同济大学

硕士学位论文

小型嵌入式逻辑控制系统研究

姓名：韦奕

申请学位级别：硕士

专业：控制理论与控制工程

指导教师：蒋式勤

20040301

声 明

本人郑重声明：本人在导师的指导下，独立进行研究工作所取得的成果，撰写成硕士学位论文“小型嵌入式逻辑控制系统研究”。除论文中已经注明引用的内容外，对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式表明。本论文中不包含任何未加明确注明的其他个人或集体已经公开发表或未公开发表的成果。

本声明的法律责任由本人承担。

学位论文作者签名：

韦奕

2004 年 3 月 12 日

摘 要

随着工业生产自动化程度的不断提高，控制系统也在向着更大容量与规模的方向发展。这使我们往往容易忽视那种只需要小型控制系统的应用场合，实际上这种低成本控制系统的需求是大量存在的。为了适应这种需求，本论文对利用单片机实现小型低成本嵌入式逻辑控制系统进行了研究。

本论文的主要工作包括以下几个方面：

1. 提出了低成本实用小型嵌入式逻辑控制器的总体方案，该方案结合了工业逻辑控制技术及最新的嵌入式技术。

2. 根据课题的具体设想，进行了逻辑控制系统电路的设计，利用 PROTEL99SE 绘制了系统的电路原理图及印刷电路板图，并进行了电路板的制作。

3. 结合 PLC 编程的特点，采用语句表的指令形式，编写了应用于逻辑控制的 C51 函数模块。其中包括 LD, AND, DIFU, DIFD, OUT，KEEP 等 16 个函数模块，并且利用软件的方法来实现控制系统中的多个定时器及计数器。

4. 针对控制系统中多个任务同时执行的状况，引入了嵌入式实时操作系统 μC/OS-II 并进行了深入的研读，分析了该操作系统的工作原理及任务调度的核心算法。进行了 μC/OS-II 在 80C552 上的移植并将其与本系统的相关功能进行结合。

5. 考虑到系统网络功能扩展的需要，在系统电路中设计了 CAN 总线通讯所需的电路部分，并且根据相关芯片的特性编制了一些用于 CAN 总线基本通讯功能的函数。

关键词：逻辑控制，单片机，嵌入式操作系统，μC/OS-II，PLC

Abstract

Accompanied with the improvement of the industrial automation. Control system keep on developing in the direction of capability and scale. It will always make us neglect the situations where the small control system is needed. Actually, these kinds of requirement are immense in china nowadays. With the instruction of my tutor, I bring forward the title of the paper combining my work experience and the embedded technology. The paper study the embedded logic control system based on the micro-controller.

The main studies carried about in the dissertation are:

1. Put forward the cheaper solution of embedded logic controller compared with PLC. Conbine logic control with embedded technology in the solution.

2. Design the circuit of the logic control system according to the concrete assumption. Make schematic drawing and Printed Circuit Board using PROTEL99SE.

3. Conbining the programming habit of PLC user and the instruction language of PLC, make programming of C51 module with reference to the need of logic control. Select 16 instructions in common use, including LD, AND, DIFU, DIFD, OUT, KEEP etc. fulfill several Timers and Counters in the way of software for the use of control system.

4. Introduce and study the embedded real time operating system μC/OS-II aiming at the status of multiple tasks operating at the same time. Analyze the work theory and the core arithmetic of task switch. Carry on the port to 80C552 and the application of μC/OS-II.

5. Consider the expandedness of the network function, design the circuit part for CAN communication in the system, make programming of functions for the basic CAN communication according to the character of relative IC.

Keywords: logic control, micro-controller, embedded operating system, μC/OS-II, PLC

第一章：绪论

1.1. 工业控制器及嵌入式系统概述

1.1.1. 工业控制器的应用现状

在现代的工业生产中，自动化设备（如 PLC，DCS，现场总线）的利用越来越广泛，这极大地节约了人力，物力，使生产的安全性及可靠性得到了很大的提高。自动化系统的发展向着网络化，信息化的方向发展，控制系统的容量向着更大的方向发展，稳定性也不断的提高，这对于造纸，钢铁，石化等控制点数多，稳定性要求很高的应用场合无疑是好的发展方向。

DCS（集散控制系统）常用在石化，钢铁行业，其系统多在 500 点以上且控制点的分布非常广泛，根据传输信息的不同，DCS 系统一般是由包括 FIELDSBUS, CONTROL BUS 和 TCP/IP 三层网络构成。

PLC（可编程逻辑控制器）是应用最广泛且最具代表性，它是 80 年代在继电器控制的基础上发展起来的，已经在工业生产上得到了极其广泛的应用，虽然工业控制器也在不断发展，新产品不断地推出，但在我国工业生产的自动化程度并不高的实际状况下，PLC 的使用前景依然是十分广阔的。

