用这两类寄存器设定优先级。

```
const tIsrFunc _InterruptVectorTable[] @0xFF10 = {
    /* ISR name                               No.   Address Pri   Name                               */
    isr_default,                               /* 0x08  0xFF10  -   ivVsi                               */
    isr_default,                               /* 0x09  0xFF12  -   ivVsyscall                          */
    isr_default,                               /* 0x0A  0xFF14  1   ivVReserved118                      */
    isr_default,                               /* 0x0B  0xFF16  1   ivVReserved117                      */
    isr_default,                               /* 0x0C  0xFF18  1   ivVReserved116                      */
    isr_default,                               /* 0x0D  0xFF1A  1   ivVReserved115                      */
    isr_default,                               /* 0x0E  0xFF1C  1   ivVReserved114                      */
    .....部分代码已省略

    isr_default,                               /* 0x62  0xFFC4  1   ivVcrgscm                          */
    isr_default,                               /* 0x63  0xFFC6  1   ivVcrgplllck                       */
    isr_default,                               /* 0x64  0xFFC8  1   ivVReserved27                      */
    isr_default,                               /* 0x65  0xFFCA  1   ivVReserved26                      */
    isr_default,                               /* 0x66  0xFFCC  1   ivVporth                          */
    isr_default,                               /* 0x67  0xFFCE  1   ivVportj                          */
    isr_default,                               /* 0x68  0xFFD0  1   ivVReserved23                      */
    isr_default,                               /* 0x69  0xFFD2  1   ivVatd0                           */
    isr_default,                               /* 0x6A  0xFFD4  1   ivVscil                           */
    SCI0_Recv,                                /* 0x6B  0xFFD6  1   ivVsci0                           */
    isr_default,                               /* 0x6C  0xFFD8  1   ivVspi0                           */
    isr_default,                               /* 0x6D  0xFFDA  1   ivVtimpaie                         */
}
```





```

isr_default,          /* 0x6E 0xFFDC 1 ivVtimpaaovf */
isrTimOver,           /* 0x6F 0xFFDE 1 ivVtimovf */
isr_default,          /* 0x70 0xFFE0 1 ivVtimch7 */

```

上面这段代码来自工程 isr.c 文件(读者可以从附带的网上光盘中获得)。通过观察可以发现,相邻的两个中断向量地址间隔为 2,那么从 0~F 就有 8 个向量,和 XS128 的 INT\_CFDATA0:7 这 8 个寄存器是对应的。例如,对于第一个中断 ivVsi,它的地址为 0xFF10,如果要设定它的优先级的话,需要先设定 INT\_CFADDR 这个寄存器的值为 0x10,然后给 INT\_CFDATA0 这个寄存器赋值来设定优先级。对于第二个中断 ivVsyscall,它的中断向量地址为 0xFF12,如果要设定它的优先级的话,需要先设定 INT\_CFADDR 这个寄存器的值为 0x10,然后给 INT\_CFDATA1 这个寄存器赋值来设定优先级。接下来的 6 个中断依次设定相应的 INT\_CFDATAx 寄存器。到中断向量地址 0xFF20 时,INT\_CFADDR 寄存器的值为 0x20。以此类推,对于串口 0 接收中断 ivVsci0,中断向量地址为 0xFFD6,那么只要设定 INT\_CFADDR 寄存器的值为 0xD0,再设定 INT\_CFDATA3 这个寄存器就可以完成优先级的设定。

图 5-7 为中断优先级的执行流程图。首先 L0(代表等级 0),然后有高等级中断 L3 发生,转入 L3。在 L3 的执行过程中,有更高等级中断 L7 发生,转入 L7 的运行。当 L7 运行完毕,转而执行 L3。当 L3 运行完毕,这个时候有优先级为 L2 的中断发生,比将要执行的 L1 高,于是执行 L2,然后 L1,再到 L0,执行完毕,RTI 返回。

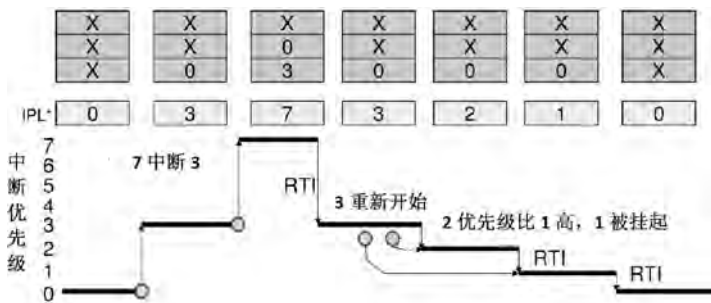


图 5-7 中断优先级执行流程图

#### 4. XS128 的中断源与中断向量表

附录 E 给出了 XS128 的中断源与中断向量表。除去 \$FFFE(复位引脚中断)、\$FFFC(时钟监控复位)与 \$FFFA(看门狗复位),其他中断可以在工程的 isr.c 文件中的找到相应的中断函数处理入口。

### 5.4.3 XS128 的中断编程方法

这里的 XS128 中断仅为 SCI 模块,其他模块中断的应用方法会在后续章节之中分别介绍。

在 XS128 之中,其中有些步骤可由芯片内部机制自行完成。当 SCI 采用中断方式收发数据时,需要编写中断处理程序。在 CW 环境下使用 XS128 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,修改中断向量表。

XS128 MCU 开始运行后,要关闭总中断,它就相当于一个总闸门。如果总闸不开,所有中断都不可能发生。这需要汇编指令来实现:

```
SEI      //关总中断
CLI      //开总中断
```

为了方便代码移植,在 MCUInit.h 文件中做了如下定义:

```
#define EnableInterrupt()   asm("CLI") //开放总中断
#define DisableInterrupt()  asm("SEI") //禁止总中断
```

XS128 的中断编程可概括为下述 3 个步骤:

- ① 新建(或者复制)一个 isr.c 文件,并加入工程中。
- ② 定义中断向量表(若复制 isr.c 文件,则应修改中断向量表)。

在 XS128 的 Flash 地址空间中,有一段是用来存储所有的中断矢量,通常放在最后的 Flash 页面中。该区域每两个字节存储的是一个中断处理函数的地址,各个中断处理函数的地址共同组成一个逻辑上十分规则的区域——中断向量表。

XS128 的中断向量表如下所示:

```
/* 未定义的中断处理函数,本函数不能删除,默认 */
__interrupt void isr_default(void)
{
    DisableInterrupt();
    /* Write your interrupt code here ... */
    EnableInterrupt();
}

//中断向量表,如果需要定义其他中断函数,请修改下表中的相应 ISR_func_t
const ISR_func_t ISR_vectors[] @0xFF10 =
{
    /* ISR name          No.    Address  Pri Name          */
```



```

isr_default,    /* 0x08  0xFF10  -  ivVsi          */
isr_default,    /* 0x09  0xFF12  -  ivVsyscall        */
isr_default,    /* 0x0A  0xFF14  1  ivVReserved118    */
isr_default,    /* 0x0B  0xFF16  1  ivVReserved117    */
isr_default,    /* 0x0C  0xFF18  1  ivVReserved116    */
isr_default,    /* 0x0D  0xFF1A  1  ivVReserved115    */
//此处略去若干中断源。
isr_default     /* 0x7C  0xFFF8  -  ivVtrap          */
//  RESET defined in Project.prm
};

```

中断向量表是一个指针数组,内容是相应中断函数的地址。

首先要定义该中断向量表的地址,XS128 的中断向量从 0xFF10 开始(不同的 MCU 中断矢量起始地址是不相同的,使用时需要查阅相关的技术手册),要使用预编译指令将中断向量表的首地址定义在 0xFF10。

中断向量表格式:const tIsrFunc \_InterruptVectorTable[] @0xFF10 == {中断处理函数名,中断处理函数名……。}。其中 tIsrFunc 的定义为:typedef void (\* near tIsrFunc)(void)。

中断向量表内容是从中断向量表起始地址开始顺序增加,均与 Flash 的中断向量地址相对应,如果某个中断不需要使用,要将在数组对应的项中填入 isr\_default。isr\_default()是中断向量表中不需要使用的中断填入的函数,它是一个空函数。参见 5.4.3 小节中断处理函数文件(isr.c)之中的 isr\_default()函数定义。

③ 定义 ISR 并在中断向量表中填入相应 ISR 的名称。如中断处理函数文件(isr.c)之中的函数\_\_interrupt void SCI0\_Recv(void)的定义。

```

//串行接收中断
__interrupt void SCI0_Recv(void)
{
    uint8 i;
    uint8 SerialBuff[1];
    DisableInterrupt();                                //禁止总中断

    i = SCIReN(0,1,SerialBuff);                        //等待接收 1 个数据
    if (i == 0) SCISendN(0,1,SerialBuff);              //发送接到的数据
    EnableInterrupt();                                  //开放总中断
}

```

通过上述 3 个步骤,就可以定义好所需要的中断。在实际编程中,可以直接从给定的 C 工程框架中得到 isr.c 文件,该文件中只定义了一个空中断处理函数 isr\_default 和由这个空函

数名组成的中断向量表。用户只须定义所需的中断处理函数,并用该函数名代替向量表中相应位置上的 `isr_default` 即可。

④ 在主函数中打开中断模块中断,再打开总中断。

#### 5.4.4 XS128 的中断优先级编程实例

对于需要设定优先级的工程,相比于普通中断例程就是多了一个中断优先级的设定过程。具体设定优先级的方法在 5.4.2 小节中已经介绍,在这里附上代码加以说明。

在网上光盘附中附有中断优先级的例程。在这个例程中共有 3 个中断,PIT 中断、串口中断、定时器中断。

//0.1 主程序使用的变量定义

```
uint32 mRuncount = 0;    //运行计数器
```

//0.2 关总中断

```
DisableInterrupt();
```

//0.3 芯片初始化

```
MCUInit(FBUS_32M);
```

//0.4 模块初始化

```
Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化
```

```
SCIInit(0, FBUS_32M,9600);    //串口 0 初始化
```

```
SCISendN(0,14,msg);          //串口发送"Hello! World!"
```

```
PITInit();                    //定时器 PIT 初始化
```

```
TimerInit();                  //定时器 Timer 初始化
```

//0.5 中断优先级

```
IntP(Vtimovf,1);              //(1)定时器 Timer 溢出中断优先级为 1
```

```
IntP(Vpit0,3);                //(2)定时器 PIT0 中断优先级为 3
```

```
IntP(Vsci0,6);                //(3)串口 0 接收中断优先级为 6
```

//0.6 开放中断

```
EnableSCIReInt0;              //开放 SCI0 接收中断
```

```
EnableTimer;                  //开放 Timer 定时器中断
```

```
EnablePIT0;                    //开放定时器 0 溢出中断
```

```
EnableInterrupt();            //开放总中断
```

为了处理中断优先级,代码中增加了 `INTP.H` 和 `INTP.C` 用于中断优先级的处理。上述代码中的 `IntP` 函数就在这两个文件中定义。

在主函数中的初始化函数中设定这几个中断的优先级。设定 PIT 的优先级为 3,串口的



中断优先级为 6, 定时器的优先级为 1。具体设定代码如下:(下面仅列出串口的优先级设定代码, 其他类似)

IntP(Vsci0, 6); 其实相当于下面的两句代码:

```
INT_CFADDR = 0xD6;
```

```
INT_CFDATA2 = 6;
```

即 Vsci0 其实等于 0x0000FFD6, 那么其实也等于 0xFFD6。

这样只要串口中断发生, 不管前边是否有定时器中断或者 PIT 中断, 都会被强制挂起, 执行串口中断。读者可以将程序下载到核心板中实验一下。

# 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。本章知识点主要有:①键盘扫描基本原理与编程方法;②LED 扫描基本原理与编程方法;③字符型 LCD 的编程举例。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。

## 6.1 键盘技术概述

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 XS128 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。

### 6.1.1 键盘模型及接口

键盘是由若干个按键组成的开关矩阵,是最简单的 MCU 数字量输入设备。操作员通过键盘输入数据或命令,实现简单的人机通信。键盘模型及按键抖动示意图如图 6-1 所示。

首先,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,通、断两种状态分别表示 0 和 1。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。本章将使用中断方法编程。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。

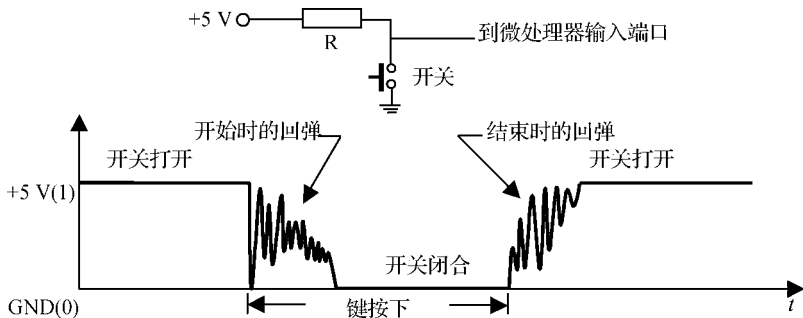


图 6-1 键盘模型及按键抖动示意图

### (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,如图 6-2 所示。读键值时直接读 I/O 口,每个键值通过读入对应 I/O 的状态来反映,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,如图 6-3 所示。读键值方法一般采用扫描方式,即 MCU 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? 下节就讨论这个问题,这对于实际应用非常重要。

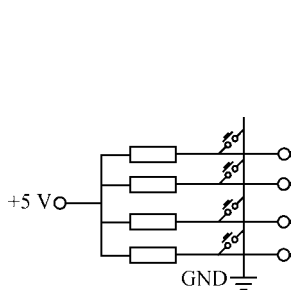


图 6-2 独立式键盘

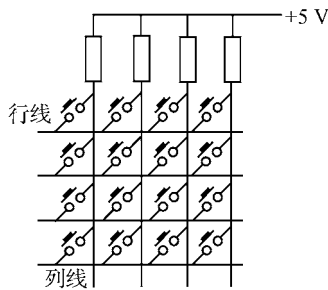


图 6-3 矩阵式键盘



## 6.1.2 键盘编程的基本问题

对于键盘编程至少应该了解下面几个问题:

- ① 如何识别键盘上的按键?
- ② 如何区分按键是否真正地被按下, 还是抖动?
- ③ 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, 则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  $5 \sim 10 \text{ ms}$  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。

为了正确理解 MCU 键盘接口方法与编程技术, 下面以  $4 \times 4$  键盘为例说明按键识别的基本编程原理。 $4 \times 4$  的键盘结构如图 6-4 所示, 图中列线( $n_1 \sim n_4$ )通过电阻接  $+5 \text{ V}$ , 当键盘上没有键闭合时, 所有的行线和列线断开, 列线  $n_1 \sim n_4$  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路。例如第 2 排第 3 个按键被按下闭合时, 行线  $m_2$  和列线  $n_3$  短路, 此时  $n_3$  线上的电平由  $m_2$  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  $n_1 \sim n_4$  接到 MCU 的输入口, 行线  $m_1 \sim m_4$  接到 MCU 的输出口, 则在微机的控制下, 使行线  $m_1$  为低电平(0), 其余 3 根行线  $m_2$ 、 $m_3$ 、 $m_4$  都为高电平, 读列线  $n_1 \sim n_4$  状态。如果  $n_1 \sim n_4$  都为高电平, 则  $m_1$  这一行上没有键闭合, 如果读出列线  $n_1 \sim n_4$  的状态不全为高电平, 那么为低电平的列线和  $m_1$  相交的键处于闭合状态; 如果  $m_1$  这一行上没有键闭合, 接着使行线  $m_2$  为低电平, 其余行线为高电平, 用同样方法检查  $m_2$  这一行上有没有键闭合; 以此类推, 最后使行线  $m_4$  为低电平, 其余的行线为高电平,

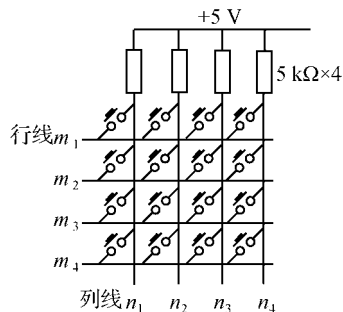


图 6-4  $4 \times 4$  键盘的结构



检查  $m_4$  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。CPU 对键盘扫描可以采用查询方式或者中断方式。即在主循环中,每次循环对键盘进行一次扫描而获取按键值;或者将键盘接在具有中断功能的 I/O 引脚,按键时触发 I/O 引脚的中断,通过中断服务程序对键盘进行一次扫描而获取按键值。键面的定义值通过查表方式将键值转换而得到。

### 6.1.3 键盘构件设计与测试实例

根据键盘扫描原理,本小节给出一个实际的  $4 \times 4$  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。键盘与 MCU 的接线原理图如图 6-5 所示,图中键盘列线( $n_1 \sim n_4$ )与 PTIP 的 0~3 号引脚相连,数据方向定义输入,内部电阻上拉(图中电阻不是外接的),该端口具有中断功能;行线( $m_1 \sim m_4$ )与 PORTA 的 0~3 号引脚相连。当键盘上没有键闭合时,所有的行线和列线断开,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。下面介绍键值计算方法。

键盘与 MCU 接线如图 6-5 所示。图 6-6 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  $n_1 \sim n_4$  分别接 PTIP0、PTIP1、PTIP2、PTIP3,且编程时将这 4 个引脚定义为输入并进行上拉,行线  $m_1 \sim m_4$  分别接 PORTA0~PORTA3,且编程时将 PORTA0~PORTA3 定义为输出,那么“1”键对应于按照 PTIP3、PTIP2、PTIP1、PTIP0、PORTA3、PORTA2、PORTA1、PORTA0 的顺序为:11101110,即 \$EE;“2”键对应于:11011110,即 \$DE;……;“D”键对应于:01110111,即 \$77。前者“1”、“2”、“D”就是定义值,后者 \$EE、\$DE、\$77 就是“键值”,这种情况“键值”是一个字节。这样,按图 6-5 的接法可以得出键值表,如图 6-6 所示。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。注意,定义值可以根据键盘的外观来进行更改,在样例程序中使用的键值与定义值与图 6-6 就不一样,具体可参见键盘中断样例程序。

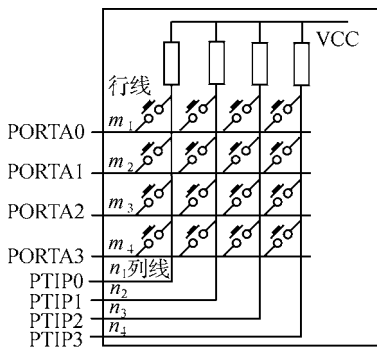


图 6-5 键盘与 MCU 的接法

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 6-6 键盘定义

从软件角度来看,键盘构件包含头文件 KBI\_I.h 和程序文件 KBI\_I.c。本章介绍的主要是上文所讲述的中断法,故还应该有一个与之相对应的中断处理函数。由图 6-5 可看出,键盘与 MCU 的 PTIP、PORTA 部分引脚连接,在进行键盘按键扫描时,需设置 PTIP、PORTA 某些引脚的输出或读取某些引脚的状态,显然,在编写键盘构件时须调用 GPIO 构件。

## 1. 键盘构件设计

### (1) 键盘构件头文件(KBI\_I.h)

```
// ----- *
// 文件名: KBI_I.h *
// 说 明: KBI_I 构件头文件 *
// ----- *

#ifndef KBI_I_H
#define KBI_I_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//接线说明:PTP0~3 键盘 4 根列线(允许中断输入),PTA3-0 接键盘 4 根行线(扫描线)
//键盘控制引脚定义
#define KB_A PA //PA 口
#define KB_P PP //PTP 口
#define EnableKBInt() PIEP |= 0x0F //开放键盘(PTP0~3)中断
#define DisableKBInt() PIEP &= ~0x0F //屏蔽键盘(PTP0~3)中断

//构件函数声明区

void KBIInit(void); //初始化键盘模块
uint8 KBScanN(uint8 N); // N 次扫描键盘,消除"抖动"问题
uint8 KBDef(uint8 valve); //键值转为定义值
uint8 KBScan1(void); //扫描 1 次 4 * 4 键盘,返回读取的键值

#endif
```



## (2) 键盘构件程序文件(KBI\_I.c)

```
// ----- *
// 文件名: KBI_I.c *
// 说 明: KBI 构件函数源文件 *
// ----- *

// 头文件包含,及宏定义区
// 头文件包含
# include "KBI_I.h"
// 构件函数实现

// ----- *
// 函数名: KBIInit: 键盘初始化函数 *
// 功 能: 初始化键盘控制寄存器及相关的键盘中断寄存器,但未开放键盘中断 *
// 参 数: 无 *
// 返 回: 无 *
// ----- *
void KBIInit(void)
{

    GPPort_Set(KB_P,PTI,1,0x0F); //复位相应寄存器
    GPPort_Set(KB_P,DDR,0,0xF0); //定义列线(PTP0~3)为输入
    GPPort_Set(KB_A,PRT,0,0xF0); //复位相应寄存器
    GPPort_Set(KB_A,DDR,1,0x0F); //行线(PTA0~3)为输出
    GPPort_Set(KB_P,PER,1,0x0F); //输入引脚(列线)有内部上拉电阻
    GPPort_Set(KB_P,PPS,0,0xF0); //下降沿产生中断
    DisableKBInt(); //禁止键盘中断
    GPPort_Set(KB_P,PIF,1,0x0F); //清除键盘中断请求
}

// ----- *
// 函数名: KBScanN: 多次扫描键盘函数 *
// 功 能: 多次扫描键盘,消除"抖动" *
// 参 数: 重复扫描键盘的次数(KB_count) *
// 返 回: 多次扫描键盘得到的键值 *
// 说 明: 本函数调用了 KBscan1 函数 *
// ----- *
uint8 KBScanN(uint8 KB_count)
{
```

```

uint8 i,KB_value_last,KB_value_now;
//先扫描一次得到的键值,便于下面比较
if (0 == KB_count || 1 == KB_count)
    return KBScan1();
KB_value_now = KB_value_last = KBScan1();
//以下多次扫描消除误差
for (i = 0; i < KB_count - 1; i++)
{
    KB_value_now = KBScan1();
    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;
}
//返回出错标志
return 0xFF;
}

//-----*
//函数名:KBDef;键值转为定义值函数                                *
//功 能:键值转为定义值                                            *
//参 数:键值 valve                                                *
//返 回:键定义值                                                  *
//-----*
//键盘定义表
const uint8 KBtable[] =
{
    0xEE,7,0xDE,8,0xBE,9,0x7E,C,
    0xED,4,0xDD,5,0xBD,6,0x7D,D,
    0xEB,1,0xDB,2,0xBB,3,0x7B,E,
    0xE7,0,0xD7,A,0xB7,B,0x77,F,
    0x00
};
uint8 KBDef(uint8 valve)
{
    uint8 KeyPress;                //键定义值
    uint8 i;
    i = 0;

```



```
KeyPress = 0xff;
while (KBtable[i] != 0x00) //在键盘定义表中搜索欲转换的键值,直至表尾
{
    if(KBtable[i] == valve) //在表中找到相应的键值
    {
        KeyPress = KBtable[i + 1]; //取出对应的键定义值
        break;
    }
    i += 2; //指向下一个键值,继续判断
}
return KeyPress;
}

//-----*
//函数名:KBScan1:扫描1次4*4键盘,返回读取的键值 *
//功 能:扫描1次4*4键盘,返回扫描到的键值,若无按键,返回0xff *
//参 数:无 *
//返 回:扫描到的键值 *
//-----*
uint8 KBScan1(void)
{
    uint8 i,tmp;
    for (i = 0; i <= 3; i++) //最多将扫描4根行线
    {
        GPPort_Set(KB_A,PRT,1,0x0F); //当前扫描的一行,输出低电平

        GPPort_Set(KB_A,PRT,0,~(1<<i));

        asm("NOP");
        asm("NOP");

        //读取键盘口数据寄存器
        tmp = GPPort_Get(KB_P,PTI)&0x0F;
        //通过观察4根列线中是否出现低电平来判断当前行有无按键
        if (tmp != 0x0F) //当前行有键按下
        {
            tmp = (tmp<<4)|(GPPort_Get(KB_A,PRT)&0x0F);
            break; //退出循环不再扫描
        }
    }
}
```

```

        if (i == 4)                                //无按键,以后将返回 0xff
            tmp = 0xFF;
        return (tmp);
    }

```

## 2. 键盘构件测试实例

```

// -----*
// 工 程 名: 键盘中断                                     *
// 硬件连接: 接 PC 机串口                                 *
//          键盘插在手动排。然后接线 PTP0~3 分别接 D 形板键盘接线处的 1~4,PTA0~3 *
//          分别接 5~8,PB 口的 1 脚接小灯                                     *
// 程序描述: 使用 SCI0 和 PC 机通信,按下键盘按键,串口发送对应的键值         *
// 目    的: 掌握键盘编程方法                                             *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*

//包含头文件
#include "Includes.h"      //包含总头文件

//在此添加全局变量定义
//主函数
void main()
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;      //运行计数器

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
    SCIInit(0,FBUS_32M,9600);      //串口 0 初始化
    KBInit();                      //键盘初始化

    //0.5 开放中断
    EnableSCIReInt0;               //开放 SCI0 接收中断
    EnableKBInt();                //开放键盘中断
}

```





```

EnableInterrupt();                                //开放总中断

// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 50000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
    }
}
}
}

```

### 3. 键盘中断处理函数

```

//键盘中断
__interrupt void isrKeyBoard(void)
{
    uint8 value;
    uint16 i;
    DisableInterrupt();
    for (i = 0 ; i < 20000; i + + );                //延时
    value = KBScanN(10);                            //扫描键值,存于 value 中
    SCISend1(0, value);                             //发送键值
    SCISend1(0, KBDef(value));                      //发送键盘值
    KBInit();                                       //键盘初始化
    EnableKBInt();                                //开放键盘中断
    EnableInterrupt();                            //开放总中断
}

```

测试结果:通过按键可在串口调试工具上显示相应按键的对应键值和定义值。

## 6.2 LED 技术概述

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。

### 6.2.1 扫描法 LED 显示编程原理

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,共阴极还是共阳极?

第二,所选 LED 的电气参数怎样?如额定功率、额定电流是多少?对这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 7 段 LED 就够了;若同时又要显示小数点,则需使用 8 段 LED。8 段数码管由 8 个发光二极管(Light-Emitting Diode,LED)组成。MCU 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(如图 6-7 所示)。例如,图 6-8 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。

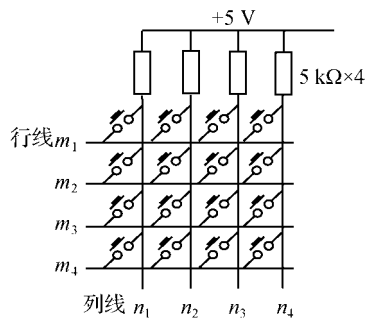


图 6-7 数码管

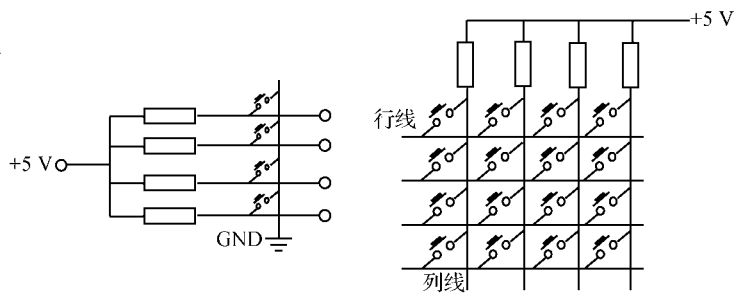


图 6-8 数码管外形

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,PORTB0~7 分别接 a~g、dp,即  $PORTB = 0b01111111$  时 dp 段亮;当  $PORTB = 0b10000000$  时除 dp 位段外,其他位段均亮。到此基本弄清对一个 LED 编程的原理了,下面需要注意的是在进行硬件连接时选用的 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  $V_{cc}$ ),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。若要控制更多的数码管,则可



以考虑外加一个译码芯片。图 6-9 为一个 4 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 6-9 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 100 ms)效应, 产生的看起来则是同时显示的效果。

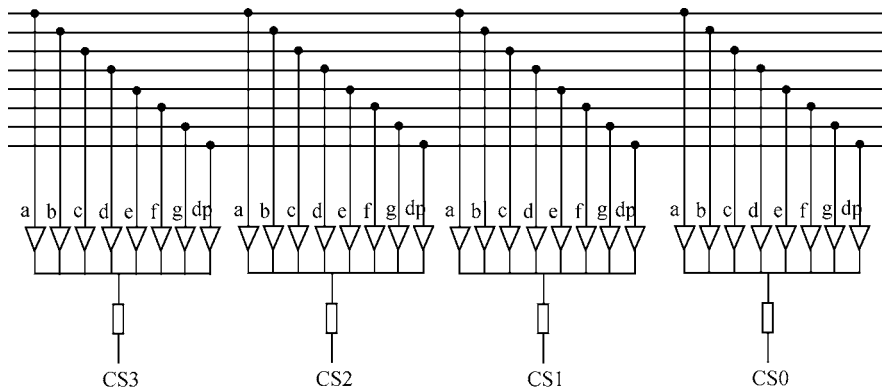


图 6-9 4 连排共阴极 8 段数码管

## 6.2.2 LED 构件设计与测试实例

图 6-10 给出了一个共阴极 4 连排 8 段数码管的硬件构件连接实例。利用 MCU 的 PTB 口控制 8 个位段(数据)。图中 PORTB7~0 分别接 h~a 位段, PORTE 口 6、5、3、2 脚作为片选端。

以下给出 XS128 对上述 4 连排 LED 的 C 语言编程实例。

### 1. LED 构件设计

#### (1) LED 构件头文件(LED.h)

```
// -----*
// 文件名: LED.h                                     *
// 说 明: LED 驱动程序头文件                         *
// -----*

#ifndef LED_H
#define LED_H

//头文件包含,及宏定义区
//头文件包含
```

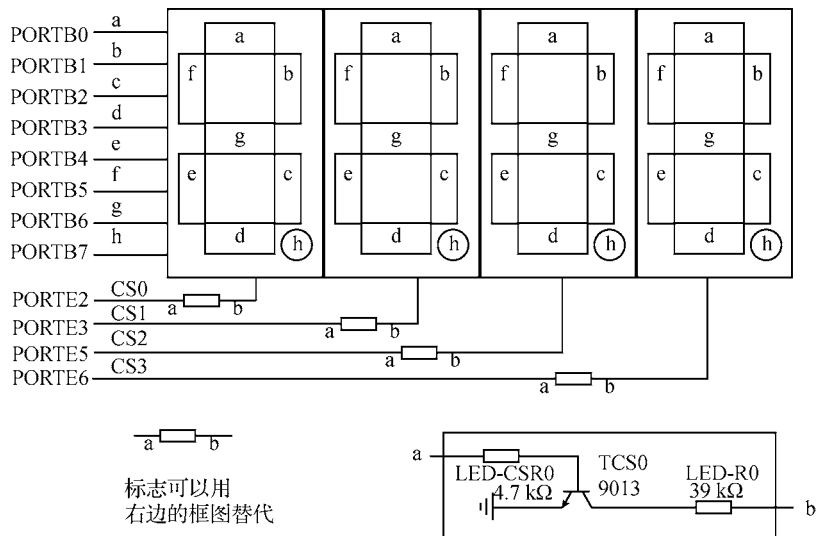


图 6-10 MCU 与 4 连排 8 段数码管的连接

```
#include "Includes.h"

//IO 引脚分配给 LED. PE2,PE3,PE5,PE6 for LED_CS; PB0~PB7 for LED_DATA.
//用户可以自由分配 12 个 IO 口给 LED
//位选口(数据)
#define LED_CS      PE

//数据口(数据)
#define LED_D        PB

//构件函数声明区

void LEDInit(void);           //定义 LED 控制引脚的数据口和位选口为输出
void LEDShow(uint8 * Buf);   //在 4 连排 LED 上显示 4 个十进制数

//内部函数声明
void LEDShow1(uint8 i, uint8 c); //在第 i 个 LED 上显示数字 c(要查表转码)

#endif
```



## (2) LED 构件程序文件(LED. c)

```
// ----- *
// 文件名: LED. c *
// 说 明: LED 驱动函数文件 *
// ----- *

// 头文件包含,及宏定义区

// 头文件包含
#include "LED. h"

// 构件函数实现
// ----- *
// LEDInit: 4 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)
{
    // 数据口定义为输出
    GPPort_Set(LED_D, DDR, 2, 0xFF);
    // 数据口输出 0
    GPPort_Set(LED_D, PRT, 2, 0x00);

    // 位选口定义为输出
    GPPort_Set(LED_CS, DDR, 1, 0b01101100);
    // 位选口输出 0
    GPPort_Set(LED_CS, PRT, 0, 0b10010011);
}

// ----- *
// LEDShow: 在 4 连排 LED 上显示 4 个十进制数 *
// 功能: 在 4 连排 LED 上显示以 Buf 为首地址的 4 个数据 *
// 参数: Buf = 待显示数据的首地址 *
// 返回: 无 *
// 内部调用: LEDShow1 *
// ----- *
void LEDShow(uint8 * Buf)
```

```

{
    uint8 i,c;
    uint16 j;
    for (i = 0;i <= 3;i++)
    {
        c = Buf[i]-0;
        LEDShow1(i,c);
        for (j = 0;j <= 100;j++);          //延时
    }
}

//-----*
//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 0-3)              *
//    (2)c:要显示的数字(0-9)                            *
//返回:无                                                *
//                                                        *
//    共阴极                                              *
//    0    1    2    3    4    5    6    7    8    9    *
//    {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F}; *
//    共阳极                                              *
//    0    1    2    3    4    5    6    7    8    9    *
//    {0xC0,0xF9,0xA4,0xD0,0x99,0x92,0x82,0xF8,0x80,0x90}; *
//    -----*
//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};

//显示片选表
const uint8 CSTable[4]=
//    0    1    2    3
    {0x04,0x08,0x20,0x40};

void LEDShow1(uint8 i, uint8 c)

```



```

{
    GPPort_Set(LED_D,PRT,2,0b00000000);    //清数据口,避免产生影子
    GPPort_Set(LED_CS,PRT,0,0b10010011);    // 全不选中
    GPPort_Set(LED_CS,PRT,1,CSTable[i]);    // 显示位选择
    GPPort_Set(LED_D,PRT,2,DTable[c]);      // 显示数值设定 0~9,(共阴极)
}

```

## 2. LED 测试实例

```

// -----*
// 工 程 名: LED                                     *
// 硬件连接: 见 LED.h 文件                           *
// 程序描述: LED 数码管测试模块,LED 数码管显示2011~ *
// 目    的: 掌握 LED 编程的基本方法                 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 年 -----*

//包含头文件
#include "Includes.h"      //包含总头文件

//在此添加全局变量定义
uint8  LEDbuf[4];        //存放需要 LED 显示数据

//主函数
void main()
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;    //运行计数器

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF);    //RUN 指示灯初始化为暗
    LEDInit();                                           //LED 初始化

    LEDbuf[0] = 2;                                       //待显示数据 2011
    LEDbuf[1] = 0;

```



```

LEDbuf[2] = 1;
LEDbuf[3] = 1;
// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 5000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
    }

    // ----- *
    //2. LED 显示
    LEDShow(LEDbuf);
} //for_end(主循环结束)
} //main_end

```

测试结果:观察 4 位 LED,显示 2011 这 4 位数字。修改 main.c 中输入数据可从 LED 上观察到不同的结果,以验证程序的正确性。

## 6.3 LCD 技术概述

本节简要概述液晶显示器(Liquid Crystal Display, LCD)的基本特点及分类方法。

### 6.3.1 LCD 的特点和分类

LCD 作为电子信息产品的主要显示器件,相对于其他类型的显示部件来说有其自身的特点,概要如下:

① 低电压低功耗:LCD 的工作电压一般为 3~5 V,每平方厘米的液晶显示屏的工作电流为  $\mu\text{A}$  级,所以液晶显示器件为电池供电的电子设备的的首选显示器件。

② 平板型结构:LCD 的基本结构是由两片玻璃组成的很薄的盒子。这种结构具有使用方便、生产工艺简单等优点。特别是在生产上,适宜采用集成化生产工艺,通过自动生产流水线可以快速、大批量地生产。

③ 使用寿命长:LCD 器件本身几乎没有劣化问题。若能注意器件防潮、防压、防止划伤、防止紫外线照射、防静电等,同时注意使用温度, LCD 可以使用很长时间。



④ 被动显示:对 LCD 来说,环境光线越强显示内容越清晰。人眼所感受的外部信息 90% 以上是外部物体对光的反射,而不是物体本身发光,所以被动显示更适合人的视觉习惯,更不容易引起疲劳。这在信息量大、显示密度高、观看时间长的场合显得更重要。

⑤ 显示信息量大且易于彩色化:LCD 与 CRT 相比,由于 LCD 没有荫罩限制,像素可以做得很小,这对于高清晰电视是一种理想的选择方案。同时液晶易于彩色化,方法也很多。特别是液晶的彩色可以做得更逼真。

⑥ 无电磁辐射:CRT 工作时,不仅会产生 X 射线,还会产生其他电磁辐射,影响环境。LCD 则不会有这类问题。

液晶显示器件分类方法有多种,这里简要介绍以下几种分类方法:

### (1) 按电光效应分类

所谓电光效应是指在电的作用下,液晶分子的初始排列改变为其他排列形式,从而使液晶盒的光学性质发生变化,也就是说以电通过液晶分子对光进行了调制。不同的电光效应可以制成不同类型的显示器件。

按电光效应分类,LCD 可分为电场效应类、电流效应类、电热写入效应类和热效应类。其中电场效应类又可分为扭曲向列效应(TN)类、宾主效应(GH)类和超扭曲效应(STN)类等。MCU 系统中应用较广泛的是 TN 型和 STN 型液晶器件,由于 STN 型液晶器件具有视角宽、对比度好等优点,几乎所有 32 路以上的点阵 LCD 都采用了 STN 效应结构,STN 型正逐步代替 TN 型而成为主流。

### (2) 按显示内容分类

按显示内容分类,LCD 可分为字段型(或称为笔划型)、点阵字符型、点阵图形型 3 种。

字段型 LCD 是指以长条笔划状显示像素组成的液晶显示器件。字段型 LCD 以 7 段显示最常用,也包括为专用液晶显示器设计的固定图形及少量汉字。字段型 LCD 主要应用于数字仪表、计算器、计数器中。

点阵字符型 LCD 是指显示的基本单元由一定数量点阵组成,专门用于显示数字、字母、常用图形符号及少量自定义符号或汉字。这类显示器把 LCD 控制器、点阵驱动器、字符存储器等全做在一块印刷电路板上,构成便于应用的液晶显示模块。点阵字符型液晶显示模块在国际上已经规范化,有统一的引脚与编程结构。点阵字符型液晶显示模块有内置 192 个字符,另外用户可自定义  $5 \times 7$  点阵字符或  $5 \times 11$  点阵字符若干个。显示行数一般为 1 行、2 行、4 行这 3 种。每行可显示 8 个、16 个、20 个、24 个、32 个、40 个字符不等。

点阵图形型除了可显示字符外,还可以显示各种图形信息、汉字等,显示自由度大。常见的模块点阵从  $80 \times 32$  到  $640 \times 480$  不等。

### (3) 按 LCD 的采光方式分类

LCD 器件按其采光方式分类,分为带背光源与不带背光源两大类。不带背光的 LCD 显示是靠背面的反射膜将射入的自然光从下面反射出来完成的。大部分计数、计时、仪表、计算

器等计量显示部件都是用自然光源,可以选择使用不带背光的 LCD 器件。如果产品需要在弱光或黑暗条件下使用,可以选择带背光型 LCD,但背光源增加了功耗。

### 6.3.2 点阵字符型液晶显示模块

我们已经知道,点阵字符型 LCD 专门用于显示数字、字母、图形符号及少量自定义符号。这类显示器把 LCD 控制器、点阵驱动器、字符存储器、显示体及少量的阻容元件等集成为一个液晶显示模块。鉴于字符型液晶显示模块目前在国际上已经规范化,其电特性及接口特性是统一的,因此只要设计出一种型号的接口电路,在指令上稍加修改即可使用各种规格的字符型液晶显示模块。

点阵字符型液晶显示模块的控制器大多为日立公司生产的 HD44780 及其兼容的控制电路,如 SED1278(SEIKO EPSON)、KS0066(SAMSUNG)、NJU6408(NEC JAPAN RADIO)等。

字符型液晶显示模块的主要特点如下:

① 液晶显示屏是以若干  $5 \times 8$  或  $5 \times 11$  点阵块组成的显示字符群。每个点阵块为一个字符位,字符间距和行距都为一个点的宽度。

② 主控制电路为 HD44780(HITACHI)及其他公司的兼容电路。从程序员角度来说,LCD 的显示接口与编程是面向 HD44780 的,只要了解 HD44780 的编程结构即可进行 LCD 的显示编程。

③ 内部具有字符发生器 ROM(CG ROM - Character - Generator ROM),可显示 192 种字符(160 个  $5 \times 7$  点阵字符和 32 个  $5 \times 10$  点阵字符)。

④ 具有 64 字节的自定义字符 RAM(CG RAM - Character - Generator RAM),可以定义 8 个  $5 \times 8$  点阵字符或 4 个  $5 \times 11$  点阵字符。

⑤ 具有 64 字节的数据显示 RAM(DD RAM - Data - Display RAM),供显示编程时使用。

⑥ 标准接口特性,与 H68HC08 系列 MCU 容易接口。

⑦ 模块结构紧凑、轻巧、装配容易。

⑧ 单 +5 V 电源供电(宽温型需要加 -7 V 驱动电源)。

⑨ 低功耗、高可靠性。

### 6.3.3 HD44780

#### (1) HD44780 的引脚信号

HD44780 的外部接口信号一般有 14 条,有的型号显示器使用 16 条,其中与 MCU 的接口有 8 条数据线、3 条控制线,见表 6-1。



表 6-1 HD44780 的引脚信号

引脚号	符 号	电 平	方 向	引脚含义说明
1	V <sub>ss</sub>			电源地
2	V <sub>dd</sub>			电源(+5 V)
3	V <sub>0</sub>			液晶驱动电源(0~5 V)
4	RS	H/L	输入	寄存器选择:1—数据寄存器 0—指令寄存器
5	R/ $\overline{W}$	H/L	输入	读写操作选择:1—读操作 0—写操作
6	E	H/LH→L	输入	使能信号:R/ $\overline{W}$ =0,E 下降沿有效 R/ $\overline{W}$ =1,E=1 有效
7~10	DB0~DB3		三态	8 位数据总线的低 4 位,若与 MCU 进行 4 位传送时,此 4 位不用
11~14	DB4~DB7		三态	8 位数据总线的高 4 位,若与 MCU 进行 4 位传送时,只用此 4 位
15~16	E1~E2		输入	上下两行使能信号,只用于一些特殊型号

## (2) HD44780 的时序信号

图 6-11 给出了 HD44780 的写操作时序,图 6-12 给出了 HD44780 的读操作时序。

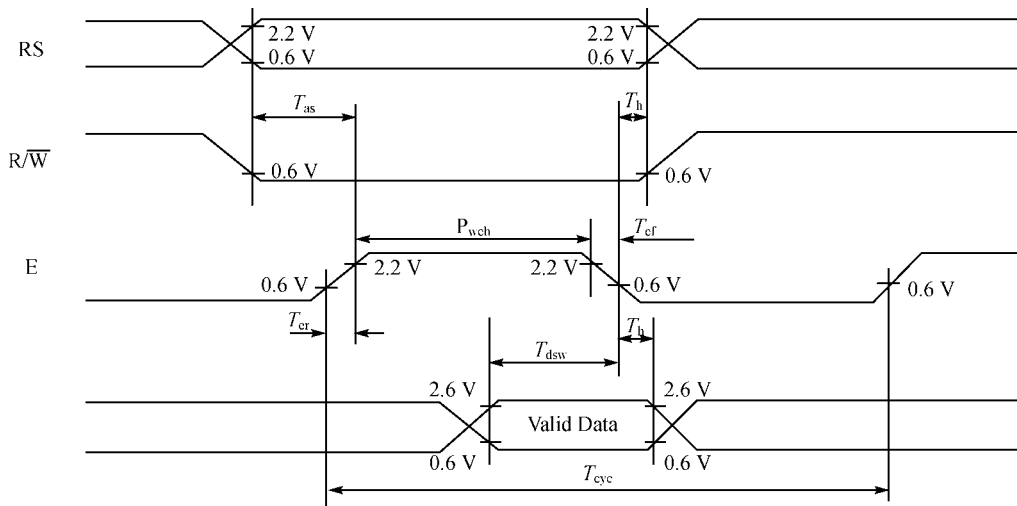


图 6-11 HD44780 的写操作时序

## (3) HD44780 的编程结构

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。

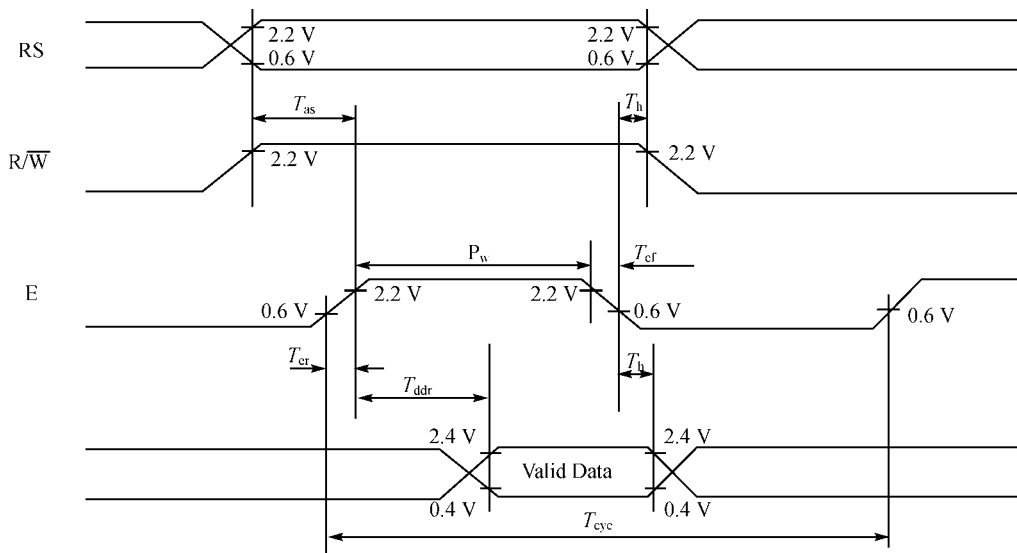


图 6-12 HD44780 的读操作时序

### 1) 指令寄存器(IR)

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 上的数据写入指令寄存器 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,表示组件正在进行内部操作,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,例如:

8×1(8 个字符,1 行)

字符位置	1	2	3	4	5	6	7	8
地 址	00	01	02	03	04	05	06	07

16×1(16 个字符,1 行)

字符位置	1	2	.....	8	9	10	.....	16
地 址	00	01	.....	07	08	09	.....	0F

16×2(每行 16 个字符,共 2 行)

字符位置	1	2	.....	8	9	10	.....	16
第一行地址	00	01	.....	07	08	09	.....	0F
第二行地址	40	41	.....	47	48	49	.....	4F

16×4(每行 16 个字符,共 4 行)

字符位置	1	2	.....	8	9	10	.....	16
第一行地址	00	01	.....	07	08	09	.....	0F
第二行地址	40	41	.....	47	48	49	.....	4F
第三行地址	10	11	.....	17	18	19	.....	1F
第四行地址	50	51	.....	57	58	59	.....	5F

具体的对应关系,可参阅使用说明书。

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,见图 6-13。图中给出了 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。

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。

(4) HD44780 的指令集

1) 清屏(Clear Display)

RS、R/ $\overline{W}$ =00,DATA=0000 0001。清屏指令使 DD RAM 的内容全部被清除,屏幕光标回原位,地址计数器 AC=0。运行时间(250 kHz)约为 1.64 ms。

Lower 4 Bits	Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	
xxxx0000	CG RAM (1)				0	a	P	`	P				一	夕	ミ	α	p	
xxxx0001	(2)			!	1	A	Q	a	q				。	ア	チ	△	ä	q
xxxx0010	(3)			"	2	B	R	b	r				「	イ	ツ	×	ƒ	θ
xxxx0011	(4)			#	3	C	S	c	s				」	ウ	テ	モ	ε	∞
xxxx0100	(5)			\$	4	D	T	d	t				、	エ	ト	チ	μ	Ω
xxxx0101	(6)			%	5	E	U	e	u				・	オ	ナ	ユ	€	Ü
xxxx0110	(7)			&	6	F	V	f	v				ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)			'	7	G	W	g	w				ア	キ	ヲ	ラ	g	π
xxxx1000	(1)			<	8	H	X	h	x				イ	ク	ネ	リ	ƒ	Σ
xxxx1001	(2)			>	9	I	Y	i	y				ウ	ケ	ノ	ル	ˆ	Ÿ
xxxx1010	(3)			*	:	J	Z	j	z				エ	コ	ハ	レ	j	千
xxxx1011	(4)			+	:	K	[	k	{				オ	サ	ヒ	ロ	*	万
xxxx1100	(5)			,	<	L	¥	l	l				ヤ	シ	フ	ワ	Φ	円
xxxx1101	(6)			-	=	M	]	m	}				ユ	ズ	ヘ	ン	も	÷
xxxx1110	(7)			.	>	N	^	n	→				ヨ	セ	ホ	°	ñ	
xxxx1111	(8)			/	?	O	_	o	+				ッ	ソ	マ	°	ö	■

图 6-13 HD44780 内藏字符集

## 2) 归位 (Return Home)

RS、 $R/\overline{W}$  = 00, DATA = 0000 001 \* (注: “\*” 表示任意, 下同)。归位指令使光标和光标所在位的字符回原点(屏幕的左上角)。但 DD RAM 单元内容不变。地址计数器 AC = 0。运行时间(250 kHz)约为 1.64 ms。

## 3) 输入方式设置 (Entry Mode Set)

RS、 $R/\overline{W}$  = 00, DATA = 0000 01AS。该指令设置光标、画面的移动方式。下面解释 A、S 位的含义。A = 1 时数据读写操作后, AC 自动增 1; A = 0 时数据读写操作后, AC 自动减 1。若 S = 1, 当数据写入 DD RAM 显示将全部左移(A = 1)或全部右移(A = 0), 此时光标看上去未动, 仅仅是显示内容移动, 但从 DD RAM 中读取数据时, 显示不移动; S = 0 时显示不移动, 光标左移(A = 1)或右移(A = 0)。

## 4) 显示开关控制 (Display ON/OFF Control)

RS、 $R/\overline{W}$  = 00, DATA = 0000 1DCB。该指令设置显示、光标及闪烁开、关。D: 显示控制,





D=1,开显示(Display ON);D=0,关显示(Display OFF)。C:光标控制,C=1,开光标显示;C=0,关光标显示。B:闪烁控制,B=1,光标所指的字符同光标一起以 0.4 s 交变闪烁;B=0,不闪烁。运行时间(250 kHz)约为 40  $\mu$ s。

### 5) 光标或画面移位(Cursor or Display Shift)

RS、 $R/\overline{W}$ =00,DATA=0001 S/C R/L \* \*。该指令使光标或画面在没有对 DD RAM 进行读/写操作时被左移或右移,不影响 DD RAM。S/C=0、R/L=0,光标左移一个字符位,AC 自动减 1;S/C=0、R/L=1,光标右移一个字符位,AC 自动加 1;S/C=1、R/L=0,光标和画面一起左移一个字符位;S/C=1、R/L=1,光标和画面一起右移一个字符位。运行时间(250 kHz)约为 40  $\mu$ s。

### 6) 功能设置(Function Set)

RS、 $R/\overline{W}$ =00,DATA=001 DL N F \* \*。该指令为工作方式设置命令(初始化命令)。对 HD44780 初始化时,需要设置数据接口位数(4 位或 8 位)、显示行数、点阵模式(5 $\times$ 7 或 5 $\times$ 10)。DL:设置数据接口位数,DL=1,8 位数据总线 DB7~DB0;DL=0,4 位数据总线 DB7~DB4,而 DB3~DB0 不用,在此方式下数据操作需两次完成。N:设置显示行数,N=1,2 行显示;N=0,1 行显示。F:设置点阵模式,F=0,5 $\times$ 7 点阵;F=1,5 $\times$ 10 点阵。运行时间(250 kHz)约为 40  $\mu$ s。

### 7) CG RAM 地址设置(CG RAM Address Set)

RS、 $R/\overline{W}$ =00,DATA=01 A5 A4 A3 A2 A1 A0。该指令设置 CG RAM 地址指针。A5~A0=00 0000~11 1111。地址码 A5~A0 被送入 AC 中,在此后,就可以将用户自定义的显示字符数据写入 CG RAM 或从 CG RAM 中读出。运行时间(250 kHz)约为 40  $\mu$ s。

### 8) DD RAM 地址设置(DD RAM Address Set)

RS、 $R/\overline{W}$ =00,DATA=1 A6 A5 A4 A3 A2 A1 A0。该指令设置 DD RAM 地址指针。若是一行显示,地址码 A6~A0=00~4FH 有效;若是二行显示,首行址码 A6~A0=00~27H 有效,次行址码 A6~A0=40~67H 有效。在此后,就可以将显示字符码写入 DD RAM 或从 DD RAM 中读出。运行时间(250 kHz)约为 40  $\mu$ s。

### 9) 读忙标志 BF 和 AC 值(Read Busy Flag and Address Count)

RS、 $R/\overline{W}$ =01,DATA=BF AC6 AC5 AC4 AC3 AC2 AC1 AC0。该指令读取 BF 及 AC。BF 为内部操作忙标志,BF=1,忙;BF=0,不忙。AC6~AC0 为地址计数器 AC 的值。当 BF=0 时,送到 DB6~DB0 的数据(AC6~AC0)有效。

### 10) 写数据到 DDRAM 或 CGRAM(Write Data to DDRAM or CG RAM)

RS、 $R/\overline{W}$ =10,DATA=实际数据。该指令根据最近设置的地址,将数据写入 DD RAM 或 CG RAM 中。实际上,数据被直接写入 DR,再由内部操作写入地址指针所指的 DD RAM 或 CG RAM。运行时间(250 kHz)约为 40  $\mu$ s。

### 11) 读 DDRAM 或 CGRAM 数据 (Read Data from DDRAM or CGRAM)

RS、R/ $\overline{W}$ =11, DATA=实际数据。该指令根据最近设置的地址, 从 DD RAM 或 CGRAM 读数据到总线 DB7~DB0 上。运行时间(250 kHz)约为 40  $\mu$ s。

## 6.3.4 LCD 构件设计与测试实例

本小节给出点阵字符型 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 PT1AD03~PT1AD07、PTJ7~6 和 PTS3 连接, LCD 的控制线 RS、R/ $\overline{W}$ 、E(4、5、6 脚) 分别与 MCU 的 PTP4、PTP5 和 PTP7 连接, 图 6-14 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。

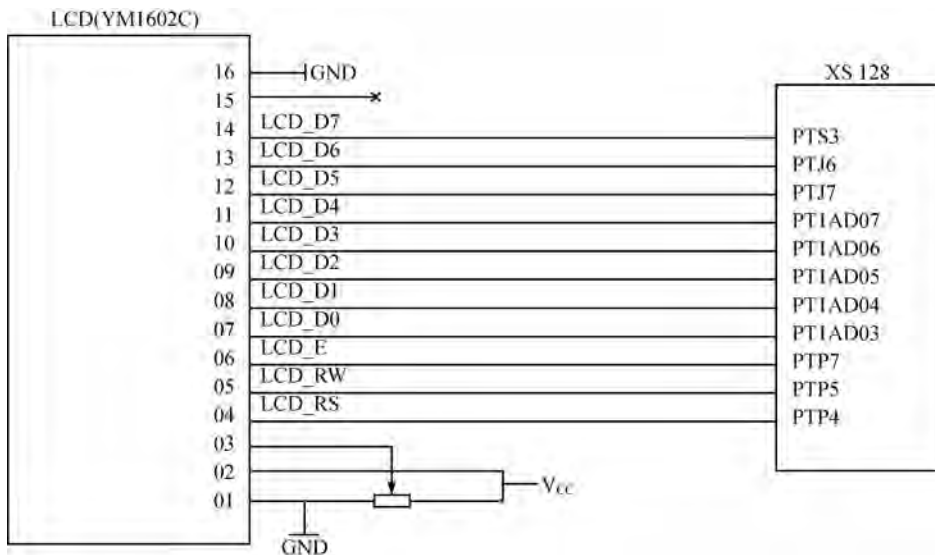


图 6-14 LCD 与 MCU 的连接

以下给出 XS128 对 LCD 的编程实例。

### 1. LCD 构件设计

#### 1) LCD 构件头文件 (LCD. h)

```
// -----*
// 文件名: LCD. h *
// 说明: LCD 驱动程序头文件 *
// -----*

#ifndef LCD_H
```



```
#define LCD_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//液晶显示寄存器及标志位定义
//LCD 数据线所用的寄存器
#define LCD_P0          PAD1
#define LCD_P1          PAD1
#define LCD_P2          PAD1
#define LCD_P3          PAD1
#define LCD_P4          PAD1
#define LCD_P5          PJ
#define LCD_P6          PJ
#define LCD_P7          PS
//LCD 数据线所用寄存器的引脚
#define LCD_B0          3
#define LCD_B1          4
#define LCD_B2          5
#define LCD_B3          6
#define LCD_B4          7
#define LCD_B5          7
#define LCD_B6          6
#define LCD_B7          3
//LCD 寄存器选择信号
#define LCDRS_P          PP
#define LCDRS_B          4
//LCD 读写信号
#define LCDRW_P          PP
#define LCDRW_B          5
//LCD 使能信号
#define LCDE_P          PP
#define LCDE_B          7

//构件函数声明区
void LCDInit(void);          //初始化 Lcd(HD44780),设置显示方式,清屏,AC 自动 + 1
void LCDShow(uint8 str[]);  //在 HD44780 显示屏显示 str 所指向的 32 个数据
```

```
void LCD_Command(uint8 cmd); //执行给定的 cmd 命令,且延时
#endif
```

## 2) LCD 构件程序文件(LCD.c)

```
//-----*
//文件名: LCD.c *
//说 明: LCD 构件函数源文件 *
//-----*

//头文件包含,及宏定义区
// 头文件包含
#include "LCD.h"

//构件函数实现
//-----*
//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;
    // 定义数据口为输出
    GPIO_Set(LCD_P0,DDR,LCD_B0,1);
    GPIO_Set(LCD_P1,DDR,LCD_B1,1);
    GPIO_Set(LCD_P2,DDR,LCD_B2,1);
    GPIO_Set(LCD_P3,DDR,LCD_B3,1);
    GPIO_Set(LCD_P4,DDR,LCD_B4,1);
    GPIO_Set(LCD_P5,DDR,LCD_B5,1);
    GPIO_Set(LCD_P6,DDR,LCD_B6,1);
    GPIO_Set(LCD_P7,DDR,LCD_B7,1);

    // 定义控制口为输出
    GPIO_Set(LCDRS_P,DDR,LCDRS_B,1);
    GPIO_Set(LCDRW_P,DDR,LCDRW_B,1);
    GPIO_Set(LCDE_P,DDR,LCDE_B,1);

    // 设置指令,RS,R/W = 00, 写指令代码
```



```
GPIO_Set(LCDRS_P,PRT,LCDRS_B,0);
GPIO_Set(LCDRW_P,PRT,LCDRW_B,0);

// 功能设置
LCD_Command(0b00111000);
    //|||||___ 可设任意值(0/1)
    //|||||___ F   = 0,5 * 7 点阵模式
    //|||||___ N   = 1,2 行显示
    //|||||___ DL  = 1,8 位数据总线
    //|||||___ 固定为 001

// 显示开关控制
LCD_Command(0b00001000);
    //|||||___ B = 0,不闪烁
    //|||||___ C = 0,关光标显示
    //|||||___ D = 0,关显示
    //|||||___ 固定为 00001

// 清屏

// 清 DD RAM 内容,光标回原位,清 AC
LCD_Command(0b00000001);

// 等待清屏完毕,时间 > 1.6ms
for (i = 0; i < 20000; i + +);

// 输入方式设置
LCD_Command(0b00000110);
    //|||||___ 显示不移动,光标左移(A = 1)
    //|||||___ 数据读写操作后,AC 自动增 1
    //|||||___ 固定为 000001

// 光标或画面移位设置
LCD_Command(0b00010100);
    //|||||___ 可设任意值(0/1)
    //|||||___ 光标右移一个字符位,AC 自动加 1
    //|||||___ 固定为 0001

// 显示开关控制
LCD_Command(0b00001100);
```

```

        //|||||||_____ B = 0,不闪烁
        //|||||||_____ C = 0,关光标显示
        //|||||||_____ D = 1,开显示
        //|||||||_____ 固定为 00001
    }

// ----- *
//函数名:Delay1 *
//功 能:用程序的方法延时一段时间 *
//参 数:k = 延时长度(0 - 65535) *
//返 回:无 *
// ----- *
void Delay1(uint16 k)
{
    uint16 i,j;
    for (i = 0; i<k; i++)
        for (j = 0; j<= 500; j++);
}

//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;
    // LCD 初始化
    LCDInit();

    // 显示第 1 行 16 个字符

    //设置显示首地址
    GPIO_Set(LCDRS_P,PRT,LCDRS_B,0); //RS,R/W = 00(写的是指令)
    GPIO_Set(LCDRW_P,PRT,LCDRW_B,0);

    LCD_Command(0x80); //后 7 位为 DD RAM 地址(0x00)

    //写 16 个数据到 DD RAM
    GPIO_Set(LCDRS_P,PRT,LCDRS_B,1); //RS,R/W = 10(写的是数据)
    GPIO_Set(LCDRW_P,PRT,LCDRW_B,0);

```



```

// 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]);
}

// 显示第 2 行 16 个字符

//设置显示首地址
GPIO_Set(LCDRS_P,PRT,LCDRS_B,0); //RS,R/W = 00(写的是指令)
GPIO_Set(LCDRW_P,PRT,LCDRW_B,0);

LCD_Command(0xC0); //后 7 位为 DD RAM 地址(0x40)

// 再写 16 个数据到 DD RAM
GPIO_Set(LCDRS_P,PRT,LCDRS_B,1); //RS,R/W = 10(写的是数据)
GPIO_Set(LCDRW_P,PRT,LCDRW_B,0);

// 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]);
}
}

//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(uint8 cmd)
{
    uint8 i;
    // 等待 > 40us
    Delay1(5);

    // 数据送到 LCD 的数据线上

    GPIO_Set(LCD_P0,PRT,LCD_B0,((cmd >> 0) & 0x01));
    GPIO_Set(LCD_P1,PRT,LCD_B1,((cmd >> 1) & 0x01));

