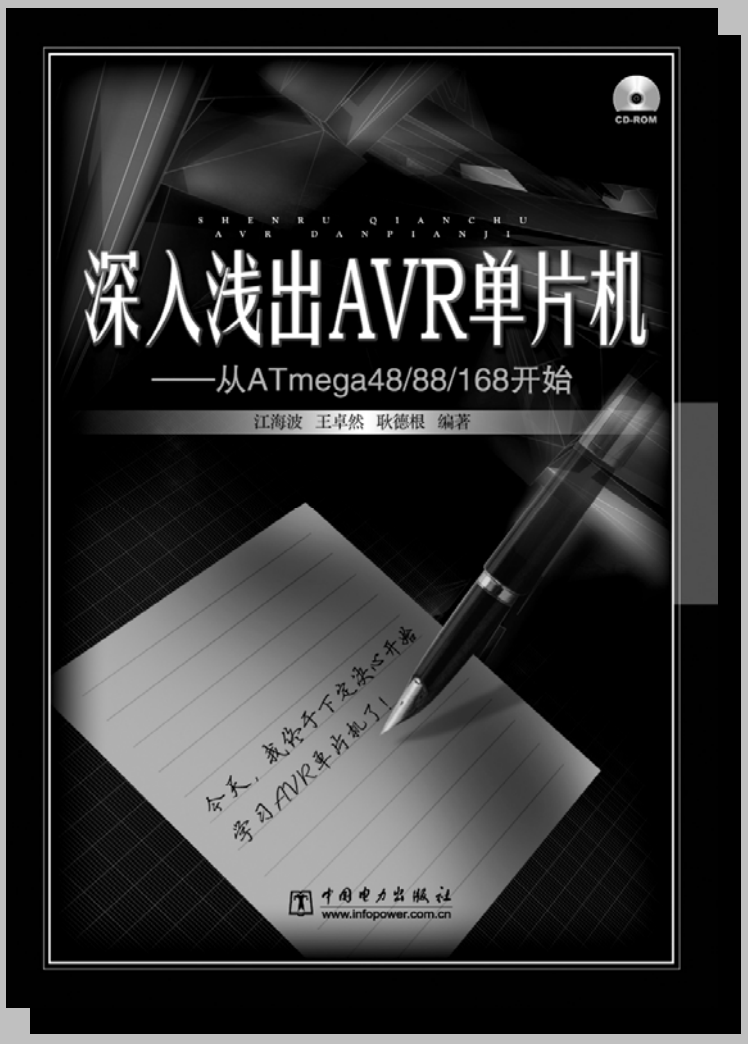
嵌入式工程师快速入门指南

《深入浅出AVR单片机》精要

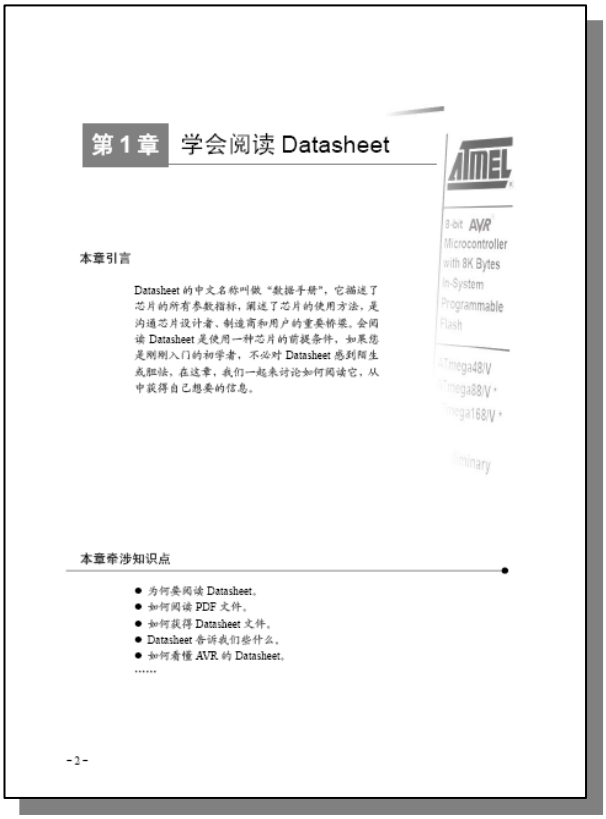
The Essential of Explaining AVR MCU in a simple way —— Getting Started from ATmega48/88/168

王卓然 金刚 ATMEL 上海



The Essential of Explaining AVR MCU in a simple way —— Getting Started from ATmega48/88/168

MENU


第 1 章 学会阅读 Datasheet

本章引言

Datasheet 的中文名称叫做“数据手册”，它描述了芯片的所有参数指标，阐述了芯片的使用方法，是沟通芯片设计者、制造商和用户的重要桥梁。会阅读 Datasheet 是使用一种芯片的前提条件，如果您是刚刚入门的初学者，不必对 Datasheet 感到陌生或胆怯，在这章，我们一起来讨论如何阅读它，从中获得自己想要的信息。

8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash

ATmega48/V
ATmega88/V
ATmega168/V

Preliminary

本章牵涉知识点

- 为何要阅读 Datasheet。
- 如何阅读 PDF 文件。
- 如何获得 Datasheet 文件。
- Datasheet 告诉我们些什么。
- 如何看懂 AVR 的 Datasheet。
- ……

- 2 -

第一章　如何获取官方支持

Rev 1.0.0.0

1.1 内容简介

- ATMEL 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

1.2 ATMEL 简介

ATMEL 公司是是世界上高级半导体产品设计、制造和行销的领先者，产品包括了微处理器、可编程逻辑器件、非易失性存储器、安全芯片、混合信号及 **RF** 射频集成电路。通过对这些核心技术的整合，**ATMEL** 设计生产了各种面向通用目的及特定应用的系统级芯片，以满足当今电子系统设计工程师不断增长和演进的需求。**ATMEL** 在系统级集成方面所拥有的世界级专业知识和丰富的经验，使其产品可以在现有模块的基础上进行快速高效率的开发，并保证最小的风险和研发周期。

ATMEL 专注于高速增长的电子设备市场，如通讯、计算、消费类产品、安全产品、汽车电子和工业应用。作为非易失性存储器技术（**None-Volatile Memory**）之父，**ATMEL** 将该核心技术集成到计算和消费产品之中（比如 **PC**，存储产品，**DVD**，娱乐平台，游戏和玩具）。**ATMEL** 的高密度存储器产品、微控制器和 **ASIC** 同样可以应用到工业控制、军事设备、图像处理和汽车设备。

AVR 是 **ATMEL** 推出的 **8** 位的 **RISC** 微控制器，它在指令执行效率和数据吞吐能力方面表现突出。**AVR** 除了基本的片内 **FLASH** 程序存储器和 **EEPROM** 数据存储器以外，还集成了较 **2** 传统 **51** 单片机更大容量的 **SRAM**（例如 **M88** 系列芯片拥有 **1K** 的 **SRAM** 空间），加之丰富的片内模拟和数字外设，AVR 的灵活性得到了极大的提高，消除了访问外部存储器的瓶颈，提高了程序和数据的安全性。**AVR** 系列微处理器从推出以来至今，产品线已经从 **tinyAVR** 系列延伸到 **XmegaAVR** 系列，从传统的工业控制到消费类电子产品；从汽车电子到公共安全领域；从智能电池到仪器仪表——在社会生产和生活的方方面面都得到了充分的应用。

1.2.1 如何获取最新的产品信息

ATMEL公司通过互联网和电子邮件为用户提供了丰富的产品资料和信息，并通过编写应用手册、产品数据手册、用户指南等文档积极为用户应用提供尽可能详细的参考。不同国家的用户都可以通过网址**www.ATMEL.com**直接访问公司的网站，下载和查阅最新的软件工具和产品信息。

以 **AVR 8-bit** 微控制器为例，为了获取最新的工具软件和文档，用户可以通过网址 **http://www.atmel.com/products** 进入公司产品索引页面（如图 **2.1** 所示）；也可以直接通过主页上的快速导航栏 **Products** 进入产品目录（如图 **2.2** 所示）。

图 2.1 ATEML 产品页面

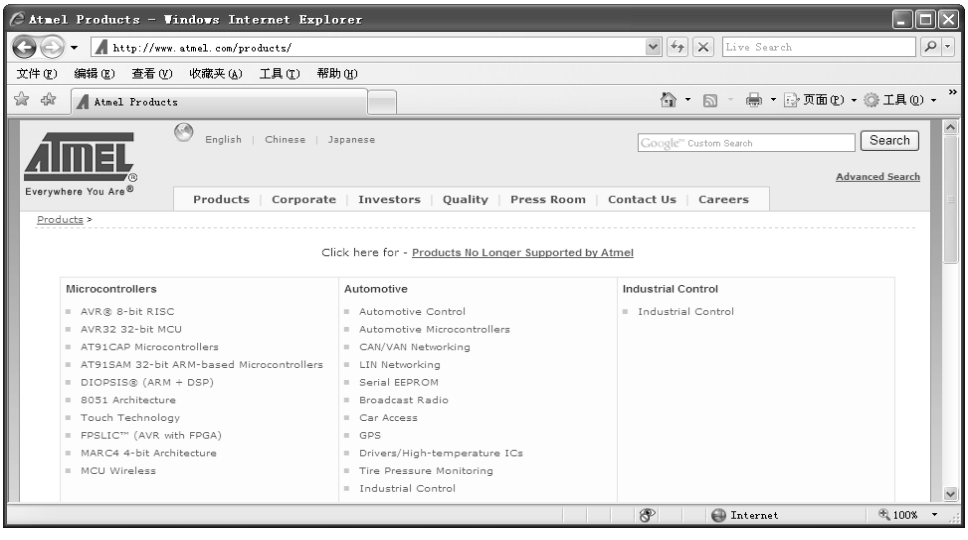
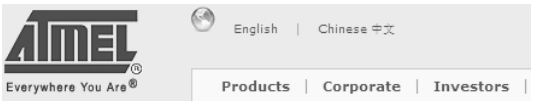
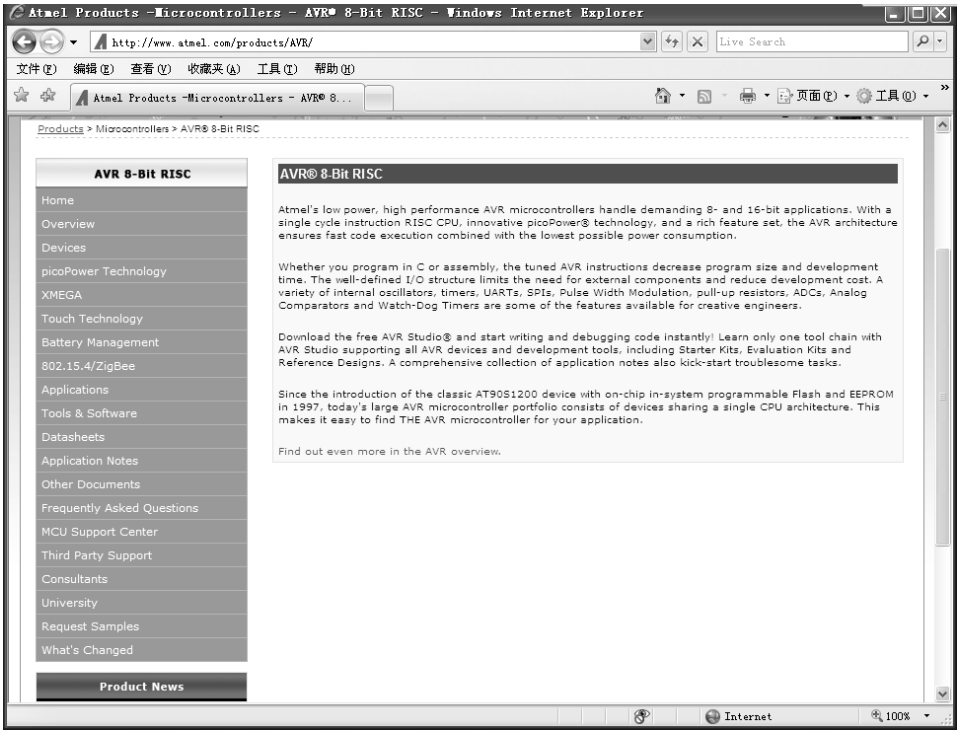


图 2.2 主页导航条



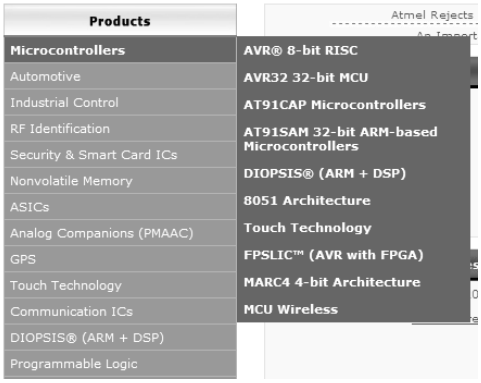
选中具体的产品将进入对应的页面，例如，客户单击 **AVR(R) 8-bit RISC** 将进入 **8** 位 **AVR** 单片机的产品页面（如**图 2.3** 所示）。产品页面的左边是导航目录，通常包含产品特性描述、数据手册/应用文档下载和开发工具等内容。

图 2.3 AVR 8-bit 微处理器产品页面



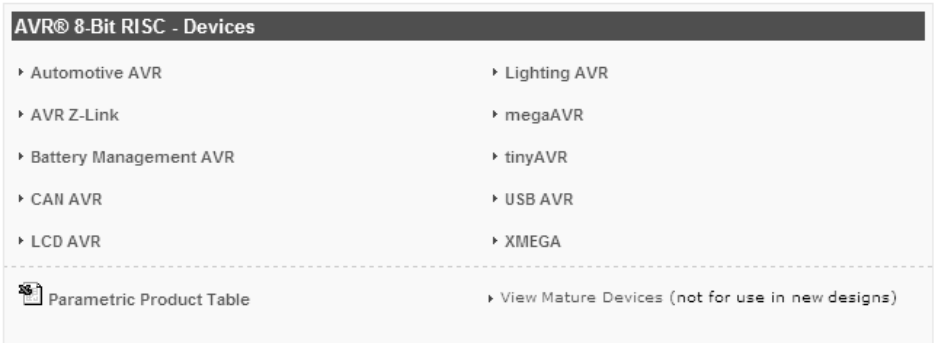
用户也可以通过官方网站首页的产品快捷菜单直接进入对应的产品页面。例如，我们通过 **Microcontrollers->AVR(R) 8-bit RISC** 选项进入**图 2.3** 所示的页面（如**图 2.4** 所示）。

图 2.4 首页产品快捷导航菜单



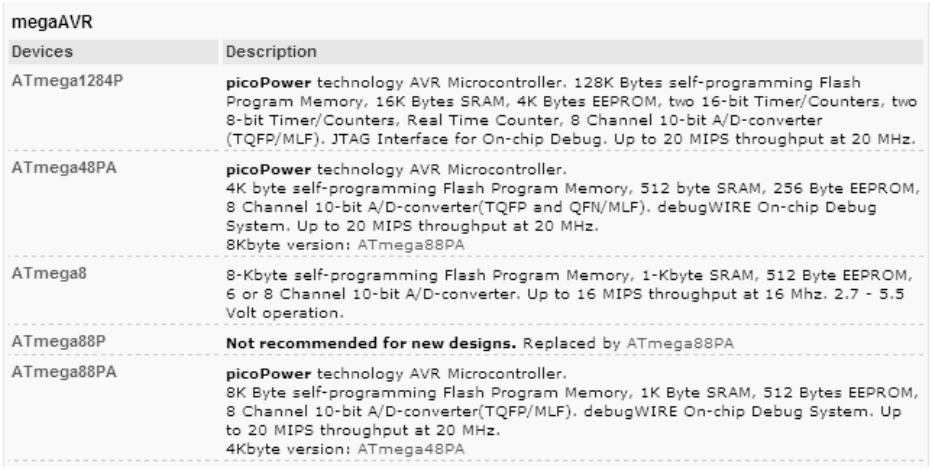
进入产品菜单以后，单击 **Device**，可以选择具体的产品型号。如**图 2.5** 所示，选择 **AVR(R) 8-bit RISC->Devices** 选项，将看到所有 **8** 位 **AVR** 单片机的型号列表。客户可以根据该列表选择具体的单片机系列，查找与具体的芯片型号相关的内容（如**图 2.6** 所示）。

图 2.5 AVR 单片机型号列表



单击具体的芯片型号，例如 **Atmega48P**，将打开对应的页面，列举所有与该芯片有关

图 2.6 megaAVR 具体芯片型号索引



megaAVR

Devices	Description
ATmega1284P	**picoPower** technology AVR Microcontroller. 128K Bytes self-programming Flash Program Memory, 16K Bytes SRAM, 4K Bytes EEPROM, two 16-bit Timer/Counters, two 8-bit Timer/Counters, Real Time Counter, 8 Channel 10-bit A/D-converter (TQFP/MLF). JTAG Interface for On-chip Debug. Up to 20 MIPS throughput at 20 MHz.
ATmega48PA	**picoPower** technology AVR Microcontroller. 4K byte self-programming Flash Program Memory, 512 byte SRAM, 256 Byte EEPROM, 8 Channel 10-bit A/D-converter(TQFP and QFN/MLF). debugWIRE On-chip Debug System. Up to 20 MIPS throughput at 20 MHz. 8Kbyte version: ATmega88PA
ATmega8	8-Kbyte self-programming Flash Program Memory, 1-Kbyte SRAM, 512 Byte EEPROM, 6 or 8 Channel 10-bit A/D-converter. Up to 16 MIPS throughput at 16 Mhz. 2.7 - 5.5 Volt operation.
ATmega88P	**Not recommended for new designs.** Replaced by ATmega88PA
ATmega88PA	**picoPower** technology AVR Microcontroller. 8K Byte self-programming Flash Program Memory, 1K Byte SRAM, 512 Bytes EEPROM, 8 Channel 10-bit A/D-converter(TQFP/MLF). debugWIRE On-chip Debug System. Up to 20 MIPS throughput at 20 MHz. 4Kbyte version: ATmega48PA

的信息，包括：综述信息、数据手册、应用手册、工具和软件等等。

图 2.7 ATmega48PA 综述

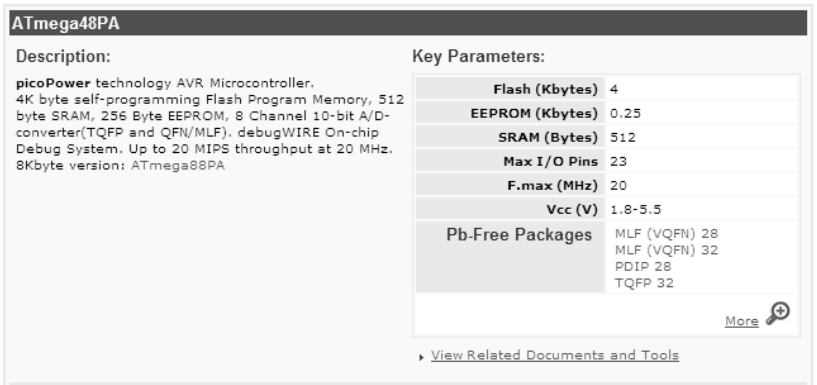


图 2.8 数据手册

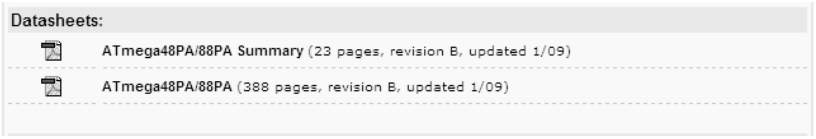


图 2.9 应用手册

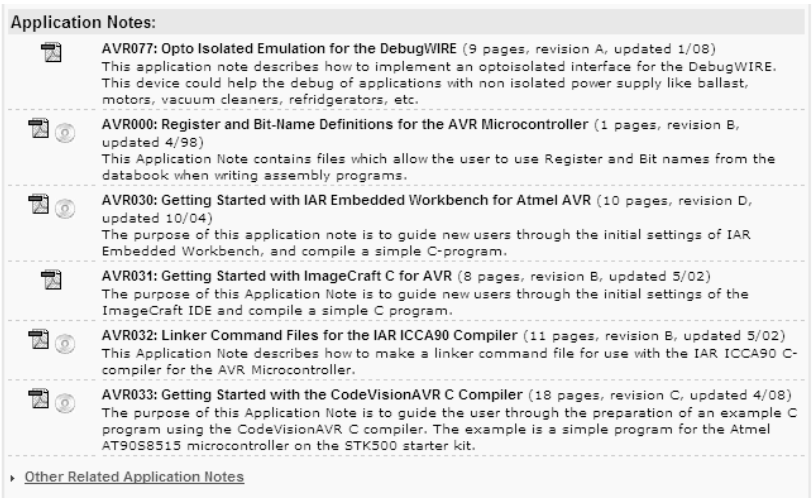
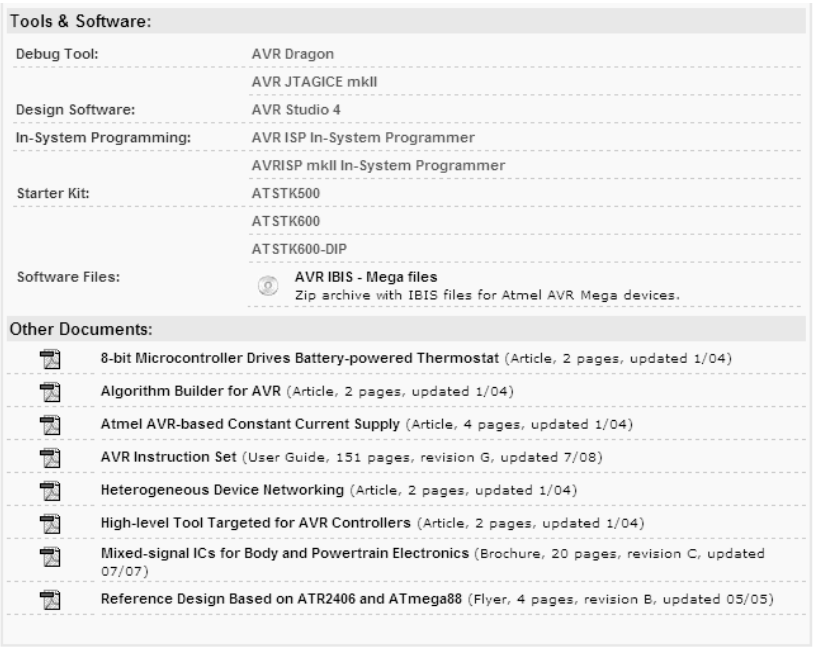


图 2.10 软件和工具

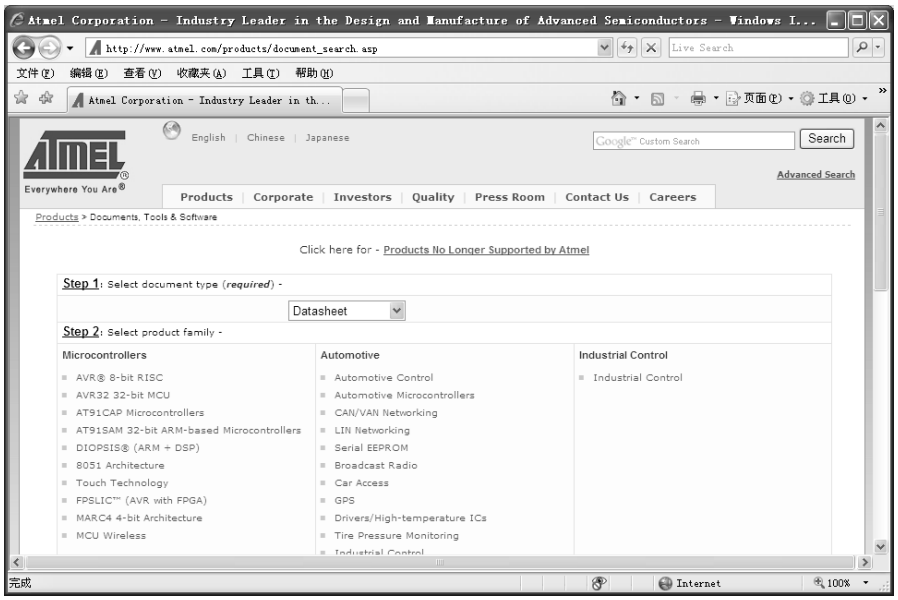


1.2.2 如何获取最新的文档支持

ATMEL 为客户提供了两种主要的途径用以获取最新的技术文档，一种就是通过前文介绍的方式，在具体型号的产品页面中查找相关的文档；一种则是通过官方首页提供的专用文档下载页面来实现。首先，用户进入官方主页以后，选择 **Design Information** 板块的 **Documents, Tools & Software** 选项（如图 **2.11** 所示），打开文档/工具索引页面。

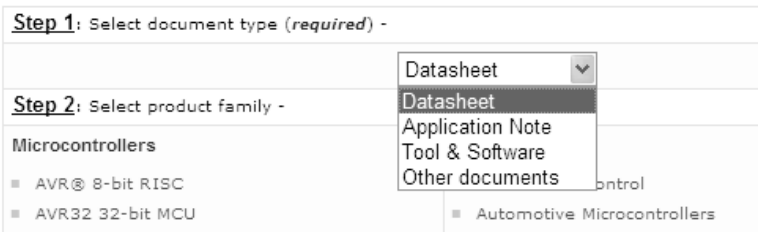
图 2.11 产品设计信息

图 2.12 文档/工具索引页面



按照步骤，首先在 **Step 1** 栏中选择查找对象的类型：数据手册（**Datasheet**）、应用手册（**Application Note**）、软件和工具（**Tool & Software**）还是其它文档（**Other documents**）。接下来，在 **Step 2** 种选择对应的产品类型，即可获得所有与该产品相关的制定信息。比如，查找所有 **AVR(R) 8-bit RISC** 的数据手册（如 **2.13** 所示）。

图 2.13 文档/工具索引页面



1.2.3 如何获取技术支持

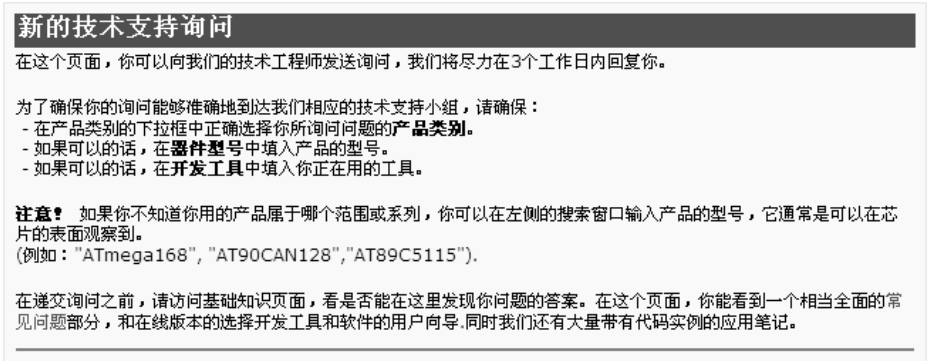
ATMEL 为客户提供了基于电子邮件的技术支持，用户只需要注册一个账号，并填写相关的信息即可以公司或者个人的名义进行提问，获取来自官方团队的直接指导。对中国的 **AVR** 用户来说，可以直接使用中文进行提问。

首先，打开公司主页，在 **Support Center** 栏中选择 **MCU Support Center**（如**图 2.14** 所示）。通常情况下，我们应该首先查阅常见问题 **FAQs**，检查官方对于和我们类似疑问是否已经有了一个明确的答案。首先求助 **FAQ** 是一个好习惯，也能往往能够避免邮件交流所带来的延迟（通常为 **24~72** 小时）。

图 2.14 支持中心快捷导航

对于中文用户，进入 **MCU Support Center** 以后，可以通过页面左上角的语言选择切换到中文状态（如**图 2.2** 所示）。在中文状态下的操作与英文模式并不同。用户将在屏幕的中心看到以中文书写的“提问向导”：

图 2.15 提问向导



通过屏幕左边的导航菜单，用户可以登录/注册帐号、“提新的问题”或者列出以往的提问历史。注册的过程非常简单，这里就不再赘述。为了方便后面的讲解，我们假设用户已经注册成功了一个有效的帐户。单击提交新的问题，将弹出新的页面：以邮件的形式指导用户提交新的问题（如**图 2.17** 所示）。

图 2.16 支持中心用户菜单

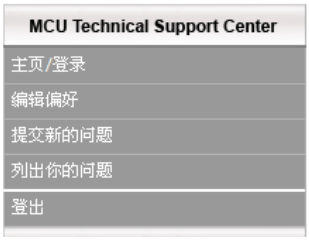


图 2.17 新问题提交页面

1.2.4 如何参与交流

ATMEL 官方提供了为不同语言的用户提供了不同的超级链接，用以为不同语言用户推荐指定的 **AVR** 交流社区。如**图 2.18** 所示，所有的英语用户都被推荐参与 **AVR Freaks** 的讨论和交流，而所有的中文用户都可以通过“原 **Ouravr.com** 技术论坛”进入“我们的 **AVR**”中文讨论社区。

图 2. 18 官方推荐的交流社区链接

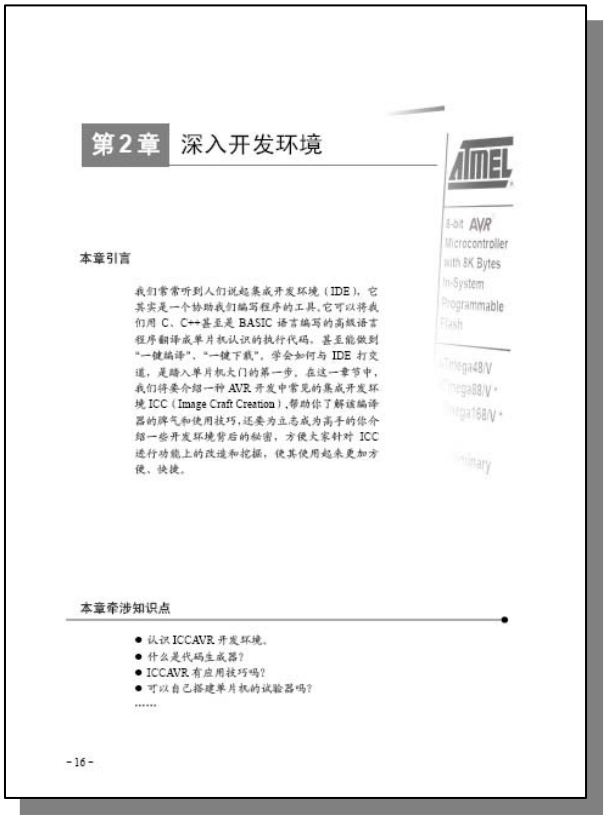
1.3 学生用书及幻灯片注解

1.4 常见课堂问题 FAQ

1.5 参考资料

1.6 背景知识

1.7 参考设计与实验方法



第2章 深入开发环境

本章引言

我们常常听到人们说起集成开发环境（IDE），它其实是一个协助我们编写程序的工具，它可以将我们用 C、C++甚至是 BASIC 语言编写的高级语言程序翻译成单片机认识的执行代码，甚至能做到"一键编译"、"一键下载"，学会如何与 IDE 打交道，是踏入单片机大门的第一步，在这一章节中，我们将要介绍一种 AVR 开发中常见的集成开发环境 ICC（Image Craft Creation），帮助你了解该编译器的脾气和使用技巧，还要为立志成为高手的你介绍一些开发环境背后的秘密，方便大家针对 ICC 进行功能上的改造和挖掘，使其使用起来更加方便、快捷。

本章牵涉知识点

- 认识 ICCAVR 开发环境。
- 什么是代码生成器？
- ICCAVR 有应用技巧吗？
- 可以自己搭建单片机的试验器吗？
- ……

- 16 -

第二章　AVR Studio 4 开发环境

Rev 1.0.0.0

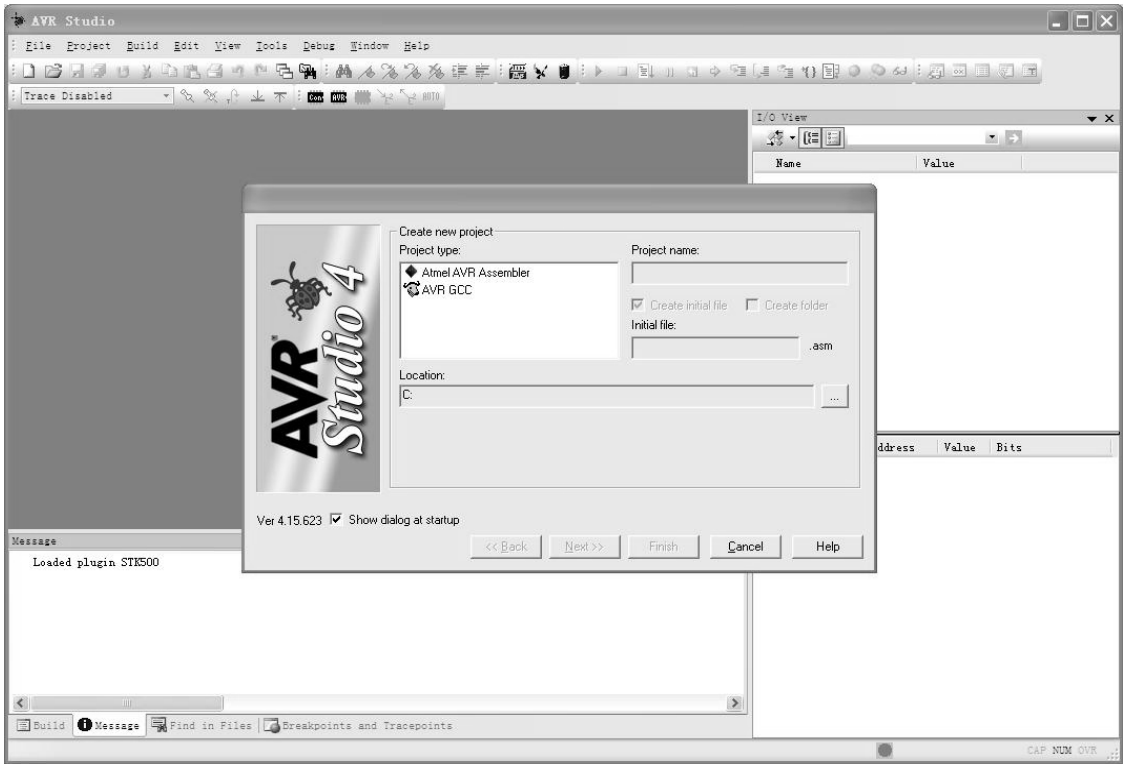
10.1 内容简介

- AVR Studio 4 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

10.2 AVR Studio 4 简介

AVR Studio 4 是一款由 **ATMEL** 开发维护并免费提供的集成开发环境软件（**IDE**）。设计者可以在 **AVR Studio 4** 平台上编写，编译，管理及仿真 **AVR C/C++**和汇编代码。**AVR Studio 4** 自身并没有集成 **C** 编译器，而是将 **Winavr**（**GNU GCC**）作为插件来实现 **AVR** 系列的编译功能。（如**图 10.1** 所示）。

图 10.1AVRStidio 4 界面



10.2.1 名词解释

● **集成开发环境（IDE）**

集成开发环境 **(Integrated Development Environment,** 简称 **IDE,** 也有人称为 **Integration Design Environment、Integration Debugging Environment）**是一种辅助程序开发人员开发软件的应用软件。

IDE 通常包括编程语言编辑器、编译器/解释器、自动建立工具、通常还包括调试器。有时还会包含版本控制系统和一些可以设计图形用户界面的工具。许多支持面向对象的现代化 **IDE** 还包括了类别浏览器、物件检视器、物件结构图。**IDE** 主要还是针对特定的编程语言而量身打造。

● **GNU 编译器套装(GCC)**

GCC（GNU Compiler Collection，GNU 编译器套装**)**，是一套由 **GNU** 开发的编程语言编译器。它是一套以 **GPL** 及 **LGPL** 许可证所发行的自由软件，**也是 GNU** 计划 的关键部分，

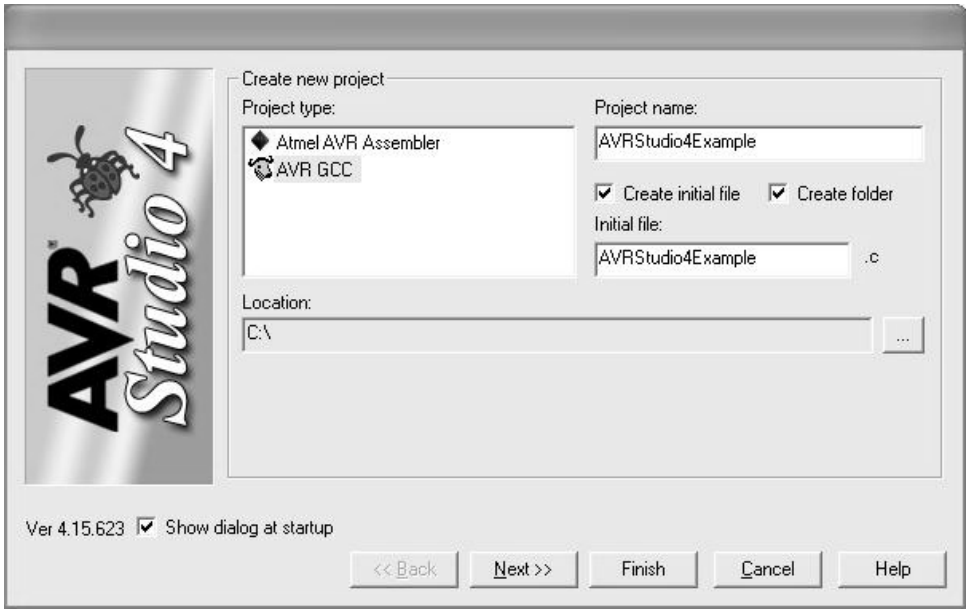
亦是自由的 类 **Unix** 及苹果计算机 **Mac OS X** 操作系统的标准编译器。**GCC (**特别是其中的 **C** 语言编译器**)** 也常被认为是跨平台编译器的事实标准。

GCC 原名为 **GNU C** 语言编译器 **(GNU C Compiler),** 因为它原本只能处理 **C** 语言。**GCC** 很快地扩展，变得可处理 **C++**。之后也变得可处理 **Fortran**、**Pascal**、**Objective-C**、**Java,** 以及 **Ada** 与其他语言。

10.2.2 图解 AVRStudio4 开发流程

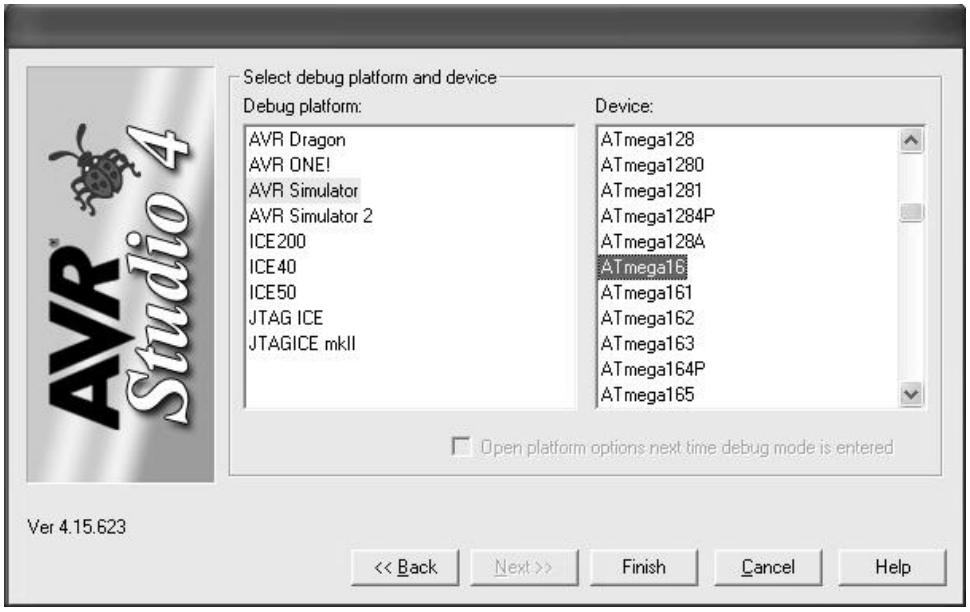
第一步：新建工程。打开 **AVR Studio4**，通过菜单 **Project->New Project** 新建一个 **AVR GCC** 工程（如**图 10.2** 所示）。图中例子所用的工程名为 **AVRStidio4Example**。系统会自动指定一个同名的源程序文件 **AVRStidio4Example.c**。