根据中国工控网市场研究部在 2003 年底到 2004 年初做的一个中国 PLC 的市场调查，我们得到了 PLC 相关产品结构的调查结果。根据业内的划分习惯和运用领域的不同将 PLC 划分为 4 个范围，主要是根据 IO 点数。图 1.1 显示了这一调查的相关数据结果。

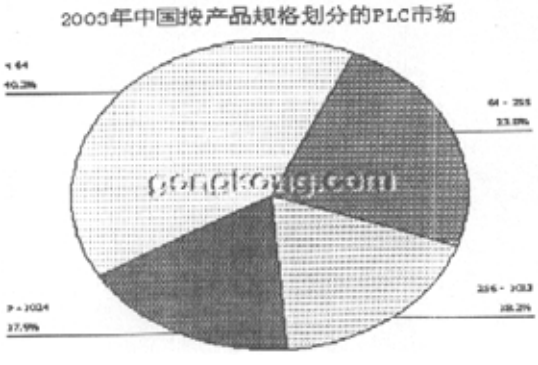


图 1.1 中国 PLC 产品应用结构调查结果

大型 PLC(DCS)指的是不少于 1,024 个 IO 点的系统。这一类 PLC 主要用于冶金，自动化生产线和电厂。该系列产品随着钢铁行业及电厂的大幅度增长而不断增加；中型 PLC 指的是 256－1,023 点的系统。这一类 PLC 主要用做控制系统，运用于冶金，电力，造纸，化工，加工/组装生产流水线等领域。小型 PLC 指的是 64－255 点的系统。主要用于设备控制，也有用作小型系统控制器的。

微型系统一般少于 64 点，这一类 PLC 主要用于单台设备的监控，在纺织机械，数控机床，塑料加工机械，小型包装机械上运用广泛。而这部分产品的应用比例最大，占到所有产品的 40%左右。而本论文主要针对的目标也就是占控制器应用比例最大的微型系统。

1.1.2. 嵌入式处理器及嵌入式操作系统现状

嵌入式系统主要由嵌入式处理器/微控制器，包括存储器、I/O 端口等在内的外围硬件和包括嵌入式操作系统、应用软件系统的软件部分组成。嵌入式系统几乎深入到生活中的所有电器设备，如 PDA、移动计算设备、汽车、微波炉、数字相机、家庭自动化系统、电梯、空调、安全系统、自动售货机、消费电子设备、工业自动化仪表与医疗仪器等。

嵌入式系统一般指非 PC 系统，嵌入式系统的核心部件是各种类型的嵌入式处理器，目前主要有以下几类：嵌入式微处理器，其基础是通用计算机的 CPU；嵌入式微控制器，就是我们通常所说的单片机，将许多必要的功能与外设集成在一块芯片里；嵌入式 DSP 处理器，此种处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法；嵌入式片上系统（SOC），利用硬件描述语言及 SOC 设计库，可以将嵌入式系统大部分集成到一块或几块芯片中。目前据不完全统计，全世界嵌入式处理器的品种总量达到 1000 多种，体系有 30 多个，其中 8051 体系的占有多半，生产 8051 单片机的半导体厂家有 20 多个，仅 PHILIPS 就有近 100 种产品。[6][8]

嵌入式系统软件包括操作系统软件（要求实时与多任务操作）和应用程序编程。嵌入式操作系统近年来得到了飞速的发展，从支持 8 位微处理器到 16 位，32 位，从支持单一品种的微处理器芯片到支持多品种微处理器芯片，从只有实

时内核到除了内核外还提供其他的功能模块如：高速文件系统，TCP/IP 网络系统等。嵌入式处理器的应用软件是实现嵌入式系统功能的关键，对嵌入式处理器系统软件和应用软件的要求和通用计算机有所不同：一般要求将软件固化在存储器芯片或单片机本身中；要求软件代码高质量，高可靠性；系统软件（操作系统）的高实时性是基本要求。[9] [10]

现在的嵌入式操作系统种类繁多，大体上可以分为两类一商用型和免费型。商用型的系统性能稳定可靠，但价格昂贵。免费型的主要有两种，LINUX 和 μ COS，本文将重点研究 μ COS-II 系统，uC/OS-II 是一个针对嵌入式系统而开发的实时操作系统，具有可移植性强、可固化、可剪裁、占先式、多任务、可确定性、稳定性和可靠性等特点。由于它的高度灵活性，又是免费的，已经在各个领域取得了广泛的应用。[8]它是源码完全公开的系统，是占先式的多任务系统，可以针对多种微处理器，而且绝大多数代码使用 C 语言编写的，非常适合我们的学习与实践.关于 μ COS-II 操作系统的具体特点我将在接下来的章节进行详细的介绍。