```



```

GPIO_Set(LCD_P2,PRT,LCD_B2,((cmd >> 2) & 0x01));
GPIO_Set(LCD_P3,PRT,LCD_B3,((cmd >> 3) & 0x01));
GPIO_Set(LCD_P4,PRT,LCD_B4,((cmd >> 4) & 0x01));
GPIO_Set(LCD_P5,PRT,LCD_B5,((cmd >> 5) & 0x01));
GPIO_Set(LCD_P6,PRT,LCD_B6,((cmd >> 6) & 0x01));
GPIO_Set(LCD_P7,PRT,LCD_B7,((cmd >> 7) & 0x01));

// 给出 E 信号的下降沿,使数据写入 LCD
GPIO_Set(LCDE_P,PRT,LCDE_B,1);

for(i = 0;i<10;i + )    //延时
    asm("NOP");

// Lcd 结束接收数据
GPIO_Set(LCDE_P,PRT,LCDE_B,0);

// 等待 > 40us
Delay1(5);
}

```

## 2. LCD 构件测试实例

```

// -----*
// 工 程 名: LCD *
// 硬件连接: 串口与 PC 机相连 *
// 程序描述: (1)初始显示: *
//          "Wait Receiving.." *
//          "Soochow 2010.06" *
//          (2)MCU 接收 PC 发来的 32 个字符(只有达到 32 个字符才能显示),并送 LCD *
//          显示 *
// 目 的: 掌握 LCD 编程的基本方法 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 年 -----*

//包含头文件
#include "Includes.h"

//在此添加全局变量定义
uint8 receive_data[32];

void main()

```



```
{  
    //0.1 主程序使用的变量定义  
    uint32 mRuncount = 0;           //运行计数器  
  
    //0.2 关总中断  
    DisableInterrupt();  
  
    //0.3 芯片初始化  
    MCUInit(FBUS_32M);  
  
    //0.4 模块初始化  
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗  
    SCIIInit(0,FBUS_32M,9600);           //串口 0 初始化  
    LCDInit();  
  
    LCDShow("Wait Receiving.. Soochow 2010.06");    // LCD 显示初始化  
  
    // 主循环  
    for(;;)  
    {  
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换  
        mRuncount + + ;  
        if (mRuncount >= 10)  
        {  
            mRuncount = 0;  
            Light_Change(Light_Run_PORT,Light_Run);    //指示灯的亮、暗状态切换  
        }  
  
        // -----  
        //2. LCD 显示  
        //接收 PC 发送的 32 个数据,并存放 Data 数组中  
        if (0 == SCIReN(0,32,receive_data))           //接收成功  
        {  
            // LCD 显示 MCU 从串口接收到的 32 个数据  
            LCDShow((uint8 *)receive_data);  
        }  
        // -----  
    }  
}
```

## 定时器相关模块

本章把参考手册中与定时相关模块(PIT、通用定时器、PWM)作为一个整体内容进行阐述。主要知识点有:①计数器/定时器的工作原理;②定时器模块的基本编程方法;③定时器模块的输入捕捉、输出比较、脉冲累加功能;④脉宽调制模块 PWM;⑤周期中断定时器模块 PIT。重点是介绍定时器溢出中断处理系统时钟执行一些周期性工作及 PWM 模块的应用方法。

### 7.1 计数/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。有些设备要求每间隔一定时间开启并在一段时间后关闭,有些指示灯要求不断地闪烁,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,计数与定时问题的解决方法是一致的,只不过是同一个问题的两种表现形式。

实现计数与计时的基本方法有 3 种:完全硬件方式、完全软件方式、可编程计数/定时器。

#### (1) 完全硬件方式

在过去的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,则必须改变电路参数,通用性和灵活性差。微型电子计算机出现以后,特别是随着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的方法已较少使用。

#### (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精确的计算和测试,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 的使用效率,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,这种方法在实际应用中还是经常使用的,尤其是在已有系统上作软件开发以及延时时间较小而重复次数又较少的情况。在计算机控制软件开发过程中,作为粗略的延时,经常使用软件方法来实现定时。

### (3) 可编程计数/定时器

利用专门的可编程计数/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,综合利用了它们各自的优点,其计数/定时功能可由程序灵活地设置,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/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/定时器设置定时常数,并用指令启动计数/定时器开始计数,当计数到指定值时自动产生一个定时输出。在计数/定时器开始工作以后,CPU 不必去管它而可以去做其他工作。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/定时器产生中断信号还可以建立多作业的环境,所以可大大提高 CPU 的利用率。加上计数/定时器本身的开销并不很大,因此这种方法在微机应用系统中得到广泛地使用。

## 7.2 定时器模块的基本编程方法与实例

图 7-1 给出了定时器模块 TIM 的功能框图。表 7-1 给出了有关寄存器的地址偏移、名称和访问权限。

表 7-1 TIM 的寄存器

地址偏移	寄存器名称	访问权限
\$ 00	定时器输入捕捉/输出比较选择寄存器(TIOS)	读/写
\$ 01	定时器强制输出比较寄存器(CFORC)	读/写
\$ 02	输出比较 7 掩码寄存器(OC7M)	读/写
\$ 03	输出比较 7 数据寄存器(OC7D)	读/写
\$ 04~\$ 05	定时器计数器(TCNT)	读/写
\$ 06	定时器系统控制寄存器 1(TSCR1)	读/写
\$ 07	定时器翻转溢出寄存器(TTOV)	读/写
\$ 08~\$ 0B	定时器控制寄存器 1~4(TCTL1~TCTL4)	读/写
\$ 0C	定时器中断使能寄存器(TIE)	读/写
\$ 0D	定时器系统控制寄存器 2(TSCR2)	读/写

续表 7-1

地址偏移	寄存器名称	访问权限
\$ 0E~\$ 0F	主定时器中断标志寄存器 1~2(TFLAG1~TFLAG2)	读/写
\$ 10~\$ 1F	定时器输入捕捉/输出比较寄存器 0~7(TC0~TC7)	读/写
\$ 20	16 位脉冲累加器 A 控制寄存器(PACTL)	读/写
\$ 21	脉冲累加器 A 标志寄存器(PAFLG)	读/写
\$ 22~\$ 23	脉冲累加器计数器(PACNT)	读/写
\$ 2C	输出比较引脚断开寄存器(OCPD)	读/写
\$ 2E	精确定时器预分频选择寄存器 (PTPSR)	读/写

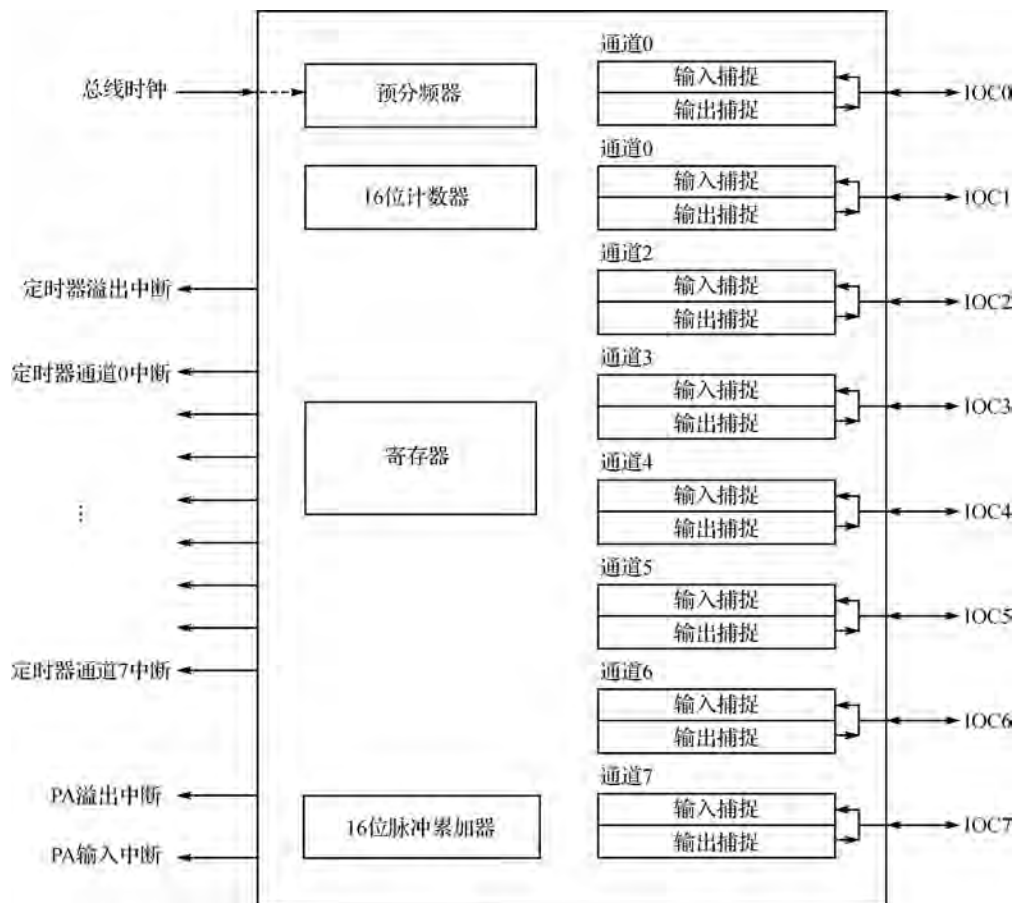


图 7-1 TIM 功能框图



## 7.2.1 定时器模块计时功能的基本寄存器

定时器模块计时功能的基本寄存器分别是定时器系统控制寄存器、定时器系统计数寄存器和定时器中断标志寄存器。这些寄存器用于设置定时器模块的时钟源、时钟分频因子、是否允许中断、标识定时器是否产生溢出等。下面分别介绍这些寄存器。

### 1. 定时器系统控制寄存器 1(Timer System Control Register 1, TSCR1)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TEN	TSWAI	TSFRZ	TFFCA	PRNT	读时为 0, 写无效		
复位	0	0	0	0	0	0	0	0

D7—TEN, 定时器使能位。TEN=1, 允许定时器正常工作。TEN=0, 禁止定时器(包括计数器), 可以降低功耗。

D6—TSWAI, 等待模式下定时器允许位。TSWAI=1, MCU 处于 WAIT 模式时, 禁止定时器, 此时不能用定时器中断的方式让 MCU 退出 WAIT 模式。TSWAI=0, MCU 处于 WAIT 模式时, 定时器继续工作。这一位也会影响脉冲累加器。

D5—TSFRZ, FREEZE 模式下定时器允许位。TSFRZ=1, 当 MCU 处于 FREEZE 模式时, 禁止定时/计数器。这对仿真非常有用。TSFRZ=0, 当 MCU 处于 FREEZE 模式时, 定时/计数器继续正常工作。TSFRZ 位对脉冲累加器无影响。

D4—TFFCA, 定时器标志快速清除位。TFFCA=1 时, 对于 TFLG1 寄存器(\$0E), 读输入捕捉通道或写输出比较通道(\$10~\$1F)将清空相应的通道标志位 CxF; 对于 TFLG2 寄存器(\$0F), 访问 TCNT 寄存器(\$04, \$05)将清空 TOF 标志位。访问 PACN3 和 PACN2 寄存器(\$22, \$23)将清空 PAFLG 寄存器(\$21)中的 PAOVF 和 PAIF 位。访问 PACN1 和 PACN0 寄存器(\$24, \$25)将清空 PBFLG 寄存器(\$31)中的 PBOVF 位, 这样做的好处是可以节省软件上的开支, 不需要单独的清 0 操作步骤。不过使用时要特别小心, 以免将其他的标志位清 0。TFFCA=0, 清空定时器标志位, 以进行正常的操作。

D3—PRNT, 高精度定时器位。PRNT=0 时, 启动传统计时器, TSCR2 寄存器 PR0、PR1 和 PR2 位用于定时器计数器预分选。PRNT=1 时, 启动紧密计时器。在 PTPSR 寄存器的所有位都用于精密定时器预分频选择。该位只有在复位后才可进行写入。

### 2. 定时器系统控制寄存器 2(Timer System Control Register 2, TSCR2)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TOI	读为 0 写无效			TCRE	PR2	PR1	PR0
复位	0	0	0	0	0	0	0	0

D7—TOI,定时器溢出中断使能位。TOI=1 时,如果 TOF 位为 1 则产生硬件中断。TOI=0,则禁止中断。

D3—TCRE,定时器计数器复位使能位。允许输出比较寄存器 7 的事件来复位定时器计数器。这种操作模式与加 1 计数模块的功能相似。TCRE=1,当 OC7 比较成功时复位计数器。否则禁止复位,定时器计数器自由运行。

如果 TC7 = \$ 0000,并且 TCRE = 1,则 TCNT 继续保持 \$ 0000 值。如果 TC7 = \$ FFFF,并且 TCRE=1,那么当 TCNT 从 \$ FFFF 复位变为 \$ 0000 时,TOF 位永远不会被置位。

D2~D0—PR2~PR0,定时器分频因子选择位。这 3 位用来设置对总线时钟的分频因子。分频因子可以为 1、2、4、8、16、32、64 或 128。PR2~PR0 的值与对应的分频因子的关系见表 7-2。

表 7-2 定时器时钟选择

PR2	PR1	PR0	定时器时钟
0	0	0	总线时钟/1
0	0	1	总线时钟/2
0	1	0	总线时钟/4
0	1	1	总线时钟/8
1	0	0	总线时钟/16
1	0	1	总线时钟/32
1	1	0	总线时钟/64
1	1	1	总线时钟/128

新设置的分频因子不会立即生效,直到所有的预置计数器值都等于 0 的下一个同步时钟沿发生时才能生效。

现举例说明要产生给定的时间间隔,应该如何计算分频因子:假定总线时钟的频率  $f_{\text{BUS}}$  为 4 MHz,希望产生  $t=1\text{ s}$  的定时间隔,设分频因子为  $p$ ,则它们的关系为: $t=n/(f_{\text{BUS}}/p)$ ,其中即  $n=2^{16}=65\,536$ ,所以分频因子  $p=t/n\times f_{\text{BUS}}=1/65\,536\times 4\,000\,000\approx 61$ ,但是分频因子只能是 2 的整数幂,我们可以选择 64 分频,这时产生的定时间隔就稍微少于 1 s。由于这个缘故,我们可以将 1 s 的定时间隔分成几次定时中断来完成。

### 3. 定时器计数寄存器(Timer Count Register, TCNT)

16 位的定时器是一个加 1 计数器。要求在一个时钟周期内访问完这个计数器。对高字节和低字节的单独进行读/写访问将产生与整体读写不一样的结果。用户随时可以读取这个寄存器的值。正常模式下写这个寄存器无意义,也不产生任何影响。仅在特殊模式(test\_



mode=1)下可写。

#### 4. 主定时器中断标志寄存器 2(Main Timer Interrupt Flag 2,TFLG2)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TOF							

D7—TOF,定时器溢出标志位。当计数器从\$FFFF计数到\$0000定时器发生溢出时置位该位。TFLG2显示什么时候发生中断。当TSCR1寄存器中的TEN位和PACTL寄存器中PAEN位都置1,对该位写1可以清除该位。

### 7.2.2 定时器构件设计与测试实例

#### 1. 定时器构件设计

定时器通过计算溢出中断次数来实现定时。它涉及分频因子 $p$ ,溢出中断间隔 $t$ 、总线频率 $f_{\text{bus}}$ 和计数寄存器的值 $n$ 的取值。将定时器系统控制寄存器的TEN位置1即 $\text{TSCR1} = \$80$ ,使得主定时器的计数功能有效;将定时器系统控制寄存器的TOI位置1即 $\text{TSCR2} | = 1 \ll 7$ ,使得定时器溢出中断有效。对于分频因子 $p$ ,这里通过设置定时器系统控制寄存器2的低3位全为1即 $\text{TSCR2} = 0b00000111$ ,将 $p$ 设置为最大值128。在此选择总线时钟频率 $f_{\text{bus}} = 32 \text{ MHz}$ ,计数器的值 $n = 2^{16} = 65535$ 。可以算得 $t$ 约为1/4秒,即定时器每隔1/4s就产生一次溢出中断,每隔1s即4次溢出中断就会转到计数函数 $\text{SecAdd1}(\text{uint8} * p)$ ,将秒加1。

##### (1) 定时器构件头文件(Timer.h)

```
// -----*
//文件名 : Timer.h *
//说明 : (1)该头文件为 Timer 构件的头文件 *
//          (2)在需要用到 Timer 的地方,直接 Include "Timer.h" *
//          (3)为了避免重复包含相同的文件,使用 *
//          #ifndef... #define... #endif 句法 *
// -----*

#ifndef TIMER_H
#define TIMER_H

//头文件包含,及宏定义区
// 头文件包含
#include "Includes.h"
```



```

//开放或禁止定时器溢出中断
#define EnableTimer      TSCR2 |= 1<<7      //开放定时器溢出中断
#define DisableTimer     TSCR2 &= ~(1<<7)    //禁止定时器溢出中断

//构件函数声明区
void TimerInit(void); //定时器初始化,中断一次时间为 1/4 秒
void SecAdd1(uint8 *p); //以秒为最小单位递增时,分,秒缓冲区的值(00:00:00 - 23:59:59)
#endif

```

## (2) 定时器构件程序文件(Timer.c)

```

//-----*
//文件名 : Timer.c *
//说 明 : 该头文件为 Timer 模块初始化及功能函数实现文件 *
//      (1)TimerInit:定时器初始化 *
//      (2)TimerUpDate:定时器更新 *
//-----*

//头文件包含,及宏定义区

//头文件包含
#include "Timer.h"

//构件函数实现
//TimerInit:定时器初始化函数-----*
//功 能:定时器初始化,中断一次时间为 1/4 秒 *
//参 数:无 *
//返 回:无 *
//-----*

void TimerInit(void)
{
    //允许主定时器开始计数
    TSCR1 = $ 80;
    //禁止定时器溢出中断,中断一次时间计算:  $t = n / (f_{bus}/p) \approx 1/4$  秒,
    //其中  $n = 65535$ ,  $f_{bus} = 32\text{MHz}$ 
    TSCR2 = 0b00000111;
    //      |||_
    //      ||__\分频因子为  $2^7 = 128$ 
    //      |___/
}

```



```
// ----- *
// 函数名: SecAdd1(计时函数) *
// 功 能: 以秒为最小单位递增时,分,秒缓冲区的值(00:00:00 - 23:59:59) *
// 参 数: *p:计数变量的首地址 *
// 返 回: 无 *
// ----- *

void SecAdd1(uint8 *p)
{
    *(p+2) += 1;                //秒加 1
    if (*(p+2) >= 60)            //秒溢出
    {
        *(p+2) = 0;            //清秒
        *(p+1) += 1;            //分加 1
        if (*(p+1) >= 60)        //分溢出
        {
            *(p+1) = 0;         //清时
            *p += 1;             //时加 1
            if (*p >= 24)         //时溢出
                *p = 0;          //清时
        }
    }
}
```

## 2. 定时器构件测试实例

在做测试的时候,将串口接 PC 机,XS128 核心板的 PA1 口接小灯。运行程序时,小灯不停地闪烁,这里主要是利用了定时器的溢出功能,在 PC 机用一个简单的程序接收单片机发送来的时、分、秒并显示,高端界面如图 7-2 所示。通过本实例,要充分理解定时器的工作原理。

```
// ----- *
//工 程 名: Timer *
//说 明: 定时器溢出中断。不停的计时加 1,并通过串口发送到 PC *
//硬件连接: 串口接 PC 机,PA1 口接小灯。 *
//目 的: 理解定时器模块的工作原理 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 ----- *

//头文件包含
#include "Includes.h"
```

```
//在此添加全局变量定义
```

```

uint8 time[3];

void main()
{
    //0.1 主函数中的变量定义
    uint8 remember;
    uint32 mRuncount = 0;                //运行计数器

    //0.2 关总中断
    DisableInterrupt();                  //禁止总中断

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    TimerInit();                        //(1)定时器 1 初始化
    SCIIInit(0,FBUS_32M,9600);          //(2)串行口初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗

    //0.5 开放中断
    EnableSCIReInt0;                    //(1)开放串口接受中断
    EnableTimer;                        //(2)开放定时器 1 溢出中断
    EnableInterrupt();                  //(3)开放总中断

    time[0] = 0;                        //(1) "时分秒"缓存初始化(00:00:00)
    time[1] = 0;
    time[2] = 0;
    remember = time[2];                 //(2) 临时变量 remember 初始化

    // 主循环
    for(;;)
    { //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + + ;
        if (mRuncount >= 50000)
        {
            mRuncount = 0;
            Light_Change(Light_Run_PORT,Light_Run); //指示灯的亮、暗状态切换
        }
    }
}

```



```

// -----
//2. 将当前的时分秒发送给 PC 机
if (time[2] != remember)
{
    SCISendN(0,3, time);          //发送当前"时分秒"
    remember = time[2];          //remember 中存放当前秒值
}
// -----
} //for_end(主循环结束)
} //main_end

```



图 7-2 定时器高端界面

## 7.3 定时器模块输入捕捉功能的编程方法与实例

输入捕捉功能是定时器模块的基本功能之一, 在学习了定时器的基本编程方法之后, 可以进一步学习定时器模块输入捕捉功能的编程方法。本节讨论 MC9S12XS128 定时器模块输入捕捉功能的编程方法。

### 7.3.1 输入捕捉的基本含义

输入捕捉相关通道详如图 7-1 所示。

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沿跳变(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沿跳变后,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沿跳变,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 7-3 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 时刻处将再次进入中断。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。



图 7-3 输入捕捉过程

## 7.3.2 输入捕捉的寄存器

### 1. 定时器输入捕捉/输出比较选择寄存器

该寄存器简称 TIOS(Timer Input Capture/Output Compare Select),用于选择相应定时器通道的是输入捕捉还是输出比较,读/写都有效,复位后全为 0。各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
复位	0	0	0	0	0	0	0	0

D7~D0—IOS[7:0]。通道配置输入捕捉或输出比较的位。清 0,相应通道开放输入捕捉功能;置 1,相应通道开放输出比较功能。

程序运行时首先清除输入/输出捕捉选择位、通道号 x。将通道 x 设置为输入捕捉通道。输入捕捉功能会捕捉外部动作发生的时间。当输入捕捉通道引脚接收到一个上升沿时,定时器会将定时器计数寄存器中的值送到定时器通道寄存器 TCx。

输入捕捉输入的最小脉宽要比两个总线时钟的时间还要长。输入捕捉通道 x 设置 CxF 位标志位。CxI 位使能标志位 CxF 产生中断请求。



## 2. 定时器控制寄存器 3/4(Timer Control Register 3/4, TCTL3/TCTL4)

### 1) 定时器控制寄存器 3(TCTL3)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A
复位	0	0	0	0	0	0	0	0

### 2) 定时器控制寄存器 4(TCTL4)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A
复位	0	0	0	0	0	0	0	0

D7~D0—EDGnB/EDGnA,输入捕捉沿跳变控制位。这 8 位控制位用于配置边缘检测电路。具体如表 7-3 所列。

表 7-3 边缘检测电路配置

EDGnB	EDGnA	功能描述
0	0	禁止输入捕捉功能
0	1	只有在沿上升时,输入捕捉才会发生
1	0	只有在下降沿时,输入捕捉才会发生
1	1	沿上升或者下降,输入捕捉都会发生

## 3. 定时器中断使能寄存器(Timer Interrupt Enable Register, TIE)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
复位	0	0	0	0	0	0	0	0

D7~D0—C7I~C0I,输入捕捉/输出比较通道 x 中断使能。TIE 中的每一位与 TFLG1 状态寄存器中的每一位相对应。当 CxI 被清零时,相应的标志位将不能产生中断,反之将会被使能产生一个中断。

### 7.3.3 输入捕捉构件设计与测试实例

#### 1. 输入捕捉构件设计

在构件设计时要特别注意输入捕捉/输出比较选择寄存器各位的设置,它决定相应的通道

被设置为输入捕捉还是输出比较。例如输入捕捉输出比较选择寄存器中的 IOS7 被置位 0, 则通道 7 开放输入捕捉功能。反之, 开放输出比较功能。设置 TCTL3 与 TCTL4 的值可以确定捕捉那种沿跳变。注意: TPMCh.c 文件中的 CHNo 表示通道号,  $TCTL4 | = \$03 \ll (CHNo * 2)$ ,  $TCTL3 | = (\$03 \ll ((CHNo - 3) * 2))$ , 表明在此示例中沿上升和沿下降都会被捕捉到。

### 1) 输入捕捉构件头文件(TPMCh.h)

```
// ----- *
//文件名 : TPMCh.h *
//说 明 : 该头文件为输入捕捉构件的头文件。 *
//          在需要用到 LCD(液晶)的地方,直接 Include "TPMCh.h" 。 *
//          为了避免重复包含相同的文件,使用 #ifndef... #define... #endif 句法 *
// ----- *
#ifndef TPMCH_H
#define TPMCH_H

//头文件包含,及宏定义
//头文件包含
#include "Includes.h"

//开放或禁止定时器通道 x 输入捕捉的宏定义
#define EnableIncapInt(x)    TIE| = (1<<x) //开放定时器通道 x 输入捕捉中断
#define DisableIncapInt(x)  TIE& = ~(1<<x) //禁止定时器通道 x 输入捕捉中断

//构件函数声明区
void TPMChInit(uint8 CHNo); //初始化,设置通道 0 为沿跳变输入捕捉

#endif
```

### 2) 输入捕捉构件程序文件(TPMCh.c)

```
// ----- *
//文件名 : TPMCh.c *
//说 明 : 该头文件为输入捕捉模块初始化及功能函数实现文件 *
//          (1)TPMChInit:定时器初始化 *
//          (2)TPMChInit:输入捕捉系统配置初始化 *
// ----- *

//头文件包含及宏定义区
// 头文件包含
```



```

#include "TPMCh.h"

//构件函数实现
//ICInit:初始化输入捕捉系统配置 ----- *
//功 能:初始化,设置通道 0 为沿跳变输入捕捉 *
//参 数:CHNo:通道号 *
//返 回:无 *
// ----- *

void TPMChInit(uint8 CHNo)
{
    if(CHNo>7) CHNo = 7;                //错误输入处理
    TSCR1 = 0;                          //禁止时钟
    BCLR(CHNo,TIOS);                    //设置通道 CHNo 为输入捕捉功能

    if(CHNo<= 3)                        //0~3 通道
        TCTL4| = $ 03<<(CHNo * 2);    //设置沿跳变捕捉
    else                                //4~7 通道
        TCTL3| = ($ 03<<((CHNo - 3) * 2)); //设置沿跳变捕捉

    BCLR(CHNo,TIE);                    //禁止通道 CHNo 输入捕捉中断
    TSCR2 = $ 06;                      //不允许溢出中断,分频因子为 64
    BSET(7,TSCR1);                     //时钟计数
}

```

## 2. 输入捕捉构件测试实例

在做测试时,将核心板上的 PA1 口接小灯,PT0 口接拨码开关,当运行程序时,可以看到小灯不停地闪烁,拨动拨码开关可以调节蜂鸣器。在做测试的过程中要充分理解输入捕捉的含义以及它的工作原理。

```

// ----- *
//工 程 名: Incapture *
//说 明: 输入捕捉中断处理,捕捉拨码开关控制小灯,蜂鸣器会响,拨动 *
//      拨码开关 可以调节小灯的暗亮和控制蜂鸣器 *
//硬件连接: PA 口 1 接小灯。PT0,即定时器 1 通道 0 接拨码开关。 *
//目 的: 理解输入捕捉的执行过程 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 ----- *

//头文件包含
#include "Includes.h"

```



```

void main()
{
    //0.1 主函数变量定义区
    uint32 mRuncount = 0;    //运行计数器

    //0.2 关总中断
    DisableInterrupt();      //禁止总中断

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT, Light_Run, Light_OFF); // (1) 小灯初始化
    Light_Init(Light_Run_PORT, 2, 1);                // (2) 小灯初始化
    TPMChInit(1);                                     // (3) 定时器通道初始化

    //0.5 开放中断
    EnableIncapInt(1);    // (1) 开放定时器通道 0 输入捕捉中断
    EnableInterrupt();    // (2) 开放总中断

    //主循环
    for(;;)
    {
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + +;
        if (mRuncount >= 50000)
        {
            mRuncount = 0;
            Light_Change(Light_Run_PORT, Light_Run); // 指示灯的亮、暗状态切换
        }
        //-----
        //2. 放置主循环中其他代码
        //-----
    }
    //for_end(主循环结束)
} //main_end

```

## 7.4 定时器模块输出比较功能的编程方法与实例

输出比较功能也是 MCU 定时器模块的基本功能之一,在学习了定时器的输入捕捉功能之后,可以进一步学习定时器模块输出比较功能的编程方法。本节简要讨论定时器模块输出

比较功能的基本编程方法。

### 7.4.1 输出比较的基本知识

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。

对比使用延时来得到所需输出电平的方法,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他中断的影响,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信。用输入捕捉作为数据输入,而用输出比较作为数据输出。首先根据通信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。

### 7.4.2 用于输出比较功能的相关寄存器

#### (1) 定时器输入捕捉/输出比较选择寄存器(Timer Input Capture/Output Compare Select)

这个寄存器的含义已经在上一节进行了介绍。

#### (2) 定时器强制输出比较寄存器

定时器强制输出比较寄存器(Timer Compare Force Register, CFORC)用于各个通道配置为强制输出比较,其地址偏移是 \$01 任何时候读为 0,写有效。各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	FOC7	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0
复位	0	0	0	0	0	0	0	0

D7~D0—FOC[7:0],通道 7~0 强制输出比较行为。对该寄存器进行写操作同时相应的数据位置位,会立即引起输出比较通道 x 行为的产生。该行为的发生如同一次成功的比较那样,除了中断标记位之外,TCx 寄存器各位不会置位。

#### (3) 输出比较 7 屏蔽寄存器

输出比较 7 屏蔽寄存器(Output Compare 7 Mask Register, OC7M)的地址偏移为 \$02,各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	OC7M7	OC7M6	OC7M5	OC7M4	OC7M3	OC7M2	OC7M1	OC7M0
复位	0	0	0	0	0	0	0	0

D7~D0—OC7M[7:0],输出比较 7 屏蔽位。在通道 7 上比较行为时,当 TTOV[7]置位,此时就像一个计数器溢出,会覆盖其他通道上的比较行为。对于每位 OC7M 的置位,输出比较行为反应相应的 OC7D 位的状态。该位为 0 时,即使相应的引脚设置为输出比较,在输出

比较 7 数据寄存器中相应的 OC7D<sub>x</sub> 位不会送到定时器端口。该位为 1 时,输出比较 7 数据寄存器中相应的 OC7D<sub>x</sub> 位会送到定时器端口。

#### (4) 输出比较 7 数据寄存器(Output Compare 7 Data Register, OC7D)

输出比较 7 数据寄存器地址偏移为 \$ 03,读/写有效定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	OC7D7	OC7D6	OC7D5	OC7D4	OC7D3	OC7D2	OC7D1	OC7D0
复位	0	0	0	0	0	0	0	0

D7~D0—OC7D[7:0],输出比较 7 数据。当 TTOV[7]置位或者在通道 7 上有一次成功的输出比较则此时如同计数器溢出,此时根据输出比较 7 屏蔽寄存器将输出比较 7 数据寄存器中的位传输到定时器数据寄存器端口。

#### (5) 定时器控制寄存器 1/2(Timer Control Register 1/2, TCTL1/TCTL2)

##### 1) 定时器控制寄存器 1(TCTL1)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
复位	0	0	0	0	0	0	0	0

##### 2) 定时器控制寄存器 2(TCTL2)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
复位	0	0	0	0	0	0	0	0

D7~D0—OM<sub>x</sub>/OL<sub>x</sub>,输出模式/输出电平位。输出比较产生时,对应输出比较功能的引脚输出指定的电平。当 OM<sub>x</sub>=1 或 OL<sub>x</sub>=1 时,引脚 x 具有输出比较功能。具体如表 7-4 所列。

表 7-4 比较产生的输出电平

OM <sub>x</sub>	OL <sub>x</sub>	动作描述
0	0	定时器输出信号但不产生输出比较动作
0	1	输出比较翻转
1	0	输出低电平
1	1	输出高电平



## 7.4.3 输出比较构件设计与测试实例

### 1. 输出比较构件设计

构件设计时首先要考虑选择哪个通道作为沿跳变输出比较。除此之外还要考虑比较产生的电平是高电平还是低电平还是翻转电平。通过设置 TCTL1 与 TCTL2 可以确定是哪一种电平。注意： $TCTL2| = 1 << (CHNo * 2)$  与  $TCTL1| = (1 << ((CHNo - 3) * 2))$ ，表明在此实例中比较产生的输出电平只能为比较翻转电平或者是低电平。

#### 1) 输出比较构件头文件(TPMCh.h)

```
// ----- *
// 文件名: TPMCh.h *
// 说明: 该头文件为 TPMCh(输出比较)的头文件。 *
//      在需要用到 TPMCh(输出比较)的地方,直接 Include "TPMCh.h"。 *
//      为了避免重复包含相同的文件,使用 #ifndef... #define... #endif 句法 *
// ----- *

#ifndef TPMCH_H
#define TPMCH_H

//头文件包含及宏定义区
//头文件包含
#include "Includes.h"

//构件函数声明区
void TPMChInit(uint8 CHNo); //初始化,设置通道 0 为沿跳变输出比较
#endif
```

#### 2) 输出比较构件程序文件(TPMCh.c)

```
// ----- *
// 文件名: TPMCh.c *
// 说明: 该头文件为串口模块初始化及功能函数实现文件 *
//      TPMChInit;输出比较初始化 *
// ----- *

//头文件包含,及宏定义区
//头文件包含
#include "TPMCh.h"
```

```

//构件函数实现
//TPMChInit:初始化输出比较系统配置 -----*
//功 能:初始化,设置通道 0 为沿跳变输出比较 *
//参 数:CHNo:通道号 *
//返 回:无 *
// -----*
void TPMChInit(uint8 CHNo)
{
    if(CHNo>7) CHNo = 7;                //错误输入处理

    TSCR1 = 0;                          //禁止时钟
    BSET(CHNo,TIOS);                    //设置通道 CHNo 为输出比较功能

    if(CHNo<= 3)                        //0~3 通道
        TCTL2|= 1<<(CHNo * 2);          //设置沿跳变输出
    else                                //4~7 通道
        TCTL1|= (1<<((CHNo - 3) * 2));  //设置沿跳变输出

    TSCR2 = $06;                        //不允许溢出中断,分频因子为 64
    BSET(7,TSCR1);                      //时钟计数
}

```

## 2. 输出比较构件测试实例

在测试时,读者要充分理解输出比较的含义。核心板的 PT1 口接小灯,同时 PA1 口接另外一个小灯。运行程序时,可以看到 PA1 口接的那个小灯不停地闪烁,PT1 口接的那个小灯根据电平变化亮暗切换。

```

// -----*
//工 程 名: Outcompare *
//说 明: 输出比较。通过输出电平翻转的电平变化控制小灯闪烁 *
//硬件连接: PT 口 1,即定时器 1 通道 1 接小灯 *
//目 的: 理解输出比较的含义 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 -----*

```

```

//头文件包含
#include "Includes.h"

```

```

void main()
{

```



```
//0.1 数函数变量定义区
uint32 mRuncount = 0;                                //运行计数器
//0.2 关总中断
DisableInterrupt();                                  //禁止总中断
//0.3 芯片初始化
MCUInit(FBUS_32M);
//0.4 模块初始化
TPMChInit(1);                                         //定时器通道初始化
Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
//0.5 开放中断
EnableInterrupt();                                   //开放总中断
//主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount>= 50000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT,Light_Run); //指示灯的亮、暗状态切换
    }
    //-----
    //2. 在此放置主循环中的其他代码
    //-----
} //for_end(主循环结束)
} //main_end
```

## 7.5 定时器模块脉冲累加功能的编程方法与实例

### 7.5.1 脉冲累加的基本知识

#### (1) 脉冲累加功能的相关引脚

定时器模块提供了一个 16 位的脉冲累加器,该脉冲累加器与定时器的通道 7 共用引脚,外部脉冲输入引脚为 PT7。

#### (2) 脉冲累加的含义

脉冲累加器的输入通道有效沿有两种形式:一种是用于事件计数方式的极性沿,另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的跳变沿就会使得脉冲累加计数器的值加 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。

输入引脚上每产生一个有效的沿跳变就会使得脉冲累加计数寄存器的值加 1,PACTL 中的 PEDGE 位用来选择沿跳变,可以选择下降沿,也可以选择上升沿。脉冲累加计数寄存器保存了上一次复位时 PACNT 输入的有效输入沿的个数。当累加器的值从 \$FFFF 翻转到 \$0000 时,PAOVF 位被置位。如果使能脉冲累加器溢出中断(PAOVI=1),PAOVF 位产生中断。

### 7.5.2 脉冲累加功能的相关寄存器

与定时器模块的脉冲累加功能相关的寄存器主要有 3 个:脉冲累加控制寄存器、脉冲累加标志寄存器、脉冲累加计数寄存器。

#### 1. 脉冲累加控制寄存器(Pulse Accumulator Control Register, PACTL)

该寄存器用于配置脉冲累加相关功能,第 7 位为保留位,读为 0,写无效。复位后寄存器各位默认为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义		PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI

当 PAEN 设置完成时,脉冲累加控制寄存器才有效,并且它与输入引脚共用 IOC7。

D6—PAEN,脉冲累加使能位。0:禁止 16 位脉冲累加器功能;1:使能脉冲累加功能。

D5—PAMOD,脉冲累加器模式选择位。0:选择事件计数模式;1:选择门控时间累加模式。

D4—PEDGE,脉冲累加有效沿控制位。只有当脉冲累加使能能位为 1 时,该位才有效。在时间计数模式下,若该位为 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。在门控时间累加模式下,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 PA 计数器输入时钟 CLK[1:0]。

- 该两位为 00 时,输入时钟为 PA 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/65 536。

D1—PAOVI,脉冲累加溢出中断使能位。0:中断被禁止;1: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。0:中断被禁止;1: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(Pulse Accumulator Flag Register ,PAFLG)

寄存器的 D2~D7 位为保留位,读为 0,写无效。复位后寄存器各位默认为 0。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义							PAOVF	PAIF

D7~D2—未定义。

D1—PAOVF,脉冲累加溢出标志位。当 16 位器计数从 \$FFFF 到 \$0000 后,该位被置位,此时,如果脉冲累加控制寄存器 PACTL 寄存器中的 PAOVI 位为 1,那么会产生一个中断请求,再向该位写 1 将清零该位。

D0—PAIF,脉冲累加输入沿标志位。当检测到脉冲累加设置的有效电平时,该位被置位。如果脉冲累加控制寄存器 PACNT 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(Pulse Accumulators Count Registers,PACNT)

该寄存器保存上次复位检测到的脉冲累加所设置的有效电平次数。当计数器从 0xFFFF 计数到 0x0000,PACNT 发生溢出,中断标志位 PAVOF 会置位。注意:在脉冲累加输入引脚一个有效沿之后立刻读脉冲累加计数寄存器,可能会丢失最后一个数,因为输入跟总线时钟是异步的。

### 7.5.3 脉冲累加器构件设计

定时器的脉冲累加器的构件设计比较难理解,它也用了溢出中断,这里的溢出中断与定时器溢出中断有些不同。它有一个专门的累加计数寄存器 PACNT,当捕捉到一个有效的沿跳变时它就会自动加 1,当达到一定值时,就会自动产生溢出中断,此时 PACNT 中的值赋给一个变量,通过串口发送到 PC 机。PACTL=0b1011001 为脉冲累加控制寄存器中有效位的定义,PAEN 为 1 表示脉冲累加功能有效,PAMOD 为 0 表示选择时间计数模式,PEDGE 为 1 表示上升沿捕捉,接下来的两位为 0 表明输入的时钟为 PA 的预分频时钟。

#### 1) 脉冲累加构件头文件(PA.h)

```
// ----- *
//文件名: PA.h *
//说 明: 该头文件为 PA(累加器)的头文件 *
//      在需要用到 PA(累加器)的地方,直接 Include "PA.h" *
```



```
//          为了避免重复包含相同的文件,使用 #ifndef... #define... #endif 句法      *
// -----*

#ifndef PA_H
#define PA_H

    //头文件包含,及宏定义区

    //头文件包含
    #include "Includes.h"

    //构件函数声明区
    void PAInit(void); //初始化,设置通道 0 为沿跳变输入捕捉
#endif
```

## 2) 脉冲累加器构件程序文件(PA.c)

```
// -----*
//文件名: PA.c *
//说 明: 该头文件为串口模块初始化及功能函数实现文件。 *
//          PAInit:输入捕捉初始化 *
// -----*

//头文件包含,及宏定义区

//头文件包含
#include "PA.h"

//构件函数实现
//PAInit:初始化输入捕捉系统配置 -----*
//功 能:初始化,设置通道 0 为沿跳变输入捕捉 *
//参 数:无 *
//返 回:无 *
// -----*
void PAInit(void)
{
    //设置脉冲累加控制寄存器
    PACTL = 0b1011001;
    //    |||  __禁止脉冲输入中断
    //    |||  __禁止脉冲累加器溢出中断
```



```

//      |||_____上升沿捕捉
//      ||_____事件计数方式
//      |_____启动脉冲累加系统
//清脉冲累加寄存器
PACNT =  $ 0000;
}

```

### 3) 脉冲累加器构件中断程序文件(isr.c)

此处特别注意中断向量表的变化。

```

// ----- *
//文件名: isr.c *
//说 明: 该文件为中断处理文件,用于写中断服务程序 *
//      在需要的中断位置开启中断并书写相应的中断服务程序 *
// ----- *

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//变量定义区
uint8 TimInterCount = 0;

#pragma CODE_SEG __NEAR_SEG NON_BANKED

//中断服务程序区

//未定义的中断处理函数,本函数不能删除,默认
__interrupt void isr_default(void)
{
    DisableInterrupt();
    //此处放置中断代码
    EnableInterrupt();
}

//脉冲累加器外部事件输入中断处理程序
__interrupt void isrPAIE(void)
{
    uint16 tmp;

```

```

DisableInterrupt();           //关总中断
tmp = PACNT;                 //读脉冲累加器的值
SCISend1(0,(uint8)(tmp>>8)); //通过串口发送
SCISend1(0,(uint8)tmp);
PAFLG |= (1<<0);            //清空标志位
EnableInterrupt();           //开总中断

}

#pragma CODE_SEG DEFAULT

//中断矢量表对应区

//中断处理子程序类型定义
typedef void (* near tIsrFunc)(void);
//中断矢量表,如果需要定义其他中断函数,请修改下表中的相应项目
const tIsrFunc _InterruptVectorTable[] @0xFF10 = {
    /* ISR name                No.    Address  Pri Name                */
    isr_default,               /* 0x08  0xFF10  -   ivVsi                */
    ...
    isrPAIE,                   /* 0x6D  0xFFDA  1   ivVtimpaie            */
    ...
    isr_default                /* 0x7C  0xFFF8  -   ivVtrap                */
};

```

## 7.6 脉宽调制模块

脉宽调制器(Pulse Width Modulator,PWM)是嵌入式应用系统常用功能之一,本节在讨论 PWM 基本原理基础上,给出编程实例。

### 7.6.1 PWM 工作原理

PWM 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,也叫脉宽调制波。通过指定所需的时钟周期和占空比来控制高电平和低电平的持续时间。通常定义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,方波的占空比是 50%。脉冲宽度是指脉冲处于高电平的时间。图 7-4 给出了几个不同占空比的示意图。这里给出 PWM 信号的周期是 8 个时钟周期  $T_{\text{PWM}}=8T_{\text{CLK}}$ ,图 7-4(a)中,PWM 的高电平为  $2T_{\text{CLK}}$ ,所以占空比  $=2/8=25\%$ ,图 7-4(b)、7-4(c)可以类似计算。

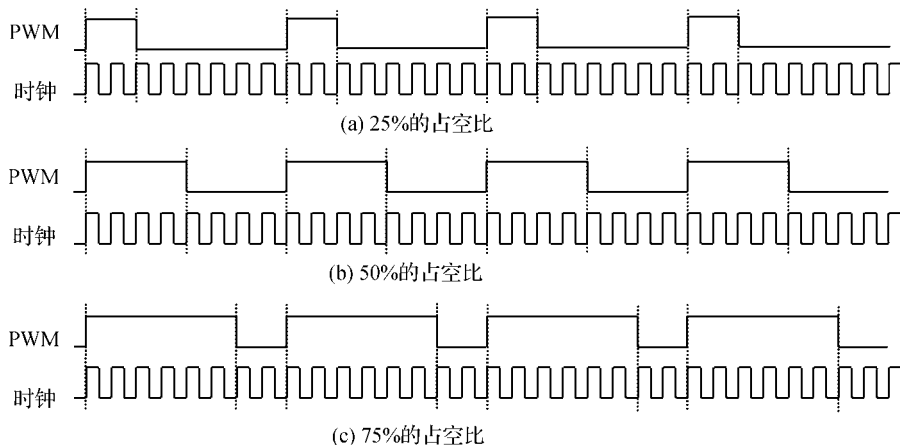


图 7-4 PWM 的占空比的计算方法

PWM 的常见应用是为其他设备产生类似于时钟的信号。例如,PWM 可用来控制灯以一定频率闪烁。

PWM 的另一个常见用途是控制输入到某个设备的平均电流或电压。例如,一个直流电机在输入电压时会转动,而转速与平均输入电压的大小成正比。假设每分钟转速(rpm)=输入电压的 100 倍,如果转速要达到 125 rpm,则需要 1.25 V 的平均输入电压;如果转速要达到 250 rpm,则需要 2.50 V 的平均输入电压。在图 7-4 中,如果逻辑 1 是 5 V,逻辑 0 是 0 V,则(a)的平均电压是 1.25 V,(b)的平均电压是 2.5 V,(c)的平均电压是 3.75 V。可见,利用 PWM 可以设置适当的占空比值来得到所需的平均电压,如果所设置的周期足够小,电机就可以平稳运转(即不会明显感觉到电机在加速或减速)。

PWM 的另一种应用是控制命令字编码。例如,通过发送不同宽度的脉冲,代表不同含义。假如用此来控制无线遥控车,宽度 1 ms 代表左转命令,4 ms 代表右转命令,8 ms 代表前进命令。接收端可以使用定时器来测量脉冲宽度,在脉冲开始时启动定时器,脉冲结束时停止定时器,由此来确定所经过的时间,从而判断收到的命令。

## 7.6.2 XS128 的 PWM 的特点及模块框图

### (1) XS128 的 PWM 特点

8 个独立的带有可编程周期和占空比的 PWM 通道;每个 PWM 通道都有一个独立的计数器;采用软件方式为每个通道选择 PWM 脉冲极性;周期和占空比都为双缓冲;每个通道都带有可编程中心对齐或者左对齐输出;8 个 8 位通道或者 4 个 16 位通道的 PWM 分辨率;4 个时钟源(A、B、SA、SB)提供宽范围的频率选择;可编程的时钟选择逻辑;紧急关闭功能。

## (2) XS128 的 PWM 模块框图

PWM 框图如 7-5 所示。PWM 会产生一个在高电平和低电平之间重复交替的输入信号,这个信号称为 PWM 信号,通常定义的 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占整个信号周期的百分比。

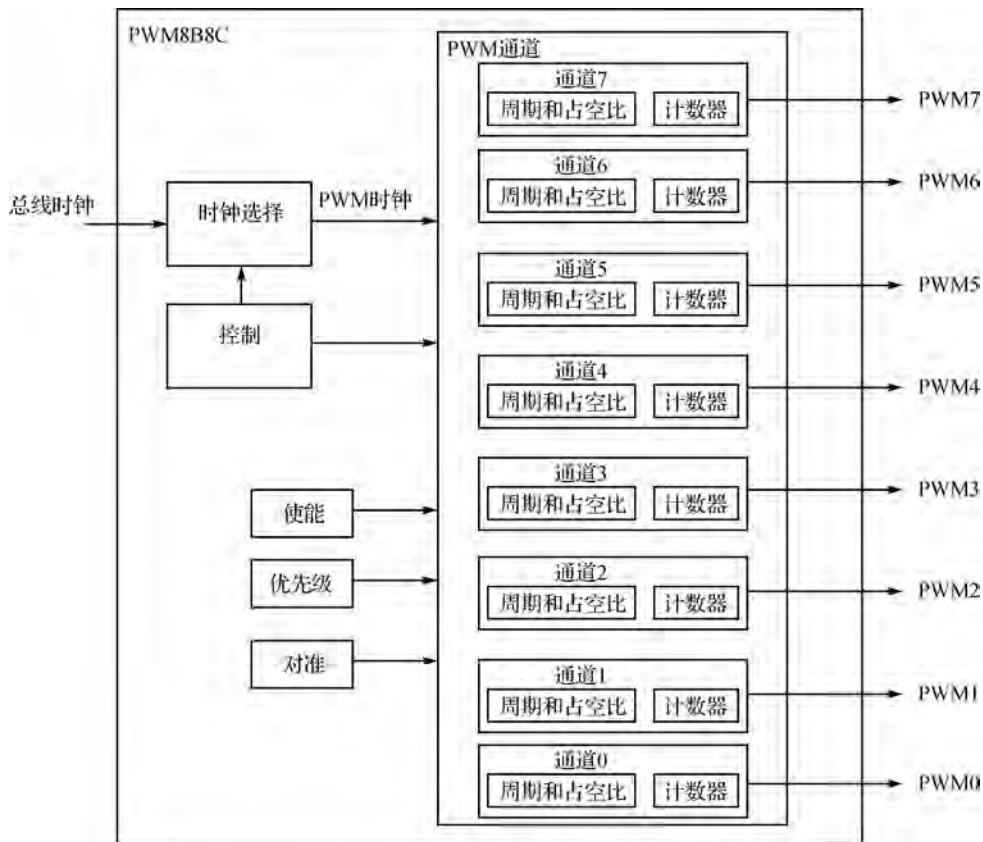


图 7-5 PWM 模块框图

## 7.6.3 脉宽调制模块 PWM 相关寄存器

MC9S12XS128 的脉宽调制模块 PWM 有以下相关寄存器,下面将依次介绍这些寄存器。

### (1) PWM 使能寄存器(PWM Enable Register, 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<sub>n</sub>=1,使能 PWM 通道 n 输出;PWME<sub>n</sub>=0,禁止 PWM 通道 n 输出。其中 n=0~7。

### (2) PWM 极性寄存器(PWM Polarity Register, 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<sub>n</sub>=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<sub>n</sub>=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(PWM Clock Select Register, 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<sub>n</sub>=1,PWM 通道 n 选择 SB 为时钟源;PCLK<sub>n</sub>=0,PWM 通道 n 选择 B 为时钟源。其中 n=2、3、6、7。

PCLK<sub>n</sub>=1,PWM 通道 n 选择 SA 为时钟源;PCLK<sub>n</sub>=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(PWM Prescale Clock Select Register, 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		PCKB				PCKA		

时钟 A 与时钟 B 的分频设置如下式:

$$f_A = f_{\text{Bus}} \div 2^{\text{PCKA}} \quad f_B = f_{\text{Bus}} \div 2^{\text{PCKB}} \quad (\text{式 } 7-1)$$

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

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(PWM Center Align Enable Register, 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(PWM Control Register, 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 通道占空比寄存器(PWM Channel Duty Registers, 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 PMW 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:

$$\text{PWM 信号占空比} = (1 - \text{占空比寄存器值} \div \text{周期寄存器值}) \times 100\%$$



## (8) PWM 关闭寄存器(PWM Shutdown Register, PWMSDN)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PWMIF	PWMIE	PWMRSTRT	PWMLVL		PWM7IN	PWM7INL	PWM7ENA
复位	0	0	0	0	0	0	0	0

该寄存器提供了在紧急情况下关闭 PWM 模块的功能。复位后该寄存器全为 0。

D7—PWMIF, 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 0 该位, 写 0 无效。

D6—PWMIE, 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—PWMRESTART, 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—PWMLVL, 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—PWM7INL, 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—PWM7ENA, 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。

## 7.6.4 PWM 构件设计及测试实例

### 1. PWM 构件设计

该部分构件设计涉及的寄存器比较多, 但总的来说是围绕通道(channel)、周期(period)和占空比(duty)来设置的。void PWMEnable(uint8 channel)使能相应的通道, 相应通道设置为 0 时表明禁止通道输出, 相反允许相应通道输出, void PWMSetting(uint8 channel, uint8 period, uint8 duty)函数设置相应的周期和占空比。

#### 1) PWM 构件头文件(PWM.h)

```
// ----- *
// 文件名: PWM.h *
// 说明: 该头文件为 PWM 构件的头文件 *
//      在需要用到 PWM 的地方, 直接 Include "PWM.h" *
//      为了避免重复包含相同的文件, 使用 #ifndef... #define... #endif 句法 *
// ----- *
```



```
#ifndef PWM_H
#define PWM_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//构件函数声明区
void PWMInit(uint8 channel); //初始化 PWM
void PWMSetting(uint8 channel, uint8 period, uint8 duty); //根据参数设置周期和占空比
void PWMEnable(uint8 channel); //根据参数设置相应通道 PWM 有效
void PWMDisable(uint8 channel); //根据参数设置相应通道 PWM 无效
#endif
```

## 2) PWM 构件程序文件(PWM.c)

```
//-----*
// 文件名: PWM.c *
// 说 明: 该头文件为 PWM 模块初始化及功能函数实现文件 *
//      (1)PWMInit: PWM 初始化 *
//      (2)PWMSetting: 设置周期和占空比 *
//      (3)PWMEnable: 使能 PWM *
//      (4)PWMDisable: 禁止 PWM *
//-----*
//头文件包含,及宏定义区

//头文件包含
#include "PWM.h"

//构件函数实现
//PWMInit: 初始化输入捕捉系统配置 -----*
//功 能: 设置通道的 PWM *
//参 数: channel: 所要设置的通道号(0~7) *
//返 回: 无 *
//-----*
void PWMInit(uint8 channel)
{
    //参数越界处理
```



```

    if (channel < 0)    channel = 0;
    if (channel > 7)    channel = 7;
    //(1) 禁止 PWM
    PWMDisable(channel);
    //(2) 设置 A,B 的时钟频率

    PWMPRCLK = 0b00000000;
    //      ||| |||_
    //      ||| |||_ \时钟 A 频率为 Fbus/2^0
    //      ||| |___/
    //      |||_
    //      ||_ \时钟 B 频率为 Fbus/2^0
    //      |___/

    //(3) PWM 时钟源选择,选择 A 时钟作为 PWM 通道 0 的时钟源
    PWMCLK&= ~(1<<channel);
    //(4) 设置对齐方式
    PWMCAE&= ~(1<<channel);
    //(5) 设置极性
    PWMPOL|= 1<<channel;                                //正极性
}

//PWMSetting:PWM 周期和占空比设置 -----*
//功 能:根据参数设置 f 周期和占空比                                     *
//参 数:period-PWM 周期所占用的时钟周期个数                           *
//      duty-PWM 占空比所占用的时钟周期个数                           *
//      channel:所要设置的通道号(0~7)                                  *
//返 回:无                                                                *
//说 明:duty 的值<= period 的值,并且两者的值都在 0~65535 之间        *
//-----*

void PWMSetting(uint8 channel, uint8 period, uint8 duty)
{
    // 参数越界处理
    if (channel < 0)    channel = 0;
    if (channel > 7)    channel = 7;
    //使相应通道的 PWM 无效
    PWMDisable(channel);
    switch(channel)
    {
        case 0:

```

```
//该路为 8 位
//设置占空比寄存器
PWMDTY0 = (uint8)duty;
//设置周期寄存器
PWMPER0 = (uint8)period;
//清 0 通道 X 计数器
PWMCNT0 = $ 00;
break;

case 1:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY1 = (uint8)duty;
    //设置周期寄存器
    PWMPER1 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT1 = $ 00;
    break;

case 2:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY2 = (uint8)duty;
    //设置周期寄存器
    PWMPER2 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT2 = $ 00;
    break;

case 3:
    //设置占空比寄存器
    PWMDTY3 = (uint8)duty;
    //设置周期寄存器
    PWMPER3 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT3 = $ 00;
    break;

case 4:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY4 = (uint8)duty;
    //设置周期寄存器
```



```

        PWMPER4 = (uint8)period;
        //清 0 通道 X 计数器
        PWMCNT4 = $ 00;
        break;
case 5:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY5 = (uint8)duty;
    //设置周期寄存器
    PWMPER5 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT5 = $ 00;
    break;
case 6:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY6 = (uint8)duty;
    //设置周期寄存器
    PWMPER6 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT6 = $ 00;
    break;
case 7:
    //该路为 8 位
    //设置占空比寄存器
    PWMDTY7 = (uint8)duty;
    //设置周期寄存器
    PWMPER7 = (uint8)period;
    //清 0 通道 X 计数器
    PWMCNT7 = $ 00;
    break;
default:
    break;
}
PWMPEnable(channel);
}
//PWMPEnable,PWM 通道有效 ----- *
//功 能:根据参数设置相应通道 PWM 有效 *
//参 数:channel:所要设置的通道号(0~7) *
```

```

//返回:无
//-----*
void PWMEnable(uint8 channel)
{
    //参数越界处理
    if (channel < 0)    channel = 0;
    if (channel > 7)    channel = 7;
    //设置相应的通道 PWM 有效
    if (channel < 8)
        PWME |= (1<<channel);
    else if (channel == 8)
        PWME |= (1<<1);
}

//PWMDisable;PWM 通道无效-----*
//功 能:根据参数设置相应通道 PWM 无效
//参 数:channel;所要设置的通道号(0~7)
//返回:无
//-----*
void PWMDisable(uint8 channel)
{
    //参数越界处理
    if (channel < 0)    channel = 0;
    if (channel > 7)    channel = 7;
    //设置相应的通道 PWM 无效
    if (channel < 8)
        PWME &= ~(1<<channel);
    else if (channel == 8)
        PWME &= ~(1<<1);
} //设置相应的通道 PWM 无效
    if (channel < 8)
        PWME &= ~(1<<channel);
    else if (channel == 8)
        PWME &= ~(1<<1);
}

```

## 2. PWM 构件测试实例

要求: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。PWM 波的周期要根据驱动的器件的特性设置,不能太长,也不能太短。



```
// ----- *
//工 程 名: PWM *
//说 明: PWM 波控制小灯。通过调节占空比使小灯亮度发生变化 *
//硬件连接: PP 口 1 接小灯,通过调节实时延的大小来控制小灯由暗变亮的时间。 *
//目 的: 理解 PWM 的工作原理 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 -----*

//头文件包含
#include "Includes.h"

void main()
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;    //运行计数器
    uint8 period,duty;

    //0.2 关总中断
    DisableInterrupt();      //禁止总中断

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    PWMInit(1);              //初始化 PWM 通道 1
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
    period = 0xFF;           //PWM 周期
    duty = 0x00;             //PWM 占空比

    //主循环
    for(;;)
    {
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + + ;
        if (mRuncount >= 50)
        {
            mRuncount = 0;
            Light_Change(Light_Run_PORT,Light_Run); //指示灯的亮、暗状态切换
        }
    }
}
```

```

// -----
//2. 通过占空比的变化来调节小灯的亮度
if(duty >= period) duty = 0x00;
PWMSetting(1,period, duty + );           //不断增加 PWM 的占空比
Delay(10);                               //延时
// -----
} //for_end(主循环结束)
} //main_end

```

## 7.7 周期中断定时器模块 PIT

周期中断定时器(Periodic Interrupt Timer,PIT)是一个 24 位阵列定时器,用于产生周期性中断或触发外设模块。PIT 模块没有外部引脚,具有以下特点:4 路定时器可以按不同的定时周期同时进行工作,并可分别产生中断信号;4 路定时器可分别被打开或关闭;4 路定时器都是 24 位定时器;定时周期可在  $1 \sim 2^{24}$  个总线周期之间选择。

### 7.7.1 PIT 模块功能描述

PIT 模块框图如图 7-6 所示,主要包括有关的控制寄存器、状态和数据寄存器以及 2 个 8 位递减式计数器、4 个 16 位递减式数器和一个中断/触发接口。

#### 1. PIT 的基本组成

PIT 模块 4 路 24 位定时器实际上是由两个 8 位微计数器和 4 个 16 位计数器构成。4 路定时器共享两个 8 位微计数器,通过写 PIT 复用寄存器,可以为每路定时器分配一个 8 位计数器。4 路定时器都各自有独立的 16 位计数器,定时器通道 0、1、2、3 分别使用 16 位计数器 0、1、2、3。这 6 个计数器都是递减式计数器,每个计数器都有一个对应的计数器加载寄存器和一个强制加载控制位。

要使能某一路定时器,不仅要写 PIT 通道寄存器来打开该路定时器,还需要写 PIT 控制寄存器来使能 PIT 模块。假设微计数器 1 被分配给定时器通道 0,那么当定时器通道 0 被使能的时候,微计数器 1 对应的加载寄存器的值也会自动加载到微计数器 1 中,16 位计数器 0 那个的加载寄存器的值也会自动加载到 16 位计数器 0 中,然后按照 MCU 内部总线时钟,微计数器 1 将不断递减,当减为 0 的时候,微计数器 1 对应的加载寄存器的值将会被重新加载到微计数器 1 中,同时,会让 16 位计数器 0 做减 1 操作。当计数器 1 和 16 位计数器 0 都为 0 的时候,16 位计数器 0 对应的加载寄存器的值会被重新加载到 16 位计数器 0 中,同时,产生超时信号,如果定时器通道 0 的定时中断被使能,就会产生定时中断。当某个定时通道使用的微计数器和 16 位计数器分别对应的加载寄存器的值为  $N$  和  $M$ ,总线时钟频率为  $f_{\text{BUS}}$  可以按以



图 7-6 PIT 模块框图

下公式计算该定时器通道的定时周期。

$$\text{Time-out-period} = (M+1) \times N+1) / f_{\text{BUS}}$$

例如总线时钟频率为 40 MHz, 则最大定时周期为:

$$256 \times 65\,536 \times 25\text{ ns} = 419.43\text{ ms}$$

当前 16 位模数倒数计数器中的值不能通过 PITCNT 寄存器读出。微定时器倒数计数器中的值不能被读出。

## 2. 中断接口

可以使用任何一个定时器溢出事件来产生一个中断服务请求。对每个定时器通道来说,在 PIT 中断使能寄存器(PITINTE)中都存在一个独立的位(PINTE)来使能上述特点。如果置位 PINTE,不管 PIT 溢出标志寄存器(PITTF)中的溢出标志 PTF 是否被置位,都产生一个中断服务请求。通过写 1 到 PTF 可以将该位清 0。