图 10.2 New Project 窗口



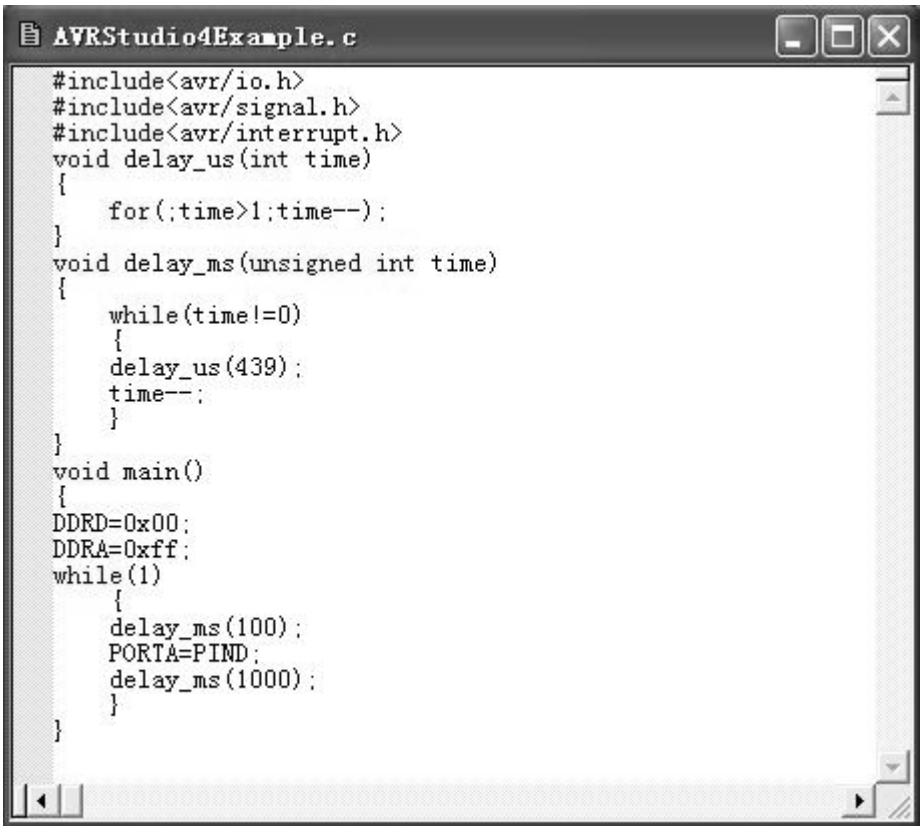
第二步：选择仿真器。设置 **AVR** 模拟器（**Simulator**），并选择目标芯片的型号（如**图 10.3** 所示）。图中例子选择 **AVR Simulator** 来实现 **ATmega16** 的仿真。

图 10.3 AVR 仿真器设置窗口



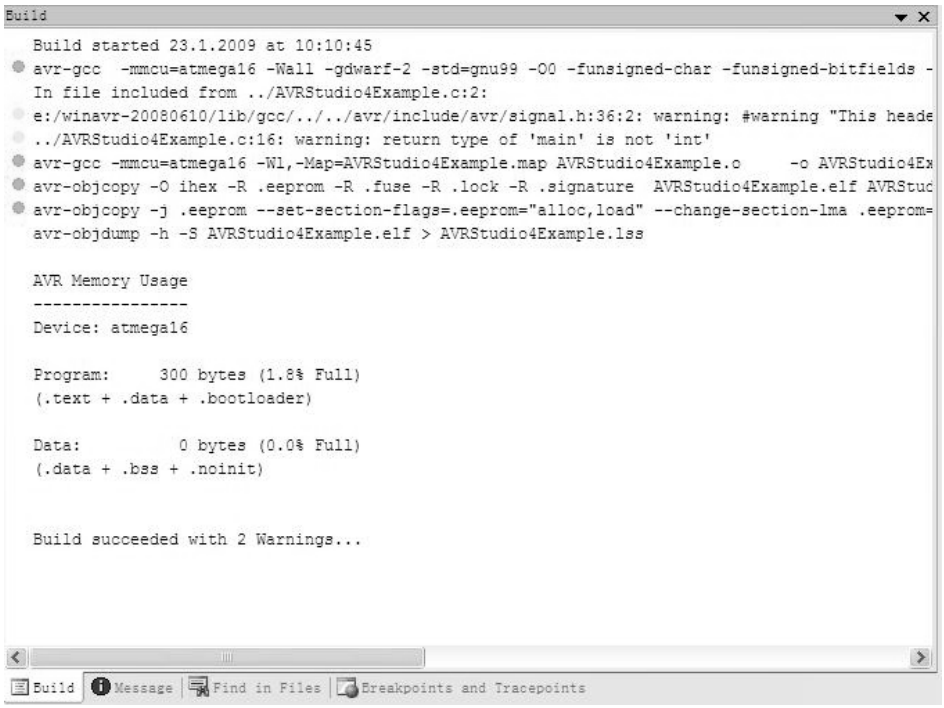
第三步： 编码。在编辑窗口中输入 **C** 代码（如**图 10.4** 所示）。图中的代码演示了 **ATmega16** 的简单程序。

图 10.4 演示程序工程代码窗口



编译（如图 **10.5** 所示）。

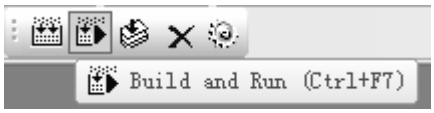
图 10.5 Build 信息窗口



编译完成后在工程设定的工程文件夹下将生成对应的可执行文件。

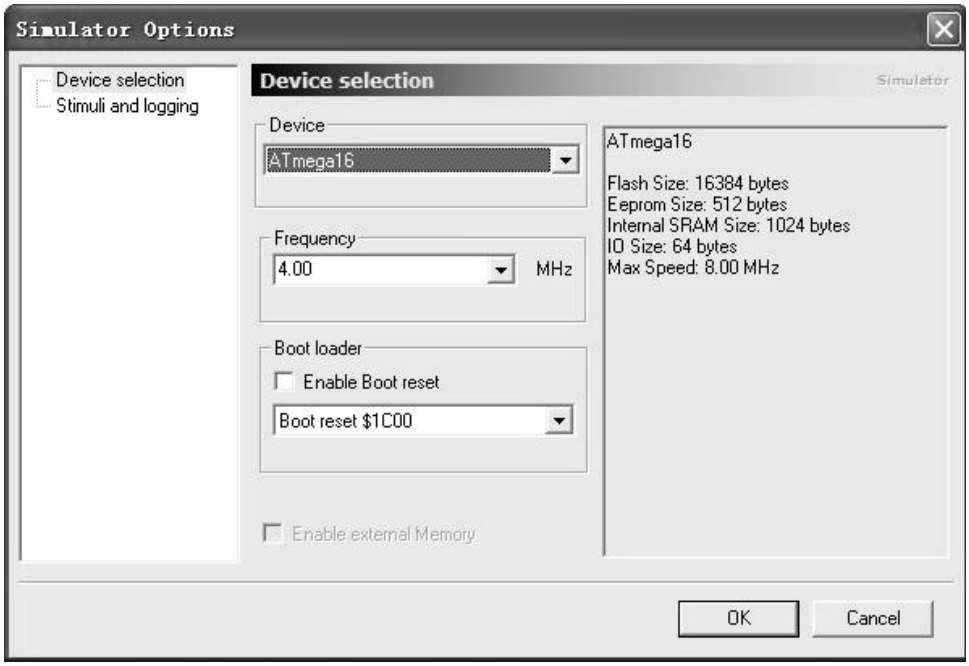
第五步：配置仿真器。通过菜单项 **Build->Build and Run** 或者单击工具栏快捷按钮（如**图 10.7** 所示），对程序进行编译并启动仿真器。

图 10.7 编译工具栏



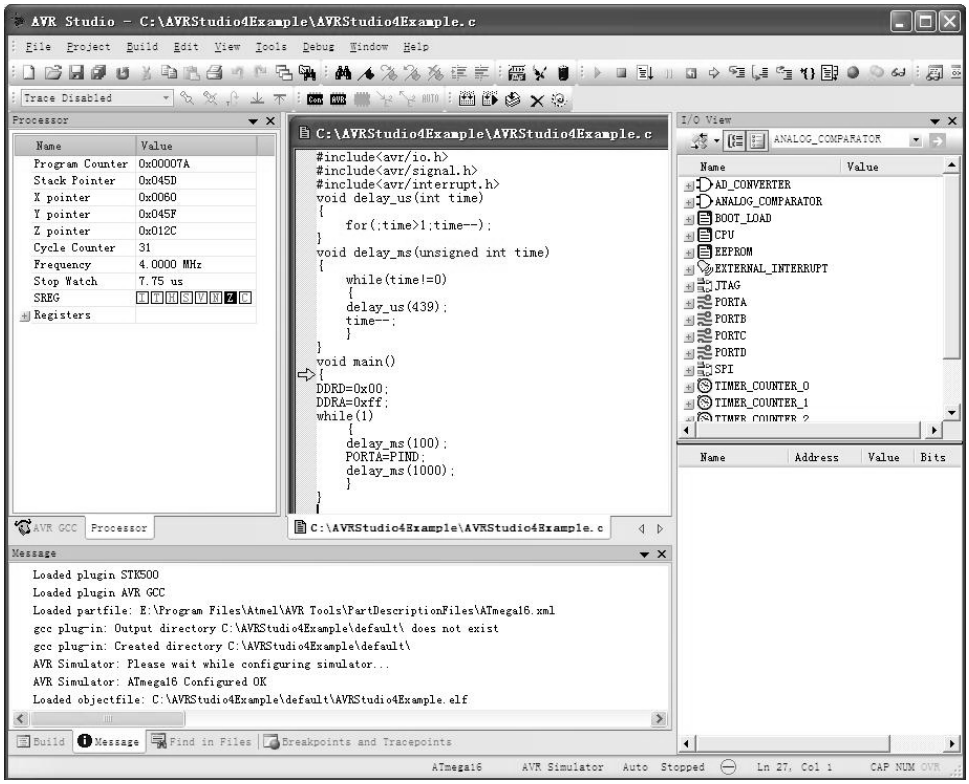
启动仿真器后，选择菜单 **Debug-> AVR Simulator Options** 设置仿真器的工作环境，包括仿真器件、工作频率和 **Bootloader** 设定。

图 10.8 仿真配置窗口



第六步： 调试。选择菜单 **Debug->Run** 运行工程。（如**图 10.9** 所示）。

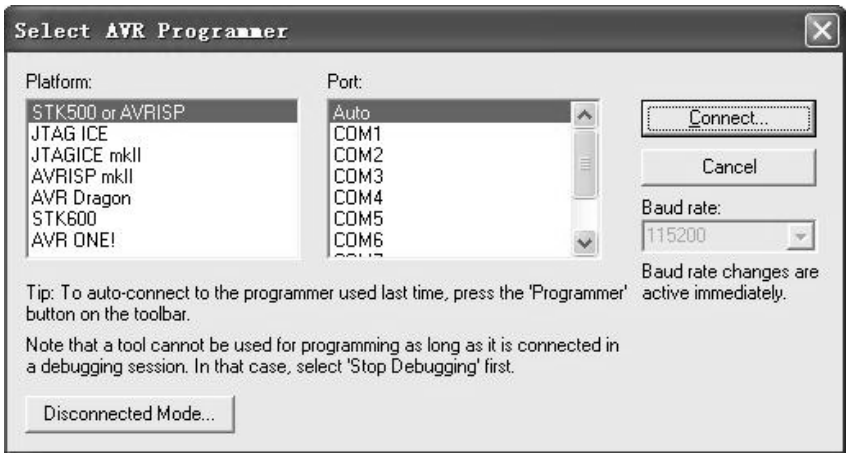
图 10.9 仿真界面



设计者可以在仿真界面中查看所有的芯片寄存器信息以及当前 **CPU** 个寄存器状态，并可以切换为汇编代码。

第七步：下载。编译通过后可以通过下载工具将可执行文件下载至目标板上。通过菜单项 **Tools->Program AVR->Connect** 或者单击工具栏快捷按钮  （如**图 10.10** 所示）。

图 10.10 下载界面



设计者根据实际的开发环境进行设置后，即可将成完成下载，并检查运行情况。更详细，请参考 **AVR Studio 4** 帮助文档。

10.3 学生用书及幻灯片注解

10.4 常见课堂问题 FAQ

10.5 参考资料

10.5.1 应用手册（Application Note）

- **Designing for Efficient Production with In-System Re-programmable Flash Microcontrollers**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/issue4_pg16_17_DesignC.pdf**

- **Release notes AVR Studio 4**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/ReleaseNotesStudio4.txt**

- **AVR Studio - Software Development Environment**

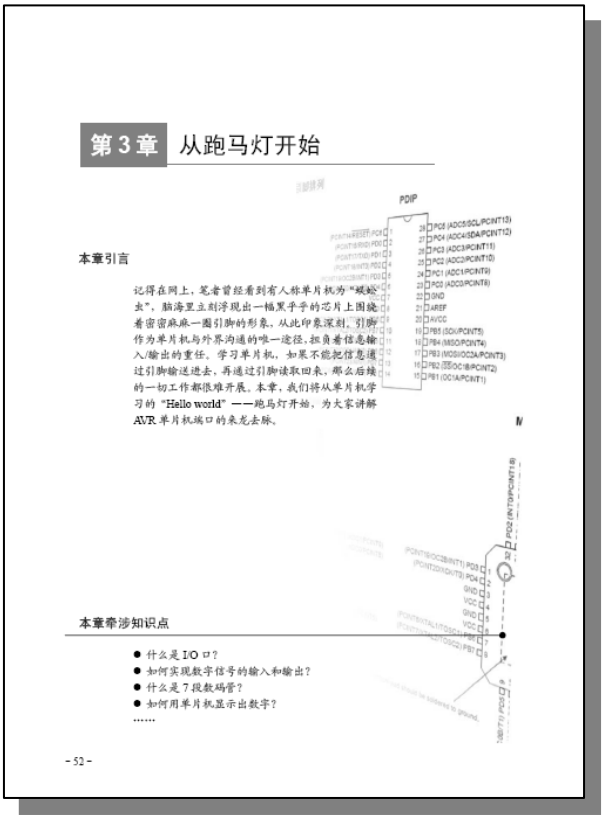
下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2510.pdf**

- **Novice's Guide to AVR Development**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/novice.pdf**

10.6 背景知识

10.7 参考设计与实验方法


第 3 章 从跑马灯开始

本章引言

记得在网上，笔者曾经看到有人称单片机为“蝴蚊虫”，脑海里立刻浮现出一幅黑乎乎的芯片上围绕着密密麻麻一圈引脚的形象，从此印象深刻。引脚作为单片机与外界沟通的唯一途径，担负着信息输入/输出的重任。学习单片机，如果不能把信息通过引脚输送进去，再通过引脚读取回来，那么后续的一切工作都很难开展。本章，我们将从单片机学习的“Hello world”——跑马灯开始，为大家讲解AVR 单片机端口的来龙去脉。

本章牵涉知识点

- 什么是 I/O 口？
- 如何实现数字信号的输入和输出？
- 什么是 7 段数码管？
- 如何用单片机显示出数字？
- ……

- 52 -

第三章　普通端口操作

Rev 1.0.0.3

3.1 内容简介

- GPIO 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

3.2 GPIO 简介

GPIO 是通用输入输出 **General Purpose Input/Output** 的英文缩写。**GPIO** 以端口（**Port**）为组织单位实现芯片与外界的电平交换。一个典型的 **GPIO** 端口通常由 **8** 个、**16** 个或 **32** 个引脚（**Pin**）构成。

如果一个引脚只具有电平的输出能力，称该引脚为输出引脚或驱动引脚；如果一个引脚具有电平的输入能力，则称该引脚为输入引脚。同时具备输入输出能力的引脚称为通用引脚。如果一个端口上所有的引脚都是输入引脚，称为输入端口；如果一个端口上所有的引脚都是输出引脚，则称为输出端口或驱动端口。如果一个端口上所有的引脚都是通用引脚，并且引脚与引脚之间可以独立的输入、输出电平而互不干涉，则称该端口具有独立读取、修改和写入特性 **(Read-Modify-Write Functionality)** 。

习惯上，我们用 **A**、**B**、**C**……这样的字母来给端口编号，用 **0**、**1**、**2**、**3**……这样的整数给引脚编号。当的端口任意时，我们用字母 **x** 来指代，例如：**DDRx**、**PORTx**、**PINx**。当引脚任意时，我们用字母 **n** 来表示，例如 **PAn** 表示端口 **A** 上的任意引脚；**Pxn** 表示任意端口上的任意引脚。

引脚输出高电平时形成的电流称为拉电流；引脚输出低电平时形成的电流称为灌电流。拉电流和灌电流的大小是衡量端口驱动能力的重要指标。为了增强引脚的驱动能力，有时需要配合推挽电路实现推挽输出；或者使用开漏输出配合上拉电阻的电平输出模式。除了基本的高低电平输出以外，**GPIO** 还可能具有开漏/高阻态模式。当引脚处于输入状态时往往保持高阻态，配合上拉电阻读取来自外部的低电平信息。

3.2.1 名词解释

- **数字引脚三态**

数字引脚输出高电平、低电平和高阻态称为“三态”。连接在总线上的数字引脚往往被要求具有三态输出能力；当数字引脚连接了上拉电阻时，引脚只有低电平和高电平两种状态。

- **独立读取/修改/写入特性（Read-Modify-Write Functionality）**

数字端口上的引脚可以被独立的读取、修改和写入而不会影响同端口上其它引脚的特性称为独立读取/修改/写入特性（**Read-Modify-Write Functionality**）。

- **开集/开漏输出**

使用三极管集电极开路输出称为“开集”输出；使用 **MOS** 管漏集开路输出称为“开漏”输出。开集/开漏输出的特点是：可以稳定的输出低电平，可灌入大电流；无法独立输出高电平，需要配合对应的上拉电阻，输出的高电平由上拉电平决定，可以用于信号的电平转换；当多个开集/开漏输出引脚连接在一起时，可以实现线与逻辑。使用开集/开漏输出高电平时，上拉电阻的大小与电平的建立时间呈反比，与高电平的稳定性成正比。

- **推挽输出**

一种输出结构，它使用一个有源器件输出电流，另一个吸收电流。常见的例子有：使用 **n** 沟道器件拉至地或负电压；使用 **p** 沟道器件源出电流提升输出的 **CMOS** 电路。推挽输出具有缩短电平建立时间，提升输出能力的作用。

● **线与逻辑**

TTL 电路或 **CMOS** 电路中，多个开集/开漏输出的引脚连接在一起时，当任意一个引脚输出低电平信号都将变成低电平的特性，称为线与特性。线与逻辑常被用于总线设计和简易电平转换。

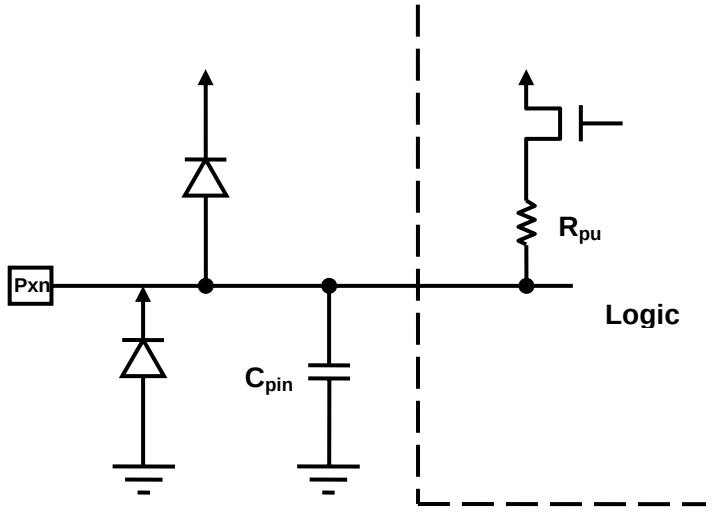
3.2.2 AVR 片上 GPIO 综述

ATmega48/88/168 的 **GPIO** 模块由 **PB**、**PC** 和 **PD** 三个端口组成，每个端口都是 **8** 位的。每个端口都可以实现电平的输入和输出。端口内每一个引脚都可以实现三态输出。每一个引脚都有一个程序可控的内部上拉电阻，其电阻范围在 **20K~50K** 之间。**GPIO** 模块的每个端口都具有独立读取、修改和写入特性。

当作为普通的电平输入输出时，任意时刻，端口上每个引脚的当前实际电平信息都可以通过寄存器 **PINx** 来读取；端口方向寄存器 **DDRx** 用以控制端口上引脚的输入输出方向。无论引脚是否处于输出模式，引脚的实际电平信息都可以通过 **PINx** 来获取；因此可以认为端口方向寄存器 **DDRx** 实际上是引脚驱动使能寄存器（**Pin Driver Enable Register**），其功能用于开启或关闭引脚对外输出电平的功能。

当引脚对外输出电平的功能被开启时，寄存器 **PORTx** 用于设置端口上对应引脚的输出电平。每个引脚都具有 **20mA** 的驱动能力（拉电流和灌电流都是 **20mA**），极限情况下可以达到 **40mA**，但需要注意所有引脚的总电流不应该超过 **200mA**。当引脚对外输出电平的功能被关闭时，寄存器 **PORTx** 用于开启或关闭对应引脚的上拉电阻。置位表示开启上拉电阻；清零表示将对应引脚设置为高阻态。**ATmega48/88/168** 在系统寄存器 **MCUCR** 中有一个全局的上拉电阻屏蔽标志 **PUD**，默认情况下该位为 **0**，所有 **GPIO** 端口的上拉电阻都是可以通过 **DDRx** 寄存器和 **PORTx** 寄存器来控制的；当 **PUD** 标志被置位时，所有的上拉电阻将被强制关闭。

图 3.1 AVR GPIO 引脚等效原理图



每个 **GPIO** 引脚除了基本的电平输入输出功能以外都支持一个或多个第二设备功能，

例如 **PD0** 除了基本的电平输入输出功能以外，还是 **USART** 模块的数据接收引脚 **RXD**，同时也是引脚电平变化中断的电平输入引脚 **PCINT16**。当引脚工作于第二设备功能状态时，可能会完全、部分或者不取代普通 **GPIO** 功能。例如，当 **PB6** 和 **PB7** 作为外部晶振时钟引入引脚时普通 **GPIO** 的功能将被完全取代；当 **PD1** 作为 **USART** 的数据发送引脚时，还可以通过 **GPIO** 控制上拉电阻的开启或者关闭；当 **PD2** 作为外中断 **0** 的信号引脚时，普通 **GPIO** 的输入输出功能完全不受影响。具体的细节因模块和设备而不同，请查阅器件手册 **13.I/O-Ports** 章节的 **13.3 Alternate Port Functions** 小结获得每个引脚设备功能的具体信息。

3.2.3 代码范例

3.2.3.1 基本的输入输出操作

```
# define _BV(__VAL)                    (1 << (__VAL))
……

/* 将 PD0~PD3 设置为输出状态 */
DDRD |= 0x0F;         //或者写为 DDRD |= _BV(0) | _BV(1) | _BV(2) | _BV(3);

/* 在 PD0 和 PD1 上输出高电平 */
PORTD |= _BV(2) | _BV(3);
/*在 PD2 和 PD3 上输出低电平 */
PORTD &= ~(_BV(0) | _BV(1));

/* 将 PD4、PD5 和 PD6 设置为输入状态 */
DDRD &= ~(_BV(4) | _BV(5) | _BV(6));
/* 开启 PD4、PD5 和 PD6 的上拉电阻 */
PORTD |= _BV(4) | _BV(5) | _BV(6);

/* 利用 PD4:5 读取外部信息，
* 使用 PD0:3 输出选通信号实现一个 24 译码器，
* PD6 作为使能信号，低电平有效
*/
while(1)
{
    /* 等待低电平使能信号 */
    while ( PIND & _BV(6))
    {
        /* 当使能信号为高电平的时候，表示 24 译码器不工作，输出引脚应伽处于高阻态 */
        DDRD &= ~(0x0F);   //或者 DDRD &= ~(_BV(0) | _BV(1) | _BV(2) | _BV(3));
        PORTD &= ~(0x0F);
    }
    DDRD |= 0x0F;
```

```
    //读取 PD4:5
    switch (PIN & (_BV(4) | _BV(5)))
    {
        case 0x00:                                  //PD5:4   [0][0]
            PORTD = (PORTD | 0x0F) & ~_BV(0);       //PD3:0   [1][1][1][0]
            break;
        case _BV(4):                                //PD5:4   [0][1]
            PORTD = (PORTD | 0x0F) & ~_BV(1);       //PD3:0   [1][1][0][1]
            break;
        case _BV(5):                                //PD5:4   [1][0]
            PORTD = (PORTD | 0x0F) & ~_BV(2);       //PD3:0   [1][0][1][1]
            break;
        case _BV(4)|_BV(5):                         //PD5:4   [1][1]
            PORTD = (PORTD | 0x0F) & ~_BV(3);       //PD3:0   [0][1][1][1]
            break;
    }
}
```

3.2.3.2 虚拟的开漏输出

```
# define _BV(__VAL)                 (1 << (__VAL))
……

/* 在 PB3 上以虚拟开漏输出的方式输出方波 */
/* 初始化开漏高电平：进入输入模式，关闭上拉电阻，实现高阻态输出 */
DDRB &= ~_BV(3);
PORTB &= ~_BV(3);

while(1)
{
    /* 以模拟开漏的形式输出方波 */
    DDRB |= _BV(3);                 /* 输出低电平 */
    NOP();                          /* 延迟一个周期 */
    DDRB &= ~BV(3);                 /* 开漏输出高电平 */
}
/* 上面的代码还有一个等效的简化形式：
while(1)
{
    DDRB ^= _BV(3);                 /* 在输入输出模式间切换 */
}
```

```
*/
```

3.2.4 注意事项

- 引脚电平的读取

当引脚方向由输入切换为输出后，外部电平需要额外的 **0.5** 至 **1.5** 个系统周期才能通过对应的 **PINx** 寄存器被读取。因此在进行端口方向切换后、第一次读取读取引脚电平信息前，请插入一个额外的系统等待周期。例如：

```
unsigned char chPinValue = 0;
/* 在 GCC 下使用嵌入式汇编插入一个额外的等待周期 */
#define NOP()                    __asm__ __volatile__ (“nop”)

/* 一个通道切换后读取引脚电平的例子 */
DDRD &= ~_BV(3);                                    //切换通道方向
NOP();                                              //延迟一个周期
chPinValue = PIND;                                  //读取 PINx 电平
```

- **ADC** 端口

ADC 端口是独立供电的。如果 **AVCC** 未供应电源，**ADC** 引脚作为普通引脚时，将无法正常的输出电平。当 **ADC** 引脚作为模拟引脚使用时，允许用户通过寄存器 **DIDR0** 屏蔽对应引脚的数字电平输入功能。**ATmega48/88/168** 的 **ADC6** 和 **ADC7** 只有模拟输入功能。当 **ADC** 引脚的数字电平输入被屏蔽后，**PINx** 寄存器的对应位将只能读取到“**0**”。

3.3 学生用书及幻灯片注解

3.4 常见课堂问题 FAQ

3.5 参考资料

3.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168。**

  >> 13. I/O-Ports

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

3.5.2 应用手册（Application Note）

- **AVR240: 4 × 4 Keypad - Wake-up on Keypress**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1232.pdf**

- **AVR241: Direct driving of LCD display using general IO**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2569.pdf**

- **AVR243: Matrix Keyboard Decoder**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2532.pdf**

- **AVR340: Direct Driving of LCD Using General Purpose IO**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc8103.pdf**

3.5.3 参考文献

3.6 背景知识

3.6.1 C/C++语言

本章的内容牵涉到 **C/C++**语言中位运算的相关内容。例如：与运算“**&**”、或运算“**|**”、非运算“**~**”、异或运算“**^**”、左移运算“**<<**”和右移运算“**>>**”。

3.6.2 模拟电路

本章的内容牵涉到模拟电路中关于 **TTL** 电平、**CMOS** 电平的相关内容。例如：开漏输出、开集输出、上拉、下拉、推挽输出、线与等等。

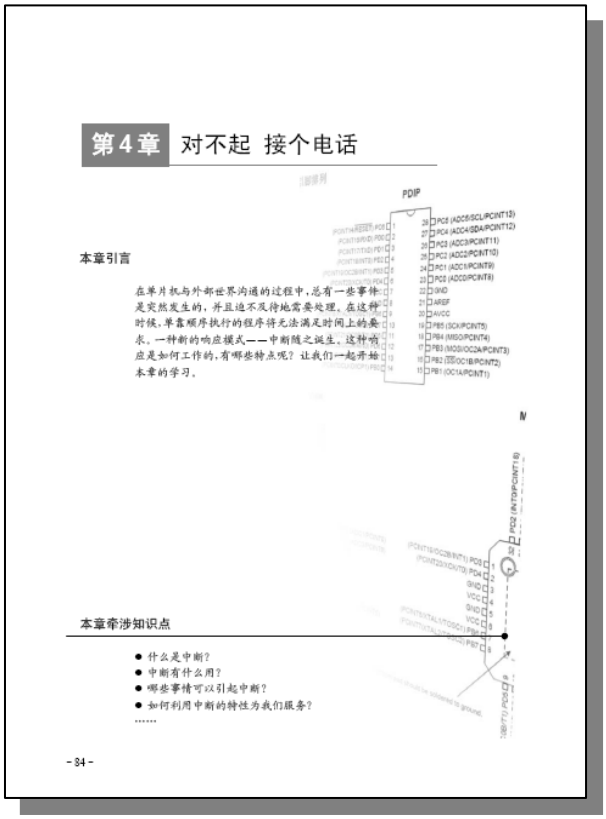
3.6.3 数字逻辑与电路

本章内容牵涉到数字电路中关于电平输入输出、数字逻辑与运算、真值表、卡洛图等相关知识。其中使用端口实现数字显示的内容还牵涉到段码译码电路的设计。**N×M** 矩阵键盘的扫描牵涉到组合时序逻辑电路设计的相关内容。

3.6.3 微机原理

本章内容牵涉到微机原理中关于总线地址空间中关于端口输入输出操作的相关内容。

3.7 参考设计与实验方法


第4章 对不起 接个电话

本章引言

在单片机与外部世界沟通的过程中，总有一些事情是突然发生的，并且迫不及待地需要处理。在这种时候，单靠顺序执行的程序待无法满足时间上的要求。一种新的响应模式——中断随之诞生。这种响应是如何工作的，有哪些特点呢？让我们一起开始本章的学习。

本章牵涉知识点

- 什么是中断？
- 中断有什么用？
- 哪些事情可以引起中断？
- 如何利用中断的特性为我们服务？
- ……

- 84 -

第四章　中断与外中断

Rev 1.0.0.1

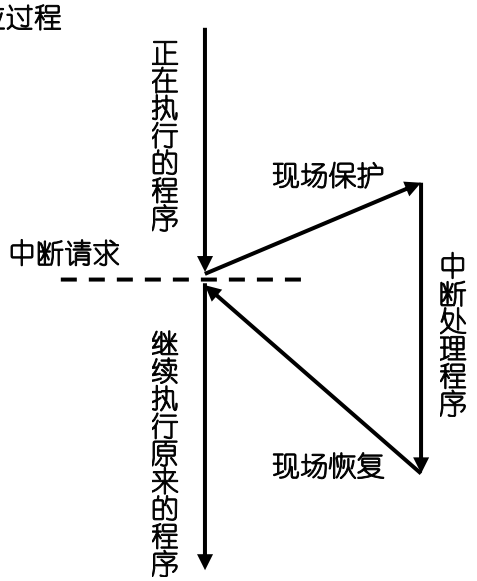
4.1 内容简介

- 中断系统简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

4.2 中断系统简介

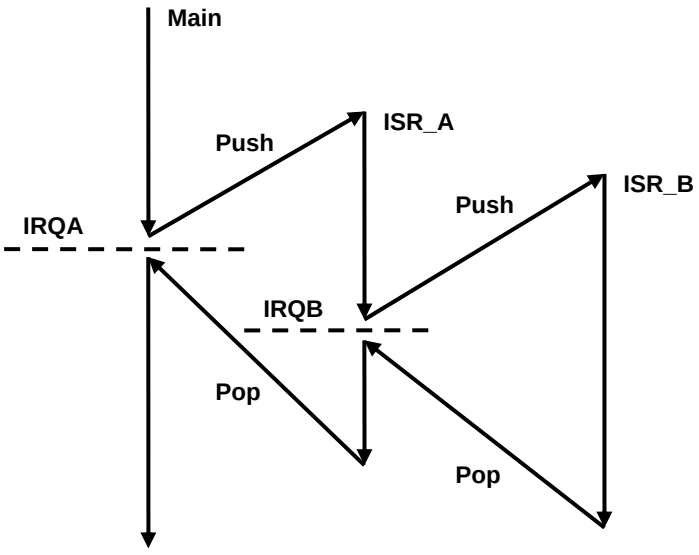
中断是指当一个事件（**Event**）发生时，计算机暂时搁置当前正在执行的代码，自动执行指定的事件处理程序；当事件处理完毕后再回到先前的位置继续运行的过程。中断发生时，计算机暂时搁置当前执行的代码并保存当前必要的环境信息的过程，称为现场保护；事件处理完毕以后，计算机将先前保存的环境信息加以恢复的过程称为现场恢复，一个典型的中断响应过程如**图 4.1** 所示。

图 4.1 一个典型的中断响应过程



如果在中断处理程序执行的过程中又出现了中断请求，并且系统被允许对该中断进行响应，中断嵌套就发生了。中断嵌套的层数受到系统堆栈大小的限制。这与函数递归层数的限制是类似的。**图 4.2** 展示的就是一个中断嵌套发生时典型的系统执行流程。

图 4.2 一个中断嵌套的例子



当多个中断请求同时发生时，系统如果按照一定的优先顺序响应中断，那么这里的优先顺序就被称为中断优先级。当发生中断请求时，系统可以事先选择不响应，这一过程被称为中断屏蔽。当某一优先级的中断发生时，系统可以事先选择不响应，则称为针对某一中断优先级的屏蔽。

4.2.1 名词解释

- **中断源（Interrupt Request Source）**

中断源是指能够向 **MCU** 发出中断请求信号的部件或设备。来自 **MCU** 内部设备的中断源称为内部中断源；来自 **MCU** 外部的中断源称为外部中断源，有时简称外中断。

- **硬件中断和软件中断**

由硬件产生的中断请求，称为硬件中断请求；由软件产生的中断请求，称为软件中断请求。

- **事件系统（Event System）**

以中断响应的方式处理来自内核、外设和 **MCU** 外部各种事件请求的系统，统称为事件系统。通常意义上的中断系统是指处理来自外设和 **MCU** 外部请求的事件系统，是事件系统的子集。

- **异常（Excption）**

来自系统内核的事件称为异常，对应的处理系统称为异常处理系统。

- **中断屏蔽（Interrpt Mask）**

阻止响应中断请求的过程称为中断屏蔽。通常在每个中断系统中都有一个全局中断屏蔽标志，用于屏蔽整个中断系统；在具体的中断源设备中，通常还有一些独立的中断屏蔽标志，用于屏蔽外设内部的某些中断请求。

- **不可屏蔽中断（None Maskable Interrupt）**

无法通过任何手段屏蔽的中断请求，称为不可屏蔽中断。这类中断在某些 **MCU** 中存在。

- **中断临界区（Interrpt Critical Area）**

局部屏蔽中断响应的区域称为中断临界区。中断临界区中代码所执行的操作称为原子操作（**Atom Operation**）。一般情况下，中断处理程序都会被系统自动设置为中断临界区，用以防止中断嵌套的发生。

- **中断嵌套**

在中断处理程序中再次响应中断请求的过程称为中断嵌套。对 **SRAM** 有限的单片机来说，中断嵌套是危险的，一般情况下应该谨慎的应对。如果不是必须，应该尽可能在这类系统中避免中断嵌套的发生。

● **中断优先级（Interrupt Priority）**

当多个中断请求同时发生时，系统优先响应中断请求的顺序，叫做中断优先级。中断优先级有固定和可变两种类型。一般情况下，高优先级中断发生时可以打断低优先级中断处理程序的执行；低优先级中断无法打断高优先级中断处理程序的执行。这只是一般情况，有些单片机的中断系统允许在对应优先级的中断未被屏蔽的情况下打断高优先级中断处理程序的执行。

● **中断向量（Interrpt Vector）**

中断响应时用以指示中断处理程序所在位置的数值信息称为中断向量。不能简单地将中断向量理解为是一个地址。它也可能是一个地址偏移或者是地址索引表中的下标等等。

● **中断处理程序（ISR）**

响应中断请求时执行的程序称为中断处理程序。

4.2.2 AVR 片上中断系统综述

ATmega48/88/168 及其衍生系列芯片提供了丰富的外设资源，与之相适应，也提供了丰富的中断资源。每个中断请求都有一个独立的中断向量，这些中断向量的先后顺序是固定的，决定了中断响应的优先级。所有的中断向量存放在一起形成中断向量表。默认情况下，中断向量表被放置在程序存储器的起始位置，也就是 **FLASH** 存储器的地址 **0x0000**。如果用户通过熔丝配置的方法开启了 **Bootloader** 功能，中断向量表可以通过寄存器 **MCUCR** 使用特定的写入时序放置到 **Bootloader** 熔丝指定的位置。请参考 **ATmega48/88/168** 数据手册或者本书**第十章 Bootloader** 获取更多信息。

4.2.2.1 中断向量

中断向量表的大小由中断向量的数量以及每个中断向量的大小共同决定：

中断向量表的大小 = 每个中断向量的大小 × 中断向量的数量

对 **ATmega48/88** 来说，由于其 **FLASH** 存储器较小，每条中断向量只有 **2** 个字节大小，用于放置一个相对跳转语句 **rjmp**，实现 **2K** 代码范围内的跳转；对 **ATmega168** 来说，其 **FLASH** 存储器空间较大，每个中断向量由 **4** 个字节组成，可以放置一个绝对跳转语句 **jmp**，实现 **16K** 程序空间内的跳转。**ATmega48/88/168** 的中断向量表在顺序上是相同的，如**表 4.1** 所示：

表 **4.1 ATmega48/88/168** 中断向量优先级顺序表

向量编号	中断源	描述
1	RESET	外部、上电、BOD 和看门狗复位
2	INT0	外中断 0 中断请求
3	INT1	外中断 1 中断请求
4	PCINT0	引脚电平变化中断请求 0
5	PCINT1	引脚电平变化中断请求 1
6	PCINT2	引脚电平变化中断请求 2

7	WDT	看门狗超时中断
8	TIMER2 COMPA	定时/计数器 2 比较匹配 A
9	TIMER2 COMPB	定时/计数器 2 比较匹配 B
10	TIMER2 OVR	定时/计数器 2 溢出
11	TIMER1 CAPT	定时/计数器 1 捕获事件
12	TIMER1 COMPA	定时/计数器 1 比较匹配 A
13	TIMER1 COMPB	定时/计数器 1 比较匹配 B
14	TIMER1 OVR	定时/计数器 1 溢出
15	TIMER0 COMPA	定时/计数器 0 比较匹配 A
16	TIMER0 COMPB	定时/计数器 0 比较匹配 B
17	TIMER0 OVR	定时/计数器 0 溢出
18	SPI,STC	SPI 通讯完成
19	USART,RX	USART,接收完成
20	USART,UDRE	USART,数据寄存器为空
21	USART,TX	USART,发送完成
22	ADC	ADC 采样完成
23	EE READY	EEPROM 就绪
24	ANALOG COMP	模拟比较器
25	TWI	2 线串行接口
26	SPM READY	写程序存储区就绪

4.2.2.2 优先级与中断嵌套

ATmega48/88/168 及其衍生芯片提供 **26** 个中断优先级，每个中断源对应一个中断优先级。如**表 4.1** 所示，向量编号越小中断优先级越高。当多个中断请求同时发生时，高优先级的中断请求将被响应。系统会在响应中断时，关闭位于系统状态寄存器 **SREG** 的全局中断响应标志（清“**0**”），此时即便发生更高优先级的中断也无法得到响应。在中断处理程序中，用户可以通过汇编指令 **sei** 人为的开启全局中断响应，此时如果发生任何中断请求，无论其优先级高低都将导致中断嵌套的发生。

4.2.2.3 中断响应

系统按照下面的时序来响应一个中断请求：

a、 清除系统状态寄存器 **SREG** 的全局中断响应标志 **I**——禁止响应其它中断；

b、 将被响应中断的中断请求标志清零；

c、 将 **PC** 指针压入硬件堆栈中（**SP** 寄存器会自动减 **2**）；

d、 根据中断向量压入 **PC** 指针中，系统将从向量指定的位置开始执行；

中断返回的过程与之相反：

a、 从硬件堆栈中取出栈顶元素送入 **PC** 指针中（**SP** 寄存器会自动加 **2**）；

b、 将系统状态寄存器 **SREG** 的全局中断响应标志置位——允许响应其它中断。

4.2.3 AVR 片上外中断综述

ATmega48/88/168 及其衍生系列支持两个外部中断，分别通过引脚 **INT0** 和 **INT1** 输入。**INT0** 和 **INT1** 并行于 **GPIO** 的普通电平输入输出功能存在，根据引脚上的实际电平状态触发中断请求，不会影响普通端口操作。当系统处于休眠模式时，可以被工作于异步模式下的外中断唤醒。

系统支持 **4** 种电平中断触发模式，分别为：低电平触发、任意边沿触发、下降沿触发和上升沿触发。电平的触发模式可以通过寄存器 **EICRA** 来设置。当中断输入引脚 **INTn** 上发生了满足设置的电平变化时，中断标志寄存器 **EIFR** 对应的中断标志将被置位。如果此时寄存器 **EIMSK** 中对应外中断使能标志被置位，并且全局中断响应处于开启状态，系统将根据中断向量执行对应的中断处理程序；**EIFR** 中的中断请求标志将在执行中断处理程序时被自动清除，也可以通过对应位写“**1**”的方法实现清零。

4.2.4 AVR 片上引脚电平变化中断综述

ATmega48/88/168 支持 **3** 组共 **23** 个引脚电平变化中断，分别对应端口 **PB**、**PC** 和 **PD**。其中，除因为引脚 **PC7** 并不存在，**PCINT15** 空缺外，每个端口都有 **8** 个引脚电平变化中断。简而言之，每个 **GPIO** 都能触发引脚电平变化中断：引脚上任意的电平变化（包括上升沿和下降沿）都将触发中断，并且触发中断的边沿是不可选择的。