1.2. 课题的提出

在涉及控制系统时，我们往往容易忽视那种只需要小的控制系统的场合，实际上这种场合是大量存在的，从前面的 PLC 市场分析图表中我们可以清楚地看到这一点。尤其在我国现阶段的社会生产水平下这种应用场合是大量存在的，例如饮料生产线的传送控制部分，造纸工艺中的纸浆制配部分等，它们的共有特点是控制的点数不多，一般输入输出点总和只有 20 到 40 点，而且基本上是以逻辑量的控制为主。在这样的项目背景下，系统的成本控制往往占有重要的地位，因而对于控制系统不仅要求实现控制的目标，往往也要求节约成本。

本人不久前曾参与过一个茶饮料后段输送带控制系统（23 点输入，15 点输出）的设计，组态过程，在此过程中深入的了解了 PLC 控制器的性能价格关系，事实上各大 PLC 设备供应商都提供小型的控制系统，但是它们的价格都比较高，例如说在自动化产品领域中价格比较低的欧姆龙公司的产品 CQM2AE（24 点入，16 点出）的价格为 3000 元左右，而像 GE 等公司的相关产品的价格基本上都是欧姆龙产品的 2 到 3 倍，这对于一个小型控制系统而言显得成本较大，而事实上 PLC 中提供的大多数功能(例如许多针对内存的指令)在实践中几乎没有利用到。

综合上面所介绍的情况，本人考虑利用单片机来设计一个系统，它主要实现一个小型 PLC 的精简的控制功能，主要是逻辑控制功能及与上位机进行通讯的功能。又可以让 PLC 使用者能够很快地熟悉并使用本系统。它的优点在于实现基本功能的同时成本可以大大的降低。此外在本系统中结合最新的嵌入式操作系统技术，使之成为一个可以处理多个任务的控制系统。

1.3. 本论文的主要研究内容

本论文的主要目标是设计一个嵌入式逻辑控制系统，目标是可以实现一些常用的逻辑控制功能。具体工作包括：

1. 分析当前工业控制器的市场，应用状况及相应的性能价格关系，将工业逻辑控制与嵌入式操作系统技术相结合，提出了与 PLC 相比较更加简化实用的，低成本的嵌入式逻辑控制器的总体方案。

2. 根据课题的具体设想，进行嵌入式逻辑控制系统电路的设计与相关芯片的选型，利用 PROTEL99SE 绘制了系统的电路原理图及印刷电路板图，并进行电路板的制作。

3. 结合 PLC 编程中语句表的指令形式，以 OMRON PLC 语句表为指令格式参照，舍弃了一些在实际现场并未得到应用的指令，选取了一些常用的逻辑控制指令，编写应用于逻辑控制的 C51 函数模块。其中包括 LD, AND, DIFU, DIFD, OUT, KEEP 等 16 个函数，并且利用软件的方法来实现控制系统中的多个定时器及计数器。

4. 针对控制系统中多个任务同时执行的状况，引入嵌入式实时操作系统 μC/OS-II 并进行深入的研读，分析该操作系统的工作原理及任务调度的核心算法。进行 μC/OS-II 在 80C552 上的移植并将其与本系统的相关功能进行结合。

5. 考虑到系统网络功能扩展的需要，在系统电路中设计 CAN 总线通讯所需的电路部分并且根据相关芯片的特性编制了一些用于 CAN 总线基本通讯功能的函数供未来扩展通讯功能时调用。

第二章:逻辑控制系统的低成本方案

根据论文前面部分的分析，本人将设计和制作一个可以部分实现小型 PLC 功能的逻辑控制系统，这个系统的最初目标是面向应用，它包括电路方案与软件两个部分。进一步的说，它可以说是建立在单片机基础上的 PLC 系统的精简形式。

2.1. PLC 系统结构剖析

实际上 PLC 的 CPU 部分是由多个微控制器和总线所综合构成的。图 2.1 是在工业控制中广泛使用的西门子 CPU945 的结构示意图。

由该图可以很直观地看到，CPU945 实际上包括了 4 个处理器/微控制器，分别是 SP90（用于逻辑程序的处理）；CP90（用于浮点计算的协处理器）；AMBUS（总线控制器）；Motorola68302 微控制器（串行通讯，控制网络，数据总线等的协调处理）。当然本 CPU 所应用的场合属于中型的控制系统，控制量与通讯量都比较大，系统做此配置也是必要的。不过从这个结构中我们可以作出以下的判断，现在的小型 PLC 系统必然也是使用多个处理器。而且为配合多个处理器的协同工作，无论是软件部分还是硬件部分的成本都会增加。