### 3. 硬件触发功能

PIT 模块有 4 个硬件触发信号线 PITRIG[3:0], 分别对应 4 个定时器通道。当达到定时器通道溢出时间, 置位对应的 PTF 标志, 并且相应的触发信号 PITTRIG 寄存器输出一个上升沿。该操作至少需要 2 个总线时钟周期。因为触发维持高电平至少需要一个总线时钟周



期。本书不再阐述这方面功能。

## 7.7.2 PIT 模块的编程寄存器

### (1) PIT 控制与强制加载微定时寄存器(PIT Control and Force Load Micro Timer Register)

PIT 控制与强制加载微定时寄存器简称 PITCFLMT,地址偏移为 \$00,各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PITE	PITSWAI	PITFRZ				PFLMT1	PFLMT0
复位	0	0	0	0	0	0	0	0

D7—PITE,PIT 模块允许位。0 为禁止 PIT 模块,1 为允许 PIT 模块。

D6—PITSWAI,等待模式 PIT 停止控制位。0 为 PIT 在等待模式下正常运行。1 为 PIT 在等待模式下时钟发生停止或冻结。

D5—PITFRZ,冻结模式 PIT 计数器冻结控制位。在调试中遇到断点时,许多情况下为避免中断等事件的产生,常常用来冻结 PIT 计数器。0:在冻结模式下 PIT 计数器正常运行。1:在冻结模式下 PIT 计数器失效。

D4~D2 未定义。

D1~D0—PFLMT[1:0],PIT 强制加载微计数器数值控制位。只有当某个微计数器处于工作状态并且 PIT 模块被使能(PITE=1)的时候,微计数器的强制加载位才是有效的,此时,向该微计数器的强制加载位写 1,会使对应的加载寄存器中的值被加载到该微计数器中;若写 0,则无效;这两位读为 0。

### (2) PIT 强制加载寄存器(PIT Force Load Timer Register,PITFLT)

地址偏移为 \$01,各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					PFLT3	PFLT2	PFLT1	PFLT0
复位	0	0	0	0	0	0	0	0

读写都有效。D7~D4 未定义。

D3~D0—PFLT[3:0],PIT 定时器 3—0 强制加载位。当相应定时通道和 PIT 模块都有效时这些位才有效。此时,向该计数器的强制加载位写 1,会使对应的加载寄存器中的值被加载到 16 位定时倒数计数器中;写 0,则无效。对这 4 位读为 0。

### (3) PIT 通道使能寄存器(PIT Channel Enable Register ,PITCE)

地址偏移为 \$02,各位定义如下:



数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					PCE3	PCE2	PCE1	PCE0
复位	0	0	0	0	0	0	0	0

读写都有效。复位时各位被清 0。

D7~D4 未定义。

D3~D0—PCE[3 : 0], PIT 使能控制位。该寄存器使能 PIT 的 4 个通道, 如果 PCE<sub>n</sub> 写 0, PIT<sub>n</sub> 通道将被禁止, 在 PITTF 中相应的标志位也将被清除; 如果 PCE<sub>n</sub> 写 1, PIT<sub>n</sub> 通道使能, 16 位定时器计数器被加载起始值并开始计数。0: 通信 PIT 通道被禁用。1: 通信 PIT 通道被启用。

#### (4) PIT 多路选择寄存器(PIT Multiplex Register, PITMUX)

地址偏移为 \$03, 各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					PMUX3	PMUX2	PMUX1	PMUX0
复位	0	0	0	0	0	0	0	0

读写都有效。复位时各位被清 0。

D7~D4 未定义。

D3~D0—PMUX[3 : 0], PIT 定时通道多路选择寄存器控制位。该寄存器控制选择 16 位定时器与 8 位微定时器时基 0 或者时基 1 的连接。当 PMUX<sub>n</sub> 被置 1 时, 第 n 路定时器使用微计数器 1; 当 PMUX<sub>n</sub> 位被清 0 时, 第 n 路定时器使用微计数器 0。0: 相应 16 位定时器与微定时器时基 0 连接; 1: 相应 16 位定时器与微定时器时基 1 连接。

#### (5) PIT 中断使能寄存器(PIT Interrupt Enable Register, PITINTE)

地址偏移为 \$04, 各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					PINTE3	PINTE2	PINTE1	PINTE0
复位	0	0	0	0	0	0	0	0

读写都有效。复位时各位被清 0。

D7~D4 未定义。

D3~D0—PINTE[3 : 0], PIT 定时器溢出中断使能控制位。当 PINTE<sub>n</sub> 被置 1 时, 第 n 路定时器的定时中断被使能, 此时, 只要第 n 路定时器的溢出标志位 PTF<sub>n</sub> 被置 1, 就会产生中断请求; 当 PMUX<sub>n</sub> 被清零时, 第 n 路定时器的定时中断被禁止。0: PIT 相应通道溢出中断被禁止, 1: PIT 相应通道溢出中断使能。

## (6) PIT 定时溢出标志寄存器(PIT Time - Out Flag Register, PITTF)

地址偏移为 \$05,各位定义如下:复位时各位被清 0。读写都有效。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义					PTF3	PTF2	PTF1	PTF0
复位	0	0	0	0	0	0	0	0

读写都有效。复位时各位被清 0。

D7~D4 未定义。

D3~D0—PTF[3:0],PIT 通道超时标志位。当 16 位定时器计数时和 8 位微定时器计数器递减到 0 时,相应位的标志位被置 1。写 1 清除标志位,写 0 无效。0:PIT 相应通道没有发生溢出中断,1:PIT 相应通道发生了溢出中断。

## (7) PIT 微定时器加载寄存器 0~1(PIT Micro Timer Load Register 0 to 1,PITMTLD0~1)

### 1) PIT 微定时器加载寄存器 0

地址偏移为 \$06,各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PMTLD7	PMTLD6	PMTLD5	PMTLD4	PMTLD3	PMTLD2	PMTLD1	PMTLD0
复位	0	0	0	0	0	0	0	0

### 2) PIT 微定时器加载寄存器 1

地址偏移为 \$07,各位定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PMTLD7	PMTLD6	PMTLD5	PMTLD4	PMTLD3	PMTLD2	PMTLD1	PMTLD0
复位	0	0	0	0	0	0	0	0

读写都有效。复位时各位被清 0。

D7~D0—PMTLD[7:0],PIT 微定时器初值加载位。该寄存器用于设置 PIT 模块中的 8 位微计数器初值。设定值为 0~255。递减计数器减至 0 后会自动重载。

## (8) PIT 加载寄存器 0~3(PIT Load Register 0 to 3 ,PITLD0~3)

读写都有效。寄存器的第 15 位到 0 位分别对应 PLD15~PLD0[15:0],PIT 加载位,这些位设置递减模数计数器的初始值,为确保数据的一致性,写入到 PITLD 寄存器的一个新值必须是 16 位数据。该寄存器写入数据时,它不会马上更新。等到计时器计数减到 0 时,PTF 超时标志会被重新设置,寄存器中的值才被加载。如果想要马上装入计数寄存器,可以在 PITFLT 寄存器中的相应位写 1。



### (9) PIT 计数寄存器 0~3(PIT Count Register 0 to 3, PITCH0~3)

读写都有效。寄存器的第 15 位到 0 位分别对应 PCNT15~PCNT0[15:0], PIT 计数位, 这些位反映递减计数器的计数值。读取该 16 位的计数值必须在一个时钟周期内完成。

## 7.7.3 PIT 构件设计与测试实例

### 1. PIT 构件设计

PIT 构件设计与定时器构件设计类似, 原理基本是一样的。不同的地方是 PIT 的计数器是递减操作。当某个定时通道使用的微计数器和 16 位计数器分别对应的加载寄存器的值分别为 PITMTLD、PITLD, 它们都递减到 0 的时候可以计算 PIT 一次中断的时间。

#### 1) PIT 构件头文件(PIT.h)

```
// ----- *
// 文件名 : PIT.h *
// 说 明 : (1)该头文件为 PIT 构件的头文件 *
//          (2)在需要用到 PIT 的地方,直接 Include "PIT.h" *
//          (3)为了避免重复包含相同的文件,使用 *
//          #ifndef... #define... #endif 句法 *
// ----- *

#ifndef PIT_H
#define PIT_H

//头文件包含,及宏定义区

// 头文件包含
#include "Includes.h"

//开放或禁止 PIT0 溢出中断
#define EnablePIT0 PITINTE |= 1<<0 //开放定时器溢出中断
#define DisablePIT0 PITINTE &= ~(1<<0) //禁止定时器溢出中断

//构件函数声明区
void PITInit(void); //定时器初始化函数
void SecAdd1(uint8 * p); //以秒为最小单位递增时,分,秒缓冲区的值(00:00:00 - 23:59:59)
#endif
```

#### 2) PIT 构件程序文件(PIT.c)

```
// ----- *
// 文件名 : PIT.c *
```

```
// 说明：该头文件为 PIT(定时器)模块初始化及功能函数实现文件 *
//          (1)PITInit: 定时器初始化 *
//          (2)SecAdd1:定时器更新 *
// ----- *
```

//头文件包含,及宏定义区

//头文件包含

```
#include "PIT.h"
```

//构件函数实现

```
//PITInit:定时器初始化函数 ----- *
//功 能:定时器初始化,中断一次时间为 1/38 秒 *
//参 数:无 *
//返 回:无 *
// ----- *
```

```
void PITInit(void)
{
    //禁止定时器
    PITCFLMT&= ~(1<<7);
    // 使能 PIT 通道 0
    PITCE|= 1<<0;
    // 选用 8 位模寄存器 0 产生的计数基准
    PITMUX&= ~(1<<0);
    // 定时器一次中断时间 = (PITMTLD + 1) * (PITLD + 1) / fBUS
    //                      = ( $ F4 + 1) * ( $ FFFF + 1) / 32MHz ≈ 0.5s
    PITMTLD0 = $ F4; //8 位模寄存器
    PITLD0 = $ FFFF; //16 位模寄存器
    // 使能定时器
    PITCFLMT|= 1<<7;
    // 清通道 0 溢出标志,加载新的计时时间
    PITTF|= 1<<0;
    // 禁止 PIT 通道 0 中断
    PITINTE&= ~(1<<0);
}

// ----- *
```

//函数名: SecAdd1(计时函数) \*

//功 能: 以秒为最小单位递增时,分,秒缓冲区的值(00:00:00 - 23:59:59) \*

//参 数: \* p:计数变量的首地址 \*



```

//返 回：无 *
//-----*
void SecAdd1(uint8 *p)
{
    *(p+2) += 1; //秒加 1
    if (*(p+2) >= 60) //秒溢出
    {
        *(p+2) = 0; //清秒
        *(p+1) += 1; //分加 1
        if (*(p+1) >= 60) //分溢出
        {
            *(p+1) = 0; //清时
            *p += 1; //时加 1
            if (*p >= 24) //时溢出
            {
                *p = 0; //清时
            }
        }
    }
}

```

## 2. PIT 构件测试实例

在做测试的时候单片机的串口接 PC 机,核心板的 PA1 口接小灯,运行程序可以看到指示灯不停地闪烁,PC 机可以接收到 PIT 定时器当前时、分、秒的值。

```

//-----*
// 工 程 名 : PIT *
// 说 明 : PIT 中断计时。使用 SCIO 和 PC 机通信。并每隔 1s 将计时加 1 发送给 PC。 *
// 硬件连接 : 串口接 PC 机 *
// 程序描述 : 无 *
// 目 的 : 理解 PIT 工作原理 *
//-----苏州大学飞思卡尔嵌入式系统研发中心 2011-----*

//头文件包含
#include "Includes.h"
// 在此添加全局变量定义
uint8 time[3];
void main()
{
    //0.1 主函数变量定义
    uint8 remember;
}

```

```

uint32 mRuncount = 0;           //运行计数器
//0.2 关总中断
DisableInterrupt();             //禁止总中断
//0.3 芯片初始化
MCUInit(FBUS_32M);
//0.4 模块初始化
PITInit();                      // (1) 定时器 1 初始化
SCIInit(0, FBUS_32M, 9600);     // (2) 串行口初始化
Light_Init(Light_Run_PORT, Light_Run, Light_OFF); //RUN 指示灯初始化为暗
//0.5 开放中断
EnableSCIReInt0;                // (1) 开放串口接受中断
EnablePIT0;                     // (2) 开放定时器 1 溢出中断
EnableInterrupt();              // (3) 开放总中断
time[0] = 0;                    // (1) "时分秒"缓存初始化(00:00:00)
time[1] = 0;
time[2] = 0;
remember = time[2];             // (2) 临时变量 remember 初始化
//主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + +;
    if (mRuncount >= 50000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT, Light_Run); //指示灯的亮、暗状态切换
    }
    // -----
    //2. 将 PC 机与串口相连可以看到接受到的当前的时分秒的值
    if (time[2] != remember)
    {
        SCISendN(0, 3, time); //发送当前"时分秒"
        remember = time[2];    //remember 中存放当前秒值
    }
    // -----
} //for_end(主循环结束)
} //main_end

```

# 第 8 章

## A/D 与 SPI

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。

SPI(串行外设接口)是 Freescale 公司推出的一种同步串行通信接口,用于 MCU 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的带有 SPI 接口的芯片,主要用于外扩 RAM、EEPROM、FLASH、A/D、D/A 转换、LED/LCD 驱动、I/O 接口、实时时钟等。

本章主要知识点有:①A/D 转换的基础知识;②XS128 内部 A/D 转换模块的编程寄存器以及相应的编程方法和构件化编程实例;③SPI 的基本工作原理、XS128 的 SPI 模块寄存器以及相应的编程方法和构件化编程实例。本章重点为 A/D 和 SPI 构件化编程方法以及 SPI 模块中时钟极性和时钟相位的理解与设置方法。

### 8.1 A/D 通用知识

#### 8.1.1 A/D 的基本问题

A/D 转换模块(Analog To Digital Convert Module)即模/数转换模块,功能是将电压信号转换为相应的数字信号。数字控制系统框图如图 8-1 所示。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



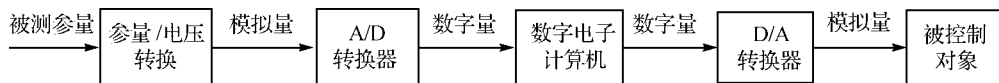


图 8-1 数字控制系统框图

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 XS128 为例，其采样精度最高为 12 位，若参考电压为 5 V，则能检测到的模拟量变化为  $5/2^{12} = 1.22 \text{ mV}$ 。

## (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 中要花费大约 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  $25.1^\circ\text{C}$ ，则 A/D 值 126 就代表实际温度  $25.1^\circ\text{C}$ 。

# 8.1.2 A/D 转换器

A/D 转换器实现转换可以使用几种不同技术。下面简单介绍两种常用的方法及其特性。

## (1) 积分型 A/D 转换器

积分式的 A/D 转换器先以一个固定长的时间对一个电容器连续充电，然后以固定的放电速率使这个电容器放电，并测试放电过程所花的时间。用时钟测量放电时间，而时钟脉冲的数量即为数字输出。这种方法相对较慢，但极其精确并具有线性关系。这种方法常用在脉冲幅度分析器。

## (2) 逐次逼近型 A/D 转换器

逐次逼近是一般微控制器应用中最常用的技术，它的转换速度中等（标称 20 s）。逐次逼近型 A/D 转换器采用二分查找法逐位确定转换后输出的数据。这种方法成本低，速度快，能处理较大的数，但转换过程中，需要采样保持放大电路提供恒定的输入。

### 8.1.3 A/D 转换常用传感器简介

传感器是指把物理量或化学量转变成电信号的器件,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。下面介绍一些简单的传感器。

#### (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也随之变化。图 8-2(a) 是 NTC 热敏电阻器。

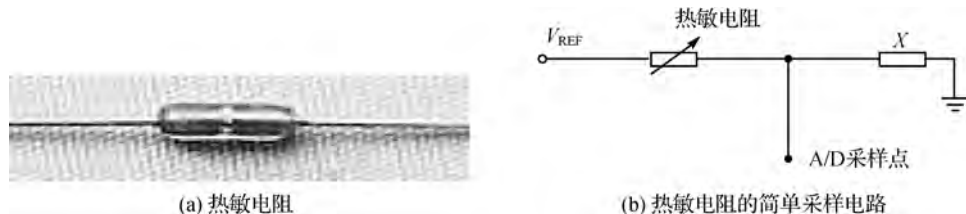


图 8-2 热敏电阻及其采样电路

在实际应用中,将热敏电阻接入图 8-2(b)的采样电路中,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) 光敏电阻器

光敏电阻器是利用半导体的光电效应制成的一种电阻值随入射光的强弱而改变的电阻器;入射光强,电阻减小,入射光弱,电阻增大。光敏电阻器一般用于光的测量、光的控制和光

电转换(将光的变化转换为电的变化)。

通常,光敏电阻器都制成薄片结构,以便吸收更多的光能。当它受到光的照射时,半导体片(光敏层)内就激发出电子—空穴对,参与导电,使电路中电流增强。一般光敏电阻器结构如图 8-3(a)所示。

图 8-3(b)给出了简单的光敏电阻采样电路,其 A/D 采样点的电压的计算方法类似于上述热敏电阻 A/D 采样点电压的计算。

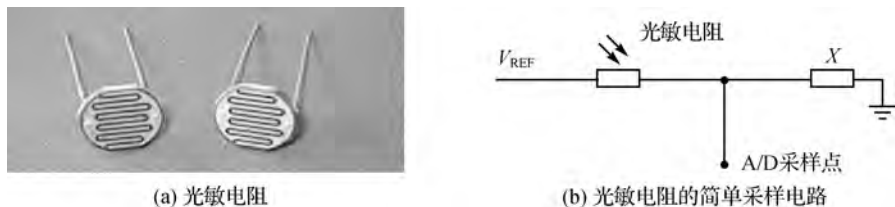


图 8-3 光敏电阻及其采样电路

### (3) 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  $\Omega$ 。这样就能检测到物体的颜色的灰度了。

## 8.1.4 电阻型传感器采样电路设计

如前所述,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 A/D 转换芯片得到电压值,最后利用 A/D 值和外部信号的对照表得出当前外部信号的值。8.1.3 小节中给出了简单的采样电路,实际应用中,为了获取更精确的采样值,常用的采样设计有恒流驱动电路和恒压驱动电路。

在恒流驱动电路中,用恒定电流驱动传感器,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 A/D 转换测电压之前,需先对电压值进行运算放大。最后,建立 A/D 值同外部温度的对照表,在获得电压的 A/D 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 A/D 值,而特征点之间的 A/D 值通过线性插值得出。

通常,电阻型传感器采集电路由 3 部分组成:传感器接口、恒流源电路和放大电路。



图 8-4 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 1.25 V 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(100 $\mu$ F),目的是过滤低频信号。在电源输入端加入电容 RG\_C3(1.5 nF),目的是过滤高频信号。

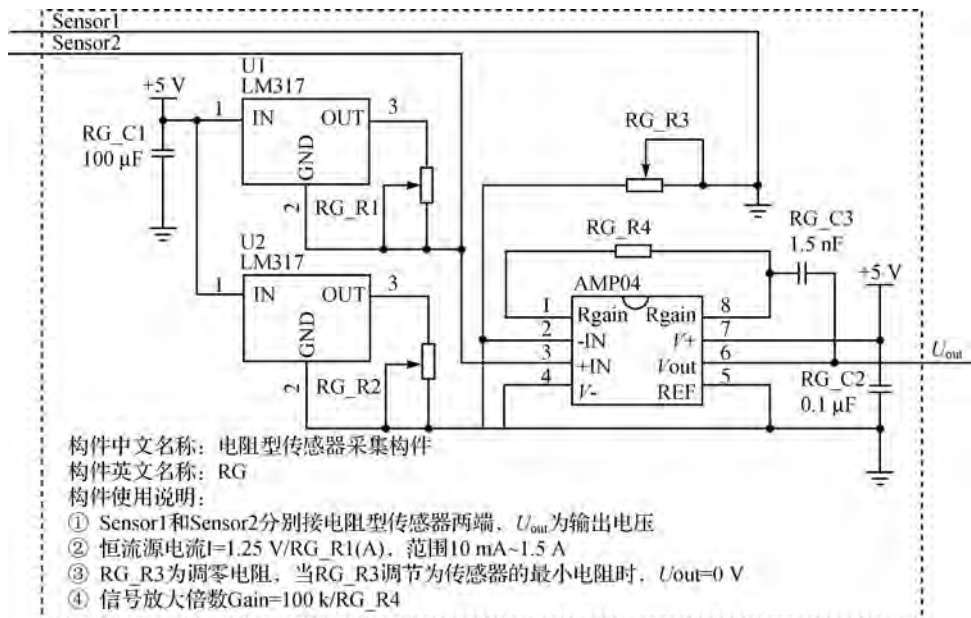


图 8-4 电阻型传感器通用采集电路

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  $Gain=100K/RG\_R4$ ,输出电压  $U_{out} = ((+IN) - (-IN)) \times Gain$ 。

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 5~10 mA,电阻变化范围 10~20  $\Omega$ ,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 10 mA,因此  $RG\_R1 = 1.25 \text{ V} / 10 \text{ mA} = 125 \Omega$ 。根据公式  $Gain = 100 \text{ k}\Omega / RG\_R4$ ,放大倍数为 50,则  $RG\_R4 = 100 \text{ k}\Omega / 50 = 2 \text{ k}\Omega$ 。

## 8.2 A/D 模块的编程寄存器

MC9S12XS128 MCU 的 16 路 12 位 A/D 转换模块引脚为 AN15/PAD15/～AN0/PAD0，每个引脚都可以实现模拟量(AN<sub>x</sub>)/数字量(PAD<sub>x</sub>)的复用。为了与外部信号同步进行 A/D 转换，A/D 模块具有 4 个外部触发转换通道 ETRIG3～ETRIG0，用户可以选定触发的方式(沿触发或电平触发)。这里讨论 16 路 A/D 转换器的编程方法。

A/D 转换有个参考电压问题，在使用到 A/D 转换器的输入引脚时，芯片的 V<sub>RH</sub>、V<sub>RL</sub> 引脚必须分别接参考电压的正、负，通常就是接电源的正、负端。VDDA、VSSA 引脚作为 A/D 转换器的电源提供引脚，分别接电源的正、负端。

MC9S12XS128 的 A/D 转换模块有 27 个寄存器，包括 6 个 A/D 控制寄存器(ATDCTL0～ATDCTL5)、2 个 A/D 状态寄存器(ATDSTAT0，ATDSTAT2)、1 个 A/D 比较使能寄存器(ATDCMPE)、1 个 A/D 输入使能寄存器(ATDDIEN)、1 个 A/D 比较方式寄存器(ATDCMPHT)、16 个 A/D 转换结果寄存器(ATDDR0～ATDDR15)。其中 ATDCTL0～ATDCTL5 和 ATDSTAT0 为 8 位寄存器，其余的寄存器都是 16 位寄存器。通过对这些寄存器的编程，就可以获取 A/D 转换数据。当不进行 A/D 转换的时候，A/D 模块自动处于电源关闭状态。下面对各寄存器进行介绍。

### 1. A/D 控制寄存器

#### 1) A/D 控制寄存器 0(ATDCTL0)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	预留	0	0	0	WRAP3	WRAP2	WRAP1	WRAP0
写								
复位	0	0	0	0	1	1	1	1

WRAP3～WRAP0：回绕通道选择位。在任何时候都可以进行读/写操作，这些位只有在 ATDCTL5 的 MULT 为 1 的情况下，也就是在多通道转换模式下，才有效。WRAP3～WRAP0 决定选择哪些通道，WRAP3～WRAP0—0001～1111，对应通道 AN1～AN15，当 WRAP3～WRAP0 为 0000 时，通道预留。WRA[3：0]=x(1≤x≤15)时，在多通道转换模式下，当完成对第 x 个模拟输入通道(AN<sub>x</sub>)的 A/D 转换后，下一个 A/D 转换通道将回绕到第 0 个通道 AN0，而不是 x+1 个通道。



## 2) A/D 控制寄存器 1(ATDCTL1)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	ETRIGESL	SRES1	SRES0	SMP_DIS	ETRIGCH3	ETRIGCH2	ETRIGCH1	ETRIGCH0
写								
复位	0	0	1	0	1	1	1	1

D7—ETRIGSEL 位,外部触发源选择,该位选择任一个 A/D 通道或 ETRIG3~0 输入之一作为外部触发源。ETRIGSEL 与 ETRIGCH3~ETRIGCH0 的值分别对应 00000~01111 时选择 AN0~AN15,10000~10011 时选择 ETRIG0~ETRIG3,其他情况预留。由于 S12XS 系列 MCU 没有引出外部触发输入引脚 ETRIG0~ETRIG3,此位无效。

D6~D5—SRES1~SRES0 位,A/D 采样精度选择位,这些位决定 A/D 转换结果的采样精度。当 SRES1~SRES0 为 00 时,8 位采样精度;当 SRES1~SRES0 为 01 时,10 位采样精度;当 SRES1~SRES0 为 10 时,12 位采样精度;当 SRES1~SRES0 为 11 时,预留。

D4—SMP\_DIS 位,采样前放电控制位。当 SMP\_DIS=0 时,不放电。当 SMP\_DIS=1 时,放电,采样时间增加两个 A/D 时钟周期,这样可以用来检测一个开路,而不是测试先前的采样通道。

D3~D0—ETRIGCH3~ETRIGCH0 位,外部触发通道选择,这些位选择其中一个 A/D 通道或一个 ETRIG3~0 输入为外部触发源,见 D7 位的描述。

## 3) A/D 控制寄存器 2(ATDCTL2)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	0	AFFC	ICLKSTP	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ACMPIE
写								
复位	0	0	0	0	0	0	0	0

D6—AFFC 位,A/D 快速清所有标志。AFFC=0,通过向各自的 CCF[n]标志清 A/D 标志。AFFC=1,当该标志位为 1 时,只要读结果寄存器,MCU 就会自动把 A/D 转换的完成标志位 CCF[n]清 0。

D5—ICLKSTP 位,停止模式中内部时钟位,该位在停止模式中使能 A/D 转换。ICLKSTP=0,如果在进入停止模式时,A/D 转换时序还在进行,那么将会暂停该时序,当退出停止模式时,暂停的时序自动重启。ICLKSTP=1,在停止模式中,A/D 转换继续,使用内部产生的时钟(ICLK)。

D4~D3—ETRIGLE~ETRIGP 位,外部触发信号触发条件选择位。当这两位为 00 时表示下降沿触发;为 01 时表示上升沿触发;10 时为低电平触发;11 时为高电平触发。

D2—ETRIGE 位,外部触发信号使能位。当该位为 1 时使能外部触发信号。为 0 时禁止



外部触发信号。在使能外部触发信号时,由 ETRIGCH[3:0]选择外部触发信号通道,由 ETRIGLE 和 ETRIGP 选择触发条件。

D1—ASCIE 位,A/D 转换完成中断使能位。当 ASCIF=1 时,A/D 转换结束时使能中断,也就是说当 SCF=1 时,将引发中断。ASCIE=0 时,A/D 转换结束不会引发中断。

D0—ACMPIE 位,A/D 比较中断使能位。如果 ACMPIE=1,使能 A/D 比较中断请求。如果 ACMPIE=0,禁用 A/D 比较中断请求。在使能比较中断时,如果一个 A/D 转换序列中的第 n 次 A/D 转换序列所对应的 CCF[n]和 CMPE[n]都为 1 时,将引发中断。

**4) A/D 控制寄存器 3(ATDCTL3)**

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	DJM	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
写								
复位	0	0	1	0	0	0	0	0

D7—DJM 位,结果寄存器数据对齐位。如果 DJM=0,左对齐,如果 DJM=1,右对齐。

D6~D3—SxC 位(x=8、4、2、1),A/D 转换序列的长度选择位。这些位控制每一个 A/D 转换序列的长度。每个序列的 A/D 转换个数由 S8C、S4C、S2C、S1C 这 4 位决定,当 S8C、S4C、S2C、S1C=0001~1111,每个序列的 A/D 转换个数分别为 1~15;当 S8C、S4C、S2C、S1C=0000 时,每个序列的 A/D 转换个数为 16。

D2—FIFO 位,结果寄存器先进先出模式位。当该位为 0(非先进先出模式)时,根据转换序列的先后顺序将转换结果放入结果寄存器中。即第一次转换的结果放在第一个结果寄存器中,第二次转换的结果放在第二个结果寄存器中,依次类推。当该位为 1(先进先出模式)时,A/D 转换的结果会依次顺延地放到结果寄存器中。当使用完最后一个结果寄存器后,会重新回到第一个结果寄存器循环使用。ATDSTAT0 中的 CC[3:0]位为循环计数,表示当前 A/D 转换结果将放到哪一个寄存器中。

D1~D0—FRZ1~FRZ0 位,背景调试冻结使能位。在进行背景调试时,若需要断点,则 MCU 进入冻结模式。在冻结模式下,当 D1~D0 为 00 时 A/D 转换继续工作;为 01 时为保留值;为 10 时完成当前的转换后进入冻结模式;为 11 时立即进入冻结模式。

**5) A/D 控制寄存器 4(ATDCTL4)**

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	SMP2	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
写								
复位	0	0	0	0	0	1	0	1

D7~D5—SMP2~SMP0 位,采样时间选择位。这 3 位用来定义采样时间包含的 A/D 转



换时钟周期的个数。A/D 转换的时钟周期可以通过预分频因子的值来确定 (PRS4 - 0 位)。

表 8-1 为采样时间选择列表。

表 8-1 采样时间选择

SMP2	SMP1	SMP0	采样时间 (A/D 时钟周期个数)
0	0	0	4
0	0	1	6
0	1	0	8
0	1	1	10
1	0	0	12
1	0	1	16
1	1	0	20
1	1	1	24

D4~D0—PRS4~PRS0 位, A/D 转换时钟预分频因子。这 5 位用来设定 A/D 转换的时钟频率。设 A/D 转换的时钟周期为  $f_{\text{ADclock}}$ , 总线周期为  $f_{\text{BusClock}}$ , A/D 转换的预分频因子为 PRS, 则计算公式如下:

$$f_{\text{ADCLK}} = f_{\text{BusClock}} / (2 \times (\text{PRS} + 1))$$

**注意:** 设置预分频因子时, 应使 ATDCLK 不小于 0.25 MHz, 同时不大于 8.3 MHz。

#### 6) A/D 控制寄存器 5 (ATDCTL5)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	0	SC	SCAN	MULT	CD	CC	CB	CA
写								
复位	0	0	0	0	0	0	0	0

D6—SC 位, 特殊通道转换位, 通过使用 CD、CC、CB 与 CA 来选择特殊的通道。如果 SC=1, 使能特殊通道, 当 CD、CC、CB、CA 的值为 0000~1111 时, 分别选择通道 AN0~AN15, 如果 SC=0 禁用特殊通道。

D5—SCAN 位, A/D 连续转换序列模式。该标志位决定 A/D 转换序列是执行一次还是持续执行。为 1 表示持续进行转换; 为 0 表示只转换一次。

D4—MULT 位, 多通道采样模式位。当 MULT 为 0 时, A/D 转换控制器只从特定的通道采样模拟数据。通道号由 ATDCTL5 中的 CD/CC/CB/CA 位决定。当 MULT 为 1 时, A/D 控制器从多个通道中采样数据, 采样的通道个数由序列长度选择位 S8C、S4C、S2C 和 S1C 来决定。其中第一个通道由 CD/CC/CB/CA 位决定。



D3~D0—CD、CC、CB、CA 位,模拟量输入通道选择位。这 4 位用来预设 A/D 转换的模拟量输入通道。

## 2. A/D 状态寄存器

### 1) A/D 状态寄存器 0(ATDSTAT0)

该寄存器包含转换完成标志、外部触发中断溢出标志、先入先出模式和转换计数器。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	SCF	0	ETORF	FIFOR	CC3	CC2	CC1	CC0
写								
复位	0	0	0	0	0	0	0	0

D7—SCF 位,转换序列完成标志位。在一次转换序列完成后置该标志位。如果转换序列是连续进行的(SCAN=1),那么在每一次转换序列完成后都会置该标志位。为 1 表示转换已完成,0 表示转换尚未完成。当下面 3 种情况出现时,该标志位会被清 0。

- 向 SCF 位写 1。
- 写 ATDCTL5 寄存器(开始一次新的转换序列)。
- 当 AFFC 为 1 时,读结果寄存器。

D5—ETORF 位,外部边沿触发溢出标志位。在边沿触发模式(ETRIGLE=0),如果 A/D 转换过程中探测到外部的边沿触发信号时,该标志位被置 1;否则该位为 0。当下面 3 种情况出现时,该标志位会被清 0:

- 向 ETORF 位写 1。
- 写 ATDCTL2/ATDCTL3/ATDCTL4 寄存器(终止转换序列)。
- 写 ATDCTL5 寄存器(开始新的转换序列)。

D4—FIFOR,先入先出溢出标志位。当一个结果寄存器对应的 A/D 转换完成标志(CCF)还没有被清 0 时,新的转换结果又写入该结果寄存器。就会置 FIFOR 位,表示发生溢出。当出现以下任何情况时,该标志位会被清 0:

- 向 FIFOR 位写 1。
- 写 ATDCTL0~ ATDCTL4、ATDCMPE、ATDCMPHT 寄存器
- 写 ATDCTL5 寄存器

D3~D0—CC3~CC0,转换计数器(Conversion Counter)。这 4 位为只读位,表示当前转换的结果将要写入的结果寄存器的编号。例如:CC3、CC2、CC1、CC0=0110 表示当前转换结果放在第 6 个结果寄存器中。如果不在先进先出模式中,则在转换开始和结束时,该计数都被初始化为 0。如果在先进先出模式中,转换计数不被初始化,当达到最大值时,该标志位又回到最小值。不论是否在先进先出模式下,终止 A/D 转换序列或开始新的 A/D 转换序列都会



将转换计数清 0。

## 2) A/D 状态寄存器 2(ATDSTAT2)

该寄存器为只读寄存器,包含一些转换完成标志位。

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	CCF[15 : 0]															
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

D15~D0—CCF[15 : 0]位,转换或比较完成标志位,当 CCF[n]为 1 时,如果比较功能被禁止(CMPE[n]=0),表示转换序列中的第 n(n=15 : 0)次转换完成,结果放在第 n 个结果寄存器中;如果比较功能被禁止(CMPE[n]=1),表示对转换序列中的第 n 次转换结果进行比较的结果为真,由于结果寄存器存储的是比较阈值,A/D 转换结果则丢失。当 CCF[n]为 0 时,表示转换未未完成或比较不成功。出现以下任何情况,标志位 CMPE[n]会被清 0:

- 写 ATDCTL5 寄存器;
- 当 AFFC=0 时,向 CCF[n]写 1;
- 当 AFFC=1 并且 CMPE[n]=0 时,读结果寄存器 ATDDRn;
- 当 AFFC=1 并且 CMPE[n]=1 时,写结果寄存器 ATDDRn。

## 3. A/D 比较使能寄存器 1(ATDCMPE)

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	CMPE[15 : 0]															
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

D15~D0—CMPE[15 : 0]位,比较使能位。当 CMPE[n]=1 时,对一个转换序列中的第 n 次 A/D 转换的结果进行比较。当 CMPE[n]=0 时,不做比较。要对一个转换序列中的第 n 次转换结果进行比较,除了置(CMPE[n])为 1,还需要把比较阈值写入第 n 个结果寄存器 ATDDRn,以及通过写 ATDCPMHT 寄存器的 CMPHT[n]位来选择比较方法。

## 4. A/D 输入使能寄存器(ATDDIEN)

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	IEN[15 : 0]															
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

D15~D0—IEN<sub>x</sub>,通道 x 的数字输入使能标志位(x=15~0)。通用输入输出端口 A/D 口与 A/D 模块的模拟输入输出引脚复用。如果要把某个模拟输入输出引脚作为通用输入输出引脚使用,必须置对应的 IEN 位。当 IEN[x]为 1 时,使能 AN<sub>x</sub> 引脚上的数字输入缓冲器,当 IEN[x]为 0 时,关闭 AN<sub>x</sub> 引脚上的数字输入缓冲器。

### 5. A/D 比较方式寄存器(ATDCMPHT)

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	CMPH[15 : 0]															
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

D15~D0—CMPHT<sub>x</sub> 位,比较方式选择位(n=15~0)。当 CMPHT<sub>x</sub> 为 1 时,则对应的比较使能位 CMPE[n]被置 1,如果转换序列中的第 n 次转换结果大于 ATDDR<sub>n</sub> 中的比较阈值,CCF[n]被置位,表示比较结果为真。当 CMPHT<sub>x</sub> 为 0 时,且对应的比较使能位 CMPE[n] 被置 1,果转换序列中的第 n 次转换结果小于或等于 ATDDR<sub>n</sub> 中的比较阈值,CCF[n]被置位,表示比较结果为真。

### 6. A/D 转换结果寄存器(ATDDRn)

A/D 转换的结果存放在 16 个结果寄存器中。转换的结果总是以无符号形式表示。因此,首先需要将结果寄存器中数据进行左对齐或者右对齐以取得正确的数据。这是由 AT-DCTL3 中的 DJM 控制位决定的。

#### 1) 左对齐(DJM=0)

模块基址+

0x0010 = ATDDR0, 0x0012 = ATDDR1, 0x0014 = ATDDR2, 0x0016 = ATDDR3  
 0x0018 = ATDDR4, 0x001A = ATDDR5, 0x001C = ATDDR6, 0x001E = ATDDR7  
 0x0020 = ATDDR8, 0x0022 = ATDDR9, 0x0024 = ATDDR10, 0x0026 = ATD-  
 DR11

0x0028 = ATDDR12, 0x002A = ATDDR13, 0x002C = ATDDR14, 0x002E = ATD-  
 DR15

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	位 11	位 10	位 9	位 8	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	0	0	0	0
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



## 2) 右对齐(DJM=1)

模块基址 +

0x0010 = ATDDR0, 0x0012 = ATDDR1, 0x0014 = ATDDR2, 0x0016 = ATDDR3

0x0018 = ATDDR4, 0x001A = ATDDR5, 0x001C = ATDDR6, 0x001E = ATDDR7

0x0020 = ATDDR8, 0x0022 = ATDDR9, 0x0024 = ATDDR10, 0x0026 = ATDDR11

0x0028 = ATDDR12, 0x002A = ATDDR13, 0x002C = ATDDR14, 0x002E = ATDDR15

数据位	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
读	0	0	0	0	位 11	位 10	位 9	位 8	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
写																
复位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

当关闭比较寄存器操作时,A/D 转换结果会被写入结果寄存器中。A/D 状态寄存器 0 中的 CC3~CC0 为转换计数,表示当前转换的结果将要写入的寄存器的编号。转换结果在结果寄存器中的放置方式与转换精度(8/10/12 位)和结果对齐方式有关,如表 8-2 所列。

表 8-2 转换结果放置方式与转换精度、对齐方式关系

A/D 转换精度	DJM	转换结果在 ATDDRn 中的放置方式
8 位	0	Bit[11 : 4]=转换结果, Bit[3 : 0]=0000
8 位	1	Bit[7 : 0]=转换结果, Bit[11 : 8]=0000
10 位	0	Bit[11 : 2]=转换结果, Bit[1 : 0]=00
10 位	1	Bit[9 : 0]=转换结果, Bit[11 : 10]=00
12 位	x	Bit[11 : 0]=转换结果

## 8.3 A/D 模块编程方法与实例

### 8.3.1 A/D 模块基本编程方法

A/D 转换编程主要涉及 6 个控制寄存器(ATDCTL0~ATDCTL5)、状态寄存器 0(ATDSTAT0)、数据寄存器(ATDDR0)。

#### (1) A/D 转换初始化

在程序初始化时应对 A/D 转换的 5 个控制寄存器(ATDCTL0~ATDCTL4)写入控制字

节,决定序列长度,设置分频系数和转换精度等。

```
ATD0CTL4 = 0b11101011; //设置采样时间和频率,fATDCLK = fBUS/(2 × (PRS + 1))
ATD0CTL3 = 0b10001000; //采样结果右对齐,每个序列的转换个数为 1
ATD0CTL0 = 0b00001111; //外部触发时,选择通道 15
ATD0CTL1 = 0b01001111; // 12 位精度,采样前不卸载内部采样电容
ATD0CTL2 = 0b01000010; //下降沿触发,不接受外部信号,禁用 A/D 比较中断请求
```

## (2) 启动 A/D 转换

对 A/D 转换状态和控制寄存器 ATDCTL5 写入控制字节,选取要转换的通道、结果寄存器的调整方式、设置是连续转换还是一次转换,此时就开始了一路 A/D 转换。

```
ATDCTL5 = channel; // channel 为用户调用时设置的数,采用什么样的控制方式由用户决定。
```

## (3) 获得 A/D 转换结果

若是中断方式,在 A/D 中断程序中取得,若是查询方式,通过 A/D 转换状态寄存器 0 (ATDSTAT0)的第 7 位(SCF 位)取得,当 SCF=1 时可从 A/D 数据寄存器中取的数据。

```
//取 A/D 转换结果
for(;;)
//判断 ATDSTAT0 的第 7 位是否为 1
if ((ATD0STAT0 & (1 << 7)) != 0)
{
    temp = ATD0DR0; //从 A/D 数据寄存器 0 中读 12 位数据
    break;
}
return temp;
```

# 8.3.2 A/D 构件设计与测试实例

## 1. 构件设计

A/D 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。A/D 构件包括头文件 AD.h 和 AD.C 文件。A/D 构件头文件中主要包括相关宏定义、A/D 的功能函数原型说明等内容。A/D 构件程序文件的内容是给出 A/D 各功能函数的实现过程。

本小节给出了 XS128 内部 A/D 转换模块的程序,包含了中值滤波与平均值滤波的复合滤波方式,因为不含软件滤波的 A/D 转换程序很少能够实际应用。本程序中的中值滤波,就是对 3 次采样值比较大小,取中间的一个。所谓平均值滤波,就是  $N$  个采样值求平均。而中值滤波与平均值滤波的复合滤波方式,就是先进行中值滤波,再进行平均值滤波。



下面以 MC9S12XS128 的 A/D 转换程序为例,给出相应的程序代码。

### 1) A/D 构件头文件(AD.h)

```
// ----- *
// 文件名: AD.h *
// 说 明: AD 构件头文件 *
// ----- *

#ifndef AD_H
#define AD_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//构件函数声明区
void ADCInit(void); //初始化
uint16 ADCValue(uint8 channel); //获取通道 channel 的 A/D 转换结果
uint16 ADCMid(uint8 channel); //获取通道 channel 中值滤波后的 A/D 转换结果
uint16 ADCave(uint8 channel, uint8 n); //通道 channel 进行 n 次均值滤波的 A/D 转换结果

#endif
```

### 2) A/D 构件程序文件(AD.c)

```
// ----- *
//文件名: AD.c *
//说 明: AD 构件函数源文件 *
// ----- *

//头文件包含,及宏定义区

//头文件包含
#include "AD.h" //函 数 名:ADCInit

//构件函数实现
// ----- *
//函 数 名:ADCInit *
//功 能:A/D 转换初始化,设置 A/D 转换时钟频率为 1MHz *
//参 数:无 *
```

```

//返    回:无
//-----*
void ADCInit(void)
{
    //ATD0CTL4: SMP2 = 1,SMP1 = 1,SMP0 = 1,PRS4 = 0,PRS3 = 1,PRS2 = 0,PRS1 = 1,PRS0 = 1
    ATD0CTL4 = 0b11101011; //设置采样时间和频率,  $f_{ADCLK} = f_{BUS} / (2 \times (PRS + 1))$ 

    //ATD0CTL3: DJM = 1,S8C = 0,S4C = 0,S2C = 0,S1C = 1,FIFO = 0,FRZ1 = 0,FRZ0 = 0
    ATD0CTL3 = 0b10001000; //采样结果右对齐,每个序列的转换个数为 1

    //ATD0CTL0: WRAP3 = 1,WRAP2 = 1,WRAP1 = 1,WRAP0 = 1
    ATD0CTL0 = 0b00001111; //外部触发时,选择通道 15

    //ATD0CTL1: ETRIGSEL = 0,SRES1 = 1,SRES0 = 0,SMP_DIS = 0,ETRIGCH3 = 1,ETRIGCH2 = 1
    //,ETRIGCH1 = 1,ETRIGCH0 = 1
    ATD0CTL1 = 0b01001111; // 12 位分辨率,采样前不卸载内部采样电容

    //ATD0CTL2: AFFC = 1,ICLKSTP = 0,ETRIGLE = 0,ETRIGP = 0,ETRIGE = 0,ASCIE = 1,
    //ACMPIE = 0
    ATD0CTL2 = 0b01000010; //下降沿触发,不接受外部信号,禁用 AD 比较中断请求
}

//-----*
//函 数 名:ADCValue
//功    能:1 路 A/D 转换函数,获取通道 channel 的 A/D 转换结果
//参    数:channel = 通道号(0~15)
//返    回:该通道的 A/D 转换结果
//-----*
uint16 ADCValue(uint8 channel)
{
    uint16 temp; //暂存 A/D 转换的结果
    ATD0CTL5 = channel; //取 A/D 转换结果
    for(;;)
    //判断 ATDSTAT0 的第 7 位是否为 1
    if ((ATD0STAT0 & (1 << 7)) != 0)
    {
        temp = ATD0DR0; //从 A/D 数据寄存器 0 中读 12 位数据
        break;
    }
}

```



```
        return temp;
    }

// -----*
//函数名:ADCMid                                     *
//功    能:1 路 A/D 转换函数(中值滤波),获取通道 channel 中值滤波后的 A/D *
//      转换结果                                     *
//参    数:channel = 通道号(0~15)                   *
//返    回:该通道中值滤波后的 A/D 转换结果         *
//内部调用:ADCValue                                 *
// -----*
uint16 ADCMid(uint8 channel)
{
    uint16 i,j,k,tmp;

    //1. 取三次 A/D 转换结果
    i = ADCValue(channel);
    j = ADCValue(channel);
    k = ADCValue(channel);

    //2. 从三次 A/D 转换结果中取中值
    tmp = (i > j) ? j : i;
    tmp = (tmp > k) ? tmp : k;

    return tmp;
}
// -----*
//函数名:ADCAve                                     *
//功    能:1 路 A/D 转换函数(均值滤波),通道 channel 进行 n 次中值滤波,求和再作 *
//      均值,得出均值滤波结果                     *
//参    数:channel = 通道号(0~15),n = 中值滤波次数(1~255) *
//返    回:该通道均值滤波后的 A/D 转换结果         *
//内部调用:ADCMid                                   *
// -----*
uint16 ADCAve(uint8 channel, uint8 n)
{
    uint16 i;
    uint32 j;
```



```

j = 0;
for (i = 0; i < n; i++)
    j += ADCMid(channel);
j = j/n;

return (uint16)j;
}

```

## 2. A/D 构件测试实例

```

// ----- *
// 工 程 名: main.c *
// 硬件连接: PAD0,PAD1 分别接电位器 1,2 ,PA 口的 1 脚接小灯 *
// 程序描述: AD 采集。两个 AD 口分别于电位器连接,调节电位器,将采集到的值通过 *
//          串口发送到 PC *
// 目    的: 掌握 A/D 转换的基本知识 *
// -----苏州大学飞思卡尔嵌入式系统研发中心 2011 年 ----- *
//包含头文件
#include "Includes.h"      //包含总头文件

//在此添加全局变量定义

//主函数
void main()
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;    //运行计数器
    uint16 mADResult;

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
    SCIIInit(0,FBUS_32M,9600);                      //串口 0 初始化
    ADCInit();                                       //AD 初始化
}

```



```

// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 2)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
    }

    // -----
    //2. AD 采集
    mADResult = ADCave(0,10);                      //通道 0 转换,滤波 10 次
    SCISend1(0,0xAA);                               //将 12 位转换结果发给 PC 机
    SCISend1(0,(uint8)(mADResult>>8));
    SCISend1(0,(uint8)mADResult);

    mADResult = ADCave(1,10);                      //通道 1 转换,滤波 10 次
    SCISend1(0,0xBB);                               //将 12 位转换结果发给 PC 机
    SCISend1(0,(uint8)(mADResult>>8));
    SCISend1(0,(uint8)mADResult);

    Delay(10000);
    // -----
} //for_end(主循环结束)
} //main_end

```

## 8.4 SPI 的基本工作原理

### 8.4.1 SPI 基本概念

串行外设接口(SPI, Serial Peripheral Interface)是 Freescale 公司推出的一种同步串行通信接口,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,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 SCK、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

SPI 系统是典型的“主机—从机”(Master - Slave)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信,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),其他设备则称为 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 $\overline{SS}$

一些芯片带有从机选择引脚 $\overline{SS}$ (Slave Select)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 8-5 的接法。对于一个主 MCU 带多个从属 MCU 的系统,主机 MCU 的 $\overline{SS}$ 接高电平,每一个从机 MCU 的 $\overline{SS}$ 接主机的 I/O 输出线,由主机控制其电平高低,以便主机选中该从机。

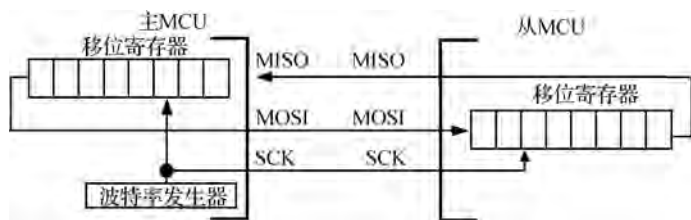


图 8-5 SPI 全双工主—从连接

### (3) 主出从入引脚 MOSI

主出从入引脚 MOSI(Master Out/Slave In)是主机输出、从机输入数据线。对于 MCU 被设置为主机方式,主机送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

主入从出引脚 MISO(Master In/Slave Out)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,送向主机的数据从该引脚输出。

### (5) SPI 串行时钟引脚 SCK

SPI 串行时钟引脚 SCK(Serial Clock)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 SCK 引脚输出给从机的 SCK 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 SCK 引脚输出自动产生的 8 个时钟周期信号,SCK 信号的一个跳变进行一位数据移位传输。



### (6) 主出主入引脚 MOMI

主出主入引脚 MOMI(Master Out/Master In)是单线传输时,主机的输入输出数据线,与 MOSI 引脚复用。对于 MCU 被设置为主机方式,该引脚发送或接收从机的数据。

### (7) 从入从出引脚 SISO

从入从出引脚 SISO(Slave In/Slave Out)是单线传输时,从机的输出输入数据线,与 MISO 引脚复用。对于 MCU 被设置为从机方式,该引脚发送或接收主机的数据。

### (8) 时钟极性

表示时钟信号在空闲时是高电平还是低电平。

### (9) 时钟相位

决定数据是在时钟信号 SCK 的上升沿还是在 SCK 的下降沿进行采样。

## 8.4.2 SPI 的数据传输

图 8-5 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 SCK 引脚依次送出,在 SCK 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 8-4 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,其他处于从机地位,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。

## 8.4.3 SPI 模块的时序

SPI 的数据传输是在时钟信号 SCK(同步信号)的控制下完成的。数据传输过程涉及时钟相位与时钟极性两个概念。所谓时钟极性是指时钟信号在空闲时是高电平还是低电平,所谓时钟相位是指接收方从数据线上取数的时刻是在时钟信号 SCK 的上升沿还是在下降沿。这样就有 4 种可能,分别如图 8-6、图 8-7、图 8-8 和图 8-9 所示。以下讲解使用 CPHA 表示时钟相位,CPOL 表示时钟极性。从时序图上可以看出,CPOL 为 1 或 0 时,时钟信号 SCK 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。

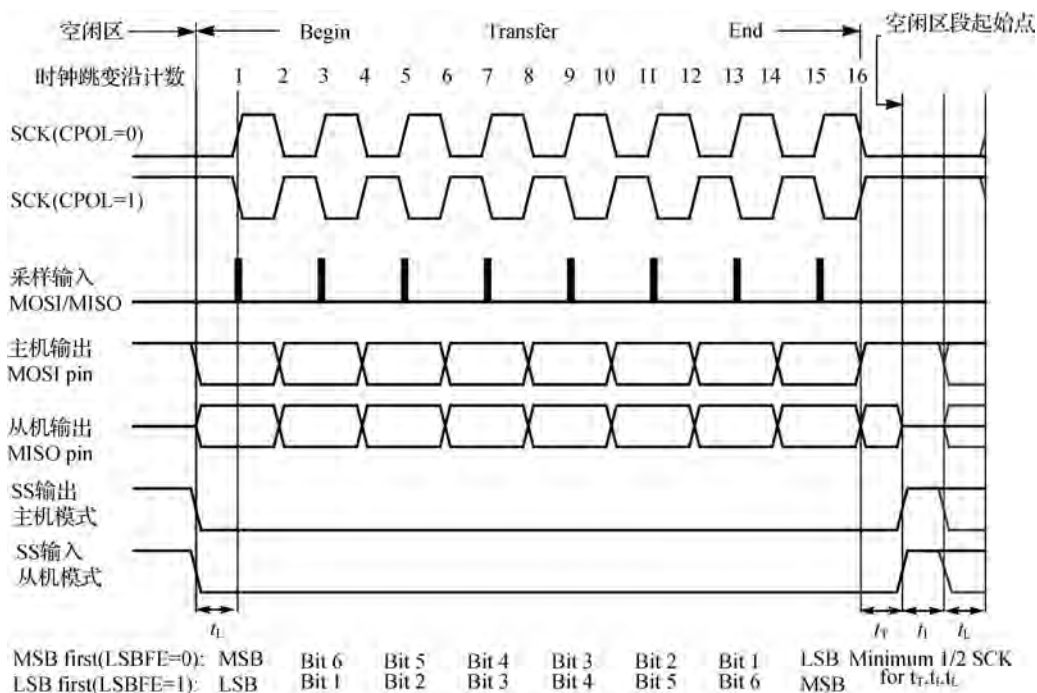


图 8-6 CPHA=0 时 8 位(XFRW=0)数据/时钟时序图

如图 8-6、图 8-7 所示,当 CPHA=0 时,MISO/MOSI 引脚上输出的 MSB 位/LSB 位在第一个 SCK 跳变沿前至少半个时钟周期就已经上线了,即 $\overline{SS}$ 一拉低就可以上线了,而在第一个跳变沿锁存传送的数据位,第二个跳变沿将该位移入移位寄存器,之后,继续下一个数据位传送,依次进行。XFRW=0 时,SCK 线共有 16 个跳变沿,完成 8 位(一个字节)传送;XFRW=1, SCK 线共有 32 个跳变沿时,完成 16 位(一个字)传送。在奇数个跳变沿时,锁存数据位,偶数个跳变沿时,移入移位寄存器,最后一位移入时才将数据转入数据寄存器。主机的 $\overline{SS}$ 引脚要么为输出高电平口要么为普通的输出口。主机的使能选择从机的 $\overline{SS}$ 引脚和从机的 $\overline{SS}$ 引脚在每两个传送数据之间至少有半个时钟周期是高电平。

如图 8-8、图 8-9 所示,当 CPHA=1 时,第一个 SCK 跳变沿是启动数据开始传送,该跳变沿在空闲结束后延迟半个时钟周期出现,在第二个跳变沿锁存传送的 MISO/MOSI 引脚上输出的 MSB 位/LSB 位数据位,第三个跳变沿将该位移入移位寄存器,之后继续下一个数据位传送,依次进行。XFRW=0 时,SCK 线共有 16 个跳变沿,完成 8 位(一个字节)传送;XFRW=1, SCK 线共有 32 个跳变沿时,完成 16 位(一个字)传送。在偶数个跳变沿时,锁存

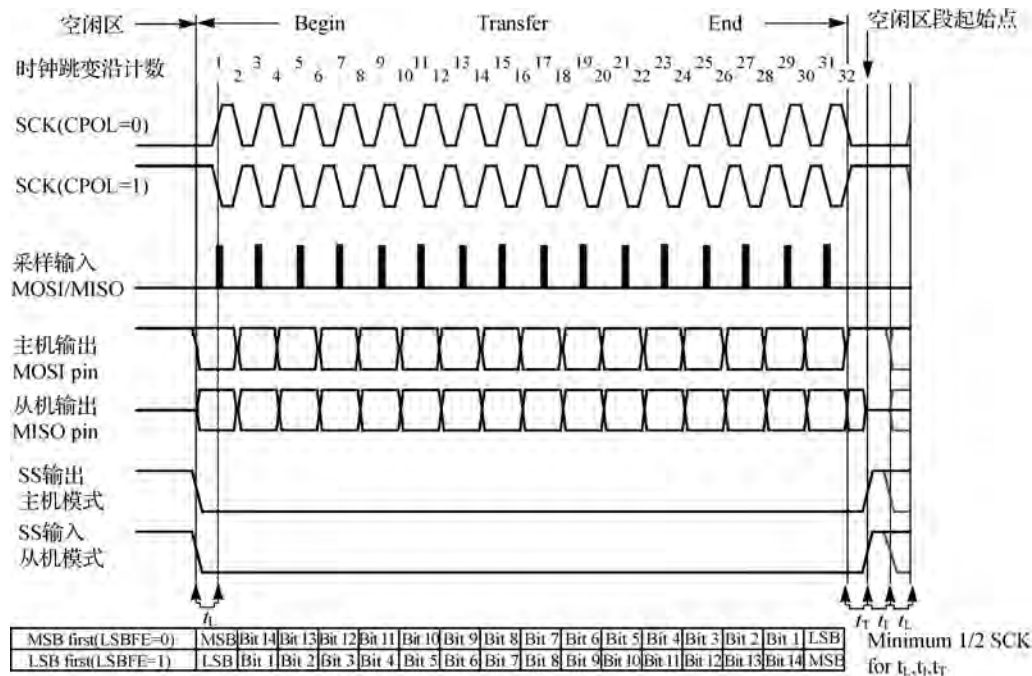


图 8-7 CPHA=0 时 16 位 (XFRW=1) 数据/时钟时序图

数据位；奇数个跳变沿时，移入移位寄存器，最后一位移入时才将数据转入数据寄存器。主机的  $\overline{SS}$  引脚要么为输出高电平口要么为普通的输出口。CPHA=1 这种传送方式，在连续传送时， $\overline{SS}$  引脚可以一直保持低电平，没有高电平。

**注意：**SPI 主设备的时钟极性和时钟相位选择是依据从设备的时钟极性和时钟相位，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/发送数据，是第一个时钟跳变沿还是第二个时钟跳变沿，是在时钟的上升沿还是下降沿？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。但是主机和从机时序的极性和相位应该配置为一样，只是主机和从机的接收和发送时序会相差半个周期。



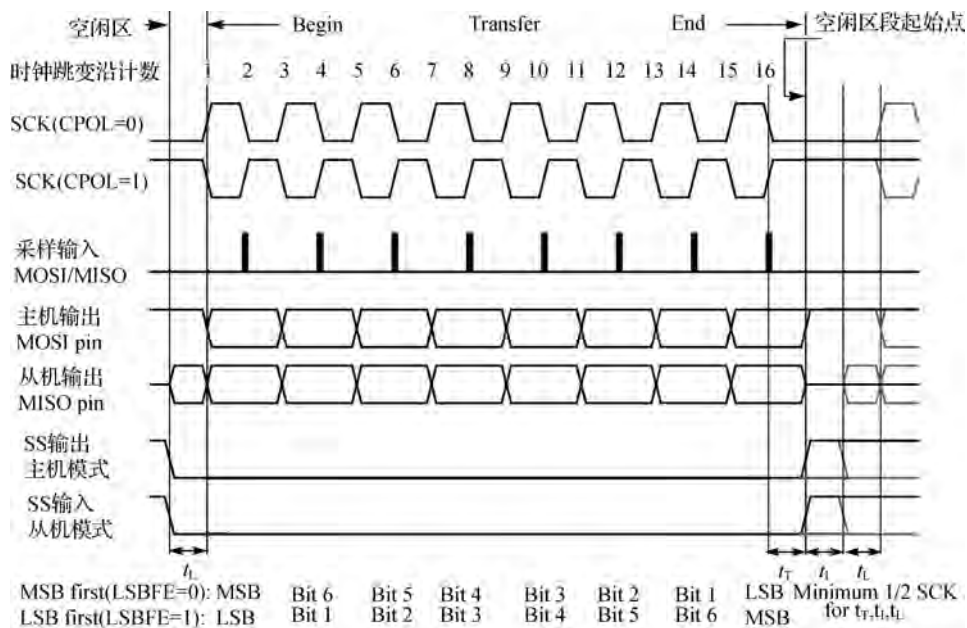


图 8-8 CPHA=1 时 8 位(XFRW=0)数据/时钟时序图

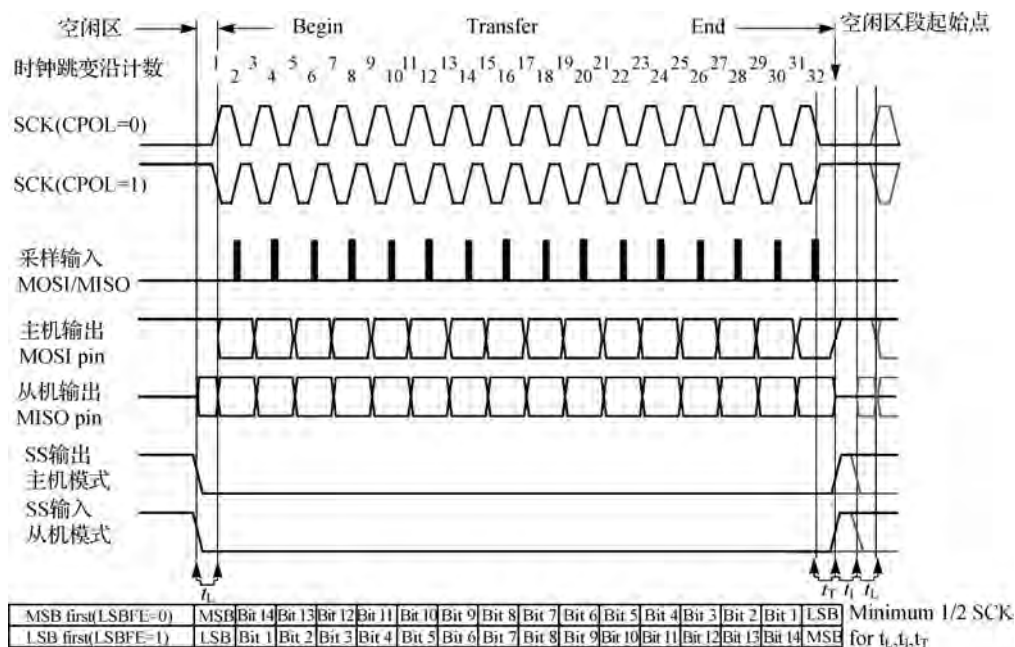


图 8-9 CPHA=1 时 16 位(XFRW=1)数据/时钟时序图



## 8.4.4 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,可以使用 3 个普通 I/O 口,分别定义为 pin\_SCK、pin\_MISO、pin\_MOSI 来模拟 SPI 器件的 SCK、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。假设某 SPI 外围芯片在 SCK 的上升沿输入数据和在下降沿输出数据,则初始化 pin\_SCK=1,在 MCU 正常工作后,置 pin\_SCK=0。这样,MCU 在输出 1 位 SCK 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 SCK 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 SCK=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 SCK 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\_SCK=0,在 MCU 正常工作后,置 SCK=1,使 SPI 器件输出 1 位数据(MCU 接收 1 位数据),之后再置 SCK=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。