ATmega48/88/168 支持三组引脚电平变化中断，分别对应编号 **0~3**。除组 **1** 外，其余两组内都是 **8** 个引脚共享一个中断向量。例如，组 1 的中断向量对应 **PCINT0~PCINT7**；组 **2** 对应 **PCINT8~PCINT14**；组 **3** 对应 **PCINT16~PCINT23**。寄存器 **PCMSKn** 用于控制组内哪些引脚可以触发该组的中断请求。每个组内的引脚电平变化中断请求之间都是逻辑或的关系：只要任意一个通过寄存器 **PCMSKn** 使能的引脚上发生了电平变化，整个组都将触发中断请求。在中断处理程序中，无法简单的辨别是组内哪个使能了的引脚触发了中断。

对三个引脚电平变化中断组来说，寄存器 **PCICR** 用于它们的中断使能控制。当某一组触发了中断请求，寄存器 **PCIFR** 中将有对应的标志 **PCIFn** 被设置，如果此时 **PCICR** 中使能了该组的中断响应，并且全局中断响应处于开启状态，系统将执行中断向量所对应的处理程序，系统将自动清除 **PCIFR** 中的中断请求标志，否则可以通过软件置位的方法实现清零。

4.2.5 外中断和引脚电平变化中断引脚配置

ATmega48/88/168 的外中断和引脚电平变化中断都是根据引脚的实际电平情况来触发中断请求的，只要引脚上发生了任何符合要求的电平变化，并且此时全局中断响应处于开启状态将立即触发中断。外中断控制器直接从引脚进行电平采样，使能了外中断或引脚电平变化中断以后，并不会对 **GPIO** 的普通电平输入输出功能造成任何影响。对引脚电平变化中断来说，对应的引脚甚至可以独立于引脚的其它外设功能工作。例如当 **PB2** 作为 **SPI** 从机模式下的 **SS** 控制信号输入引脚时，我们还可以通过引脚电平变化中断 **PCINT2** 监控 **SS** 引脚的电平变化，产生对应的中断。相关代示例请参考代码范例 **4.2.6.3**。

利用这一特性，我们可以使用外中断模拟出一个软件中断：例如，将 **INT0** 所在的引脚设置为输出状态，上升沿触发外中断；当我们通过 **GPIO** 引脚操作产生一个电平上升沿，将触发 **INT0** 中断。于是，在程序执行的任意时刻，我们就拥有了一个通过 **GPIO** 操作触发的“虚拟软件”中断。相关代示例请参考代码范例 **4.2.6.2**。

一般情况下，我们将引脚设置为输入状态并开启上拉电阻，消除端口悬浮时可能的中断误触发，读取来自外部的电平边沿信息。相关代示例请参考代码范例 **4.2.6.1**。

4.2.6 代码范例

4.2.6.1 普通外中断的使用

```
/* 这段代码演示如何使用外中断 */

/* 位操作宏 */
# define _BV(__VAL)              (1 << (__VAL))

/* 外中断触发方式选择 */
# define INT_LOW_LEVEL            0x00
# define INT_ANY_CHANGE           0x01
# define INT_FALLING_EDGE         0x02
# define INT_RASING_EDGE          0x03

/* 外中断初始化函数 */
void int1_init(void)
{
      /* 初始化 INT0 */
     EICRA = (INT0_FALLING_EDGE << ISC00);          /* 下降沿触发 */
     EIMSK = _BV(INT1);                             /* 使能 INT1 中断 */
}

/* 中断处理程序(使用 GCC)*/
ISR(INT1_vect)
{
     /* 当 INT1 下降沿中断触发时，系统将自动执行这里的代码 */
}
```

4.2.6.2 模拟软件中断

```
/* 这段代码演示如何使用 INT0 来实现一个模拟的软件中断*/

/* 位操作宏 */
# define _BV(__VAL)              (1 << (__VAL))

/* 外中断触发方式选择 */
# define INT_LOW_LEVEL            0x00
```

```
# define INT_ANY_CHANGE          0x01
# define INT_FALLING_EDGE        0x02
# define INT_RASING_EDGE         0x03

/* 触发中断 */
# define TRIGGER_SOFTWARE_INT()          \
                do     \
                {\
                    PORTD &= ~_BV(PD2);\
                    PORTD |= _BV(PD2);\
                }\
                while(0)

/* 中断初始化函数 */
void int0_init(void)
{
    /* 初始化端口 */
    DDRD |= _BV(PD2);
    PORTD |= _BV(PD2);

    /* 初始化 INT0 */
    EICRA = (INT0_RASING_EDGE << ISC00);          /* 上升沿触发 */
    EIMSK = _BV(INT0);                            /* 使能 INT0 中断 */
}

/* 中断处理程序(使用 GCC)*/
ISR(INT0_vect)
{
    /* 当用户通过宏 TRIGGER_SOFTWARE_INT()
       触发中断以后会转入这里执行。请插入你自己的代码
    */
}
```

4.2.6.3 引脚电平变化中断的使用

```
/* 这段代码演示如何使用引脚电平变化中断，并判断是哪个边沿触发的中断 */

/* 位操作宏 */
# define _BV(__VAL)             (1 << (__VAL))

/* 引脚电平变化中断初始化函数 */
void pcint0_init(void)
```

```
{
    /* PCINT8 将触发引脚电平变化中断 1 */
    PCMASK1 = _BV(PCINT2);
    PCICR = _BV(PCINT0);                    //使能引脚电平变化中断端口 0
}

/* 引脚电平变化中断端口 0 中断处理程序（使用 GCC） */
ISR(PCINT0_vect)
{
    if (PINB & _BV(PB2))
    {
        /* 下降沿触发了中断 */
    }
    else
    {
        /* 上升沿触发了中断 */
    }
}
```

4.3 学生用书及幻灯片注解

4.4 常见课堂问题 FAQ

4.5 参考资料

4.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168。**
  - **>> 11. Interrupts**
  - **>> 12. External Interrupts**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

4.5.2 应用手册（Application Note）

- **AVR240: 4 × 4 Keypad - Wake-up on Keypress**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1232.pdf**

- **AVR313: Interfacing the PC AT Keyboard**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1235.pdf**

4.5.3 参考文献

4.6 背景知识

4.6.1 数据结构

本章内容牵涉到数据结构关于栈的相关知识。

4.6.2 编译原理

本章内容牵涉到编译原理关于堆栈管理，栈帧存储器栈式分配的相关知识。

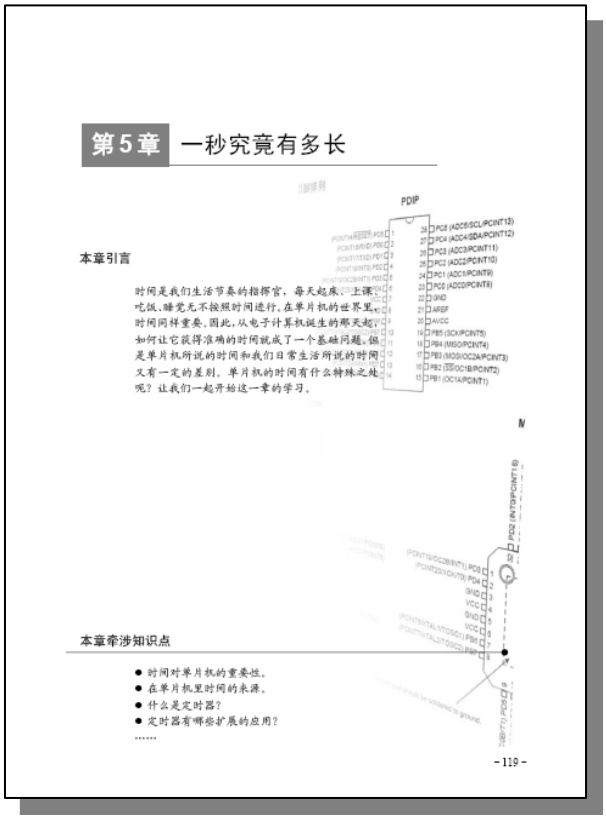
4.6.1 操作系统

本章内容牵涉到操作系统关于存储器管理的相关知识。

4.6.1 微机原理

本章内容牵涉到微机原理中关于中断、中断现场保护、中断现场恢复以及中断优先级的相关知识。

4.7 参考设计与实验方法


第5章 一秒究竟有多长

本章引言

时间是我们生活节奏的指挥官，每天起床、上课、吃饭、睡觉无不按照时间进行。在单片机的世界里，时间同样重要。因此，从电子计算机诞生的那天起，如何让它获得准确的时间就成了一个基础问题。但是单片机所说的时间和我们日常生活所说的时间又有一定的差别。单片机的时间有什么特殊之处呢？让我们一起开始这一章的学习。

本章牵涉知识点

- 时间对单片机的重要性。
- 在单片机里时间的来源。
- 什么是定时器？
- 定时器有哪些扩展的应用？

……

-119-

第五章　定时计数器

Rev 1.0.0.0

5.1 内容简介

- 定时计数器简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

5.2 定时计数器简介

定时/计数器（**Timer/Counter**）是单片机系统中最重要的核心外设。作为可编程的数字逻辑器件，单片机无论是作为微控制器还是微处理器都离不系统对时间的精确把握。定时计数器的用途非常广泛，通常用于计数、延时、测量信号的周期、频率、脉冲宽度以及输出指定的脉冲信号。

定时计数器的本质是计数器（**Counter**），当时钟信号通过一定的分频（或倍频）措施以后作为计数器的计数输入时（**Input Source Clock**），就可以被称为定时器（**Timer**）。当使用外部引脚的电平变化作为计数输入时，计数器主要用来记录外部事件的发生次数。

根据计数器的计数方向划分，有单向计数（**Sigle-Slope**）和双向计数（**Double-Slope**）两种。其中，单向计数是指计数器按照设定的方式单向递增或单向递减；双向计数是指计数器首先递增，当到达某一个上限时更改计数方向为递减，达到计数的下限再次改为递增的计数方式。

计数器的位数定义了其可表达的最大和最小数值范围，一般用**MAX**和**MIN**来表示。对于一个 **8** 位长度的计数器来说，其最大值为 2^8**-1**，也就是十六进制数字 **0xFF**、最小值为 **0**；对于一个 **16** 位的计数器来说，其最大值为 2^{16}**-1**，也就是 **0xFFFF**，最小值为 **0**。

在技术范围内，计数器通常还有一个计数上限和计数下限，分别用 **TOP** 和 **BOTTOM** 表示。无论计数器的实际位数有多少，一旦定义了计数的上限和下限，计数器就被要求在 **TOP** 和 **BOTTOM** 之间计数。对于某些情况，比如在计数器工作时突然将 **TOP** 值设定为低于当前计数值的大小，往往会导致意想之外的结果。具体情况，需要参考外设的相关说明。一般情况下 **BOTTOM** 值就等于计数器的最小值 **MIN**。

为了实现某些功能，除了用于计数的寄存器外，通常还配有一些辅助功能寄存器，用以配合计数器产生特定的事件，甚至产生对应的中断信号。常见的事件有：计数器溢出中断（**Overflow Event**）、计数器比较匹配中断（**Match Event**）、计数器捕获中断（**Capture Event**）

5.2.1 名词解释

- **上溢(Overflow)**

超出计数器所能表示的最大值的事件，称为上溢。一般发生上溢后，计数值在逻辑上会自动归为 **0** 或者最小值 **MIN**。有时候，将上溢和下溢统称为溢出**(Overflow)**。

- **下溢(Underflow)**

超出计数器所能表示的最小值的事件，称为下溢。一般发生下溢后，计数值在逻辑上会自动变为为最大值 **MAX**。

- **比较匹配(Compare Match)**

当计数器的数值与指定寄存器的数值相同时，称为比较匹配。

- **捕获(Capture)**

从计数器开始计数起，第一次捕捉到指定的外部事件，例如某一信号量的跳变等，称为捕获。当捕获事件发生时，计数器中的数值通常被用来计算与捕获到的目标事件有关的信息。例如，使用捕获来测量指定脉冲的信号的频率和周期（用信号中每个上升沿之间的

计数值来计算)。

● 脉冲宽度调制(PWM)

PWM 是脉冲宽度调制（**Pulse Width Modulator**）的缩写。**PWM** 是一个连续的方波，在一个周期中其高低电平所占事件的长度是不同的。高电平事件与整个波形周期的比值被称为占空比。**PWM** 波经过积分器后，可以得到一个介于 **VCC** 和 **GND** 之间的电压。因此，在实际应用中，常常利用 **PWM** 波实现数字信号到模拟信号的转换（**D/A**）。一个 **PWM** 波的参数有频率（**Frequence**）、占空比（**Duty Cycle**）和相位（**Phase**）。

5.2.2 片上 Timer/Counter 简介

ATmega48/88/168 拥有 **3** 个片上定时/计数器。其中定时器 **0** 和定时器 **2** 为 **8** 位定时/计数器，定时器 **1** 位 **16** 位定时/计数器。三个定时器都可以使用系统时钟经过分频后作为计数时钟源。其中定时/计数器 0 和定时/计数器 **1** 可以分别使用引脚 **T0** 和 **T1** 引入外部计数时钟，但时极限钟频率不应该超过系统时钟的 **1/2**。为保证计数器稳定工作，系统时钟频率最好大于外部时钟频率的 **2.5** 倍。定时/计数器 **2** 可以连接外部 **32KHz** 石英晶体振荡器，以获取精确的秒时钟，经过 **32** 分频后成为实时时钟计数器。

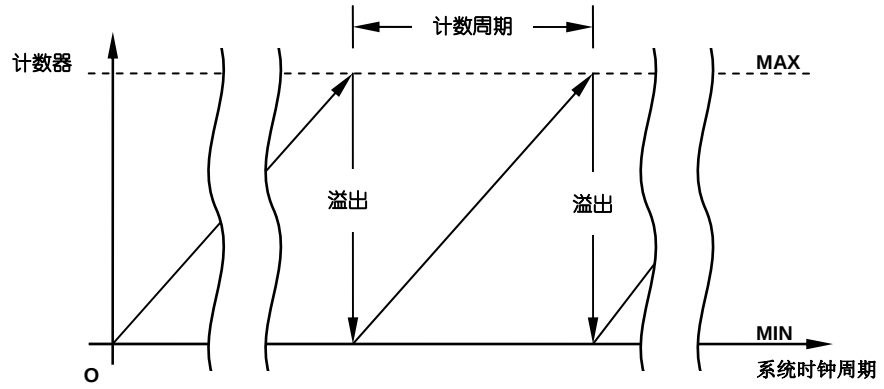
定时/计数器 **0**、定时/计数器 **1** 和定时/计数器 **2** 都有比较匹配单元，可以产生比较匹配中断事件，并根据需要输出脉冲信号（包括 **PWM** 信号）。其中定时/计数器 **1** 可以产生 **2** 路 **16** 位工作于相位频率修正模式下的精确 **PWM** 信号；定时/计数器 **0** 和定时/计数器 **2** 总共可以产生 **4** 路 **8** 位 **PWM** 信号。计数器比较匹配清零模式（**CTC,Clear Timer on Compare match**）可以触发精确时间间隔的比较匹配事件，通常用以产生指定频率的方波，也可以用于驱动精确的系统时钟节拍计数器（**SystemTimerCounter**），实现精确软件延时等功能。快速 **PWM**（**Fast PWM**）通常用于 **D/A** 转换等高频率 **PWM** 应用场合。基于双斜坡的相位修正模式（**Phase Correct PWM Mode**）和相位频率修正模式（**Phase and Frequence Correct PWM Mode**）适合电机控制、舵机控制信号发生等需要高精度 **PWM** 的应用场合。16 位定时/计数器 1 具有输入捕获功能，可以根据 **ICP1** 引脚上的脉冲信号产生捕获事件，并自动记录此时计数器的计数值。该功能通常用于脉冲频率的测量、红外解码等需要对信号时间参数精确测量的场合。

5.2.3 基本计数器

基本的定时/计数器函数模型如图 **5.1** 所示：计数器根据输入系统时钟从最小值 **MIN** 开始向最大值 **MAX** 单向计数，当计数值超过最大值时发生溢出事件（**Overflow**），系统可以根据这一事件产生中断，称为定时/计数器溢出中断（**Timer/Counter Overflow Interrupt**）。定时/计数器从最小值 **MIN** 累加到最大值 **MAX** 所花费的时间称为计数周期 **T**，以系统时钟周期为单位。当最小值 **MIN** 为 **0** 时，计数周期 **T** 与(**MAX+1**)的比值被称为定时/计数器预分频比（**Prescale**），简单说来，就是定时/计数器使用系统时钟作为计数输入时，每多少个系统时钟触发一次计数。通常，预分频比为 2 的整数次幂，例如 **1**、**8**、**64**、**256**、**1024** 等等。如果使用外部计数时钟源，计数器将直接被驱动，没有预分频的概念。

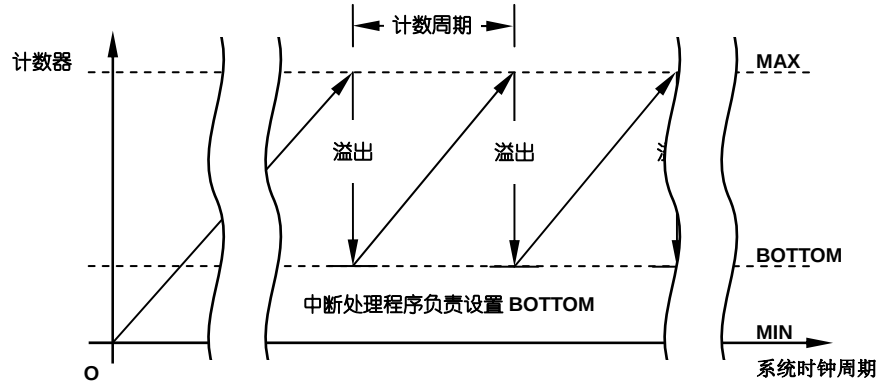
使用系统时钟驱动计数器时，一旦固定了预分频比，就确立了计数周期。我们可以利用定时/计数器溢出中断，在事件发生时预先给计数器一个初值，从而改变计数周期。由于

图 5.1 AVR 基本定时/计数器函数模型示意图



中断处理程序本身的执行过程存在一定的延迟；或者由于我们在定时/计数器溢出中断处理程序的起始位置开启了全局中断响应，在系统重置计数器初值之前有可能被其它中断源打断，计数器已经有了长短不定的计数，从而导致计数周期不稳定。我们把这种使用溢出中断进行定时的计数方式称为**欠精确定时器**。（如图 **5.2** 所示）相关代码示例，请参考 **5.2.9.1** 小节。

图 5.2　欠精确定时器函数模型示意图



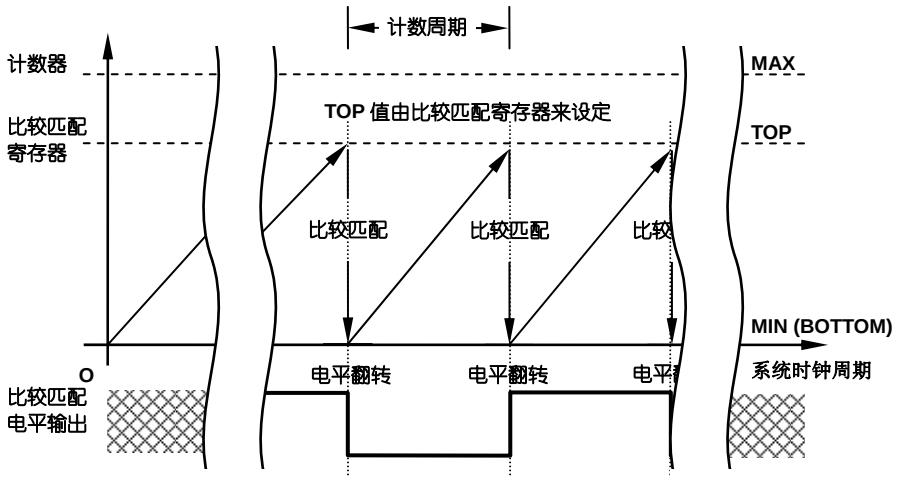
习惯上，**AVR** 使用 **TCNTn** 来命名定时/计数器保存计数值的寄存器，例如定时计数器 **0** 的计数寄存器称为 **TCNT0**，定时/计数器 **1** 的称为 **TCNT1**。在随后的讲解中，如果没有特别说明，我们默认使用定时器（**Timer**）来指代定时/计数器（**Timer/Counter**）。

5.2.4 精确定时器

为了使定时器工作时拥有一个固定的计数周期，**AVR** 提供了一个被称为**比较匹配后计数器自动清零**的工作模式（**CTC**，**Clear Timer on Compare match**）。在该模式下，计数器从最小值 **MIN** 开始向一个事先通过比较匹配寄存器 **OCRnx** 设置的 **TOP** 值单向计数。当定时器计数值与 **OCRnx** 中的内容相等时，触发比较匹配事件（**Compare Match**），计数器自动清零，重新从最小值 **MIN** 开始向 **TOP** 值计数。AVR 允许用户利用比较匹配时间触发中断，由于比

较匹配并清零的动作是由硬件自动完成的，因而计数周期恒定，定时器比较匹配中断的时间间隔也是恒定的。使用 **CTC** 模式实现定时功能的计数器，称为精确定时器。（如**图 5.3** 所示）相关代码示例，请参考 **5.2.9.2** 小节。

图 5.3　CTC 定时/计数函数模型示意图



AVR 定时器允许用户根据 **CTC** 模式下的比较匹配事件在固定的引脚上输出方波信号。习惯上，比较匹配寄存器使用 **OCRnx** 来命名。其中 **OCR** 是输出比较寄存器 **Output Compare Register** 的英文缩写，**n** 表示定时器的编号，**x** 则用于指定具体使用定时器的哪个通道来输出波形，使用 **A**、**B**、**C** 来替换。

AVR 定时器允许用户使用 **OCRnA**（对于定时器 **1** 来说还包括 **ICR1**）来指定 **CTC** 模式下计数器的 **TOP** 值。在定时器运行时刻，动态修改 **TOP** 值将立即生效。注意，如果此时 **TOP** 值比当前的计数值小，计数器将会一直递增，直到溢出并最小值开始重新计数时才有机会触发下一次比较匹配。**CTC** 模式下，输出引脚 **OCnx** 只有在比较匹配输出模式为翻转（**COMnx1/COMnx0** 为 **0/1**, **Toggle**）时才有波形输出，否则将为长低（**COMnx1/COMnx0** 为 **1/0**）或常高（**COMnx1/COMnx0** 为 **1/1**）。如果 **OCnx** 引脚的方向寄存器 **DDRx** 没有被设置为输出状态，也不会有任何波形发生。定时器 **CTC** 模式的具体设置请参考数据手册。

5.2.5 输入捕获

定时器 **1** 支持一路输入捕获功能。系统会根据用户的设置以 **ICP1** 引脚上的跳变延（上升沿 **Raising Edge** 或下降沿 **Falling Edge**）来触发输入捕获事件。当输入捕获中断被使能时，输入捕获中断处理程序将被自动执行，输入捕获中断标志自动清零；用户也可以通过手动向寄存器 **TIFR1** 的 **ICF1** 位写“**1**”实现清零。当输入捕获事件触发时，**TCNT1** 的内容将被复制到输入捕获寄存器 **ICR1** 中（**Input Capture Regster**）。如果用户没有及时读取寄存器 **ICR1**，其内容将会在新的输入捕获事件发生时被覆盖。

定时器 **1** 为输入捕获提供了一个可选的噪声抑制功能，通过向寄存器 **TCCR1B** 中的 **ICNC1** 标志置位开启这一选项。噪声抑制器会对同一个电平信号进行额外的 **4** 次采样，这将在有效的触发边沿发生时引入 **4** 个系统时钟周期的延时。寄存器 **TCCR1B** 的 **ICES1** 标志用于选择输入捕获事件的触发边沿，向其置位表示选择上升沿触发；清零表示下降沿触发。

输入捕获功能可以用于测试 **ICP1** 引脚上的脉冲频率，或者脉冲低电平/高电平的宽度。相关的示例代码请参考 **5.2.9.3** 小节和 **5.2.9.4** 小节。

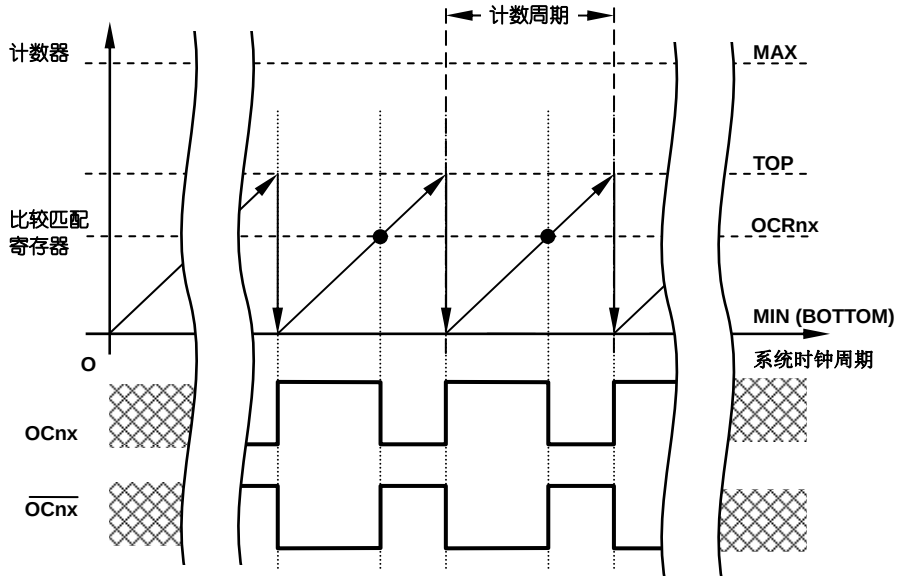
5.2.6 快速 PWM

快速 **PWM** 模式与 **CTC** 模式类似，只不过在快速 **PWM** 模式中，**TOP** 值即可以被配置为固定的常量 **0xFF**（对定时器 **1** 来说还包含 **0x01FF** 或 **0x03FF**），也可以通过比较匹配寄存器 **OCRnA**（对定时器 1 来说还包括 **ICR1**）来设置。快速 **PWM** 模式使用计数器单项递增模式。当计数器值与比较匹配寄存器 **OCRnx** 相等时，对应的波形输出引脚 **OCnx** 可以配置为匹配时输出低电平（计数器变回 **BOTTOM** 时输出高电平）或者比较匹配时输出高电平（计数器变回 **BOTTOM** 时输出低电平）或者比较匹配时翻转当前电平。注意，此时 **OCnx** 引脚方向寄存器应该设置为输出状态。

使用 **OCRnA** 设定 **TOP** 值时，任何针对 **TOP** 值的修改都将在计数器变为 BOTTOM 时生效。对于定时器 **1** 来说，如果使用 **ICR1** 设定 **TOP** 值，将会立即生效。注意，如果此时 **TOP** 值比当前的计数值小，计数器将会一直递增，直到溢出并最小值开始重新计数时才有机会触发下一次从 **TOP** 值到 **BOTTOM** 的跳变。

使用 **OCRnA** 定义计数 **TOP** 值，**OCnA** 输出波形时，只有在电平翻转模式下（**COMnx1/COMnx0** 为 **0/1**, **Toggle**）才会输出波形，否则将根据模式的不同输出常高（**COMnx1/COMnx0** 为 **1/1**）或常低（**COMnx1/COMnx0** 为 **1/0**）。快速 **PWM** 适合输出高频率 **PWM**，一般用于 **D/A** 信号的输出。范例代码请参考 **5.2.9.5** 小节。

图 5.4 快速 PWM 函数模型示意图



5.2.7 相位频率修正模式下的 PWM

ATmega48/88/168 通过定时器 **1** 提供了两路工作于相位频率修正模式下的比较匹配，通常用于产生指定频率和相位的精确 **PWM** 信号。在这一模式下，**OCR1A** 和 **ICR**1 二者之一可以用于定义计数器的 **TOP** 值。与快速 **PWM** 类似，使用 **OCR1A** 定义 **TOP** 值时，其数据会在计数器 **BOTTOM** 阶段被更新；使用 **ICR1** 时，任何针对 **TOP** 值的修改都会立即生效。

如图 **5.5** 所示，相位频率修正模式下，计数器使用的是双斜坡模式，即首先从 **BOTTOM** 向 **TOP** 递增，到达 **TOP** 值后再由 **TOP** 向 **BOTTOM** 递减。由于双斜坡计数模式的使用，相同请况下，该模式的计数周期是普通单斜坡计数模式的两倍。

当我们使用 **ICR1** 定义 **TOP** 值时，任何修改都会立即生效，如果此时 **TOP** 值比当前的计数值小，计数器将会一直递增，直到溢出并最小值开始重新计数时才有机会触发下一次从 **TOP** 值到 **BOTTOM** 的跳变。因此，**ICR1** 更适合使用固定频率 **PMW** 输出的场合。此时，**OCR1A** 和 **OCR1B** 可以用来设定 **OC1A** 和 **OC1B** 输出 **PWM** 的占空比。对于比较匹配，用户可以设定以下三种输出模式：

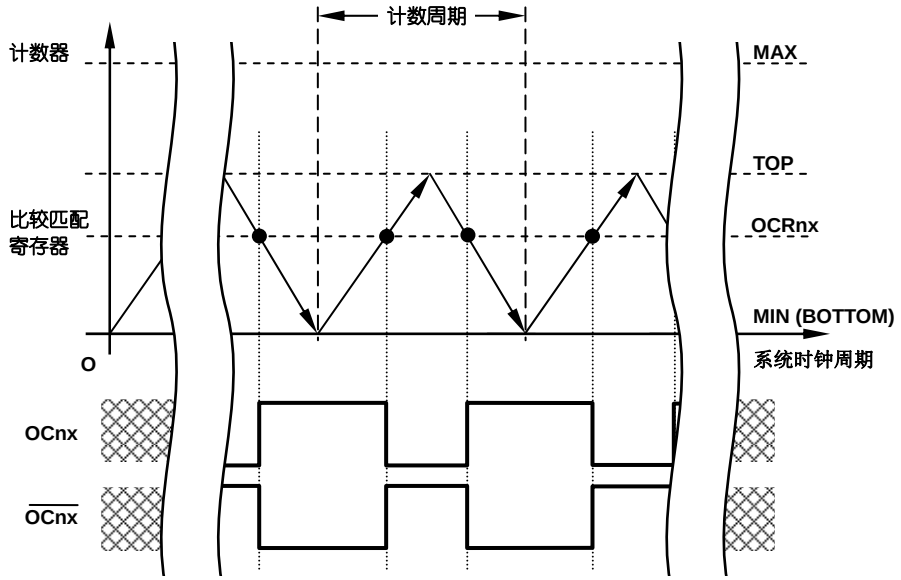
a、 在计数的上升沿，匹配时输出低电平，在下降沿，匹配时输出高电平；

b、 在计数的上升沿，匹配时输出高电平，在下降沿，匹配时输出低电平；

当我们使用 **OCR1A** 定义 **TOP** 值时，任何修改都将在计数器的 **BOTTOM** 阶段更新。由于 **OC1A** 通道始终只能在计数器到达 **TOP** 时发生比较匹配，因而只有在电平翻转模式下（**COMnx1/COMnx0** 为 **0/1**, **Toggle**）才会输出波形，否则将根据模式的不同输出常高（**COMnx1/COMnx0** 为 **1/1**）或常低（**COMnx1/COMnx0** 为 **1/0**）。

相位频率修正模式适合产生指定频率和相位的精确 **PWM**。**5.2.9.6** 小节示例代码以输出指定频率和脉冲数量的列波为例，介绍了该模式的使用方法。

图 **5.5** 相位频率修正 **PWM** 函数模型示意图



5.2.8 相位修正模式下的 PWM

相位修正模式与相位频率修正模式非常类似。其 **TOP** 值可以被设定为固定的常量 **0xFF**（对定时器 **1** 来说还包含 **0x01FF** 或 **0x03FF**）也可以通过 **OCRnA**（对定时器 1 来说还包含 **ICR1**）来设定。在生成 **PWM** 信号时，相位修正模式在 **TOP** 端更新 **OCRnx**；相位频率修正

模式在 **BOTTOM** 端更新 **OCRnx**。这一区别直接导致了前者输出的波形是非对称的，而后者输出的波形是对称的，因而相位频率修正模式更适合电机的控制。

5.2.9 代码范例

5.2.9.1 普通毫秒级系统时钟

```
/* 定时器 0 预分频宏定义 */
# define TMR0_CLK_STOP                  0x00
# define TMR0_CLK_NO_PRESCALE           0x01
# define TMR0_CLK_8_PRESCALE            0x02
# define TMR0_CLK_64_PRESCALE           0x03
# define TMR0_CLK_256_PRESCALE          0x04
# define TMR0_CLK_1024_PRESCALE         0x05
# define TMR0_CLK_T0_FALLING_EDGE       0x06
# define TMR0_CLK_T0_RAISING_EDGE       0x07

/* 定时器 0 初始化程序，用以产生 1KHz 的系统时钟 */
/* 系统时钟 8MHz ,使用 64 分频产生 1Khz 溢出中断 */
void Timer0_INIT(void)
{
    TCCR0B =    (TMR0_CLK_STOP << CS00);              /* 关闭 Timer/Counter0 */
    TCCR0A =    0x00;                                 /* 工作在普通计数模式 */
    TCNT0 =     0x83;                                 /* 预填充值 */
    TIMSK0 =    (1 << TOIE0);                         /* 开启溢出中断响应 */
    TCCR0B =    (TMR0_CLK_64_PRESCALE << CS00); /* 64 分频 */
}

/* 定义一个 16 位系统毫秒时钟 */
volatile unsigned short g_hwSystemTimer = 0;          /* 全局变量 */

/* 定时/计数器 0 溢出中断处理程序 */
ISR(TIMER0_OVF_vect)
{
    TCNT0 = 0x83;                                     /* 重新填充 */
    g_hwSystemTimer ++;
    /* 在这里添加你自己的处理代码，注意不要做过多的操作 */
}
```

5.2.9.2 基于 CTC 模式的精确毫秒系统时钟

```
/* 定时器 0 预分频宏定义 */
……

/* 定时器 0 初始化程序，用以产生精确的 1KHz 的系统时钟 */
/* 系统时钟 8MHz ,使用 64 分频产生 1Khz CTC 比较匹配中断 */
/* 实际 CTC 工作频率为 0.992Khz，误差恒定 */
void Timer0_INIT(void)
{
    TCCR0B =    (TMR0_CLK_STOP << CS00);              /* 关闭 Timer/Counter0 */
    TCCR0A =    (1<<WGM01);                           /* 工作在 CTC 模式下 */
    TCNT0 =     0x00;
    OCR0A =     0x7D;                                 /* 定义 TOP 值 */
    TIMSK0 =    (1 << OCIE0A);                        /* 开启比较匹配 A 中断响应 */
    TCCR0B =    (TMR0_CLK_64_PRESCALE << CS00); /* 64 分频 */
}

/* 定义一个 16 位系统毫秒时钟 */
volatile unsigned short g_hwSystemTimer = 0;          /* 全局变量 */

/* 定时/计数器 0 比较匹配中断处理程序 */
ISR(TIMER0_COMPA_vect)
{
    g_hwSystemTimer ++;
    /* 在这里添加你自己的处理代码，注意不要做过多的操作 */
}
```

5.2.9.3 使用捕获计算脉冲频率

```
/* 定时器 1 预分频宏定义 */
……
/* 定时器 1 初始化程序，使用输入捕获，并开启噪声抑制 */
/* 系统时钟 8M，为获取高精度，不进行时钟分频 */
void Timer1_INIT(void)
{
    TCCR1B =    (TMR1_CLK_STOP << CS10);       /* 关闭定时/计数器 0 */
    TCCR0A =    0x00;                          /* 普通工作模式 */
    ICP1 =      0x0000;
    TIMSK1 =    (1 << ICIE1);                  /* 开启输入捕获中断 */
    DDRB &=   ~(1 << PB0);                     /* 将 ICP1 引脚设置为输入 */
    PORTB |=    (1 << PB0);                    /* 开启上拉电阻 */
    TCCR1B =    (1 << ICNC1)   |               /* 开启输入噪声抑制 */
```

```
                    (1 << ICES1)    |                   /* 上升沿触发捕获 */
                    (TMR1_CLK_NO_PRESCALE << CS10);
}

volatile unsigned long  g_wPulseWidth = 0;          /* 脉冲周期 */
static volatile unsigned short s_hwLastICP = 0;     /* 辅助变量 */

/* 定时器 1 捕获中断处理程序 */
/* 增量式频率计算 */
ISR(TIMER1_CAPT_vect)
{
    unsigned long wTemp = ICP1;                     /* 记录捕获结果 */
    if (ICP1 <= s_hwLastICP)                        /* 检测是否发生了溢出 */
    {
        wTemp += 0xFFFF;
    }
    wTemp -= (unsigned long)s_hwLastICP;            /* 计算周期 */
    s_hwLastICP = ICP1;                             /* 保存本次的测量数值 */
    g_wPulseWidth = wTemp;                          /* 更新测量结果 */
}
```

5.2.9.4 使用捕获计算低电平宽度

```
/* 定时器 1 预分频宏定义 */
……
typedef unsigned char       BOOL;
# define FALSE              (0x00)
# define TRUE               (!FALSE)

/* 定义捕获电平边沿选择宏 */
# define TRIGGLE_FALLING_EDGE       TCCR1B &= ~(1 << ICES1);
# define TRIGGLE_RAISING_EDGE       TCCR1B |= (1 << ICES1);

/* 定时器 1 初始化程序，使用输入捕获，并开启噪声抑制 */
/* 系统时钟 8M，为获取高精度，不进行时钟分频 */
void Timer1_INIT(void)
{
    TCCR1B =    (TMR1_CLK_STOP << CS10);        /* 关闭定时/计数器 0 */
    TCCR0A =    0x00;                           /* 普通工作模式 */
    ICP1 =      0x0000;
    TCNT1 =     0x0000;
    TIMSK1 =    (1 << ICIE1) | (1 << TOIE1);    /* 开启输入捕获中断 */
```

```
    DDRB &=   ~ (1 << PB0);                          /* 将 ICP1 引脚设置为输入 */
    PORTB |=    (1 << PB0);                          /* 开启上拉电阻 */
    TCCR1B =    (1 << ICNC1)   |                     /* 开启输入噪声抑制 */
                (0 << ICES1)   |                     /* 下降沿触发捕获 */
                (TMR1_CLK_NO_PRESCALE << CS10);
}

/* 公用体，用来将字拆分为两个半字 */
typedef union
{
    unsigned long     Word;
    unsigned short    HalfWord[2];
} WORD_HWORD;

volatile unsigned long       g_wLowWidth = 0x00;   /* 低电平的长度 */
volatile WORD_HWORD    s_LowWidth;                 /* 中间变量 */
volatile BOOL g_bIfStartCount = FALSE;             /* 边沿选择标志 */

/* 定时器 1 捕获中断处理程序 */
/* 使用溢出中断扩展测量范围 */
ISR(TIMER1_CAPT_vect)
{
    if (!g_bIfStartCount)
    {
        /* 截获了下降沿 */
        TRIGGLE_RAISING_EDGE;                      /* 等待上升沿 */
        s_LowWidth.Word = 0;                       /* 初始化计数器 */
        TCNT1 = 0x00;                              /* 初始化硬件计数器 */
        g_bIfStartCount = TRUE;                    /* 设置标志 */
    }
    else
    {
        /* 截获了上升沿 */
        g_bIfStartCount = FALSE;                   /* 设置标志 */
        s_LowWidth.Word += ICP1;                   /* 记录数值 */
        TRIGGLE_FALLING_EDGE;                      /* 等待下降沿 */
        g_wLowWidth = s_LowWidth.Word;             /* 刷新采样结果 */
    }
}

/* 定时/计数器 1 溢出中断处理程序 */
ISR(TIMER1_OVF_vect)
{
```

```
    if (g_bIfStartCount)
    {
        s_LowWidth. HalfWord[1] ++;                    /* 处理溢出进位 */
    }
}
```

5.2.9.5 使用快速 PWM 实现 LED 渐明渐暗闪烁

```
/* 定时器 0 预分频宏定义 */
……
/* 定时器 0 初始化函数，快速 PWM 模式 */
/* 系统时钟 8M，不分频 */
void Timer0_INIT(void)
{
    TCCR0B =    (TMR0_CLK_STOP << CS00);              /* 关闭 Timer/Counter0 */
    TCCR0A =    (1 << COM0A1) | (1 << COM0A0) |       /* 增量计数匹配时输出高电平 */
                (1 << WGM00) | (1 << WGM01);          /* 快速 PWM 模式 */
    TCNT0 =     0x00;
    OCR0B =     0x80;
    DDRD |=     (1 << PD6);                           /* 在 OC0A 引脚输出 PWM */
    TIMSK0 =    (1 << TOIE0);                         /* 开启溢出中断响应 */
    TCCR0B =    (TMR0_CLK_NO_PRESCALE << CS00);/* 不分频 */
}

static unsigned char s_chSINTable[16] =
                        {
                            0x80,0xB1,0xDA,0xF5,0xFF,0xF5,0xDA,0xB1,
                            0x80,0x4F,0x26,0x0B,0x01,0x0B,0x26,0x4F
                        };

/* 定时/计数器 0 溢出中断处理程序 */
ISR(TIMER0_OVF_vect)
{
    static unsigned short s_hwFlashTimer = 0;
    static unsigned char s_n = 0;
    s_hwFlashTimer++;
    if (!(s_hwFlashTimer & ((1 << 9) – 1)))
    {
        OCR0A = s_chSINTable[n++];
        n &= 0x0F;
    }
}
```