## 8.5 SPI 模块的编程寄存器

从程序员的角度看,SPI 模块有 6 个 8 位寄存器,这些寄存器的基本信息由表 8-3 所列。在理解和掌握这 6 个寄存器的用法的基础上,就可以进行 SPI 编程了。

表 8-3 SPI 寄存器简单介绍

SPI0 地址	寄存器名称与缩写	访问权限	基本功能
\$ 00D8	控制寄存器 1(SPI0CR1)	读/写	设置主从方式、时钟极性和相位、数据流位顺序、中断使能等
\$ 00D9	控制寄存器 2(SPI0CR2)	读/写	设置发送与接收数据宽度、单双线模式等
\$ 00DA	波特率寄存器(SPI0BR)	读/写	设置波特率
\$ 00DB	状态寄存器(SPI0SR)	只读	发送空中断标志和接收满中断标志以及模式错误标志
\$ 00DC	数据寄存器(SPI0DRH)	读/写	收发的高 8 位数据
\$ 00DD	数据寄存器(SPI0DRL)	读/写	收发的低 8 位数据

下面从编程角度来介绍 SPI 模块的各寄存器,包括波特率寄存器、控制寄存器、状态寄存器和数据寄存器。

### 1. SPI0 控制寄存器

#### (1) SPI0 控制寄存器 1(SPI0CR1)

SPI0 控制寄存器 1 一般情况下只能复位时写一次,以后不须再对其写入,不能更改对 SPI



的设置。定义如下：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
复位	0	0	0	0	0	1	0	0

D7—SPIE 位,SPI 中断允许位(SPI Interrupt Enable Bit),SPIE=1 时允许 SPI 中断;SPIE=0 时禁止 SPI 中断。当该位为 1 时,每次 SPRF 或 MODF 状态标志置位时发出硬件中断请求。

D6—SPE 位,SPI 系统允许位(SPI System Enable Bit)。该位将使能 SPI 系统,并且将与 SPI 系统有关的引脚设置为 SPI 系统功能。SPE=1 时,允许 SPI,并且将与 SPI 系统有关的引脚设置为 SPI 功能;SPE=0 时,禁止 SPI。

D5—SPTIE 位,SPI 发送中断允许位(SPI Transmit Interrupt Enable)。SPTIE=1 时,SPI 允许发送中断;SPTIE=0 时,禁止发送中断。

D4—MSTR 位,SPI 主机/从机模式选择位(SPI Master/Slave Mode Select Bit)。该位决定 SPI 的工作方式,MSTR=1 为主机方式,MSTR=0 为从机方式。

D3—CPOL 位,SPI 时钟极性选择位(SPI Clock Polarity Bit)。该位决定 SCK 引脚在无传输动作时为高电平还是低电平,在传输过程中改变该位的状态将使 SPI 系统进入闲置状态(IDLE)。在 SPI 传输数据过程中,CPOL 要有相应的值。CPOL=1 时,时钟选择低有效,SCK 引脚空闲时为高电平;CPOL=0 时,时钟选择高有效,SCK 空闲时为低电平。

D2—CPHA 位,SPI 时钟相位选择位(SPI Clock Phase Bit)。该位控制串行时钟与 SPI 数据的时序关系。CPHA=1 时,数据在 SCK 时钟周期的第一个跳变沿上线,在偶数个跳变沿锁存接收;CPHA=0 时,发送数据以 $\overline{SS}$ 变低开始,数据在 SCK 每个时钟周期的第二个跳变沿上线,在奇数个跳变沿锁存接收。

D1—SSOE 位,从机选择信号输出允许位(Slave Select Output Enable),只有在主机模式下才能开启 $\overline{SS}$ 引脚的输出功能。该位与 MODFEN(SPI0 控制寄存器 2 的 D4 位,下面将会介绍)配合使用,定义如表 8-4 所列。

表 8-4 SSOE 和 MODFEN 使用表

MODFEN	SSOE	主机模式	从机模式
0	0	$\overline{SS}$ 不可用	$\overline{SS}$ 输入
0	1	$\overline{SS}$ 不可用	$\overline{SS}$ 输入
1	0	$\overline{SS}$ 具有 MODF 特性的输入	$\overline{SS}$ 输入
1	1	$\overline{SS}$ 为从机选择输出	$\overline{SS}$ 输入



D0—LSBFE 位,最低位(LSB)先传输顺序选择位,该位只决定传输过程中各位的先后顺序,不影响数据位在寄存器中的顺序,因此读/写操作正常进行,即最高位(MSB)在第 7 位(BIT7)。LSBFE=1,LSB 先传输;LSBFE=0,MSB 先传输。对于仅由 MC9S12XS128 构成的互连系统,只要各个 SPI 的 LSBF 相同,对传输结果无影响,当外设或器件连接时,SPI 必须根据该设备或器件所要求的位顺序正确设置 LSBFE。

## (2) SPI0 控制寄存器 2(SPI0CR2)

一般情况下,SPI0 控制寄存器 2 只能在复位时写一次,以后不再对其写入,不再更改对 SPI 的设置。定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	—	XFRW	—	MODFEN	BIDIROE	—	SPISWAI	SPC0
复位	0	0	0	0	0	0	0	0

D6—XFRW 位,传输宽度选择位(Transfer Width)。XFRW=0 数据传输宽度为 8 位,只使用 SPIDRL 寄存器;XFRW=1 数据传输宽度为 16 位,SPIDRH 和 SPIDRL 将形成一个 16 位数据寄存器。

D4—MODFEN 位,模式错误使能位。模式错误表示的是主从模式选择的设置和 $\overline{SS}$ 引脚的连接不一致。如果作为主机并且 MODFEN=0, $\overline{SS}$ 引脚将不再用作从机选择的引脚;对于从机而言,不论该位为何值, $\overline{SS}$ 引脚只有作为输入引脚时才有效,具体情况参考表 8-4。在主机模式下,数据传输过程中改变该位将终止传输并且将强制 SPI 进入闲置状态(idle)。MODFEN=1,允许模式错误标志位置位;MODFEN=0,禁止模式错误标志位置位。

D3—BIDIROE 位,双向模式下输出缓冲使能位(Output enable in the Bidirectional mode of operation)。如果开启了双向模式(SPC0=1),该位用来控制 SPI 的 MOSI 以及 MISO 的输出缓冲区。在主机模式下,该位用来控制 MOSI 端口的缓冲区;在从机模式,该位用来控制 MISO 端口的输出缓冲区。BIDIROE=1 时,双向模式下允许输出缓冲;BIDIROE=0 时,双向模式下禁止输出缓冲。具体情况参考表 8-5。

D1—SPISWAI 位,等待模式下 SPI 工作方式(SPI Stop in Wait Mode Bit)。该位是用于设置等待模式下 SPI 的节能模式。SPISWAI=1 时,等待模式下,SPI 停止 SPI 时钟;SPISWAI=0 时,等待模式下,SPI 时钟工作正常。

D0—SPC0 位,串行引脚控制位(Serial Pin Control Bit 0)。该位与 MSTR 位一起决定串行引脚功能设置,如表 8-5 所列。

表 8-5 双向引脚配置

引脚模式		SPC0	BIDIROE	MSTR	MISO	MOSI
A	普通	0	X	0	从出	从入
B				1	主入	主出
C	双向	1	0	0	从输入	不用
D				1	不用	主输入
E			1	0	从输入/输出	不用
F				1	不用	主输入/输出

2. SPI0 波特率寄存器(SPI0BR)

SPI0 波特率寄存器 SPI0BR 的作用是设置 SPI0 的波特率,通常的情况下选择内部总线时钟为 SPI 通信的时钟源,此时利用 SPI0BR 对总线频率可以进行分频得到 SPI 的波特率。定义如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	—	SPPR2	SPPR1	SPPR0	—	SPR2	SPR1	SPR0
复位	0	0	0	0	0	0	0	0

**说明:**当数据传送的时候,写入寄存器会造成错误的假象。

D6~D4 SPPR2~SPPR0—SPI0 波特率预选择位(SPI Baud Rate Preselection Bits),这 3 位定义波特率预分频值。

D2~D0 SPR2~SPR0—SPI0 波特率选择位(SPI Baud Rate Selection Bits),这 3 位定义波特率另一分频值。

波特率的计算公式:

波特率分频因子
 =
 (SPPR + 1) × 2<sup>(SPR+1)</sup>

(式 8-1)

SPI 波特率
 =
 总线时钟 / 波特率分频因子
 

(式 8-2)

例如:SPPR[2-0]=001,SPR[2-0]=011

分频因子
 =
 (SPPR + 1) × 2<sup>(SPR+1)</sup>
 =
 (0b001 + 1) × 2<sup>(0b011+1)</sup>
 =
 32

3. SPI0 状态寄存器(SPI0SR)

SPI0 状态寄存器 SPI0SR 是一个只读寄存器,反映了 SPI 的工作状态,其中包括接收满中断标志位、发送空中断标志位和模式错误 3 个标志位。程序可以检查各位的状态,也可以通过特定的寄存器访问序列将标志位清 0。定义如下:



数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SPIF	—	SPTEF	MODF	—	—	—	—
复位	0	0	1	0	0	0	0	0

D7—SPIF 位, SPIF 中断标志位(SPIF Interrupt Flag)。当 SPI 数据寄存器接收到一个字节的数时, 该位置为 1, 将 SPI 数据寄存器中的数据读出后, 该标志位清 0。当 SPI0CR1 寄存器中 SPIF 位为 1 时, 能产生一个接收中断请求; SPIF=0 时, 则说明数据没有传送结束, 还不能接收数据。如何清除该位参考表 8-6。

表 8-6 清除 SPIF 中断标志位序列

XFRW	清除 SPIF 中断标志为序列		
0	当 SPIF=1 时, 读 SPISR	然后	读 SPIDRL
1	当 SPIF=1 时, 读 SPISR	然后	读 SPIDRL <sup>1</sup>
			或者
			读 SPIDRH 字节 <sup>2</sup>   读 SPIDRL 字节
			或者
			读(SPIDRH;SPIDRL)字

注: ① 在这种情况下, SPIDRH 寄存器中的数据已经丢失。

② SPIDRH 可以被重复的读取并且不会影响 SPIF 标志位; 只有当 SPIF=1 时读取过 SPISR 寄存器之后, 然后再读取 SPIDRL 寄存器才能清除 SPIF 标志位。

D5—SPTEF 位, SPI 发送空中断标志(SPI Transmit Empty Interrupt Flag)。

发送数据寄存器为空, SPTEF 为 1, 读 SPI0SR 然后写一个数据至 SPI0DR, 则清空该标志位。当 SPI0CR1 寄存器中 SPTIE 位为 1 时, 则产生一个发送中断请求; SPTEF=0, SPI 发送数据寄存器为满, 不能向数据寄存器中写数据。如何清除该位参考表 8-7。

**说明:** 只有 SPTEF=1 时才可以向 SPI 数据寄存器中写数据。若 SPTEF=0 时, 任何向 SPI 数据寄存器中写入的操作都将忽略。

D4—MODF 位, 模式错误标志位(Mode Fault Flag)。

主机方式下,  $\overline{SS}$  的输入为低电平, 且 MODFEN=1, 该标志位置 1, 代表模式发生错误。当 MODF 为 1 时, 表示系统中已经出现另一个主机, 并正在选中从器件。此时读 SPI0SR, 然后写 SPI0CR1, MODF 自动清 0。MODF=0, 无异常。

表 8-7 清 SPTEF 中断标志位序列

XFRW	清 SPTEF 中断标志为序列		
0	当 SPTEF=1 时,读 SPISR	然后	写 SPIDRL <sup>1</sup>
1	当 SPTEF=1 时,读 SPISR	然后	写 SPIDRL <sup>2</sup>
			或者
			写 SPIDRH 字节 <sup>3</sup>   写 SPIDRL 字节
			或者
			写 (SPIDRH;SPIDRL)字

注：① 当 SPTEF=0 时写 SPIDRH 或 SPIDRL 将会被忽略。

② 在这种情况下,SPIDRH 寄存器中的数据没有定义。

③ SPIDRH 寄存器可以重复地写但是不会对 SPTEF 有影响。当 SPTEF=1 时,只有当读 SPISR 之后再写 SPIDRL 寄存器之后将清除 SPTEF 标志位。

4. SPI0 数据寄存器(SPI0DRH 和 SPI0DRL)

SPI 数据寄存器由两个 8 位数据寄存器构成。定义分别如下：  
SPI0DRH：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	R15	R14	R13	R12	R11	R10	R9	R8
写	T15	T14	T13	T12	T11	T10	T9	T8

SPI0DRL：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	R7	R6	R5	R4	R3	R2	R1	R0
写	T7	T6	T5	T4	T3	T2	T1	T0

当选择 16 位数据传输宽度时(XFRW=1),两个数据寄存器 SPI0DRH 和 SPI0DRL 都有效,可共同组成一个 16 位的数据寄存器。当选择 8 位数据传输宽度时(XFRW=0),只有低位数据寄存器 SPI0DRL 有效,高位数据寄存器 SPI0DRH 无效。

SPI 数据寄存器对 SPI 数据而言既是输入寄存器又是输出寄存器。写该寄存器允许数据进行排队然后传送。主机方式下,前面的数据传送完成,后面队列中的数据可以立即传送。SPTEF=0 时,数据寄存器可接收新数据;但只有当 SPTEF=1 时,才可以向数据寄存器中写数据。SPIF=1 时,在 SPIDR 中接收的数据是有效的。如果设置 SPIF=0,并且接收数据,那么接收的数据从接受移位寄存器转移到 SPIDR,并且置 SPIF 为 1。



## 8.6 SPI 构件设计与测试实例

SPI 具有初始化、接收和发送 3 种基本操作。按照构件的思想,可将它们封装成几个独立的功能函数,初始化函数完成对 SPI 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 SPI 模块进行编程,实际上已经涉及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 SPI.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 SPI.h 的头文件,用来定义模块的基本信息和对外接口。

在进行 SPI 编程时,首先在程序初始化部分必须进行 SPI 系统初始化。SPI 的初始化主要是对 SPI 控寄存器 SPI0CR1、SPI0CR2 进行设置,定义 SPI 工作模式、时钟的空闲电平及相位、允许 SPI 等,以及对 SPI 的波特率寄存器 SPI0BR 设置,波特率根据传送速度要求计算而得到。(本节以 SPI0 为例,给出最基本的用法,以此作为 XS128 MCU 的 SPI 的编程入门。)

### 1. SPI 构件设计

SPI 初始化一般需要两步:

① 写控制字到 SPI0CR1,确定是否允许 SPI 接收中断、SPI 的工作方式、时钟极性、时钟相位、是否允许 SPI 等。SPI0CR2 采用默认值。

② 写控制字到 SPI0SR,确定 SPI 的波特率,方法参考式 8-1、式 8-2。

具体程序如下:

#### 1) SPI 构件头文件(SPI.h)

```
// -----*
// 文件名: SPI.h                                     *
// 说 明: SPI 构件头文件                             *
// -----*

#ifndef SPI_H
#define SPI_H

//头文件包含,及宏定义区

//头文件包含
#include "Includes.h"

//构件函数声明区
void SPI_Init(void); // SPI 通信初始化
```

```

uint8 SPI_Send1(uint8 SendData); //通过 SPI 发送 1 字节数据
uint8 SPI_Rec1(uint8 * flag); //通过 SPI 接收 1 字节数据
#endif

```

## 2) SPI 程序文件(SPI.c)

```

// ----- *
// 文件名: SPI.c *
// 说 明: SPI 构件函数源文件 *
// ----- *

//头文件包含,及宏定义区
//头文件包含
#include "SPI.h"

//构件函数实现
//SPI_Init;SPI 通信初始化 ----- *
//功 能:SPI 通信初始化 *
//参 数:无 *
//返 回:无 *
// ----- *

void SPI_Init(void)
{
    MODRR |= 0x10; //使用 PM 口

    SPI0CR1 = 0x04; //复位 SPI 控制寄存器 1
    (void)SPI0SR; //读状态寄存器
    (void)SPI0DR; //读数据寄存器

    //设置波特率
    SPI0BR = 0b00110011; //分频系数 64,得到波特率 B = buscls/64 = 0.5Mbit/s

    //配置控制寄存器 2
    SPI0CR2 = 0b00010000;
    //      |||||___0 串行引脚控制
    //      |||||___1STOP 模式下 SPI 工作模式
    //      |||||___2 无定义
    //      |||||___3 双向模式下输出缓冲使能位
    //      |||||___4 模式错误使能为
    //      |||||___5 无定义

```



```
//          ||_____6 传输宽度
//          |_____7 无定义

//配置控制寄存器 1
SPI0CR1 = 0b01010110;
//          |||||___0MSB 方式传送
//          |||||___1 从器件选中输出信号(/SS)允许位
//          |||||___2 奇数沿取数
//          |||||___3 时钟空闲低电平
//          |||||___4 主机方式
//          |||___5 禁止发送完成中断
//          ||___6 使能 SPI 功能
//          |___7 中断使能
}

//SPI_Send1e:SPI 发送 1 字节数据-----*
//功 能:通过 SPI 发送 1 字节数据                                     *
//参 数:SendData- 要发送的 1 字节数据                               *
//返 回:1:发送成功;0:发送失败                                       *
//-----*
uint8 SPI_Send1(uint8 SendData)
{
    uint16 i;
    while(!(SPI0SR_SPTEF));    //等到发送队列空,开始写入数据
    SPI0DRL = SendData;

    //等待一段时间
    for(i = 0;i<0xFFFF;i++)
    {
        if(SPI0SR_SPTEF)      //若发送标志为 1,发送成功
        {
            return 1;
        }
    }

    return 0;                  //否则认为发送失败
}

//SPI_Rec1:SPI 接收 1 字节数据-----*
```



```

//功 能:通过 SPI 接收 1 字节数据                                     *
//参 数:flag: = 1:接收成功; = 0:接收失败                             *
//返 回:接收到的 1 字节数据                                         *
//-----*
uint8 SPI_Rec1(uint8 * flag)
{
    uint16 i;
    uint8 u8Data;
    for(i = 0;i<0xFFFF;i++)
    {
        if(SPI0SR_SPIF)        //若接收标志为 1,接收成功
        {
            u8Data = SPI0DRL;    //返回接收数据
            * flag = 1;
            return  u8Data;
        }
    }
    * flag = 0;
    return 0;                    //否则认为失败
}

```

## 2. SPI 测试实例

在进行 SPI 编程时,程序初始化部分必须进行 SPI 模块初始化。SPI 的初始化主要是对 SPI 控制寄存器 SPI0CR1、SPI0CR2 进行写入,并定义其基本操作模式(时钟相位、时钟极性 etc)。这里以两个相同的 XS128MCU 为例,分别将其设计为主机模式与从机模式进行两个 MCU 之间的 SPI 通信,按照构件的设计原则,给出 SPI 构件的头文件以及若干通信函数的设计。

SPI 双机测试功能分两个主要部分,主机 Master 和从机 Slave。主机定时 3 s 向从机发送一个字节的信,在从机接收主机数据信息的同时,同样向主机回送一个字节,同时从机将收到的字节通过串口发送到 PC 机,在 PC 机上运行的串口调试工具软件的显示界面中总是显示从机接收到主机发送的信息。从机接收数据使用查询方式。

### 1) 主机测试实例

```

//-----*
//工 程 名: SPI 主机程序                                           *
//硬件连接: 将 SPI 主机与另外一个 XS128 从机连接对应关系         *
//          主机 - - - 从机                                         *
//          PM4 - - - PM4                                           *

```



```
//          PM2 - - - PM2                                *
//          PM5 - - - PM5                                *
//          主机 PM3 通过 3.3K 电阻上拉到 5V              *
//          从机机 PM3 直接连接到 GND                    *
//          主机从机需要共地                              *
// 程序描述：主机在主循环中向从机循环发送 A              *
// 说    明：1.SPI 模块使用 PM 口                        *
//          2. 与从机连线不应松动,并且正确连接          *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*
```

```
//包含头文件
```

```
#include "Includes.h" //头文件
```

```
//在此添加全局变量定义
```

```
void main(void)
```

```
{
```

```
    //0.1 主程序使用的变量定义
```

```
    uint32 mRuncount = 0;    //运行计数器
```

```
    uint32 mSendcount = 0;   //发送计数器
```

```
    //0.2 关总中断
```

```
    DisableInterrupt();
```

```
    //0.3 芯片初始化
```

```
    MCUInit(FBUS_32M);
```

```
    //0.4 模块初始化
```

```
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
```

```
    SCIIInit(SCI_NUM_1, FBUS_32M, 38400); //串口 0 初始化
```

```
    SCISendN(SCI_NUM_1, 13, (uint8 *) "Hello! World!"); //发送 "Hello! World!"
```

```
    SPI_Init();    //SPI 模块初始化
```

```
    //0.5 开放中断
```

```
    EnableInterrupt();    //开总中断
```

```
    // 主循环
```

```
    for(;;)
```

```
    {
```

```
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
```

```

mRuncount + + ;
if (mRuncount >= 50000)
{
    mRuncount = 0;
    Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
}
// -----
// -----
// 2. 主循环计数到一定的值,向从机循环发送 0x010~0xFF 数据
mSendcount + + ;
if(mSendcount >= 50000)
{
    mSendcount = 0;
    if(SPI_Send1('A'))
    {
        SCISendString(0,"OK\r\n");
    }
    Delay(2500);
}

} //for_end(主循环结束)
} //main_end

```

## 2) 从机测试实例

```

// ----- *
// 工 程 名: SPI 从机程序 *
// 硬件连接: 将 SPI 主机与另外一个 XS128 从机连接对应关系 *
//          主机 - - - 从机 *
//          PM4 - - - PM4 *
//          PM2 - - - PM2 *
//          PM5 - - - PM5 *
//          主机 PM3 通过 3.3K 电阻上拉到 5V *
//          从机机 PM3 直接连接到 GND *
//          主机从机需要共地 *
// 程序描述: 从机在中断中接收主机发送的数据,并通过串口打印到 PC 机 *
// 说 明: 1. SPI 模块使用 PM 口 *
//          2. 与从机连线不应松动,并且正确连接 *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*

```



```
//包含头文件
#include "Includes.h" //头文件
//在此添加全局变量定义
void main(void)
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;    //运行计数器

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT, Light_Run, Light_OFF); //RUN 指示灯初始化为暗
    SCIInit(SCI_NUM_1, FBUS_32M, 38400);    //串口 0 初始化
    SCISendN(SCI_NUM_1, 13, (uint8 *) "Start Test! \r\n"); //发送 "Hello! World!"
    SPI_Init();    //SPI 模块初始化

    //0.5 开放中断
    EnableInterrupt();    //开总中断

    // 主循环
    for(;;)
    {
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + + ;
        if (mRuncount >= 50000)
        {
            mRuncount = 0;
            Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
        }
        // -----
        // -----
    } //for_end(主循环结束)
} //main_end
```

## Flash 存储器在线编程

Flash 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,这使得 Flash 存储器在嵌入式系统中的使用量迅速增长。Flash 存储器编程方法有写入器模式与在线编程模式两种。写入器模式是指开发写入器而使用的编程方式,目的是通过写入器将程序机器码从 PC 机中写入到目标芯片的 Flash 中。在线编程模式是指在程序运行过程中对 Flash 进行擦除与写入的编程方式,其目的是利用 Flash 保存一些希望掉电不丢失的参数。本章给出 XS128 的 Flash 存储器在线编程方法。本章主要知识点有:①S12X 的存储器分页机制及 XS128 的 D-Flash 和 P-Flash 的全局地址、局部地址、逻辑地址表示方法;②XS128 的 D-Flash 存储器编程的方法;③XS128 的 P-Flash 存储器编程的方法;④XS128 芯片的加密方法与解除密码方法。本章重点为:XS128 的 D-Flash、P-Flash 编程方法。

### 9.1 S12X 系列 MCU 的 Flash 存储器的特点及分页机制

理想的存储器应该具备存取速度快、数据不易丢失、存储密度高(单位体积存储容量大)、价格低等特点,但大部分存储器很难同时具有这些功能。Flash 存储器(有的文献译为:闪存存储器或快擦型存储器)技术日益趋于成熟,是目前比较理想的存储器。Flash 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,这些特点使得 Flash 存储器在嵌入式系统中获得了广泛地使用。S12XS 系列 MCU 内部 Flash 采用了第三代  $0.25\mu\text{m}$  的闪存技术,其擦写速度更快、性能更稳定。

Flash 存储器编程方法有写入器模式与在线编程模式两种。写入器模式是指开发写入器而使用的编程方式,其目的是通过写入器将程序机器码从 PC 机中写入到目标芯片的 Flash 中。写入器模式也称为监控模式(Monitor Mode),Freescle 称为背景调试模式 BDM(Background Debug Mode)。在线编程模式(In-Circuit Program)是指在程序运行过程中对 Flash 进行擦除与写入的编程方式,目的是利用 Flash 保存一些希望掉电不丢失的参数。这种在程序运行过程中对 Flash 存储区的数据或程序进行更新的编程方法也被称为用户模式(User Mode)的 Flash 编程。

## 9.1.1 S12X 系列 MCU 的 Flash 存储器的特点

S12X128 的 Flash 存储器具有以下特点：

① S12X128 的 Flash 分为两个部分：一个部分叫程序 Flash(Program Flash, P-Flash)，大小为 128 KB，用来存储程序；另一部分叫做数据 Flash(Data Flash, D-Flash)，大小为 8 KB，用来存储程序中掉电不丢失的数据。

② S12X128 的 P-Flash 共扩展为 8 个分页，每页 16 KB。128 KB 的 P-Flash 空间分为 128 个扇区，每个扇区 1 KB。每个扇区分为 128 块，每块 8 个字节。S12XS128 的 D-Flash 共扩展为 8 个分页，每页 1 KB。8 KB 的 D-Flash 空间分为 32 个扇区，每个扇区 256 字节。每个扇区分为 32 块，每块 8 字节。

③ P-Flash 块编程一般需要  $171\ \mu\text{s}$ ，扇区擦写需要 20 ms，整个 P-Flash 擦写需要 101 ms。85℃ 环境下高达 10 000 次擦写编程后，数据一般可以保持 100 年。在  $-40\sim 150\ ^\circ\text{C}$ ，P-Flash 一般可以擦除 10 万次。D-Flash 编程 1 个字一般需要  $97\ \mu\text{s}$ ，D-Flash 扇区擦除一般需要 5.2 ms。85℃ 环境下高达 50 000 次擦写编程后，数据一般可以保持 100 年。在  $-40\sim 150\ ^\circ\text{C}$ ，D-Flash 一般可以擦除 50 万次。详情可以参见附录 B。

④ 因为 Flash 的擦写程序是放在 P-Flash 中的，所以当要擦除 P-Flash 时，要将对应程序的机器码移入 RAM，使得程序跳入 RAM 中执行。而 D-Flash 执行代码与操作对象不在同一个存储区域，所以 D-Flash 的擦写不需要特殊的操作。

⑤ D-Flash 与 P-Flash 的存储空间是靠 16 位的地址线和 MMC(Memory Mapping Control)中的 GPAGE 等寄存器来实现 23 位地址寻址的，实现了存储器寻址空间的扩展。

⑥ P-Flash 和 D-Flash 中都实现了 ECC(Error Correction Codes, 错误校正码)，可以纠正 1 位错误，检测 2 位错误。但是 P-Flash 存储器中的 ECC 实现需要按照 8 字节对齐编程。

基于以上特点，掌握 XS128 的 Flash 存储器的编程技术，充分利用 XS128 的 Flash 存储器的功能，对基于 XS128 的嵌入式系统的开发是十分必要的。后续的内容是在实际应用的基础上，总结了 XS128 的 Flash 编程方法，并给出了编程实例和编程要点。

## 9.1.2 XS128 的 Flash 存储器分页机制

### 1. XS128 的分页机制

XS128 提供了 2 种存储器映像机制，包括：

① 一个 8 MB 的 CPU(或 BDM)全局映像，其定义了一个全局页寄存器(GPAGE)(或 BDMGPR)，用于 23 位地址指令的载入和存储。

② 一个 64 KB 的 CPU(或 BDM)局部映像，定义了特定的用于请求资源页(RPAGE、PPAGE 和 EPAGE)的寄存器和默认的指令集。这 64 KB 的地址空间在任何时候对于 CPU

(或 BDM)都是可见的。

以上 2 种存储器映像机制都可以实现存储器寻址,由 MMC 具体实现。下面首先介绍几个 MMC 模块中的用于实现分页机制的几个寄存器。

### (1) 全局页索引寄存器(Global Page Index Register, GPAGE)

GPAGE 地址为 0x0010,一共 8 位,复位值为 0x00,最高位读为 0,写无效。该寄存器用于索引 23 位全局地址镜像,只在 CPU 执行全局指令(GLDAA、GLDAB、GLDD、GLDS、GLDX、GLDY、GSTAA、GSTAB、GSTD、GSTS、GSTX、GSTY)时有效。23 位全局地址是由 GPAGE 和 CPU 局部地址组成,GPAGE 占全局地址的[22:16]位,CPU 局部地址占全局地址的[15:0]位,如图 9-1 所示。

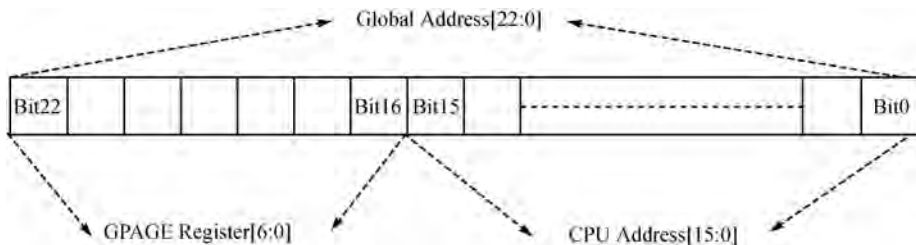


图 9-1 GPAGE 地址映像

从图 9-1 可以看出,GPAGE 寄存器的[6:0]和 CPU 局部地址[15:0]共 23 位,可以访问整个 23 位(8 MB)地址空间。

### (2) 程序页索引寄存器(Program Page Index Register, PPAGE)

PPAGE 地址为 0x0015,共 8 位,复位值为 0xFE。该寄存器与 CPU(或 BDM)局部地址的 16 KB P-Flash 窗口(0x8000\_0xBFFF)结合,用于访问 P-Flash 的全局地址。全局地址的最高位 22 位固定为 1,后 22 位如图 9-2 所示。全局地址由 PPAGE 和 CPU 局部地址组成,PPAGE 占全局地址的[21:14]位,CPU 局部地址占全局地址的[13:0]。

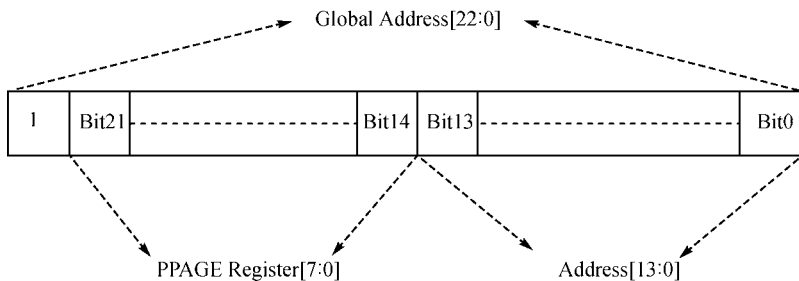


图 9-2 PPAGE 地址映像



### (3) 数据页索引寄存器(Data FLASH Page Index Register, EPAGE)

EPAGE 地址为 0x0017, 共 8 位, 复位值为 0xFE。该寄存器与 CPU(或 BDM)局部地址的 1 KB 的 D-Flash 窗口(0x0800\_0xBFFF)结合, 用于访问 D-Flash 的全局地址。全局地址高 5 位[22:18]固定为 0b00100, 其余位由 EPAGE 和 CPU 局部地址组成, EPAGE 占全局地址的[17:10], CPU 地址占全局地址的[9:0], 如图 9-3 所示。

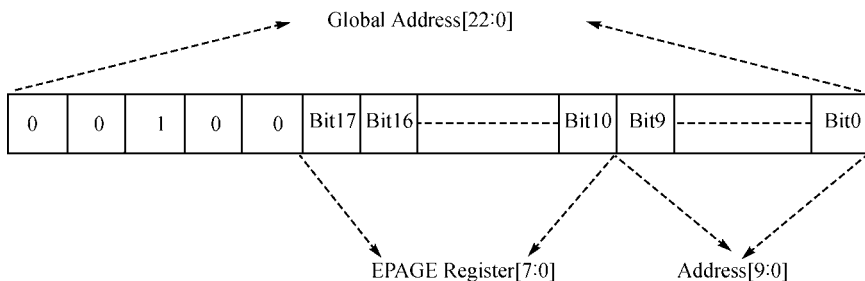


图 9-3 EPAGE 地址映像

CPU 在 64 KB 寻址空间中开出几个窗口, 利用上述几个页面寄存器, 使得 CPU 可以随时将 64 KB 以外的存储空间映射到 64 KB 中窗口中, 同时将暂时不用的那一块换出去, 以此方法扩展 XS128 的寻址空间。同样, 也可以利用这些页面寄存器来进行全局地址的访问。

对局部地址、逻辑地址和全局地址的正确理解是灵活运用 XS128 Flash 模块的基础。对于 16 位地址线的 MCU 系统来说, 局部地址就是传统意义上的 64 KB 的地址空间 0x0000~0xFFFF。逻辑地址即扩展的 24 位地址, 一般格式为 0xXX\_XXXX, 下划线“\_”前的两位 16 进制数为 PPAGE 或者 EPAGE 的值(页号), 下划线“\_”后的 4 位 16 进制数为对应的窗口地址。例如 0xFE\_8000, 0xFE 为 P-Flash 的页号, 0x8000 为 P-Flash 在 64 KB 存储空间中的窗口地址。全局地址就是数据存储的物理空间的地址, XS128 全局地址为 23 位, 即 0x00\_0000~0x7F\_FFFF。举例说明, 如果 PPAGE=0xFE, 那么当访问 0x8000~0xBFFF 这个局部地址空间中任何一个地址时, 其实访问的是逻辑地址区间 0xFE\_8000~0xFE\_BFFF, 而对应的全局地址区间为 0x7F\_8000~0x7F\_BFFF, 这两个地址是等效的, 只是两种不同的索引方式。再例如局部地址 0xFF0F 与全局地址 0x7F\_FF0F 指向同一空间, 对于无需窗口映射的地址, 局部地址和全局地址等效。

下面介绍局部地址与全局地址的逻辑对应关系, 如图 9-4 所示。XS128 的 P-Flash 对应的局部地址是 0x4000~0xFFFF, 其中 0x8000~0xBFFF 为窗口区间, 对应的全局地址空间是 0x7E\_0000~0x7F\_FFFF, 0x4000~0x7FFF 和 0xC000~0xFFFF 可以直接访问全局物理地址(分别为 0x7F\_4000~0x7F\_7FFF 和 0x7F\_C000~0x7F\_FFFF); D-Flash 对应的局部地址空间是 0x0800~0x0FFF, 其中 0x0800~0x0FFF 是用于 EPAGE 地址映射窗口, 此时该窗口对应的全局地址空间是 0x10\_0000~0x10\_1FFF。



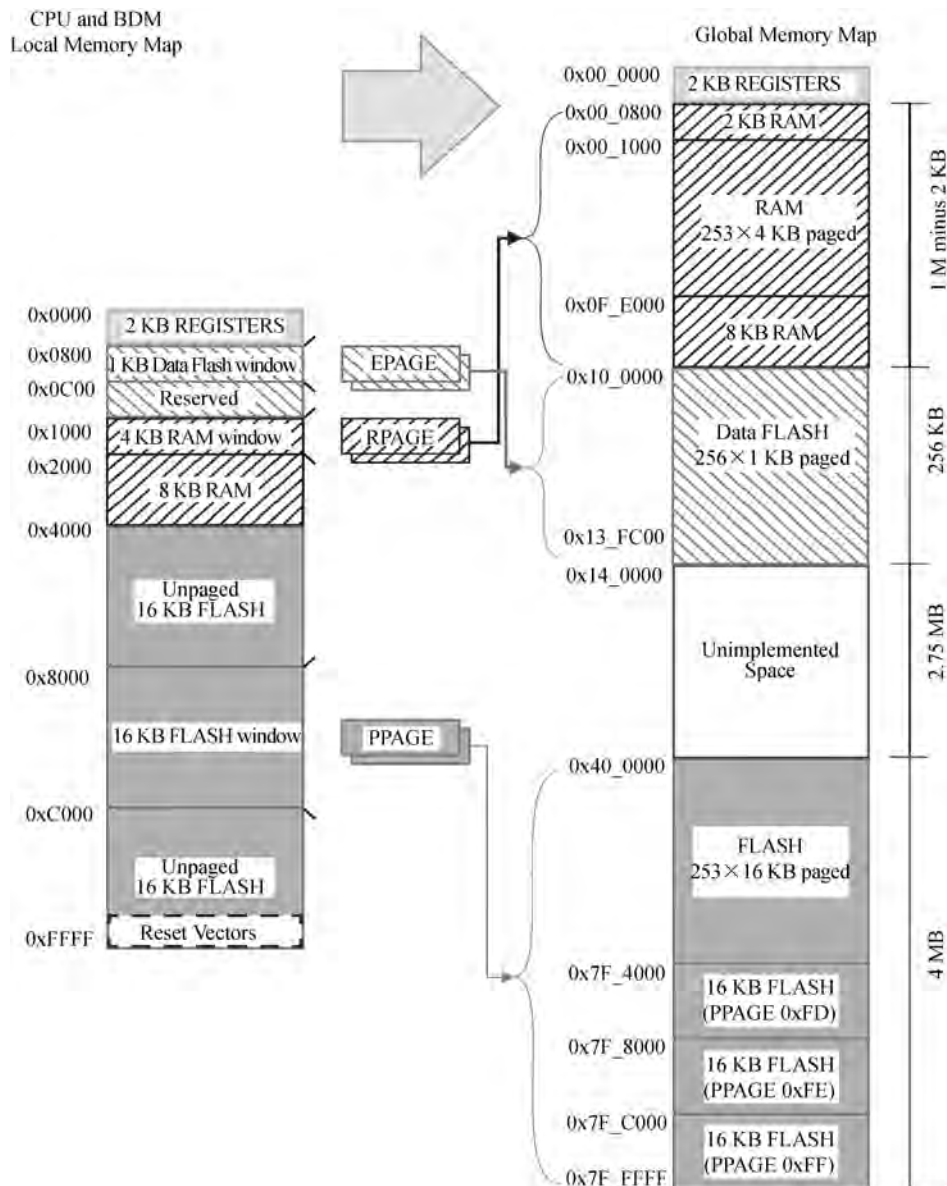


图 9-4 XS128 存储器局部地址与全局地址对应



D-Flash 具体的逻辑地址与全局地址的对应关系如表 9-1 所列。

表 9-1 D-Flash 逻辑地址与全局地址的对应关系

EPAGE	逻辑地址	全局地址
00	0x00_0800~0x00_0BFF	0x10_0000~0x10_03FF
01	0x01_0800~0x01_0BFF	0x10_0400~0x10_07FF
02	0x02_0800~0x02_0BFF	0x10_0800~0x10_0BFF
03	0x03_0800~0x03_0BFF	0x10_0C00~0x10_0FFF
04	0x04_0800~0x04_0BFF	0x10_1000~0x10_13FF
05	0x05_0800~0x05_0BFF	0x10_1400~0x10_17FF
06	0x06_0800~0x06_0BFF	0x10_1800~0x10_1BFF
07	0x07_0800~0x07_0BFF	0x10_1C00~0x10_1FFF

P-Flash 具体的逻辑地址与全局地址的对应关系如表 9-2 所列。

表 9-2 P-Flash 逻辑地址与全局地址的对应关系

PPAGE	逻辑地址	全局地址
F8	0xF8_8000~0xF8_BFFF	0x7E_0000~0x7E_3FFF
F9	0xF9_8000~0xF9_BFFF	0x7E_4000~0x7E_7FFF
FA	0xFA_8000~0xFA_BFFF	0x7E_8000~0x7E_BFFF
FB	0xFB_8000~0xFB_BFFF	0x7E_C000~0x7E_FFFF
FC	0xFC_8000~0xFC_BFFF	0x7F_0000~0x7F_3FFF
FD	0xFD_8000~0xFD_BFFF	0x7F_4000~0x7F_7FFF
FE	0xFE_8000~0xFE_BFFF	0x7F_8000~0x7F_BFFF
FF	0xFF_8000~0xFF_BFFF	0x7F_C000~0x7F_FFFF

## 2. 存储器映像控制(Memory Mapping Control, MMC)模块

地址的解析并寻址都由 XS128 内部的 MMC 模块管理和执行。图 9-5 是 MMC 自动运行的流程图。MMC 模块的主要功能是地址映射、控制 MCU 操作模式、多主体(MCU 和 BDM)优先级访问、内部资源的选择、控制内部总线(包括存储空间和外围资源)等。

从图 9-5 可以看出,对于任一个地址,不管是局部的 16 位地址,还是全局的 23 位地址,都会交由 MMC 进行解析,然后自动给 PPAGE 或 EPAGE 赋值并得到一个剩余的 16 位地址,这样 16 位机就可以直接进行计算地址空间并寻址了。如果没有这些寄存器,对于 16 位机处理 23 位地址就需要计算两次,通过这些的寄存器可以有效地提高寻址效率。而整个过程并

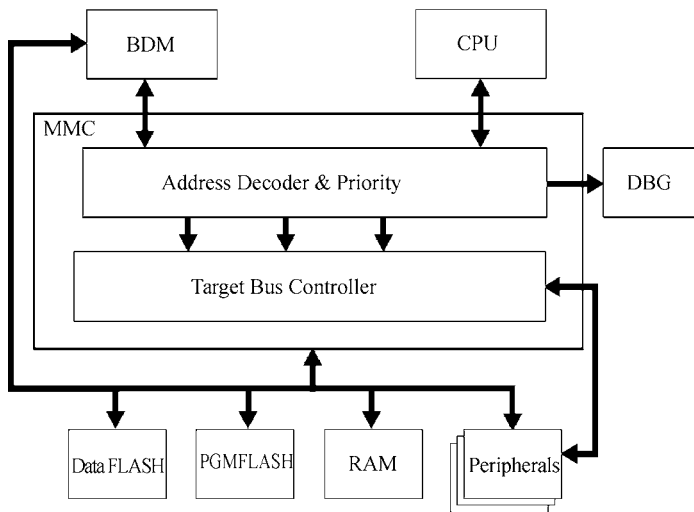


图 9-5 MMC 运行流程图

不需要用户特别关心,都由 MMC 自动完成。用户只需要提供正确的地址即可。

注:由于 Flash 模块只提供了全局地址设置寄存器,所以在编程时向相应的寄存器写入目标 Flash 存储空间的全局地址。

## 9.2 Flash 存储器编程方法

Flash 存储器一般作为程序存储器来使用,不能在运行时随时擦除、写入。当然,由于物理结构方面的原因,对 Flash 存储器的写操作更不能像对待一般 RAM 那样方便。在许多嵌入式产品开发中,需要使用掉电仍能保存数据的存储器来保存一些参数或重要数据,目前一般使用 EEPROM 来实现。S12X 系列的 Flash 存储器提供了用户模式下的在线编程功能,可以使用 Flash 存储器的一些区域来实现 EEPROM 的功能,简化了电路设计并节约了成本。但是,Flash 存储器的在线编程不同于一般的 RAM 读写,需要专门的过程,本节介绍 S12X 的 Flash 存储器的基本操作方法。

### 9.2.1 Flash 存储器编程的基本概念

虽然 Flash 存储器是一种快速的电可擦除、电可编程(写入)的存储器,但是基于其物理结构的原因,对 Flash 存储器的擦除及写入一般还是需要高于电源的电压。S12X 系列 MCU 的片内 Flash 存储器内含“升压电路”,使其能够在单一电源供电情况下进行擦除与写入。对 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。

对于 XS128 来说,对 Flash 存储器的擦除操作可以通过整体擦除操作来完成,也可以仅擦除一个扇区。也就是说,不能仅擦除某一字节或一次擦除小于一个扇区的内容。注意到这一特点,在数据安排时是十分重要的。XS128 的写入操作以半行(8 个字节)为基础,只能 8 个字节对齐的方式进行写入,详见 9.2.3 小节。

## 9.2.2 Flash 存储器的编程寄存器

从编程角度,Flash 存储器的擦除和写入操作主要用到寄存器有 5 个,分别是 Flash 存储器的时钟分频寄存器(FCLKDIV)、配置寄存器(FCNFG)、状态寄存器(FSTAT)、Flash 通用命令对象索引寄存器 FCCOBIX 及 Flash 通用命令对象寄存器 FCCOB。

### 1. Flash 时钟分频寄存器 FCLKDIV (Flash Clock Divider Register)

FCLKDIV 寄存器主要用于对擦除与写入算法时间的控制。在擦写之前,该寄存器必须被写;否则,不允许进行擦写操作。定义为:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	FDIVLD	FDIV[6 : 0]						
写								
复位	0	0	0	0	0	0	0	0

D7—FDIVLD:时钟分频被设置标志位。FDIVLD=1,当 FDIVLD 标志位位 1 时,则说明 FCLKDIV 寄存器已经被配置。FDIVLD=0,FCLKDIV 寄存器自从上次复位还没有被配置。

D6~D0:时钟分频设置位,可以从 OSCCLK 获得分频,一般需要为 Flash 模块准备 1 MHz (800 kHz~1.05 MHz)左右的频率。而 FDIV[6 : 0]的具体值可以由时钟分频对照表获得,详见表 9-3。

**注意:**当 FSTAT 寄存器的 CCIF 标志位为 0(即一个 Flash 命令正在执行)时,FCLKDIV 寄存器是不可以被配置的。

表 9-3 时钟分频对照表

OSCCLK Frequency/MHz		FDIV[6 : 0]	OSCCLK Frequency/MHz		FDIV[6 : 0]
MIN	MAX		MIN	MAX	
1. 60	2. 10	0x01	33. 60	34. 65	0x20
2. 40	3. 15	0x02	34. 65	35. 70	0x21
3. 20	4. 20	0x03	35. 70	36. 75	0x22
4. 20	5. 25	0x04	36. 75	37. 80	0x23

续表 9-3

OSCCLK Frequency/MHz		FDIV[6 : 0]	OSCCLK Frequency/MHz		FDIV[6 : 0]
MIN	MAX		MIN	MAX	
5.25	6.30	0x05	37.80	37.80	0x24
6.30	7.35	0x06	38.85	39.90	0x25
7.35	8.40	0x07	40.90	40.95	0x26
8.40	9.45	0x08	40.95	42.00	0x27
9.45	10.50	0x09	42.00	43.08	0x28
10.50	11.55	0x0A	43.05	44.10	0x29
11.55	12.60	0x0B	44.10	45.15	0x2A
12.60	13.65	0x0C	45.15	46.20	0x2B
13.65	14.70	0x0D	46.20	47.25	0x2C
14.70	15.75	0x0E	47.25	48.30	0x2D
15.75	16.80	0x0F	48.30	49.35	0x2E
16.80	17.85	0x10	49.35	50.40	0x2F
17.85	18.90	0x11			
18.90	19.95	0x12			
19.95	21.00	0x13			
21.00	22.05	0x14			
22.05	23.10	0x15			
23.10	24.15	0x16			
24.15	25.20	0x17			
25.20	26.25	0x18			
26.25	27.30	0x19			
27.30	28.35	0x1A			
28.35	29.40	0x1B			
29.40	30.45	0x1C			
30.45	31.50	0x1D			
31.50	32.55	0x1E			
32.55	33.60	0x1F			



## 2. Flash 配置寄存器 FCNFG (Flash Configuration Register)

FCNFG 寄存器用于设置允许 Flash 命令完成中断、忽视单位错误标志、强制双位错误检测和单位错误检测。定义为：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	CCIE	0	0	IGNSF	0	0	FDFD	FSFD
写								
复位	0	0	0	0	0	0	0	0

D7—CCIE:命令执行完中断允许位(Command Complete Interrupt Enable)。该位用于控制一个 Flash 命令完成时是否产生中断。CCIE=0,关闭中断;CCIE=1,启动命令完成中断。

D4—IGNSF:忽视单位错误位(Ignore Single Bit Fault)。该位用于当检测到单位错误时是否反映到 FERSTAT 寄存器。IGNSF=1,则忽视错误,不反映到 FERSTAT 寄存器。IGNSF=0,立即反映到 FERSTAT 寄存器。

D1—FDFD:强制两位错误检测位(Force Double Bit Fault Detect)。该位允许在 Flash 读操作期间产生两位错误而触发中断。通过写 FDFD 为 0,可以清除该位。FDFD=0,Flash 读操作只在两位错误被检测到时才置位 FERSTAT 寄存器中的 DFDIF 标志。FDFD=1,任何 Flash 读操作都会强制置位 FERSTAT 寄存器中的 DFDIF 标志,在 FERCNFG 寄存器中的 DFDIE 中断允许时,中断将被触发。

D0—FSFD:强制一位错误检测位(Force Single Bit Fault Detect)。该位允许在 Flash 读操作期间产生一位错误而触发中断。通过写 FSFD 为 0,可以清除该位。FSFD=0,Flash 读操作只在一位错误被检测到时才置位 FERSTAT 寄存器中的 SFDIF 标志。FSFD=1,任何 Flash 读操作都会强制置位 FERSTAT 寄存器中的 SFDIF 标志,在 FERCNFG 寄存器中的 SFDIE 中断允许时,中断将被触发。

## 3. Flash 状态寄存器 FSTAT (Flash Status Register)

FSTAT 寄存器定义了机器命令状态、Flash 阵列访问、保护及空白检测状态。定义为：

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	CCIF	0	ACCERR	FPVIOL	MGBUSY	RSVD	MGSTAT[1:0]	
写								
复位	1	0	0	0	0	0	0	0

D7—CCIF:命令完成中断标志位(Command Complete Interrupt Flag)。该位用于表示一个 Flash 命令是否已经完成,可以通过向该位写 1 来清除此位。CCIF=0,有 Flash 命令正在运行。CCIF=1,Flash 命令已经执行完毕。

D5—ACCERR:Flash 访问出错标志(Flash Access Error Flag)。该位表示有一个非法的访问发生,可能是非法的 Flash 命令或者是违例的命令写入序列。当 ACCERR 被置位的时候,CCIF 标志位不可以被清除载入一个新的命令。可以通过向 ACCERR 写入 1 来清除 ACCERR 标志,而写 0 没有任何效果。ACCERR=0,没有访问错误被侦测。ACCERR=1,有访问错误被侦测。

D4—FPVIOL:Flash 保护区侵犯标志(Flash Protection Violation Flag)。该标志表示一个尝试擦除或写入位于 P - FLASH 或者 D - FLASH 保护区域中的一个地址。可以通过向 FPVIOL 写 1 来清除该标志,写 0 对该位没有任何影响。当 FPVIOL 被置位时,载入一个命令或者执行一个命令写入序列是不可能的。当 FPVIOL=1,Flash 保护区域非法访问已经发生。FPVIOL=0,没有侦测到保护区域被非法访问。

D3—MGBUSY:存储器控制器忙标志位(Memory Controller Busy Flag)。该位表示存储器控制器当前的状态。MGBUSY=0,存储器控制器处于空闲状态。MGBUSY=1,存储器控制器正在执行一个 Flash 命令(CCIF=0)。

D2—RSVD:保留位。

D1~D0:在执行 flash 命令或者复位期间,如果一个错误被侦测,那么 MGSTAT[1 : 0]的一位或多位被置 1。

#### 4. Flash 通用命令对象索引寄存器 FCCOBIX (FCCOB Index Register)

Flash 通用命令对象索引寄存器用于索引 FCCOB 寄存器来实现不同的 Flash 命令操作。定义为:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	0	0	0	0	0	FCCOBIX[2 : 0]		
写								
复位	0	0	0	0	0	0	0	0

D2~D0:这个 FCCOBIX 位被用于选择 FCCOB 寄存器组中某个参数值被读或者写入。

#### 5. Flash 通用命令对象寄存器 FCCOB (Flash Common Command Object Register)

Flash 通用命令对象寄存器 FCCOB 是一个包含 6 个字的寄存器组,每个字为 16 位,分为高 8 位 FCCOB[15 : 8](FCCOBHI)、低 8 位 FCCOB[7 : 0](FCCOBLO)。通过 Flash 通用命令对象索引寄存器 FCCOBIX 的低 3 位 FCCOBIX[2 : 0]的值,索引每一个字,进行操作。也允许对 FCCOB 寄存器按字节操作。

### 9.2.3 FCCOB – NVM 命令模式

非易失存储器(Non – Volatile Memory,NVM)是 P – Flash、D – FLASH 或 ROM 的统称。XS128 的 NVM 命令模式是指利用 FCCOB 寄存器提供一个命令码及相关的参数,通过向 Flash 状态寄存器 FSTAT 的命令完成中断标志 CCIF 写 1,清除该位,开始新的 Flash 命令。用户通过查看该位是否为 1,判断命令是否已经执行完毕(CCIF=0,有 Flash 命令正在运行。CCIF=1,Flash 命令已经执行完毕)。

在 NVM 命令模式中,FCCOB 的一般命令格式如表 9 – 4 所列。

表 9 – 4 FCCOB – NVM 命令模式通用格式

FCCOBIX[2 : 0]	FCCOB 的高 8 位(HI)/低 8 位(LO)
000	HI=FCMD[7 : 0]定义 Flash 操作命令(见表 9 – 5)
	LO=“0”&. 全局地址[22 : 16]
001	HI=全局地址[15 : 8]
	LO=全局地址[7 : 0]
010	HI=数据 0[15 : 8]
	LO=数据 0[7 : 0]
011	HI=数据 1[15 : 8]
	LO=数据 1[7 : 0]
100	HI=数据 2[15 : 8]
	LO=数据 2[7 : 0]
101	HI=数据 3[15 : 8]
	LO=数据 3[7 : 0]

编程时只要依据具体的命令要求格式给 FCCOB 和 FCCOBIX 赋值就可以完成命令的载入操作,详细的可以参见后续章节关于具体擦除、写入操作的编程实例。

表 9 – 5 Flash 操作命令

FCMD[7 : 0]	含 义
0x01	校验 P – Flash 和 D – Flash 是否已擦除
0x02	校验 P – Flash 或 D – Flash(由全局地址决定)是否已擦除
0x03	校验 P – Flash 由指定地址开始的给定数目的字是否已擦除
0x04	读 P – Flash 第 0 页的专用 64 字节



续表 9-5

FCMD[7 : 0]	含 义
0x06	写 P-Flash 的一个指定块
0x07	写 P-Flash 第 0 页的专用 64 字节
0x08	擦除 P-Flash 和 D-Flash 的所有页
0x09	擦除 P-Flash 或 D-Flash(由全局地址决定)页
0x0A	擦除 P-Flash 指定扇区
0x0B	擦除 P-Flash 和 D-Flash 所有页来解密
0x0C	校验后门访问密钥
0x0D	设置用户权限
0x0E	设置域权限
0x10	校验地址所决定的 D-Flash 是否已擦除
0x11	写 D-Flash 的一个指定块
0x12	擦除 D-Flash 指定扇区

9.2.4 Flash 存储器的编程步骤

对于 Flash 的擦除或写入操作包括以下 4 大步骤：

① 对 Flash 预分频寄存器 FCLKDIV 进行设置。

**注意：**如果总线时钟低于 1 MHz，那么 Flash 的擦除、写入操作是无法成功完成的。如果设置 FDIV 太高有可能会损毁 Flash 存储模块。如果设置太低，则擦除、写入 Flash 存储单元可能不完全。建议参考表 9-3 时钟分频对照表，选择合适的时钟分频。

② 对 FCCOB 和 FCCOBIX 根据需要设置相应的命令及参数。

③ 置位 FSTAT 寄存器中 CCIF 位来使命令生效。

④ 判断命令执行过程有无错误产生。

如图 9-6 所示，在具体的 Flash 构件编程时，只需按照流程图擦除、写入 Flash。

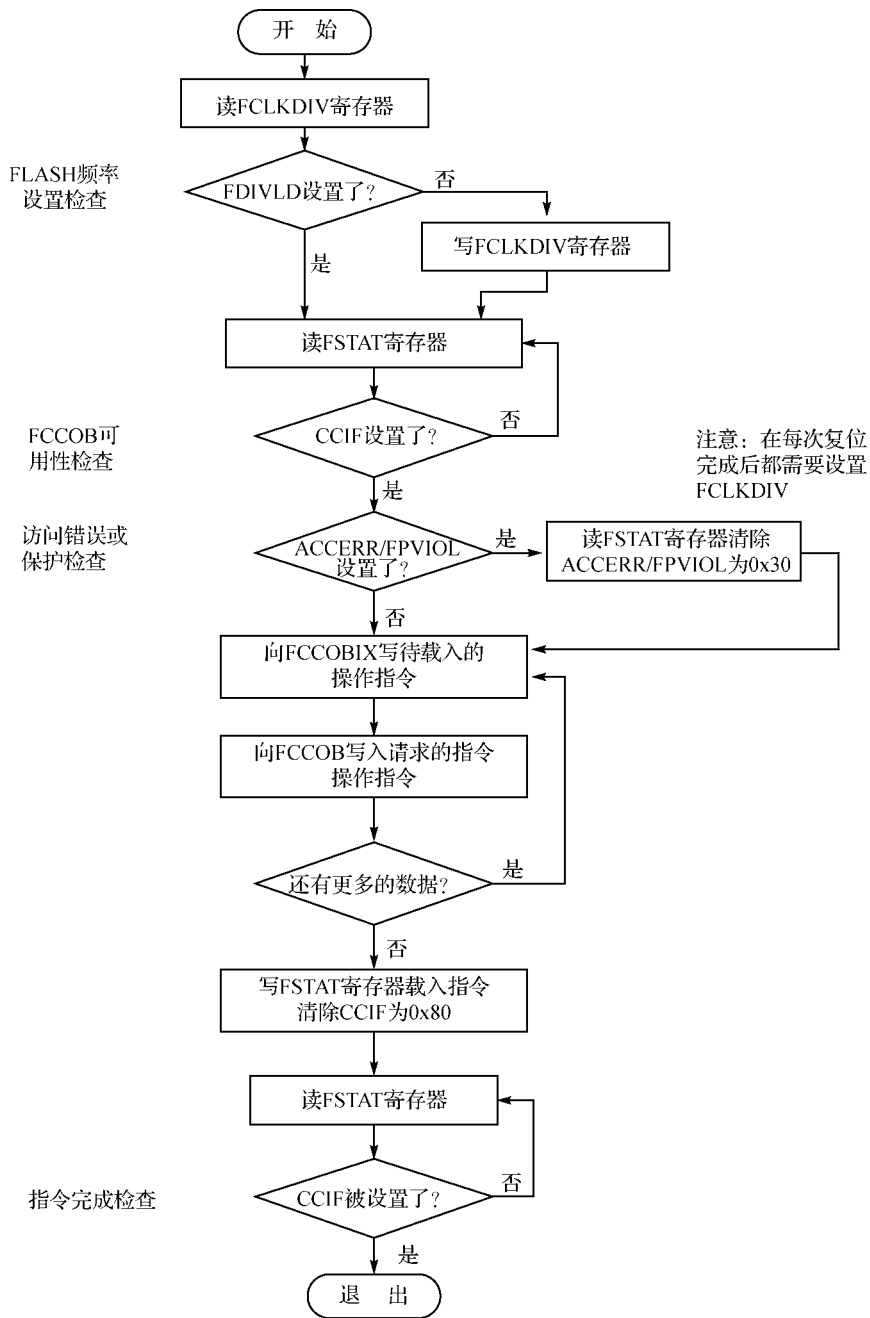


图 9-6 一般的 Flash 写入流程图

## 9.3 D-Flash 在线编程

### 1. D-Flash 编程准备

对于 XS128 而言,它有 8 KB 大小的数据 Flash(D-Flash)空间,可分为 8 页,每页 1 KB。编程可以擦除的最小单位是一个扇区,大小为 256 字节,D-Flash 共有 32 个扇区。详细的扇区分配如下(参见网上光盘该工程下的 EEPROM.h 头文件):

```
/* D-Flash 存储映像说明
1. XS128 D-Flash 存储地址为:0x100000 ~ 0x101FFF(GLOBAL ADDRESS).
2. D-FLASH 总大小为 8 KB,分为 32 个扇区,每个扇区大小 256 字节,可擦除的单位就是一个扇区.
//Flash 页编号对应表
// -----*
//扇区号      扇区地址                                     *
// 0          0x100000 - 0x1000FF,                         *
// 1          0x100100 - 0x1001FF,                         *
// 2          0x100200 - 0x1002FF,                         *
// 3          0x100300 - 0x1003FF,                         *
... .. //部分代码已省略,详见工程
// 30         0x101E00 - 0x101EFF,                         *
// 31         0x101F00 - 0x101FFF,                         *
// -----*
*/
```

### 2. 擦除和写入流程的一些公共操作

对于多分页机制的存储模式,需要设计一个函数用于扇区和块与全局地址之间的转换,对于 D-Flash 的擦除和写入需要先对预分频寄存器进行设置,在 Flash 命令执行之后,需要对错误标志进行检测来判断命令是否执行成功。