5.2.9.6 列波输出

```
/* 该示例代码用于展示如何以指定的频率输出指定数量的脉冲 */
/* 列波输出通常用于控制步进电机驱动芯片驱动步进电机以指定的速度运动指定的步数 */
/* 定时器 1 预分频宏定义 */
……
/* 定时器 1 初始化程序，产生 2K 的列波 */
/* 系统时钟 8M，使用 64 分频，实际频率为 2.016KHz */
void Timer1_INIT(void)
{
    TCCR1B =    (TMR1_CLK_STOP << CS10);          /* 关闭定时/计数器 0 */
    TCCR1A =    (1 << COM1A1);                   /* 比较匹配输出低电平 */
    DDRB |=     (1 << PB1);                      /* OC1A 输出 PWM 波 */
    ICR1 =      0x001F;                          /* 产生 2.016KHz */
    TIMSK1 =    (1 << OCIE1A);                   /* 开启 OCR1A 比较匹配中断 */
    TCCR1B =    (1 << WGM13);                    /* PWM 相位频率修正模式 */
}

volatile unsigned short s_hwPulseCounter = 0;    /* 列波输出计数器 */

/* 定时/计数器比较匹配中断处理程序 */
ISR(TIMER1_COMPA_vect)
{
    if (s_hwPulseCounter)
    {
        s_hwPulseCounter--;
    }
    else
    {
        /* 关闭定时器输出 */
        TCCR1B &= ~((1 << CS10) | (1 << CS11) | (1 << CS12));
    }
}

/* 调用该函数将启动一次列波输出，参数 hwPulseCount 传递列波中脉冲的数目 */
void Output_Pulses(unsigned short hwPulseCount)
{
    s_hwPulseCounter = hwPulseCount;             /* 初始化列波输出计数器 */
    TCCR1B =    (TCCR1B & ~((1 << CS10) | (1 << CS11) | (1 << CS12))) |
                (TMR1_CLK_64_PRESCALE << CS10); /* 开启定时器 1 列波输出 */
}
```

5.3 学生用书及幻灯片注解

5.4 常见课堂问题 FAQ

5.5 参考资料

5.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168。**
  - **>> 14. 8-bit Timer/Counter0 with PWM**
  - **>> 15. 16-bit Timer/Counter1 with PWM**
  - **>> 16. Timer/Counter0 and Timer/Counter1 Prescalers**
  - **>> 17. 8-bit Timer/Counter0 with PWM and Asynchronous Operation**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

5.5.2 应用手册（Application Note）

- **AVR360: Step Motor Controller**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1181.pdf**

- **AVR134: Real Time Clock (RTC) using the Asynchronous Timer**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1259.pdf**

- **AVR133: Long Delay Generation Using the AVR Microcontroller**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1268.pdf**

- **AVR130: Setup and Use the AVR® Timers**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2505.pdf**

- **AVR131: Using the AVR's High-speed PWM**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2542.pdf**

- **AVR055: Using a 32kHz XTAL for run-time calibration of the internal RC**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc8002.pdf**

- **AVR135: Using Timer Capture to Measure PWM Duty Cycle**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc8014.pdf**

- **AVR136: Low-Jitter Multi-Channel Software PWM**

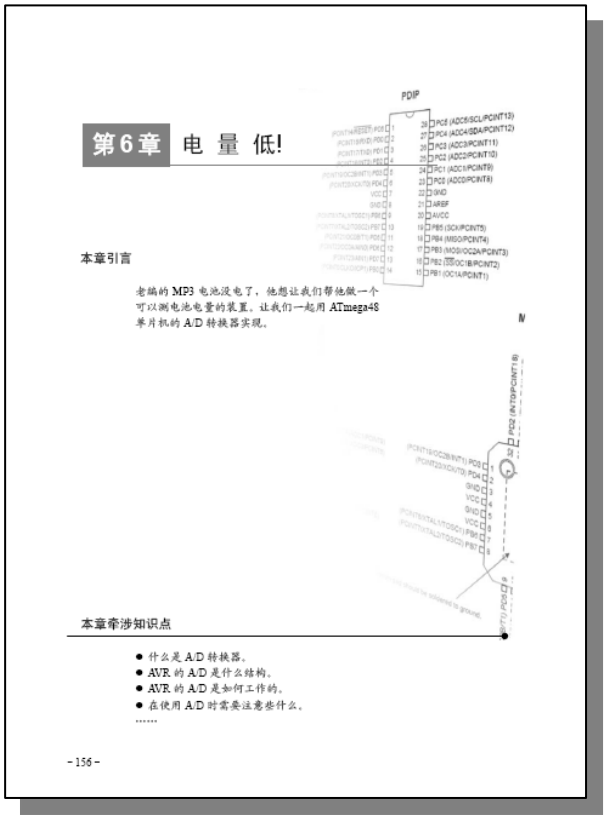
下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc8020.pdf**

5.6 背景知识

5.6.1 数字逻辑与系统

本章内容牵涉到数字逻辑与系统中关于脉冲宽度调制（**PWM**）信号、定时器和计数器的相关内容。

5.7 参考设计与实验方法



第六章　ADC 采样与数字滤波

Rev 1.0.0.2

6.1 内容简介

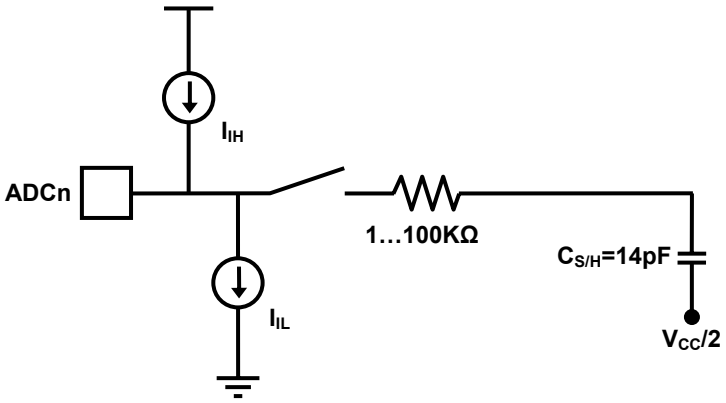
- ADC 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

6.2 ADC 简介

ADC 是模拟到数字转换器 **Analog-to-Digital Convertor** 的英文缩写，如字面上的意思，**ADC** 主要完成模拟信号量到数字信号量的转换工作。通常由模拟信号保持电路、模拟信号转换电路和数字信号处理电路三部分构成。从工作原理上分，**AD** 转换器可以分为直接式 **AD** 转换器和间接式 **AD** 转换器。前者将模拟信号量直接转换为数字信号量，其特点是转换迅速；后者先将模拟信号转换为某种中间信号量（例如，时间或者频率等），再将中间信号量转换为数字信号量，其特点是转换速率较慢，但抗干扰能力较好。从模拟信号的量化原理上分，**ADC** 又可分为并行比较式、逐次逼近式和双积分式等等。

AD 转换器根据指定的参考电压将输入的模拟信号进行数字量化。这一过程中，主要的指标和概念有：转换速率、转换精度、误差等等。**AD** 采样的速率由硬件决定；转换精度由参考电压和转换数值的二进制位数共同决定；而误差则由参考电压的噪声情况、输入信号的噪声情况，**AD** 转换器自身的特性、电路设计、数字滤波算法等因素综合决定。**图 6.1** 展示的是一典型的 **ADC** 采样信号输入保持电路原理图。

图 6.1 一个典型的连接 ADC 采样引脚的输入保持电路



6.2.1 名词解释

● **ADC 采样分辨率**

分辨率以输出二进制或十进制数字的位数表示，它说明 **A/D** 转换器对输入信号的分辨能力。从理论上讲，**n** 位输出的 **A/D** 转换器应能区分出输入电压信号的 **2** 的 **n** 次方个不同等级，每个等级相差为 **2** 的 **n** 次方分之一。

● **量化误差（离散化误差）**

将模拟信号量以有限的数字量进行表示时，某一范围内的模拟信号量都将被转化为相同的数字量，这一范围的最大宽度称为量化误差，又称为离散化误差。理想情况下，该误差应为 **±0.5 LSB**。

● **偏移误差**

偏移误差为名义上的偏移点与实际上的偏移点之间的差。对 **ADC** 而言，偏移点是当

数位输出为零时的阶梯中间值；此误差是以相同的数量改变所有的数位码，通常能够藉由一个修整过程获得补偿。假使无法修正，此错误就是（零标度错误）。理想状况下，该误差应为 **0 LSB**。

● **增益误差**

增益误差是指在偏置误差得到修正后,增益点的理想值与实际值的偏差.AD的增益点是指当数字输出转换到最大值时,模拟输入值加上 **0.5LSB**的值（当数位输出为标度的全部(**full scale**)时，增益点为阶梯中间值）。理想状况下，该误差应为 **0 LSB**。

● **积分非线性度**

积分非线性误差 **(Integral Nonlinearity(INL) Error)** 是当增益误差和偏移误差被消除时，实际的转换函数与理想直线之间的偏差值。这条直线也许是能够降低这些偏差值的最佳直线；或者是连接转换函数首位两端点的直线。积分非线性度使用 **LSB** 作为衡量单位。理想情况下，该误差应为 **0 LSB**。

● **微分非线性度**

对 **ADC** 而言，微分非线性误差 **(Differential Nonlinearity(DNL)Error)** 指一个实际的阶梯宽度和理想的 **1LSB** 阶梯宽度之差。若阶梯宽度或高度正好是 **1LSB**，则微分非线性误差就等于零，倘若 **DNL** 大于 **1LSB**，则转换器有可能成为非单调函数。这表示输入的振幅增加时，输出的振幅会变小，**ADC** 也有可能会遗失数位码，亦即在 **2** 的 **n** 次方个二进制码中，可能有一个或更多个数位码无法被输出。理想情况下，该误差应为 **0 LSB**。

● **绝对精度**

将所有误差因素计入在内，转换得到的数字信号量与实际输入电压信号间的误差，称为绝对采样精度，使用 **LSB** 作为衡量单位。理想情况下，该误差应为 **±0.5 LSB**。

6.2.2 AVR 片上 ADC 综述

ATmega48/88/168 及其衍生系列芯片集成了一个 **10** 位分辨率的 **ADC**，平均 **13~260μs** 的转换时间，在保持最高分辨率的情况下可以达到 **15 kSPS** 的转换速率。**PDIP** 封装下拥有 **6** 路模拟输入；**TQFP** 与 **MLF** 封装下拥有 **8** 路模拟通道。

片上**ADC**支持单次采样模式，也支持根据芯片内部其他模块的标志信号进行采样的自动触发模式。例如，根据定时器比较匹配、溢出标志；根据模拟比较匹配触发等 **8** 种模式。在正常的采样速率下，**ADC**转换的结果可以达到 **0.5 LSB**的积分非线性度以及 **±2 LSB**的绝对精度。**ATmega48/88/168** 提供 **3** 种可选的参考源，分别是：内部 **1.1v**参考电压、**AREF**引脚上的参考电压、**AV_{CC}**上的参考电压。参考源的电压大小决定了 **10** 位采样所能达到的分辨率；参考电源的质量决定了采样结果的准确性和稳定性。值得注意的是，对于具体的单片机来说，参考电压不能低于某一个最小的门限；对应的 **10** 位分辨率所能表达的精度也不可能无限小。以**ATmega48** 为例，其参考电压不能低于 **1.0v**。

6.2.3 端口配置

ATmega48/88/168 的**ADC**端口是独立于其他数字端口单独使用**AV_{CC}**引脚进行供电的。当**AVCC**未连接外部电源时，**ADC**所在引脚的普通**GPIO**功能将受到影响，表现为无法实现高低

电平的正常输出。在**PDIP**封装下，引脚**PC0** 至**PC5** 是**ADC**引脚，当**ADC**功能有效时，对应引脚的**GPIO**功能不受影响，当我们输出高/低电平时对应的采样结果分别为 **0x03FF**和 **0x0000**。一般情况下，我们应该将引脚电平设置为输入模式，并关闭上拉电阻；对寄存器**DIDR0** 的对应二进制位写“**1**”，关闭**PC0~PC5** 上的数字输入缓冲器。此时通过**PINC**寄存器读取对应引脚电平将只能获得“**0**”值。

一个典型的端口配置代码如下：

```
DDRC &= ~(1 << PC0);          //设置 PC0 为输入状态
PORTC &= ~(1 << PC0);         //关闭上拉电阻
```

6.2.4 参考电压和采样通道的选择

ATmega48/88/168 支持三种参考源输入，分别为：内部 **1.1v**参考电压、AV_{CC}参考电压和**AREF**引脚上引入的参考电压。无论选择何种参考源，**AREF**引脚始终与内部的V_{REF}相连，可以通过高阻抗的伏特表测量**AREF**引脚获得当前的实际采样电压信息。在**AREF**引脚与**GND**之间增加一个高频特信好、漏电电流小的电容可以有效地抑制电源噪声。对应**表 6.1** 设置寄存器**ADMUX**的**REFS0** 和**REFS1** 位，可以选择**AD**采样所使用的参考源：

表 6.1 ADC 参考电压选择

REFS1	REFS0	参考电压源
0	0	以AREF作为参考源，内部V_{REF}将被关闭
0	1	以AV_{CC}作为参考源，并需要在AREF与GND之间增加电容
1	0	Reserved
1	1	使用内部的 1.1v 参考电压，并需要在 AREF 与 GND 之间增加电容

ATmega48/88/168 在 **PDIP** 封装的芯片中支持 **6** 路模拟 **ADC** 输入通道，与数字引脚 **PC0~PC5** 复用；在 **TQFP** 与 **MLF** 封装的芯片中支持额外的 **2** 路 **ADC** 专用模拟输入通道，分别是 **ADC6** 和 **ADC7**。通过寄存器 **ADMUX** 的低 **4** 位，我们可以指定一路 **AD** 模拟输入通道。需要注意的是，为了与其它芯片兼容，**ADMUX** 的 **BIT3** 应该在设置 **AD** 通道时始终写“**0**”。

参考源与通道选择的 **C** 语言示例请参考 **6.2.8** 代码范例章节。

6.2.5 单次转换模式

单次转换模式是指 **ADC** 在参考电压和通道设置完毕以后，当用户置位 **ADCSRA** 寄存器的 **ADSC** 标志，将启动一次采样；采样完成时，**ADSC** 标志将自动清零，采样完成标志 **ADIF** 自动置位。如果中断使能位 **ADIE** 为“**1**”，并且系统开启了全局中断响应，那么采样完成中断将被触发，**ADIF** 标志被自动清零，系统执行对应的采样完成中断处理程序；如果 **ADIE** 没有被置位（采样完成中断处于关闭状态），用户可以通过向 **ADIF** 位写“**1**”实现标志的清零。完成了上述步骤之后，**ADC** 回到就绪状态，等待用户启动下一次采样。在整个采样过程中，**ADSC** 标志始终为“**1**”，用户可以通过检测该标志位了解 **ADC** 当前的工作状态。单次转换模式的 **C** 语言代码范例请参考 **6.2.8.1** 小节。

6.2.6 自动触发模式

自动触发模式是指 **ADC** 根据芯片中其它外设的标志信号自动的触发 **AD** 采样。自动触发模式可以通过置位寄存器 **ADCSRA** 的 **ADATE** 位来启动。在此之前应该完成参考源和采样通道的设置，并通过寄存器 **ADSCRB** 的 **ADTS0**、**ADTS1** 和 **ADTS0** 位选择自动采样的触发源。**ATmega48/88/168** 可选的触发源如表 **6.2** 所示：

表 6.2 ADC 自动触发源选择

ADTS2	ADTS1	ADTS0	触发源
0	0	0	连续采样模式(Free Running)
0	0	1	模拟比较器
0	1	0	外中断 0
0	1	1	定时计数器 0 比较匹配 A
1	0	0	定时计数器 0 溢出
1	0	1	定时计数器 1 比较匹配 B
1	1	0	定时计数器 1 溢出
1	1	1	定时计数器 1 捕获事件

自动触发模式的 **C** 语言代码范例请参考 **6.2.8.2** 小节。在该示例中，系统选择 **Free Running** 触发模式，**ADC** 会在一次 **AD** 转换完成之后立即开始下一次的转换，每次转换的结果都以覆盖的方式保存在寄存器 **ADCL** 和 **ADCH** 中。用户通过读取这两个寄存器的值获取最近一次 **AD** 采样的结果。**Free Running** 模式下，需要使用 **ADSC** 标志来触发第一次采样。

6.2.7 休眠模式下的 ADC 噪声抑制唤醒

为了降低来自 **CPU** 的干扰，**ATmega48/88/168** 支持一种在 **CPU** 休眠模式下进行 **AD** 采样并使用中断唤醒 **CPU** 的工作模式，称为 **ADC** 噪声抑制模式。该模式的核心思想就是利用 **CPU** 休眠来降低来自 **CPU** 工作给 **ADC** 采样带来的干扰。**ADC** 噪声抑制模式的使用必须参照以下的步骤进行：

a、 确认 **ADC** 被使能并处于就绪状态（当前没有任何正在进行的转换）；
b、 配置寄存器使 **ADC** 工作在单次转换模式下并使能 **ADC** 采样完成中断；
c、 进入 **CPU** 休眠的 **ADC** 噪声抑制模式（或空闲模式），**ADC** 会在 **CPU** 进入休眠模式以后自动触发采样。
d、 **ADC** 采样完成后会唤醒 **CPU** 并执行对应的中断处理程序。如果在此之前 **CPU** 被其他中断唤醒，系统将在完成对用的中断处理程序后回到正常工作模式下。
e、 执行完上述步骤以后，**CPU** 会保持工作状态，直到再次进入对应的休眠模式。

6.2.8 代码范例

6.2.8.1 单次转换

```
# define _BV(__VAL)                    (1 << (__VAL))
……

/* ADC 初始化函数，使用单次转换模式 */
void adc_init(void)
{
    ADMUX =   (0x01 << REFS0) |          //使用 AVCC 作为参考电压
              /*_BV(ADLAR) |*/           //使用左对齐
              (0x00 << MUX0);            //默认选通 ADC0
    ADCSRA = _BV(ADEN) |                 //使能 ADC
              (0x06 << ADPS0);           //ADC 时钟为系统时钟的 64 分频
}

/* 从指定的通道读取一个 AD 采样数值
* 输入：指定的采样通道
* 输出：采样结果。如果采样失败，返回 -1。
*/
int get_adc_value(unsigned char chChannal)
{
    /* 检查输入的通道编号是否有效 */
    if (chChannal > 7)
    {
        return -1;
    }

    /* 重新设定采样通道 */
    DIDR0 &= ~BV(ADMUX & 0x0F);          // 恢复上一次设定通道的数字输入功能
    ADMUX = (ADMUX & 0xF0) | chChannal;
    DIDR0 |= _BV(chChannal);             //关闭本次使用通道的数字输入功能

    /* 启动单次采样 */
    ADCSRA |= _BV(ADSC);                 /* 注意，这里利用了或操作的特性处理了
                                            自动向 ADSCRA 中 ADIF 标志写 1 以
                                            实现对 ADIF 标志清零的操作。*/
    while (!(ADCSRA & _BV(ADIF)));       // 等待本次采样完成

    return ADC;                          // 返回采样结果
}
```

6.2.8.1 Free Running 模式

```
# define _BV(__VAL)                    (1 << (__VAL))
……

/* ADC 初始化函数，使用 Free Running 模式
* 输入：指定的通道
* 输出：操作是否成功，成功输出 1，失败输出 0
*/
int adc_init(unsigned char chChannal)
{
    /* 检测通道选择是否有效 */
    if (chChannal > 7)
    {
        return 0;
    }

    ADCSRA = 0x00;                                  //关闭 ADC
    while (ADCSRA & _BV(ADSC));                     //等待 ADC 就绪

    ADMUX =  (0x01 << REFS0) |                      //使用 AVCC 作为参考电压
             /*_BV(ADLAR) |*/                       //使用左对齐
             (0x00 << MUX0);                        //默认选通 ADC0
    ADCSRB = (ADCSRB & 0xF0) | (0 << ADTS0);        //设置自动触发模式为 Free Running
    DIDR0 |=  _BV(chChannal);                       //关闭本次使用通道的数字输入功能
    ADCSRA = _BV(ADEN) |                            //使能 ADC
             _BV(ADATE) |                           //启动自动触发模式
             _BV(ADSC) |                            //启动第一次采样
             (0x06 << ADPS0);                       //ADC 时钟为系统时钟的 64 分频

    return 1;
}

/* 获得最近一次 ADC 采样的结果
* 输出：最近一次 ADC 采样结果
*/
int get_adc_value(void)
{
    /* 返回 ADC 采样结果 */
    return ADC;
}
```

6.2.9 注意事项

6.2.9.1 针对 ADCSRA 寄存器操作的陷阱

ADCSRA 是 **Analog-to-Digial Convertor Control and Status Register A** 的英文缩写。在 **ADC** 模块中 **ADCSR**A 同时包含了控制位和状态位 **ADIF**。状态标志 **ADIF** 用于指示一次转换的完成，在中断模式下可以用于触发采样完成中断，并在中断处理程序中被自动清零；在非中断模式下，可以通过向该标志写“**1**”实现清零。使用 **C** 语言操作寄存器时，我们习惯上使用“**|=**”和“**&=**”运算来保护寄存器中的未被操作的二进制位。由于 **ADIF** 标志与其它控制标志同时存在于寄存器 **ADCSRA** 中，当程序使用“**|=**”或“**&=**”操作 **ADSCRA** 的某些控制标志位时，如果此时 **ADIF** 标志已经为“**1**”，则重新写入 **ADIF** 的值也是“**1**”，无意间完成了对 **ADIF** 标志的清零，有可能造成一次采样完成的丢失。因此，一个正确操作 **ADSCRA** 寄存器的代码范例如下：

```
/* 安全操作 ADSCRA 寄存器的宏 */
#define SET_ADCSRA(__VAL)       ADCSRA = ((ADCSRA & ~(1 << ADIF)) | (__VAL));
#define CLR_ADCSRA(__VAL)       ADCSRA = (ADCSRA & ~((1 << ADIF) | (__VAL)));

/* 启动一次采样 */
SET_ADCSRA((1 << ADSC));
……
/* 关闭 ADC 使能 */
CLR_ADSCRA((1 << ADEN));
```

6.2.9.2 ADC 采样结果的读取

ADC 采样完成以后会将 **10** 位数据分别存放在两个 **8** 位寄存器 **ADCL** 和 **ADCH** 中。当寄存器 **ADMUX** 的 **ADLAR** 位为 **0** 时，系统采用默认的右对齐模式：**ADCL** 存放 **10** 位数据的低 **8** 位，**ADCH** 存放 **10** 位数据的高 **2** 位，其余位为 **0**；当 **ADLAR** 为 **1** 时，系统采用左对齐模式：**ADCH** 存放 **10** 位数据的高 **8** 位，**ADCL** 存放数据的低 **2** 位。无论采用何种对齐方式，一旦读取了寄存器 **ADCL**，**ADCH** 将被锁定。在这种状态下，除非 **ADCH** 被读取，所有新的采样数据都将被系统忽略。需要注意的是，如果我们直接读取 **ADCH**，**ADCL** 在读取前有被新采样数据更新的可能。因此，每次针对 **10** 位数据的操作，都应该先读取 **ADCL** 再读取 **ADCH**。当我们直接针对伪 **16** 位寄存器 **ADC** 进行操作时，**GCC** 和 **IAR** 编译器会自动生成正确的读取顺序。伪寄存器 **ADC** 并不真正存在，可以将其理解为由寄存器 **ADCL** 和 **ADCH** 组成，起始地址与 **ADCL** 相同。当不确定编译器是否支持伪寄存器 **ADC** 时，操作 **ADCL** 和 **ADCH** 的范例代码如下：

```
unsigned int wADCValue = ADCL;                  //读取 ADCL
wADCValue |= ((unsigned int)ADCH << 8);         //读取 ADCH
```

6.2.9.3 不同参考源模式下的外部连接

ATmega48/88/168 支持三种参考源模式：内部 **1.1v**参考电压、**AV_{CC}**参考电压和外部**AREF** 引脚引入的参考电压。无论使用哪种参考模式，**AREF**引脚都是直接连接在内部**V_{REF}**上的，因此应该在**AREF**引脚和**GND**之间增加一个高频特性好，漏电电流小的电容，这将有效降低电压基准源的噪声。

使用**AV_{CC}**或**AREF**作为参考源时，应该做好参考源与单片机供电端**V_{CC}**的退偶工作，推荐使用独立的电源供电或者使用**RC**低通滤波器对从**V_{CC}**上引入的电源进行滤波。**注意，当AREF引脚上连接了其它参考源时，请不要选择内部 1.1v参考源。AV_{CC}与V_{CC}之间的电压差应该小于 0.3v。**

6.2.9.4 参考源和通道的安全切换策略

单次转换模式下，应该等待上次转换完成或者第一次转换启动前更改参考源或者切换通道。可以通过检测 **ADCSRA** 寄存器的 **ADSC** 标志是否为“**0**”来判断当前系统状态：**ADSC** 为“**1**”表示转换正在进行；**ADSC** 为“**0**”表示系统处于就绪等待状态，可以安全的更改参考源或切换通道。等待系统就绪的示例代码如下：

```
while (ADSCRA & (1 << ADSC));          //等待系统就绪
/* 在这里添加切换参考源或者通道的代码 */
```

自动触发模式下，应该在关闭 **ADC** 使能标志或者关闭自动触发标志 **ADATE** 并等待系统就绪后，进行参考源或者通道的切换。示例代码如下：

```
ADCSRA = 0x00;                         //关闭 ADC
while (ADCSRA & _BV(ADSC));            //等待 ADC 就绪
/* 在这里添加切换参考源或者通道的代码 */
```

注意，参考源切换后的第一次采样结果可能存在不准确，推荐直接丢弃。

6.2.9.5 关于输入模拟信号的噪声抑制技术

- 当输入信号中包含频率高于 **ADC** 时钟二分之一的信号时，应该在信号输入前加入低通滤波器。
- **PCB** 布线时尽可能的缩短模拟信号线的长度，尽可能的远离高频数字信号线；尽可能保证模拟信号线周围存在模拟 **GND** 的附铜（网）。
- **AVCC**引脚应该通过一个**RC**电路连接在**V_{CC}**上。
- 在 **ATmega48/88/168** 系列芯片中，如果 **PC0~PC3** 用作数字输出，当 **ADC** 正在进行 **AD** 转换时，应该避免操作 **PC0** 至 **PC3** 的引脚输出电平。**PC4** 和 **PC5** 不在此限制之列。
- 如果应用允许，可以使用 **ADC** 噪声抑制模式：在 **CPU** 休眠的模式下进行 **AD** 采样，避免来自 **CPU** 的内部干扰。

6.3 学生用书及幻灯片注解

6.4 常见课堂问题 FAQ

6.5 参考资料

6.5.1 数据手册（Datasheet）

- 数据手册 ATmega48/88/168
  >> 23. Analog-to-Digital Converter

下载地址：http://www.atmel.com/dyn/resources/prod_documents/...

6.5.2 应用手册（Application Note）

- AVR400: Low Cost AD Converter

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc0942.pdf

- AVR401: 8-bit Precision AD Converter

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc0953.pdf

- AVR335: Digital Sound Recorder with AVR® and DataFlash®

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc1456.pdf

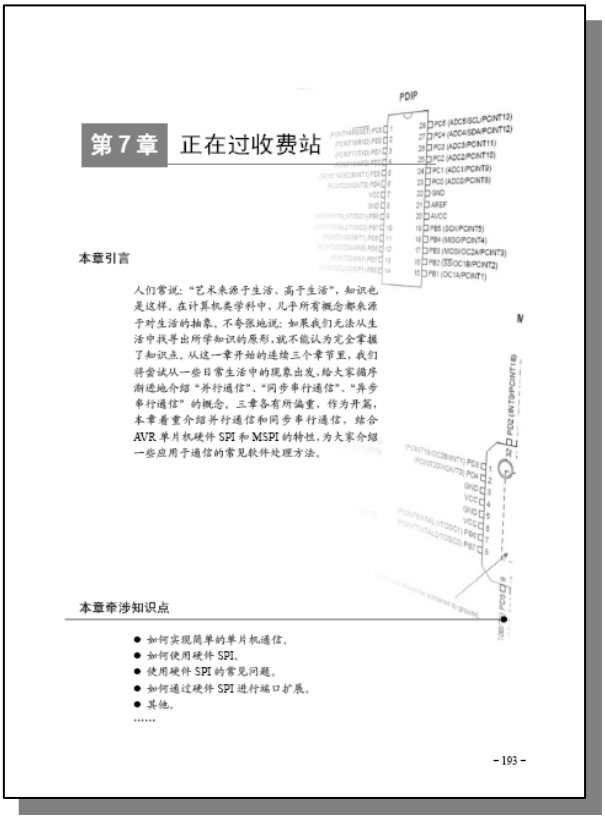
- AVR120: Characterization and Calibration of the ADC on an AVR

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2559.pdf

- AVR121 Enhancing ADC resolution by oversampling

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc8003.pdf

6.5.3 参考文献


第 7 章 正在过收费站

本章引言

人们常说：“艺术来源于生活、高于生活”，知识也是这样，在计算机类学科中，几乎所有概念都来源于对生活的抽象。不夸张地说：如果我们无法从生活中找寻出所学知识的原形，就不能认为完全掌握了知识点。从这一章开始的连续三个章节里，我们将尝试从一些日常生活中的现象出发，给大家循序渐进地介绍“并行通信”、“同步串行通信”、“异步串行通信”的概念。三章各有所偏重，作为开篇，本章着重介绍并行通信和同步串行通信，结合 AVR 单片机硬件 SPI 和 MSPI 的特性，为大家介绍一些应用于通信的常见软件处理方法。

本章牵涉知识点

- 如何实现简单的单片机通信。
- 如何使用硬件 SPI。
- 使用硬件 SPI 的常见问题。
- 如何通过硬件 SPI 进行端口扩展。
- 其他。

……

- 193 -

第七章　SPI 原理与总线设计

Rev 1.0.0.2

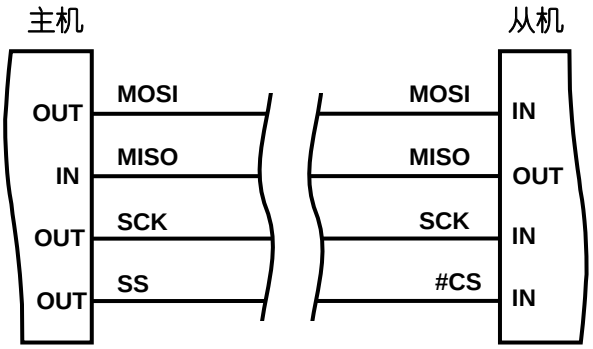
7.1 内容简介

- SPI 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

7.2 SPI 简介

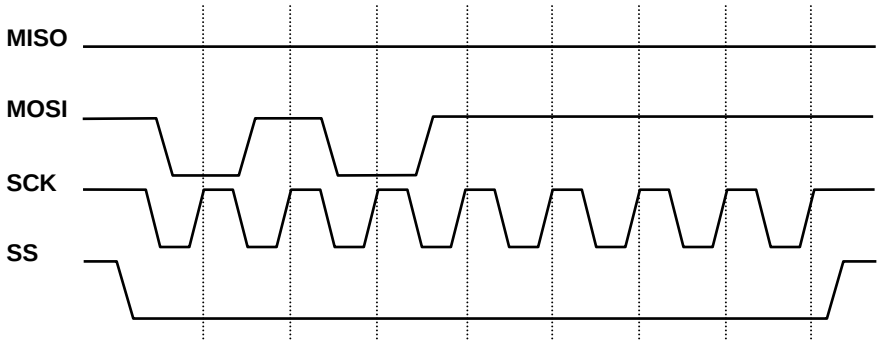
SPI 是串行设备接口（**Serial Peripheral Interface**）的英文缩写，是一种主从式的同步通信协议。一个典型的 **SPI** 接口通常由一根同步时钟信号线 **SCK**、一根主机发送从机接收的数据线 **MOSI**、一根从机发送主机接收的数据线 **MISO** 和一根（或若干）用于主从机通讯同步的控制信号线 **SS**（或片选 **CS** 信号）组成，如**图 7.1** 所示：

图 7.1 一个典型的 SPI 总线



当主机方试图发送数据给从机时，它会首先按照用户设置初始化各个信号引脚的状态，包括引脚的输入输出方向，引脚的初始电平等等。接下来，主机将 **SS** 信号线拉低完成与从机的通讯状态同步，等待主机产生 **SCK** 时钟信号，实现数据在 **MOSI** 和 **MISO** 数据线上的传输。**MOSI** 和 **MISO** 数据线使用简单的电平表示 **0** 或 **1**。**SPI** 从机会根据事先约定好的时钟相位和极性在数据线上设置或者读取信号。当一次通讯完成时，主机可以拉高 **SS** 信号迫使从机进入通讯复位状态。**图 7.2** 是一个典型的 **SPI** 单字节数据传输时序图，图中 **MOSI** 上传输的数据是 **0x5F**，以低位优先的模式发送（**MSB**）。

图 7.2 一个典型的 SPI 数据传输时序



7.2.1 名词解释

● **同步通讯协议**

在通讯过程中使用同步时钟信号进行同步的传输协议称为同步通讯协议。常见的同步时钟信号由连接通讯双方的信号电缆通过传输电平的跳变来表示，比如使用上升沿表示一

次有效的数据传输。

● 异步通讯协议

在通讯过程中使用除同步时钟信号以外的方法实现同步的传输协议称为异步通讯协议。常见的异步通讯是通过时间配合特定具有同步作用的数据帧格式来实现的。在这种异步通讯中，双方根据事先约定好的波特率及数据帧格式进行数据传输，在这一过程中通讯的双方使用不同的时钟源作为自己的工作时钟。异步通讯通常存在一定的误码率。

● 单工通讯

在通讯的任意时刻都只能从通讯的一方向另外一方单向进行数据传输的通讯称为单工通讯。

● 半双工通讯

在通讯的某一时刻只能从通讯的一方向另外一方进行数据传输，而在另外一个时刻又能改变通讯方向的通讯协议被称为半双工通讯协议。半双工通讯的要点是，在任意时刻都只能实现单向传输，但是不同的时刻可以使用不同的传输方向。

● （全）双工通讯

在通讯的任意时刻都可以同时实现数据双向传输的通讯协议称为全双工通讯，简称双工通讯。

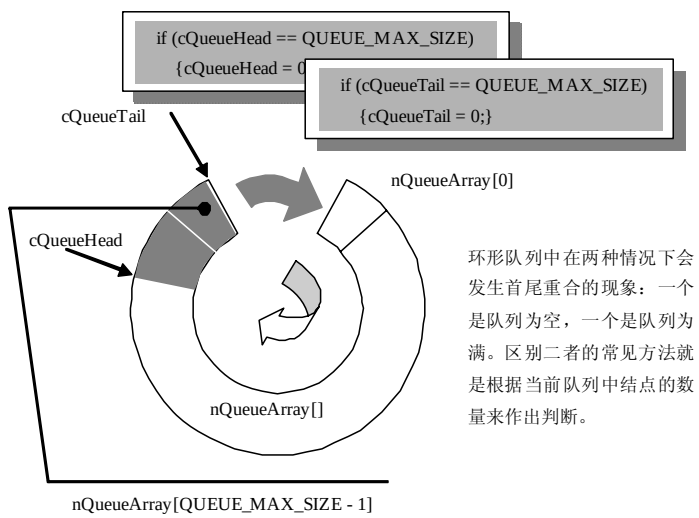
● 通讯缓冲

因为无法及时通讯而先将数据通过一定的数据结构保存下来以便以后传输的方法称为通讯缓冲。用于保存缓冲数据的数据结构被称为缓冲区。常见的通讯缓冲区使用队列作为数据结构。并不是所有的通讯缓冲区都满足先入先出的条件。有时候为了实现数据的优先级传输，会采用加权队列或者其它更适合的数据结构作为通讯缓冲区。

● 环形队列

在使用逻辑上保持线性先入先出关系，在维护逻辑上使用环形结构的队列称为环形队

图 7.3 一个典型的环形队列逻辑示意图



列。环形队列通常拥有一个可多次重复使用的数据存储空间，通过首尾相接的方法实现环形逻辑。这一环形结构可以建立在连续的存储空间上，以地址的函数运算来实现首尾相接；也可以建立在非连续的链表结构之上，以基于指针访问的链式结构来实现环形逻辑。环形队列的使用逻辑与普通队列相同，用户在函数接口上通常无法区别所使用的是环形队列还是普通队列。

7.2.2 AVR 片上 SPI 综述

ATmega48/88/168 及其衍生系列的单片机上集成了硬件 **SPI** 模块。通过简单的寄存器设置就可以实现 **SPI** 通讯协议。硬件 **SPI** 支持主机和从机两种通讯模式，全双同步通讯。

在 **SPI** 模块的内部有一个 **8** 移位数据寄存器 **SPDR**，用于保存用户需要发送的字节；同时通过通讯接收到的数据也保存在该寄存器中。逻辑上，该移位寄存器的输入和输出端连接着 **MOSI** 和 **MISO** 引脚，具体的连接方式根据主机模式和从机模式而不同。

该数据结存器有一个读取缓冲，系统每完成一个字节的移位通讯就会自动将接收的数据复制到读取缓冲中，该数据直到下次通讯完成前都是有效的。对该缓冲区的访问是硬件自动完成的，使用与移位寄存器 **SPDR** 相同的地址。没有写入缓冲。

对移位寄存器 **SPDR** 进行写入操作将立即生效。因此，为了保证通讯的有效性，在 **SPI** 传输的中途，不应该执行任何针对移位寄存器的写入操作。当 **SPI** 完成一次通讯时，**SPI** 通讯状态寄存器 **SPSR** 中的通讯完成标志 **SPIF** 将被置位，如果我们开启了 **SPI** 的通讯完成中断，同时全局中断响应处于使能状态，那么系统会根据事先设定好的中断向量调用相应的中断处理程序。中断处理程序会在调用的过程中自动清除 **SPIF** 标志，用户也可以通过向寄存器 **SPSR** 的 **SPIF** 位写“**1**”实现标志的清零。在未开启中断的情况下，用户可以在启动一次 **SPI** 总线通讯后轮询 **SPIF** 标志查询本次传输是否完成。

一个使用查询方式的 **SPI** 主机典型通讯流程如下：

1、初始化 **SPI** 模块，包括工作模式、时钟频率、时钟极性等；
2、初始化 **SPI** 引脚。将所有需要输出信号的 **GPIO** 设置为电平输出模式。包括 **SS**、**SCK** 和 **MOSI**。将 **SS** 设置为高电平，其它引脚任意；
3、将 **SS** 引脚设置为低电平以通知从机准备进行数据交换；
4、将要发送的数据送入 **SPDR** 寄存器，触发一次通讯。
5、不停的检查 **SPSR** 寄存器的 **SPIF** 标志，等待通讯完成。
6、从 **SPDR** 中读取从机发送过来的数据，并向 **SPSR** 的 **SPIF** 位写“**1**”清除标志。如果还有需要发送的数据，从步骤 **4** 开始继续操作；如果通讯已经完成，将 **SS** 引脚拉高迫使从机进入通讯复位状态，等待下一次传输。

一个使用查询方式的 **SPI** 从机典型通讯流程如下：

1、初始化 **SPI** 模块，包括工作模式、时钟频率、时钟极性等；
2、初始化 **SPI** 引脚。将 **MISO** 引脚设置为电平输出模式，其它引脚任意。
3、将需要发送给主机的数据写入 **SPDR** 寄存器，等待主机 **SCK** 时钟。
4、不停的检查 **SPSR** 寄存器的 **SPIF** 标志，等待通讯完成。
5、从 **SPDR** 中读取从机发送过来的数据，并向 **SPSR** 的 **SPIF** 位写“**1**”清除标志。如果还有需要发送的数据，直接从步骤 3 开始；否则不作任何处理。

7.2.3 SPI 引脚配置

ATmega48/88/168 在使用前，需要对所有用于输出的引脚进行初始化。例如，**SPI** 主机模式下，用户需要手工讲 **SCK**、**SS** 和 **MOSI** 引脚设置为电平输出模式；**SPI** 从机模式下，用户需要将 **MISO** 引脚设置为电平输出模式。系统会 **SPI** 总线通讯中用于输入的引脚，用户无需干预。

对 **ATmega48/88/168** 来说，一个典型的 **SPI** 主机的引脚初始化代码如下：

```
/* 将 SS、MOSI 和 SCK 设置为输出状态 */
DDRB |= (1 << PB2) | (1 << PB3) | (1 << PB5);
/* 初始化 SS 电平 */
PORTB |= (1 << PB2);
```

对 **ATmega48/88/168** 来说，一个典型的 **SPI** 从机的引脚初始化代码如下：

```
/* 将 MISO 设置为输出状态 */
DDRB |= (1 << PB4);
```

7.2.4 数据传输模式

SPI 模块支持 **4** 种不同的 **SCK** 时钟相位和极性的设置，通过寄存器 **SPCR** 的 **CPHA** 和 **CPOL** 位来配置。