下面给出了这几个公共操作的具体实现过程:

```
// -----*
//函数名: Flash_D_PreInit                                *
//功 能: 用于 FLASH 操作前的环境设置                    *
// -----*

void Flash_D_PreInit()
{
    if(FCLKDIV_FDIVLD != 1)
        FCLKDIV |= 0x0F; //STEP1.DIV FROM 16MHz TO 1MHz. 0x0F
```



```

while(FSTAT_CCIF!= 1); //STEP2. 判断有无命令正在进行,有则等待完毕
if( 0 != ( FSTAT & 0x30 ) )    //STEP3. ACCERR FPVIOL
{
    FSTAT = 0x30;        //清除 ACCERR OR FPVIOL
}
}
// -----*
//函数名: Flash_D_CMDRunIsOK                                     *
//功 能: 用于判断命令是否执行成功                               *
// -----*
uint8 Flash_D_CMDRunIsOK()
{
    if((FSTAT_ACCERR == 1)|(FSTAT_MGSTAT1 == 1)|(FSTAT_MGSTAT0 == 1))
    {
        return FAILED;                //验证失败
    }
    else
    {
        return SUCCEEDED;            //验证通过
    }
}

// -----*
//函数名: CalAddressBySector                                     *
//功 能: 根据扇区号和块内序号,计算出需要操作的块。             *
//参 数: secNum:扇区号(0 ~ 31)                                   *
//      StartPos:块号(0 ~ 31).一个扇区中的块,8字节一分。       *
//返 回: 返回地址,偏移地址,没有全局地址前缀 0x10              *
//说 明: 无                                                       *
// -----*
uint16 CalAddressBySector(uint8 SecNum,uint8 StartPos)
{
    // D-FLASH 每 256 bytes 一个扇区,每 8 bytes 一个块
    return 0x0000 + SecNum * 256 + StartPos * 8;
}

```

### 3. 擦除子程序

根据扇区号计算出扇区首地址,将需要擦除扇区(扇区是可擦除的最小基本单位)的首地址和 D-FLASH 扇区擦除命令 CMD\_D\_ERASE\_SECTOR(即 0x12)通过 NVM 命令模式载

入到 FCCOB 寄存器中,执行擦除命令。下面给出了扇区擦除子程序。

```
// -----*
//函数名: Flash_D_EraseSector *
//功 能: Flash 扇区擦除 *
//参 数: secNum;扇区号(0 ~ 31) *
//返 回: 0x00 - 成功,0xFF - 失败 *
//说 明: 无 *
// -----*

uint8 Flash_D_EraseSector(uint8 SecNum)
{
    uint16 address;
    address = CalAddressBySector(SecNum,0); //获得扇区所在的起始地址
    Flash_D_PreInit();
    FCCOBIX = 0x00; //载入指令
    FCCOBHI = CMD_D_ERASE_SECTOR; //擦出扇区的命令,0x12
    FCCOBLO = 0x10; //全局地址,GPAGE = 0x10
    FCCOBIX = 0x01; //设置查出地址
    FCCOB = address;
    FSTAT = 0x80; //开始运行
    while(!(FSTAT&0x80));
    return Flash_D_CMDRunIsOK();
}
```

#### 4. 写入子程序

根据扇区号和偏移块计算出待写入的首地址,将需要写入的首地址和 D-Flash 写入命令 CMD\_D\_PROGRAM(即 0x11)通过 NVM 命令模式载入到 FCCOB 寄存器中,执行写入命令。下面给出了写入子程序。

```
// -----*
//函数名: Flash_D_Write4Word *
//功 能: Flash 扇区写入,一次写入 4 个字 *
//参 数: SecNum;扇区号(0 ~ 31) *
//      StartPos;块号(0 ~ 31).一个扇区中的块,8 字节一分。 *
//      Data[]:将要写入的数据,4 个字 *
//返 回: 0x00 - 成功,0xFF - 失败 *
//说 明: 无 *
// -----*

uint8 Flash_D_Write4Word(uint8 SecNum,uint8 StartPos,uint16 Data[])
{

```



```

uint16 address;
address = CalAddressBySector(SecNum, StartPos);
Flash_D_PreInit();
FCCOBIX = 0x00;           //载入指令
FCCOBHI = CMD_D_PROGRAM;  //写入命令, 0x11
FCCOBLO = 0x10;           //全局地址, GPAGE = 0x10
FCCOBIX = 0x01;
FCCOB = address;
FCCOBIX = 0x02;           //写入四个字的数据, 块写入方式
FCCOB = Data[0];
FCCOBIX = 0x03;
FCCOB = Data[1];
FCCOBIX = 0x04;
FCCOB = Data[2];
FCCOBIX = 0x05;
FCCOB = Data[3];
FSTAT = 0x80;             //开始运行
while(!(FSTAT & 0x80));    //判断是否结束
return Flash_D_CMDRunIsOK();
}

```

## 5. 读取数据

关于对 Flash 区域内容的读操作, 这里需要特别补充说明。对 Flash 读取可以采用以下 3 种地址方式:

① 通过局部地址(Local Address)进行访问, 即通过 64 KB 地址空间进行访问。

例如“Data = \*(volatile uint8 \*)0x4000;”, 这里 0x4000 是 P-Flash 的未分页的局部地址首地址, 可通过局部地址直接访问。但是 D-Flash 的未分页局部地址空间 0x0C00 ~ 0x0FFF 被保留, 所以 XS128 的 D-Flash 不可通过局部地址访问。

② 通过逻辑地址(Logical Address)进行访问, 即由 EPAGE 配合分页窗体地址范围 0x0800 ~ 0x0C00, 可以通过“\_\_eptr”的方式进行访问, 注意是两个下划线。例如: “Data = \*(volatile uint8 \* \_\_eptr)0x000800;”。

③ 通过全局地址进行访问。即按照整个存储器的实际物理位置进行访问, 可以通过“\_\_far”的方式进行访问, 注意也是两个下划线。例如: “Data = \*(volatile uint8 \* \_\_far)0x100000;”。

只有准确地理解这 3 种方式, 才能正确地运用这不同的地址形式对存储器进行相应的访问方式。

下面给出的是读取 D-Flash 子程序(通过全局地址进行访问):

```
// ----- *
// 函数名: Flash_ReadBlock *
// 功 能: Flash 扇区读取,一次读取 8 个字节出来。 *
// 参 数: secNum:扇区号(0 ~ 31) *
//      StartPos:块号(0 ~ 31).一个扇区中的块,8 字节一分。 *
//      resultData:读出的数据存放区。8 个字节大小 *
// 返 回: 无 *
// ----- *

void Flash_D_ReadBlock(uint8 SecNum,uint8 StartPos,uint8 resultData[])
{
    uint16 address;
    uint8 i;
    address = CalAddressBySector(SecNum,StartPos);
    for( i = 0; i < 8; i++ )
        resultData[i] = *(volatile uint8 * __far)(1048576 + address + i);
    //获得全局地址 1048576 = 0x100000
}

```

## 6. 擦除和写入编程要点说明

使用 Flash 在线编程技术可以省去外接 EEPROM,不仅简化了电路设计,也提高了系统稳定性。由于 XS128 的 Flash 都基于分页机制,并且当 FCCOBIX=001,FCCOB 开始载入 23 位地址的低 16 地址。且全局地址[2:0]=000,否则会报 ACCERR 错误。即对于编程而言,FCCOB 可以写入的全局地址能被 8 整除。

举例说明:如图 9-7 所示,用户可以写入 FCCOB 中地址只可以为 0x100000、0x100008、0x100010、0x100018……,写入其他地址(全局地址[2:0]不全为 0)都是非法的。这是由于 flash 的数据操作的最小单位是块,使得待操作的全局地址低 3 位[2:0]=000。这样对于 D-FLASH 而言,就有 32 个扇区,而每个扇区又有 32 个块,基于块进行写入操作。

另外,在使用 BDM 在线调试的时候,要注意这两个地址 GLOBAL ADDRESS 和 LOCAL ADDRESS 地址之区分,否则可能无法看到真正的结果。如图 9-7 左边地址后面都有一个 L,说明这个地址是局部逻辑地址,可以通过鼠标右键菜单切换到全局地址模式,如图 9-8 所示。调试的时候一定要注意到这一点,否则就有可能发生一些错误认知。

再者就是要注意,扇区是可以擦除的最小单位,对于 D-Flash,可以最小操作的大小就是 256 字节。

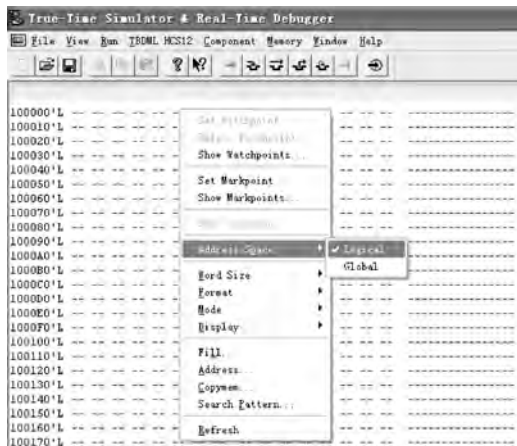


图 9-7 BDM 在线调试中的 Memory 窗体

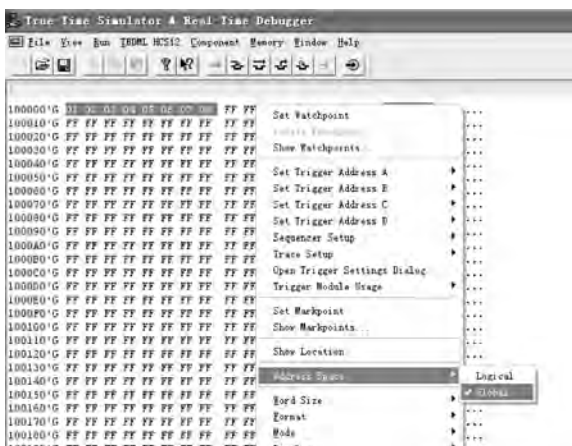


图 9-8 BDM 在线调试图

## 9.4 P-Flash 在线编程

### 1. P-Flash 编程准备

对于 XS128 而言,它有 128 KB 大小的数据 Flash(P-Flash)空间,可分为 8 页,每页 16 KB。编程可以擦除的最小单位是一个扇区,大小为 1024 字节,P-Flash 共有 128 个扇区块。详细的扇区分配如下(参见网上光盘该工程下的 **Flash.h** 头文件):

/\*

P-Flash 存储映像说明

1. XS128 P-Flash 存储地址为:0x7E0000 ~ 0xFFFFF(GLOABL ADDRESS)。

2. P-Flash 总大小为 128 KB,分为 128 个扇区,每个扇区大小为 1 KB 可擦除的最小单位就是一个扇区

//Flash 页编号对应表

// ----- \*

//扇区号      扇区地址      \*

// 0      0x7E0000 - 0x7E03FF,      \*

// 1      0x7E0400 - 0x7E07FF,      \*

// 2      0x7E0800 - 0x7E0BFF,      \*

// 3      0x7E0C00 - 0x7E0FFF,      \*

... //详细见工程代码文件

// 124      0x7FF000 - 0x7FF3FF,      \*

// 125      0x7FF400 - 0x7FF7FF,      \*



```
// 126      0x7FF800 - 0x7FFBFF,      *
// 127      0x7FFC00 - 0x7FFFFF,      *
// -----*
* /
```

## 2. 擦除和写入流程的一些公共操作

对 Flash 存储器的擦除或写入代码本身是不可以放在被擦除或写入的区域,可以通过将代码转移到其他存储器块中或者直接复制到内存 RAM 中。对于数据 Flash(D-FLASH)的擦除或写入不需要特别的操作(如把执行擦除或写入的代码放在 RAM 中执行),这是由于执行代码并不在 D-FLASH 中,而在 P-FLASH 中。然而,对于 P-FLASH 的相关操作,却必须要将执行代码存放在 RAM 中运行,这样才能正确擦除或写入。详见下面的讲解:

```
FSTAT = 0x80;          //开始运行
while(!(FSTAT&0x80));   //判断是否结束
```

上面这两句代码对应的二进制代码如下:0xC6,0x80,0x7B,0x01,0x06,0x1F,0x01,0x06,0x80,0xFB。而 0x3D 是 RTS 命令的二进制代码,它可以让程序从 RAM 返回到程序进入 RAM 前的位置处继续执行。

对于多分页机制的存储模式,需要设计一个函数用于扇区和块与全局地址之间的转换,对于 P-Flash 的擦除和写入需要先对预分频寄存器进行设置,在 Flash 命令执行之后,需要对错误标志进行检测来判断命令是否执行成功。

这些操作大多都类似于 D-Flash,与 D-Flash 不同的是 P-Flash 的执行代码需要放置到 RAM 中。

```
uint8 PrgOfRam[11] = {          //擦写程序机器码
0xC6,0x80,0x7B,0x01,0x06,
0x1F,0x01,0x06,0x80,0xFB,
0x3D                          //从 RAM 返回到程序
};
```

上面是定义了一个数组,存放执行代码。

```
asm("JSR PrgOfRam");           //转到内存运行
```

上面用到了汇编指令 JSR,可以使程序跳转到变量位置,即 RAM 中继续运行。下面给出了 P-Flash 擦除、写入命令的几个公共操作的具体 C 语言实现代码。

```
// -----*
//函数名: Flash_P_PreInit      *
//功 能: 用于 FLASH 操作前的环境设置      *
```



```
// ----- *
void Flash_P_PreInit()
{
    if( FCLKDIV_FDIVLD != 1 )
    FCLKDIV |= 0x0F;          //STEP1.DIV FROM 16MHz TO 1MHz
    while( FSTAT_CCIF != 1 );//判断有无命令正在进行,有则等待执行完毕
    if( 0 != ( FSTAT & 0x30 ) )      //STEP3. ACCERR FPVIOL
    {
        FSTAT = 0x30;              //清除 ACCERR OR FPVIOL
    }
}

// ----- *
//函数名: Flash_P_CMDRunIsOK                      *
//功 能: 用于判断命令是否执行成功                  *
// ----- *
uint8 Flash_P_CMDRunIsOK()
{
    if((FSTAT_ACCERR == 1)|(FSTAT_MGSTAT1 == 1)|(FSTAT_MGSTAT0 == 1))
    {
        return FAILED;            //验证失败
    }
    else
    {
        return SUCCEEDED;        //验证通过
    }
}

// ----- *
//函数名: CalAddressBySector                      *
//功 能: 根据扇区号和块内序号,计算出需要操作的块。      *
//参 数: secNum:扇区号(0 ~ 127)                    *
//      StartPos:块号(0 ~ 127).一个扇区中的块,8字节一分。 *
//返 回: 返回地址,偏移地址,没有全局地址前缀 0x10      *
//说 明: 无                                              *
// ----- *
uint16 CalAddressBySector(uint8 SecNum,uint8 StartPos)
{
    if( SecNum < 64 )
```

```

{
return 0x0000 + SecNum * 1024 + StartPos * 8;
}
else
{
return 0x0000 + (SecNum - 64) * 1024 + StartPos * 8;
}
}
}

```

### 3. 擦除子程序

根据扇区号计算出扇区首地址,将需要擦除扇区(扇区是可擦除的最小基本单位)的首地址和 P-Flash 扇区擦除命令 CMD\_P\_ERASE\_SECTOR(即 0x0A)通过 NVM 命令模式载入到 FCCOB 寄存器中,执行擦除命令。下面给出了扇区擦除子程序代码:

```

// ----- *
//函数名: Flash_P_EraseSector *
//功 能: Flash 扇区擦除 *
//参 数: secNum;扇区号(0 ~ 127) *
//返 回: 0x00 - 成功,0xFF - 失败 *
//说 明: 无 *
// ----- *

uint8 Flash_P_EraseSector(uint8 SecNum)
{
uint16 address;
address = CalAddressBySector(SecNum,0); //获得扇区所在的起始地址
Flash_P_PreInit();
FCCOBIX = 0x00; //载入指令
FCCOBHI = CMD_P_ERASE_SECTOR; //擦出扇区的命令
//全局地址,GPAGE = 0x10
if( SecNum < 64 ) //分对开处理,共 127
{
FCCOBLO = 0x7E;
}
else
{
FCCOBLO = 0x7F;
}
FCCOBIX = 0x01; //设置擦除地址
FCCOB = address;

```



```
asm("JSR PrgOfRam");
return Flash_P_CMDRunIsOK();
}
```

#### 4. 写入子程序

根据扇区号和偏移块计算出待写入的首地址,将需要写入的首地址和 P-Flash 写入命令 CMD\_D\_PROGRAM(即 0x06)通过 NVM 命令模式载入到 FCCOB 寄存器中,执行写入命令。下面给出了写入子程序。

```
// ----- *
//函数名: Flash_P_Write4Word *
//功 能: Flash 扇区写入,一次写入 4 个字 *
//参 数: SecNum:扇区号(0 ~ 127) *
//      StartPos:块号(0 ~ 127).一个扇区中的块,8 字节一分。 *
//      Data[]:将要写入的数据,4 个字 *
//返 回: 0x00 - 成功,0xFF - 失败 *
//说 明: 无 *
// ----- *

uint8 Flash_P_Write4Word(uint8 SecNum,uint8 StartPos,uint16 Data[])
{
    uint16 address;
    address = CalAddressBySector(SecNum,StartPos);

    Flash_P_PreInit();

    FCCOBIX = 0x00; //载入指令
    FCCOBHI = CMD_P_PROGRAM; //写入命令
                                //全局地址,GPAGE

    if( SecNum < 64 )
    {
        FCCOBL0 = 0x7E;
    }
    else
    {
        FCCOBL0 = 0x7F;
    }

    FCCOBIX = 0x01;
    FCCOB = address;
```

```

FCCOBIX = 0x02;           //写入四个字的数据,块写入方式
FCCOB = Data[0];
FCCOBIX = 0x03;
FCCOB = Data[1];
FCCOBIX = 0x04;
FCCOB = Data[2];
FCCOBIX = 0x05;
FCCOB = Data[3];

asm("JSR PrgOfRam");

return Flash_P_CMDRunIsOK();
}

```

## 5. 读取写入数据

同 D-Flash, 参见 9.3.5 小节读取写入数据, 但是对于通过逻辑地址进行访问的方式有点儿差异。对于 D-Flash, 使用的关键字是“\_\_eptr”, 而对于 P-Flash 需要使用关键字“\_\_pptr”, 其他都一样。

## 6. 擦除和写入编程要点说明

大多数操作和 D-Flash 一样, 需要特别强调的就是它需要对执行代码进行特别处理, 将部分代码编译成机器码后存放在 RAM 中运行。

再者, D-Flash 的最小单位为 256 字节, 而 P-Flash 为 1024 字节, 用户可以根据这些大小, 合理选择需要擦除的扇区以及保证数据放置位置的合理性。

# 9.5 Flash 存储器的保护特性和安全性

## 9.5.1 Flash 存储器的配置区域

飞思卡尔系列芯片都有一段区域是用于特殊寄存器配置的, 包括后门比对密钥、Flash 保护字节、Flash 安全字节和 Flash 非易失性字节。XS128 也不例外, 也存在一段配置区域, 如表 9-6 所列。

在 XS128 中, 一些寄存器很特别, 不可以直接给它们赋值或者说是直接配置, 因为它们只是只读的, 这样的寄存器值都是通过系统复位时从配置区域载入对应的值, 如 FPROT、DFPROT、FOPT、FSEC, 这些后面将会介绍。



表 9-6 Flash 存储器配置

全局地址	大 小	描 述
0x7F_FF00~0x7F_FF07	8	后门比对密钥
0x7F_FF08~0x7F_FF0B	4	保留区域
0x7F_FF0C	1	P-Flash 保护字节。参考 FPROT
0x7F_FF0D	1	D-Flash 保护字节。参考 DFPROT
0x7F_FF0E	1	FLASH 非易失性字节。参考 FOPT
0x7F_FF0F	1	FLASH 安全字节。参考 FSEC

## 9.5.2 Flash 存储器的保护特性

### 1. Flash 保护特性简介

由于 Flash 是非易失性存储器,所以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动作,XS128 芯片提供了对 Flash 整体或局部的保护机制。如果一个 Flash 区域被保护,那么是不能对这块区域进行擦写动作的。

与 XS128 Flash 存储器保护特性相关的寄存器是 P-Flash 保护寄存器 FPROT(Flash Protection Register)和 D-Flash 保护寄存器 DFPROT(D-Flash Protection Register)。通过对保护寄存器进行设置,可以确保保护的区域不可以被擦除或写入。

### 2. P-Flash 保护特性及其保护寄存器 FPROT (Flash Protection Register)

用于负责 P-Flash 保护功能的寄存器是 FPROT 寄存器,该寄存器可以用于设置 P-Flash 保护区域来避免意外的擦除和写入操作。P-Flash 可以大体再分为 3 个区域:更低区域(Lower region)、更高区域(Higher region)和剩余区域。图 9-9 是 P-Flash 存储器映像图,P-Flash 的保护区域就是基于 3 个区域的分址方式进行的。在复位的时候该寄存器的值直接从配置区域的 0x7F\_FF0C 地址处载入内容。

如果强制擦除或写入保护区域,那么寄存器 FSTAT 中的 FPVIOL 位将会被置位。如果需要解除保护,那么确保 P-Flash 最后一个扇区(即保护字节所在的那个扇区)是不在保护范围的,然后重写保护字节,即可通过相应的配置解除保护。

FPROT 寄存器的具体定义使用如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	FPOPEN	RNV6	FPHDIS	FPHS[1:0]		FPLDIS	FPLS[1:0]	
写								
复位	F	F	F	F	F	F	F	F

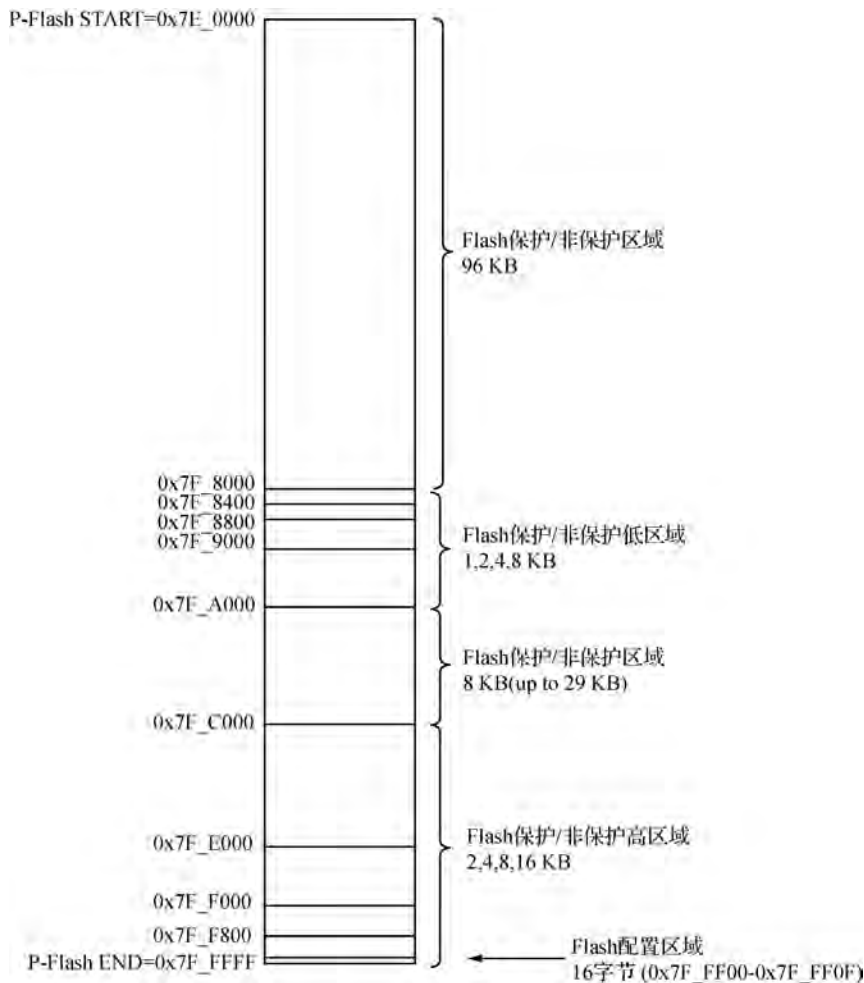


图 9-9 P-Flash 存储器映像图

通过表 9-7、表 9-8、表 9-9 可以设置不同的保护功能。为了实现保护功能须设置保护字段,但是需要注意的是 FPROT 寄存器不可以直接配置,而只能通过配置区域的映射进行设置,即配置字段只能赋值给 0x7F\_FF0C 地址。



表 9-7 P-Flash 保护功能

FPOPEN	FPHDIS	FPLDIS	功能描述
1	1	1	P-Flash 没有被保护
1	1	0	只有低区域被保护
1	0	1	只有高区域被保护
1	0	0	只有高、低区域被保护
0	1	1	P-FLASH 都被保护
0	1	0	只有低区域不被保护
0	0	1	只有高区域不被保护
0	0	0	只有高、低区域不被保护

表 9-8 P-Flash 高地址区域保护范围对应表

FPHS[1:0]	全局地址范围	保护大小
00	0x7F_F800—0x7F_FFFF	2 KB
01	0x7F_F000—0x7F_FFFF	4 KB
10	0x7F_E000—0x7F_FFFF	8 KB
11	0x7F_C000—0x7F_FFFF	16 KB

表 9-9 P-Flash 低地址区域保护范围对应表

FPLS[1:0]	全局地址范围	保护大小
00	0x7F_8000—0x7F_83FF	1 KB
01	0x7F_8000—0x7F_87FF	2 KB
10	0x7F_8000—0x7F_8FFF	4 KB
11	0x7F_8000—0x7F_9FFF	8 KB

### 3. D-Flash 保护特性及其保护寄存器 DFPROT (D-Flash Protection Register)

用于负责 D-Flash 保护功能的寄存器是 DFPROT 寄存器,该寄存器可以用于设置 D-Flash 保护区域来避免意外的擦除或写入操作。图 9-10 是 D-Flash 存储器映像图。在复位时 DFPROT 寄存器的值直接从配置区域的 0x7F\_FF0D 地址处载入内容。

如果强制擦除或写入保护区域,那么寄存器 FSTAT 中的 FPVIOL 位将会被置位。如果需要解除保护,那么确保 P-Flash 最后一个扇区(即保护字节所在的那个扇区)是不在保护范围的,然后重写保护字节,即可通过相应的配置解除保护。

DFPROT 寄存器的具体定义使用如下:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	DPOPEN	0	0	DPS[4:0]				
写								
复位	F	0	0	F	F	F	F	F

D7—DPOPEN(D-Flash Protection Control)。DPOPEN=1,关闭 D-Flash 保护功能。DPOPEN=0,使用 DPS[4:0]设定的保护地址范围,范围下界是 0x10\_0000,DPS[4:0]确定上界全局地址 12~8 位的值,而 7~0 位系统自动设为 0xFF,高位 22~13 位系统自动设为 0b0010000000。



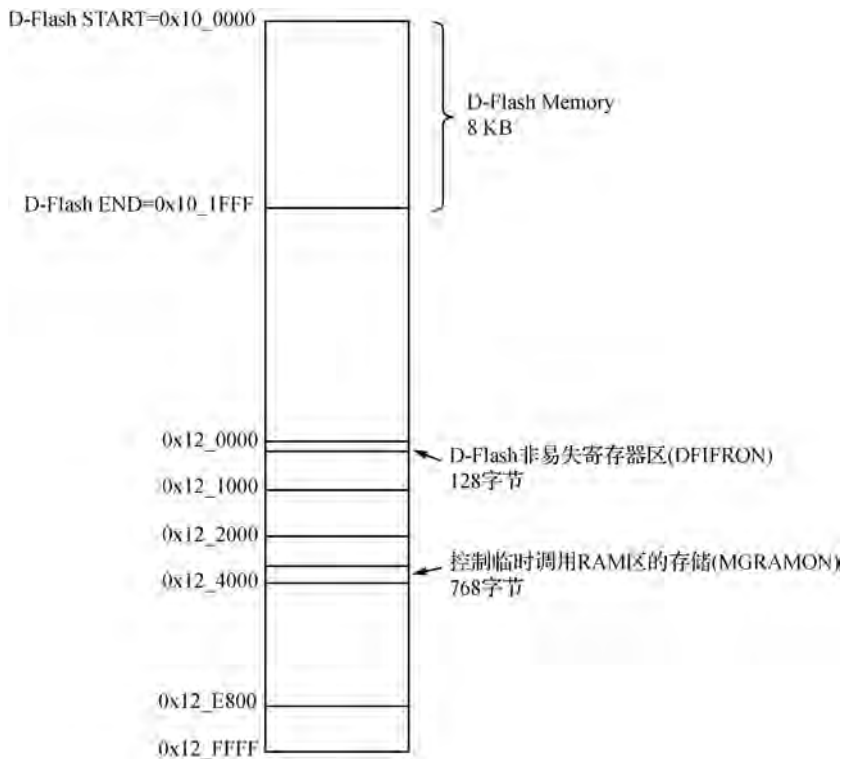


图 9-10 D-Flash 存储器映像图

### 9.5.3 Flash 存储器的安全性

XS128 中加入的调试模块增加了芯片的实用性,但却给芯片安全带来了隐患。因为普通用户(这里把程序的使用者但非拥有者称作普通用户)可能会轻而易举地通过 BDM 把数据从芯片中盗出。为了数据保密,XS128 引入了复杂的保密机制来确保芯片的安全性。当 MCU 保密后,普通用户不能通过 BDM 读出存储器中的任何内容(读出为乱码);而在芯片中运行的程序本身,可以对 MCU 的任意资源进行访问。

#### 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。涉及的寄存器是安全寄存器 FSEC(Flash Security Register),复位时,FSEC 寄存器从配置地址 0x7F\_FF0F 处自动载入值。FSEC 寄存器的所有位都与设备的安全性相关,并且这些位是可读但不可写的。定义为:



数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	KEYEN[1 : 0]		RNV[5 : 2]				SEC[1 : 0]	
复位	F	F	F	F	F	F	F	F

KEYEN[1 : 0]主要用于使能后门密钥比对功能,具体设置值参考表 9-10。

SEC[1 : 0]主要用于展示 MCU 的安全状态,具体设置值参考表 9-11。如果 Flash 模块被后门密钥比对临时解密,那么 SEC[1 : 0]会被强制为 10,即解密状态。

表 9-10 Flash KEYEN 功能对照表

KEYEN[1 : 0]	后门密钥功能启用状态
00	关闭
01	关闭
10	使能
11	关闭(默认值)

表 9-11 Flash SEC 功能对照表

SEC[1 : 0]	MCU 安全状态
00	安全
01	安全
10	不安全(即被解密状态)
11	安全(默认值)

只要把局部地址 0xFF0F 处字节设置为 0x00、0x01 或 0x03,MCU 就会被加密。编程时只需在中断处理文件中加入如下代码:

```
const uint8 Secure@0xFF0F = 0x00; //把局部地址 0xFF0F 处字节修改为 0x00
```

## 2. 解除 MCU 保密状态

MCU 被加密时,芯片不能够进行正常的擦除与写入操作,出现如图 9-11 芯片被加密情况。



图 9-11 芯片被加密

可以通过擦除整个 Flash 来解除 MCU 保密状态。打开 ~\Freescall\CodeWarrior for S12(X)V5.0\Prog\hiwave.exe,选择 Component→Set Connection 菜单项,则弹出如图 9-12 所示的设置连接写入器界面。

在 Connection 栏中选择 TBDML,然后单击 OK。在接下来的对话框中选择 5V,则弹出如图 9-13 所示的设置晶振频率界面。



图 9-12 设置连接写入器界面

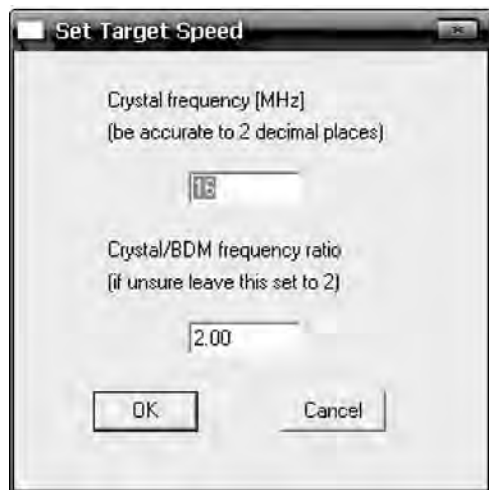


图 9-13 设置晶振频率界面

因为所使用的晶振为 16 MHz,所以在图 9-13 中选中的文本框中写入 16。单击 OK 后选择 MC9S12XS128 芯片。再单击 OK 后,选择 TBDML HCS12→Command Files 菜单项,则弹出如图 9-14 所示的设置解密命令文件界面。此文件为 XS128 工程目录下的 cmd 文件夹下的“Abatron\_BDI\_Erase\_unsecure\_hcs12xe.cmd”文件。

单击“确定”后选择 TBDML HCS12→Unsecure 菜单项,则弹出如图 9-15 所示的设置 FCLKDIV 值界面。根据页面提示,输入数值为 17(使用 16 MHz 晶振)。



图 9-14 设置解密命令文件界面



图 9-15 设置 FCLKDIV 值界面



单击 OK 后开始解密,解密成功时出现如图 9-16 所示提示。此时 MCU 解除密码成功。



图 9-16 解密成功

# 第 10 章

## CAN 总线

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需求的汽车电子和工业控制领域。本章概括了 CAN 总线的通用知识,并给出 S12XS128 的 MSCAN 模块的编程要点及编程实例。

### 10.1 CAN 总线通用知识

#### 10.1.1 CAN 总线协议的历史概况

控制器局域网(Controllor 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他节点交换数据;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。

#### 10.1.2 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  $\text{CAN}_{\text{TX}}$  和接收  $\text{CAN}_{\text{RX}}$  两个引脚。 $\text{CAN}_{\text{TX}}$  发送串行数据到 CAN 总线收发器,同时  $\text{CAN}_{\text{RX}}$  从 CAN 总线收发器接收串行数据。常用的 CAN 收发器有 NXP 公司的 PCA82C250、TI 公司的 SN65HVD230 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 10-1 所示,把所有的  $\text{CAN}_{\text{TX}}$  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), $\text{CAN}_{\text{RX}}$  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。

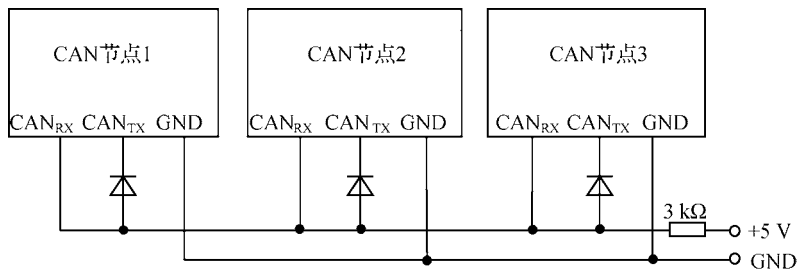


图 10-1 无需 CAN 收发器芯片的电路连接

## 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 10-2 所示。

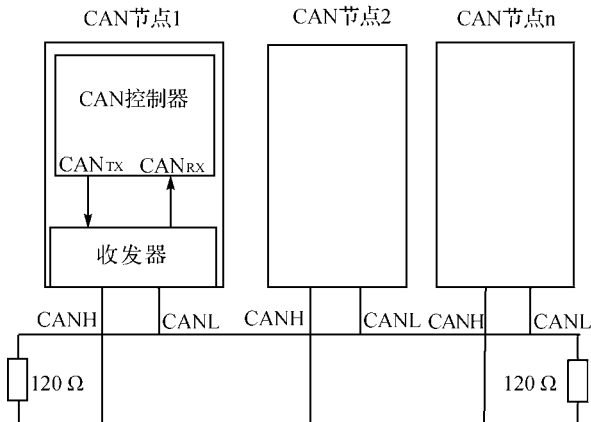


图 10-2 常用的 CAN 硬件系统组成

**注意:** CAN 通信节点上一般需要添加  $120\ \Omega$  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。

### 3. 带隔离的典型 CAN 硬件系统电路

NXP 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<sub>TX</sub>、CAN<sub>RX</sub> 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 10-3 所示。

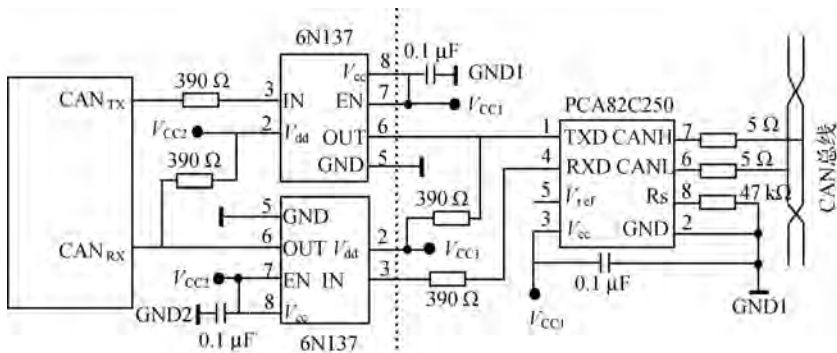


图 10-3 带隔离的典型 CAN 硬件系统电路

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  $V_{CC1}$  和  $V_{CC2}$  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 5 V 输出的开关电源模块实现。

② PCA82C250 的 CANH 和 CANL 引脚通过一个  $5\ \Omega$  的限流电阻与 CAN 总线相连, 保护 PCA82C250 免受过流的冲击。PCA82C250 的电源引脚旁应有一个  $0.1\ \mu\text{F}$  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<sup>注</sup>/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  $16\ \text{k}\Omega \sim 100\ \text{k}\Omega$  之间, 图 10-3 中选用  $47\ \text{k}\Omega$ 。关于电路相连的更多细节请参见 6N137 手册以及 PCA82C250 手册。

#### 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 10-4 所示。

注: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  $R_s$  的电流  $I_{R_s}$  成比例关系(或称斜率关系),而电流  $I_{R_s}$  的大小主要由  $R_s$  阻值决定,因此  $R_s$  的阻值变化将引起转换速度的变化。在斜率控制模式下  $R_s$  阻值一般取在  $16.5\text{ k}\Omega\sim 100\text{ k}\Omega$  之间。高速模式下,  $R_s$  阻值在  $0\sim 1.8\text{ k}\Omega$  之间。

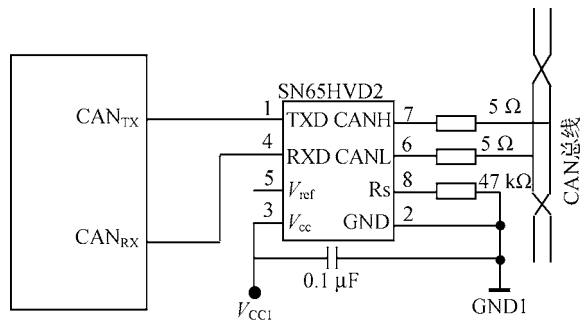


图 10-4 不带隔离的典型 CAN 硬件系统电路

### 10.1.3 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、规则做一般了解，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，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。设两条信号线分别称为 CAN\_H 和 CAN\_L，如

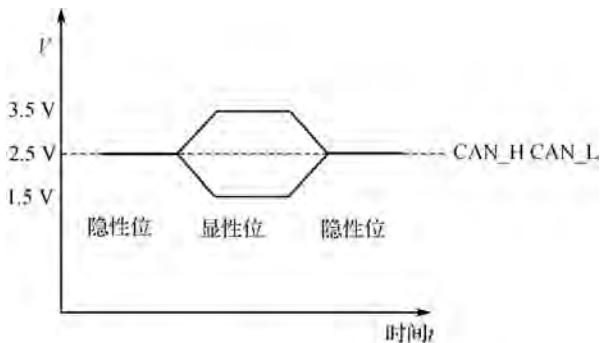


图 10-5 总线数据表示



图 10-5 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 2.5 V 左右,电压差( $V_{\text{diff}}=V_{\text{CAN\_H}}-V_{\text{CAN\_L}}$ )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=3.5 V,CAN\_L=1.5 V,电压差( $V_{\text{diff}}=V_{\text{CAN\_H}}-V_{\text{CAN\_L}}$ )在 2 V 左右。在总线空闲或隐性位期间,发送隐性位。

## 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 10.1.4 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 10-1 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 10-1 CAN 总线上任意两节点最大距离及位速率对应表

位速率/kbps	1 000	500	250	125	100	50	20	10	5
最大距离/m	40	130	270	530	620	1 300	3 300	6 700	10 000

**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 NXP 的 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配一个唯一的 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 决定是否接收该报文。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,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”<sup>注</sup>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 10-1),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)。需要远程数据

注:“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。

请求时,可通过发送远程帧实现,有关帧内容见 10.1.4 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。

## 10.1.4 帧结构

CAN 总线协议中有 4 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<sup>注</sup>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 10-6

注:这里用“场”,有的中文翻译用“域”,也有使用“字段”,均对应英文 Field。

所示。

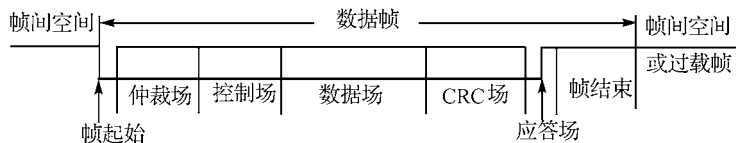


图 10-6 数据帧组成

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,其他节点检测到该信号,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(如图 10-7 所示)。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?)。

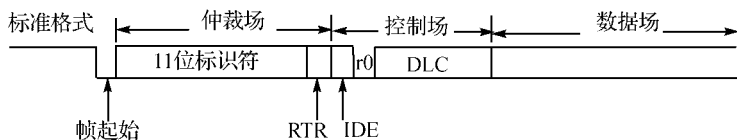


图 10-7 数据帧标准格式中的仲裁场结构

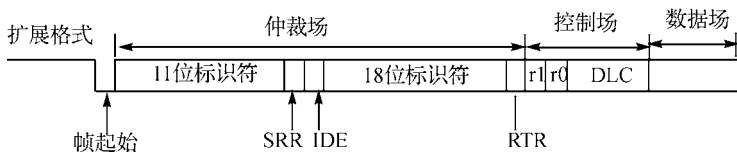


图 10-8 数据帧扩展格式中的仲裁场结构

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 10-8 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 10-9 所示。

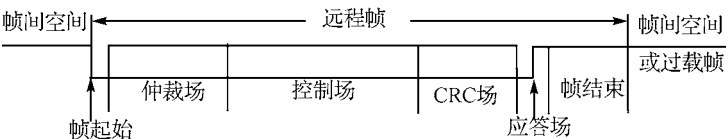


图 10-9 远程帧的组成

为方便描述,下面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 节点请求一个温度数据。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、通信效率方面的性能,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 5 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 10-10 所示。

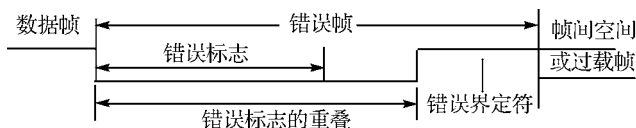


图 10-10 错误帧组成

错误标志有两种形式:主动错误(Error Active)<sup>注1</sup>标志和被动错误(Error Passive)<sup>注2</sup>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”。

注:① Error Active;也称为“错误激活”。

② Error Passive;也称为“错误认可”。



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 10-11 所示。



图 10-11 过载帧的组成

有 3 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—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。

## 10.1.5 位时间

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(PROG\_SEG)、相位段 1(PHASE\_SEG1)、相位段 2(PHASE\_SEG2)。如图 10-12 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。

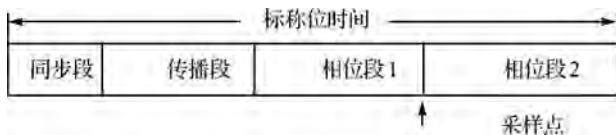


图 10-12 标称位时间组成示意图

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,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  $T_q$ ):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  $T_q$  的关系是:

$$\text{位时间} = m \times T_q$$

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  $T_q$ ,传播段可设置成 1、2、3...8 个  $T_q$ ,相位段 1 可设置成 1、2、3...8 个  $T_q$ ,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  $T_q$ 。一个位时间总的  $T_q$  值可以设置在 8~25 之间。

在确定一个 CAN 总线的通信速率时,主要根据上述参数确定。

## 10.2 MSCAN 模块简介

飞思卡尔可升级控制器局域网(MSCAN)是一种通信控制器局域网,它按照 1991 年 9 月定义的 Bosch 规范,执行 CAN2.0A/B 协议。

CAN 协议不仅用于汽车的数据总线,还可应用于实时处理、车辆在电磁干扰(Electro



magnetic Interference, EMI) 环境中。MSCAN 使用先进的缓冲器, 实现了可预测的实时性, 同时简化了程序设计。图 10-13 是 MSCAN 模块框图。

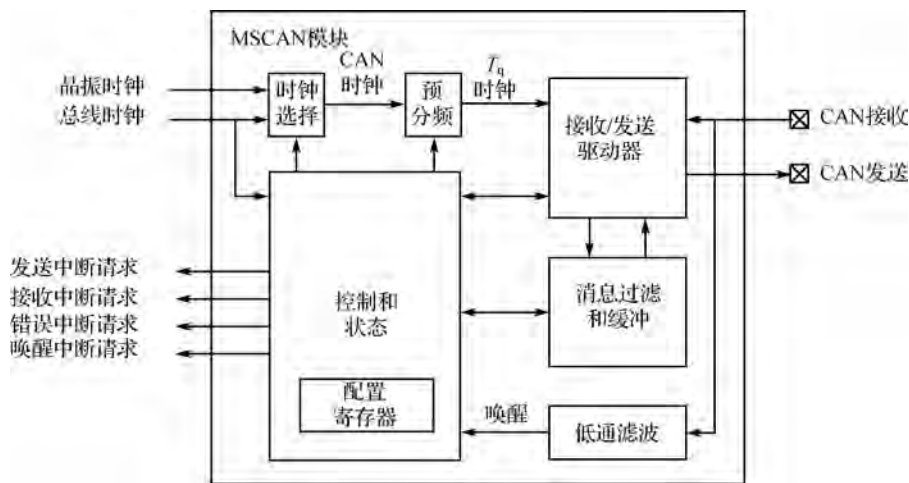


图 10-13 MSCAN 模块框图

### 10.2.1 MSCAN 特性

MSCAN 的基本特性如下：

- 完全支持 CAN 协议 2.0A/B 版: 标准或扩展数据帧、0~8 字节数据长度、高达 1 Mbps 的可编程比特率、支持远程帧；
- 5 个具有 FIFO 存储机制的接收缓冲器；
- 3 个具有“本地优先级”的发送缓冲器；
- 灵活的掩码标识符滤波器, 可配置为 2 个 32 位或 4 个 16 位或 8 个 8 位过滤掩码；
- 集成的可编程唤醒功能的低通滤波器；
- 支持自测操作的可编程闭环模式；
- 可编程仅作为 CAN 总线监听模式；
- 可编程总线脱离恢复功能；
- 独立的信号中断功能适用于所有 CAN 接收机和发送机错误状态(警报、被动错误、掉线)；
- 可编程 MSCAN 时钟源, 选择使用总线时钟或振荡器时钟；
- 内部计时器为接收和发送的报文提供时间戳；
- 3 种低功耗模式: 休眠模式、断电模式和 MSCAN 使能模式；
- 全局初始化配置寄存器。

## 10.2.2 报文存储结构、标识符验收过滤与时钟系统

### 1. 报文存储结构

MSCAN 的报文缓冲区组织结构如图 10-14 所示。MSCAN 模块使用了 5 个接收缓冲区和 3 个发送缓冲区。

#### (1) 报文发送基础

CAN 通信的建立基于两个基本前提：

**前提一：**任何 CAN 节点都能够发送出经过排序的报文流，而不需要在两条报文间释放 CAN 总线。这些节点在发送上一条报文后立即仲裁 CAN 总线，只有当仲裁丢失时才释放 CAN 总线。

**前提二：**若有多条报文准备发送，则需要安排 CAN 节点内部的报文发送队列，拥有较高优先级报文优先发出。这样的操作无法用单个发送缓冲器来实现，因为该缓冲器在上一条报文发送后必须立即重新加载，而加载流程的持续时间有限，必须在帧间顺序(IFS)内完成才能够发送不中断报文流。这对于有限总线速度的 CAN 来说是可行的，但它要求 CPU 有非常短的发送间歇时间。采用双缓冲器机制能够把发送缓冲器的正在加载和实际发送的报文分开，从而降低了 CPU 的响应要求。发送报文时 CPU 正重新加载另一个缓冲器，若此时没有缓冲器做好发送准备，CAN 总线会被释放。

无论在什么情况下，至少需要 3 个发送缓冲器来满足上述第一个要求。MSCAN 就有 3 个发送缓冲器。第二个要求需要内部优先级排序，MSCAN 以“发送结构”中描述“本地优先级”段为依据来执行发送优先级排队。

#### (2) 发送结构

MSCAN 的 3 缓冲发送机制允许提前建立多条报文，从而优化了实时性能。这 3 个缓冲器的安排如图 10-14 所示。这 3 个发送缓冲器都具有与接收缓冲器基本相同的 13 字节数据结构，还包含一个字节本地优先级(PRIO)，最后的两个字节用于报文的时间标签。

若要发送报文，首先，CPU 要找到可用的发送缓冲器，这需要查看发送器缓冲器空标志位(TXEx)来确定。若发送缓冲器可用，CPU 将写空闲缓冲器信息到 CANTBSEL 寄存器，为该缓冲器设置一个指针，使不同的缓冲器通过地址重定向通过 CANTXFG 地址空间访问。与 CANTBSEL 寄存器有关的操作简化了发送缓冲器选择。此外，这种机制使编程处理更为简单，发送流程只须访问一个地址，节省了地址空间。CPU 将标识符、控制位和数据内容等帧信息保存到的空闲发送缓冲器后，通过清 0 相应的 TXE 标志位，通知 CAN 模块发送准备就绪。

然后，MSCAN 安排报文发送，当发送完成后自动将相应的 TXE 标志位置 1。若设置了 TXEx，可触发发送中断，使中断处理程序重新加载缓冲器。当获得 CAN 总线仲裁时，若此时

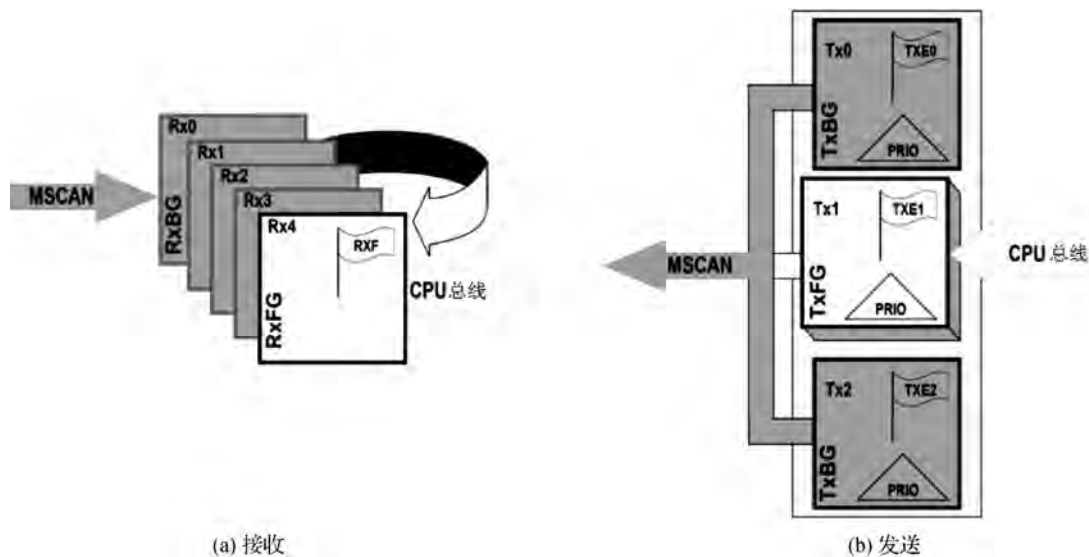


图 10-14 MSCAN 报文缓冲区组织图

有多个缓冲器等待发送, MSCAN 则使用 3 个缓冲器的本地优先级的设置来确定优先顺序。每个发送缓冲帧都有一个字节本地优先级字段(PRIO)。当报文建立时,处理程序会配置该字段。本地优先级反应了从该节点发送的有关报文之间的优先级顺序。PRIO 字段中具有较小二进制值的缓冲帧占较高优先级。每当 MSCAN 为 CAN 总线进行仲裁或出现发送错误时,都会引发内部调度程序。

当处理程序安排了高优先级报文时,可能会中止 3 个发送缓冲器的某一个低优先级报文。由于正发送的报文不能中止,因此用户必须通过设置相应的中止请求位(ABTRQ)请求中止。MSCAN 将通过以下方式处理该请求:

- 在 CANTAAK 寄存器中设置相应的中止确认标志(ABTAK)。
- 设置相关的 TXE 标志来释放缓冲器。
- 产生发送中断。发送中断处理程序能够根据 ABTAK 标志的配置确定是报文中止(ABTAK = 1)还是已发送(ABTAK = 0)。

### (3) 接收结构

收到的报文被保存在 5 级输入 FIFO 中。5 个报文缓冲器被交替映射到单个存储器区域,如图 10-14 所示。后台接收缓冲器(RxBG)只与 MSCAN 联系,前台接收缓冲器可以通过 CPU 寻址。这种机制简化了处理程序,接收程序段只须访问一个地址。在进行接收时,所有接收缓冲器都由 15 个字节的空间来保存 CAN 控制位、标识符(标准或扩展)、数据等内容。接收器已满标志(RXF)显示前台接收缓冲器的状态。当缓冲器包含带有匹配标识符的正确



接收报文时,置 1 该标志。

接收时,每条报文将被检查是否允许通过过滤器,然后被写入到有效 RxBG。成功接收到有效报文后,MSCAN 将 RxBG 的内容转移到接收器 FIFO,将 RXF 标志置 1,并向 CPU 发出一个接收中断。用户的接收处理程序将从 RxFG 读取收到的报文,然后复位 RXF 标志,确认中断、释放前台缓冲器。一般情况下,紧跟 CAN 帧的 IFS 字段后的新报文将被接收到下一个可用 RxBG 中。若 MSCAN 在 RxBG 中接收到无效报文(错误标识符、发送错误等),缓冲器的实际内容将被下一条报文覆盖,之前的无效内容不会转移到 FIFO。

当 MSCAN 模块正在发送报文时,MSCAN 把自己发送的报文接收到后台接收缓冲器 RxBG,但不会将此帧转移到接收器 FIFO,生成接收中断或在 CAN 总线上响应其自己的报文。这一规则以外的报文存在于闭环模式中,此时 MSCAN 会完全按照其他所有报文一样的方式处理报文。当仲裁丢失时,MSCAN 也会接收自己发送的报文,此时 MSCAN 必须做好成为接收器的准备。

当 FIFO 中的所有接收报文缓冲器装满了带正确标记的报文,并从 CAN 总线中正确接收到另外一条报文时,就可能会出现溢出。最后接收到的一条报文将被丢弃,并生成带有溢出标志的错误中断。若接收器 FIFO 已满,MSCAN 仍能发送报文,那么所有接收到的报文都会被丢弃,直到 FIFO 中的接收缓冲器再次可用,才可以接收新的有效报文。

## 2. 标识符验收过滤

MSCAN 标识符验收寄存器(CANIDAC)用于标准帧标识符(ID10~ID0)或扩展帧标识符(ID28~ID0)的接收。当总线上有报文到达时,MSCAN 会将该报文的标识符与标识符验收寄存器中的内容进行比较,对应位值相同的位直接验收通过,若值不同,则与标识符掩码寄存器(CANIDMR0~7)的对应位定义值有关,定义值为 1 则报文验收通过,定义值为 0 则不通过。

当标识符验收通过,MSCAN 置接收缓冲区满标志位(RXF = 1),并在标识符验收控制寄存器中(CANIDAC)利用命中标志位(IDHIT2~0)来指示是使用哪个标识符验收器进行验收。这种方式简化了编程工作量,不须依靠编程识别接收器的中断源。当多个命中产生,较低位的命中享有较高优先权。

在接收报文时,需要对哪些位进行验收比较,与当前的过滤器工作方式有关。MSCAN 有 4 种过滤器工作方式,下面进行简要介绍。

### (1) 双标识符验收过滤器工作方式

两个标识符验收过滤器,每个验收过滤器 32 位。每个过滤器被用于:

- 扩展帧:29 位标识符、RTR、IDE、SRR;
- 标准帧:11 位标识符、RTR、IDE。

如图 10-15 所示,第一个 32 位的过滤器段(CANIDAR0~3,CANIDMR0~3)对应过滤

器 0 命中,而第二个 32 位的过滤器段(CANIDAR4~7,CANIDMR4~7)对应过滤器 1 命中。

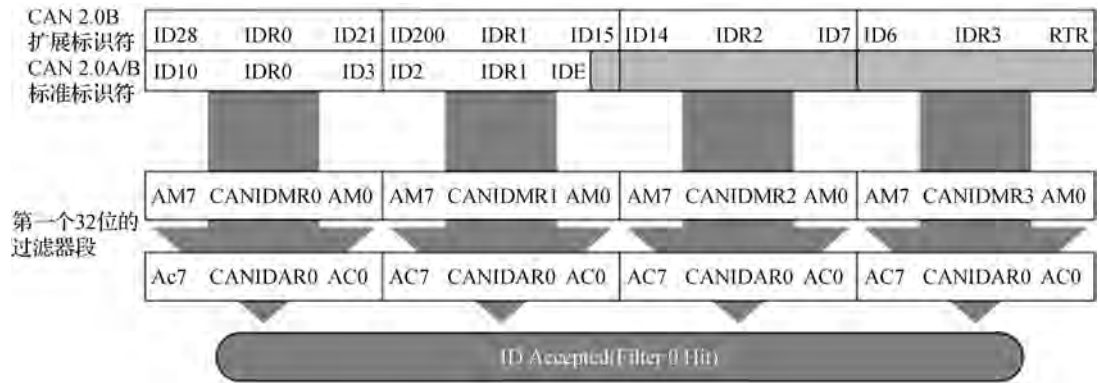


图 10-15 32 位可屏蔽标识符验收过滤器

### (2) 4 个标识符验收过滤器工作方式

4 个标识符验收过滤器,每个标识符过滤器 16 位。每个过滤器被用于:扩展帧:标识符的高 10 位、SRR、IDE;标准帧:11 位标识符、RTR、IDE。

图 10-16 显示第一个 32 位的过滤器段(CANIDAR0~3,CANIDMR0~3)对应过滤器 0 和过滤器 1 命中,而第二个 32 位的过滤器段(CANIDAR4~7,CANIDMR4~7)对应过滤器 2 和过滤器 3 命中。

### (3) 8 个标识符验收过滤器工作方式

8 个标识符验收过滤器,每个标识符过滤器 8 位。每个滤波器被用于:扩展帧:标识符的高 8 位;标准帧:标识符的高 8 位。

该方式采用 8 个独立的滤波器,可对标准帧或扩展帧的高 8 位标识符进行滤波比较,如图 10-17 所示。第一个 32 位的过滤器段(CANIDAR0~3,CANIDMR0~3)产生过滤器 0~3 命中,而第二个 32 位的过滤器段(CANIDAR4~7,CANIDMR4~7)产生过滤器 4~7 命中。

### (4) 关闭过滤器工作方式

在这种模式下,报文不会被放入到接收前台缓冲区 RxFG,RXF 接收标志也不会被置位。

## 3. 时钟系统

图 10-18 显示 MSCAN 时钟发生电路的结构。

CANCTL1 寄存器中的时钟源位(CLKSRC)决定内部 CANCLK 连接到晶体振荡器(振荡器时钟)输出还是连接到总线时钟。必须选择能满足 CAN 协议的振荡器精度要求(高达 0.4%)的时钟源。此外,对于高 CAN 总线速率(1 Mbps)来说,要保证 45%~55% 的时钟占空比。

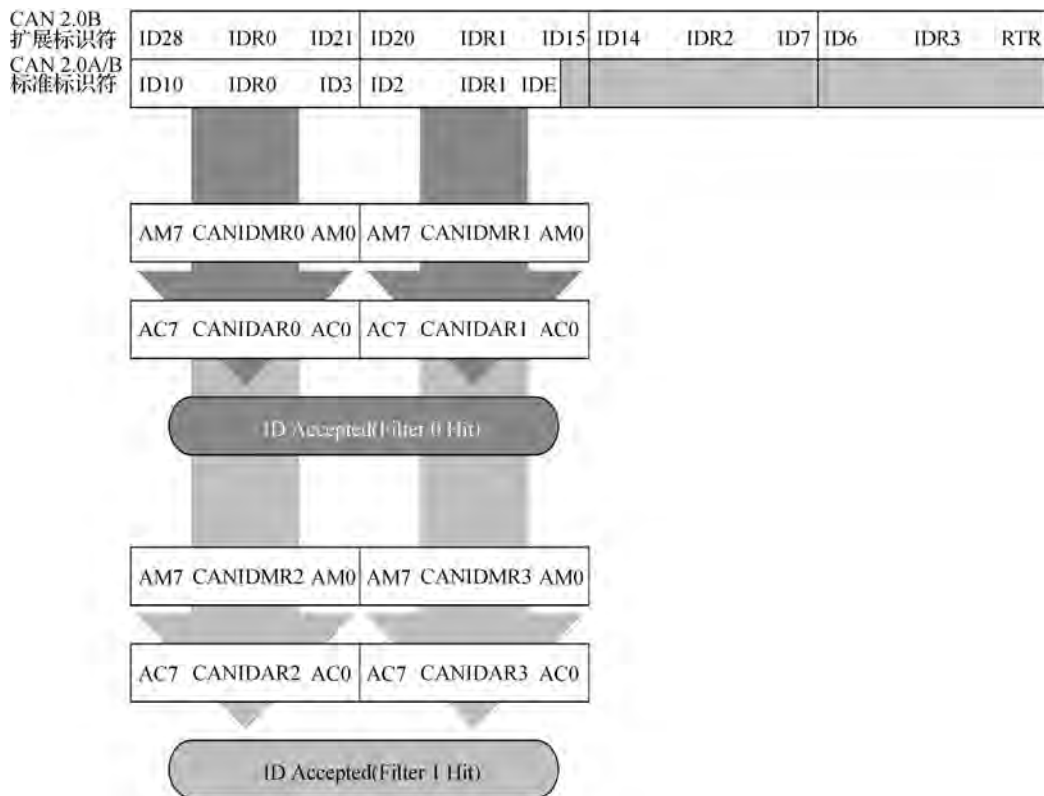


图 10-16 32 位可屏蔽标识符验收过滤器

如果总线时钟从 PLL 中生成,由于可能存在抖动,建议选择振荡器时钟而不要选择总线时钟,特别是以较快的 CAN 总线速率工作时。PLL 锁可能太宽,不能确保所需的时钟精度。