其中，**CPOL** 用于选择空闲状态下 **SCK** 的电平，**0** 表示低电平，**1** 表示高电平。显然当 **SCK** 空闲状态下为低电平时，第一个时钟边沿就是上升沿；同样当 **SCK** 空闲状态下为高电平时，第一个时钟边沿为下降沿。对 **CPOL** 的设置实际上完成了对始终信号起始和终止边沿的选择。

表 7.1 CPOL 功能设置

CPOL	起始沿	终止沿
0	上升沿	下降沿
1	下降沿	上升沿

完成了起始沿和终止沿的设定以后，我们还需要给通讯的双方约定“在什么时候设置数据”和“在什么时候读取数据”。**CPHA** 用于在起始沿和终止沿中做出选择，具体设置如**表 7.2** 所示。

表 7.2 CPHL 功能设置

CPHL	起始沿	终止沿
0	电平采样	电平设置
1	电平设置	电平采样

根据 **CPOL** 和 **CPHL** 的设置，**SPI** 总共有 **4** 种不同的时钟工作模式，对应的模式和时序如下所示：

表 **7.3 SPI** 时钟模式

	起始沿	终止沿	SPI 模式
CPOL = 0, CPHA = 0	采样（上升沿）	设置（下降沿）	0
CPOL = 0, CPHA = 1	设置（上升沿）	采样（下降沿）	1
CPOL = 1, CPHA = 0	采样（下降沿）	设置（上升沿）	2
CPOL = 1, CPHA = 1	设置（下降沿）	采样（上升沿）	3

图 **7.4** SPI 数据传输时序图 CPHL = 0

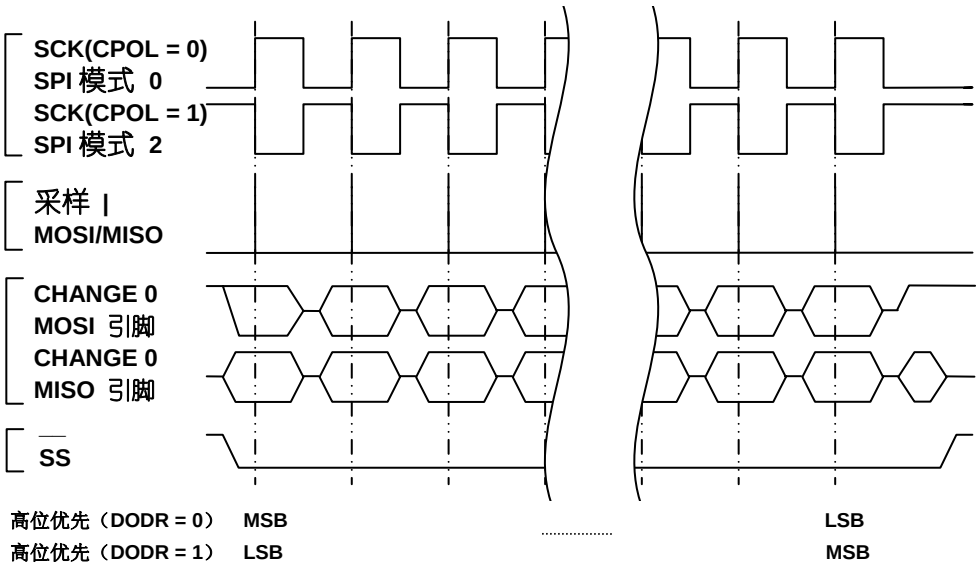
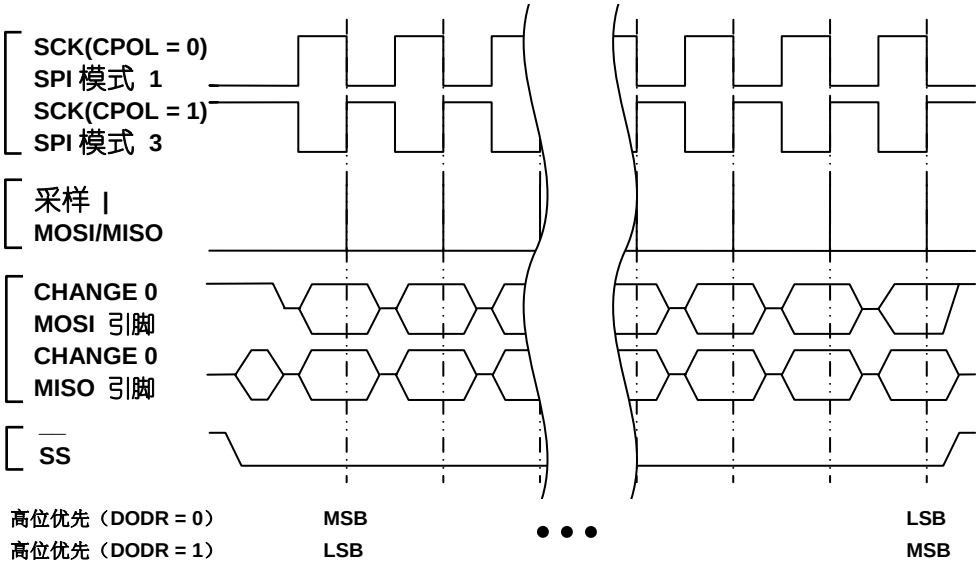


图 **7.5** SPI 数据传输时序图 CPHL = 1



SPI 在数据发送时，可以通过设置 SPCR 寄存器中的 DODR 位来选择首先发送字节的高位还是低位。如图 7.4 和图 7.5 所示，当 DODR 被置位时将以低位优先（Least significant bit 简写为 LSB）的顺序进行数据传输。

7.2.5 代码范例

7.2.5.1 主机发送/接收

```
# define set_ss_pin()          PORTB |= (1<<PB2)
# define clr_ss_pin()          PORTB &= ~(1<<PB2)

/* SPI 主机模式初始化函数
*/
void spi_init(void)
{
    /* 将 SS、MOSI 和 SCK 设置为输出状态 */
    DDRB |= (1 << PB2) | (1 << PB3) | (1 << PB5);
    /* 初始化 SS 电平 */
    PORTB |= (1 << PB2);

    /* 使用与 74HC595 兼容的时钟相位/极性设置，可以直接驱动 74HC595 */
    SPCR =    (1 << SPE) |          //使能 SPI
              (1 << DORD) |         //LSB 优先发送模式
              (1 << MSTR) |         //主机模式
                                    //SPI 传输模式 3：上升沿数据采样
              (1 << CPOL) |         //起始沿为下降沿
              (1 << CPHA) |         //在起始沿设置数据
              (0x00 << SPR0);       //fosc/4
}

/* SPI 主机发送接收函数（使用查询模式）
* 输入参数：需要发送的数据缓冲区，接收数据的缓冲区，需要发送数据的字节数
*/
void spi_master_data_exchange
        (
            unsigned char *pchOutput,
            unsigned char *pchInput,
            unsigned int iSize
        )
{
    /* 拉低 SS 引脚，通知从机准备数据交换 */
    clr_ss_pin();
```

```
        while(iSize--)
        {
            unsigned char chData = 0xFF;
            if (pchOutput != NULL)
            {
                chData = *pchOutput++;
            }
            /* 启动一次通讯 */
            SPDR = chData;
            /* 等待通讯完成 */
            while (!(SPSR & (1 << SPIF)));
            if (pchInput != NULL)
            {
                /* 接收数据 */
                *pchInput++ = SPDR;
            }
        }
        /* 拉高 SS 引脚，通知从机数据交换完成，通讯状态机复位 */
        set_ss_pin();
}
```

7.2.5.2 从机发送/接收

```
/* SPI 从机接收模式初始化函数
*/
void spi_init(void)
{
    /* 将 MISO 设置为输出状态 */
    DDRB |= (1 << PB4);

    SPCR =    (1 << SPE) |          //使能 SPI
              (1 << DORD) |         //LSB 优先发送模式
              (0 << MSTR) |         //从机模式
                                    //SPI 传输模式 3：上升沿数据采样
              (1 << CPOL) |         //起始沿为下降沿
              (1 << CPHA) |         //在起始沿设置数据
              (0x00 << SPR0);       //fosc/4
}

/* 从机 SPI 发送接收函数
* 输入参数：要发送的字节
* 输出参数：接收到的数据
*/
```

```
unsigned char spi_slave_data_exchange(unsigned char chOupputData)
{
    /* 在 SPI 数据缓冲中放入要发送给主机的数据 */
    SPDR = chOutputData;
    /* 等待与主机通讯完成 */
    while (!(SPSR & (1 << SPIF)));
    /* 返回从主机中读取到的数据 */
    return SPDR;
}
```

7.2.6 注意事项

当 **ATmega48/88/168** 的 **SPI** 工作在从机模式时，主机产生的 **SCK** 时钟必须满足以下的条件：

- **SCK** 的低电平时间必须大于从机 **2** 个系统时钟周期；
- **SCK** 的高电平时间必须大于从机 **2** 个系统时钟周期；
- **SCK** 的单个时钟周期应该大于 **4** 个从机系统时钟周期。

7.3 学生用书及幻灯片注解

7.4 常见课堂问题 FAQ

7.5 参考资料

7.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168**。
  - **>> 1. Pin Configurations**
  - **>> 18. SPI – Serial Peripheral Interface**
  - **>> 20. USART in SPI mode**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

7.5.2 应用手册（Application Note）

- **AVR151: Setup And Use of The SPI**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2585.pdf**

- **AVR320: Software SPI Master**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1108.pdf**

- **AVR303: SPI-UART Getway**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2557.pdf**

- **AVR319: Using the USI module for SPI communication**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2582.pdf**

- **AVR107: Interfacing AVR serial memories**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2595.pdf**

- **AVR317: Using the Master SPI Mode of the USART module**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2577.pdf**

7.5.3 参考文献

7.6 背景知识

7.6.1 数据结构

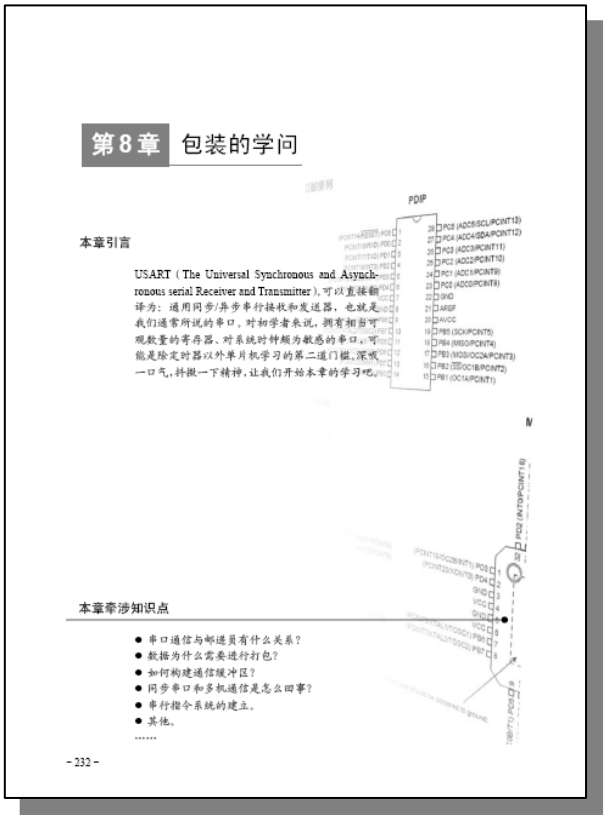
本章的内容牵涉到数据结构队列章节的内容。包括使用数组和链表实现队列以及环形队列的相关知识。

7.6.2 数字逻辑与电路

本章的内容牵涉到数字逻辑与电路中关于移位锁存器的相关内容。包括 D 触发器、移位锁存器；使用 **74HC164**、**74HC165** 和 **74HC595** 进行数字时序电路设计的相关知识。

本章在 **SPI** 环路总线设计的章节中，涉及到菊花链 **Daisy Chain** 的内容。在并行 **SPI** 总线设计中，涉及到使用 **74HC138** 进行译码电路设计的内容；在实验章节键盘显示模块的内容中，涉及到使用与非门实现“非”逻辑的内容。

7.7 参考设计与实验方法


第8章 包装的学问

本章引言

USART（The Universal Synchronous and Asynchronous serial Receiver and Transmitter），可以直接翻译为：通用同步/异步串行接收和发送器，也就是我们通常所说的串口。对初学者来说，拥有相当可观数量的寄存器、对系统时钟颇为敏感的串口，可能是除定时器以外单片机学习的第二道门槛，深吸一口气，拌擞一下精神，让我们开始本章的学习吧。

本章牵涉知识点

- 串口通信与邮递员有什么关系？
- 数据为什么需要进行打包？
- 如何构建通信缓冲区？
- 同步串口和多机通信是怎么回事？
- 串行指令系统的建立。
- 其他。

……

- 232 -

第八章　U(S)ART 通讯与串行协议

Rev 1.0.0.1

8.1 内容简介

- U(S)ART 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

8.2 U(S)ART 简介

USART 是 **The Universal Synchronous and Asynchronous serial Receiver and Transmitter** 的缩写，通称为串口。**USART** 是目前嵌入式系统中使用最普遍的一种串行通讯协议。按照串行通讯的方向来分类，**USART** 有单工、半双工和全双工三种通讯模式。其中，单工模式只有一根用于接收数据的 **RXD** 信号线或者一根用于发送数据的 **TXD** 信号线；半双工模式通常只有一根数据线，根据通讯协议的约定在不同时间内分别作为 **RXD** 或 **TXD** 来使用；全双工模式同时拥有 **TXD** 和 **RXD** 两根数据线。按照通讯的同步方式来分类，**USART** 有同步（**Synchronous Mode**）和异步（**Asynchronous**）两种通讯模式。

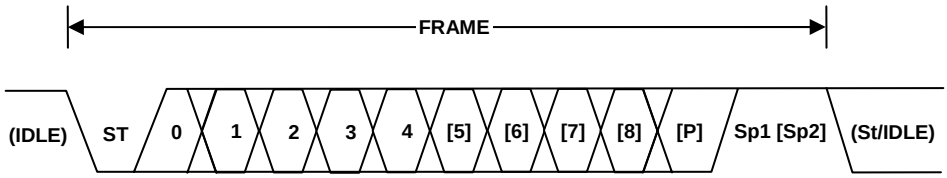
在同步模式下，串口被称为 **USRT**，是一种主从式的通讯。主机通过在同步时钟 **XCK** 上发送时钟信号与从机进行 **Bit** 级别的同步。工作于单工或者半双工模式下的 **USRT** 通常只在数据的传送时刻才会产生时钟信号。工作于全双工模式下的 **USRT**，**XCK** 同步时钟则是从主机初始开始一直存在的。

在异步模式下，串口被称为 **UART**。通讯的各方工作在不同的时钟频率下，根据事先约定好的波特率和数据帧格式进行数据交换。与 **USRT** 不同，**UART** 使用事先约定的波特率来进行 **Bit** 级别的同步。由于通讯各方工作在不同的时钟频率下，因此往往只能以接近或者说近似约定波特率的时间节拍进行通讯，因而误码在理论上是不可避免的。误码率是异步串口模式下一个重要的指标。

USART 通过事先约定数据帧格式的方法实现数据的同步。通讯数据帧存在于 **RXD** 和 **TXD** 信号线上。全双工模式下，数据的发送和接收是相互独立的。如图 **8.1** 所示，当通讯信号处于空闲状态（**IDLE**）时，信号线保持高电平；数据帧中的每一个信元都是以 **Bit** 为单位的，一个 **Bit** 信元在 **USRT** 和 **UART** 模式下分别以发生 **XCK** 时钟和约定异步波特率两种方式实现同步。

数据帧以一个 **Bit** 长度的低电平表示通讯的起始，紧接着是通讯的数据实体。根据事先约定的不同，数据的长度从 **4** 位到 **8** 位（**AVR** 拥有一个特殊的第九位）**Bit** 不等，通常使用 **8** 位数据长度来发送一个完整的字节。跟在数据后面的是校验位，可以选择奇校验、偶校验或者无校验。整个数据帧的结束通过可设的一个或者两个 **Bit** 长度的高电平来表示，称为停止位。异步通讯模式下，有时还存在 **1.5** 个 **Bit** 长度的停止位。

图 8.1 U(S)ART 通讯数据帧



St	**起始位，始终为低电平**
(n)	**数据位 (0 至 8)**
P	**校验位，可以选择奇校验或者欧校验**
Sp	**停止位，可以选择 1 停止位 2 停止位，异步模式下还可以选择 1.5 停止位**
IDLE	**通讯线（TXD 或 RXD）上无任何通讯，IDEL 状态必须为高电平**

8.2.1 名词解释

● **波特率**

串行通讯中，1 秒内传输信元的个数称为波特率。波特率的倒数就是每个信元的传送时间。

● **通讯误码**

“异步串行通讯误码”简称“通讯误码”。通讯方使用根据各自不同工作时钟频率而产生的波特率进行通讯时，由于波特率误差积累产生的错误数据帧被称为异步串行通讯误码。

● **误码率**

“异步串行通讯误码率”简称“误码率”。通讯方根据自身工作频率产生的波特率与标准波特率之差的绝对值占标准波特率的百分比称为“异步串行通讯误码率”。计算公式为：

误码率 = (|本地波特率 – 标准波特率| / 标准波特率) × 100%

● **RS232**

RS232 是由美国电子工业协会（**EIA**）提出的接口标准，它定义了串行通信用的电平范围等技术规范，由电器参数、机械参数和通讯协议组成。通常使用的 **RS232** 协议为 **RS232C**，其中 **C** 为版本号。

● **串口总线**

狭义的串口总线是指使用 **U(S)ART** 接口协议实现的通讯总线。广义的串口总线是指以字节为通讯信元的主从式总线系统；或者通过接口技术虚拟 **U(S)ART** 实现的通讯总线，而实际使用的接口可能是红外、蓝牙等。

● **串口多机通讯**

建立在串口基础上，以硬件或者软件方式实现的主从式通讯协议。在多机通讯模式中，有一个主机和若干个从机。每个从机都有一个唯一的地址，主机通过这一地址与从机建立连接，并完成数据的传输。

● **地址帧和数据帧**

在串口多机通讯模式下由主机发送的包含从机地址信息的数据帧称为地址帧。除地址帧以外的数据帧称为数据帧。主从双方都可以发送数据帧。

● **双缓冲**

串行通讯双缓冲，简称双缓冲，是指同时拥有接收缓冲和发送缓冲区的缓冲系统。有时候也可以被称为通讯管道。

8.2.2 AVR 片上 U(S)ART 综述

ATmega48/88/168 及其衍生型号提供一个硬件 **U(S)ART** 模块，支持全双工异步或同步通讯模式。**5~9** 可选数据位长度；奇校验、偶校验和无校验三种校验模式可选；**1** 至 **2** 停止位

可设。内置硬件数据帧过滤器，通过对特定的数据帧进行过滤实现对多机通讯模式的支持。当配置为同步通讯模式时，引脚 **XCK** 将用于通讯同步。主从机的选择根据 **XCK** 引脚的方向寄存器 **DDRn** 来决定：配置为输出状态时，**U(S)ART** 处于主机同步模式，系统会在完成模块初始化以后立即产生同步时钟而不管当前是否有数据需要传送，主机模式下 **XCK** 时钟极性是可设的；配置为输入状态时，**U(S)ART** 处于从机同步模式，通讯波特率由主机决定。需要注意的是，主机发生的 **XCK** 时钟频率应该小于从机系统时钟的 **1/4**。

USART 模块有一套寄存器系统来实现通讯，分别是 **UCSRnA**、**UCSRnB**、**UCSRnC**、**UBRRnL**、**UBRRnH** 和数据寄存器 **UDRn**。其中，**UCSRnA**、**UCSRnB** 和 **UCSRnC** 用于通讯模式的配置、通讯帧的设置以及当前通讯状态的查询。寄存器 **UBRRnL** 和 **UBRRnH** 用于设置当前系统时钟下的通讯波特率。寄存器 **UDRn** 用于保存发送和接收到的数据，由共用地址的发送和接收寄存器构成。**UDRn** 拥有寄存器双缓冲。

当 **USART** 接收到数据时，会设置 **UCSRnA** 中的接收完成标志 **RXCn**，如果此时开启了寄存器 **UCSRnB** 中的中断使能标志 **RCXIE**，并且系统状态寄存器 **SREG** 中的全局中断响应标志被置位，接收完成中断将被触发。只要有未读取的数据存放在 **UDRn** 中，**RCXIE** 标志将保持为 **1**；当所有接收到的数据都被读取以后，该标志将被自动清零。因此在接收完成中断处理程序中，不应该在读取 **UDRn** 寄存器前开启全局中断响应，否则将造成接收完成中断重复嵌套，严重情况下会造成系统崩溃。所有针对 **USART** 状态的读取（包括第九位数据的读取）都应该在操作 **UDRn** 前完成，针对 **UDRn** 的读取操作将改变系统当前状态。

当 **USART** 数据发送完成时，会设置 **UCSRnA** 中的发送完成标志 **TXCn**，如果此时开启了寄存器 **UCSRnB** 中的中断使能标志 **TCXIE**，并且系统状态寄存器 **SREG** 中的全局中断响应标志被置位，发送完成中断将被触发。中断处理程序执行时，**TCXIE** 标志将被自动清零；非中断模式下，可以通过向该位写“**1**”实现标志的清零。

由于 **UDRn** 拥有 **1** 个字节的发送缓冲，当 **UDRn** 中已经存在一个正在发送的数据时，还可以缓冲一个用户写入的数据。当发送缓冲为空时，**UCSRnA** 中的标志 **UDREn** 将被置位，如果此时开启了寄存器 **UCSRnB** 中的中断使能标志 **UDRIEn**，并且系统状态寄存器 **SREG** 中的全局中断响应标志被置位，将产生一个中断表示当前数据缓冲区为空。

8.2.3 U(S)ART 引脚配置

U(S)ART 模块的发送和接收功能可以独立使能或者关闭，分别通过寄存器 **UCSR0B** 的 **RXEN0** 和 **TXEN0** 来控制：置位表示使能，清零表示关闭。当发送或接收的功能被使能时，对应的引脚 **TXD** 或 **RXD** 将自动作为设备引脚，普通 **GPIO** 的功能将被取代，此时针对 **GPIO** 所作的引脚方向、电平设置都是无效的（**RXD** 引脚上的上拉电阻仍然是可以控制的）。当发送或接收的功能被关闭时，对应的引脚 **TXD** 或 **RXD** 将恢复普通 **GPIO** 的功能。

使用 **U(S)ART** 模块来构建主从式串行总线时，需要注意当发送或接收功能被禁止时 **GPIO** 的方向和输出电平设置。为了防止由于本机发送功能关闭、**TXD** 引脚恢复正常 **GPIO** 引脚功能时处于电平输出状态，应该在引脚初始化时，将 **TXD** 引脚设置为输入模式，是否开启上拉电阻则根据总线的通讯状况而定。理想状态下，应该尽可能保持连接在总线上的数据发送引脚处于高阻态。**ATmega48/88/168** 及其衍生型号的典型端口初始化代码如下：

```
/* 定义系统时钟频率 */
void port_init(void)
{
```

```
    ……
    /* usart0 txd 引脚初始化：设置为输入状态并关闭上拉电阻 */
    DDRD &= ~(1 << PD1);
    PORTD &= ~(1 << PD1);
    ……
}
```

8.2.4 波特率设置

U(S)ART 模块支持异步、异步双倍速和同步主机三种波特率模式。同步模式下的从机使用 **XCK** 时钟信号进行通讯，此时 **XCK** 的时钟不应高于从机系统时钟频率的 **1/4**。同步模式下，**XCK** 引脚的输入输出方向决定了时钟的输入输出方向，从而决定了模块的主从关系。异步、异步双倍速和同步主机模式下的波特率计算公式如所**表 8.1** 示。

表 8.1 波特率寄存器设置计算公式一览表

时钟工作模式	波特率计算公式	UBRRn 寄存器计算公式
普通异步模式（U2X = 0）	$BAUD = \frac{fosc}{16(UBRRn+1)}$	$UBRRn = \frac{fosc}{16BAUD} - 1$
双倍速异步模式（U2X = 1）	$BAUD = \frac{fosc}{8(UBRRn+1)}$	$UBRRn = \frac{fosc}{8BAUD} - 1$
同步主机模式	$BAUD = \frac{fosc}{2(UBRRn+1)}$	$UBRRn = \frac{fosc}{2BAUD} - 1$

注：BAUD 表示波特率，fosc 表示系统时钟，UBRRn 表示波特率设置寄存器。

8.2.5 数据帧设置

一个典型的 **U(S)ART** 数据帧由起始位、数据位、校验位和停止位构成。其中，数据长度、校验位的种类以及停止位的个数是可以通过寄存器 **USCRnA**、**UCSRnB** 和 **UCSRnC** 进行设置的。

对照**表 8.2** 的内容对寄存器 **USCRnC** 的 **USCZn0** 和 **USCZn1** 位以及寄存器 **USCRnB** 的 **USCZn2** 位进行设置，可以分别获得 **5~9** 位长度的数据模式。

表 8.2 数据长度寄存器设置明细表

UCSZn2	UCSZn1	UCSZn0	数据长度
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit

1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

对照**表 8.3** 的内容对寄存器 **USCRnC** 的 **UPMn** 位进行设置，可以分别设置数据帧的校验模式为：无校验、偶校验和奇校验。

表 8.3 校验类型寄存器设置明细表

UPMn1	UPMn0	校验模式
0	0	无校验
0	1	保留
1	0	偶校验
1	1	奇校验

在寄存器 **USCRnC** 中，**USBS** 位用于选择停止位的个数。清零表示使用一个停止位；置位表示使用两个停止位。**ATmega48/88/168** 不支持 **1.5** 个停止位。

8.2.6 9 数据位模式

U(S)ART 模块支持一个特殊的 **9** 数据位模式。当 **USCRnB**、**USCRnC** 寄存器中的 **USCZn2**、**USCZn1** 和 **USCZn0** 都被置位时，系统将在数据寄存器 **UDRn** 所保存的内容以外增加一个额外的第九位数据。对第九位数据的读取和写入操作分别是通过寄存器的 **USCRnB** 的 **RXB8n** 和 **TXB8n** 位来实现的。

发送数据时，在向 **UDRn** 写入其余 **8** 位数据前必须完成对第九位数据的设置，因为针对 **UDRn** 的写入操作将启动 **U(S)ART** 的数据发送。读取数据时，必须在读取 **UDRn** 寄存器前完成对第九位数据的读取，因为针对 **UDRn** 的读取操作会改变 **USCRnB** 中的状态标志。典型的 **9** 数据位操作 **C** 语言代码范例请参考 **8.2.5.2** 小节。

8.2.7 多机通讯

U(S)ART 模块有一个特殊的数据帧过滤器，将寄存器 **UCSRnA** 的 **MPCMn** 位置“**1**”时将开启数据帧过滤器。该过滤器将过滤所有数据帧，保留地址帧。关于数据帧和地址帧的定义如下：

- 通讯的数据位数在 **5~8** 位时，第一个停止位用于区别地址帧和数据帧。当第一个停止位为高电平时，当前帧为地址帧；当第一个停止位为低电平时，当前帧为数据帧。
- 通讯的数据在 **9** 位时，第九位数据用于区别地址帧和数据帧。第九位为“**1**”表示当前帧为地址帧；第九位为“**0**”当前帧为数据帧。

当过滤器被开启时，**U(S)ART** 模块只能接收到地址帧；当过滤器被关闭时，模块既可以接收地址帧也可以接收数据帧。组建串口总线时，如果发送方和接收方都支持九数据位

模式，通常使用 **9** 数据位模式来实现多机通讯。

此时，典型的主机通讯策略如下：

1、 设置第九位，表示当前帧为地址帧。将从机地址写入 **UDRn** 发送地址帧。

2、 等待从机响应。如果从机响应正确，将第九位清零并与从机进行数据交换；如果从机无相应或响应错误，结束本次总线会话，回到步骤 **1**。

3、 与从机通讯完成后，重复步骤 **1**，进行下一次总线会话。

基于以上的主机通讯协议，典型的从机通讯策略如下：

1、 开启数据帧过滤器。

2、 **U(S)ART** 模块自动过滤所有的数据帧，只有地址帧会触发接收完成标志；

3、 将接收到的地址帧与从机自身的地址进行比较，如果匹配并且系统的当前状态允许从机与主机进行通讯，则响应主机并关闭数据帧过滤器。如果地址不匹配则重复步骤 **2**。

4、 从机接收来自主机的数据，如果有主从双方明确的通讯完成标志发生，则结束本次总线会话，重复步骤 **1**；如果在总线会话时刻，接收到了地址帧，则立即结束本次总线会话并开启数据帧过滤器，重复步骤 **3**。

以上典型通讯过程中可能用到的代码范例请参考 **8.2.5.2** 小结。

8.2.8 代码范例

8.2.8.1 普通模式下数据发送和接收

```
/* 定义系统时钟频率 */
#define F_CPU            8000000
/* 定义串口波特率 */
#define BAUD_RATE        9600

/* USART 初始化函数，异步模式，查询模式
*/
void usart0_init(void)
{
    UCSR0B = 0x00;                          //关闭 USART0
    UCSR0A = 0x00;
    UCSR0C = (0 << UMSEL00) |               //异步通讯模式
             (0 << UPM00) |                 //无校验 0-无校验 2-偶校验 3-奇校验
             (0 << USBS0) |                 //1 停止位
             (3 << UCSZ00);                     //8 数据位模式
    UBRR = (F_CPU/( BAUD_RATE *16L)-1); //设置波特率
    UCSR0B = (1 << RXEN0) |                 //接收使能
             (1 << TXEN0);                  //发送使能
```

```
}

/* USRT 数据发送
* 输入参数：要发送的缓冲区指针，缓冲区的字节大小
*/
void put_stream(unsigned char *chSTR,unsigned int wLength)
{
    while(wLength--)
    {
        UDR0 = *chSTR++;                    //发送缓冲区中的数据
        while(!(UCSR0A & (1 << TXC0)));     //等待发送完成
        UCSR0A |= (1 << TXC0);              //清除发送完成标志
    }
}

/* 从串口中读取一个字节
* 输出：接收到的字节
*/
unsigned char get_char(void)
{
    while (!(UCSR0A & (1 << RXC0)));        //等待接收到数据
    return UDR0;
}
```

8.2.8.2 多机模式下的 9 数据位发送和接收

```
/* 定义系统时钟频率 */
#define F_CPU           8000000
/* 定义串口波特率 */
#define BAUD_RATE       9600

/* USART 初始化函数，异步模式，查询模式
*/
void usart0_init(void)
{
    UCSR0B = 0x00;                          //关闭 USART0
    UCSR0A = (1 << MPCM0);                  //启动多机通讯模式
    UCSR0C =  (0 << UMSEL00) |              //异步通讯模式
              (0 << UPM00) |                //无校验 0-无校验 2-偶校验 3-奇校验
              (0 << USBS0) |                //1 停止位
              (3 << UCSZ00);
    UBRR = (F_CPU/( BAUD_RATE *16L)-1); //设置波特率
```

```
    UCSR0B =  (1 << RXEN0) |                    //接收使能
              (1 << TXEN0) |                    //发送使能
              (1 << UCSZ02);                    //9 数据为模式
}


/*  关闭数据帧过滤器
*/
void usart0_disable_data_frame_filter(void)
{
    UCSR0A = UCSR0A & ~((1 << RXC0) | (1 << TXC0) | (1 << FE0) |
                        (1 << UPE0) | (1 << MPCM0));
}


/*  开启数据帧过滤器
*/
void usart0_enable_data_frame_filter(void)
{
    UCSR0A = (UCSR0A & ~((1 << RXC0) | (1 << TXC0) | (1 << FE0) | (1 << UPE0)))
             |   (1 << MPCM0);
}


/* USRT 数据发送
*  输入参数：要发送的缓冲区指针，缓冲区的字节大小，第九数据位
*/
void put_stream
        (
            unsigned char *chSTR,
            unsigned int wLength,
            unsigned char bit9thBit
        )
{
    while(wLength--)
    {
        UCSR0B |= ( (bit9thBit ? 1 : 0) << TXB80); //发送第九位数据
        UDR0 = *chSTR++;                           //发送缓冲区中的数据
        while(!(UCSR0A & (1 << TXC0)));            //等待发送完成
        UCSR0A |= (1 << TXC0);                     //清除发送完成标志
    }
}
```

```
/* 从串口中读取一个字节
* 输入参数：保存读取数据的缓冲区指针
* 输出：第九位数据
*/
unsigned char get_char(unsigned char *pchData)
{
    unsigned char ch9thBit;
    while (!(UCSR0A & (1 << RXC0)));                  //等待接收到数据
    ch9thBit = (UCSR0B & (1 << RXB80)) ? 1 : 0;   //读取第 9 位
    if (pchData != NULL)
    {
        *pchData = UDR0;                              //读取数据
    }
    return ch9thBit;
}
```

8.2.9 注意事项

当 **ATmega48/88/168** 工作在同步通讯 **USRT** 模式下时，**USRT** 主机产生的 **XCK** 时钟频率应该小于从机系统时钟频率的 **1/4**。

当 **ATmega48/88/168** 工作在 **MSPI** 通讯模式下时，如果从机是 **AVR** 芯片，应当查询对应从机芯片数据手册的 **SPI** 章节，以确定从机对主机所发生时钟的要求。例如，当从机是 **ATmega48/88/168** 时，主机发生的时钟频率其低电平长度应该大于 **2** 个从机系统时钟周期；其高电平长度应该大于 **2** 个从机系统时钟周期。

8.3 学生用书及幻灯片注解

8.4 常见课堂问题 FAQ

8.5 参考资料

8.5.1 数据手册（Datasheet）

- 数据手册 ATmega48/88/168。
  >> 19. USART0

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf

8.5.2 应用手册（Application Note）

- AVR303: SPI-UART Getway

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2557.pdf

- AVR317: Using the Master SPI Mode of the USART module

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2577.pdf

- AVR054: Run-time calibration of the internal RC oscillator

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2563.pdf

- AVR140: ATmega48_88_168 family run-time calibration of the Internal RC oscillator for LIN application

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc7653.pdf

- AVR244: AVR UART as ANSI Terminal Interface

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2547.pdf

- AVR274: Single-wire Software UART

下载地址：http://www.atmel.com/dyn/resources/prod_documents/AVR274.pdf

- AVR304: Half Duplex Interrupt Driven Software UART

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc0941.pdf

- AVR305: Half Duplex Compact Software UART

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc0952.pdf

- AVR306: Using the AVR UART in C

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc1451.pdf

- AVR317 Using the Master SPI Mode of the USART module

下载地址：http://www.atmel.com/dyn/resources/prod_documents/doc2577.pdf

8.5.3 参考文献

8.6 背景知识

8.6.1 数据结构

本章的内容牵涉到数据结构队列章节的内容。包括使用数组和链表实现队列以及环形队列的相关知识。

8.6.2 通信原理

本章的内容牵涉到通信原理中关于信元定义、信噪比以及香侬公式的相关内容。

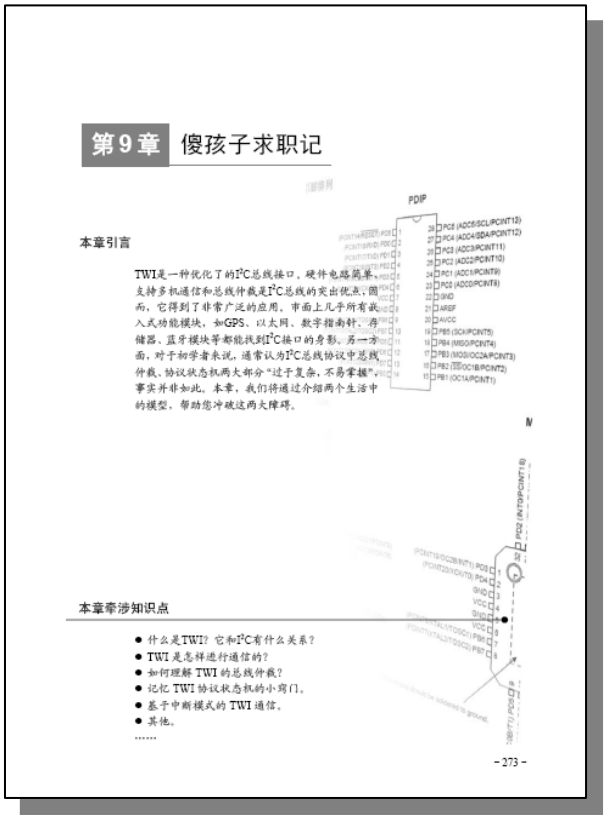
8.6.3 微机原理与接口技术

本章的内容牵涉到微机原理与接口技术中关于 **RS232** 通讯接口的相关内容。

8.6.4 数字逻辑与电路

本章的内容涉及到数字逻辑与系统中关于 **TLL** 电平与 **RS232** 电平间转换电路的设计。主要包括 **MAX232** 电平转换电路的设计。

8.7 参考设计与实验方法


第9章 傻孩子求职记

本章引言

TWI是一种优化了的I²C总线接口，硬件电路简单、支持多机通信和总线仲裁是I²C总线的突出优点；因而，它得到了非常广泛的应用。市面上几乎所有嵌入式功能模块，如GPS、以太网、数字指南针、存储器、蓝牙模块等都能找到I²C接口的身影。另一方面，对于初学者来说，通常认为I²C总线协议中总线仲裁、协议状态机两大部分"过于复杂，不易掌握"，事实并非如此。本章，我们将通过介绍两个生活中的模型，帮助您冲破这两大障碍。

本章牵涉知识点

- 什么是TWI？它和I²C有什么关系？
- TWI是怎样进行通信的？
- 如何理解TWI的总线仲裁？
- 记忆TWI协议状态机的小窍门。
- 基于中断模式的TWI通信。
- 其他。

……

- 273 -

第九章　TWI 总线

Rev 1.0.0.1

9.1 内容简介

- TWI 协议简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

9.2 TWI 协议简介

TWI是**ATMEL**公司提出的总线规范，兼容**I²C**协议，为多芯片间数据交换提供解决方案。该总线协议从提出至今，得到了近百家公司的认可，成为了一种事实上的标准。**TWI**总线协议提供了有效的**IC**控制，规定了非常简单的电路连接方式，使得多芯片硬件电路设计得以简化。

TWI 总线是一种同步办双工总线，由一根同步时钟信号 **SCL** 和一根双向数据信号 **SDA** 组成。**TWI** 总线是一种采用并行连接的主从式总线，总线上的主机负责发生时钟信号。主机可以使确定，也可以通过仲裁机制来实现主从关系的切换。总线的工作频率由主机和从机共同决定，普通情况下（非高速模式）其适应范围可以从 **100Kb/s** 以下直至 **400Kb/s**。

TWI 通讯已数据包为单位，每个数据包都由一个地址帧和若干个数据帧组成。在时钟信号 **SCL** 的高电平阶段，数据线 **SDA** 的四种电平状态：高电平、低电平、下降沿和上升沿分贝表示 **1**、**0**、数据帧起始信号 **START** 和数据帧结束信号 **STOP**。所有电平信息的转换应该在 **SCL** 的低电平阶段完成。从机可以拉低时钟信号线，延缓总线通讯，以获取足够的数据处理时间。一个数据帧总是由起始信号 **START**、若干位定长数据（**8** 位或者 **16** 位）、一个数据接收放的应答位（低电平表示 **ACK**，高电平表示 **NACK**）和一个终止信号 **STOP** 组成。当需要进行连续数据帧传输时，帧与帧之间的 **STOP** 信号可以省略，此时下一帧的 **START** 信号就被称为 **REPEAT START**；如果保留 **STOP** 信号，则下一个帧帧的 **START** 信号被称为 **START AFTER STOP**。二者的区别在于，当存在多主机竞争总线时，**START AFTER STOP** 会放弃总线而重新参与仲裁；而 **REPEAT START** 的发送方将会继续作为主机进行总线通讯。

TWI 协议唯一的缺点在于，总线上所有设备都必须遵循协议状态机，任何一个设备因为故障拉低了总线或者协议状态机不完整都会危及整个通讯系统的正常运行。遗憾的是，协议状态机必须通过用户由软件编程来实现。

9.2.1 名词解释

- **线与**

TTL 电路或 **CMOS** 电路中，多个开集/开漏输出的引脚连接在一起时，当任意一个引脚输出低电平信号都将变成低电平的特性，称为线与特性。线与逻辑常被用于总线设计和简易电平转换。

- **两线串行接口 (TWI)**

TWI是两线串行接口 **2-Wire serail Interface**的英文缩写。是一种面向字节和基于中断操作的兼容**I²C**协议的增强型**2**线同步半双工通讯协议接口。其支持的最高通讯频率为**400kb/s**。

- **流操作串行管道 (Serial Stream Pipe)**

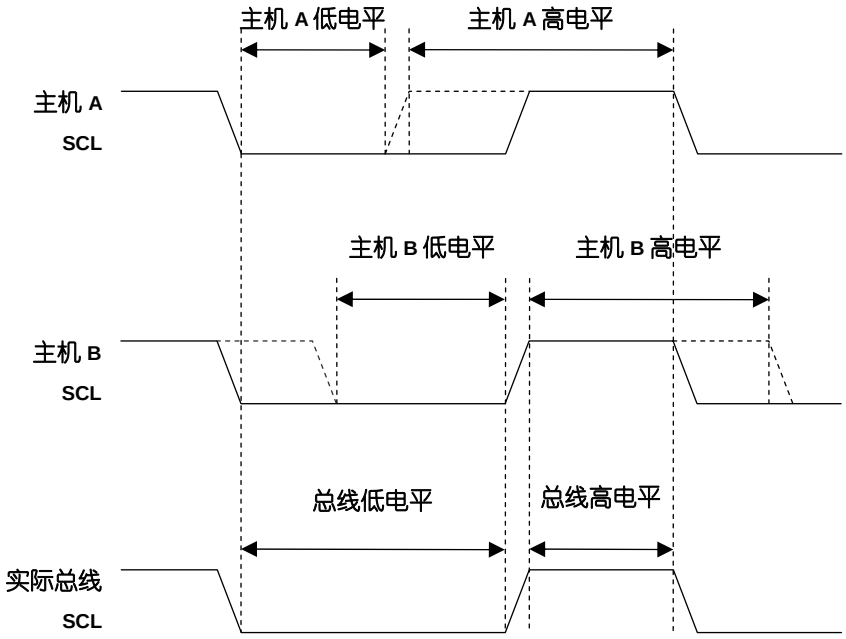
嵌入式系统中，所有串行读写操作都可以被称为流操作。串行缓冲区也可以被称为流缓冲区。一般将串行操作的发送缓冲区和接收缓冲区统称为流操作管道，或者简称为管道。