对于那些没有时钟和复位发生器 (CRG) 的微控制器, CANCLK 的驱动则来自于晶体振荡器 (振荡时钟)。

可编程预分频器从 CANCLK 生成时间冲量 ( $T_q$ ) 时钟。时间冲量是 MSCAN 所处理时间的基本单位。

$$f_{Tq} = f_{CANCLK} / (\text{预分频器值}) \quad (\text{式 } 10-1)$$

位时间再分成 3 段,如 Bosch CAN 规范所述,如图 10-19 所示。

SYNC\_SEG: 该段有一个长度固定的时间冲量,信号边沿预计出现在本段。

时段 1: 本段包括 CAN 标准的 PROP\_SEG 和 PHASE\_SEG1。可以通过编程设置参数 TSEG1,使之包含 4~16 个时间冲量。

时段 2: 本段表示 CAN 标准的 PHASE\_SEG2。可以通过编程设置 TSEG2 参数,使之具有 2~8 个时间冲量长。



CAN 2.0B  
扩展标识符  
CAN 2.0A/B  
标准标识符

ID28	IDR0	ID21	ID20	IDR1	ID15	ID14	IDR2	ID7	ID6	IDR3	RTR
ID10	IDR0	ID3	ID2	IDR1	ID0						

AM7 CIDMR0 AM0

AC7 CIDAR0 AC0

ID Accepted(Filter 0 Hit)

AM7 CIDMR1 AM0

AC7 CIDAR1 AC0

ID Accepted(Filter 1 Hit)

AM7 CIDMR2 AM0

AC7 CIDAR2 AC0

ID Accepted(Filter 2 Hit)

AM7 CIDMR3 AM0

AC7 CIDAR3 AC0

ID Accepted(Filter 3 Hit)

图 10-17 8 位可屏蔽标识符验收过滤器

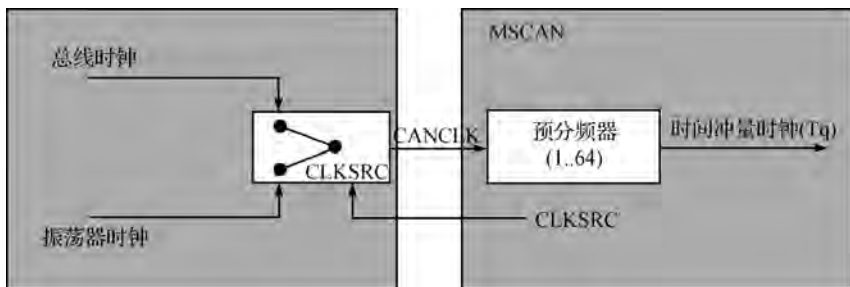


图 10-18 MSCAN 时钟发生结构图

$$\text{位速率} = f_{Tq} / (\text{时间份额}) \quad (\text{式 } 10-2)$$

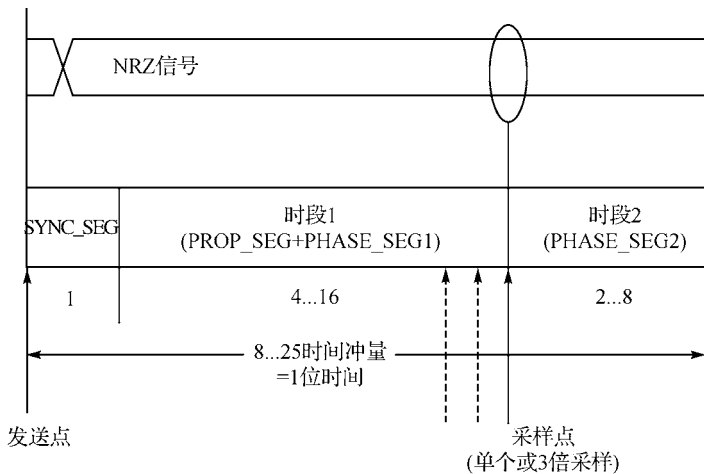


图 10-19 位时间内的段

表 10-2 时段句法

名 称	描 述
SYNC_SEG	系统希望该时段内在 CAN 总线上出现电平转换
发送点	正处于发送模式的节点在该点上向 CAN 总线传输一个新值
采样点	正处于接收模式的节点在该点采样 CAN 总线。如果选择了每位采样 3 次模式,那么该点标记第三采样点的位置

可以通过编程设置 SJW 参数,使得同步跳转宽度在 1~4 个时间冲量范围内。SYNC\_SEG、TSEG1、TSEG2 和 SJW 参数通过编程 MSCAN 总线时钟寄存器 (CANBTR0、CANBTR1) 进行设置。表 10-3 概括地描述了 CAN 段设置和相关参数值。



表 10-3 遵从 CAN 标准的位时段设置

时段 1	TSEG1	时段 2	TSEG2	同步跳转宽度	SJW
5...10	4...9	2	1	1...2	0...1
4...11	3...10	3	2	1...3	0...2
5...12	4...11	4	3	1...4	0...3
6...13	5...12	5	4	1...4	0...3
7...10	6...13	6	5	1...4	0...3
8...15	7...10	7	6	1...4	0...3
9...16	8...15	8	7	1...4	0...3

### 10.2.3 CAN 模块的主要运行模式、低功耗选项、中断与响应

#### 1. CAN 模块的主要运行模式

CAN 模块的主要运行模式有正常模式、侦听模式、初始化模式。

##### 1) 正常模式

在正常模式中,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。

##### 2) 侦听模式

在可选的 CAN 总线侦听模式中,CAN 节点能够接收有效数据帧和有效远程帧,但它只发送 CAN 总线上的“隐性”位。此外,它不能启动发送。若 MAC 层需要发送“显性”位(ACK 位、超载标志或有效错误标志),该位只能在内部传输,这样 MAC 层就监控“显性”位,此时 CAN 总线在外部仍保持隐性状态。

##### 3) MSCAN 初始化模式

在初始化模式中,正在进行的任何发送或接收行为都会立即中止,与 CAN 总线的同步丢失,并可能引起违反 CAN 协议。为了防止 CAN 总线系统出现严重的后果,MSCAN 立即驱动 TXCAN 引脚进入隐性状态。

**注意:**进入初始化模式时,用户要确定 MSCAN 不在工作态。其操作步骤是:在 CANCTL0 寄存器中设置 INITRQ 位前,把 MSCAN 置入休眠模式(SLPRQ = 1,SLPAK = 1)。否则,中止正在发送的报文可能导致错误,并影响到其他 CAN 总线节点。

在初始化模式中,MSCAN 被停止。但接口寄存器仍然可以访问。这种模式用来将 CANCTL0、CANRFLG、CANRIER、CANTFLG、CANTIER、CANTARQ、CANTAACK 和 CANTBSEL 寄存器复位为它们的默认值。此外,MSCAN 还使能 CANBTR0、CANBTR1 位时钟寄存器的配置以及 CANIDAC、CANIDAR 和 CANIDMR 报文过滤器。



由于 MSCAN 内的独立时钟源, INITRQ 必须通过采用特殊握手机制进行时钟同步。这种握手导致了进一步的同步延迟, 如图 10-20 所示。

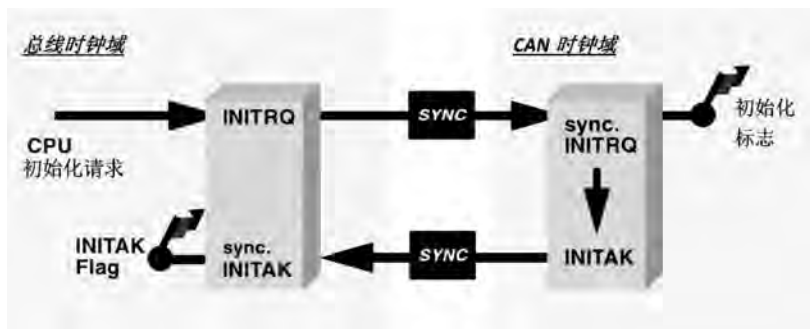


图 10-20 初始化请求/确认周期

若 CAN 总线上没有正在传输的报文, 此时, 最小延迟将是两个额外的总线时钟和 3 个额外的 CAN 时钟。当 MSCAN 的所有部件都处于初始化模式时, INITAK 标志置位。应用程序必须将 INITAK 作为握手标志, 以便请求 (INITRQ) 进入初始化模式。

**注意:**在使能初始化模式 (INITRQ = 1 和 INITAK = 1) 前, CPU 不能清除 INITRQ。

## 2. CAN 模块的低功耗选项

### (1) MSCAN 休眠模式

通过在 CANCTL0 寄存器中确定 SLPRQ 位, CPU 可以请求 MSCAN 进入低功耗模式。MSCAN 进入休眠模式的时间取决于固定的同步时延及其当前状态:

若有一个或多个报文缓冲器等待发送 (TXEx=0), 则 MSCAN 将继续发送, 直到所有发送报文缓冲器空 (TXEx=1, 成功发送或中止), 然后再进入休眠模式。

若 MSCAN 正在接收, 则继续接收, 并且一旦 CAN 总线空闲, 就立即进入休眠模式。

若 MSCAN 既不发送也不接收, 则会立即进入休眠模式。

**注意:**程序必须避免建立发送 (通过清除一个或多个 TXEx 标志) 后立即请求休眠模式 (通过设置 SLPRQ)。MSCAN 是启动发送还是直接进入休眠模式取决于实际操作顺序。

若激活休眠模式, 则需要将 SLPRQ 和 SLPK 置位, 如图 10-21 所示。程序必须把 SLPK 作为请求 (SLPRQ) 的握手标志, 以进入休眠模式。

当处于休眠模式 (SLPRQ=1, SLPK=1) 时, MSCAN 停止其内部时钟, 但 CPU 访问寄存器的时钟继续运行。

若 MSCAN 处于总线脱离状态, 由于时钟停止, 它将停止计数 11 个连续隐性位的 128 次出现。TXCAN 引脚保持隐性状态。若 RXF=1, 可以读取报文且可以清 0RXF。当处于休眠模式时, 不会出现新报文被转移到接收器 FIFO (RxFG) 的当前缓冲器的情况。访问发送缓冲

器和清 0 对应 TXE 标志是允许的。当处于休眠模式时,不会出现报文中止的情况。

若 CANCLT0 中的 WUPE 位尚未置位,MSCAN 将屏蔽它在 CAN 上检测到的任何信号,RXCAN 引脚在内部设置为隐性状态,MSCAN 将被锁在休眠模式,如图 10-21 所示。WUPE 必须在进入休眠模式前配置。

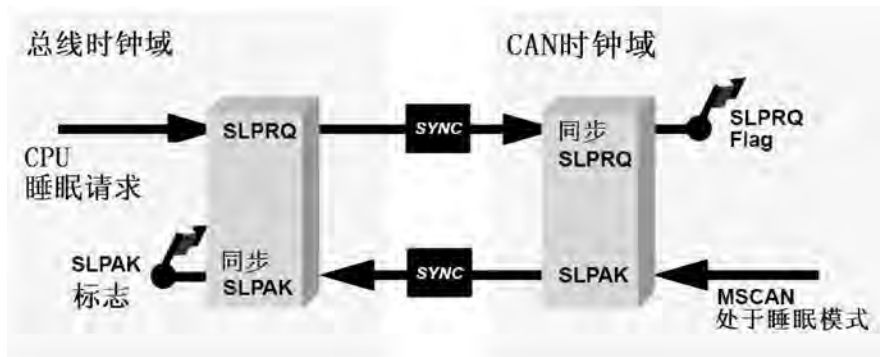


图 10-21 休眠请求/确认周期

只有当出现以下情形时,MSCAN 才能够退出休眠模式(唤醒):出现 CAN 总线有效和 WUPE=1;SLPRQ 位清 0。

**注意:**在使能休眠模式(SLPRQ=1,SLPAK=1)前,CPU 不能清 0 SLPRQ 位。唤醒之后,MSCAN 等待 11 个连续隐性位与 CAN 总线同步。因此,如果 SCAN 被 CAN 帧唤醒,就不会收到该帧。

若在进入休眠模式前已经收到报文,接收报文缓冲器(RxFG 和 RxBG)将存储该报文。所有挂起操作在唤醒后执行,复制 RxBG 至 RxFG,报文中止和报文发送。若在退出休眠模式后 MSCAN 仍处于总线脱离状态,它将继续计数 128 次 11 个连续隐性位的出现。

## (2) MSCAN 断电模式

当出现以下情况时,MSCAN 处于断电模式:CPU 处于停止模式;CPU 处于等待模式且设置了 CSWAI 位。

当进入断电模式时,MSCAN 立即停止正在进行的所有发送和接收行为,这样可能违反 CAN 协议。为了防止 CAN 总线系统出现违反上述规则而产生严重后果,MSCAN 立即驱动 TXCAN 引脚进入隐性状态。

**注意:**进入初始化模式时,用户要确保 MSCAN 不在工作态。其操作步骤是:在 CANCTL0 寄存器中设置 INITRQ 位前,MSCAN 置入休眠模式(SLPRQ=1,SLPAK=1)。否则,中止正在发送的报文可能导致错误情况,并影响到其他 CAN 总线节点。

在断电模式中,所有时钟停止,且不能访问寄存器。如果在断电模式有效前 MSCAN 未处于休眠模式,通电后该模块执行一个内部恢复周期。这会给模块再次进入正常模式一个固



定延迟。

### (3) 禁止模式

在  $CANEN=0$  时,复位后 MSCAN 进入禁止模式。为了减少功耗所有模块的时钟都停止,但是映像寄存器仍然可以访问。

### (4) 可编程唤醒功能

只要检测到 CAN 总线有效,就可以对 MSCAN 进行编程以唤醒 MSCAN。当处于休眠模式时,通过将低通滤波器功能应用于 RXCAN 输入,可以更改 CAN 总线检测的灵敏度。

该功能可以用来防止由于 CAN 总线线路上的短脉冲而唤醒 MSCAN。例如,嘈杂环境中的电磁干扰可以引起尖峰脉冲。

## 3. 中 断

本小节给出由 MSCAN 引发的所有中断,列出了使能位和触发标志。MSCAN 支持 4 个中断矢量(如表 10-4 所列),任意一个矢量都可以单独屏蔽。

表 10-4 中断矢量

中断源	CCR 掩码	本地使能
唤醒中断(WUPIF)	1 位	CANRIER (WUPIE)
错误中断(CSCIF, OVRIF)	1 位	CANRIER(CSCIE, OVRIE)
接收中断(RXF)	1 位	CANRIER(RXFIE)
发送中断(TXE[2:0])	1 位	CANRIER (TXEIE[2:0])

### 1) 发送中断

3 个发送缓冲器中至少有一个可用,并可以写入报文发送。空报文缓冲器的 TXEx 标志已置位。

### 2) 接收中断

报文成功接收,并转移到接收器 FIFO 的前端缓冲器(RxFG)。收到 EOF 信号后,立即生成该中断,RXF 标志被置 1。若接收器 FIFO 中有多条报文,一旦下一条报文转移到前端缓冲器,就立即设置 RXF 标志。

### 3) 唤醒中断

若在 MSCAN 内部休眠模式期间 CAN 总线上有信号,就生成唤醒中断。WUPE 必须使能。

### 4) 错误中断

若出现了接收器 FIFO 溢出、错误、警报或总线脱离情况,就出现错误中断。显示为以下情况中的一种:

溢出——出现接收器 FIFO 的溢出情况。

CAN 状态变化——MSCAN 的 CAN 总线状态反应实际情况。只要错误计数器进入考虑范围(Tx/Rx 警报、Tx/Rx 错误、总线脱离),MSCAN 就标识错误情况。产生错误情况的状态变化用 TSTAT 和 RSTAT 标志表示。

5) 中断响应

中断与 MSCAN 接收器标志寄存器或 MSCAN 发送器标志寄存器中的一个或多个状态标志直接相关。CANRFLG 和 CANTFLG 中的标志必须在中断处理程序中复位。写 1 清 0 对应标志。若中断条件仍然存在,标志不能被清除。只要设置了相应标志中的一个,中断就产生。

**注意:**必须确保 CPU 只清除引起当前中断的位。正是因为这个原因,不能用位操作指令(BSET)清除中断标志。这种指令可能造成意外清除进入当前中断服务程序后设置的中断标志。

10.3 MSCAN 模块的内存映射及寄存器定义

10.3.1 MSCAN 模块内存映射

图 10-22 和表 10-5 介绍了 MSCAN 所有寄存器及地址。寄存器的地址由基址和偏移量组成,基址在 MCU 中定义,而偏移量在模块内部定义。

MSCAN 占用 64 字节的内存空间。当 MCU 选定时,MSCAN 的基址就确定了。

地址偏移量

\$_00	控制寄存器 12个字节
\$_0B	
\$_0C	保留 2个字节
\$_0D	
\$_0E	错误计数器 2个字节
\$_0F	
\$_10	标识符过滤器 16个字节
\$_1F	
\$_20	接收缓冲区 16个字节(窗口机制)
\$_2F	
\$_30	发送缓冲区 16个字节(窗口机制)
\$_3F	

图 10-22 MSCAN 寄存器组织图



表 10-5 各功能模块寄存器地址分配表

地 址	功能模块	访问权限
\$ _00、\$ _01	MSCAN 控制寄存器 0、1(CANCTL0、CANCTL1)	读/写 1
\$ _02、\$ _03	MSCAN 总线时钟寄存器 0、1(CANBTR0、CANBTR1)	读/写
\$ _04	MSCAN 接收标志寄存器(CANRFLG)	读/写 1
\$ _05	MSCAN 接收中断使能寄存器(CANRIER)	读/写
\$ _06	MSCAN 发送标志寄存器(CANTFLG)	读/写 1
\$ _07	MSCAN 发送中断使能寄存器(CANTIER)	读/写 1
\$ _08	MSCAN 发送消息忽略控制请求(CANTARQ)	读/写 1
\$ _09	MSCAN 发送消息忽略控制应答(CANTAACK)	读
\$ _0A	MSCAN 发送缓冲器选择(CANTBSEL)	读/写 1
\$ _0B	MSCAN 标识符验收控制寄存器(CANIDAC)	读/写 1
\$ _0C、\$ _0D	保留	
\$ _0E	MSCAN 接收错误计数寄存器(CANRXERR)	读
\$ _0F	MSCAN 发送错误计数寄存器(CANTXERR)	读
\$ _10~\$ _13	MSCAN 标识符验收码寄存器 0~3(CANIDAR0~3)	读/写
\$ _14~\$ _17	MSCAN 标识屏蔽寄存器 0~3(CANIDMR0~3)	读/写
\$ _18~\$ _1B	MSCAN 标识符验收码寄存器 4~7(CANIDAR4~7)	读/写
\$ _1C~\$ _1F	MSCAN 标识屏蔽寄存器 4~7(CANIDMR4~7)	读/写
\$ _20~\$ _2F	接收前台缓冲区(CANRXFG)	读 2
\$ _30~\$ _3F	发送前台缓冲区(CANTXFG)	读 2/写

注意:①写访问限制参照寄存器的详细描述;②CANRXFG 和 CANTXFG 的保留位和不使用的位读时为“x”。

### 10.3.2 MSCAN 模块寄存器

本节详细描述了 MSCAN 模块的所有寄存器和寄存器位。每个寄存器都配有一个标准的寄存器位图,寄存器位的功能介绍按位的顺序安排在位图的后面。

#### 1. 控制寄存器

这里详细描述 MSCAN 模块中的部分寄存器和寄存器位。每个描述都包括带有相关图形编号的标准寄存器示意图。寄存器位和字段功能的详细说明在寄存器图后面,按位顺序。该模块中所有寄存器的所有位在寄存器读取过程中都与内部时钟完全同步。

##### (1) MSCAN 控制寄存器 0(CANCTL0)

CANCTL0 寄存器提供了如下所述的 MSCAN 模块的各种位控制:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ
复位	0	0	0	0	0	0	0	1

D7—RXFRM 位,接收帧标志位。该位是只读和只清除位。当接收器正确收到有效报文(独立于滤波器配置)时,设置该位。设置后,该位一直保持不变,直到通过软件或复位将其清除。通过写入 1 清除该位。写 0 将被忽略。该位在闭环模式中无效。RXFRM=0,自上次清除该标志以来未收到有效报文;RXFRM=1,自上次清除该标志以来收到有效报文。

D6—RXACT 位,接收器活跃状态位。该只读标志表示 MSCAN 正在接收报文。该标志由接收器前端控制。该位在闭环模式中无效。RXACT=0,MSCAN 正在发送或空闲;RXACT=1,MSCAN 正在接收报文(包括仲裁丢失时)。

D5—CSWAI 位,在等待模式中 CAN 停止位。设置此位,可以在等待模式中通过禁止 MSCAN 模块与 CPU 总线接口的所有时钟来降低功耗。CSWAI=0,在等待模式中 CAN 模块不受影响;CSWAI=1,在等待模式中,CAN 模块停止时钟。

D4—SYNCH 位,同步状态位。该只读标志显示 MSCAN 是否与 CAN 总线同步,是否能够参与通信流程。MSCAN 设置和清除该位。SYNCH=0,MSCAN 与 CAN 总线不同步。SYNCH=1,MSCAN 与 CAN 总线同步。

D3—TIME 位,计时器使能位。该位激活一个内置 16 位自由运行计时器。若是能定时器,则在每个发送/接收缓冲区内的消息上打上 16 位时间戳。一旦消息被 CAN 总线接收,时间戳就将写在相应缓冲器的最高字节(0x000E, 0x000F)。禁止时,内部计时器复位(所有位都设置为 0)。该位在初始化模式中保持低。TIME=0,禁止内部 MSCAN 计时器。TIME=1,使能内部 MSCAN 计时器。

D2—WUPE 位,唤醒使能位。该位允许 MSCAN 模块检测到 CAN 总线通信时从休眠模式中重启。为了让所选功能发挥作用,该位在进入休眠模式前必须进行配置。WUPE=0,唤醒禁止,MSCAN 忽略 CAN 总线通信;WUPE=1,唤醒使能,MSCAN 能够重启。

D1—SLPRQ 位,休眠模式请求位。该位要求 MSCAN 进入休眠模式,这是一个内部节电模式。当 CAN 总线空闲时,也就是说该模块不接收任何报文且所有发送缓冲器为空,休眠模式请求被受理。通过设置 SLPK=1,表示该模块进入休眠模式。当设置了 WUPE 标志时,不能设置 SLPRQ。休眠模式维持有效,直到 SLPRQ 被 CPU 清除或者根据 WUPE 的设置,MSCAN 检测到 CAN 总线通信并自行清除。SLPRQ=0,运行中,MSCAN 正常工作;SLPRQ=1,休眠模式请求,当 CAN 总线空闲时,MSCAN 进入休眠模式。

D0—INITRQ 位,初始化模式请求位。当 CPU 设置该位时,MSCAN 切换至初始化模式。任何正在进行的发送或接收都将被中止,与 CAN 总线的同步也丢失。通过设置 INITAK=1,表示该模块进入初始化模式。以下寄存器进入其硬复位状态并恢复它们的默认值:





CANCTL08、CANRFLG9、CANRIER10、CANTFLG、CANTIER、CANTARQ、CANTAACK 和 CANTBSEL。MSCAN 处于初始化模式 (INTRQ = 1, INITAK = 1) 时, 寄存器 CANCTL1、CANBTR0、CANBTR1、CANIDAC、CANIDAR07 和 CANIDMR0—7 只能通过 CPU 写入。错误计数器的值不受初始化模式的影响。当该位通过 CPU 清除时, MSCAN 重启, 然后试图与 CAN 总线同步。如果 MSCAN 未处于总线脱离状态, 它在 CAN 总线上出现 11 个连续隐性位后同步。如果 MSCAN 处于总线脱离状态, 它将继续等待 11 个连续隐性位重复出现 128 次。只有当退出初始化模式后, 才可以在 CANCTL0、CANRFLG、CANRIER、CANTFLG 或 CANTIER 中写入其他位, 这时 INTRQ = 0, INITAK = 0。INTRQ = 0, 正常运行; INTRQ = 1, MSCAN 处于初始化模式。

## (2) MSCAN 控制寄存器 1(CANCTL1)

CANCTL1 寄存器如下文所述提供了 MSCAN 模块的各种控制位和握手状态报文:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	CANE	CLKSRC	LOOPB	LISTEN	BORM	WUPM	SLPAK	INITAK
复位	0	0	0	1	0	0	0	1

当 MSCAN 处于初始化模式 (INTRQ = 1, INITAK = 1) 的特殊系统运行模式时, 对该寄存器可以任意写入, 除 CANE 在正常情况下只可写入一次例外。

D7—CANE 位, MSCAN 使能位。当 CANE = 0, MSCAN 模块禁止; 当 CANE = 1, MSCAN 模块使能。

D6—CLKSRC 位, MSCAN 时钟源位。该位定义 MSCAN 模块的时钟源 (仅适用于具有时钟发生模块的系统)。当 CLKSRC = 0, MSCAN 时钟源是振荡器; 当 CLKSRC = 1, MSCAN 时钟源是总线时钟。

D5—LOOPB 位, 闭环自测模式位。当设置了该位时, MSCAN 执行可用于自测操作的内部闭环。发送器的输出位流从内部流回到接收器。RXCAN 输入被忽略, 且 TXCAN 输出进入隐性状态 (逻辑 1)。在发送时, MSCAN 表现的和正常运行时一样, 将自己发送的报文看作是接收远程节点发送的一样。在这种状态里, MSCAN 忽略 ACK 间隙发送的位确保正确接收它自己的报文。发送和接收中断都会发生。当 LOOPB = 0, 闭环自测禁止; 当 LOOPB = 1, 闭环自测使能。

D4—LISTEN 位, 监听模式位。该位把 MSCAN 配置为 CAN 总线监控器。当设置了 LISTEN 时, 将接收所有 ID 匹配的有效 CAN 报文, 但不发出确认或错误帧。此外, 错误计数器停止计数。监听模式可以支持需要“热插拔”或“吞吐量分析”的应用。当监听模式处于有效状态时, MSCAN 不能发送任何报文。LISTEN = 0, 正常运行; LISTEN = 1, 监听模式使能。

D3—BORM 位, 总线脱离恢复模式位。该位配置 MSCAN 的总线脱离恢复模式。当 BORM = 0, 自动总线脱离恢复 (参见 Bosch CAN2.0A/B 协议规范); 当 BORM = 1, 根据用户



请求总线脱离恢复。

D2—WUPM 位,唤醒模式位。如果 CANCTL0 中的 WUPE 被使能,该位定义了是否应用集成低通滤波器来防止 MSCAN 出现假唤醒。WUPM=0,MSCAN 被 CAN 总线上的任意显性信号唤醒;WUPM=1,MSCAN 只有在 CAN 总线上的显性脉冲长度为  $T_{wup}$  时才唤醒。

D1—SLPAK 位,只读位,休眠模式确认位。该标志指示 MSCAN 模块是否已经进入休眠模式。它用作 SLPRQ 休眠模式请求的握手标志。当 SLPRQ = 1、SLPAK = 1 时,休眠模式是有效的。根据 WUPE 设置,如果在处于休眠模式检测到 CAN 总线有信号,MSCAN 将清除该标志。CPU 清除 SLPRQ 位也将复位 SLPK 位。SLPAK=0,正在运行,MSCAN 正常运行;SLPAK=1,休眠模式使能,MSCAN 已经进入休眠模式。

D0—INITAK 位,只读位,初始化模式确认位。该标志显示 MSCAN 模块是否处于初始化模式。它用作 INITRQ 初始化模式请求的握手标志。当 INITRQ=1,INITAK=1 时,初始化模式被使能。当 MSCAN 处于初始化模式时,寄存器 CANCTL1、CANBTR0、CANBTR1、CANIDAC、CANIDAR0~CANIDAR7 和 CANIDMR0~CANIDMR7 只能通过 CPU 写入。INITAK=0,正在运行,MSCAN 正常运行;INITAK=1,初始化模式使能,MSCAN 处于初始化模式。

### (3) MSCAN 总线时钟寄存器 0(CANBTR0)

CANBTR0 寄存器配置 MSCAN 模块的各种 CAN 总线定时参数。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
复位	0	0	0	0	0	0	0	0

可在任意时间对该寄存器读取;在 INITRQ=1 和 INITAK=1 时的初始化模式下的任意时间对该寄存器写入。

D7~D6—SJW[1:0],同步跳转宽度位。同步跳转宽度定义了要实现 CAN 总线上的数据传输重新同步,一个位可以缩短或延长的时间冲量( $T_q$ )的最大值,参见表 10-6。

D5~D0—BRP[5:0],波特率预分频器位。该位确定用来构建位计时的时间冲量( $T_q$ )时钟,如表 10-7 所列。

表 10-7 波特率预分频器

BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	预分频器值(P)
0	0	0	0	0	0	1
0	0	0	0	0	1	2
0	0	0	0	1	0	3
0	0	0	0	1	1	4
:	:	:	:	:	:	:
1	1	1	1	1	1	64

表 10-6 同步跳转宽度

SJW1	SJW0	同步跳转宽度
00	1	$T_q$
01	2	$T_q$
10	3	$T_q$
11	4	$T_q$



#### (4) MSCAN 总线定时寄存器 1(CANBTR1)

CANBTR1 寄存器配置 MSCAN 模块的各种 CAN 总线定时参数。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
复位	0	0	0	0	0	0	0	0

可在任意时间对该寄存器读取;在 INITRQ=1 和 INITAK=1 时的初始化模式下的任意时间对该寄存器写入。

D7—SAMP 位,采样位。该位确定每位时间的 CAN 总线采样次数。如果 SAMP=0,得到的位值等于采样点位置的单个位的值。如果 SAMP=1,得到的位值是在 3 次采样中使用多数规则来决定。要实现更高比特速率,建议每个位时间只采样一次。

D6~D4—TSEG2[2:0]位,时间段 2 位。位时间内的时间段固定每个位时间的时钟周期数和采样点的位置。时间段 2(TSEG2)值可以如表 10-7 所列进行编程。

D3~D0—TSEG1[3:0]位,时间段 1 位。位时间内的时间段固定每个位时间的时钟周期数和采样点的位置。时间段 1(TSEG1)值可以如表 10-8 所列进行编程。

表 10-9 时间段 1 值

TSEG13	TSEG12	TSEG11	TSEG10	时间段 1
0	0	0	0	1 $T_q$ 时钟周期
0	0	0	1	2 $T_q$ 时钟周期
0	0	1	0	3 $T_q$ 时钟周期
0	0	1	1	4 $T_q$ 时钟周期
...	...	...	...	...
1	1	1	0	15 $T_q$ 时钟周期
1	1	1	1	16 $T_q$ 时钟周期

表 10-8 时间段 2 值

TSEG22	TSEG21	TSEG20	时间段 2
0	0	0	1 $T_q$ 时钟周期
0	0	1	2 $T_q$ 时钟周期
...	...	...	...
1	1	0	7 $T_q$ 时钟周期
1	1	1	8 $T_q$ 时钟周期

位时间由振荡器频率、波特率预分频器和每位的时间冲量( $T_q$ )数量确定,如表 10-6 和表 10-7 所列。公式为:

$$\text{Bit Time} = ((\text{预分频器值}) \div f_{\text{CANCLK}}) \times (1 + \text{时间段 1} + \text{时间段 2}) \quad (\text{式 } 10-3)$$

#### (5) MSCAN 接收器标志寄存器(CANRFLG)

每个标志只有在导致该设置不再有效的情况时才能通过软件清除(将 1 写入相应位位置)。每个标志在 CANRIER 寄存器中都有相关的中断使能位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	WUPIF	CSCIF	RSTAT1	RSTAT0	TSTAT1	TSTAT0	OVRIF	RXF
复位	0	0	0	0	0	0	0	0

**注意:**当处于初始化模式(INITRQ=1,INITAK=1)时,CANRFLG 寄存器保持复位状态。一旦退出初始化模式(INITRQ=0,INITAK=0),该寄存器即可重写。除了 RSTAT[1:0]和 TSTAT[1:0]标志是只读的;其余的位,写入 1 表示清除标志,写入 0 表示忽略标志。

D7—WUPIF 位,唤醒中断标志位。如果在处于休眠模式时 MSCAN 检测到 CAN 总线有信号且 CANTCTL0 中的 WUPE=1,那么该模块将设置 WUPIF。如果未被屏蔽,当设置了该标志时有一个唤醒中断产生。WUPIF=0,处于休眠模式时未检测到唤醒有效;WUPIF=1,MSCAN 检测到 CAN 总线上有信号并请求唤醒。

D6—CSCIF 位,CAN 状态变化中断标志位。当 MSCAN 由于发送错误计数器(TEC)和接收错误计数器的实际值而更改其当前 CAN 总线状态时,设置该标志。另外一个 4 位(RSTAT[1:0]、TSTAT[1:0])状态寄存器被分为几个独立段作为 TEC/REC,告知系统实际的 CAN 总线状态。如果未被屏蔽,当设置了该标志时有一个错误中断产生。CSCIF 提供一个拦截中断,这保证了接收器/发送器状态位(RSTAT/TSTAT)只有在无 CAN 状态变化中断产生时才进行更新。如果 TEC/REC 在 CSCIF 置位后更改其当前值,就会引起 RSTAT/TSTAT 位的其他状态变化。这些位会一直保持它们的状态,直到当前 CSCIF 中断被再次清除。CSCIF=0,自上次中断以来 CAN 总线状态未发生变化;CSCIF=1,MSCAN 更改了当前 CAN 总线状态。

D5~D4—RSTAT[1:0]位,接收器状态位。错误计数器的值控制着 MSCAN 的实际 CAN 总线状态。只要设置了状态变化中断标志(CSCIF),这些位就显示 MSCAN 的与接收器有关的适当 CAN 总线状态。位 RSTAT1、RSTAT0 的编码是:RSTAT[1:0]=00,RxOK:0≤接收错误计数器≤96;RSTAT[1:0]=01,RxWRN:96<接收错误计数器≤127;RSTAT[1:0]=10,RxERR:127<接收错误计数器;RSTAT[1:0]=11,Bus-off:发送错误计数器>255。

D3~D2—TSTAT[1:0]位,发送器状态位。错误计数器的值控制着 MSCAN 的实际 CAN 总线状态。只要设置了状态变化中断标志(CSCIF),这些位就显示 MSCAN 的与接收器有关的适当 CAN 总线状态。位 TSTAT1、TSTAT0 的编码是:TSTAT[1:0]=00,TxOK:0≤发送错误计数器≤96;TSTAT[1:0]=01,TxWRN:0<发送错误计数器≤127;TSTAT[1:0]=10,TxERR:127<发送错误计数器≤255;TSTAT[1:0]=11,Bus-off:发送错误计数器>255。

D1—OVRIF 位,溢出中断标志。在出现数据溢出情况时设置该标志。如果没有被屏蔽,当设置了该标志时有一个错误中断产生。OVRIF=0,无数据溢出情况;OVRIF=1,检测到数



据溢出。

D0—RXF 位,接收缓冲器已满标志位。当新报文被转移到接收器 FIFO 中时,RXF 由 MSCAN 进行置位。该标志表示移位缓冲器是否接收了正确的报文(匹配标识符,匹配循环冗余代码(CRC)和未检测到其他错误)。在 CPU 从接收器 FIFO 中的 RxFG 缓冲器那里读取了该报文后,RXF 标志必须清除,以释放缓冲器。已设置的 RXF 标志禁止下一个 FIFO 条目转移到前景缓冲器(RxFG)。如果未被屏蔽,当设置了该标志时有一个接收中断产生。RXF=0,RxFG 中没有新报文;RXF=1,接收器 FIFO 非空。RxFG 中有报文。

### (6) MSCAN 接收器中断使能寄存器(CANRIER)

该寄存器包含用于 CANRFLG 寄存器中描述的中断标志的中断使能位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	WUPIE	CSCIE	RSTATE1	RSTATE0	TSTATE1	TSTATE0	OVRIE	RXFIE
复位	0	0	0	0	0	0	0	0

当初始化模式处于有效状态时(INITRQ=1,INITAK=1),CANRIER 寄存器保持复位状态。一旦退出初始化模式(INITRQ=0,INITAK=0),该寄存器是可读的。RSTATE[1:0],TSTATE[1:0]位不受初始化模式影响。可在任意时间对该寄存器读取;可在初始化模式下的任意时间对该寄存器写入。

D7—WUPIE 位,唤醒中断使能位。WUPIE=0,无中断请求从该事件中产生;WUPIE=1,唤醒事件引起唤醒中断请求。

D6—CSCIE 位,CAN 状态变化中断使能位。CSCIE=0,无中断请求从该事件中产生;CSCIE=1,CAN 状态变化事件引起错误中断请求。

D5~D4—RSTATE[1:0],位接收器状态变化使能位。这些 RSTAT 使能位控制接收器状态变化而引起 CSCIF 中断的电平状态。独立于所选电平状态,RSTAT 标志继续显示实际接收器状态,且只有在没有 CSCIF 中断产生时才会更新。

00,未生成由于接收器状态变化而引起的任何 CSCIF 中断。

01,仅当接收器进入或离开 Bus-Off 状态时才会生成 CSCIF 中断。

为产生 CSCIF 中断丢弃其他接收器状态变化。

10,仅当接收器进入或离开 RxErr 或 Bus-Off 状态时才会产生 CSCIF 中断。

为产生 CSCIF 中断丢弃其他接收器状态变化。

11,所有状态变化都产生 CSCIF 中断。

D3~D2—TSTATE[1:0],发送器状态变化使能位。这些 TSTAT 使能位控制发送器状态变化而引起 CSCIF 中断的电平状态。独立于所选电平状态,TSTAT 标志继续显示实际发送器状态,且只有在没有 CSCIF 中断产生时才会更新。

00,未产生由于接收器状态变化而引起的任何 CSCIF 中断。

01,仅当发送器进入或离开 Bus-Off 状态时才会生成 CSCIF 中断。为产生 CSCIF 中断丢弃其他接收器状态变化。

10,仅当发送器进入或离开 Bus-Off 状态时才会生成 CSCIF 中断。为产生 CSCIF 中断丢弃其他接收器状态变化。

11,所有状态变化都产生 CSCIF 中断。

D1—OVR1E 位,溢出中断使能位。OVR1E=0,无中断请求从该事件中生成;OVR1E=1,溢出事件引起错误中断请求。

D0—RXF1E 位,总接收器已满中断使能位。RXF1E=0,无中断请求从该事件中生成;RXF1E=1,接收缓冲器满事件(成功报文接收)引起接收器中断请求。

### (7) MSCAN 发送器标志寄存器(CANTFLG)

每个发送缓冲区空标志在 CANTIER 寄存器中都有相关的中断使能位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	0	0	0	TXE2	TXE1	TXE0
复位	0	0	0	0	0	1	1	1

D7~D3—未定义。当初始化模式处于有效状态时(INITRQ=1,INITAK=1),CANT-FLG 寄存器保持复位状态。当未处于初始化模式时(INITRQ=0,INITAK=0),该寄存器可以写入。可在任意时间对该寄存器读取;可在 TXEx 标志不处于初始化模式的任意时间对该寄存器写入。写入 1 清除标志,写入 0 被忽略。

D2~D0—TXE[2:0]位,发送器缓冲区空标志位。该标志表示相关发送报文缓冲区空,因此没有预定的报文用于发送。在发送缓冲区中放好报文并准备好发送后,CPU 必须清除该标志。报文发送成功后,MSCAN 设置该标志。当发送请求由于中止请求而被成功中止时,MSCAN 也设置该标志。如果未被屏蔽,当设置了该标志时产生发送中断。清除 TXEx 标志也会清除相应的 ABTAKx。当设置了 TXEx 标志时,相应的 ABTRQx 位被清除。当监听模式处于有效状态时,TXEx 标志不能清除且不进行发送。当相应的 TXEx 位被清除(TXEx=0)且缓冲器被安排用于发送时,对发送缓冲器的读写操作会被阻止。TXE[2:0]=0,相关报文缓冲区已满(加载了准备发送的报文)TXE[2:0]=1,相关报文缓冲区空(未预定)。

### (8) MSCAN 发送器中断使能寄存器(CANTIER)

该寄存器包含发送缓冲器空中断标志的中断使能位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	0	0	0	TXEIE2	TXEIE1	TXEIE0
复位	0	0	0	0	0	0	0	0



D7~D3—未定义。当初始化模式处于有效状态时( $\text{INITRQ}=1, \text{INITAK}=1$ ), CANTIER 寄存器保持复位状态。当未处于初始化模式时( $\text{INITRQ}=0, \text{INITAK}=0$ ), 该寄存器可以写入。可在任意时间对该寄存器读取; 可在不处于初始化模式的任意时间对该寄存器写入。

D2~D0—TXEIE[2:0]位, 发送器空中断使能位。TXEIE[2:0]=0, 发送缓冲区空不产生中断请求。TXEIE[2:0]=1, 发送缓冲区空产生中断请求。

### (9) MSCAN 发送器报文中止请求寄存器(CANTARQ)

CANTARQ 寄存器中止队列报文的发送请求。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	0	0	0	ABTRQ2	ABTRQ1	ABTRQ0
复位	0	0	0	0	0	0	0	0

D7~D3—未定义。当初始化模式处于有效状态时( $\text{INITRQ}=1, \text{INITAK}=1$ ), CANTARQ 寄存器保持复位状态。当未处于初始化模式时( $\text{INITRQ}=0, \text{INITAK}=0$ ), 该寄存器可以写入。可在任意时间对该寄存器读取; 可在不处于初始化模式的任意时间对该寄存器写入。

D2~D0—ABTRQ[2:0]位, 中止请求位。CPU 设置 ABTRQ<sub>x</sub> 位, 请求中止预定的报文缓冲区( $\text{TXE}_x=0$ )。如果报文还没有开始发送, 或者如果发送没有成功(仲裁丢失或错误), MSCAN 就接受请求。当报文被中止时, 相关 TXE 位和中止确认标志被设置, 且若使能就触发发送中断。CPU 不能复位 ABTRQ<sub>x</sub>。每当设置了相关的 TXE 标志时, ABTRQ<sub>x</sub> 就被复位。ABTRQ[2:0]=0, 无中止请求; ABTRQ[2:0]=1, 中止请求产生。

### (10) MSCAN 发送器报文中止确认寄存器(CANTAACK)

如果由 CANTARQ 寄存器中的适当位请求的话, CANTAACK 寄存器表示成功中止报文发送队列的请求。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	0	0	0	ABTAK2	ABTAK1	ABTAK0
复位	0	0	0	0	0	0	0	0

D7~D3—未定义。当初始化模式处于有效状态时( $\text{INITRQ}=1, \text{INITAK}=1$ ), CANTAACK 寄存器保持复位状态。可在任意时间对该寄存器读取; 对该寄存器写入, ABTAK<sub>x</sub> 标志未定义。

D2~D0—ABTAK[2:0]位, 中止确认位。该标志确认由于 CPU 的产生发送中止请求而中止报文。当某个报文缓冲区被标志为空时, 软件可以使用该标志来确认报文是成功中止还是已发送出去。每当相应 TXE 标志被清除时, ABTAK<sub>x</sub> 标志就会清除。ABTAK[2:0]=0, 报文未被中止; ABTAK[2:0]=1, 报文被中止。



### (11) MSCAN 发送缓冲区选择寄存器(CANTBSEL)

CANTBSEL 允许实际发送报文缓冲区的选择,在 CANTXFG 寄存器空间访问。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	0	0	0	TX2	TX1	TX0
复位	0	0	0	0	0	0	0	0

D7~D3—未定义。当初始化模式处于有效状态时(INITRQ=1,INITAK=1),CANTBSEL 寄存器保持复位状态。当未处于初始化模式时,该寄存器可以写入(INITRQ = 0 ,INITAK = 0)。

D2~D0—TX [2 : 0]位,发送缓冲区选择位。编号最低位将各自的发送缓冲区放置到 CANTXFG 寄存器空间里(例如 TX1 = 1、TX0 = 1 选择发送缓冲器 TX0;TX1 = 1、TX0 = 0 选择发送缓冲器 TX1)。如果相应 TXEx 位被清除,缓冲器被安排用于传输,读写所选发送缓冲区将会被阻止。TX [2 : 0]=0,相关报文缓冲器没被选择;TX [2 : 0]=1,选择了相关报文缓冲器。

为了获得下一个有效的发送缓冲区,应用软件必须读取 CANTFLG 寄存器并将这个值写回到 CANTBSEL 寄存器中。在这个例子中,Tx 缓冲区 TX1 和 TX2 是有效的。因此,从 CANTFLG 寄存器读到的值是 0b0000\_0110。在将这个值写回到 CANTBSEL 中时,由于在位 1 的位置上已经将低编号为设置成 1,所以在 CANTXFG 中已经选择了 Tx 缓冲区的 TX1。因为只有最低编号位的位置被置 1,所以将从 CANTBSEL 寄存器中读回 0b0000\_0010。这个机制擦除应用软件的选择的下一次有效的 Tx 缓冲区。

LDAA CANTFLG;读到的值是 0b0000\_0110;

STAA CANTBSEL;写入的值是 0b0000\_0110;

LDAA CANTBSEL;读到的值是 0b0000\_0010。

若没有发送报文缓冲区被选择,则禁止访问 CANTXFG 寄存器。

### (12) MSCAN 标识符验收控制寄存器(CANTBSEL)

CANIDAC 寄存器用来标识符验收控制。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	0	0	IDAM1	IDAM0	0	IDHIT2	IDHIT1	IDHIT0
复位	0	0	0	0	0	0	0	0

D7、D6、D3—未定义。

D5~D4—IDAM[1 : 0]位,标识符验收模式位。CPU 设置这些标志来定义标识符验收滤波器。在滤波器关闭模式中,不接收任何消息以至于前端缓冲区不会被加载。设置模式如表 10-10 所列。



D2~D0—IDHIT[2:0]位,标识符验收命令中的指示位。MSCAN 设置这些标志来指示标识符验收命中,如表 10-11 所列。

表 10-11 标识符验收命中指示

IDHIT2	IDHIT1	IDHIT0	标识符验收命中
0	0	0	过滤器 0 命中
0	0	1	过滤器 1 命中
0	1	0	过滤器 2 命中
0	1	1	过滤器 3 命中
1	0	0	过滤器 4 命中
1	0	1	过滤器 5 命中
1	1	0	过滤器 6 命中
1	1	1	过滤器 7 命中

表 10-10 标识符验收模式设置

IDAM1	IDAM2	标识符验收模式
0	0	2 个 32 位验收过滤器
0	1	4 个 16 位验收过滤器
1	0	8 个 8 位验收过滤器
1	1	过滤器关闭

IDHIT<sub>x</sub> 指示符始终与在前台缓冲区(RxFG)中的报文相关。当报文移到接收器 FIFO 的前台缓冲区中时,将更新该指示符。

### (13) MSCAN 预留寄存器

该寄存器给厂家测试 MSCAN 模块预留的,在正常运行模式里是无效的。

### (14) MSCAN 杂项寄存器(CANMISC)

该寄存器提供附加功能。

D7~D1—未定义。

D0—BOHOLD 位,总线掉线状态直到用户请求。若 MSCAN 控制寄存器 1 中设置了 BORM 位,则这个位指示模块是否进入总线掉线状态。清除该位将请求从总线掉线中恢复。BOHOLD=0,模块没有掉线或者用户已经在掉线状态中请求恢复;BOHOLD=1,模块掉线并且在用户请求之前保持这个状态。复位各位置 0。

### (15) MSCAN 接收错误计数器(CANRXERR)

该寄存器反映了 MSCAN 接收错误计数器的状态。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RXERR7	RXERR 6	RXERR 5	RXERR 4	RXERR 3	RXERR 2	RXERR 1	RXERR 0
复位	0	0	0	0	0	0	0	0

**注意:**除了在睡眠或初始化模式里,读取该寄存器将返回错误的结果。在双核 CPU 的 MCU 中,这可能导致 CPU 错误状况。在专用模式里写这个寄存器可改变 MSCAN 功能。

### (16) MSCAN 发送错误计数器(CANTXERR)

该寄存器反映了 MSCAN 发送错误计数器的状态。



数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TXERR7	TXERR 6	TXERR 5	TXERR 4	TXERR 3	TXERR 2	TXERR 1	TXERR 0
复位	0	0	0	0	0	0	0	0

**注意:**除了在睡眠或初始化模式里,读取该寄存器将返回错误的结果。在双核 CPU 的 MCU 中,这可能导致 CPU 错误状况。在专用模式里写这个寄存器可改变 MSCAN 功能。

### (17) MSCAN 标识符验收寄存器(CANIDAR0~7)

接收时,每个报文被写入到后台接收缓冲区。若报文通过了标识符验收和标识符掩码寄存器的验收,CPU 只被告知读取报文;否则会被下一次报文覆盖。

在扩展标识符中,所有这 4 个验收和屏蔽寄存器都会使用。在标准标识符中,只有前两个(CANIDAR0/1,CANIDMR0/1)被使用。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
复位	0	0	0	0	0	0	0	0

D7~D0—AC[7:0]位,验收码位。AC[7:0]包括了用户定义的位串,接收报文缓冲区的相关标识符寄存器(IDRn)位将和这个位串比较。这个比较结果将被相应的标识符屏蔽寄存器屏蔽。

### (18) MSCAN 标识符掩码寄存器(CANIDMR0~7)

标识符掩码寄存器指定标识符验收寄存器的相应位与验收滤波器相关。为了在 32 位滤波模式里接收正确的标识符,需要编程掩码寄存器 CANIDMR1 和 CANIDMR5 的最后 3 位(AM[2:0])为无关位。为了在 16 位滤波模式里接收正确的标识符,需要编程掩码寄存器 CANIDMR1、CANIDMR3、CANIDMR5 和 CANIDMR7 的最后 3 位(AM[2:0])为无关位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
复位	0	0	0	0	0	0	0	0

D7~D0—AM[7:0]位,验收掩码位。若这个寄存器的某一位被清除,则这就表明在标识符验收寄存器中的相应位必须和它的标识符位一样,才进行匹配检测。如果所有的位都匹配,那么就接收这个报文。若有一位被设置了,则表示在标识符验收寄存器中的相应位不会影响报文是否被接收。为 0,匹配验收代码寄存器和标识符相应位;为 1,忽略验收代码寄存器相应位。

## 2. 报文存储机制

为了简化编程接口,接收和发送报文缓冲区采用统一的结构。如表 10-12 所列,每个缓



缓冲区拥有 16 个字节,其中包括 13 个字节的数据结构。发送报文缓冲区有一个优先级寄存器 (TBPR),而接收报文缓冲区没有。最后的两个字节存储了一个特殊的 16 位的时间戳,只能由 MSCAN 模块进行写操作,CPU12 只能读取其中的内容。

表 10-12 报文缓冲区结构

地址	寄存器名称
\$_x0~\$_x3	标识符寄存器 0~3
\$_x4~\$_xB	数据段寄存器 0~7
\$_xC	数据长度寄存器
\$_xD	发送缓冲区优先级寄存器
\$_xE	时间戳寄存器(高字节)
\$_xF	时间戳寄存器(低字节)

图 10-23 描述了适应扩展帧标识符的发送、接收缓冲区中的 13 个字节的数据结构。发

me		Bit7	6	5	4	3	2	1	Bit0	地址
IDR0	读写	ID28	ID27	ID26	ID25	ID24	ID24	ID24	ID21	\$_x0
IDR1	读写	ID20	ID19	ID18	SRR(=1)	IDE(=1)	IDE(=1)	ID16	ID15	\$_x1
IDR2	读写	ID14	ID13	ID12	ID11	ID10	ID10	ID8	ID7	\$_x2
IDR3	读写	ID6	ID5	ID4	ID3	ID2	ID2	ID1	RTR	\$_x3
DSR0	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x4
DSR1	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x5
DSR2	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x6
DSR3	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x7
DSR4	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x8
DSR5	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_x9
DSR6	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_xA
DSR7	读写	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	\$_xB
DLC	写					DLC3	DLC2	DLC1	DLC0	\$_xC


 = 不用

图 10-23 发送、接收缓冲区扩展标识符

送缓冲区任何时刻都可以读取,但只有在发送标志 TXEx 置位且相应的发送缓冲区被选中(通过 CANTBSEL 的设置来确定)时才可以写入。接收缓冲区不能写,但在 RXF 标志置位时可以读取。

图 10-24 描述了适应标准帧的 IDR 寄存器部分。

	Bit 7	6	5	4	3	2	1	Bit 0	地址
IDR0 读写	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	\$_x0
IDR1 读写	ID2	ID3	ID3	RTR	IDE(=0)				\$_x1
IDR2 读写									\$_x2
IDR3 读写									\$_x3


 = 不用

图 10-24 标准帧标识符的映射

(1) 标识符寄存器(IDR0~3)

扩展帧格式的标识符由全部的 32 位组成,包括 ID28~ID0、SRR、IDE 和 RTR。而标准帧格式的标识符由 13 位组成:ID10~ID0、RTR 和 IDE。

ID28~ID0——扩展帧标识符

此 29 位的标识符适用于扩展帧,ID28 是最重要的位并在总线仲裁时先被发送,数值小的标识符拥有更高的优先级。

ID10~ID0——标准帧标识符

此 11 位的标识符适用于标准帧,ID10 是最重要的位并在总线仲裁时先被发送,数值小的标识符拥有更高的优先级。

SRR——替代远程请求位

这个固定的接收位仅适用于扩展帧模式。它在扩展格式的标准帧 RTR 位位置,代替标准帧的 RTR 位。

IDE——识别符扩展位

该标志指示是标准模式还是扩展模式。IDE=1,扩展帧模式(29 位);IDE=0,标准帧模式(11 位)。

RTR——远程发送请求位

该标志指示是数据帧还是远程帧。RTR=1,远程帧;RTR=0,数据帧。

(2) 数据段寄存器(DSR0~7)

此 8 个寄存器中包含实际发送或接收的数据。

(3) 数据长度寄存器(DLR)

数据长度代码指示了数据场里的字节数量,取值范围 0~8。



#### (4) 发送缓冲区优先级寄存器(TBPR)

这个寄存器定义了相应发送缓冲区的局部优先级。较小的二进制数值具有较高的优先级。

## 10.4 MSCAN 模块双机通信测试实例

### 10.4.1 测试模型

CAN 双机测试工程分两个主要部分,发送方 Sender 和接收方 Receiver。Sender 定期在 CAN 总线上发送广播帧;Receiver 接收到总线上的广播帧后经过简单地解析,将接收到的帧信息通过串口发送到 PC 机,在 PC 机上运行的串口调试工具软件的显示界面中显示总线上帧的信息。

在此用到的测试模型是经过简化的实例,只实现了 CAN 总线的物理层通信,即 CAN 帧的收与发。屏蔽了本地标识符匹配,对帧标识符 ID 的判断交由高层执行。屏蔽了 CAN 通信协议,握手通信也交由高层执行。但需要注意的是,MSCAN 模块本身在硬件上内置了标识符匹配和通信协议 CAN Protocol Version 2.0A/B 的功能,若希望使用这些功能,进行相应的配置即可启动。用户亦可根据实际的需要,在此测试模型的基础之上定制适合于特定情况下的标识符匹配与通信协议,同时可以扩展为多节点通信。

### 10.4.2 编程要点

#### (1) 初始化函数 CANInit

在初始化函数中需要完成的步骤有:

- ① 使能 CAN 模块;
- ② 进入休眠模式;
- ③ 进入 CAN 模块初始化模式;
- ④ 置 CAN 模块过滤器与本地标识符 ID;
- ⑤ 配置 CAN 总线时钟频率及时钟源;
- ⑥ 配置 CAN 模块工作方式(侦听,回环等);
- ⑦ 配置 CAN 模块中断;
- ⑧ 退出 CAN 模块初始化模式;
- ⑨ 等待 CAN 总线通信时钟同步。

#### (2) CAN 发送帧函数 CANSendFrame

此函数完成通过 CAN 总线发送帧的功能,主要执行过程如下:

- ① 判断 CAN 总线上的同步状态,确保通信时总线是同步的。

- ② 查找并选中 CAN 模块中空闲的发送缓冲区以写入帧数据。
- ③ 配置帧数据(标识符 ID,数据段,数据长度,发送优先级)。
- ④ 清 TXEx 标志位,通知 CAN 模块发送帧。

### (3) CAN 接收帧函数 CANReceiveFrame

此函数完成从接收缓冲区中读取帧的功能,主要执行过程如下:

- ① 检测接收状态标志位 RXF,若没有信息则退出。
- ② 判断帧类型(标准帧/扩展帧,数据帧/远程帧),以进行相应的处理。
- ③ 处理标准数据帧,读取帧中数据段内容。
- ④ 清 RXF 标志位,释放使用过的接收缓冲区。

### (4) 封装帧结构函数 CANFillFrame

在编写和使用 CAN 模块处理帧的程序时,用到的多是封装成帧数据结构类型 CAN-Frame 的变量。通过将帧信息(标识符 ID、数据段、数据长度、发送优先级等)封装到帧结构,便于控制操作,并且便于简单明了地理解程序,也体现了面向对象的思想。

## 10.4.3 CAN 模块底层构件设计

### 1. CAN 模块头文件 CAN.h

```
// -----*
// 文件名: CAN.h                                     *
// 说 明: CAN 构件头文件                             *
// -----*

#ifndef CAN_H
#define CAN_H

//头文件包含
#include "Includes.h"

//定义 CAN 通信的帧结构
typedef struct CanFrame
{
    uint32 m_ID; //msg 发送方 ID
    uint8 m_IDE; //扩展帧为 1,标准帧为 0
    uint8 m_RTR; //远程帧为 1,数据帧为 0
    uint8 m_data[8]; //帧数据
    uint8 m_dataLen; //帧数据长度
    uint8 m_priority; //发送优先级
}
```



```
} CANFrame;

//CANInit:CAN 通信初始化-----*
//功 能:CAN 初始化                                     *
//参 数:无                                             *
//返 回:无                                             *
//说 明:在 9.83M 外部晶振频率下将总线速率设为 197kbps, *
//      不过滤接收 ID,ID 判断留在高层执行             *
//-----*
void CANInit(void);

//CANSendFrame:CAN 发送 1 帧数据-----*
//功 能:CAN 发送 1 帧(数据长度≤ 8)                   *
//参 数:sendFrame:发送帧结构的引用                   *
//返 回:uint8 发送状态,                               *
//      0:发送成功                                     *
//      1:数据长度错误                               *
//      2:总线未同步                                   *
//      3:发送帧为扩展帧                             *
//-----*
uint8 CANSendFrame(CANFrame * sendFrame);

//CANReceiveFrame:CAN 接收 1 帧数据-----*
//功 能:CAN 接收 1 帧                                 *
//参 数:receiveFrame:接收帧结构的引用                 *
//返 回:uint8 接收帧状态,                             *
//      0:接收成功                                     *
//      1:接收标志未置位,未接收帧                   *
//      2:收到扩展帧                                 *
//说 明:接收帧数据被封装在帧结构中,可以通过访问帧结构获取接收信息 *
//-----*
uint8 CANReceiveFrame(CANFrame * receiveFrame);

//CANFillFrame:封装 CAN 数据帧结构-----*
//功 能:封装 CAN 数据帧结构                           *
//参 数:receiveFrame:帧结构的引用                     *
//      id:帧标识符 ID                               *
//      ide:扩展帧为 1,标准帧为 0                   *
//      rtr:远程帧为 1,数据帧为 0                   *
```

```

//      data:帧数据(数组)                                *
//      len:帧数据长度                                    *
//      priority:帧发送优先级                              *
//返回:uint8 接收帧状态,                                  *
//      0:封装成功                                        *
//      1:标准/扩展帧模式选择错误                          *
//      2:数据/远程帧模式选择错误                          *
//      3:数据长度输入错误                                *
//说明:将帧参数封装在帧结构中,便于发送与接收的操作      *
//-----*
uint8 CANFillFrame( CANFrame * frame,
                   uint32 id,
                   uint8 ide,
                   uint8 rtr,
                   uint8 * data,
                   uint8 len,
                   uint8 priority);

#endif

```

## 2. CAN 模块代码文件 CAN.c

```

//-----*
//文件名: CAN.c                                          *
//说明: CAN 构件函数源文件                              *
//-----*

//头文件包含,及宏定义区

// 头文件包含
#include "CAN.h"

//构件函数实现
//CANInit:CAN 通信初始化-----*
//功 能:CAN 初始化                                      *
//参 数:无                                              *
//返 回:无                                              *
//说明:在 9.83M 外部晶振频率下将总线速率设为 197 kbps, *
//      不过滤接收 ID,ID 判断留在高层执行              *
//-----*

```



```
void CANInit(void)
{
    //MSCAN12 模块使能
    CANOCTL1_CANE = 1;

    //请求进入初始化模式
    CANOCTL0_INITRQ = 1;

    //等待应答进入初始化模式
    while (CANOCTL1_INITAK == 0);

    //关闭过滤器
    CAN0IDMR0 = 0xFF;
    CAN0IDMR1 = 0xFF;
    CAN0IDMR2 = 0xFF;
    CAN0IDMR3 = 0xFF;
    CAN0IDMR4 = 0xFF;
    CAN0IDMR5 = 0xFF;
    CAN0IDMR6 = 0xFF;
    CAN0IDMR7 = 0xFF;

    //配置时钟
    //同步跳转宽度为 1, 预分频因子为 5。将位速率设置成 197kbps
    //bit.7-6=00, 同步跳转宽度为 1, bit.5-0=000100, 预分频因子为 5
    CANOBTR0 = 0x04;
    //bit.7=0, 单次采样, bit.6-4=010, 时间段 2 为 3, bit.3-0=0101,
    //时间段 1 为 6, 0b00100101, 故位速率 = 1.97M/10 = 197kbps
    CANOBTR1 = 0x25;
    CANOCTL1_CLKSRC = 0; //采用内部时钟

    //配置工作模式
    CANOCTL1_LISTEN = 0; //侦听模式禁止
    CANOCTL1_LOOPB = 1; //启动自环测试模式

    //设置中断方式
    CANOTIER = 0x00; //禁止发送中断
    CANOTIER = 0x00; //禁止接收中断

    //退出初始化模式
```



```

CANOCTL0_INITRQ = 0;

//等待应答初始化模式
while (CANOCTL1_INITAK == 1);

//等待总线通信时钟同步
while (CANOCTL0_SYNCH == 0);
}

//CANSendFrame:CAN 发送 1 帧数据 -----*
//功 能:CAN 发送 1 帧(数据长度≤ 8) *
//参 数:sendFrame:发送帧结构的引用 *
//返 回:uint8 发送状态, *
//      0:发送成功 *
//      1:数据长度错误 *
//      2:总线未同步 *
//      3:发送帧为扩展帧 *
// -----*
uint8 CANSendFrame(CANFrame * sendFrame)
{
    uint8 txEmptyBuf;//空闲缓冲区掩码
    uint8 i;

    //检查数据长度
    if (sendFrame->m_dataLen > 8)
    {
        return 1;
    }

    //检查总线是否同步
    if (CANOCTL0_SYNCH == 0)
    {
        return 2;
    }

    //寻找空闲缓冲区
    txEmptyBuf = 0;
    do
    {
        CANOTBSEL = CANOTFLG;