9.2.2 总线仲裁

当 **TWI** 总线处于空闲状态时，如果有多个主机同时想占有总线，仲裁就发生了。由于 **TWI** 总线由 **SCL** 和 **SDA** 两根信号线构成，因此仲裁也是分两个部分同时进行的，分别称为时钟同步和主机仲裁。I2C 总线的仲裁是基于线与特性而建立的。

仲裁进行时，多个主机同时发生自己时钟信号，这一信号由于线与特性的存在将在 **SCL** 上形成一个复合时钟。该时钟信号的特征是：其低电平时间由所有主机共同决定，其高电平时间由高电平最短的一个主机来决定。所有通讯行为都将根据这一复合后的时钟进行（如图 **9.1** 所示）。在仲裁中失败的主机可以继续产生时钟，直到整个数据帧发送完成。

图 9.1 主机时钟同步原理示意图



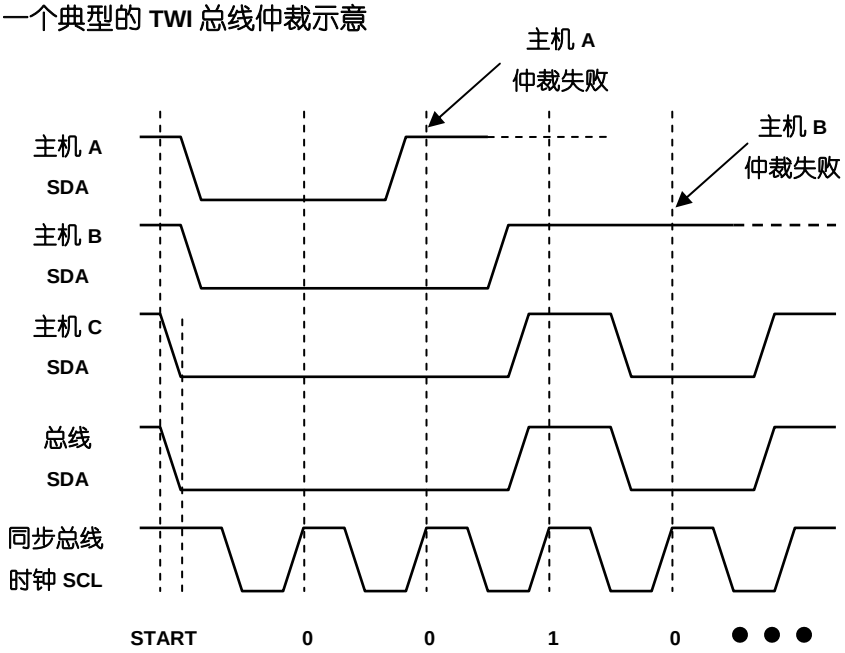
在复合时钟的高电平期间，所有的主机在 **SDA** 信号线上发送自己的高低电平信息，同时比较总线上实际的电平与自身发送的是否相同，一旦信号出现偏差，则表示总裁失败，自动转为从机模式。仲裁失败的主机不会丢失此时总线上已经发送的数据。（如图 **9.2** 所示）

9.2.3 外设操作模式

TWI协议是一个基于状态机的总线通讯协议。当总线的主从关系明确，且不存在总线总裁时，可以按照一个简单的顺序操作流程来进行数据的收发操作。一个典型的实例就是单个芯片操作多个**I^2C**外设，比如读取多个 **24Cxxx**系列存储器。

如果总线必须支持多主机仲裁，则任何连接到总线的设备都必须具有较为强壮的协议状态机。否则，任何异常的设备都将影响整个总线的正常工作。**AVR** 片上 **TWI** 模块同时提供了对以上两种操作模式的支持。**AVR** 单片机片上 **TWI** 模块提供基于中断的状态机驱动模式；也提供基于标志位查询的顺序操作模式。根据实际需求使用正确的操作模式是调试 **TWI** 总线的关键。

图 9.2 一个典型的 TWI 总线仲裁示意



9.2.6 虚拟 USART 接口设计

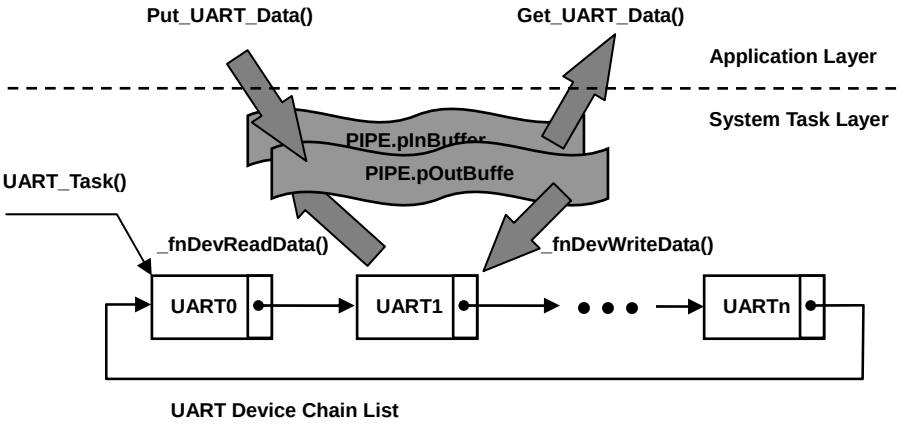
TWI 总线是一种对等总线，所有连接在总线上的设备都有能力成为主机，借助仲裁机制和协议状态机，我们可以将 TWI 协议进行封装，为所有设备建立虚拟的点对点通信管道。习惯上，这种管道会被虚拟成 UART 接口。对上层软件来说，TWI 协议和总线本身都是透明的，若干个 UART“直接”与目标设备相连。我们把这种机制称为虚拟 UART 接口封装。其模块结构如图 9.5 所示。

首先，我们需要设计虚拟 UART 的组织形式和工作方式。一个虚拟 UART 接口应该同时具有封装普通 UART 设备和虚拟 UART 设备的能力。虚拟 UART 设备通常来源于对其它串行外设的封装，而这些外设往往具有自己独特的工作方式，虚拟 UART 接口应该能够兼容并提供一定的手段访问或设置这些外设（例如 TWI 的初始化、配置等等）。各类不同的虚拟 UART 应该具有统一的操作方式，对上层应用来说，接口与接口之间实际操作的差异应该是透明的。因此，系统可以使用结构体将管道缓冲、具体硬件设备的数据读写操作函数指针、初始化函数和设置函数的指针封装在一起，并通过一个指向结构体的指针将所有的虚拟 UART 串联起来，方便系统通过一个 UART_Task()任务来访问所有的设备。参考代码如下：

```
/* 定义虚拟 UART 接口，该接口也可以用于封装实际的 UART 设备，和其他虚
   拟成 UART 的接口设备这些内容应该放置在 UART_Interface.h 中，对应的
   函数实体应该放置在 UART_Interface.c 中 */

/* 假设系统中有一个队列的抽象数据类型 */
#include “QUEUE.h”

/* 定义管道 Pipe*/
```

```
typedef struct StreamPipe
{
    QUEUE *pInBuffer;
    QUEUE *pOutBuffer;
}PIPE;

/* 定义虚拟 UART */
typedef struct UartInterface UART_INTERFACE;
struct UartInterface
{
    PIPE   *pBuffer;
    BOOL (*fnSetOption)(void *pOption);          /* 定义参数设置函数指针 */
    void   (*fnInitial)(void);                   /* 定义初始化函数指针 */
    BOOL (*_fnDevWriteData)(void *pData);        /* 定义指向硬件数据写入函数指针 */
    BOOL (*_fnDevReadData)(void *pData);         /* 定义指向硬件数据读取函数指针 */
    UART_INTERFACE *pNext;                       /* 定义设备链表 */
};

extern void UART_Task(void);                     /* 虚拟 UART 任务，可以加入到超级
                                                    循环或者操作系统中，用于完成整个
                                                    UART 设备链上设备的刷新 */
```

图 9.3 UART 驱动工作原理示意图（通用模式）



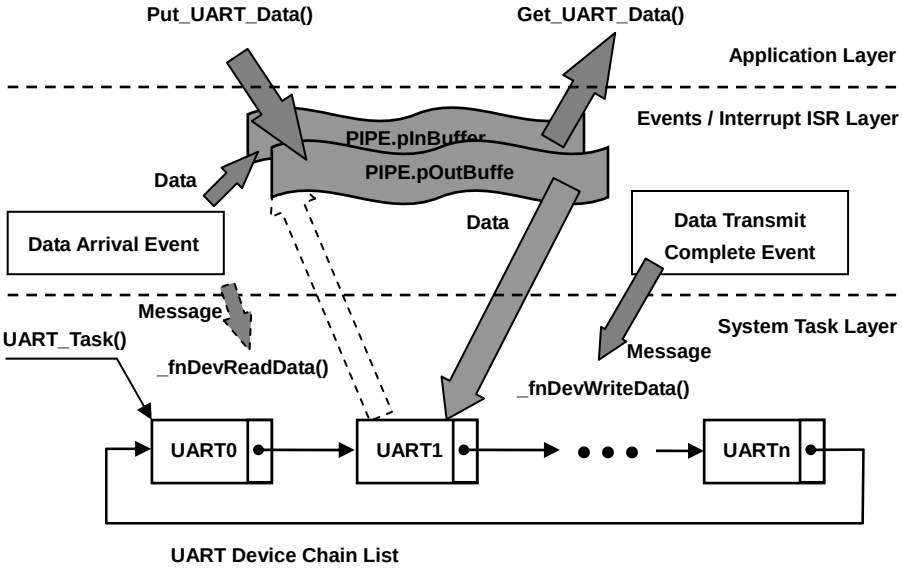
与普通的 **UART** 设备相同，用户可以通过缓冲区（**Pipe**）读写数据，而系统任务 **UART_Task( )** 则利用结构体中指针 **pNext** 构建起设备链，完成对所有底层硬件的读写操作：从发送缓冲中取出数据，通过函数指针**_fnDevWriteData** 向指定的设备写入数据；通过函数指针**_fnDevReadData** 从指定的设备中读取数据，并写入到对应的缓冲区中（如图 **9.3** 所示）。

对于类似 **TWI** 这样的外设来说，其基于中断状态机的硬件工作模式无法与传统的 **UART** 顺序读写相兼容，因而针对函数指针**_fnDevReadData()**和**_fnDevWriteData()**的操作都需

要特殊处理：数据的发送和接收可以被分别视为“数据到达事件”**(Data Arrival Event)**和“数据发送完成事件”**(Data Transmit Complete Event)**。在数据到达事件处理程序中，系统将接收到的数据直接放入管道中，同时向**_fnDevReadData()**函数发送一个消息——实际上，通常在这一情况下，消息连同对指针**_fnDevReadData** 的访问都是可以省略的（将函数指针**_fnDevReadData** 设置为 **NULL**）；在数据发送完成事件处理程序中，系统向**_fnDevWriteData()**发送一个消息，用以启动下一次数据传输（如**图 9.4** 所示）。

以 **TWI** 的底层驱动为例，在初始化时我们可以简单地将函数指针**_fnDevReadData** 设置为 **NULL**，即不需要 **UART_Task()**任务来维护硬件的读取操作。建立一个 **TWI** 状态迁移中段处理函数，编写主机发送和从机接收两个模式的状态机，加入对主机仲裁的处理。建立一个数据发送函数，用以启动 **TWI** 主机发送模式。如果主机仲裁成功，并正确的发送了数据，则该函数返回**TRUE**，否则返回**FALSE**。用函数指针**_fnDevWriteData** 指向该函数，**UART_Task()**任务将尝试从管道中读取一个数据并通过 **TWI** 总线发送出去。在 **TWI** 通讯过程中，如果处理器被其它主机寻址，进入了从机接收模式，则获得的数据应当直接送入管道中，等待用户的读取。

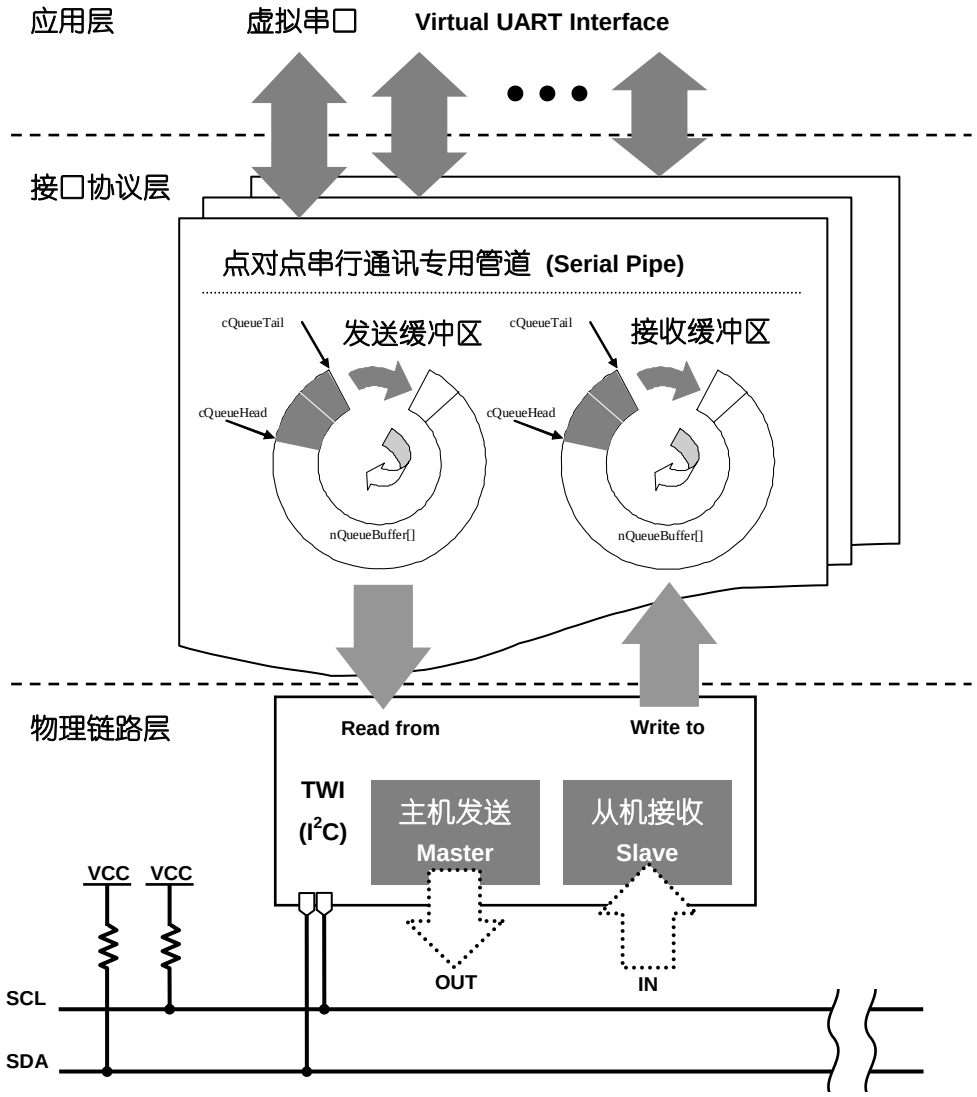
图 9.4 UART 驱动工作原理示意图（中断模式）



实际应用中，具体的硬件操作细节对上层接口来说是完全透明的，用户所看到的仅仅是一堆可以无差别使用的 **API** 函数和一个维护整个系统正常运行的**UART_Task()**。作为范例，这里仅列举了一些常用的接口函数原形，而省略了具体的实现过程：

```
/* 以下的接口函数实体都应该由统一的函数库
   UART_Interface.c 提供，用户无需编写这部分内容 */

extern BOOL Get_UART_Data                    /* 从指定的 UART 接口读取数据 */
                (
                    UART_INTERFACE *pUart,   /* 接口指针 */
```

图 9.5 片间TWI (I²C)虚拟串口通讯网络协议栈模块结构图




```
                    void *pData,                /* 指向存放读取数据的指针 */
                );

extern BOOL Put_UART_Data                       /* 向指定的 UART 接口写入数据 */
                (
                    UART_INTERFACE *pUart,      /* 接口指针 */
                    void *pData,                /* 指向存放要写入数据的指针 */
                );

extern void UART_Interface_INIT(void);          /* 初始化所有的 UART 设备 */

extern BOOL Set_UART_Property                   /* 修改对应 UART 接口的设置 */
                (
```

```
                    UART_INTERFACE *pUart,    /* 接口指针 */
                    void *pOption,            /* 指向配置信息的指针 */
                );

extern BOOL Add_UART_Device                   /* 向设备链中添加一个设备 */
                (
                    UART_INTERFACE *pUART     /* 接口指针 */
                );

extern BOOL Delete_UART_Device                /* 向设备链中删除一个设备 */
                (
                    UART_INTERFACE *pUART     /* 接口指针 */
                );
```

基于前面所述的结构，虚拟 **UART** 的使用方法如下：

1. 将所有的虚拟设备通过 **Add_UART_Device()**函数添加到系统中；
2. 通过函数 **UART_Interface_INIT()**完成所有设备的初始化；
3. 在超级循环或者周期性调用的任务中加入函数 **UART_Task()**，或者将 **UART_Task()** 函数修改为某一操作系统的任务，以系统高优先级任务的形式运行；
4. 通过 **Put_UART_Data()** 函数向指定的虚拟 **UART** 设备写入数据；通过 **Get_UART_Data()**函数读取数据；通过 **Set_UART_Property()**函数对具体的 **UART** 设备进行独立设置。
5. 如果某些设备允许动态添加或删除，用户可以通过函数 **Add_UART_Device()**和 **Delete_UART_Device()**来维护虚拟 **UART** 设备链。

9.3 学生用书及幻灯片注解

9.4 常见课堂问题 FAQ

9.5 参考资料

9.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168。**
  >> 21. 2-wire Serial Interface

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

9.5.2 应用手册（Application Note）

- **AVRxxx: xxx**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/docxxx.pdf**

9.6 背景知识

9.7 参考设计与实验方法


第10章 MISSION UPDATE

本章引言

当产品交到用户手上后，设计师还能完成对产品的维护工作吗？能，自豪地告诉顾客："把单片机连接到互联网上，它会自动更新程序！"

本章牵涉知识点

- 为什么要实现程序的更新？
- 实现程序更新的要点和难点是什么？
- 通信打包协议的应用。
- 亲手实现 UPDATE。
- ……

-316-

第十章　Bootloader 自编程

Rev 1.0.0.0

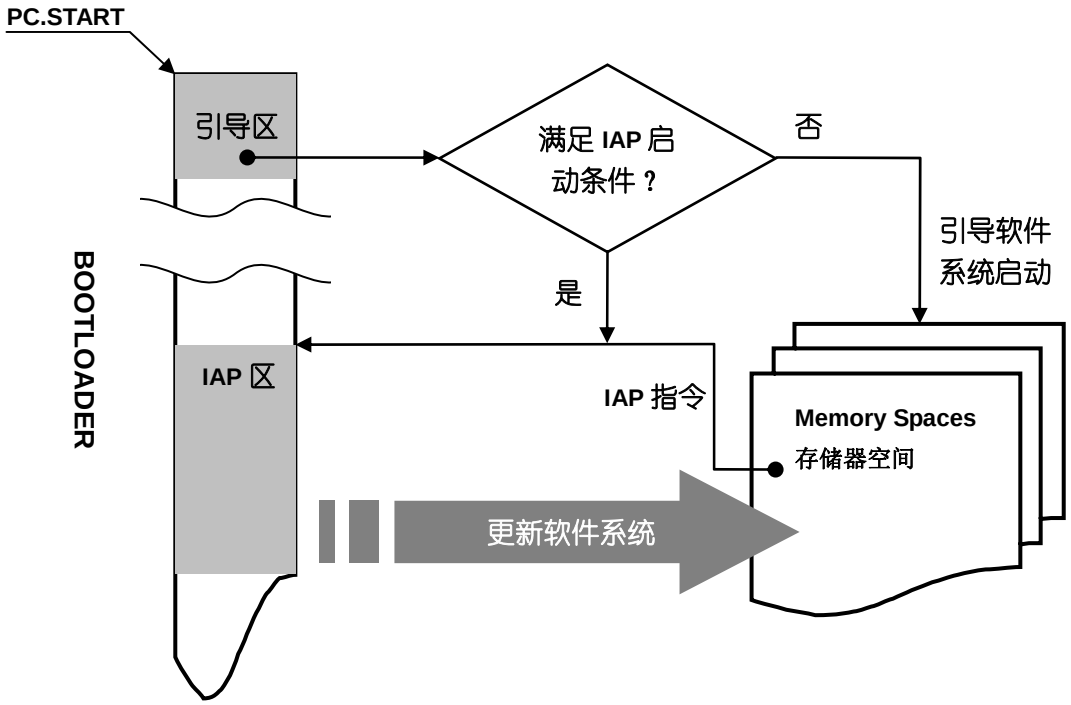
10.1 内容简介

- Bootloader 简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

10.2 Bootloader 简介

Bootloader 是一小段固化在嵌入式系统中的系统程序，用于引导用整个嵌入式软件系统的启动或者通过在应用编程实现系统软件的自动更新。一个典型的 **Bootloader** 工作原理如**图 10.1** 所示：

图 10.1 Bootloader 工作原理示意图



10.2.1 名词解释

- **在应用编程 (IAP)**

IAP 是在应用编程 **In Application Programming** 的英文缩写，嵌入式软件在系统运行时刻从外界获取数据，写入到指定的存储器中以实现软件系统更新的方式，称为在应用编程。

- **运行时可编程 (RWW)**

RWW 是运行时可编程 **Read-While-Write** 的缩写。**RWW** 区作为 **Bootloader** 区段时，可以对其它区域进行编程，而不影响本区域内程序的读取。**RWW** 对自身所在区域进行编程时，将暂时挂起 CPU。

- **编程时挂起 (NRWW)**

NRWW 是编程时挂起 **None-Read_While_Write** 的缩写。**NRWW** 一般不可作为 **AVR** 单片机的 **Bootloader** 区域，因为写入 **Flash** 的汇编指令 **SPM** 不能在 **NRWW** 中执行，**ATMega48** 例外。**ATMega48** 只有 **NRWW** 区域，当 **Bootloader** 程序运行时，将暂时挂起 **CPU**。

● 页面写入缓冲 (Page Writing Buffer)

AVR 的 Bootloader 的 Flash 写入操作是以页（Page）为单位的。数据必须先通过一个页缓冲区进行暂存，然后通过一个写入指令完成整页数据的更新。不同型号的 AVR 单片机，其页面大小是不同的。以 ATmega48 为例，其页面大小为 32 words（64Byte）；ATmega168，其页面大小为 64 words（128 Bytes）。读取操作不支持页面缓冲。

10.2.2 AVR 片上 Bootloader 简介

AVR 单片机支持用户通过特定的 Bootloader 区域实现对整个 FLASH 区域的更新。为了解决 FLASH 更新与 Bootloader 程序本身冲突的问题，系统引入了运行时可编程区域（RWW）和编程时挂起区域（NRWW）。Bootloader 程序一般被放置在一个专门的 RWW 区域中，其大小可以配置。ATmega48 的 FLASH 空间较小，因而没有 RWW 区：Bootloader 程序与普通的应用程序一起被放置在 NRWW 区域中，任何对其 FLASH 的写入操作都将暂时挂起 CPU 的执行。

普通模式下，系统会从 FLASH 的 0x0000 地址运行程序，中断向量表也被放置在同一起始位置。当用户通过熔丝配置了 RWW 区域的大小，并编程了 BOOTRST 熔丝位，系统在复位时将从 RWW 的起始区域开始程序的执行。因此，用户 Bootloader 应该被放置在 RWW 的起始地址上。一般情况下，系统不会在 Bootloader 程序中使用中断处理；否则用户就需要通过设置 CPU 控制寄存器 MCUCR（或 MCUCSR）的 IVSEL 位将中断向量表放置到 RWW 区域中。

ATmega48/88/168 提供了一整套实现 Flash 编程的寄存器和操作时序，具体内容用户可以参照最新的数据手册（Datasheet）获取更多信息。如果用户同时安装了 AVR Studio4 和 AVR 专用的 GCC 编译器（Winavr），将会获得系统对 Bootloader 功能 C 语言库函数级别的支持，无须关心具体的底层汇编接口和操作时序。

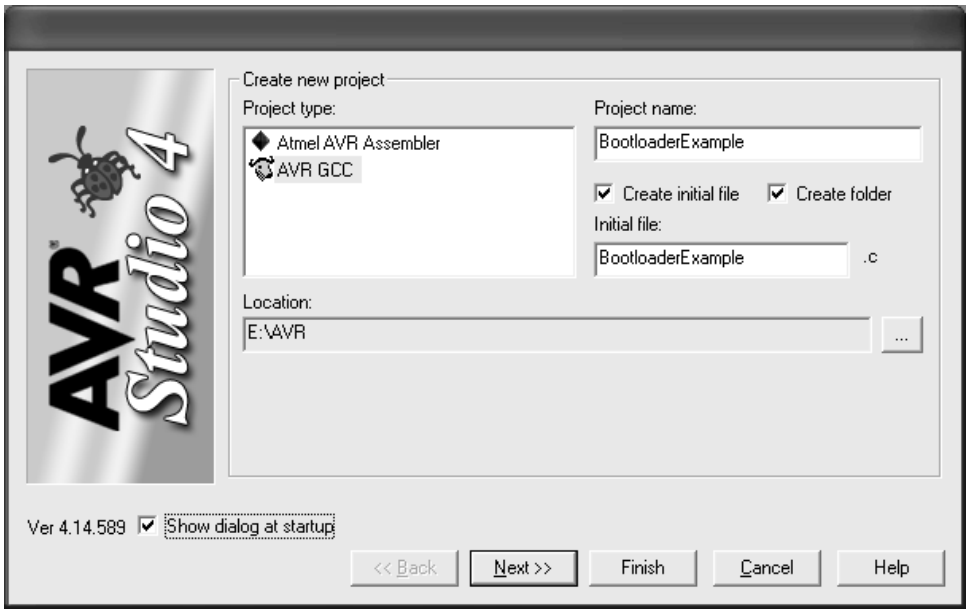
AVR GCC 通过库函数 boot.h 提供了一些列常用的接口。例如，用户可以通过宏 BOOTLOADER_SECTION 修饰变量或函数，告知编译器将这些内容放置到 Bootloader 区域中。针对 FLASH 的擦除、编程操作都有专门的函数或宏来实现。用户仅仅需要根据 RWW 的熔丝设置，通过 AVR Studio4 在工程中为“.bootloader”区段指定具体的地址（“.bootloader”区段由宏 BOOTLOADER_SECTION 通过在线汇编的方式引入 C 语言源程序，如果不具体指定该区段所定义的起始地址，“.bootloader”指定的函数和变量将被放置到普通的“.text”区段中）。具体的函数接口定义及相关说明，用户可以通过阅读定义在 boot.h 的帮助信息来获得，本文将不作详细论述。您也可以参考 10.2.3 图解 AVR Studio4 建立 Bootloader 章节来建立自己的 IAP 工程实例。

```
/* boot.h 常用函数接口和宏定义 */
# define BOOTLOADER_SECTION          __attribute__ ((section (".bootloader")))

#define boot_page_fill(address, data)    __boot_page_fill_normal(address, data)
#define boot_page_erase(address)         __boot_page_erase_normal(address)
#define boot_page_write(address)         __boot_page_write_normal(address)
#define boot_rww_enable()                __boot_rww_enable()
```

10.2.3 图解 AVR Studio4 建立 Bootloader

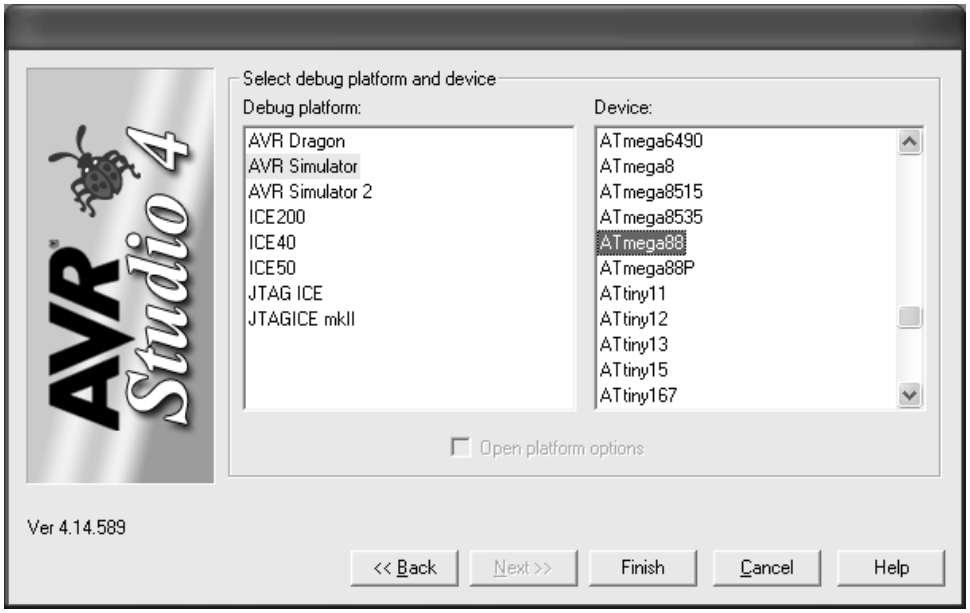
第一步：新建工程。打开 **AVR Studio4**，通过菜单 **Project->New Project** 新建一个 **AVR GCC** 工程（如**图 10.2** 所示）。图中例子所用的工程名为 **BootloaderExample**。系统会自动指定一个同名的源程序文件 **BootloaderExample.c**。

图 10.2 New Project 窗口



第二步：选择仿真器。设置 **AVR** 模拟器（**Simulator**），并选择目标芯片的型号（如**图 10.3** 所示）。图中例子选择 **AVR Simulator** 来实现 **ATmega88** 的仿真。

图 10.3 AVR 仿真器设置窗口

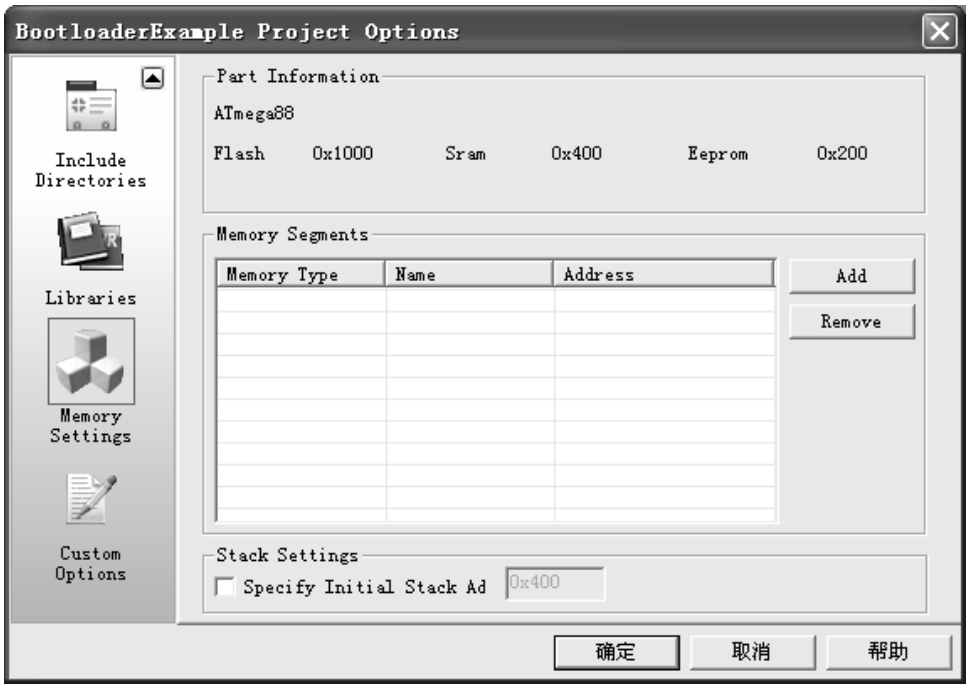


第三步：编码。在编辑窗口中输入 **Bootloader** 代码（如**图 10.4** 所示）。图中的代码演示了向 **ATmega88** 的 **0x0040**（**Flash** 页的第二页）地址依次写入 **0x00~0x40**。

图 10.4 Bootloader 工程代码窗口

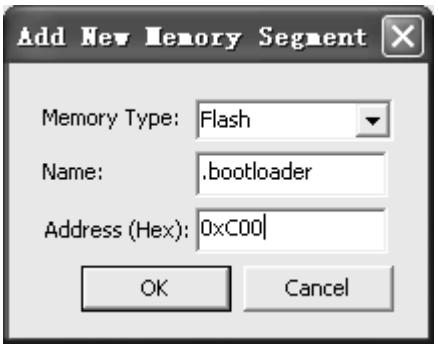
第四步：**定义 Bootloader 区段。**选择菜单 **Project->Configuration Options->Memory Settings** 打开 **Memory** 区段设置窗口（如**图 10.5** 所示）。

图 10.5 AVR 存储器区段设置窗口



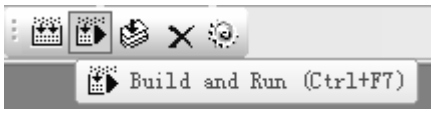
单击按钮 **Add**，在弹出的窗口中添加“**.bootloader**”区段，将我们的工程放置到目标芯片的指定 **Bootloader** 区域中（如和**图 10.6** 所示）。图中例子使用 **ATmega88**，**Bootloader** 区域大小为 **2K word**，因此需要将地址 **Bootloader** 段绑定到 **FLASH** 地址 **0x0C00** 上。请注意图中设置的格式和字母大小写。

图 10.6 AVR 存储器区段设置窗口



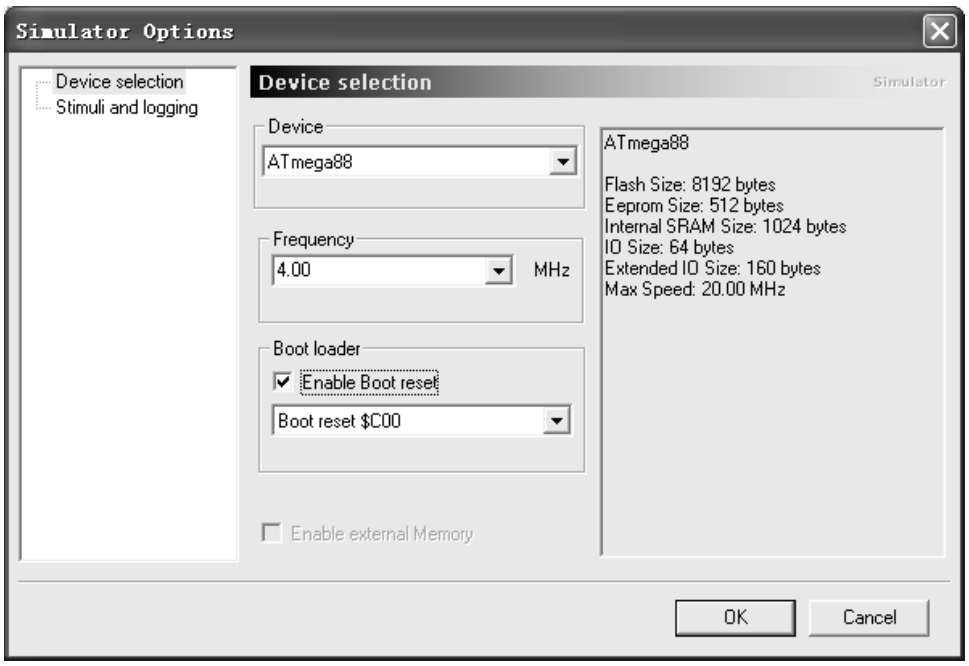
第五步：配置仿真器。通过菜单项 **Build->Build and Run** 或者单击工具栏快捷按钮（如**图 10.7** 所示），对程序进行编译并启动仿真器。

图 10.7 编译工具栏



启动仿真器后，选择菜单 **Debug-> AVR Simulator Options** 设置仿真器的工作环境，包括仿真器件、工作频率和 **Bootloader** 设定。进行 **Bootloader** 项目仿真时，一定选中 **Enable Boot reset** 选项，并在下拉列表中选择正确的 **Bootloader** 区段起始地址（如**图 10.8** 所示）。这一地址应该与我们在上一步骤中设置的“**.bootloader**”区段地址保持一致，否则程序将无法正常运行。有时候，我们忘记了将 **Enable Boot reset** 选项选中，程序在仿真器中似乎也能运行，这是一种假象，因为此时从 **FLASH** 起始地址 **0x0000** 到**.bootloader** 段之间数据都为 **0xFFFF**，当程序从默认的 **0x0000** 地址开始运行时，**AVR** 将 **0xFFFF** 视作无效指令（**Data or unknown opcode**）而直接忽略，直到进入“**.bootloader**”区段。

图 10.8 仿真配置窗口

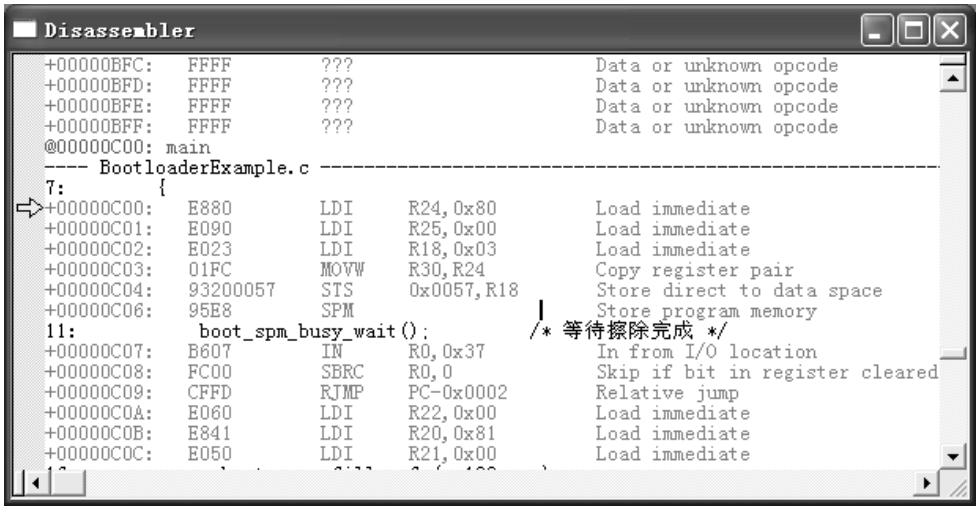


第六步： **调试**。选择菜单 **Debug->Reset** 重置工程。通过视图工具栏的汇编窗口快捷按钮（如**图 10.9** 所示），切换到汇编视图。

图 10.9 视图工具栏（汇编视图按钮）

检查程序是否从指定的“**.bootloader**”区段开始执行、代码是否被放置到了正确的位置（如**图 10.10** 所示）。

图 10.10 汇编视图



第六步：观察结果。设置断点，运行程序。通过视图工具栏的存储器视图快捷按钮（如**图 10.11** 所示），切换到存储器视图，并观察 **Bootloader** 程序是否运行正常。图**图 10.12** 中，字地址 **0x0020~0x003F**（也就是字节地址的 **0x0040~0x007F**）被写入了 **0x00~0x3F** 的数字，符合示例程序的设计意图。

图 10.11 视图工具栏（存储器视图按钮）

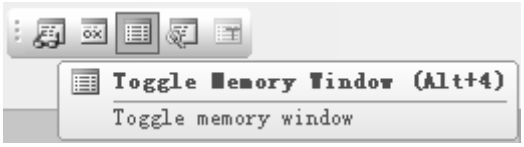
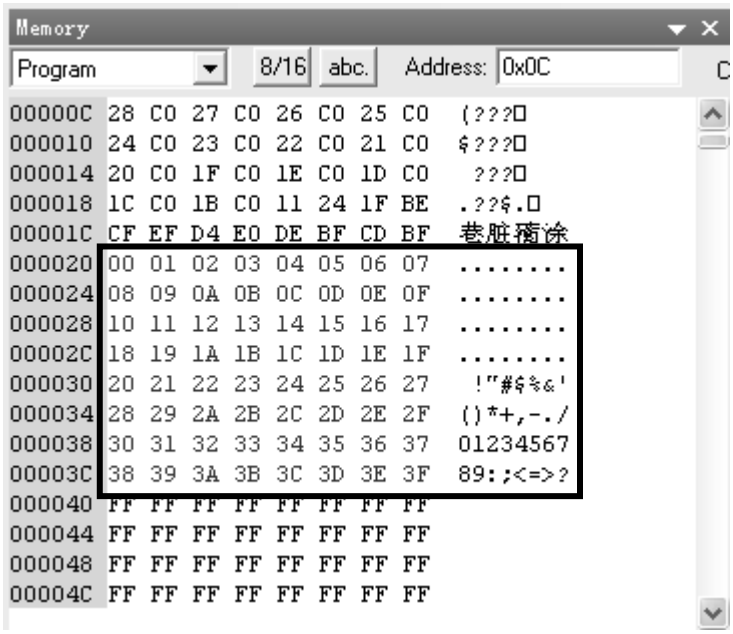


图 10.12 存储器视图



10.3 学生用书及幻灯片注解

10.4 常见课堂问题 FAQ

10.5 参考资料

10.5.1 数据手册（Datasheet）

- 数据手册 **ATmega48/88/168**。

  >> 25. Self-Programming the Flash, ATmega48

  >> 26. Boot Loader Support –

  Read-While-Write Self-Programming, ATmega88 and ATmega168

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf**

10.5.2 应用手册（Application Note）

- **AVR100: Accessing the EEPROM**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc0932.pdf**

- **AVR102: Block Copy Routines**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc0933.pdf**

- **AVR101: High Endurance EEPROM Storage**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2526.pdf**

- **AVR104: Buffered Interrupt Controlled EEPROM Writes**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2540.pdf**

- **AVR105: Power Efficient High Endurance Parameter Storage in Flash Memory**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2546.pdf**

- **AVR106: C functions for reading and writing to Flash memory**

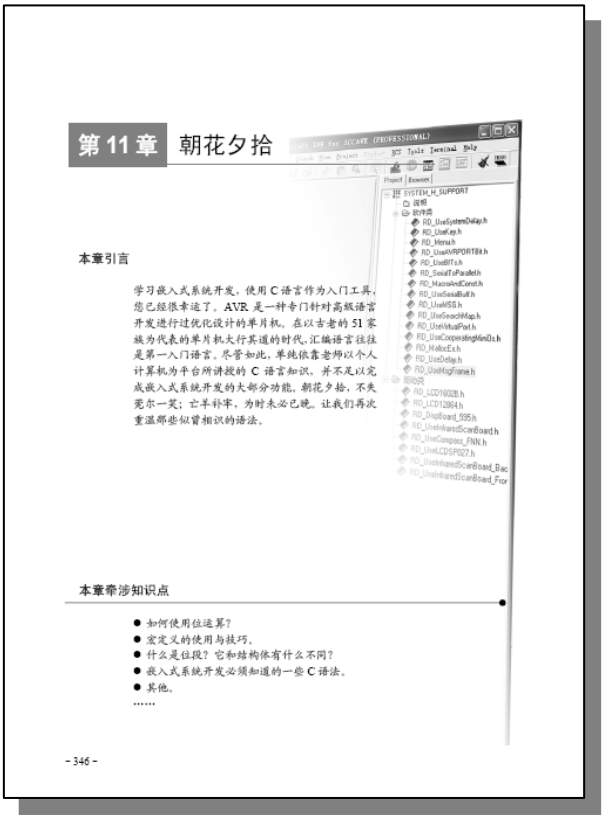
下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2575.pdf**

- **AVR103: Using the EEPROM Programming Modes**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc2578.pdf**

10.6 背景知识

10.7 参考设计与实验方法


第 11 章 朝花夕拾

本章引言

学习嵌入式系统开发，使用 C 语言作为入门工具，您已经很幸运了。AVR 是一种专门针对高级语言开发进行过优化设计的单片机。在以古老的 51 家族为代表的单片机大行其道的时代，汇编语言往往是第一入门语言。尽管如此，单纯依靠老师以个人计算机为平台所讲授的 C 语言知识，并不足以完成嵌入式系统开发的大部分功能。朝花夕拾，不失莞尔一笑；亡羊补牢，为时未必已晚。让我们再次重温那些似曾相识的语法。

本章牵涉知识点

- 如何使用位运算？
- 宏定义的使用与技巧。
- 什么是位段？它和结构体有什么不同？
- 嵌入式系统开发必须知道的一些 C 语法。
- 其他。

……

- 346 -

第十一章　嵌入式 C 语言

Rev 1.0.0.0

11.1 内容简介

- 嵌入式 C 语言简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

7.2 嵌入式 C 语言简介

C 语言，作为一门高级语言，在越来越多的嵌入式软件开发中得到应用，其优点是明显的：减少开发周期、易维护、可移植性强、代码易重用；其缺点也随着编译技术以及嵌入式微处理技术的发展越发显得次要，比如：相对汇编程序来说实现相同功能的 C 语言代码生成的目标代码可能会更大一些，其执行效果可能会稍微慢一些，等等。

实现相同的功能，C 语言的表达方式不同，经过编译所产生的结果往往大相径庭。经过优化的 C 语言代码配合性能良好的编译器，可以获得与汇编语言相差无几的效果。本章将结合 AVR 微控制器的特点，着重介绍高效率 C 语言编程的一些基本原则和注意事项。

7.2.1 名词解释

- **位屏蔽（Bit-mask）法位运算**

位屏蔽法位运算是一种使用二进制掩码配合算术逻辑运算（与运算“&”、运算“|”、非运算“~”和异或运算“^”）进行的位运算。这种方法不存在大/小端系统兼容性问题。

- **位域（Bit-field）法位运算**

位域法位运算是一种使用位域配合算术逻辑运算进行位运算的方法。其使用收到存储器大/小端排列方式以及内存地址对其方式的影响，兼容性较差。

- **标准变量类型**

由于 C 的基本变量类型并没有规定 int 型变量的长度，因此，在某些平台中 int 型变量长度为 16 位、某些平台是 32 位。为了解决 C 语言代码移植时 int 型变量宽度问题带来的不确定性，ANSI-C 在 C99 标准中加入了一个头文件 stdint.h 用于指定对应平台下 int 型变量的长度。例如，uint16_t 表示长度为 16 位的无符号整数；int8_t 表示长度为 8 位的有符号整数。具体细节请参考文档：ISO/IEC 9899:1999 (E) 。

- **在线汇编**

嵌入式 C 语言中为了实现某些底层硬件的操作而在 C 语言源代码中加入对局部使用汇编语言的支持称为在线汇编。

7.2.1 基本数据类型的标准定义

在不同字长的微处理器中整型变量 int 往往具有不同的长度，例如在 AVR 单片机 8 位系统中 int 型变量的长度为 16 位二进制；在 32 位微处理器 AVR32 中 int 型变量长度为 32 位。在某些编译器中 char 型变量默认是有符号的，在某些系统中 char 型变量是无符号的，可以表示 0~255 之间的任意整数。为了解决不同平台间代码的移植问题，与 ANSI-C99 标准兼容的编译器都会为专门的处理器平台提供了一个标准的库函数 stdint.h。该头文件为我们定义了常用的基本变量类型：

```
/* 针对 AVR 单片机 8 位系统的专属的 stdint.h */
typedef    unsigned char      uint8_t;
typedef    signed char        int8_t;
```

```
typedef    unsigned short     uint16_t;
typedef    signed short       int16_t;
typedef    unsigned long      uint32_t;
typedef    signed long        int32_t;
……
```

编写程序时，我们应该统一使用 **C99** 标准规定的标准数据类型以提高代码在不同平台间的可移植性。

7.2.2 自定义布尔类型

ANSI-C 中并没有定义布尔型变量（**Boolean**），我们可以利用 **unsigned char** 型变量自行进行定义，以下定义方式是嵌入式系统中常见的一种：

```
/* 自定义的布尔型变量 */
typedef    unsigned char      BOOL;

/* 定义布尔型常量 */
# define   TRUE               0x01
# define   FALSE              0x00
```

根据布尔型变量的精确定义，**FALSE** 应该为“**0**”值；**TRUE** 应该为非“**0**”。非“**0**”值可以是“**1**”但也可以是其它任何的非“**0**”整数，因此将 **TRUE** 直接定义为“**1**”是不妥当的。另一种推荐的定义方式如下：

```
/* 定义布尔型常量 */
# define   FALSE              0x00
# define   TRUE               (!FALSE)
```

注意，正因为存在上面两种不确定的定义方式，为了避免带来歧义尽可能不要在逻辑表达式中与 **TRUE** 进行“**==**”或“**!=**”的判断。下面的代码是非常不妥当的：

```
/* 判断布尔型变量 bFlag 是否为 TRUE */
if (bFlag == TRUE)
{
    ……
}
```

正确的写法应为：

```
if (bFlag != FALSE)
{
    ……
}
```

或者简写为：

```
if (bFlag)
{
    ……
}
```

有时候我们会在别人的代码中看到类似下面的代码：

```
if (FALSE == bFlag)
{
    ……
}
```

这种方法可以有效防止将“==”误写为“=”，因为表达式的左边是一个常量，如果错误发生，编译器会提示：“**lvalue required**”，我们就能发现这一误写的逻辑表达式。

7.2.3 掩码位操作

不同的处理器在内存的对齐方式上可能存在大端对齐（**Big-endian**）和小端对齐（**Little-endian**）的差别。使用位域进行位运算时，随着对齐方式的不同运算的结果也不同，例如，对于下面的位域，大端对齐和小端对齐中产生的结果是完全不同的；

```
/* 定义一个联合体，用于观察结果 */
union
{
    struct
    {
        unsigned   BIT0 :     1;
    }LSB;
    uint8_t      Value;
}Example;

/* 测试代码 */
Example.Value = 0x00;                    /* 对变量进行初始化 */
Example.LSB.BIT0 = 1;                    /* 尝试将 BIT0 设置为 1 */
printf( “Value = %x”,Example.Value);     /* 以 HEX 方式输出 Example 的值 */
```

在小端对齐的系统中，**Example.Value** 的值为 **0x01**；而在大端对齐的系统中，**Example.Value** 的值为 **0x80**，可见使用位域进行位运算时可能存在歧义，因而不适合代码的跨平台复用。使用掩码位运算则不存在这一问题，例如：

```
/* 定义一个位掩码 */
# define MASK_BIT0            0x01
```

```
/* 定一个侧使用的变量 */
uint8_t     Example = 0x00;

/* 测试代码，将 LSB 设置为 1 */
Example |= MASK_BIT0;                    /* 将 LSB 设置为 1 */
printf(“ Value = %x”,Example);           /* 输出测试结果 */
```

通过测试可以发现，无论系统是大端对齐还是小端对齐方式，**Example** 的值都为 **0x01**。关于大端对齐和小端对齐的具体内容请参考第十二章存储器与指针。常见的掩码位运算有置位、清零、取反和取数四种。置位运算通常配合“**|=**”实现仅针掩码指定位进行置位操作，例如：

```
/* 将 BIT0 设置为 1 */
Example |= MASK_BIT0;                    /* 仅仅将 BIT0 设置为 1 */
```

清零运算通常配合“**&=**”并将掩码取反后实现将指定二进制位进行清零操作，例如：

```
/* 将 BIT0 清零*/
Example &= ~MASK_BIT0;                   /* 仅仅将 BIT0 清零 */
```

取反运算通常配合“ˆ=”实现将指定二进制位进行取反操作，例如：

```
/* 将 BIT0 取反 */
Example ^= MASK_BIT0;                    /* 仅仅将 BIT0 取反 */
```

取数运算用于从指定的变量中单独取出指定的一个或一些二进制位，例如：

```
/* 判断 BIT0 的值是否为 0 */
if ((Example & MASK_BIT0) == 0)          /* 从 Example 中提取 BIT0 */
{
……
}
```

用于位运算的掩码通常有两种方式产生：一种是事先定义好指定的宏，并通过宏的命名为指定的掩码一个明确的意义，很多针对寄存器操作的掩码采用的都是这种方式；另一种是通过参数宏的方式在需要的时候临时定义掩码。例如：

```
/* 使用参数宏的方式来产生掩码 */
# define    _BV(__VALUE)                (1 << (__VALUE))
```

另外一种实现同样功能的参数宏：

```
# define    BIT(__VALUE)            (1 << (__VALUE))
```

使用时，在**_BV()**或 **BIT()**中填入的整数，用于产生针对整数所指定二进制位的掩码，例如 **BIT(0)**等效于针对 **BIT0** 的掩码，其值为 **0x01**。

7.2.4 基于 AVR 单片机的嵌入式 C 语言

7.2.4.1 指针与寻址

AVR 拥有 **32** 个通用寄存器，每个通用寄存器都和普通 **51** 中的累加器相同。在一个指令周期中，系统可以从寄存器文件中将任意两个寄存器的值送入运算器（**ALU**）、完成指定的运算并将计算结果送回寄存器文件。某些寄存器可以被两两组合为 **16** 位指针，用于辅助

系统寻址操作。在某些 **AVR** 型号中，配合额外的寄存器 **RAMPX** 可以生成一个 **24** 位的指针，用于完成 **16M** 地址空间的寻址。例如：寄存器 **R31** 和 **R30** 构成指针 **Z**、**R26** 和 **R27** 构成指针 **X**，这两个指针用于辅助 **C** 语言进行间接寻址。对于下面的 **C** 语言代码，由于 **X** 和 **Z** 指针的协助生成的汇编代码相当高效：

```
;/* C 语言代码 */
;char *point1 = &table[0];
;char *point2 = &table[49];
;*point1++ = *--point;
;/* IAR 编译器生成的汇编代码 */
LD R16 ,-Z              ;先 Z 自减 1 以后，将 Z 指针指向空间的内容送入 R16
ST X+   ,R16            ;先将 R16 中的数据送入 X 指针指向的空间，然后 X 自增 1
```

AVR 还为高级语言提供一个堆栈指针 **SP**，由寄存器 **R28** 和 **R29** 组成。通过这些指针，**C** 语言的寻址操作将得到极大的优化，例如，以下操作可以在 **2** 个指令周期中完成：

- 使用数组或指针访问数据元素

```
*point = 0x00;
```

- 结构体操作
- 使用指针访问元素后自增一；

```
*point++ = 0x00;
```

- 在使用指针访问元素前自减一；

```
*--point = 0x00;
```

7.2.4.2 整数运算

AVR 嵌入式 **C** 语言中，**16** 位整数运算通常可以在 **2** 个时钟周期内完成；**32** 位整数运算通常可以在 **4** 个时钟周期内完成。例如，对于两个 **16** 位整数进行相加，将消耗 **6** 个时钟周期：

```
;/* C 语言加法源程序示例（a,b 都是局部变量） */
;int a = 1;
;int b = 2;
;a = a + b;
;/* IAR 生成的汇编代码 */
;2 周期 16 位整数初始化
LDI    R16, 1           ;装载 a 的低 8 位，耗费 1 个指令周期
LDI    R17, 0           ;装载 a 的高 8 位，耗费 1 个指令周期
;2 周期 16 位整数初始化
LDI    R18, 2           ;装载 b 的高 8 位，耗费 1 个指令周期
LDI    R19, 0           ;装载 b 的高 8 位，耗费 1 个指令周期
;2 周期 16 位整数加法
ADD    R16, R18         ;进位加法，耗费 1 个指令周期
```

```
ADC     R17, R19          ;无进位加法，耗费 1 个指令周期
```

在 **AVR** 嵌入式 **C** 语言软件开发中，如果可能尽可能使用小数据类型，以换换取较高的代码执行效率和较小的代码尺寸。

7.2.4.3 代码尺寸优化

- 在不影响功能的前提下，尽可能提供编译器的优化等级；
- 尽可能使用局部变量，局部静态变量不在此列；
- 尽可能使用小的数据类型，尽可能使用 **unsigned** 来提高小数据类型的整数表达上限；
- 如果一个非局部变量仅仅被一个函数使用，应该将该变量声明为局部静态变量；
- 尽可能将多个全局变量或局部静态变量分类组合成结构体，以增加使用指针进行间接寻址的机会，提高访问效率；
- 使用指针加偏移量的方法或者结构体地址绑定的方法访问应设在 **I/O** 空间的存储器。
- 对于非映射的 **I/O** 地址，尽可能直接访问。
- 如果可能，请使用 **do{}while()**结构。
- 如果可能，请使用递减型 **while()**循环，并且使自减运算“**--**”优先于逻辑判断进行；
- 当所调用的函数只有 **2~3** 句代码时，可以使用宏定义的方法来实现内联函数；
- 对于常用的功能，尽可能的封装为函数；
- 极端情况下，如果编译器支持，可以逐个函数的设置优化等级；
- 如果可能，尽量不要在中断处理程序中调用其它函数；
- 在某些编译器中，使用小内存模式。

7.2.4.4 SRAM 尺寸优化

- 所有常量和字符串都应该放置在 **FLASH** 中；
- 尽可能避免使用全局变量和局部静态变量；
- 在函数内部，使用中括号来限制某些大数据类型的有效范围；
- 正确的评估软件堆栈和硬件堆栈的大小；

7.3 学生用书及幻灯片注解

7.4 常见课堂问题 FAQ

7.5 参考资料

7.5.1 应用手册（Application Note）

- **AVR035: Efficient C Coding for AVR**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1497.pdf**

- **AVR034: Mixing C and Assembly Code with IAR Embedded Workbench for AVR**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1234.pdf**

- **AVR072: Accessing 16-bit I/O Registers**

下载地址：**http://www.atmel.com/dyn/resources/prod_documents/doc1493.pdf**

7.5.2 参考文献

- **ISO/IEC 9899:1999 (E)**

7.6 背景知识

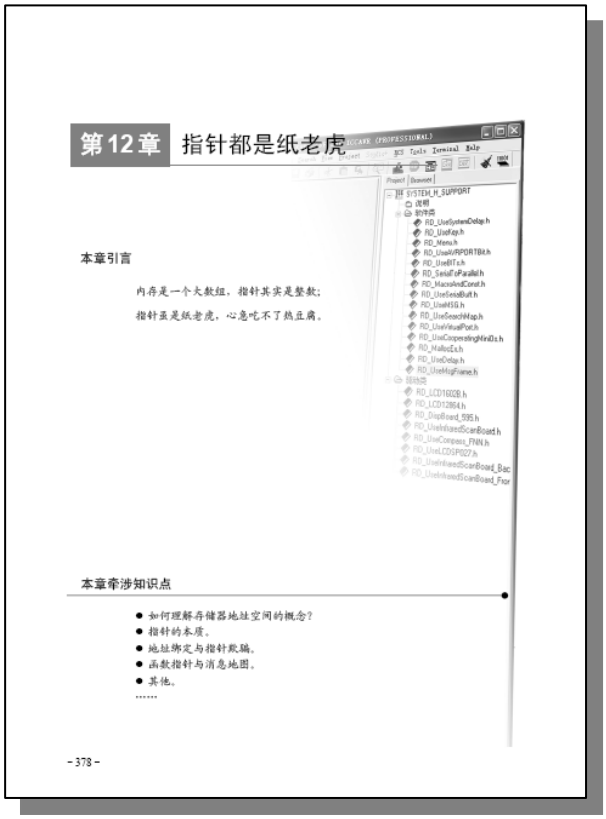
7.6.1 编译原理

本章内容牵涉到编译原理的相关内容。

7.6.2 C/C++语言

本章内容牵涉到 **C/C++**语言与应用的相关内容。

7.7 参考设计与实验方法



第十二章　存储器与指针

Rev 1.0.0.0

12.1 内容简介

- 存储器操作简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

12.2 存储器操作简介

12.2.1 名词解释

● **尾端 (Endianness)**

尾端由 **Danny Cohen(Cohen 1980)**由一篇论文引入计算机界，作者注意到计算机体系结构基于字节寻址和整数定义在通信系统中的不同定义分为“大尾端”**(Big-endian)**和“小尾端”**(Little-endian)**两个阵营。一个十六进制整形数据 **0x12345678** 传递给另外一个采用不同尾端的计算机系统时可能会被理解为 **0x78563412** 甚至 **0x1E6A2C48**。这就是尾端命题。尾端直接决定着数据在存储器内的组织形式。

● **地址空间 (Address Space)**

存储器中，由存储单元地址或集合所构成的集合称为地址空间。

● **线性地址空间 (Linear Address Space)**

如果一个地址空间中所有的元素可以组成一个由整数构成的唯一闭区间，则称该地址空间为线性地址空间。

● **扁平地址空间 (Flat Address Space)**

一个地址空间可以由多个不同纬度的线性地址空间构成，当地址空间中只有唯一的一个线性地址空间时，或者地址空间中不同的线性地址空间可以被影射到同一个彼此不相交的纬度时，则称该地址空间为扁平地址空间。

12.2.2 内存组织方式

除去字节、字符、字符串和字节流，**C** 语言中几乎所有的数据类型，准确说是这些数据类型在存储器内部的组织方式都要受到尾端的影响。出于历史的原因，**IBM** 及其兼容的计算机系统采用大尾端系统，习惯上又被称为大端对齐系统；**DEC** 和 **Intel** 及其兼容的计算机系统采用小尾端系统，习惯上又称为小端对齐系统。

只有在进行字节寻址时才会有尾端的问题。早期计算机工程师在 **20** 世纪 **60** 年代晚期都是对单字节进行操作：指令、整型和内存宽度都是相同的字宽度。这样的计算机系统不存在尾端的概念，只有字在内存中的顺序、以及字内部二进制位序的问题。

我们在书写十进制数字式，都是先写高位再写低位；对数据的读取也采用同样的方式。为了符合这一习惯，**IBM** 的工程师们尝试寻找一种以字为单位的数值表示方法。当我们按照习惯将内存单元按照从左到右地址增加的顺序依次排列开来，可以发现字的数值是直接可读的，或者说字的表示形式符合人们从左至右、从高位开始读取数字的习惯。（如**图 12.1** 所示）此时，由于存储器从左到右地址是依次递增的，字节内二进制位序从左到右也是依次递增的关系，因而在一个字内，二进制位序实际上与数值的位序正好相反。这就是大尾端的形成原因。

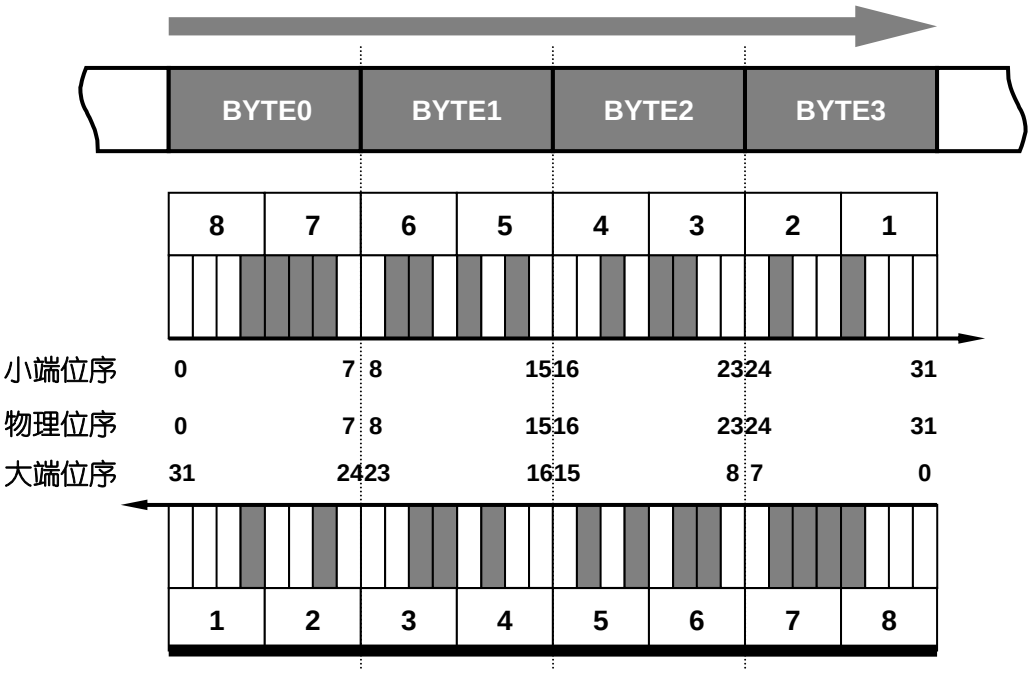
另一方面，我们也习惯于很自然的给每一个字节的二进制位赋予一个权值，比如**BIT0** 的权值为 **1**、**BIT1** 的权值为 **2**、**BIT2** 为 **4**……**BIT7** 为 **128**，这实际上就是将字节的二进制位序作为该位权值以 **2** 位底的幂指数，简单的说，**BITn**的权值就等于 2^n。用这种顺序可以非常自然的由多个字节组成较大的数据类型，这就是小尾端形成的原因。

12.2.3 硬件与尾端

在物理上，大尾端以字为单位组织数据（如图 **12.1** 大尾端图示的粗横线所示），字内的位序与实际的物理位序完全相反。大尾端认为，在字内第一字节物理位 **0** 为最高位**(MSB,Most Significant Bit)**；第四字节物理位 **7** 为最低位**(LSB, Least Significant Bit)**。小尾端以字节为单位组织数据，数据元素的字节序与存储器的字节序相同，字节内部的位序也与物理位序相同：物理位 **0** 为最低位 **LSB**、物理位 **7** 位最高位 **MSB**。

对比图 **12.1** 中两种尾端对十六进制数字 **0x12345678** 的组织形式很容易发现：当我们把存储器按照从左到右地址依次增加的顺序排列开来的时候，小尾端系统读取到的数据依次为 **0x78**、**0x56**、**0x34** 和 **0x12**；大尾端读取到的数据依次为 **0x12**、**0x34**、**0x56** 和 **0x78**，符合人们的阅读习惯。在大尾端系统中，数据的组织必须以字为单位，否则因为逻辑位序与物理位序的冲突而导致数据的混乱。在大尾端系统中，**CPU** 内核与存储器往往都是按照大尾端形式组织数据的；在小尾端系统中，**CPU** 与存储器往往都是按照小尾端形式组织数据的；外设与总线的尾端往往可以根据不同的设计需求而有所不同。

图 12.1 存储器大尾端和小尾端结构示意图（表示十六进制数字 0x12345678）



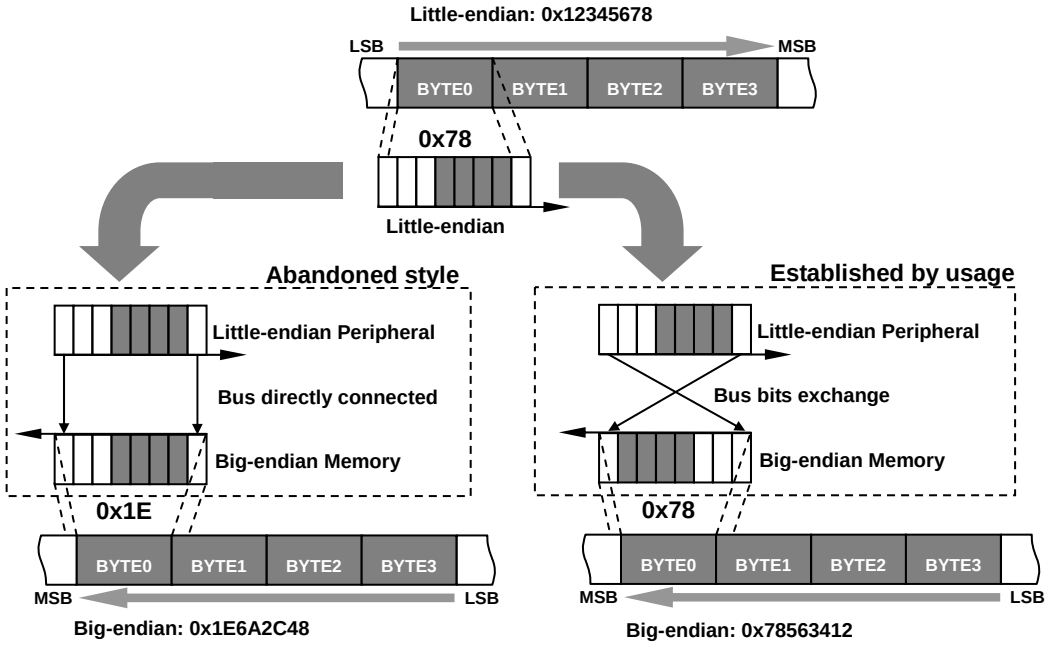
12.2.4 设备与尾端

虽然存储器和 **CPU** 存在尾端上的分歧，不知从何时起人所有的外设控制器都约定俗成的使用小尾端方式进行数据交换，有些外设甚至可以设定通讯时先传输字节高位还是线传输数据底位。这对大尾端计算机系统来说也许是个尴尬的问题，它们不得不将外设的数据在连接系统总线时做一次字节内位序的交换，如果不这么做造成的后果将更加严重（如图 **12.2** 所示）。

在不同尾端计算机系统间进行数据交换，如果通讯双方都以为对方的尾端与自己相同，那么就会造成对数据理解上的分歧。前面说过，由于外设控制器统一使用小尾端方式

进行数据交换，事实上造成了不同尾端系统进行数据交换时仅对于大数据类型的字节序存在歧义。

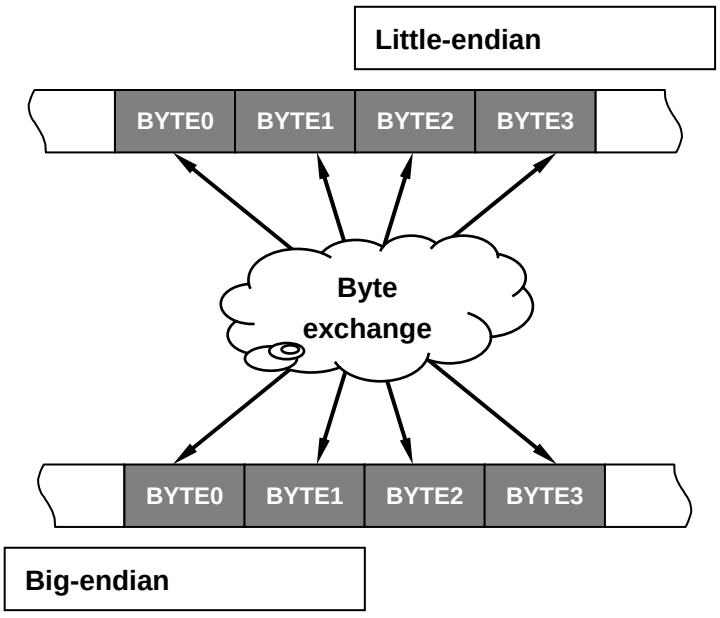
图 12.2 不同尾端系统数据交换示意图



12.2.5 软件与尾端

除了极少数情况外，在软件层面上讲，不同尾端的计算机系统进行数据通讯时，仅需要以字（或半字）为单位进行字节序的颠倒而已。（如图 12.3 所示）

图 12.3 不同尾端系统间数据交换的软件处理示意图



12.2.6 存储器对齐方式

很多计算机系统为了寻址的便利，或者由于数据操作类型的限制（例如，**AVR32** 每次只能以 **32** 位数据为单位进行读写操作）要求用户将数据或变量尽可能的对齐到指定的地址上。**32** 位计算机往往要求用户将数据对齐到字**(WORD，4Byte)**或者半字**(Half WORD,2Byte)**，**AVR** 作为 **8** 位微处理器，以字节为单位进行寻址，因而对用户没有数据对齐的要求。使用 **C** 语言进行软件开发时，编译器会自动对用户声明的变量按照处理器的要求进行对齐，例如，针对下面的全局变量声明，字对齐模式下的内存示意如图 **12.4** 所示；半字对齐模式下的内存示意如图 **12.5** 所示；字节对齐模式下的内存示意如图 **12.6** 所示：

```
/* 使用 C99 标准数据类型 */
#include <stdint.h>

/* 全局变量声明 */
uint8_t     a;
uint16_t    b;
uint32_t    c;
```

图 12.4 字对齐模式（4 字节对齐）

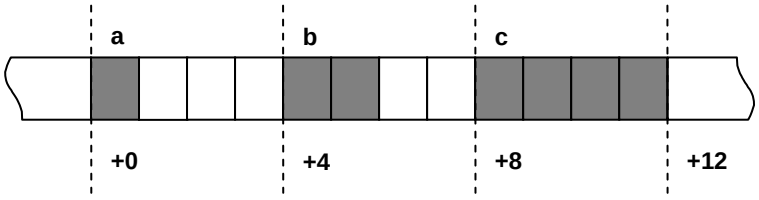


图 12.5 半字对齐模式（2 字节对齐）

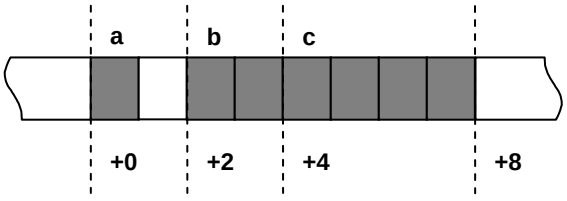
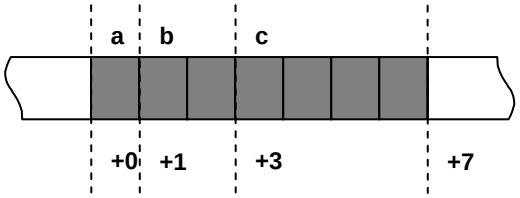


图 12.6 字节对齐模式



12.3 学生用书及幻灯片注解

12.4 常见课堂问题 FAQ

12.5 参考资料

12.6 背景知识

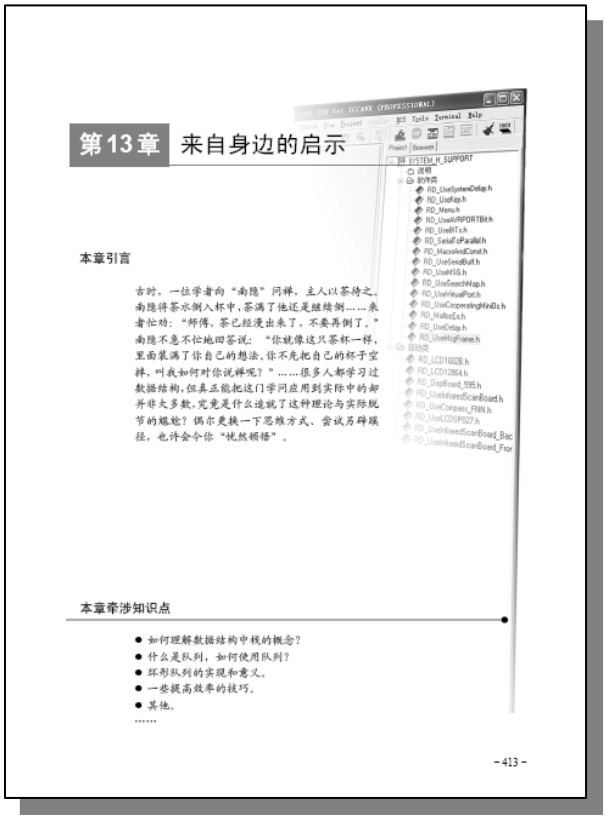
12.6.1 微机原理

本章内容牵涉到微机原理中关于存储器对齐方式以及数据组织方式的相关内容。

12.6.1 计算机组成原理

本章内容牵涉到计算机组成原理关于内核寻址、译码机制的相关内容。

12.7 参考设计与实验方法


第13章 来自身边的启示

本章引言

古时，一位学者向“南隐”问禅，主人以茶待之。南隐将茶水倒入杯中，茶满了他还是继续倒……来者忙劝：“师傅，茶已经溢出来了，不要再倒了。”南隐不急不忙地回答说：“你就像这只茶杯一样，里面装满了你自己的想法。你不先把自己的杯子空掉，叫我如何对你说禅呢？”……很多人都学习过数据结构，但真正能把这门学问应用到实际中的却并非大多数。究竟是什么造就了这种理论与实际脱节的尴尬？偶尔更换一下思维方式、尝试另辟蹊径，也许会令你“视然顿悟”。

本章牵涉知识点

- 如何理解数据结构中栈的概念？
- 什么是队列，如何使用队列？
- 环形队列的实现和意义。
- 一些提高效率的技巧。
- 其他。

……

- 413 -

第十三章　数据结构

Rev 1.0.0

13.1 内容简介

- 抽象数据类型简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

13.2 抽象数据类型简介

抽象数据类型是数据结构的一种应用形式。与普通数据类型相似，抽象数据类型由类型描述（例如取值范围）和操作集合共同定义。例如，普通的整型数据类型由整数的取值范围和整数能参与的一系列加减乘除运算共同定义：**unsinged char** 的取值范围是 **0~255**，其操作集合为“**+**、**-**、*****、**/**、**&**、**|**、**~**……”。抽象数据类型是由一类数据结构及相关操作函数构成。例如，我们可以将存储和表达环形队列的数据缓冲以及访问指针封装成结构体类型，此时该结构体类型与相关的队列操作函数就构成了环形队列的抽象数据类型。

值得一提的是，如果我们将队列初始化、释放和访问的函数以函数指针的形式存储在抽象数据类型的结构体中，则完成了面向对象的封装。此时的抽象数据类型就可以被称之为类**(Class)**；具体的结构体变量就可以被称之为对象的实例（**Instance**）；结构体中所有的函数指针构成了虚函数表**(Virtual Function Table)**；队列的初始化函数和释放函数可以分别被称之为构造**(Constructor)**/析构函数**(Destructor)**；其它队列的操作函数称之为方法（**Method**）。结构体中允许外部直接访问的变量称之为成员变量**(Member)**或属性**(Property)**，其公有或是私有属性由类型的掩码结构体决定：掩码结构体中公开的成员变量具有公有**(Public)**属性；被屏蔽的部分具有私有**(Private)**属性。

13.2.1 名词解释

- **抽象数据类型 (Abstract Data Type)**

一类数据结构，其实现形式和内部操作过程对用户是不透明的，我们称这类数据结构及其操作为抽象数据类型。

- **面向对象编程 (Object-Oriented Programming)**

将数据以及对其进行操作的函数封装在一起构成抽象数据类型的编程方式，称为面向对象编程。

- **面向对象 C 语言开发 (Object-Oriented Programming with ANSI-C)**

程序员使用 **C** 语言手工构造抽象数据类型实现面向对象的开发，称为面向对象 **C** 语言开发。**C** 语言本身是面向过程的**(Procedure-Oriented)**，并不能很好的支持面向对象的开发。**C++**是面向对象的编程语言**(Object-Oriented Language)**，编译器本身会根据用户定义的类去自动构造抽象数据类型。

- **掩码结构体 (Masked Structure)**

在定义抽象数据类型时，为了向外界屏蔽结构体内的某些信息而定义的与实际抽象数据类型占用相同存储器空间、仅公开某些成员而屏蔽大部分信息的结构体类型，称之为掩码结构体。这种结构常用于使用 **C** 语言实现的类封装结构中。

13.2.2 抽象数据类型构成要素

使用 **C** 语言构建抽象数据类型主要借助以下手段：

i) 使用结构体来封装数据与成员变量；

ii) 使用 **typedef** 来定义抽象数据类型；

iii) 如果需要将抽象数据类型封装成对象，还需要借助函数指针构成虚函数表；

iv) 如果需要在不同的对象操作中使用不同的参数，还需要借助可变参数列表的支持。

本节内容着重介绍函数指针和可变参数列表的使用方法。

13.2.2.1 函数指针

指向函数入口地址的指针称为函数指针。与普通指针类似，函数指针的本质也是一个整型变量；定义函数指针时需要详细的制定其指向函数的参数表和返回值类型。下面的代码就定义了一个指向函数 **void Example(unsigned char *pstrData)**的函指针：

```
/* 测试函数 */
void ExampleA(unsigned char *pstrData)      { };
void ExampleB(unsigned char *pstrData)      { };
……
/* 声明一个函数指针 fnPointToFunction 并对其进行初始化 */
void (* fnPointToFunction)(unsigned char *pstrData) = &ExampleA;
……
void main(void)
{
    ……
    /* 通过函数指针调用函数 */
    (*fnPointToFunction)(“Hello world”);
    /* 直接调用函数 */
    Example(“Hello world”);
    /* 更改函数指针的值 */
    fnPointToFunction = ExampleB;      /* “&”运算符不是必须的，可以省略*/
                                       /* 函数的名称就代表它的入口地址 */
    /* 使用函数指针调用 ExampleB */
    fnPointToFunction(“Bye!”);         /* “*“运算符也不是必须的，可以省略 */
    ……
}
```

函数指针可以像普通指针那样作为函数的参数或返回值，也可以用来构建结构体和数组。下面的代码声明了一个使用函数指针作为参数和返回值的函数，其中，函数名为 **Example**，下划线指示的部分为函数名以及参数列表，其余部分为函数返回值类型说明：

```
/* 一个使用函数指针作为参数和返回值的函数例子 */
void (*Example (void (* fnParameter)(unsigned char *pchData)))(unsigned char *pchData);
```

这样的函数声明可读性非常的糟糕，因此我们使用 **typedef** 来逃离这一窘境：

```
/* 使用 typedef 定义函数指针 */
typedef    void (* P_FUN)(unsigned char *pchData);    /* 定义了一个函数指针类型 P_FUN */
```

```
/* 测试函数 */
void ExampleA(unsigned char *pstrData)      { };   /*假设这个函数在 LCD 上输出字符 */
void ExampleB(unsigned char *pstrData)      { };   /*假设这个函数在打印机上输出字符 */

/* 使用新定义的函数指针类型声明函数指针和数组 */
P_FUN      fnFunctionToPoint;                          /* 声明了一个函数指针 */
P_FUN      fnFunctionTable[] =                         /* 声明了一个函数指针数组 */
                        {
                              &ExampleA, &ExampleB      /* 初始化数组 */
                        };

/* 使用新定义的类型声明了一个和前面代码片断相同的函数:
   使用函数指针作为输入参数和返回值 */
P_FUN      Example(P_FUN fnParameter)
{
      ......
}

void main(void)
{
      /* 使用范例 */
      unsigned char n = 0;
      /* 在所有有效的输出设备上都输出 Hello world */
      for (n = 0;n < (sizeof(fnFuntionTable) / sizeof(fnFunctionTable[0])); n++)
      {
            /* 依次遍历函数数组 */
            fnFunctionTable[n](“Hello world!”);
      }
}
```