```



```
    txEmptyBuf = CAN0TBSEL;
}
while (! txEmptyBuf);

//写入标识符 ID
if (sendFrame->m_IDE == 0) //按标准帧填充 ID
{
    CAN0TXIDR0 = (uint8)(sendFrame->m_ID>>3);
    CAN0TXIDR1 = (uint8)(sendFrame->m_ID<<5);
    CAN0TXIDR1_SRR = sendFrame->m_RTR;
    CAN0TXIDR1_IDE = sendFrame->m_IDE;
}
else
{
    return 3; //发送帧为扩展帧
}

//写入数据段
if (sendFrame->m_RTR == 0) //按数据帧填充
{
    for (i = 0; i < sendFrame->m_dataLen; i++)
    {
        *(&CAN0TXDSR0) + i = sendFrame->m_data[i];
    }
    CAN0TXDLR = sendFrame->m_dataLen;
}
else //按远程帧填充
{
    CAN0TXDLR = 0;
}

//配置发送优先级
CAN0TXBPR = sendFrame->m_priority;

//清 TXEx 标志,通知 CAN 模块发送
CAN0TFLG = txEmptyBuf;

return 0; //配置发送完成
}
```

```

//CANReceiveFrame:CAN 接收 1 帧数据 -----*
//功 能:CAN 接收 1 帧 *
//参 数:receiveFrame:接收帧结构的引用 *
//返 回:uint8 接收帧状态, *
//      0:接收成功 *
//      1:接收标志未置位,未接收帧 *
//      2:收到扩展帧 *
//说 明:接收帧数据被封装在帧结构中, *
//      成功接收后,可以通过访问帧结构获取接收信息 *
//-----*

uint8 CANReceiveFrame(CANFrame * receiveFrame)
{
    uint8 i;

    //检测接收标志
    if (CANORFLG_RXF == 0)
    {
        return 1; //未接收帧
    }

    //判断标准帧/扩展帧
    if (CANORXIDR1_IDE == 0) //收到标准帧
    {
        receiveFrame->m_ID = (uint32)(CANORXIDR0<<3)
            | (uint32)(CANORXIDR1>>5);
        receiveFrame->m_RTR = CANORXIDR1_SRR;
        receiveFrame->m_IDE = 0;
    }
    else //收到扩展帧
    {
        return 2;
    }

    //判断数据帧/远程帧
    if (CANORXIDR1_SRR == 0) //收到数据帧
    {
        receiveFrame->m_dataLen = CANORXDRLR_DLC;
        //读取数据段内容
        for (i = 0; i < receiveFrame->m_dataLen; i++)
        {

```



```
        receiveFrame->m_data[i] = * ((&CAN0RXDSR0) + i);
    }
}
else //收到远程帧
{
    receiveFrame->m_dataLen = 0;
}

//清 RXF 标志位,准备接收下一帧
CANORFLG_RXF = 1;

return 0; //成功接收标准帧
}

//CANFillFrame:封装 CAN 数据帧结构 ----- *
//功 能:封装 CAN 数据帧结构 *
//参 数:receiveFrame:帧结构的引用 *
//      id:帧标识符 ID *
//      ide:扩展帧为 1,标准帧为 0 *
//      rtr:远程帧为 1,数据帧为 0 *
//      data:帧数据(数组) *
//      len:帧数据长度 *
//      priority:帧发送优先级 *
//返 回:uint8 接收帧状态, *
//      0:封装成功 *
//      1:标准/扩展帧模式选择错误 *
//      2:数据/远程帧模式选择错误 *
//      3:数据长度输入错误 *
//说 明:将帧参数封装在帧结构中,便于发送与接收的操作 *
// ----- *
uint8 CANFillFrame(CANFrame * frame,
                   uint32 id,
                   uint8 ide,
                   uint8 rtr,
                   uint8 * data,
                   uint8 len,
                   uint8 priority)
{
    uint8 i;
```

```

frame->m_ID = id;

if (ide > 2)    //标准/扩展帧模式选择错误
{
    return 1;
}
frame->m_IDE = ide;

if (rtr > 2)    //数据/远程帧模式选择错误
{
    return 2;
}
frame->m_RTR = rtr;

if (len > 8)    //数据长度输入错误
{
    return 3;
}
frame->m_dataLen = len;
for (i = 0; i < frame->m_dataLen; i++)
{
    frame->m_data[i] = data[i];
}
frame->m_priority = priority;

return 0;    //封装成功
}

```



### 3. 发送方 Sender 的主程序 main.c

```

// -----*
// 工 程 名: CAN 总线通信发送方 *
// 硬件连接: 详情参见书中说明 *
// 程序描述: CAN 总线双机通信 *
//          定期广播在总线上发送包含"IamNodeA"信息的 CAN 数据帧 *
// 目    的: 提供 CAN 总线多机通信测试样例 *
// 说    明: 此程序为发送方,至少再需要一个接收方才能完成实验 *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*

//包含头文件

```



```
#include "Includes.h"      //包含总头文件

//在此添加全局变量定义

//主函数
void main(void)
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;      //运行计数器
    uint8 sendData[] = {'T','a','m','N','O','d','e','A'};  //CAN 发送数据内容
    uint8 sendDataLength = 8;  //CAN 发送数据长度
    CANFrame sendFrame;
    uint8 SNDFlag;

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
    Light_Init(Light_Fun1_PORT,Light_Fun1,Light_OFF); //CAN 运行指示灯初始化
    SCIIInit(0,FBUS_32M,9600);          //串口 0 初始化
    CANInit();          //CAN 模块初始化为非自环测试模式

    SCISendString(0, "Start sending CAN frame.\n");

    //封装 CAN 发送帧
    (void)CANFillFrame(&sendFrame, 1, 0, 0, sendData, sendDataLength, 0);

    // 主循环
    for(;;)
    {
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + + ;
        if (mRuncount >= 50000)
        {
            mRuncount = 0;
        }
    }
}
```

```

    Light_Change(Light_Run_PORT,Light_Run);    //指示灯的亮、暗状态切换
}

// -----
//2. 主循环计数到一定的值,通过 CAN 发送数据,并闪烁功能指示灯
if (mRuncount == 25000)
{
    //发送 CAN 帧
    SNDFlag = CANSendFrame(&sendFrame);
    if(SNDFlag == 0)    // 若发送成功
    {
        SCISendString(0, "Send successfully.\n");
    }
    else                // 发送不成功
    {
        SCISendString(0, "Send failure.\n");
    }
}

    Light_Change(Light_Fun1_PORT,Light_Fun1);    //CAN 指示灯亮暗切换
}

// -----
} //for_end(主循环结束)
} //main_end

```

#### 4. 接收方 Receiver 的主程序 main.c

```

// ----- *
// 工 程 名: CAN 总线通信接收方 *
// 硬件连接: 详情参见书中说明 *
// 程序描述: CAN 总线双机通信 *
//          接收定期在总线上广播的包含"IamNodeA"信息的 CAN 数据帧 *
// 目    的: 提供 CAN 总线多机通信测试样例 *
// 说    明: 此程序为接收方,需要一个发送方才能完成实验 *
// -----*
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 -----*

//包含头文件
#include "Includes.h"    //包含总头文件

//在此添加全局变量定义

```



```
//主函数
void main(void)
{
    //0.1 主程序使用的变量定义
    uint32 mRuncount = 0;          //运行计数器
    CANFrame rcvFrame;  //接收帧结构
    uint8 RCVFlag;      //接收状态标志
    uint8 i;

    //0.2 关总中断
    DisableInterrupt();

    //0.3 芯片初始化
    MCUInit(FBUS_32M);

    //0.4 模块初始化
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
    Light_Init(Light_Fun1_PORT,Light_Fun1,Light_OFF); //CAN 运行指示灯初始化
    SCIInit(0,FBUS_32M,9600);                          //串口 0 初始化
    CANInit();      //CAN 模块初始化为非自环测试模式

    SCISendString(0, "Start receiving CAN frame.\n");

    // 主循环
    for(;;)
    {
        //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
        mRuncount + + ;
        if (mRuncount >= 50000)
        {
            mRuncount = 0;
            Light_Change(Light_Run_PORT,Light_Run);    //指示灯的亮、暗状态切换
        }

        // -----
        //2. 主循环计数到一定的值,通过 CAN 接收数据,并闪烁功能指示灯
        if (mRuncount == 25000)
        {
```



```

RCVFlag = CANReceiveFrame(&rcvFrame);
switch (RCVFlag)
{
    case 0: //若接收一标准数据帧,对帧内容进行解析
        SCISendString(0, "Receive a standard frame.\n");
        //解析帧源 ID
        SCISendString(0, "IDinfo = ");
        for (i = 0; i < 4; i++)
        {
            SCISend1(0, (uint8)(rcvFrame.m_ID >> ((3 - i) * 8)) + '0');
            SCISend1(0, '');
        }
        SCISend1(0, '\n');
        //解析帧数据长度
        SCISendString(0, "DataLen = ");
        SCISend1(0, rcvFrame.m_dataLen + '0');
        SCISend1(0, '\n');
        //解析帧数据内容
        SCISendString(0, "Data = ");
        for (i = 0; i < rcvFrame.m_dataLen; i++)
        {
            SCISend1(0, rcvFrame.m_data[i]);
            SCISend1(0, '');
        }
        SCISendString(0, "\n\n");
        Light_Change(Light_Fun1_PORT, Light_Fun1); //CAN 指示灯亮暗切换
        break;
    case 1: //若未接收到帧
        SCISendString(0, "Rceived no frame.\n\n");
        break;
    case 2: //若接收到远程帧
        SCISendString(0, "Receive a remote frame.\n\n");
        break;
    default:
        break;
}

}

// -----

```



```

    } //for_end(主循环结束)
} //main_end

```

#### 10.4.4 测试操作要点

测试时要注意正确地连接跳线：

- ① 保证每个节点上芯片的 CANT<sub>x</sub> 和 CANR<sub>x</sub> 同收发器芯片的对应引脚相连。
- ② 将两个通信节点从收发器引出的 CANH 和 CANL 分别对接,在 CANH 与 CANL 线路之间并联两个 120Ω 的电阻,形成环路,如图 10-2 所示。
- ③ 确保两个通信节点的总线共地共电源,即将两个节点的 GND 和 VCC 连接在一起。
- ④ 将 Receiver 端通过串口连接到 PC 机,在 PC 机端启动串口调试工具软件。
- ⑤ 分别将 Sender 和 Receiver 的程序写入两个节点的 MCU 后,先启动 Receiver,再启动 Sender。

若上述操作无误,将会从串口调试软件输出界面中看到总线上发送的广播帧信息。

在此测试工程的基础之上可以实现多节点通信和自定义通信协议的通信。

### 10.5 MSCAN 模块的自环通信实例

#### 10.5.1 测试模型

当 MSCAN 模块发送时,MSCAN 接收自己发送的报文到接收后台缓冲区(RxBG)中,且在自环模式下 MSCAN 会将该报文当成是从总线上接收到的。这样,利用自环模式就可以在不增加硬件的条件下模拟发送和接收报文。

#### 10.5.2 编程要点及设计代码

##### 1. 对模块代码文件的修改

进行自环通信测试,只需在双机通信的模块代码的基础之上,在模块的初始化函数中激活自环模式即可。从修改后的初始化函数代码(如 CAN. c.)中可以发现,与之前的测试工程代码相比,只添加了一句激活自环模式的代码。

##### 2. 自环通信测试主程序 main. c

```

// -----*
// 工 程 名: CAN 总线自环通信测试 *
// 硬件连接: 详情参见书中说明 *
// 程序描述: 本机通过 CAN 模块发送并接收数据帧 *

```

```
// 目的：提供 CAN 总线自环通信测试样例 *
// 说明：此程序可在不加任何跳线的前提下进行 CAN 模块测试 *
// -----苏州大学飞思卡尔嵌入式系统实验室 2011 年 ----- *
```

```
//包含头文件
```

```
#include "Includes.h" //包含总头文件
```

```
//在此添加全局变量定义
```

```
//主函数
```

```
void main(void)
```

```
{
```

```
    //0.1 主程序使用的变量定义
```

```
    uint32 mRuncount = 0; //运行计数器
```

```
    CANFrame rcvFrame; //接收帧结构
```

```
    uint8 RCVFlag; //接收状态标志
```

```
    uint8 i;
```

```
    uint8 sendData[] = {'T','a','m','N','O','d','e','A'}; //CAN 发送数据内容
```

```
    uint8 sendDataLength = 8; //CAN 发送数据长度
```

```
    CANFrame sendFrame;
```

```
    uint8 SNDFlag;
```

```
    //0.2 关总中断
```

```
    DisableInterrupt();
```

```
    //0.3 芯片初始化
```

```
    MCUInit(FBUS_32M);
```

```
    //0.4 模块初始化
```

```
    Light_Init(Light_Run_PORT,Light_Run,Light_OFF); //RUN 指示灯初始化为暗
```

```
    Light_Init(Light_Fun1_PORT,Light_Fun1,Light_OFF); //CAN 运行指示灯初始化
```

```
    SCIInit(0,FBUS_32M,9600); //串口 0 初始化
```

```
    CANInit(); //CAN 模块初始化为非自环测试模式
```

```
    //封装 CAN 发送帧
```

```
    (void)CANFillFrame(&sendFrame, 1, 0, 0, sendData, sendDataLength, 0);
```



```
// 主循环
for(;;)
{
    //1. 主循环计数到一定的值,使 Run 灯的亮、暗状态切换
    mRuncount + + ;
    if (mRuncount >= 50000)
    {
        mRuncount = 0;
        Light_Change(Light_Run_PORT, Light_Run);    //指示灯的亮、暗状态切换
    }

    // -----
    //2. 主循环计数到一定的值,进行一次 CAN 自环测试,并闪烁功能指示灯
    if (mRuncount == 25000)
    {
        SCISendString(0, "Start receiving CAN frame.\n");
        //发送 CAN 帧
        SNDFlag = CANSendFrame(&sendFrame);
        if(SNDFlag == 0)    // 若发送成功
        {
            SCISendString(0, "Send successfully.\n");
            //接收 CAN 帧
            RCVFlag = CANReceiveFrame(&rcvFrame);
            switch (RCVFlag)
            {
                case 0:    //若接收一标准数据帧,对帧内容进行解析
                    SCISendString(0, "Receive a standard frame.\n");
                    //解析帧源 ID
                    SCISendString(0, "IDinfo = ");
                    for (i = 0; i < 4; i + + )
                    {
                        SCISendl(0, (uint8)(rcvFrame.m_ID >> ((3 - i) * 8)) + '0');
                        SCISendl(0, ',');
                    }
                    SCISendl(0, '\n');
                    //解析帧数据长度
                    SCISendString(0, "DataLen = ");
                    SCISendl(0, rcvFrame.m_dataLen + '0');
                    SCISendl(0, '\n');
```

```

//解析帧数据内容
SCISendString(0, "Data = ");
for (i = 0; i < rcvFrame.m_dataLen; i + +)
{
    SCISend1(0, rcvFrame.m_data[i]);
    SCISend1(0, '');
}
SCISendString(0, "\n\n");
Light_Change(Light_Fun1_PORT, Light_Fun1); //CAN 指示灯亮暗切换
break;
case 1: //若未接收到帧
    SCISendString(0, "Rceived no frame.\n\n");
    break;
case 2: //若接收到远程帧
    SCISendString(0, "Receive a remote frame.\n\n");
    break;
default:
    break;
}
Light_Change(Light_Fun1_PORT, Light_Fun1); //CAN 指示灯亮暗切换
}
else // 发送不成功
{
    SCISendString(0, "Send failure.\n");
}
}
// -----
} //for_end(主循环结束)
} //main_en

```

将程序编译后写入芯片,复位运行。操作无误,将会从串口调试软件输出界面中看到回环测试模式下自己发送给自己的帧信息。

# 第 11 章

## 系统时钟与其他功能模块

通过前面的学习,掌握了 Freescale S12X 的基础知识和大部分基础功能模块,如通用 I/O、串口等,本章主要知识点包括:①系统时钟与复位;②系统时钟锁相环;③实时中断;④看门狗;⑤低功耗模式,这些内容一般会在程序的初始化中使用。本章的重点为时钟与复位产生模块概述、CRG 模块的初始化。相比前面的章节,此章内容复杂且较难理解,但若要全面掌握 Freescale S12X 的开发,这又是必不可少的一部分。

### 11.1 时钟与复位产生模块概述

时钟与复位产生模块 CRG(Clock and Reset Generator)内含锁相环 PLL(Phase Locked Loop)电路,负责产生 MCU 内部模块所需要的时钟信号,如串口、AD 转换、PWM、定时器等。

#### 11.1.1 锁相环技术

MCU 的支撑电路一般需要外部时钟源向 MCU 提供时钟信号,由于过高频率的外部时钟容易受到干扰,所以不能使用太高的外部时钟源;但若系统运行更加快速,就需要在芯片内部提升系统所使用的时钟频率。例如,在 MCU 外部使用 8 MHz 无源晶振产生的时钟信号进入 MCU 内部通过锁相环倍频处理后,让系统用到的总线频率达到 16 MHz 或更高,从而使系统工作在稳定并且较高的总线频率上。首先,将介绍与锁相环有关的基本概念。

##### 1. 锁相环技术与频率合成技术

在电子设备、仪器仪表中常常需要有稳定性强、精度高的频率源,而锁相环技术就是实现相位自动控制的一门科学,利用它可以得到频带范围宽、信道多、稳定性强、精度高的频率源。所谓频率合成技术,就是利用一个或几个具有高稳定性和高精度的频率源(一般由晶体振荡器产生),通过对它们进行加减(混频),乘(倍频),除(分频)运算,产生大量的具有相同频率稳定性和频率精度的频率信号。锁相环频率合成技术在通信、雷达、导航、宇航、遥控遥测、电子技术测量等领域都有广泛的应用。

为了得到稳定性强、精度高的频率源,通常采用频率合成技术。频率合成技术主要有两种:直接频率合成技术和间接频率合成技术。直接频率合成技术是将一个或几个晶体振荡器产生的频率信号通过谐波发生器产生一系列频率信号,然后再对这些频率信号进行倍频、分频和混频,最后得到大量的频率信号。优点是:频率稳定性强,频率转换时间短(可达微秒量级),能做到很小的频率间隔。缺点是:系统中要用到大量的混频器、滤波器等,从而导致体积大,成本高,安装调试复杂,故只用于频率精度要求很高的场合。间接频率合成技术是利用锁相环技术来产生大量的具有高稳定性和高精度的频率源。由于间接频率合成器的关键部件是锁相环,故通常称为锁相环频率合成器。由于锁相环频率合成器一般只加一个分频器和一个一阶低通滤波器,易于集成,故其具有体积小、重量轻、成本低、安装和调试简单等优点。锁相环频率合成器在性能上逐渐接近直接频率合成器,所以它在电子技术中得到了日益广泛的应用,并在应用中得到迅速发展。

## 2. 锁相环频率合成器的基本原理

锁相环电路是一个负反馈环路。图 11-1 给出了一种最简单的 PLL 频率合成器的框图。它由基准频率源、鉴相器、低通滤波器、压控振荡器和反馈分频器组成。

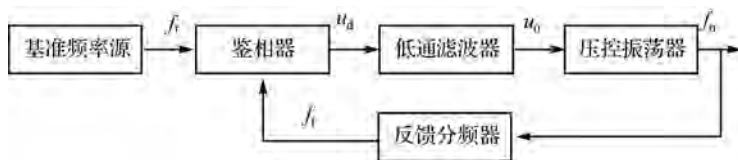


图 11-1 锁相环频率合成器的原理框图

**基准频率源:**基准频率源提供一个稳定频率源,其频率为  $f_r$ ,一般用精度很高的石英晶体振荡器产生,是锁相环的输入信号。

**鉴相器:**鉴相器是一个误差检测元件。它将基准频率源的输出信号  $f_r$  的相位与压控振荡器输出信号  $f_o$  的相位相比较,产生一个电压输出信号  $u_d$ ,其大小取决于两个输入信号的相位差。

**低通滤波器:**低通滤波器的输入信号是鉴相器的输出电压信号  $u_d$ ,经过低通滤波器后  $u_d$  的高频分量被滤除,输出控制电压  $u_o$  去控制压控振荡器。

**压控振荡器(VCO):**压控振荡器的输出信号频率  $f_o$  与它的输入控制电压  $u_o$  成一定比例。

**反馈分频器:**分频器为环路提供一种反馈机制,反馈分频器将锁相环的输出信号  $f_o$  反馈给鉴相器,形成一个负反馈,从而使输入信号和输出信号之间的相位差保持恒定。当分频系数  $N=1$  时,锁相环系统的输出信号频率  $f_o$  等于输入信号频率  $f_r$ :

$$f_o = f_r$$

信号锁定后有:

$$f_o = f_i = f_r$$

当分频器的分频系数  $N > 1$ , 有:

$$f_o = N \cdot f_i \quad \text{即} \quad f_i = f_o / N$$

环路锁定后有:

$$f_i = f_r$$

$$f_o = N \cdot f_i = N \cdot f_r$$

若改变  $N$ , 则  $f_i \neq f_r$ , 环路失锁, 这时环路就进行频率捕捉和相位捕捉。经过一段时间后, 环路重新进入锁定状态, 频率合成器完成一个频率转换过程, 此时频率合成器输出为一个新的稳定频率。

当环路处于稳定状态时, 输出和输入之间存在一定量相位误差。而对于输入信号频率和输出信号频率而言, 二者却是成比例的, 这时环路处于锁定状态, 这是锁相环电路的一个特点。用这种方法可以得到非常精确的频率控制。而其他的频率控制方法, 在稳态时总是存在一定的频率误差。

### 11.1.2 CRG 模块框图

如图 11-2 所示, 时钟与复位产生模块 CRG 主要的引脚说明如下:

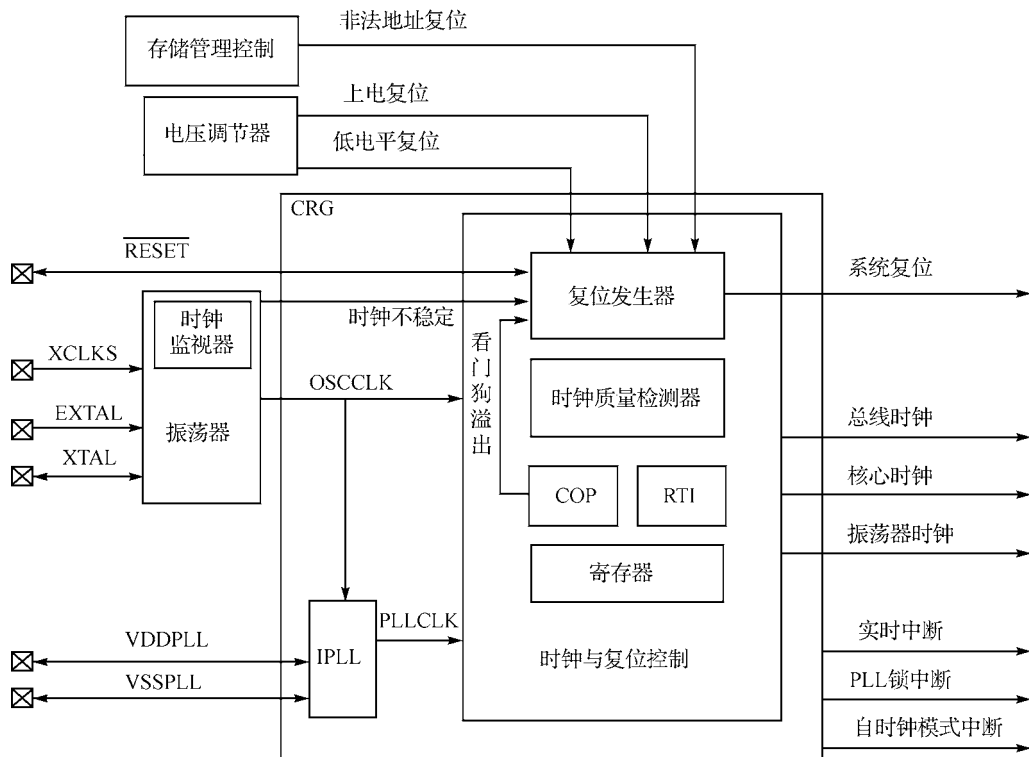


图 11-2 S12X 的 CRG 模块框图



① VDDPLL 和 VSSPLL:IPLL 内部过滤频率调整的 IPLL 时钟产生电路的电源和地引脚;

② EXTAL、XTAL、XCLKS:EXTAL 提供外部时钟输入,XTAL 是时钟输出。XCLKS 是晶振选择输入引脚(有内部上拉),它决定 EXTAL 与 XTAL 是接内部的科尔皮兹(Colpitts)晶振,还是接皮尔兹(Pierce)晶振或者外部时钟电路。在复位时,系统会探测 XCLKS 的电平信号,若 XCLKS 为高电平,则 EXTAL 与 XTAL 一起接皮尔斯振荡器或者外部时钟电路,若 XCLKS 是低电平,则 EXTAL 接内部的科尔皮兹振荡器,XTAL 不用。

③ RESET:复位引脚。当此引脚为低电平时,芯片复位,MCU 内部的程序重新开始运行。若是芯片内部自动复位,则会在此引脚输出低电平。

### 11.1.3 CRG 模块的工作模式

XS128 的 CRG 模块提供了 5 种工作模式以适应不同的情况,分别为运行模式、调试模式、等待模式、停止模式和自时钟模式。运行模式为实际应用系统的工作模式。调试模式主要用于硬件调试配置。等待模式与停止模式属于低功耗模式。运行模式与停止模式均通过软件进行设置。自时钟模式为安全工作模式。下面对各模式进行简要说明。

#### 1. 运行模式

在运行模式下,XS128 所有的功能模块都能够正常运行。若需要使用实时中断或者看门狗功能,可通过配置相应控制寄存器中溢出周期进行激活。

#### 2. 调试模式

调试模式用于程序写入与底层调试,是通过硬件设置的,即将 MCU 的 BKGD 引脚上的电位下拉至低电平。写入器使用此模式工作。通过写入器可以对 MCU 内部的 Flash 存储器进行擦除与写入(编程),只有 MCU 内被写入了用户程序之后,才能工作于运行模式。通常情况下,在出厂时,MCU 内的 Flash 存储器默认是空白的(被擦除过的)。

关于 S12 系列 MCU 的调试模式的功能与操作方法,用户可参阅相关技术手册详细了解。

#### 3. 低功耗模式:等待模式与停止模式

一般的嵌入式开发中,对功耗的要求并不严格,但在电池供电的嵌入式产品中或一些嵌入式产品的待机状态,对于系统的整体功耗有较高的要求。低功耗与 MCU 软件编程、外围器件选择、电路设计等方面密切相关,这里简要说明 S12 系列 MCU 软件编程时对低功耗模式的设置。

S12 系列 MCU 提供了两种低功耗模式供用户选择使用,分别对应指令 WAIT 和 STOP,相应的低功耗模式被称为等待模式和停止模式。

S12 系列 MCU 为静态 CMOS 工艺。CMOS 电路的特点是在静态时功耗极小,功耗主要

来自电平变化过程中的瞬态电流,所以时钟频率越高,系统功耗越大。降低功耗的方法之一是在不需要 MCU 工作的时候(例如,等待某个信号输入或中断请求),关闭系统时钟,同时关闭一些不必要的外设模块,使功耗达到最小,进入“睡眠”状态。而当有外部输入需要处理的时候,再让 MCU 脱离“睡眠”被“唤醒”,恢复时钟,继续运行指令。若外部事件不频繁,采用这种方法就可以大大降低系统功耗。

若要使用低功耗模式,在硬件设计时就需要注意,将未使用的 MCU 引脚接地,并在程序初始化时使这些引脚为输入。若让这些引脚悬空,则需要在程序初始化时,定义这些引脚为输出,且输出为逻辑 0。建议使用引脚不悬空的方式。

在软件设计上,一旦使用指令使 MCU 进入低功耗模式,则程序停止运行,只能等待某些中断的唤醒。等待模式与停止模式的功耗不同,唤醒条件也有差异。

### (1) 等待模式

若要使 MCU 在低功耗模式下运行,可以通过时钟选择寄存器(CLKSEL)来设置系统时钟、锁相环时钟、核心时钟、实时中断、看门狗等在等待模式下是否使能。若需要在等待模式禁止实时中断,可通过将 CLKSEL 的 RTIWAI 位置 1 来实现。类似地,若要禁止锁相环或看门狗功能,可将 PLLWAI 或 COPWAI 对应位置 1。

有 5 种方法可以让 MCU 离开等待模式:外部复位、时钟监控器复位、看门狗复位、实时中断、自时钟模式中断。

### (2) 停止模式

由 CLKSEL 寄存器中的 PSTP 位可以配置停止模式为完全停止模式(PSTP=0)或不完全停止模式(PSTP=1)。

在完全停止模式下,振荡器被停用后所有系统和核心时钟也停止工作,看门狗和实时中断也停止工作。

在不完全停止模式下,振荡器仍工作,但大部分系统与核心时钟停止工作。在此模式下,可选择配置 PCE 与 PRE 位的值以决定看门狗和实时中断继续运行或停止工作,若值为 1,则对应功能继续工作,值为 0,则停止工作。

## 4. 自时钟模式

自时钟模式是一个新的概念。锁相环电路内的压控振荡器 VCO 有一个最小工作频率  $f_{SCM}$ ,若外部时钟因晶振启动时间过长或者其他原因导致时钟不稳,则总线时钟(外频)和核心时钟(CPU 执行所用的时钟,主频)会自动产生  $f_{SCM}$ ,这种模式称为自时钟模式。该模式是在外部时钟电路不能正常工作时,为确保程序继续正常运行,而不致让程序终止的一种机制。

若要启用该模式,需要设置 CRG 模块的锁相环控制寄存器 PLLCTL 中的两位:CME=1、SCME=1。当外部时钟电路工作不正常时,将 CLKSEL 寄存器的 PLLSEL 位清 0,MCU 自动转入自时钟模式,在  $f_{SCM}$  时钟下继续正常工作;待外部时钟信号稳定后,系统会自动切换

OSCCLK 作为系统时钟,但此时 PLLSEL 位仍为 0。若要选用 PLLCLK 作为系统时钟,则须开放 PLL LOCK 中断,并在 PLL LOCK 中断服务程序中设置 CLKSEL 寄存器的 PLLSEL 位为 1,此时使用锁相环输出信号作为系统时钟源。

### 11.1.4 XS128 内部锁相环结构

锁相环结构如图 11-3 所示。

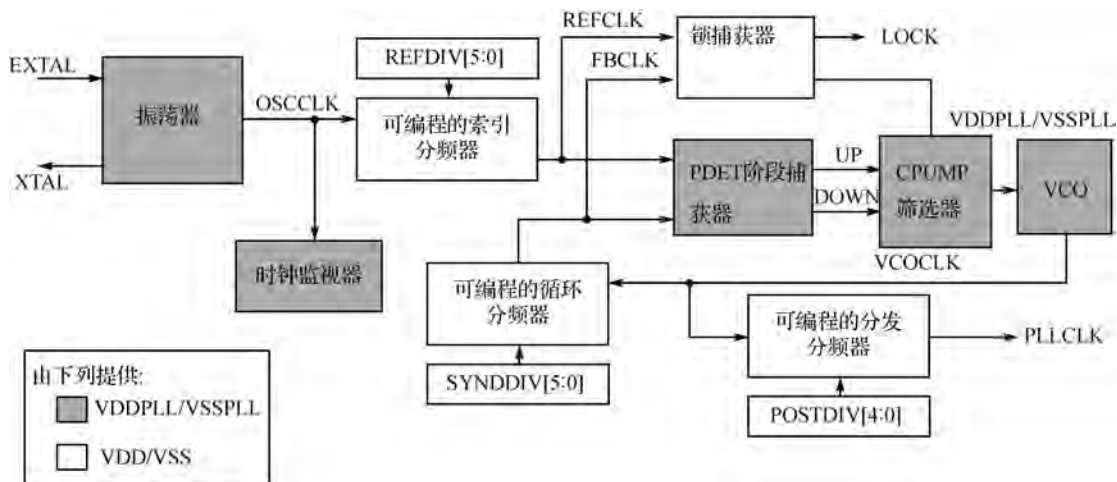


图 11-3 锁相环结构

#### (1) 内部锁相环的构成

##### 1) 振荡器模块

振荡器模块提供了连接外部晶振或谐振器的途径。可以通过程序来选择两种不同的时钟频率,使系统在启动及稳定时达到最佳状态。另外,振荡器模块可以对外提供系统时钟频率的方波。振荡器可被配置为低功耗模式或高增益模式。

##### 2) 锁频环模块

在锁频环阶段,利用内部或者外部的时钟源,通过乘(倍频)运算,可以得到更高的时钟频率。通过状态位查询电路处于锁定状态还是失锁状态。此外,这个模块可以监控外部参考时钟和信号来判断时钟是否有效。

#### (2) 内部锁相环的外部连接

EXTAL 和 XTAL 这两个引脚分别为外部晶体振荡器信号的输入与输出引脚。MCU 的内部系统时钟从 EXTAL 输入信号获得。在完全停止模式下(PSTP=0),EXTAL 引脚在内部被 200 kΩ 电阻拉向低电平。



## 11.2 XS128 的 CRG 模块的初始化

### 11.2.1 XS128 的 CRG 模块寄存器

#### 1) CRG 合成寄存器 SYNRR(Synthesizer Register)

该寄存器用于配置内部锁相环的乘积因子并选择 VCO 的频率范围。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	VCOFRQ1	VCOFRQ0	SYNDIV5	SYNDIV4	SYNDIV3	SYNDIV2	SYNDIV1	SYNDIV0
复位	0	0	0	0	0	0	0	0

VCOFRQ[1:0]用于重新设定 VCO 的值以获得安全稳定的时钟。为使内部锁相环正常工作,应根据实际的 VCO 频率(其具体分频值参照表 11-1)选择 VCOFRQ[1:0]的值。待锁相环稳定时,则根据公式 11-1 计算出 VCO 频率(其中 REFDIV 由 REFDV 寄存器给出):

$$f_{VCO} = 2 \times f_{OSC} \times (SYNDIV + 1) / (REFDIV + 1) \quad (\text{式 } 11-1)$$

表 11-1 VCO 时钟频率的选择

VCOCLK 频率范围	VCOFRQ[1:0]
$32 \text{ MHz} \leq f_{VCO} \leq 48 \text{ MHz}$	00
$48 \text{ MHz} < f_{VCO} \leq 80 \text{ MHz}$	01
保留	10
$80 \text{ MHz} < f_{VCO} \leq 120 \text{ MHz}$	11

**注意:**  $f_{VCO}$  必须在规定的 VCO 锁频率范围内;总线频率不能超过规定的最大值。

#### 2) 后分频寄存器 POSTDIV(Post Divider Register)

该寄存器用于配置 VCOCLK 和 PLLCLK 之间的频率比例,如式 11-2 所示。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	无	无	无	POSTDIV4	POSTDIV3	POSTDIV2	POSTDIV1	POSTDIV0
复位	0	0	0	0	0	0	0	0

$$f_{PLL} = f_{VCO} / (2 \times \text{POSTDIV}) \quad (\text{式 } 11-2)$$

**注意:** 若 POSTDIV=0x00,则  $f_{PLL}$  的值与  $f_{VCO}$  相同。

#### 3) CRG 参考分频寄存器 REFDV(Reference Divider Register)

该寄存器设定了内部锁相环电路倍乘步长的粒度。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	REFFRQ1	REFFRQ0	REFDIV5	REFDIV4	REFDIV3	REFDIV2	REFDIV1	REFDIV0
复位	0	0	0	0	0	0	0	0

REFFRQ[1 : 0]用于设置内部锁相环稳定工作的频率范围。为了使内部锁相环电路正常工作,应根据实际的 REFCLK 频率(具体分频值参照表 11-2)选择正确的 REFFRQ[1 : 0]值。

表 11-2 参照时钟频率的选择

REFCLK 频率范围	REFFRQ[1 : 0]
$1\text{ MHz} \leq f_{\text{REF}} \leq 2\text{ MHz}$	00
$2\text{ MHz} < f_{\text{REF}} \leq 6\text{ MHz}$	01
$6\text{ MHz} < f_{\text{REF}} \leq 12\text{ MHz}$	10
$f_{\text{REF}} > 12\text{ MHz}$	11

REFDV 寄存器确定参考频率  $f_{\text{REF}}$ ,计算公式如式 11-3 所示:

$$f_{\text{REF}} = f_{\text{OSC}} / (\text{REFDIV} + 1) \quad (\text{式 } 11-3)$$

通过时钟选择寄存器 CLKSEL 的 PLLSEL 位选择系统时钟源是锁相环输出时钟 PLL-CLK 还是晶振时钟 OSCCLK。只有在寄存器 CLKSEL 的位 PLLSEL=0 的情况下才能设置上述 3 个寄存器。

#### 4) CRG 标志寄存器 CRGFLG(Flags Register)

该寄存器包含 XS128CRG 模块的状态位和标志位。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RTIF	PORF	LVRF	LOCKIF	LOCK	ILAF	SCMIF	SCM
复位	0	1	0	0	0	0	0	0

D7—RTIF 位,实时中断标志。每个实时中断周期结束时,该标志位被置为 1。写 1 清除该标志位,写 0 不影响该位。若 RTIE=1,该位被置为 1 时,将会产生中断请求。

D6—PORF 位,上电复位标志。当上电复位发生时,该位被置为 1。写 1 清除该标志位,写 0 无效。

D5—LVRF 位,低电压复位标志。当低电压复位发生时,该位被置为 1。写 1 清除该标志位,写 0 不影响该标志位。若 LVRF=1,LVRF 会产生重置请求。

D4—LOCKIF 位,锁中断标志。LOCK 状态位变化时,该标志位被置为 1,写 1 清除该标志位。LOCK 用于反映外部时钟信号的稳定性,LOCK 标志为 1 时,表示输入的外部时钟信号稳定。若 LOCKIE=1,当该位被置为 1 时,将产生中断请求。



D3—LOCK 位,锁状态位。当 VCOCLK 输出的频率在期望频率的允许范围内时,该位被置为 1,此时 PLLCLK 已经稳定,可以作为系统时钟源。该位只读,但在自时钟模式时,该位能够被清 0。

D2—ILAF 位,非法地址定位标志。当一个非法地址重定位事件发生时,该位被置为 1 (详情参见芯片手册)。写 1 清除该标志位,写 0 不影响该标志位。当 ILAF=1,将会产生重置请求。

D1—SCMIF 位,自时钟模式中断标志位。SCM 位变化的时候,该标志位置 1。写 1 清除该标志位,写 0 不影响该位。若 SCME=1,当该位为 1 时,将产生一个中断请求。

D0—SCM 位,自时钟模式状态位,只读。SCM 反映当前的时钟模式,当 OSCCLK 不稳定时,该位为 1,系统会进入自时钟模式,此时 PLLCLK 以  $f_{SCM}$  频率工作。

### 5) CRG 时钟选择寄存器 CLKSEL (Clock Select Register)

该寄存器用于控制 CRG 时钟的选择。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	PLLSEL	PSTP	XCLKS	无	PLLWAI	无	RTIWAI	COPWAI
复位	0	0	0	0	0	0	0	0

D7—PLLSEL 位,锁相环选择位。当 CRG 标志寄存器 CRGFLG 的 LOCK=0 时向该寄存器写 1 无效,这样可防止选择不稳定的 PLLCLK 为系统时钟源。当 PLLWAI 位置为 1 且 MCU 进入自时钟模式、停止模式或等待模式时,PLLSEL 位将被清 0。它被用于回读 PLLSEL 位以确定 PLLCLK 是否真的已经被选为系统时钟源,因此理论上在写 PLLSEL 位时 LOCK 状态位能改变。PLLSEL=1,选择 PLLCLK 为系统时钟源。PLLSEL=0,选择 OSCCLK 为系统时钟源。

D6—PSTP 位,伪停止位,该位控制在停止模式下晶振源的功能。PSTP=1,在停止模式下晶振继续运行。PSTP=0,在停止模式下晶振不工作。

D5—XCLKS 位,振荡器配置状态位,当 XCLKS=1 时,外部晶振时钟被选择。XCLKS=0 时,回路控制选择皮尔斯晶振。

D3—PLLWAI 位,等待模式 PLL 停止位,当 PLLWAI=1 时,在进入等待模式之前,CRG 将清 PLLSEL 位。在等待模式中,寄存器 PLLCTL 的 PLLON 位保持置 1 的状态并且 IPLL 停止工作以降低功耗。在离开等待模式以后,可以设置 PLLSEL 位,以继续使用 IPLL 时钟。PLLWAI=0,在等待模式下 IPLL 仍然工作。但当 PLLWAI 置 1,IPLL 将在等待模式下停止。

D1—RTIWAI 位,等待模式 RTI 停止位,当 RTIWAI=1,只要进入等待模式,RTI 就停止并将 RTI 的分频器清 0;RTIWAI=0,在等待模式下 RTI 仍可工作。

D0—COPWAI 位,等待模式 COP 停止位,COPWAI=1,只要进入等待模式,COP 就停止并将 COP 的分频器清 0;COPWAI=0,在等待模式下 COP 仍可工作。

6) IPLL 控制寄存器 PLLCTL(IPLL Control Register)

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	CME	PLLON	FM1	FM0	FSTWKP	PRE	PCE	SCME
复位	1	1	0	0	0	0	0	1

CME—时钟监控器使能位。时钟监控器在检测到时钟信号不稳定时会复位 MCU 或进入自时钟模式。CME=1,使能时钟监控器。CME=0,禁止时钟监控器。

PLLON—IPLL 启动位。PLLON=1,启动 IPLL 电路。PLLON=0,关闭 IPLL 电路。

FM[1 : 0]—IPLL 频率调整使能位。能够在 VCOCLK 上设置额外的调整频率用来降低噪音。调制频率是  $f_{ref}/16$ 。FM 振幅的变化范围参照表 11-3 译码。

表 11-3 FM 振幅选择

FM1	FM0	FM 振幅/ $f_{VCO}$ 变化范围
0	0	FM off
0	1	±1%
1	0	±2%
1	1	±4%

FSTWKP—完全停止快速唤醒位。FSTWKP 使得从完全停止模式下快速唤醒。若自时钟模式禁止(SCME=0),则该位不受影响。FSTWKP=0,从完全停止模式快速唤醒禁止。FSTWKP=1,从完全停止模式快速唤醒使能。当从完全停止模式中被唤醒时系统立即在自时钟模式下重新运行,SCMIF 标志将不被置位。系统将保持在自时钟模式,在 FSTWKP 位被清 0 之前,振荡器和时钟监视器都不可用。若时钟质量检测成功,MCU 会将所有的系统时钟转换为振荡器时钟。SCMIF 标志将被置 1。

PRE—在不完全停止模式下使能实时中断。PRE=1,在不完全停止模式下实时中断仍然运行。PRE=0,在不完全停止模式下实时中断停止运行。

PCE—在不完全停止模式下使能看门狗。PCE=1,在不完全停止模式下看门狗仍然运行。PCE=0,在不完全停止模式下看门狗停止运行。

SCME—自时钟模式使能位。SCME 只能在自时钟模式下才能被清除。SCME=1,检测到晶振时钟失效后将强制 MCU 进入自时钟模式。SCME=0,晶振时钟失效将导致时钟监控器复位。





### 7) 中断使能寄存器 CRGINT(Interrupt Enable Register)

该寄存器用于使能 XS128CRG 的一些中断功能。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RTIE	无	无	LOCKIE	无	无	SCMIE	无
复位	0	0	0	0	0	0	0	0

RTIE—实时中断使能位。RTIE=1,在任何时刻 RTIF 被置位,将会产生中断请求。  
RTIE=0,禁止实时中断。

LOCKIE—锁定中断使能位。LOCKIE=1,在任何时刻 LOCKIF 被置位,将会产生中断请求。  
LOCKIE=0,禁止 LOCK 中断。

SCMIE—自时钟模式中中断的使能位。SCMIE=1,在任何时刻 SCMIF 被置位,将会产生中断请求。  
LOCKIE=0,禁止 SCM 中断。

### 8) 实时中断控制寄存器 RTICTL(RTI Control Register)

RTICTL 为实时中断选择溢出周期。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	RTDEC	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
复位	0	0	0	0	0	0	0	0

RTDEC—十进制或二进制分频选择位。该位根据预分频的值选择十进制或二进制。  
RTDEC=0,二进制基数分频值。RTDEC=1,十进制基数分频值。

RTR[6 : 4]—实时中断比率选择位。这些位用于为 RTI 选择预分频的比率。

RTR[3 : 0]—实时中断模数计算选择位。分频系数选择位。

该寄存器选择实时中断的溢出时钟周期。计算公式如下：

$$\text{溢出时钟周期} = (\text{RTR}[3 : 0] + 1) \times 2^{(\text{RTR}[6 : 4] + 9)} / \text{OSCCLK} \quad (\text{式 } 11 - 4)$$

RTI 分频值计算公式如下：

RTDEC=0 时分频值：

$$\text{分频值} = (\text{RTR}[3 : 0] + 1) \times 2^{(\text{RTR}[6 : 4] + 9)} \quad (\text{式 } 11 - 5)$$

**注意：**当 RTR[6 : 4]=000 时,禁止实时时钟。

RTDEC=1 时分频值:RTR[6 : 4]=000、001、010、100、101、110、111 时,对应的数据入口分频值( $\times 10^3$ )分别为:1、2、5、10、20、50、100、200。公式如下：

$$\text{分频值} = (\text{RTR}[3 : 0] + 1) \times \text{RTR}[6 : 4] \quad (\text{式 } 11 - 6)$$



## 9) 看门狗控制寄存器 COPCTL(COP Control Register)

此寄存器用于设置看门狗功能。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	WCOP	RSBCK	WRTMASK	无	无	CR2	CR1	CR0
复位	0	0	0	0	0	0	0	0

看门狗模块用于监测程序的正常运行。启动看门狗后,必须在看门狗复位之前向寄存器 ARM COP 中依次写入 0x55、0xAA。这样看门狗计数器就会重新开始。

WCOP—窗口 COP 模式使能位。置 1 后,必须在看门狗周期的后 25%时间内向 ARM-COP 中依次写入 0x55、0xAA。在其他时间内写入或者写入其他值无效,会让 MCU 复位。WCOP=1,窗口 COP 模式。WCOP=0,正常 COP 模式。

RSBCK—在 BDM 模式下,COP 和 RTI 停止位。RSBCK=1,只要进入 BDM 模式,就停止 COP 和 RTI 计数。RSBCK=0,在 BDM 模式下允许 COP 和 RTI 运行。

WRTMASK—WCOP 和 CR[2:0]写掩蔽位。当对 COPCTL 寄存器进行写操作时,此只写位为 WCOP 和 CR[2:0]的掩蔽位。WRTMASK=0 时,对 WCOP 和 CR[2:0]的写操作有效。WRTMASK=1 时,对 WCOP 和 CR[2:0]的写操作无效。

CR[2:0]—COP 计数溢出速率选择位。COP 溢出时钟周期 = OSCCLK/CR[2:0]。CR[2:0]对应的分频值为:CR[2:0]=000 为禁用 COP、其他 CR[2:0]=001、010、011、100、101、110、111,对应溢出速率值为  $2^{14}$ 、 $2^{16}$ 、 $2^{18}$ 、 $2^{20}$ 、 $2^{22}$ 、 $2^{23}$ 、 $2^{24}$ 。

## 10) 看门狗定时复位寄存器 ARM COP(COP Timer Arm/Reset Register)

当禁止看门狗功能时(CR[2:0]=000),该寄存器不启用;当 COP 被激活时(CR[2:0]不为 000),其应用如下:写入值非 0x55 或 0xAA 时会引起 COP 复位。若要重新启动 COP 的溢出周期,必须在写入 0xAA 后加上 0x55。其他的操作指令可以在 0x55、0xAA 依次写入后,并在溢出周期的 COP 结束之前来避免 COP 的重置时被执行。0x55 与 0xAA 的写操作都是被允许的。当 WCOP 位被置 1 时,必须在选定的溢出周期的后 25%时间内向 ARM COP 中依次写入 0x55、0xAA,在其他时间内写入或者写入其他值无效,并会使 COP 复位。

## 11.2.2 初始化编程方法与实例

### 1. 初始化参数计算方法

MCU 上电复位后,应先执行初始化程序,为之后 MCU 能够正确运行配置所需的运行环境参数。系统初始化的主要内容是设置 MCU 内部总线的工作频率以及是否激活 IRQ、COP 功能。

CRG 模块的锁相环电路由 OSCCLK 时钟产生一个固定的频率时钟 PLLCLK,为系统提

供各种时钟频率源。

OSCCLK 经过参考分频电路分频得到参考信号  $f_{\text{REF}}$ ,  $f_{\text{REF}}$  作为基准频率源输给鉴相电路, 进入锁相环。反馈信号  $f_{\text{FB}}$  是压控振荡器输出信号  $f_{\text{VCO}}$  经过时钟合成电路的输出信号, 该信号  $f_{\text{FB}}$  与信号  $f_{\text{REF}}$  一起输给鉴相电路, 进行相位比较。当锁相环稳定,  $f_{\text{FB}} = f_{\text{REF}}$ , 信号  $f_{\text{VCO}}$  经过后分频电路输出到锁相环电路, 得到信号  $f_{\text{PLL}}$  时输给 MCU 的内核形成系统时钟  $f_{\text{SYS}}$ , 为总线形成总线时钟  $f_{\text{BUS}}$ 。各电路输入输出信号的关系如下:

参考信号  $f_{\text{REF}}$  与振荡器信号  $f_{\text{OSC}}$  关系见式 11-3。

反馈信号  $f_{\text{FB}}$  与压控振荡器输出信号  $f_{\text{VCO}}$  关系, 如下:

$$f_{\text{FB}} = f_{\text{VCO}} / [2 \times (\text{SYNDIV} + 1)] \quad (\text{式 } 11-7)$$

压控振荡器输出信号  $f_{\text{VCO}}$  与预分频电路输出信号  $f_{\text{PLL}}$  关系见式 11-2。

VCOCLK 和 PLLCLK 之间的频率比例一般为 1, 即  $f_{\text{VCO}} = f_{\text{PLL}}$ , 所以 POSTDIV 的值一般被设定为 0。

如果锁相环不稳定, 即不满足  $\text{LOCK} = 1$  的条件, 则  $f_{\text{FB}} \neq f_{\text{REF}}$ , 锁相环继续自动修正相位差, 直到  $f_{\text{FB}} = f_{\text{REF}}$  成立, 由此得出表达式 11-1 的关系, 并且可以变形得:

$$2 \times (\text{SYNDIV} + 1) / (\text{REFDIV} + 1) = f_{\text{VCO}} / f_{\text{OSC}} = f_{\text{PLL}} / f_{\text{OSC}} = 2 \times f_{\text{BUS}} / f_{\text{OSC}}$$

根据时钟稳定的需要, 建议 REFDIV 尽可能取最大值。  $f_{\text{BUS}}$  和  $f_{\text{OSC}}$  分别为已知的目标值, 则 REFDIV、SYNDIV 和 POSTDIV 由此可以确定。根据  $f_{\text{VCO}}$  和  $f_{\text{REF}}$  取值范围, 由表 11-1 和 11-2 分别确定稳定系数 VCOFRQ[1:0] 和 REFFRQ[1:0]。至此 SYNRR、POSTDIV、REFDV 这 3 个寄存器的值可以确定。

最后选择 PLLCLK 作为系统时钟源, 即 CLKSEL 寄存器的 PLLSEL = 1, 所以  $f_{\text{BUS}} = f_{\text{PLL}} / 2$ 。

## 2. 初始化计算示例

在 MCU 系统初始化时, 假设外部晶振频率  $f_{\text{OSC}}$  为 16 MHz, 欲设置总线频率  $f_{\text{BUS}}$  达到 32 MHz, 系统时钟为 64 MHz (= 总线频率的 2 倍), 可通过启用时钟发生器将系统时钟提高至原来的 4 倍即可。故由式 11-1 和式 11-2 (其中 POSTDIV 一般为 0) 得:

$$2 \times (\text{SYNDIV} + 1) / (\text{REFDIV} + 1) = f_{\text{PLL}} / f_{\text{OSC}} = 64 / 16 = 4 \quad (\text{式 } 11-8)$$

选择参考分频因子 REFDIV 时, 为了锁相环稳定, 选择尽可能大的 REFDIV, 由表 11-2 可知选择锁频环的参考频率最小范围为  $1 \text{ MHz} \leq f_{\text{REF}} \leq 2 \text{ MHz}$ 。若选择  $f_{\text{REF}} = 2 \text{ MHz}$ , REFFRQ[1:0] = 0, 由表达式 11-3, 得 REFDIV = 7, 故有 REFDV = 0b00000111。

由表达式 11-8, 得 SYNDIV = 15。

$f_{\text{VCO}} = f_{\text{PLL}} = 64 \text{ MHz}$ , 查表 11-1 得 SYNRR 寄存器的 VCOFRQ[1:0] = 01, 故有 SYNRR = 0b01001111。

## 3. 初始化编程步骤

在初始化程序中需要进行设置系统时钟、选择激活 COP 与 RTI。初始化步骤为:

① 禁止中断。

② 设置 CLKSEL 寄存器。在锁相环电路没有产生稳定的输出频率前暂时不能启用,故应先将 CLKSEL 的 PLLSEL 位清 0,选择系统时钟源为 OSCCLK,则在 IPLL 程序稳定运行前,采用内部总线频率 = OSCCLK/2。不对等待模式和停止模式进行设置,所以,CLKSEL=0x00。

③ 禁止 IPLL。

④ 通过 PLLCTL 寄存器启动 IPLL 电路,其余位未涉及,故均为默认值。

⑤ 设置 IPLL 参数,根据需要的时钟频率设置 SYNR 和 REFDV 寄存器。

⑥ 启动 IPLL 运行。

⑦ 通过判断 CRGFLG 寄存器的 LOCK 位,确保 IPLL 稳定工作。

⑧ 时钟频率稳定后,允许 IPLL 时钟源作为系统时钟源。

⑨ 设置是否允许 IRQ 中断、是否允许看门狗。

## 4. 初始化编程实例

初始化文件 MCUInit.c 的代码如下:

```
// 头文件包含
#include "MCUInit.h"
// ----- *
//函数名: Light_Init 时钟设置函数定义 *
//功 能: 基于 16MHz 的外部晶振,设置总线时钟频率 *
//参 数: fbus 为目标总线频率值,推荐值为 16、20、32、40、48、60MHz *
//返 回: 无 *
// ----- *

void SetBusCLK_M(int fbus)
{
    CLKSEL = 0x00;          // 不加载 IPLL 到系统
    REFDV = 0x00 | 0x07;    // REFFRQ[1:0];REFDIV[5:0]
    switch(fbus)             // BUS CLOCK
    {
        case 16: SYNR = 0x00 | 0x07; break;
        case 20: SYNR = 0x00 | 0x09; break;
        case 32: SYNR = 0x40 | 0x0F; break;
        case 40: SYNR = 0x40 | 0x13; break;
        case 48: SYNR = 0xC0 | 0x17; break;
        case 60: SYNR = 0xC0 | 0x1D; break;
        default: break;
    }
}
```



```
    PLLCTL_PLLON = 1;          // 启动 IPLL
    _asm(nop);
    _asm(nop);
    while(! (CRGFLG_LOCK == 1)); //等待 IPLL 稳定
    POSTDIV = 0x00;             //POSTDIV[4:0], fPLL = fVCO/(2xPOSTDIV)
                                //若 POSTDIV = 0x00,则 fPLL = fVCO
    CLKSEL_PLLSEL = 1;          //加载 IPLL 到系统
}

//芯片的初始化
void MCUInit(int fbus)
{
    SetBusCLK_M(fbus);          //设置总线频率
    IRQCR_IRQEN = 0;            //禁止 IRQ 中断(默认开)
    COPCTL = 0x00;              //COPCTL[2:0] = 000 禁止看门狗
}
```

## 11.3 CRG 模块的其他功能

### 11.3.1 CRG 产生复位信号

所有复位源如表 11-4 所列。

#### 1. CRG 产生复位的说明

CRG 可以产生复位信号对 MCU 进行复位。有 4 种操作将引起复位:RESET 引脚低电平复位、上电复位、看门狗复位和时钟监控器复位。

表 11-4 复位源汇总

复位源	使能设置
上电复位	无
低电压复位	无
外部复位	无
非法地址复位	无
时钟监控器复位	PLLCTL(CME=1,SCME=0)
看门狗复位	COPCTL(CR[2:0]为非零值)

检测到复位事件之后,内部电路将保持 $\overline{\text{RESET}}$ 引脚上的电平为低,持续时间为 128 个 SYSCLK(详见图 11-6),之后 $\overline{\text{RESET}}$ 引脚恢复高电平。复位产生模块会等待接下来的 64 个 SYSCLK,然后再次采样 $\overline{\text{RESET}}$ 引脚以判断复位源。表 11-5 说明了复位向量的选取。

表 11-5 向量选取复位

引脚实例(释放后的 64 位循环)	待定时钟监控器复位	待定看门狗(COP)复位	向量获取
1	0	0	POR/LVR/非法地址复位/外部复位
1	1	X	时钟监控器复位
1	0	1	看门狗(COP)复位
0	X	X	POR/LVR/非法地址复位/复位引脚确定的外部复位

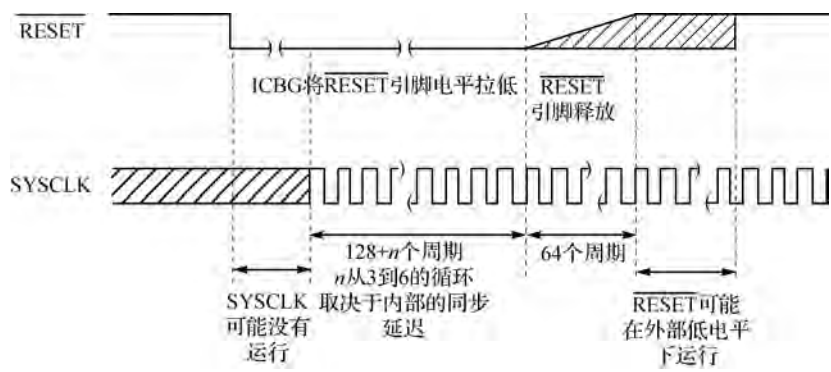


图 11-4 复位时序图

## 2. 与复位相关的寄存器说明

### (1) 控制寄存器 VREGCTRL

该寄存器允许配置 VREG\_3V3 低电平检测功能。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	无	无	无	无	无	LDVS	LVIE	LVIF
复位	0	0	0	0	0	0	0	0

D2—LDVS 位,低电平检测状态位。此只读状态位表示输入为低电压,写入无效。LDVS=0 时,表示输入电压  $V_{\text{DDA}}$  高于  $V_{\text{LVID}}$  或 RPM,或者处于关断模式。LDVS=1 时,表示输入电压  $V_{\text{DDA}}$  低于  $V_{\text{LVID}}$  和 FPM。

D1—LVIE 位,低电平中断使能位。LVIE=0,禁止低电平中断请求。LVIE=1,允许产



生低电平中断。

D0—LVIF 位,低电平中断标志。当发生低电平事件时,该位被设置为 1。写 1 清除该标志位,写 0 无效。若允许低电平中断(LVIE=1),则当该位为 1 时,同时产生中断请求。LVIF=0, LVDS 位不发生改变。LVIF=1, LVDS 发生改变。

**注意:**在进入低功耗模式时, LVIF 不被 VREG\_3V3 清 0。

## (2) 自治周期中断控制寄存器 VREGAPICL

该寄存器允许配置 VREG\_3V3 自治周期中断功能。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	APICLK	无	无	APIES	APIEA	APIFE	APIE	APIF
复位	0	0	0	0	0	0	0	0

D7—APICLK 位,自治周期中断时钟选择位,为自治周期中断选择时钟源。当 APIFE=0 时,可进行写入;若以相同的值执行写操作设置 APIFE,则 APICLK 不发生改变。APICLK=0,自治周期中断时钟被用作时钟源。APICLK=1,总线时钟作为时钟源。

D4—APIES 位,自治周期中断外部选择位。选择外部引脚上的信号波形,若被置 1,则在外部引脚的时钟信号是所选 API 周期的 2 倍(见表 11-6)。若未被设置,在外部引脚上,将会在每一个被选定的最小周期中间结束有一个高脉冲(见表 11-6)。APIES=0,若 APIEA 和 APIFE 被设置,在外部引脚上可以检测到周期高脉冲。APIES=1,若 APIEA 和 APIFE 被设置,在外部引脚的检测到的是时钟。

D3—APIEA 位,自治周期中断外部访问使能位。若被设置,由 APIES 选择的波形可以从外部引脚被访问。APIEA=0,由 APIES 选择的波形禁止从外部引脚被访问。APIEA=1,若 APIES 被设置,由 APIES 选择的波形可以从外部引脚被访问。

D2—APIFE 位,自治周期中断功能使能位。当此位被设置,API 功能使能,API 定时器打开。APIFE=0,自治周期中断关闭。APIFE=1,自治周期中断使能,定时器开始工作。