13.2.2.2 可变参数列表

可变参数是指某一函数被调用时，并不知道具体传递进来的参数类型和参数的数目，例如大家熟知的函数 **printf()**：

```
/* 一个使用可变参数的例子 */
printf(“Hello! \n”);                                    /* 只有一个字符串参数 */
printf(“Today is the %dth day of this week. \n”,Week);  /* 有两个参数 */
printf(“ %d + %d = %d”,a,b,a * b);                      /* 三个参数 */
```

C 语言是通过软件堆栈的方式进行参数传递的，对于下面的函数，从左至右依次压入栈中的变量为：**a**、**b**、**c**。如果存在更多的参数，只要在函数在真正被调用前按照同样的顺序依次压入栈中就可以完成任意数量参数的传递。这就是可变参数传递的原理，在函数声

明时，在参数列表最右边加入一个省略号“...”作为参数就可以将一个函数声明为可变参数传递。例如：

```
/* 一个使用可变参数的例子 */
void printf(char *pString,…);              /* 使用可变参数作为函数参数 */
```

可变参数实际上是具有参数类型 **va_list**。在函数内部必须要首先声明一个可变参数变量，以便依次取出所有传入的数据，例如：

```
va_list     Example;                       /* 定义一个可变参数列表 */
```

va_list 可以像普通变量类型一样充当函数的参数和返回值，例如：

```
/* 定义一个函数，需要上级函数传递一个 va_list 型变量的指针 */
void FunctionExample(va_list *pva);
```

我们可以通过宏 **va_start()**告知函数准备从堆栈中取数据。其中，使用 **va_start()**需要传递两个参数，分别为 **va_list** 变量及函数参数列表中“**…**”左边的第一个形参的名称。例如：

```
va_start(Example,pString);          /* 告知函数准备开始从可变参数列表 Example 中取数据 */
```

与 **va_start()**对应，我们可以通过宏 **va_end()**告知函数不再继续进行参数的提取。例如：

```
va_end(Example);                    /* 结束参数提取 */
```

在 **va_start()**和 **va_end()**所划定的范围内，我们可以通过 **va_arg()**依次提取所需的参数，其中提取参数的顺序与调用函数时传送参数的顺序相同。例如：

```
unsigned int A = va_arg(Example,unsigned int);   /* 提取一个 unsigned int 型的数据 */
```

也可以通过 **va_copy** 为当前的参数列表做一个备份（备份当前的参数读取位置），例如：

```
/* 保存当前的参数栈 */
va_list ExampleB;                   /* 定一个新的可变参数列表 */
va_copy(ExampleB,Example);          /* 复制当前的参数栈信息到 ExampleB 中 */
```

13.2.2.3 综合演示

该范例用于实现向指定的设备输出可变数量的字符串。我们首先需要利用函数指针构造一个输出设备驱动函数表，将所有的输出设备已数组的形式组织在一起：

```
/* 定义输出设备驱动函数的原形 */
typedef void OUTPUT_DRV(unsigned char *pstr,va_list *pArg);
                                                    /* 注意这里不是定义函数指针 */
                                                    /* 而是定义了一个函数原形   */
OUTPUT_DRV  LCD_Drv;                                /* 定义了一个函数 LCD_Drv() */
OUTPUT_DRV  PRN_Drv;                                /* 定义了一个函数 PRN_Drv() */

/* 定义指向 OUTPUT_DRV 类型函数的函数指针 */
typedef OUTPUT_DRV *        P_DRV;                  /* 这里定义了一个函数指针 */


/* 使用函数指针构造了一个驱动函数表 */
P_DRV     OutputDrivers[ ] = {
```

```
        &LCD_Drv, & PRN_Drv
    };
```

接下来我们将通过可变参数实现一个向指定设备输出类似 **printf** 格式字符串的函数，具体的设备需要用户通过字符串的形式给出，例如“**LCD**”或者“**PRN**”。该函数将根据用户输入的字符串决定输出的设备和字符串：

```
#include <stdarg.h>
#include <string.h>

/* 定义输出设备驱动函数的原形 */
int Print(unsigned char *DrvNAME,...)
{
    unsigned char *pstr = NULL;
    P_DRV       fnDrv = NULL;
    va_list     Arg;                        /* 定义可变参数列表 */

    /* 健壮性检测 */
    if (DrvNAME == NULL)
    {
        return -1;
    }

    /* 确定使用哪个设备进行输出 */
    if (strcmp(DrvNAME,"LCD") == 0)         /* 如果输入的第一个字符串为 LCD */
    {
        fnDrv = OutputDrivers [0];          /* 使用 LCD 驱动 */
    }
    else if (strcmp(DrvNAME,"PRN") == 0)    /* 如果输入的第一个字符串为 PRN */
    {
        fnDrv = OutputDrivers [1];          /* 使用打印机驱动 */
    }
    else                                    /* 未知的设备 */
    {
        return -1;
    }
    va_start(Arg, DrvNAME);                 /* 开始取参数 */
    pstr = va_arg(Arg,unsigned char *);     /* 获取一个字符串 */
    fnDrv(pstr,&Arg);                       /* 调用指定的设备驱动 */
    va_end(Arg);                            /* 结束取参数 */
}
```

```
/* 驱动函数实体 */
void LCD_Drv(unsigned char *pstr,va_list *pArg)
{
    ……
    /* 在函数中可以通过 va_arg(*pArg,类型) 来依次提取参数，不需要
       通过 va_end(*pArg)来标注取参数结束，如果通过 va_copy 生成了
       一个新的 va_list 变量，则需要在取出参数后通过 va_end()将该变
       量关闭。*/
    ……
}

void PRN_Drv(unsigned char *pstr,va_list *pArg)
{
    ……
}
```

可以使用下面的方式调用函数 **Print()**：

```
/* Print() 的操作范例 */
unsigned char Day = 3;
Print("LCD", "It`s the %dth day of this week.\n",Day);
```

如果 **LCD** 驱动编写无误，我们将在 **LCD** 设备上看到以下的内容：

```
It`s the 3th day of this week.
```

13.2.3 构建抽象数据类型

抽象数据类型就是对数据结构和相关操作的封装。以前面介绍的字符串输出系统为例，通过改造和对比，你可以很明显找到抽象数据类型和普通数据结构的区别。首先，我们需要分析所需抽象的对象在数据结构上有那些共同的特质，或者说我们如何将对象的共性和特型区别开来。

以字符串输出设备为例，**LCD**、打印机、显示器、超级终端……这些设备都可以通过数据流的方式进行输出，都在一定程度上兼容流控制，因此可以使用类似的函数接口形式，以“**print** 打印”格式字符串的形式进行输出，基于以上条件我们定义如下的驱动接口函数原型（**Prototype**）：

```
/* 定义驱动函数原型 OUTPUT_DRV */
typedef   int OUTPUT_DRV(unsigned char *pstr,va_list *pArg);
/* 定义指向 OUTPUT_DRV 的函数指针 */
typedef OUTPUT_DRV *              P_DRV;
```

接下来，我们可以为不同的驱动指定一个唯一的 **ID**，并将这些信息同驱动接口函数（指针）封装在一起，我们称之为一个输出设备 **(Output Device)**：

```
# include <stdint.h>

/* 定义输出设备(Output Device) */
typedef struct
{
    uint16_t    ID;                 /* 设备 ID */
    P_DRV       fnDriver;           /* 驱动函数指针 */
}OUTPUT_DEV;
```

这就初步完成了对抽象数据类型的封装。**OUTPUT_DEV** 是一个自定义类型，包含了设备 **ID** 和设备的驱动函数指针——这是一个典型的抽象数据类型。当用户使用我们下面将要定义的通用接口函数来操作设备时，所有的具体细节都是透明的，这就是抽象（**Abstract**）一词的精髓所在：

```
# include <stdint.h>

typedef uint16_t      ERR_CODE;
# define ERROR_NONE_ERROR              0x0000
# define ERROR_ILLEGAL_PARAMETER       0xFFFF

/* 设备通用接口函数 */
ERR_CODE Print_Device(OUTPUT_DEV *pDev,unsigned char *pStr,...)
{
    ERR_CODE      Err;
    va_list       Arg;

    if ((pDev == NULL) || (pStr == NULL))      /* 强壮性检测 */
    {
        return ERROR_ILLEGAL_PARAMETER;
    }

    va_start(Arg,pStr);
    Err = (pDev -> fnDriver)(pStr,&Arg);
    va_end(Arg);

    return Err;                                /* 无错误产生 */
}
```

如果我们将上面的代码存在在一个叫 **OutputDevice.c** 的文件中，将前面的定义部分（前一个代码片断）连同设备通用接口函数的声明一起存放到 **OutputDevice.h** 中，我们事实上就完成了一个抽象数据类型的函数库封装，这也是利用抽象数据类型完成系统模块化的一个典型范例。当然，其中还包含一些函数封装的细节问题，比如包含、接口描述、头文件保护等等。详细内容您可以参考本书的相关章节。

13.2.4 面向对象

在 **13.3.2** 结中，我们通过一个简单的例子说明了如何构建和使用抽象数据类型，也许例子中的驱动结构已经相当实用了，但是这种程度的抽象与面向对象技术相比仍然显得不够“强大”。在 **C** 语言中应用面向对象技术并不仅仅是利用结构体将数据和函数封装在一起那么简单，完成了抽象数据类型的定义只是万里长征的第一步。如何充分利用抽象数据类型带来的优势，是这一小结将要讨论的内容。

在前面的 **13.2.2.3** 章节中，我们演示了一种通过字符串指定设备，并完成输出的功能。示例代码只给出了针对两种设备的操作，用户如果想添加新的设备，必须修改函数 **Print**。显然 **Print** 是封装在 **OutputDevice.c** 中的一个函数，不应该要求用户对其进行修改，为了实现扩展，需要借助两个面向对象的概念：继承（**Inherite**）和派生（**Derive**）。

在给出示例代码之前，我们应该强调一点：以下的示例也许看起来显得笨拙，但是它演示了在 **C** 语言中实现面向对象技术的一些手法，毕竟使用 **C** 语言实现的面向对象都是近似的，或者说类和对象的概念仅仅存在于开发人员的大脑中。从这一点出发，你就会明白，很多时候不应该执着于让程序看起来“像”面向对象，而是应该让程序运行起来与面向对象“效果相同”。这也是使用 **C** 语言实现面向对象技术的精髓。

首先，我们定义一个新的抽象数据类型，用以将所有的设备组织起来，并能通过一个唯一的名称来识别他们（很显然，用户不应该被要求知晓设备内部 **ID** 编号）：

```
#ifndef _OUTPUT_DEVICE_H_
#define _OUTPUT_DEVICE_H_
……
typedef uint16_t       ERR_CODE;

# define ERROR_NONE_ERROR                0x0000
# define ERROR_DEVICE_NOT_FOUND          0x0001
# define ERROR_ILLEGAL_PARAMETER         0xFFFF
……

/* 新的数据类型 */
typedef struct   Output_Dev_Item          OUTPUT_DEV_ITEM;

struct   Output_Dev_Item
{
    OUTPUT_DEV              Device;       /* 继承了 OUTPUT_DEV */
    unsigned char           DevName[4];   /* 增加了一个 3 个字符长度的名称 */
    OUTPUT_DEV_ITEM *       pNext;        /* 构成链表 */
};

/* 我们需要为新派生的类添加三个成员函数 */
```

```
/*  向设备链上增加一个新设备，并为其指定一个名称*/
extern OUTPUT_DEV_ITEM *Add_Device
        (
            OUTPUT_DEV *pDevice,
            unsigned char chDevName[4]
        );

/*  从设备链上删除一个指定名称的设备  */
extern OUTPUT_DEV_ITEM *Remove_Device(unsigned char chDevName[4]);

/*  向指定的设备输出格式字符串  */
extern ERR_CODE Print(unsigned char chDevName[4],unsigned char *pStr,…);

……
#endif
```

经过一次对原有抽象数据类型 **OUTPUT_DEV** 的继承，我们至少在 **OutputDevice.h** 中派生出了一个看起来非常强大的新“类”**OUTPUT_DEV_ITEM**，所有针对其进行的操作对用户都是透明的：添加一个设备、删除一个设备甚至直呼其名的在设备上输出格式字符串。这一切都是通过在继承老对象的同时，增加了新的成员变量 **Name** 和用于构建链表的指针 **pNext** 实现的。具体的函数，我们不再一一给出，仅就核心的操作函数 **Print** 给出其代码：

```
/* OutputDevice.c */
#include <stdarg.h>
#include <stdint.h>
#include <string.h>
#include “OutputDevice.h”
……
static OUTPUT_DEV_ITEM *Find_Device(unsigned char chDevName[4]);
……
static OUTPUT_DEV_ITEM *s_pChainRoot = NULL;
……
ERR_CODE Print(unsigned char chDevName[4],unsigned char *pStr,…)
{
    va_list Arg;
    ERR_CODE Err;
    OUTPUT_DEV_ITEM *pDeviceItem = NULL;

    /*  系统强壮性检测  */
    if ((pStr == NULL) || (chDevName == NULL))
    {
        return ERROR_ILLEGAL_PARAMETER;
    }
```

```
    pDeviceItem = Find_Device(chDevName);      /* 查找指定的设备 */
    if (pDeviceItem == NULL)
    {
        return ERROR_DEVICE_NOT_FOUND;     /* 设备没有找到 */
    }

    /* 提取可变参数列表 */
    va_start(Arg, chDevName[4]);
    Err = (pDeviceItem -> Device). fnDriver(pStr,&Arg);
    va_end(Arg);

    return Err;                                    /* 返回操作结果 */
}
```

值得一提的是 **OutputDevice.c** 中使用到了一个静态函数 **Find_Device()**，由于函数的静态特性与私有成员很类似，我们可以将该其视作私有**(Private)**成员函数，而其它在模块外部通过头文件 **OutputDevice.h** 可以找到接口并调用的函数则称为公有**(Public)**成员函数。同理，全局静态变量 **s_pChainRoot** 可以被视作私有成员变量。**Add_Device()**和 **Remove_Device()**分别充当了构造函数和析构函数的角色，只不过这些函数都不是初始化时由系统自动执行的。抽象数据类型 **OUTPUT_DEV_ITEM** 接口函数 **Add_Device()**、**Remove_Device()**和 **Print()**完成了对整个“类”的封装。更为完善和复杂的 **C** 语言面向对象技术，可以通过查阅文献《**Object-Oriented Programming with ANSI-C**》获得。

```
/* 使用范例 */
#include “OutputDevice.h”

# define DEVICE_ID_LCD                  0x0001
/* 声明一个 LCD 驱动函数 */
OUTPUT_DRV    LCDDrv;
/* 声明一个 LCD 设备 */
OUTPUT_DEV       LCDDev =
                    {
                        DEVICE_ID_LCD, &LCDDrv;
                    };

void main(void)
{
    uint8_t Day = 3;
    Add_Device(&LCDDev,”LCD”);          /* 添加一个 LCD 设备，名为“LCD”*/
    Print(“LCD”, “It`s the %dth day of this week.\n”,Day);
}
```

13.3 学生用书及幻灯片注解

13.4 常见课堂问题 FAQ

13.5 参考资料

13.5.1 参考文献

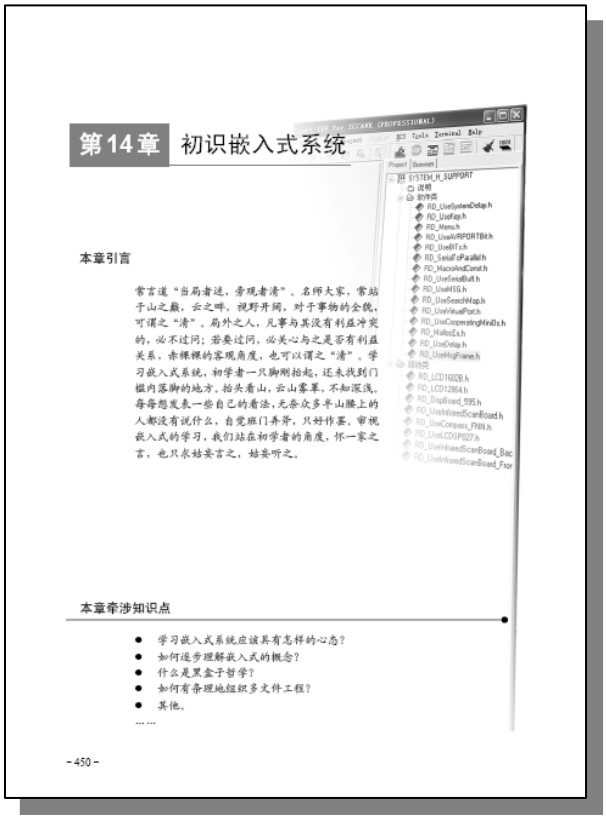
- **ISO/IEC 9899:1999 (E)**
- **《Object-Oriented Programming with ANSI-C》 Axel-Tobias Schreiner 1993.10**
- **《UML + OOPC 嵌入式 C 语言开发精讲》高焕堂著 电子工业出版社 2008 年 9 月第一版**

13.6 背景知识

13.6.1 数据结构

本章内容牵涉到数据结构的相关内容。

13.7 参考设计与实验方法


第14章 初识嵌入式系统

本章引言

常言道“当局者迷，旁观者清”，名师大家，常站于山之巅，云之畔，视野开阔，对于事物的全貌，可谓之“清”，局外之人，凡事与其没有利益冲突的，必不过问；若要过问，必关心与之是否有利益关系，赤裸裸的客观角度，也可以谓之“清”。学习嵌入式系统，初学者一只脚刚抬起，还未找到门槛内落脚的地方，抬头看山，云山雾罩，不知深浅。每每想发表一些自己的看法，无奈众多平山腰上的人都没有说什么，自觉班门弄斧，只好作罢。审视嵌入式的学习，我们站在初学者的角度，怀一家之言，也只求姑妄言之，姑妄听之。

本章牵涉知识点

- 学习嵌入式系统应该具有怎样的心态？
- 如何逐步理解嵌入式的概念？
- 什么是黑盒子哲学？
- 如何有条理地组织多文件工程？
- 其他。

……

- 450 -

第十四章　嵌入式软件构架

Rev 1.0.0.0

14.1 内容简介

- 嵌入式软件构架简介
- 学生用书及幻灯片注解
- 常见课堂问题 FAQ
- 参考资料
- 背景知识
- 参考设计与实验方案

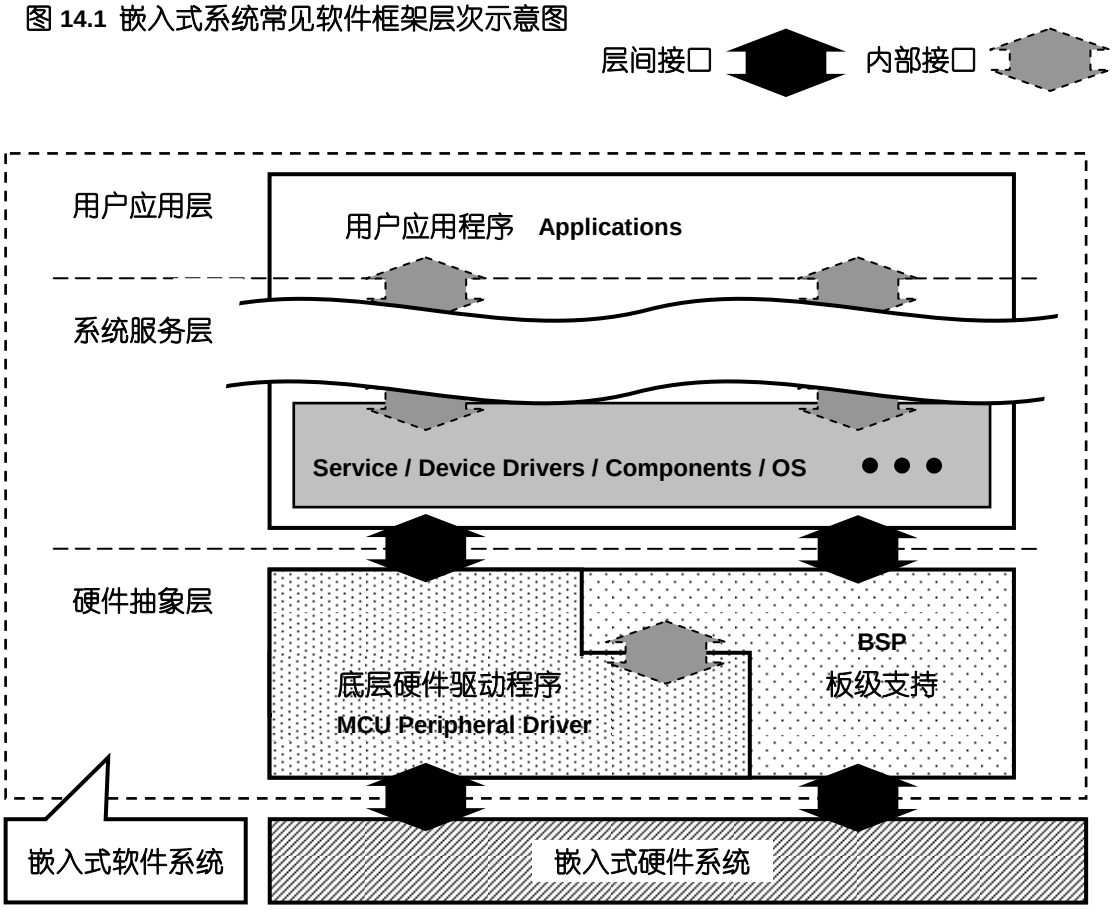
14.2 嵌入式软件构架简介

嵌入式系统是一种专用计算机系统，面向具体的应用开发硬件系统和软件系统。与通用计算机系统不同，嵌入式系统的硬件设计直接面向具体的需求，仅保留实现应用所需的最基本硬件配置和输入输出接口，在对成本敏感的项目开发中（比如消费类电子产品的设计），通常会进行最大限度地系统裁减，有时仅会保留有限的扩展能力。

嵌入式硬件系统设计直接限定了软件系统的开发：专用硬件系统的特殊性决定了嵌入式软件系统必须依照具体的硬件配置进行开发。不同嵌入式系统的软件无法像通用计算机系统那样相互兼容，移植几乎是必须的。为了降低开发周期，嵌入式软件开发团队通常对程序的可移植性、可裁减性、高效性、可靠性和可读性有着严格的要求，具体表现为文档的编写、编码规范和接口规范的制定等。

图 14.1 展示的是一个常见嵌入式系统的框架结构，忽略了软件内部的具体细节，但保留了基本的层次框架。经过多年发展，嵌入式微处理器的内核、总线和外设已经有了相当的规范性。与之对应，嵌入式软件也形成了基本的硬件驱动接口标准，并拥有了一个约定俗成的基本框架构思：即通过硬件抽象的方法为上层硬件无关的软件层次提供操作硬件设备的接口和 **API** 函数；提供驱动集成电路板上特殊电路模块的板级支持包（**BSP**）。习惯上，底层硬件驱动程序与板级支持包一起被称为硬件抽象层（**HAL,Hardware Abstract Layer**），其上是由与硬件无关的各类服务、驱动、组件等组成的系统服务层（**SSL,System Serve Layer**）；其下是具体的嵌入式硬件系统，包括嵌入式微处理器和周边外设电路。

图 14.1 嵌入式系统常见软件框架层次示意图



硬件抽象层的建立极大地提高了嵌入式软件的可移植性，然而由于很多历史原因，在数据的存储顺序上，嵌入式微处理器存在大端对齐和小端对其两个阵营；使用 **C** 语言编写程序时，不同字长的微处理器对 **int** 型变量和 **long** 型变量存在不同的解释；有的嵌入式微处理器没有 **CPU** 高速缓冲存储器，而有的微处理器又有对高速缓冲隐式管理和显式管理的区别；不同 **C** 语言编译器对 **ANSI-C** 标准有着不同的理解和实现方式……所有这些都是可能导致嵌入式软件移植失败的因素。

一个良好的嵌入式软件构架应该综合考虑硬件抽象、编译环境、可移植范围和可裁减程度、可读性、维护周期和成本等诸多因素。作为引导性内容，本章只就与嵌入式软件构架相关的一些概念进行简单的讨论和介绍，具体细节请参考相关的技术书籍和文献。

14.2.1 名词解释

- **黑盒子**

嵌入式系统开发中，习惯上把只需要关注实现功能的接口和方法，而具体内部结构对用户不可见或者可以忽略的模块称之为黑盒子。黑盒子既可以是软件模块也可以是硬件模块。

- **硬件抽象**

为具体的硬件编写驱动程序使其具有符合某一规范的抽象接口，并为上层软件提供一整套接口操作 **API** 函数的开发过程称为硬件抽象。通过抽象的硬件可以视作黑盒子。

- **预编译**

在程序源文件正式编译前由编译器完成的准备性工作称为预编译。**C** 语言环境中，这些准备工作包括完成源程序文件对其它文件的包含、宏替换、根据条件对参与实际编译的代码进行选择等等。

- **C 语言头文件(.h)**

C 语言中以“**.h**”为扩展名，在预编译阶段被其它程序源文件通过预编译语句 **include** 包含的文本文件称为头文件。为了防止头文件被同一个源程序重复包含，一般会在头文件中加入条件编译语句进行保护。**C** 语言头文件不会以独立文件的形式参与到实际的编译过程中。

- **C 语言源文件(.c)**

C 语言中以“**.c**”为扩展名，在编译阶段将被编译器逐个编译并生成同名对象文件的文本文件称为源文件。**C** 语言源文件是程序编译的基本单位，通常包含了一个或若干头文件。

- **十六进制对象文件(.hex)**

十六进制对象文件是一种将二进制对象用可显示的 **ASCII** 码进行表示的文本文件格式。通过 **ASCII** 码，二进制对象可以存储在非二进制的存储介质中，比如纸带、穿孔卡片等等，可以在显示器上显示也可以通过打印后阅读。

● **编译(Compile)**

根据程序源文件和资源文件生成对象文件并链接为目标机可执行代码的过程称为编译。

● **目标文件(.obj)**

高级语言源程序或者汇编语言源程序经过编译后生成的中间文件。通常为目标机的二进制代码片断，经过链接后生成可执行的二进制代码或存储器镜像。

● **链接(Link)**

将目标文件整合在一起生成目标机可执行二进制代码或者存储器镜像的过程称为链接。

● **单文件编译**

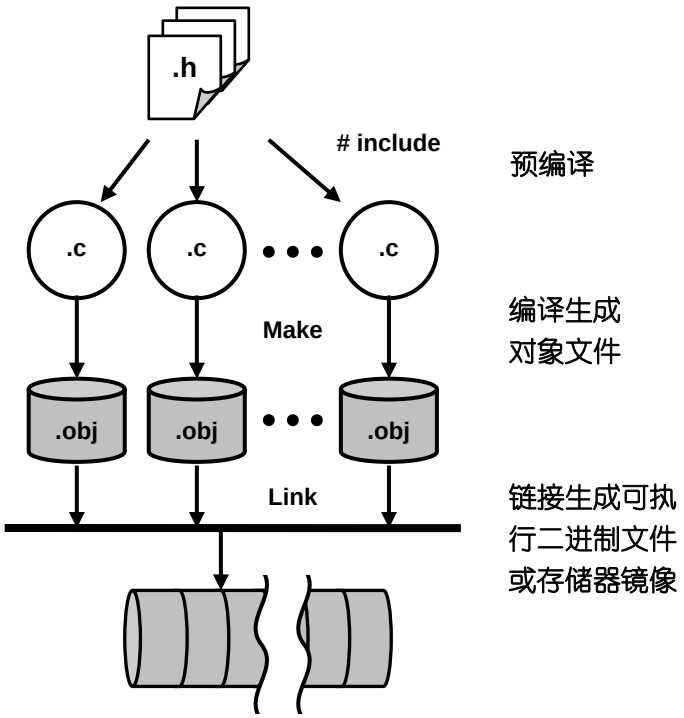
在一个工程中，只针对单个程序源文件进行编译，直接生成目标机可执行代码或存储器镜像的过程称为单文件编译。

● **工程编译**

在一个工程中，对所有的程序源文件进行依次编译，生成同名对象文件，并在链接阶段将这些对象文件合并为目标机可执行二进制代码或者存储器镜像的过程称为工程编译。

14.2.2 多文件编译流程

多文件编译又称工程编译，其大体流程如所图 **14.2** 示，分为预编译、对象文件生成和链接三个阶段：



- 预编译（**Preproccess**）。整个编译过程都是基于“**.c**”文件进行的。在预编译阶段，编译器会独立的处理每一个程序源文件，包括将**#include** 所包含的内容添加到程序源文件中、进行宏替换、根据条件编译选项进行代码裁减等等。预编译阶段的产物是一个用户不可见的中间程序源文件，该文件包含了所有实际参加对象生成的源代码。头文件和预编译宏已经替换为实体内容存在于中间代码中，不会进入下一阶段。

- 对象文件生成（**Make**）。对象文件生成阶段，编译器会根据预编译阶段产生的中间源代码进行编译，生成二进制对象文件。该阶段仍然是以源代码模块为单位进行独立编译，如果模块代码中出现了对其它模块中函数或者变量的引用，系统会简单地使用引用标志进行标记和替换，并不会对所引用函数或变量的有效性进行检测。该阶段的产物是与程序源文件同名的对象文件，一般以“**.o**”或“**.obj**”作为文件扩展名。如果单个源程序中存在语法错误，会在该阶段产生警告或错误报警。通常这些警告和错误可以直接定位到程序源文件的具体位置。

- 链接（**Link**）。链接阶段，编译器会搜集所有的对象文件或工程引用的库文件（一般以小写字母“**lib**”为主文件名前缀、以“**.a**”为扩展名，例如“**libm.a**”），并尝试将这些对象文件合并为可执行二进制文件或存储器镜像。在该阶段，系统会检测每个对象文件对其它模块函数和变量引用的有效性，例如，检测函数和变量实体是否存在，类型是否匹配等。链接生成的内容可以是一个库函数（前面说的“**.a**”文件），也可以是一个可执行程序，二者的区别在于工程中是否存在 **main** 函数。如果在链接阶段发生了函数或变量引用等错误，系统会发出警告或错误报警，这些信息将无法定位到程序源文件中具体的位置。

14.2.3 头文件包含和路径设定

在一个工程中可能存在多个文件夹，源程序文件所包含的头文件可能存在于不同的目录中，对于这种情况 **ANSI-C** 规定了以下的应对策略：

a、 如果明确所要包含的头文件就存放在与源程序文件相同的目录下，则直接在**#inlude** 语句中使用双引号包含头文件名，例如

```
# include “app_cfg.h”
```

b、 如果所要包含的头文件存放在工程设置所指定的路径下，比如编译器安装文件夹的“**include\avr**”目录以及用户指定的目录，则直接在**#include** 语句中使用尖括号包含头文件名，例如：

```
# include <avr/io.h>
```

对于引号所包含的头文件，系统会首先在源文件当前目录下查找；如果文件并不存在，系统会从用户指定的路径开始，依次确认文件是否存在。

14.2.4 头文件多次包含的保护

在多文件工程中，同一个头文件可能会通过各种不同的途经被同一个源程序文件多次包含，导致编译时刻发生“重复定义”的错误。为了预防这种情况的发生，每一个头文件

都应该使用预编译宏进行保护，例如对于配置头文件 **app_cfg.h**，一个典型的保护结构如下：

```
/*It`s the begin of Application Configuration Head file */
#ifndef     __APP_CFG_H_
#define     __APP_CFG_H_
……
/* 添加 app_cfg.h 的实体部分 */
……
/*It`s the end of Application Configuration Head file */
#endif
```

14.2.4 多文件包含的模块封装

在一个工程中，如果模块和层次划分已经明确，那么建议按照以下原则来处理模块的多文件包含问题：

- 在模块内部，建立一个总的头文件用于包含模块内所有其它头文件。所有模块内的程序源文件都需要包含该头文件。可以命名为“**xxx_includes.h**”
- 为每个模块建立一个只包含配置信息的头文件，一般命名为“**xxx_cfg.h**”。该配置头文件用于存放所有的模块配置信息，原则上在配置头文件中不再包含其它“非配置用途”的头文件。该头文件一般直接被模块内程序源文件所包含，或者直接被模块的“**xxx_includes.h**”包含。
- 为每个模块建立一个以模块命名的接口头文件（例如, 文件系统的接口头文件 **fs.h**），其中包含所有对模块外部开放的函数接口、宏定义、变量声明、结构体和类型定义。该头文件用于向外界提供操作模块功能的接口，一般直接被模块内程序源文件所包含，或者直接被模块的“**xxx_includes.h**”包含。
- 如果模块内源文件要用到别的模块接口，直接包含对应模块的接口头文件即可，并不需要将该接口头文件包含在本模块的“**xxx_includes.h**”、“**xxx_cfg.h**”或者接口头文件中。

如果若干个工程组成了一个层次，或者某一个大模块由若干子模块组成，那么建议按照以下的原则来组织文件结构：

- 为整个层次或大模块建立配置文件，命名原则与模块内的配置头文件相同，也为“**xxx_cfg.h**”。其中添加对整个模块或层次的配置信息。将该配置文件加入到所有子模块配置文件的首部。
- 建立一个配置和初始化用的源程序文件，在其中添加一个初始化函数，调用所有子模块的初始化函数。
- 建立一个以层次或模块命名的接口头文件，将所有子模块的接口头文件添加在其中并添加将刚才编写的层次（或模块）初始化函数接口。不能将层次（或大模块）的接口头文件等同于“**xxx_includes.h**”。当子模块中的源程序文件试图访问同一层次下其它子模块的接口，不能通过包含层次（或大模块）接口头文件的形式来实现，这种方式称为“向上引用”，违反了模块封装原则，屏蔽了模块间的依赖关系，将给模块的移植带来不便。

对于一个包含了若干模块（和层次）的工程来说，建议按照以下的原则来组织文件结构：

- 建立一个工程配置文件，一般命名为“**app_cfg.h**”。头文件中添加对整个工程的配置信息。如果工程存在若干个不同的配置，则分别建立对应的配置文件以“**xxx_app_cfg.h**”命名，并在“**app_cfg.h**”中以条件编译的方法进行选择性包含。例如：

```
#ifndef     __APP_CFG_H_
#define     __APP_CFG_H_
……
#if defined(条件 1)
    # include “example1”
#elif define(条件 2)
    # include “example2_app_cfg.h”
……
#else
    # error Need for application configuration file
#endif
……
#endif
```

  将“**app_cfg.h**”加入到所有模块和层次配置文件的首部。其中，用于选择具体配置文件的宏一般通过工程配置属性进行定义。
- 建立一个源程序文件，包含所有模块的接口头文件，并在该源程序中编写函数，完成对所有模块和层次的初始化。该函数应该在主程序中被调用。
- 可以建立一个全局包含头文件，用于包含工程中所有最顶层模块或层次的接口头文件，命名为“**includes.h**”，但一般情况下不建议这么做。因为用户应用程序一旦通过“**includes.h**”来引用所有的模块，就会屏蔽应用程序与具体模块的依赖关系信息，为用户应用程序的移植和裁剪带来不便。用户应用程序对模块的引用应该遵循“按需分配”的原则。

通过上面的封装策略，模块与模块之间、层次与层次之间调用和依赖关系清晰；模块仅对外部提供接口文件而模块内部结构对外界是透明的，可以视为黑盒子。源程序想要使用某一模块的功能时，无须知晓模块的内部接口细节，直接包含模块的接口头文件即可，降低了程序阅读的复杂程序。模块的独立性的提高将带来代码可移植性和复用性的提高。

14.2.5 编译器无关性

在多文件工程中，我们通常将与编译器以及处理器字长相关的部分独立出来，采用统一的规范进行重新定义，例如在 **8** 位和 **16** 位系统中使用 **uint16_t** 代替 **unsigned int**，在 **32** 位系统中使用 **uint16_t** 代替 **unsigned short**。这些信息一般都放置在名为“**compiler.h**”的头文件中。编译器无关性的内容还包括，大小端数据的转换、位段对其方式的转换，在线汇编的实现方式、中断向量的处理等等。具体信息请参考相关文档。

14.2.6 工程的纵向层次划分

一个嵌入式软件系统自底向上的层次结构依次为：硬件抽象层、系统服务层、平台抽象层和应用层。

其中硬件抽象层由低层硬件驱动函数库（**Drivers**）和板级支持包（**BSP**）组成，用于为上层结构提供硬件无关性接口。系统服务层由硬件无关的服务（**Service**）、通用/虚拟设备驱动、组件（**Components**）和操作系统（**Operating System**）构成，例如：文件系统支持（**File System Service**），网络支持（**TCP/IP Servcie**）, 通用显示设备（**General Display Driver**）的支持, **μc/os II** 操作系统引入等等。平台抽象层由面向应用的模块、驱动和 **API** 函数组成，意在为用户应用层提供专门的环境支持。建立在这一层次之上，用户应用将支持使用不同于系统编写语言的其它高级语言进行开发，例如 **Java**、**C#**等等。所有用户程序都属于应用层。

14.3 学生用书及幻灯片注解

14.4 常见课堂问题 FAQ

14.5 参考资料

14.5.1 参考文献

14.6 背景知识

14.6.1 编译原理

本章内容牵涉到编译原理的相关知识点和定义。

14.6.2 微机原理与接口技术

本章内容牵涉到微机原理与接口技术中关于外设接口技术的相关内容。

14.6.3 C/C++语言

本章内容牵涉到 C/C++语言中关于 ANSI-C 标准、文件编译和链接、预编译处理等相关内容。

14.6.4 其它

本章内容牵涉到软件工程、软件测试、软构件技术、嵌入式操作系统的相关内容。

14.7 参考设计与实验方法

Documents Reversion History

Rev A

第一章 如何获取官方支持

Rev 1.0.0.0

- 增加了 **1.2 ATMEL** 简介。

第二章 AVR Studio4 开发环境

Rev 1.0.0.0

- 增加了 **2.2 AVR Studio 4** 简介。
- 增加了 **2.5** 参考资料。包括应用手册（**Application Note**）的章节。

第三章 普通端口操作

Rev 1.0.0.1

- 增加了 **3.2.4** 关于电平读取和 **ADC** 引脚的注意事项。
- 更新了 **3.5.2** 应用手册。

Rev 1.0.0.0

- 增加了 **3.2 GPIO** 简介。
- 增加了 **3.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **3.6** 背景知识。

第四章 中断与外中断

Rev 1.0.0.1

- 更新了 **4.5.2** 应用手册。

Rev 1.0.0.0

- 增加了 **4.2** 中断系统简介。
- 增加了 **4.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **4.6** 背景知识。

第五章 定时计数器

Rev 1.0.0.0

- 增加了 **5.2** 定时计数器简介。
- 增加了 **5.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **5.6** 背景知识。

第六章 ADC 采样与数字滤波

Rev 1.0.0.1

- 更新了 **6.2.6** 自动触发模式。
- 更新了 **6.2.8.1 Free Running** 模式。

Rev 1.0.0.0

- 增加了 **6.2 ADC** 器简介。
- 增加了 **6.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **6.6** 背景知识。

第七章 SPI 原理与总线设计

Rev 1.0.0.2

- 增加了 **7.2.6** 关于 **SPI** 工作于从机模式下的注意事项。

Rev 1.0.0.1

- 增加了 **7.2.5** 代码范例。其中包括主机发送与接收、从机发送与接收的内容。
- 删除了 **7.5** 参考资料中数据手册与应用手册里部分贴图。

第八章 U(S)ART 通讯与串行协议

Rev 1.0.0.1

- 增加了 **8.2.9** 关于 **USART** 同步模式及 **MSPI** 模式下 **XCK** 时钟频率的注意事项。
- 更新了 **8.5.2** 应用手册。

Rev 1.0.0.0

- 增加了 **8.2 U(S)ART** 简介。
- 增加了 **8.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）

的章节。

- 增加了 **8.6** 背景知识。

第九章 TWI 总线

Rev 1.0.0.1

- 删除了 **9.6.1** 节 模拟电路。
- 统一了全篇对 **TWI** 协议的解释。

Rev 1.0.0.0

- 增加了 **6.2 TWI** 协议简介。
- 增加了 **6.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **6.6** 背景知识。其中包括数据结构、数字逻辑与电路的章节。

第十章 Bootloader 自编程

Rev 1.0.0.0

- 增加了 **10.2 Bootloader** 简介。
- 增加了 **10.5** 参考资料。其中包括数据手册（**Datasheet**）和应用手册（**Application Note**）的章节。
- 增加了 **10.6** 背景知识。

第十一章 嵌入式 C 语言

Rev 1.0.0.0

- 增加了 **11.2** 嵌入式 **C** 语言简介章节。
- 增加了 **11.5** 参考资料。
- 增加了 **11.6** 背景知识。

第十二章 存储器与指针

Rev 1.0.0.0

- 增加了 **12.2** 存储器操作简介章节。

第十三章 数据结构

Rev 1.0.0.0

- 增加了 **13.2** 抽象数据类型简介。
- 增加了 **13.6** 背景知识。

第十四章 嵌入式软件构架

Rev 1.0.0.0

- 增加了 **14.2** 嵌入式软件构架简介章节。
- 增加了 **14.6** 背景知识。