D1—APIE 位,自治周期中断使能位。APIE=0,API 中断请求被禁止。APIE=1,当 APIF 被设置时,API 中断将会被请求。

D0—APIF 位,自治周期中断标志—APIF 被设置为 1 当 API 配置时间结束。向该位写入 1 可清空该标志。清空该标志优先于设置。向该位写 0 没有影响。如果使能(APIE=1), APIF 引发中断请求。APIF=0,API 未发生超时。APIF=1,API 发生超时。

## (3) 自治周期中断频率高、低字节寄存器 VREGAPIRH/VREGAPIRL

高位 VREGAPIRH 和低位 VREGAPIRL 寄存器允许配置 VREG\_3V3 自治周期中断率。高位 VREGAPIRH 寄存器对应于 APIR[15:8],低位 VREGAPIRL 寄存器对应于 APIR[7:0]。APIR[15:0](自治周期中断率位)这些位定义了 API 的溢出周期。只有在 VREGAPICL 寄存器的 APIFE=0 时是可写的。自治周期可以根据 APICLK 选用不同的公

式进行计算：

$$\text{自治周期} = 2 \times (\text{APIR}[15:0] + 1) \times 0.1 \text{ ms} \text{ 或 } 2 \times (\text{APIR}[15:0] + 1) \times \text{总线时钟周期}$$

表 11-6 可选的自治周期

APICLK	APIR[15:0]	选定的周期	APICLK	APIR[15:0]	选定的周期
0	0000	0.2 ms	1	0000	2×总线时钟周期
0	0001	0.4 ms	1	0001	4×总线时钟周期
0	0002	0.6 ms	1	0002	6×总线时钟周期
0	0003	0.8 ms	1	0003	8×总线时钟周期
0	0004	1.0 ms	1	0004	10×总线时钟周期
0	0005	1.2 ms	1	0005	12×总线时钟周期
0	.....	.....	1	.....	.....
0	FFFD	13 106.8 ms	1	FFFD	131 068×总线时钟周期
0	FFFE	13 107.0 ms	1	FFFE	131 070×总线时钟周期
0	FFFF	13 107.2 ms	1	FFFF	131 072×总线时钟周期

### 3. 3 种复位

#### (1) 时钟监控复位

当同时满足下列情况时,将会产生时钟监控复位:

- 时钟监控使能(CME=1);
- 自时钟模式禁止(SCME=0);
- 检测到时钟不稳。

复位后,CME 和 SCME 均被初始化为 1,此时,系统会立刻进入自时钟模式。若检测到可用的 OSCCLK 时钟,则 MCU 将离开自时钟模式并使用 OSCCLK 作为系统时钟。

#### (2) 看门狗(COP)复位

计算机正常运行(Computer Operating Properly, COP)看门狗的功能是确保 MCU 中运行的程序为指定的程序内容,若 MCU 的脱离了指定程序(跑偏),则看门狗模块将使 MCU 自动复位。这里将介绍看门狗功能的具体工作过程。

在 MCU 初始化时,可以选择启动一个看门狗计数器模块,并设定此计数器的溢出周期。启动后,看门狗模块的计数器开始计数。如果要保证 MCU 的主程序顺利运行,必须在看门狗计数器溢出之前将其复位,即进行“喂狗”操作,否则,当看门狗计数器溢出之后,模块将通知 MCU 复位,重新开始执行主程序。为了保证主程序能够持续顺利地执行,在主程序中,必须定期进行“喂狗”,相当于主程序定期告诉 MCU:“我还在正确运行,先不要重新启动”。

需要注意的是,如果主程序中某个函数的执行时间比较长,也需要在函数中插入一些“喂



狗”操作。看门狗计数器的溢出周期可以参考芯片手册中相关表格进行设定;若设定溢出周期较短,则看门狗较灵敏,但过多的“喂狗”操作会影响 MCU 主程序的执行效率;若设定溢出周期较长,则看门狗反应比较慢,但会减小对 MCU 主程序执行效率的影响。

另外,清除 COP 计数器的写系统复位状态寄存器(SRS)操作不应放置在中断服务程序中,因为尽管主程序已经出现了问题,但在中断中仍可能会定时执行“喂狗”操作,从而使看门狗守护的功能失去意义。

仅当 MCU 工作在 BDM 模式时,看门狗计数器会默认停用。一旦脱离 BDM 模式,看门狗计数器将继续递增计数。

在特定情况下,过多地复位会影响到程序运行的连贯性,例如,在不稳定的条件下,程序经常跑偏,会导致 MCU 还未执行完一次任务就不得不复位。在这种情况下,需要完善整体设计,而不仅仅是依赖看门狗功能进行守护。

### (3) 上电复位和低电压复位

MCU 的 VDD 电平到达一个设定的电平值,片上电压调节器将会检测到此状态,并启动上电复位或低电平复位。当触发了一个上电复位或低电平复位,CRG 将执行输入时钟信号质量检测。当时钟质量检测功能指示为有效的振荡器时钟信号,则复位程序开始使用振荡器时钟。若在 50 次检测之后,时钟质量检测功能仍指示为无效振荡时钟,则复位程序将进入自时钟模式。

当复位引脚被锁定到 VDD 电平和复位引脚被持续拉低时的上电流程,如图 11-5 和图 11-6 所示。

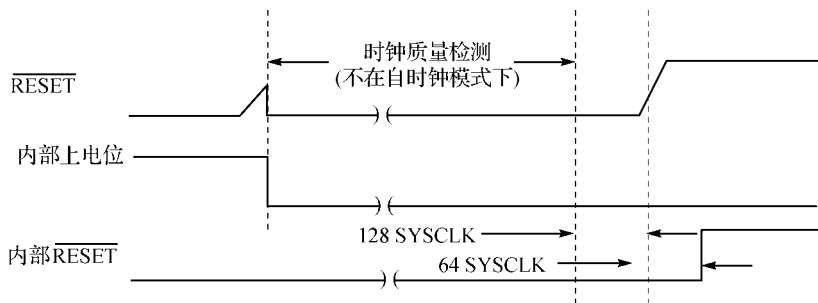


图 11-5 内部电压(VDD)的复位引脚(通过拉动电阻)



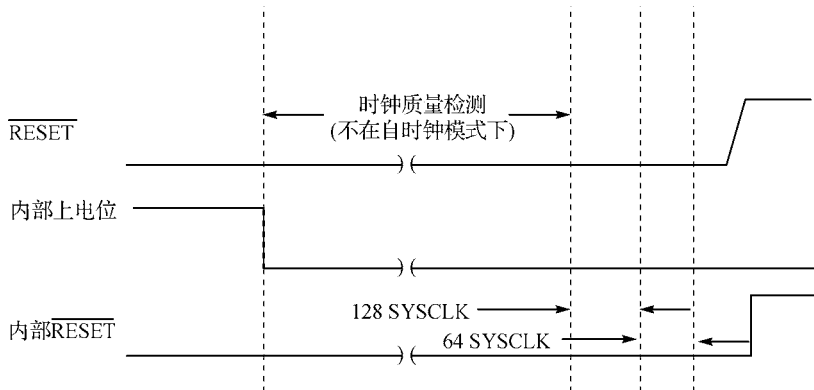


图 11-6 复位引脚锁住外部低电压当

## 11.3.2 中 断

在 11.2.1 小节介绍 CRG 中断使能寄存器时曾说明了 CRG 模块支持的中断,此模块支持的中断有 3 种:实时中断、内部锁相环锁定中断和自时钟模式中断。这 3 种中断均可在 CRG 中断使能寄存器(CRGINT)中使能或禁止,如表 11-7 所列。

表 11-7 S12XECRG 中断向量

中断源	CCR 掩码	使能控制
实时中断	1 字节	CRGINT(RTIE)
锁中断	1 字节	CRGINT(LOCKIE)
自时钟模式中断	1 字节	CRGINT(SCMIE)

### (1) 实时中断

实时中断在默认情况下是关闭的,当设置了实时中断寄存器(RTICtrl)并在 CRGINT 寄存器中将 RTIE 位置 1 即可开启此中断。在实时中断每个周期结束时就会产生中断请求。

### (2) 内部锁相环锁定中断

当 CRG 标志寄存器的 LOCK 位变化时将产生中断,这表示外部时钟从稳定变为不稳定或从不稳定变为稳定。

### (3) 自时钟模式中断

当系统的自时钟模式状态发生变化时,包括进入或者离开自时钟模式,MCU 将会产生自时钟模式中断。自时钟模式是由上电复位(POR)、低电压复位(LVR)、从完全停止状态恢复(PSTP=0)或时钟监视器失效而导致时钟检查失效等的状态。若时钟监视器处于激活状态(CME=1),外部时钟源的失效错误也同样会触发自时钟模式(SCME=1)状态。

禁止自时钟模式中断是通过设置 SCMI $\overline{E}$  位为 0 而实现的。当自时钟模式状态改变时, 自时钟模式中断标记位(SCMIF)为 1, 向 SCMI $\overline{F}$  位写入 1 可将该位清 0。

## 11.4 XS128 的 $\overline{IRQ}$ 、 $\overline{XIRQ}$ 引脚、RTI、BRK 及 SWI 中断

除了前面章节介绍的各个中断, XS128 还有 $\overline{IRQ}$ 与 $\overline{XIRQ}$ 、RTI、BRK 及 SWI 等中断资源可供开发者使用。

### 11.4.1 $\overline{IRQ}$ 与 $\overline{XIRQ}$ 引脚中断

$\overline{IRQ}$ 与 $\overline{XIRQ}$ 引脚是 XS128 唯一的两个仅用作中断的引脚, 内部具有上拉电阻。 $\overline{IRQ}$ 是可屏蔽的电平或下降沿触发的中断引脚, 而 $\overline{XIRQ}$ 是不可屏蔽的电平触发的中断引脚, 通过 CPU 的 CCR 条件代码寄存器 X-bit 清零, 以使能 $\overline{XIRQ}$ 。这两个引脚分别复用 E 口 1 号和 0 号引脚。

IRQ 控制寄存器 IRQCR(IRQ Control Register)的地址是 0x001E, 定义为:

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	IRQE	IRQEN	无	无	无	无	无	无
复位	0	1	0	0	0	0	0	0

D7—IRQE 位, IRQ 触发模式位。D7=0, 低电平触发 IRQ 事件; D7=1, 下降沿触发 IRQ 事件, 在 IRQ 中断服务程序或复位清除该位。

D6—IRQEN 位: IRQ 中断使能位。D6=0, 允许 IRQ 中断; D6=1, 禁止 IRQ 中断。

D5~D0, 未定义。

### 11.4.2 实时中断

实时中断(Real Time Interrupt, RTI)功能可定时产生中断。与 PIT 不同的是, RTI 的计时时钟源使用的是内部时钟, 只与晶振有关, 与总线时钟频率无关。

实时中断 RTI 在默认情况下是关闭的, 若要启动 RTI 功能, 配置步骤如下:

① 配置实时中断控制寄存器 RTICTL。配置 RTICTL 的 RTR[6:0]: 当 RTR[6:4]=000 时, 实时中断被禁止; 当 RTR[6:4] 不全为 0 时, RTI 功能开启。同时, RTI 的溢出周期也由该寄存器配置。RTI 的参考时间是外部晶振的时钟 OSCCLK。

RTI 的溢出时钟周期 =  $(RTR[3:0] + 1) \times 2^{RTR[6:4] + 9} / OSCCLK$

其次, 配置寄存器时钟与复位寄存器 CRGINT。若 RTI 使能位 CRGINT\_RTIE=1, 则每个周期结束时, 就会产生一个中断。

CRGFLG\_RTIF 为 RTI 的标志位。当 RTI 溢出时, CRGFLG\_RTIF 由硬件置 1, 向

CRGFLG\_RTIF 写 1,可以清 0 标志位。

与 RTI 相关的寄存器还有 CLKSEL\_RTIWAI,一般较少用到。当 CLKSEL\_RTIWAI=1 时,只要系统进入等待模式,RTI 就停止工作。当 CLKSEL\_RTIWAI=0 时,在等待模式下,RTI 可仍然工作。

### 11.4.3 调试模块 DBG 与软件中断 SWI 指令

调试模块(Debug Module,DBG)可在 BDM 模式下提供给用户断点调试的功能,可以在指定的地址处产生一个中断,该中断称为调试中断,它使 CPU 中止当前程序的执行转而进入调试中断服务程序。调试中断可由下述两种方式触发:

- ① 程序计数器的值与调试地址寄存器的内容相匹配时产生调试中断。
- ② 用软件向调试控制寄存器 1(DBGC1)的 TRIG 位写 1 时产生调试中断。

当调试中断产生时,CPU 在结束当前指令后,将一条 SWI 指令装入内部指令寄存器作为下一条指令执行。这样就如同发生一个软件中断,调试中断向量地址是 0xFFFF6 和 0xFFFF7,与软件中断 SWI 指令产生的中断是同一个中断向量地址。实际上,即使是调试工具的开发也极少单独使用 SWI 指令,而是设置调试中断产生 SWI 中断,在中断服务程序中将当前 MCU 工作状态发送给 PC 机。

从编程角度,调试模块 DBG 主要涉及围绕断点触发和地址比较追踪来产生中断进行调试程序的。根据控制寄存器进行调试功能的设置,A、B 或 C、D 比较器或地址计数结果标示状态寄存器,得到调试结果。

调试中断功能在开发调试工具时使用,一般读者很少用到,这里不详细说明。

# XS128 的映像寄存器

表 A-1 XS128 的映像寄存器

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x0000	PORTA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0x0001	PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0x0002	DDRA	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0
0x0003	DDRB	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0x 0004~0x 0007	Reserved								
0x0008	PORTE	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
0x0009	DDRE	DDRE7	DDRE6	DDRE5	DDRE4	DDRE3	DDRE2	0	0
0x000A	Reserved	Reserved							
0x000B	MODE	MODC	0	0	0	0	0	0	0
0x000C	PUCR	PUPKE	BKPUE	0	PUPEE	0	0	PUPBE	PUPAE
0x000D	RDRIV	RDPK	0	0	RDPE	0	0	RDPB	RDPA
0x000E~0x 000F	Reserved								
0x0010	GPAGE	0	GP6	GP5	GP4	GP3	GP2	GP1	GP0
0x0011	DIRECT	DP15	DP14	DP13	DP12	DP11	DP10	DP9	DP8
0x0012	Reserved								
0x0013	MMCCTL1	MGRAMON	0	DFIFRON	PGMIFRON	0	0	0	0
0x0014	Reserved								
0x0015	PPAGE	PIX7	PIX6	PIX5	PIX4	PIX3	PIX2	PIX1	PIX0
0x0016	RPAGE	RP7	RP6	RP5	RP4	RP3	RP2	RP1	RP0
0x0017	EPAGE	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0
0x0018	Reserved								
0x0019	Reserved								
0x001A	PARTIDH	PARTIDH							

续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x001B	PARTIDL	PARTIDH							
\$ 001C	ECLKCTL	NECLK	NCLKX2	DIV16	EDIV4	EDIV3	EDIV2	EDIV1	EDIV0
0x001D		Reserved							
0x001E	IRQCR	IRQE	IRQEN	0	0	0	0	0	0
0x001F		Reserved							
0x0020	DBGC1	ARM	TRIG	reserved	BDM	DBGBRK	reserved	COMRV	
0x0021	DBGSR	TBF	0	0	0	0	SSF2	SSF1	SSF0
0x0022	DBGTCR	reserved	TSOURCE	TRANGE		TRCMOD		TALIGN	
0x0023	DBGC2	0	0	0	0	CDCM		ABCM	
0x0024	DBGTBH	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
0x0025	DBGTBL	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x0026	DBGCNT	0	CNT						
0x0027	DBGSCRX	0	0	0	0	SC3	SC2	SC1	SC0
0x0027	DBGMFR	0	0	0	0	MC3	MC2	MC1	MC0
0x0028(1)	DBGXCTL (COMP/A/C)	0	NDB	TAG	BRK	RW	RWE	reserved	COMPE
0x0028(2)	DBGXCTL (COMP/B/D)	SZE	SZ	TAG	BRK	RW	RWE	reserved	COMPE
0x0029	DBGXAH	0	Bit 22	21	20	19	18	17	Bit 16
0x002A	DBGXAM	Bit 15	14	13	12	11	10	9	Bit 8
0x002B	DBGXAL	Bit 7	6	5	4	3	2	1	Bit 0
0x002C	DBGXDH	Bit 15	14	13	12	11	10	9	Bit 8
0x002D	DBGXDL	Bit 7	6	5	4	3	2	1	Bit 0
0x002E	DBGXDHM	Bit 15	14	13	12	11	10	9	Bit 8
0x002F	DBGXDLM	Bit 7	6	5	4	3	2	1	Bit 0
0x0030		Reserved							
0x0031		Reserved							
0x0032	PORTK	PK7	0	PK5	PK4	PK3	PK2	PK1	PK0
0x0033	DDRK	DDRK7	0	DDRK5	DDRK4	DDRK3	DDRK2	DDRK1	DDRK0
0x0034	SYNR	VCOFRQ[1 : 0]		SYNDIV[5 : 0]					
0x0035	REFDV	REFFRQ[1 : 0]		REFDIV[5 : 0]					



续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x0036	POSTDIV	0	0	0	REFDIV[5 : 0]				
0x0037	CRGFLG	RTIF	PORF	LVRF	LOCKIF	LOCK	ILAF	SCMIF	SCM
0x0038	CRGINT	RTIE	0	0	LOCKIE	0	0	SCMIE	0
0x0039	CLKSEL	PLLSEL	PSTP	XCLKS	0	PLLWAI	0	RTIWAI	COPWAI
0x003A	PLLCTL	CME	PLLON	FM1	FM0	FSTWKP	PRE	PCE	SCME
0x003B	RTICTL	RTDEC	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
0x003C	COPCTL	WCOP	RSBCK	WRTMAS	0	0	CR2	CR1	CR0
0x003D	FORBYP	0	0	0	0	0	0	0	0
0x003E	CTCTL	0	0	0	0		0	0	0
0x003F	ARMCOP	Bit7	6	5	4	3	2	1	0
0x0040	TIOS	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
0x0041	CFORC	0	0	0	0	0	0	0	0
0x0042	OC7M	OC7M7	OC7M6	OC7M5	OC7M4	OC7M3	OC7M2	OC7M1	OC7M0
0x0043	OC7D	OC7D7	OC7D6	OC7D5	OC7D4	OC7D3	OC7D2	OC7D1	OC7D0
0x0044	TCNTH	Bit15	14	13	12	11	10	9	8
0x0045	TCNTL	Bit7	6	5	4	3	2	1	0
0x0046	TSCR1	TEN	TSWAI	TSFRZ	TFFCA	PRNT	0	0	0
0x0047	TTOV	TOV7	TOV6	TOV5	TOV4	TOV3	TOV2	TOV1	TOV0
0x0048	TCTL1	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
0x0049	TCTL2	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
0x004A	TCTL3	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A
0x004B	TCTL4	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A
0x004C	TIE	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
0x004D	TSCR2	TOI	0	0	0	TCRE	PR2	PR1	PR0
0x004E	TFLG1	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
0x004F	TFLG2	TOF	0	0	0	0	0	0	0
0x0050	TC0H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x0051	TC0L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0052	TC1H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x0053	TC1L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0054	TC2H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8

续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x0055	TC2L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0056	TC3H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x0057	TC3L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0058	TC4H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x0059	TC4L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x005A	TC5H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x005B	TC5L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x005C	TC6H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x005D	TC6L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x005E	TC7H	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
0x005F	TC7L	Bit 7	Bit6	Bit 5	Bit4	Bit3	Bit2	Bit1	Bit0
0x0060	PACTL	0	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI
0x0061	PAFLG	0	0	0	0	0	0	PAOVF	PAIF
0x0062	PACNTH	PACNT15	PACNT14	PACNT13	PACNT12	PACNT11	PACNT10	PACNT9	PACNT8
0x0063	PACNTL	PACNT7	PACNT6	PACNT5	PACNT4	PACNT3	PACNT2	PACNT1	PACNT0
0x0064~0x006B		Reserved							
0x006C	OCPD	OCPD7	OCPD6	OCPD5	OCPD4	OCPD3	OCPD2	OCPD1	OCPD0
0x006D		Reserved							
0x006E	PTPSR	PTPS7	PTPS6	PTPS5	PTPS4	PTPS3	PTPS2	PTPS1	PTPS0
0x006F~0x00C7		Reserved							
0x00C8	SCI0BDH <sub>(1)</sub>	IREN	TNP1	TNP0	SBR12	SBR11	SBR10	SBR9	SBR8
0x00C9	SCI0BDL <sub>1</sub>	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
0x00CA	SCI0CR1 <sub>1</sub>	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
0x00C8	SCI0ASR1 <sub>(2)</sub>	RXEDGIF	0	0	0	0	BERRV	BERRIF	BKDIF
0x00C9	SCI0ACR1 <sub>2</sub>	RXEDGIE	0	0	0	0	0	BERRIE	BKDIE
0x00CA	SCI0ACR2 <sub>2</sub>	0	0	0	0	0	BERRM1	BERRM0	BKDFE
0x00CB	SCI0CR2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
0x00CC	SCI0SR1	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
0x00CD	SCI0SR2	AMAP	0	0	TXPOL	RXPOL	BRK13	TXDIR	RAF
0x00CE	SCI0DRH	R8	T8	0	0	0	0	0	0
0x00CF	SCI0DRL	R7	R6	R5	R4	R3	R2	R1	R0



续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x00D0	SCI1BDH <sub>(1)</sub>	IREN	TNP1	TNP0	SBR12	SBR11	SBR10	SBR9	SBR8
0x00D1	SCI1BDL <sub>1</sub>	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
0x00D2	SCI1CR <sub>1</sub>	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
0x00D0	SCI1ASR <sub>1(2)</sub>	RXEDGIF	0	0	0	0	BERRV	BERRIF	BKDIF
0x00D1	SCI1ACR <sub>12</sub>	RXEDGIE	0	0	0	0	0	BERRIE	BKDIE
0x00D2	SCI1ACR <sub>22</sub>	0	0	0	0	0	BERRM1	BERRM0	BKDFE
0x00D3	SCI1CR <sub>2</sub>	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
0x00D4	SCI1SR <sub>1</sub>	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
0x00D5	SCI1SR <sub>2</sub>	AMAP	0	0	TXPOL	RXPOL	BRK13	TXDIR	RAF
0x00D6	SCI1DRH	R8	T8	0	0	0	0	0	0
0x00D7	SCI1DRL	R7	R6	R5	R4	R3	R2	R1	R0
0x00D8	SPI0CR <sub>1</sub>	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
0x00D9	SPI0CR <sub>2</sub>	0	XFRW	0	MODFEN	BIDIROE	0	SPISWAI	SPC0
0x00DA	SPI0BR	0	SPPR2	SPPR1	SPPR0	0	SPR2	SPR1	SPR0
0x00DB	SPI0SR	SPIF	0	SPTEF	MODF	0	0	0	0
0x00DC	SPI0DRH	R15	R14	R13	R12	R11	R10	R9	R8
0x00DD	SPI0DRL	R7	R6	R5	R4	R3	R2	R1	R0
0x00DE - 0x00FF	Reserved								
0x0100	FCLKDIV	FDIVLD	FDIV6	FDIV5	FDIV4	FDIV3	FDIV2	FDIV1	FDIV0
0x0101	FSEC	KEYEN1	KEYEN0	RNV5	RNV4	RNV3	RNV2	SEC1	SEC0
0x0102	FCCOBIX	0	0	0	0	0	CCOBIX2	CCOBIX1	CCOBIX0
0x0103	FECCR1X	0	0	0	0	0	ECCR1X2	ECCR1X1	ECCR1X0
0x0104	FCNFG	CCIE	0	0	IGNSF	0	0	FDFD	FSFD
0x0105	FERCNFG	0	0	0	0	0	0	DFDIE	SFDIE
0x0106	FSTAT	CCIF	0	ACCERR	FPVIOL	MGBUSY	RSVD	MGSTAT1	MGSTAT0
0x0107	FERSTAT	ERSERIF	PGMERIF	EACCEIF	EPVIOLIF	ERSVIF1	ERSVIF0	DFDIF	SFDIF
0x0108	FPROT	FPOPEN	RNV6	FPHDIS	FPHS1	FPHS0	FPLDIS	FPLS1	FPLS0
0x0109	DFPROT	DPOPEN	0	0	DPS4	DPS3	DPS2	DPS1	DPS0
0x010A	FCCOBHI	CCOB15	CCOB14	CCOB13	CCOB12	CCOB11	CCOB10	CCOB9	CCOB8
0x010B	FCCOBLO	CCOB7	CCOB6	CCOB5	CCOB4	CCOB3	CCOB2	CCOB1	CCOB0



续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x010C – 0x010D		Reserved							
0x010E	FECCRHI	ECCR15	ECCR14	ECCR13	ECCR12	ECCR11	ECCR10	ECCR9	ECCR8
0x010F	FECCRLO	ECCR7	ECCR6	ECCR5	ECCR4	ECCR3	ECCR2	ECCR1	ECCR0
0x0110	FOPT	NV7	NV6	NV5	NV4	NV3	NV2	NV1	NV0
0x0111 – 0x0120		Reserved							
0x0121		IVBR	IVB_ADDR[7 : 0]						
0x0122 – 0x0125		Reserved							
0x0126	INT_XGP RIO	0	0	0	0	0	XILVL[2 : 0]		
0x0127	INT_CFA DDR	INT_CFADDR[7 : 4]				0	0	0	0
0x0128	INT_CFD ATA0	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x0129	INT_CFD ATA1	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012A	NT_CFD ATA2	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012B	NT_CFD ATA3	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012C	NT_CFD ATA4	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012D	NT_CFD ATA5	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012E	NT_CFD ATA6	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x012F	INT_CFD ATA7	RQST	0	0	0	0	PRIOLVL[2 : 0]		
0x0130 – 0x013F		Reserved							
0x0140	CAN0CTL0	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ
0x0141	CAN0CTL1	CANE	CLKSRC	LOOPB	LISTEN	BORM	WUPM	SLPAK	INITAK
0x0142	CAN0BTR0	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
0x0143	CAN0BTR1	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
0x0144	CAN0RFLG	WUPIF	CSCIF	RSTAT 1	RSTAT 0	TSTAT 1	TSTAT 0	OVRIF	RXF
0x0145	CAN0RIER	WUPIE	CSCIE	RSTATE 1	RSTATE 0	TSTATE 1	TSTATE 0	OVRIE	RXFIE
0x0146	CAN0TFLG	0	0	0	0	0	TXE2	TXE1	TXE0
0x0147	CAN0TIER	0	0	0	0	0	TXEIE2	TXEIE1	TXEIE0



续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x0148	CAN0TARQ	0	0	0	0	0	ABTRQ2	ABTRQ1	ABTRQ0
0x0149	CAN0TAAK	0	0	0	0	0	ABTAK2	ABTAK1	ABTAK0
0x014A	CAN0TBSEL	0	0	0	0	0	TX2	TX1	TX0
0x014B	CAN0IDAC	0	0	IDAM1	IDAM0	0	IDHIT2	IDHIT1	IDHIT0
0x014C		Reserved							
0x014D	CAN0MISC	0	0	0	0	0	0	0	BOHOLD
0x014E	CAN0RXERR	RXERR7	RXERR6	RXERR5	RXERR4	RXERR3	RXERR2	RXERR1	RXERR0
0x014F	CAN0TXERR	TXERR7	TXERR6	TXERR5	TXERR4	TXERR3	TXERR2	TXERR1	TXERR0
0x0150～ 0x0153	CAN0IDAR0～ CAN0IDAR3	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
0x0154～ 0x0157	CAN0IDMR0～ CAN0IDMR3	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
0x0158～ 0x015B	CAN0IDAR4～ CAN0IDAR7	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
0x015C～ 0x015F	CAN0IDMR4～ CAN0IDMR7	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
0x0160～ 0x016F	CAN0RXFG	FOREGROUND RECEIVE BUFFER							
0x0170～ 0x017F	CAN0TXFG	FOREGROUND TRANSMIT BUFFER							
0xXXX0	Extended ID	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
	Standard ID	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
	CANxRIDR0	Reserved							
0xXXX1	Extended ID	ID20	ID19	ID18	SRR＝1	IDE＝1	ID17	ID16	ID15
	Standard ID	ID2	ID1	ID0	RTR	IDE＝0	Reserved		
	CANxRIDR1	Reserved							
0xXXX2	Extended ID	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
	Standard ID	Reserved							
0xXXX3	Extended ID	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
	Standard ID	Reserved							
0xXXX4～ 0xXXXB	CANxRDSR0～ CANxRDSR7	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0xXXxC	CANRxDLR	Reserved				DLC3	DLC2	DLC1	DLC0
0xXXXD		Reserved							
0xXXXF	CANxTSRL	TSR7	TSR6	TSR5	TSR4	TSR3	TSR2	TSR1	TSR0
0xXX10	Extended ID CANxTIDR0	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21e
	Standard ID	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
0xXX0x XX10	Extended ID CANxTIDR1	ID20	ID19	ID18	SRR = 1	IDE = 1	ID17	ID16	ID15
	Standard ID	ID2	ID1	ID0	RTR	IDE = 0	Reserved		
0xXX12	Extended ID CANxTIDR2	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
	Standard ID	Reserved							
0xXX13	Extended ID CANxTIDR3	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
	Standard ID	Reserved							
0xXX14~ 0xXX1B	CANxTDSR0~ CANxTDSR7	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0xXX1C	CANxTDLR	Reserved	DLC3	DLC2	DLC1				DLC0
0xXX1D	CANxTTBPR	PRI07	PRI06	PRI05	PRI04	PRI03	PRI02	PRI01	PRI00
0xXX1E	CANxTTSRH	TSR15	TSR14	TSR13	TSR12	TSR11	TSR10	TSR9	TSR8
0xXX1F	CANxTTSRL	TSR7	TSR6	TSR5	TSR4	TSR3	TSR2	TSR1	TSR0
0x0180~0x023F		Reserved							
0x0240	PTT	PTT7	PTT6	PTT5	PTT4	PTT3	PTT2	PTT1	PTT0
0x0241	PTIT	PTIT7	PTIT6	PTIT5	PTIT4	PTIT3	PTIT2	PTIT1	PTIT0
0x0242	DDRT	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0
0x0243	RDRT	RDRT7	RDRT6	RDRT5	RDRT4	RDRT3	RDRT2	RDRT1	RDRT0
0x0244	PERT	PERT7	PERT6	PERT5	PERT4	PERT3	PERT2	PERT1	PERT0
0x0245	PPST	PPST7	PPST6	PPST5	PPST4	PPST3	PPST2	PPST1	PPST0
0x0246		Reserved							
0x0247	PTTRR	PTTRR7	PTTRR6	PTTRR5	PTTRR4	0	PTTRR2	PTTRR1	PTTRR0
0x0248	PTS	PTS7	PTS6	PTS5	PTS4	PTS3	PTS2	PTS1	PTS0
0x0249	PTIS	PTIS7	PTIS6	PTIS5	PTIS4	PTIS3	PTIS2	PTIS1	PTIS0
0x024A	DDRS	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0



续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x024B	RDRS	RDRS7	RDRS6	RDRS5	RDRS4	RDRS3	RDRS2	RDRS1	RDRS0
0x024C	PERS	PERS7	PERS6	PERS5	PERS4	PERS3	PERS2	PERS1	PERS0
0x024D	PPSS	PPSS7	PPSS6	PPSS5	PPSS4	PPSS3	PPSS2	PPSS1	PPSS0
0x024E	WOMS	WOMS7	WOMS6	WOMS5	WOMS4	WOMS3	WOMS2	WOMS1	WOMS0
0x024F		Reserved							
0x0250	PTM	PTM7	PTM6	PTM5	PTM4	PTM3	PTM2	PTM1	PTM0
0x0251	PTIM	PTIM7	PTIM6	PTIM5	PTIM4	PTIM3	PTIM2	PTIM1	PTIM0
0x0252	DDRM	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0
0x0253	RDRM	RDRM7	RDRM6	RDRM5	RDRM4	RDRM3	RDRM2	RDRM1	RDRM0
0x0254	PERM	PERM7	PERM6	PERM5	PERM4	PERM3	PERM2	PERM1	PERM0
0x0255	PPSM	PPSM7	PPSM6	PPSM5	PPSM4	PPSM3	PPSM2	PPSM1	PPSM0
0x0256	WOMM	WOMM7	WOMM6	WOMM5	WOMM4	WOMM3	WOMM2	WOMM1	WOMM0
0x0257	MODRR	MODRR7	MODRR6	0	MODRR4	0	0	0	0
0x0258	PTP	PTP7	PTP6	PTP5	PTP4	PTP3	PTP2	PTP1	PTP0
0x0259	PTIP	PTIP7	PTIP6	PTIP5	PTIP4	PTIP3	PTIP2	PTIP1	PTIP0
0x025A	DDRP	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0
0x025B	RDRP	RDRP7	RDRP6	RDRP5	RDRP4	RDRP3	RDRP2	RDRP1	RDRP0
0x025C	PERP	PERP7	PERP6	PERP5	PERP4	PERP3	PERP2	PERP1	PERP0
0x025D	PPSP	PPSP7	PPSP6	PPSP5	PPSP4	PPSP3	PPSP2	PPSP1	PPSP0
0x025E	PIEP	PIEP7	PIEP6	PIEP5	PIEP4	PIEP3	PIEP2	PIEP1	PIEP0
0x025F	PIFP	PIFP7	PIFP6	PIFP5	PIFP4	PIFP3	PIFP2	PIFP1	PIFP0
0x0260	PTH	PTH7	PTH6	PTH5	PTH4	PTH3	PTH2	PTH1	PTH0
0x0261	PTIH	PTIH7	PTIH6	PTIH5	PTIH4	PTIH3	PTIH2	PTIH1	PTIH0
0x0262	DDRH	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0
0x0263	RDRH	RDRH7	RDRH6	RDRH5	RDRH4	RDRH3	RDRH2	RDRH1	RDRH0
0x0264	PERH	PERH7	PERH6	PERH5	PERH4	PERH3	PERH2	PERH1	PERH0
0x0265	PPSH	PPSH7	PPSH6	PPSH5	PPSH4	PPSH3	PPSH2	PPSH1	PPSH0
0x0266	PIEH	PIEH7	PIEH6	PIEH5	PIEH4	PIEH3	PIEH2	PIEH1	PIEH0
0x0267	PIFH	PIFH7	PIFH6	PIFH5	PIFH4	PIFH3	PIFH2	PIFH1	PIFH0
0x0268	PTJ	PTJ7	PTJ6	0	0	0	0	PTJ1	PTJ0
0x0269	PTIJ	PTIJ7	PTIJ6	0	0	0	0	PTIJ1	PTIJ0

续表 A-1

地 址	寄存器名	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0x026A	DDRJ	DDRJ7	DDRJ6	0	0	0	0	DDRJ1	DDRJ0
0x026B	RDRJ	RDRJ7	RDRJ6	0	0	0	0	RDRJ1	RDRJ0
0x026C	PERJ	PERJ7	PERJ6	0	0	0	0	PERJ1	PERJ1
0x026D	PPSJ	PPSJ7	PPSJ6	0	0	0	0	PPSJ1	PPSJ0
0x026E	PIEJ	PIEJ7	PIEJ6	0	0	0	0	PIEJ1	PIEJ0
0x026F	PIFJ	PIFJ7	PIFJ6	0	0	0	0	PIFJ1	PIFJ0
0x0270	PT0AD0	PT0AD07	PT0AD06	PT0AD05	PT0AD04	PT0AD03	PT0AD02	PT0AD01	PT0AD00
0x0271	PT1AD0	PT1AD07	PT1AD06	PT1AD05	PT1AD04	PT1AD03	PT1AD02	PT1AD01	PT1AD00
0x0272	DDR0AD0	DDR0AD07	DDR0AD06	DDR0AD05	DDR0AD04	DDR0AD03	DDR0AD02	DDR0AD01	DDR0AD00
0x0273	DDR1AD0	DDR1AD07	DDR1AD06	DDR1AD05	DDR1AD04	DDR1AD03	DDR1AD02	DDR1AD01	DDR1AD00
0x0274	RDR0AD0	RDR0AD07	RDR0AD06	RDR0AD05	RDR0AD04	RDR0AD03	RDR0AD02	RDR0AD01	RDR0AD00
0x0275	RDR1AD0	RDR1AD07	RDR1AD06	RDR1AD05	RDR1AD04	RDR1AD03	RDR1AD02	RDR1AD01	RDR1AD00
0x0276	PER0AD0	PER0AD07	PER0AD06	PER0AD05	PER0AD04	PER0AD03	PER0AD02	PER0AD01	PER0AD00
0x0277	PER1AD0	PER1AD07	PER1AD06	PER1AD05	PER1AD04	PER1AD03	PER1AD02	PER1AD01	PER1AD00
0x0278~0x02BF	Reserved								

# 附录 B

## S08 /S12 /ColdFire BDM 简明使用方法

### B.1 S08 /S12 /ColdFire BDM 简介

S08/S12/ColdFire BDM 为 S08/S12/ColdFire 三合一写入器,适用于 Freescale S08/RS08/HCS12/ S12X/ColdFire 全系列 MCU。开发环境使用 Freescale CodeWarrior。写入器使用 USB 线缆连接 PC,使用 BDM 排线连接目标板。其中 S08/RS08/S12 芯片使用 6 针 BDM 接口,ColdFire 芯片使用 26 针 BDM 接口。该写入器配有苏州大学飞思卡尔嵌入式系统研发中心开发的独立写入软件,主要供生产时使用。

### B.2 驱动安装

将写入调试器与 PC 机的 USB 口相连,系统弹出“发现新硬件”的提示,并弹出“找到新的硬件向导”对话框,选择“从列表或指定位置安装(高级)”选项。单击“下一步”,选择“不要搜索,我要自己选择要安装的驱动程序。”,单击“下一步”,选择 Windows CE USB Devices,单击“下一步”,选择“从磁盘安装...”,然后选择驱动程序的路径,选择 driver 文件夹,单击确定将完成 USB 驱动的安装。

### B.3 调试库安装

调试库共包含 5 个文件:OpenSourceBDM.dll、tbdm.dll、tblcf.dll、tblcf.cfg 和 tblcf\_gdi.dll。

S08:将 OpenSourceBDM.dll 复制到 Codewarrior for S08 安装目录\Prog\gdi 目录中。

S12:将 tbdm.dll 复制到 Codewarrior for S12 安装目录\Prog\gdi 目录中。

ColdFire:新建 Codewarrior for ColdFire 安装目录\bin\Plugins\Support\ColdFire\usb-dm\_cf 目录。复制 tblcf.dll、tblcf.cfg 和 tblcf\_gdi.dll 到新建目录中。

以上步骤可通过执行 Install.cmd 脚本完成,首先使用记事本打开 Install.cmd,修改其中 CF\_HCS12\_DIR、CF\_S08\_DIR 和 CF\_CFX\_DIR 这 3 个变量,3 个变量分别对应 Codewarrior for S12、S08 和 Coldfire 的安装路径。修改结束保存,双击执行 Install.cmd 即可完成调试库安装。

## B.4 S08/S12/ColdFire BDM 使用

### B.4.1 S08/RS08/HCS12/S12X MCU 使用

#### 1) 建立 BDM 工程

使用 S08/S12/ColdFire 三合一调试器需要在建立工程时选择相应的连接方式。其中对于 S08/RS08 MCU 选择 S08 OpenSource BDM;S12/S12X MCU 使用 TBDML。S12 连接方式选择见附图 B.1。

#### 2) 使用 S08/S12/ColdFire BDM

单击 Debug 按钮,则弹出配置界面,如图 B.2 所示。选择相应选项,单击 OK 执行写入,写入成功后可以进行在线调试。



图 B.1 S12 连接方式选择

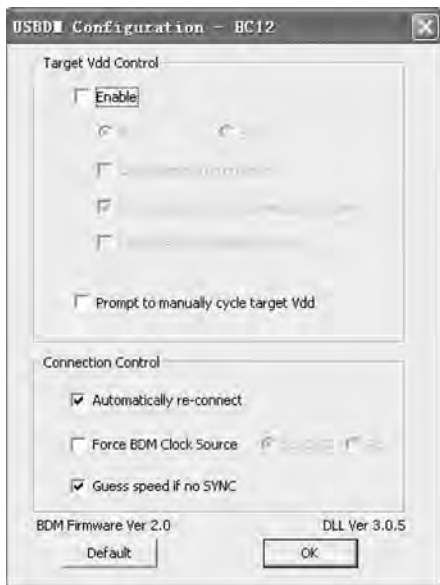


图 B.2 TBDM 配置窗口

### B.4.2 ColdFire MCU 使用

#### 1) 配置 CodeWarrior IDE

- ① 打开 CodeWarrior IDE,并打开工程。
- ② 选择 Editor→Preferences 菜单项。
- ③ 打开 Debugger 树下的 Remote Connection 对话框。
- ④ 单击 Add 输入一个任意名字(比如 USBDM GDI),在 Debugger 下拉列表中选择 ColdFire GDI,然后在 GDI DLL 中输入 tblcf\_gdi.dll 的路径,StartUp 中不用输入内容,单击 OK。
- ⑤ 新连接被建立,单击 OK 关闭 IDE Preferences 对话框。



⑥ 选择 Editor→XXX Preferences 菜单项。打开目标设置。在 Debugger 树中选择 Remote Debugging 设置。在 Connection 中选择刚刚创建的连接(比如 USBDM GDI)。

⑦ 单击 OK 关闭目标设置,现在调试器可以使用 USBDM 接口。

## 2) 使用 S08/S12/ColdFire BDM

① 设置工程连接方式:选择 Edit→XXX Setting(XXX 为工程名)菜单项,则弹出工程设置界面,选择 Debugger 树中的 Remote Debugging,在其中的 Connection 下拉框中选择设置的 S08/S12/ColdFire BDM 连接名称(比如 USBDM GDI)。

② Flash 写入:选择 Tools→Flash Programmer 菜单项,则弹出 Flash Programmer 界面,首先在 Target configuration 中设置目标芯片属性,常用芯片可以通过点击 Load Settings 完成属性设置,然后在 Erase/Blank Check 中单击 Erase 按钮擦除目标芯片 Flash,最后 Program/Verify 中选择工程 Bin 目录中生成的 S19 文件,单击 Program 执行写入。

③ 在线调试:单击 Debug 按钮进行在线调试。

④ 芯片解除密码:打开写入器,将两个拨码开关拨至 JTAG 方向,连接目标板,打开 CF\_Unlocker.exe,单击 Initial JTAG Chain,填写目标芯片晶振频率,单击 Unlock and Erase 进行解除密码。

## B.5 S08/S12/ColdFire BDM 配套独立写入软件

现在以 S12 系列芯片独立写入软件 SD-Programmer-S12.exe 为例,介绍 S08/S12/ColdFire BDM 配套独立写入软件的使用。

① 双击运行 SD-Programmer-S12.exe,则弹出如图 B.3 所示的 BDM 配置界面。选中 Enable 和其他的选用选项。

② 单击 OK 按钮进入如图 B.4 所示的界面即可对芯片进行擦除、写入及读取操作(在芯片未加密的情况下)。

### 1) 擦除操作

选择相应的芯片型号后,单击“擦除”按钮即可擦除整个芯片。

### 2) 写入操作

单击 S19 文件的选择按钮,选择要写入的 S19 文件,然后单击“写入”按钮即可。

### 3) 读取操作

若芯片没有加密,则可以根据需要读取指定地址范围的 Flash 空间的数据。在“起始地址”和“结束地址”中输入要读取的 Flash 地址,单击“读取”按钮即可。



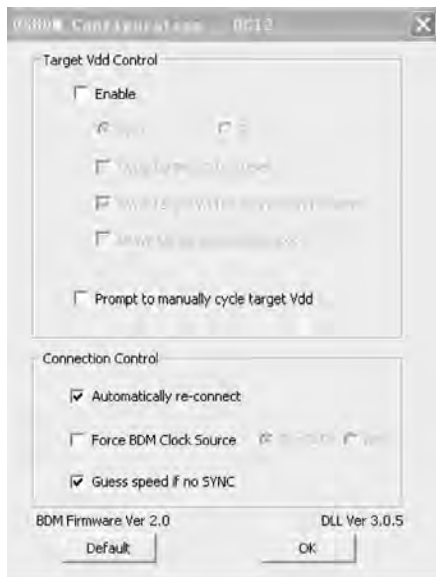


图 B.3 BDM 配置窗口



图 B.4 SD - Programmer - S12 写入窗口

## 常见实践问题集锦

**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的有关评估板调试本书的程序?

**答:**可以。对于 MCU 内部模块,如通用 I/O、定时器、I<sup>2</sup>C、串口通信、Flash、嵌入式以太网、USB 可以使用最小系统板;有些程序需要其他外接模块(如键盘、LED、LCD 等),此时可自己搭建有关电路,也可购买扩展接口板进行硬件调试。

**2 问:**本书介绍的工程测试实例涉及哪些核心板?

**答:**XS128 核心板。

**3 问:**如何焊接和测试,以确保一个最小系统的硬件正常工作?

**答:**①焊接电源、滤波、复位、晶振、PLL 以及写入器接口等电路;②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;③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 机的 USB 接口进行连接。

**5 问:**在 CodeWarrior 集成开发环境中使用写入器对目标 MCU 进行擦除或写入时,出现错误提示“Error: Connect Failed. See Details for additional information”,原因是什么?

**答:**没有装入正确的 xml 文件或 Connection 项目选择错误。

**6 问:**在 CodeWarrior 集成开发环境中使用写入器对目标 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”是什么原因?

**答:**评估板要单独供电,不能仅靠 USB 供电。

**8 问:**在将 XS128 核心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:**若将键盘接上排插座 KeyBoard-ZD,无需任何接线。若将键盘接下排插座 KeyBoard-SD,需从 KeyBoard-SD-JK 处人工连线。将 KeyBoard-SD-JK 的 1~8 脚依次与 PTG 口的 0~4 脚、PTD 口的 2、3、7 脚(或 Core2 的 KEY1~KEY8)连接。

**9 问:**在将 XS128 核心板插入扩展接口板上调试本书 4 连排 LED 显示程序时,还需要人工连接哪些线?

**答:**无需连线。

**10 问:**在调试本书 4 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极之分。扩展接口板仅适用于共阴极类型的数码管。

**11 问:**如何通过万用表的测量判断一个 4 连排 LED 是共阴极还是共阳极的?

**答:**使用万用表的电阻挡(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,则为共阴极数码管。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,则为共阳极数码管。

**12 问:**在将 XS128 核心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:**若将液晶 HD44780 接上排插座 LCD-ZD,无需任何接线。若将液晶 HD44780 接下排插座 LCD-SD,需从 LCD-SD-JK 人工连线。将 LCD-SD-JK 的 D0~D7 与 PTAD 口的 PTAD3~PTAD7 以及 PTJ 口的 6 脚、7 脚和 PTS 口的 3 脚(或 Core2 的 LCD-D0~LCD-D7)连接;将 LCD-SD-JK 的 RS、RW 和 E 分别与 PTP 口的 4 脚、5 脚、7 脚(或 Core2 的 LCD-RS、LCD-RW 和 LCD-E)连接。

**13 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造成的。



**14 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。这是为什么?

**答:**这是由于写入器的干扰造成的。运行程序时,将写入器接插口拔掉即可解决此问题。

**15 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,实际上是将 Flash 测试程序擦除了,程序运行当然就不正常了。

# XS128 的 C 语言函数库

C 语言中的部分库函数在 CodeWarrior 中可以直接使用,和标准 C 的用法一致,下面列出了部分常用函数仅供参考。这些库文件在 C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\lib\hc12c\include 下。不过不推荐大家经常使用,除非特别需要。

## 1. ctype.h 字符类型函数

ctype.h 包含下列的字符函数:

函数名	说 明
int isalnum(char c)	如果 c 是数字或字母,返回非零
int isalpha(char c)	如果 c 是字母,返回非零
int iscntrl(char c)	如果 c 是一个控制字符,返回非零(例如:FF、BELL、LF 等)
int isdigit(char c)	如果 c 是数字,返回非零
int isgraph(char c)	如果 c 是一个可打印的字符且不是空格,返回非零
int islower(char c)	如果 c 是一个小写字母,返回非零
int isprint(char c)	如果 c 是一个可打印的字符,返回非零
int ispunct(char c)	如果 c 是一个可打印的字符,且不是空格、数字、字母,返回非零
int isspace(char c)	如果 c 是一个空字符:space、CR、FF、HT、NL 和 VT,返回非零
int isupper(char c)	如果 c 是一个大写字母,返回非零
int isxdigit(char c)	如果 c 是一个十六进制数字,返回非零
int tolower(int c)	如果 c 是大写字母,返回其小写形式。否则返回 c
int toupper(int c)	如果 c 是小写字母,返回其大写形式。否则返回 c

## 2. math.h 浮点数处理函数

math.h 包含下列的浮点处理函数:



math.h 浮点处理函数

函数名	说 明
double exp(double x)	返回 e 的 x 次方
double fabs(double x)	返回 x 的绝对值
double fmod(double x, double y)	返回 x 除以 y 的余数
double log(double x)	返回 x 的自然对数值
double log10(double x)	返回 x 以 10 为底的对数值
double pow(double x, double y)	返回 x 的 y 次方的值
double sqrt(double x)	返回 x 的正平方根
double sin(double x)	返回弧度 x 的正弦值
double cos(double x)	返回弧度 x 的余弦值
double tan(double x)	返回弧度 x 的正切值
double asin(double x)	返回弧度 x 的反正弦值
double acos(double x)	返回弧度 x 的反余弦值
double atan(double x)	返回弧度 x 的反正切值

### 3. stdio.h 标准输入输出函数

标准输入输出函数提供对串行口的操作,而标准 C 是面向屏幕输出。studio 包含下列函数:

函数名	说 明
int printf(char * fmt,...)	根据 fmt 给出的格式,输出有格式的文本 %d — 输出一个十进制的整数 %o — 输出一个无符号的八进制整数 %x — 输出一个无符号的十六进制整数 %X — 输出一个无符号的十六进制整数字母用大写'A'—'F' %u — 输出一个无符号的十进制整数 %s — 输出一个字符串 %c — 输出一个 ASCII 字符 %f — 输出一个浮点数
int puts(char c)	输出一个单独的字符,它可以在终端上输出'\n'这样的字符
int puts(char * s)	换行输出一个字符串
int sprintf(char * buf, char * fmt)	根据 fmt 给出的格式,输出一串字符到 buf 所指向的内存缓冲区

函数名	说 明
int scanf(CONST char *, ...)	格式化输入
	%d — 输出一个十进制的整数
	%o — 输出一个无符号的八进制整数
	%x — 输出一个无符号的十六进制整数
	%X — 输出一个无符号的十六进制整数字母用大写‘A’-‘F’
	%u — 输出一个无符号的十进制整数
	%s — 输出一个字符串
	%c — 输出一个 ASCII 字符
	%f — 输出一个浮点数

#### 4. stdlib.h 包含内存分配函数

标准的库头文件 stdlib.h 中定义的宏 NULL 和 RAND\_MAX,并重定义的 size\_t。

```
#define NULL    0
#define RAND_MAX    INT_MAX
typedef unsigned int size_t;
```

下面的函数也是在此声明的。在使用内存分配函数(如:calloc, malloc, and realloc)一定要先调用\_NewHeap 初始化 heap。

函数名	说 明
int abs(int i)	返回 i 的绝对值
void * calloc(size_t nelem, size_t size)	返回一块可以容纳 nelem 个对象的内存段,每个对象的大小为 size。初始时内存中均为 0。如果分配不成功则返回 0
void exit(status)	终止程序。主要用在嵌入式环境中(其程序一般是一个永远的循环),在主函数中设一个返回点
void free(void * ptr)	释放一块内存单元
int rand(void)	返回一个 0 到 RAND_MAX 的随机数
void * realloc(void * ptr, size_t size)	重新分配一个以前分配过的内存块 ptr,大小为 size
void srand(unsigned seed)	为 rand(void)调用初始化一个值
long strtol(char * s, char * * endptr, int base)	根据 base 的值将字符 s 转变成一个长整型的数。如果 base 等于 0,函数 strtol 选择 s 中的一个字符(0x 或 0X 表示一个十六进制整数,0 表示一个八进制数,其他表示一个十进制数)。如果 endptr 不为空,则 endptr 将指向 s 中已转变字符的末尾
unsigned long strtoul(char * s, char * * endptr, int base)	返回类型为 unsigned long,其余的与 long strtol(char * s, char * * endptr, int base)同



## 5. string.h 字符串处理函数

string.h 包含下列的字符串函数。其中, string.h 中定义了 NULL、size\_t 和字符数组函数:

函数名	说 明
void * memchr(void * s, int c, size_t n)	在 s 数组的前 n 中查找 c 第一次出现的位置。返回匹配的元素的地址。如果没有匹配的,则返回 null
int memcmp(void * s1, void * s2, size_t n)	比较 s1 和 s2 中前 n 个字符的大小。相等则返回 0, 如果 s1 中第一个和 s2 中不等的字符大于 s2 中对应位置的字符, 返回一个小于 0 的数, 否则返回一个大于 0 的数
void * memcpy(void * s1, void * s2, size_t n)	从 s2 中复制 n 个字符到 s1 中
void * memmove(void * s1, void * s2, size_t n)	从 s2 中复制 n 个字符到 s1 中并返回 s1
void * memset(void * s, int c, size_t n)	在 s 中的前 n 个字符用 c 代替, 返回 s
char * strcat(char * s1, char * s2)	在 s1 之后连接 s2, 并返回 s1
char * strchr(char * s, int c)	在 s 数组中查找 c 第一次出现的位置。返回匹配的元素的地址。如果没有匹配的, 则返回 null
int strcmp(char * s1, char * s2)	比较 s1 和 s2 大小。相等则返回 0, 如果 s1 中第一个和 s2 中不等的字符大于 s2 中对应位置的字符, 返回一个小于 0 的数, 否则返回一个大于 0 的数
char * strcpy(char * s1, char * s2)	复制 s2 中的字符到 s1 中
size_t strcspn(char * s1, char * s2)	查找 s2 中的任一字符在 s1 中第一次匹配的位置。字符串结束符也作为其一部分。返回 s1 中第一次匹配的位置
size_t strlen(char * s)	返回 s 的长度。字符串结束符不计
char * strncat(char * s1, char * s2, size_t n)	连接 s2 到 s1 后, 不包括 s2 的字符串结束符, 并在 s1 的末尾加上字符串结束符。返回 s1
int strncmp(char * s1, char * s2, size_t n)	同 strcmp 函数(但其只比较前 n 个字符)
char * strncpy(char * s1, char * s2, size_t n)	同 strcpy 函数(但其只复制前 n 个字符)
char * strpbrk(char * s1, char * s2)	查找 s2 中的任一字符在 s1 中第一次匹配的位置。不包括字符串结束符。返回 s1 中第一次匹配的位置, 如没找到则返回 null
char * strrchr(char * s, int c)	在 s 数组中查找 c 最后一次出现的位置。返回匹配的元素的地址。如果没有匹配的, 则返回 null
size_t strspn(char * s1, char * s2)	查找 s1 中的任一字符在 s2 中没出现过的的位置。字符串结束符也作为其一部分。返回 s1 中第一次匹配的位置
char * strstr(char * s1, char * s2)	在 s1 中查找和 s2 匹配的子串。如果找到则返回在 s1 中的地址, 否则返回 null



## XS128 的中断源与中断向量表

表 E.1 XS128 的中断源与中断向量表

向量地址	中断源	CCR 掩码	本地使能	停止唤醒	等待唤醒
\$ FFFE	复位引脚(RESET)	None	None	—	—
\$ FFFC	时钟监控复位	None	None	—	—
\$ FFFA	看门狗复位	None	None	—	—
Vector base + \$ F8	Unimplemented instruction trap	None	None	—	—
Vector base + \$ F6	SWI	None	None	—	—
Vector base + \$ F4	XIRQ	X Bit	None	Yes	Yes
Vector base + \$ F2	IRQ	1 Bit	IRQCR (IRQEN)	Yes	Yes
Vector base + \$ F0	Real time interrupt	1 Bit	CRGINT (RTIE)	Refer to CRG interrupt section	
Vector base + \$ EE	TIM timer channel 0	1 Bit	TIE (C0I)		
Vector base + \$ EC	TIM timer channel 1	1 Bit	TIE (C1I)	No	Yes
Vector base + \$ EA	TIM timer channel 2	1 Bit	TIE (C2I)	No	Yes
Vector base + \$ E8	TIM timer channel 3	1 Bit	TIE (C3I)	No	Yes
Vector base + \$ E6	TIM timer channel 4	1 Bit	TIE (C4I)	No	Yes
Vector base + \$ E4	TIM timer channel 5	1 Bit	TIE (C5I)	No	Yes
Vector base + \$ E2	TIM timer channel 6	1 Bit	TIE (C6I)	No	Yes
Vector base + \$ E0	TIM timer channel 7	1 Bit	TIE (C7I)	No	Yes
Vector base + \$ DE	TIM timer overflow	1 Bit	TSRC2 (TOF)	No	Yes
Vector base + \$ DC	TIM Pulse accumulator A overflow	1 Bit	PACTL (PAOVI)	No	Yes
Vector base + \$ DA	TIM Pulse accumulator input edge	1 Bit	PACTL (PAI)	No	Yes
Vector base + \$ D8	SPI0	1 Bit	SPI0CR1 (SPIE, SPTIE)	No	Yes
Vector base + \$ D6	SCI0	1 Bit	SCI0CR2 (TIE, TCIE, RIE, ILIE)	Yes	Yes



续表 E.1

向量地址	中断源	CCR 掩码	本地使能	停止唤醒	等待唤醒
Vector base + \$D4	SCI1	1 Bit	SCI1CR2 (TIE, TCIE, RIE, ILIE)	Yes	Yes
Vector base + \$FD2	ATD0	1 Bit	ATD0CTL2 (ASCIE)	Yes	Yes
Vector base + \$D0	预留				
Vector base + \$CE	Port J	1 Bit	PIEJ(PIEJ7 – PIEJ0)	Yes	Yes
Vector base + \$CC	Port H	1 Bit	PIEH (PIEH7 – PIEH0)	Yes	Yes
Vector base + \$CA	预留				
Vector base + \$C8	预留				
Vector base + \$C6	CRG PLL lock	1 Bit	CRGINT(LOCKIE)	Refer to CRG interrupt section	
Vector base + \$C4	CRG self – clock mode	1 Bit	CRGINT(SCMIE)	Refer to CRG interrupt section	
Vector base + \$C2 To Vector base + \$BC	预留				
Vector base + \$BA	FLASH Fault Detect	1 Bit	FCNFG2 (SFDIE, DFDIE)	No	No
Vector base + \$B8	FLASH	1 Bit	FCNFG (CCIE)	No	Yes
Vector base + \$B6	CAN0 wake – up	1 Bit	CAN0RIER (WUPIE)	Yes	Yes
Vector base + \$B4	CAN0 errors	1 Bit	CAN0RIER (CSCIE, OVRIE)	No	Yes
Vector base + \$B2	CAN0 receive	1 Bit	CAN0RIER(RXFIE)	No	Yes
Vector base + \$B0	CAN0 transmit	1 Bit	CAN0TIER (TXEIE[2 : 0])	No	Yes
Vector base + \$AE To Vector base + \$90	预留				
Vector base + \$8E	Port P Interrupt	1 Bit	PIEP (PIEP7 – PIEP0)	Yes	Yes
Vector base + \$8C	PWM emergency shutdown	1 Bit	PWMSDN (PWMIE)	No	Yes
Vector base + \$8A To Vector base + \$82	预留				
Vector base + \$80	Low – voltage interrupt (LVI)	1 Bit	VREGCTRL (LVIE)	No	Yes

续表 E.1

向量地址	中断源	CCR 掩码	本地使能	停止唤醒	等待唤醒
Vector base + \$7E	Autonomous periodical interrupt (API)	1 Bit	VREGAPICTRL (APIE)	Yes	Yes
Vector base + \$7C	预留				
Vector base + \$7A	Periodic interrupt timer channel 0	1 Bit	PITINTE (PINTE0)	No	Yes
Vector base + \$78	Periodic interrupt timer channel 1	1 Bit	PITINTE (PINTE1)	No	Yes
Vector base + \$76	Periodic interrupt timer channel 2	1 Bit	PITINTE (PINTE2)	No	Yes
Vector base + \$74	Periodic interrupt timer channel 3	1 Bit	PITINTE (PINTE3)	No	Yes
Vector base + \$72 To Vector base + \$40	预留				
Vector base + \$3E	ATD0 Compare Interrupt	1 Bit	ATD0CTL2 (ACMPIE)	Yes	Yes
Vector base + \$3C To Vector base + \$14	预留				
Vector base + \$12	System Call Interrupt (SYS)	—	None	—	—
Vector base + \$10	Spurious interrupt	—	None	—	—
Vector base(16 位向量地址基址)					

## 参考文献

- [1] Freescale. CPU12/CPU12X Reference Manual.
- [2] Freescale. MC9S12XS256 Reference Manual.
- [3] Jack Ganssle, Michael Barr. 英汉双解嵌入式系统词典[M]. 马广云, 译. 北京: 北京航空航天大学出版社, 2006.
- [4] Jack Ganssle. 嵌入式硬件[M]. 和凌志, 译. 北京: 电子工业出版社, 2010.
- [5] 王宜怀, 张书奎, 王林. 嵌入式技术基础与实践(第2版)[M]. 北京: 清华大学出版社, 2011.
- [6] 王宜怀, 陈建明, 蒋银珍. 基于32位 ColdFire 构建嵌入式系统[M]. 北京: 电子工业出版社, 2009.
- [7] 王宜怀, 刘晓升. 嵌入式系统: 使用 HCS12 微控制器的设计与应用[M]. 北京: 航空航天大学出版社, 2008.
- [8] 薛涛, 宫辉, 曾鸣. 单片机与嵌入式系统开发方法[M]. 北京: 清华大学出版社, 2009.
- [9] John Catsoulis. 嵌入式硬件设计[M]. 徐君明, 许铁军, 黄年松, 译. 北京: 中国电力出版社, 2004.
- [10] Randall Hyde. 编程卓越之道(第一卷 深入理解计算机)[M]. 韩东海, 译. 北京: 电子工业出版社, 2006.
- [11] Randall Hyde. 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)[M]. 张菲, 译. 北京: 电子工业出版社, 2007.
- [12] Raj Kamal. 嵌入式系统: 体系结构、编程与设计[M]. 陈曙晖, 译. 北京: 清华大学出版社, 2005.
- [13] Peter Spasov. 微控制器原理与应用(第5版)[M]. 李小洪, 译. 北京: 清华大学出版社, 2006.