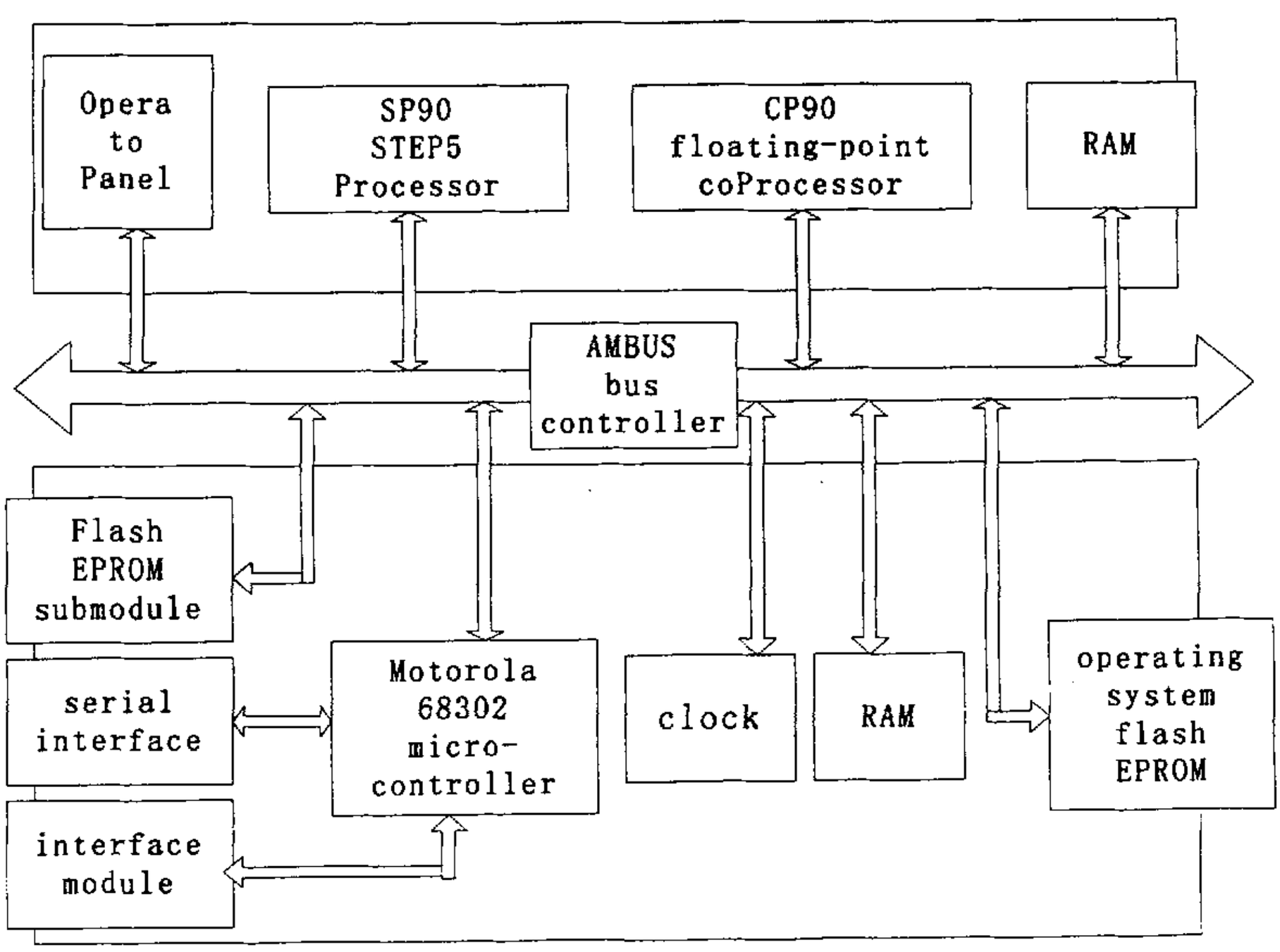


图 2.1 PLC CPU 的结构分析

联系到本人所设计的系统，由于应用于较小的控制场合，所以只使用一个微控制器，至于多个任务之间的切换，则是嵌入式操作系统的任务了。

2.2. 逻辑控制系统的电路方案

既然是面向应用的设计，关于硬件部分的功能设计，就要与实际的过程控制场合相配合。本设计的电路部分由如下几个部分构成：微处理器也就是 MCU；输入输出部分；存储器部分，包括扩展 RAM 及 ROM；复位电路，包括上电复位与按键复位；串口通讯部分；CAN 总线通讯部分；以及一路模拟量的输入输出控制。由于 PLC 系统处理的基本上是 24 伏的逻辑信号。所以本人设计的系统也以处理逻辑信号为主，系统电路板可以处理 16 路的输入与 14 路的输出（当然，适量地增加输入输出接口是完全可行的）。80C552 不带内部的 ROM，256 字节的 RAM 也不满足系统的要求，故需要扩展 ROM 及 RAM。另外考虑到未来扩展上位机的监控功能，串口通讯提供 RS232 接口可供本系统与上位机进行数据交换；带有 CAN 总线硬件接口以提供未来可能的网络扩展。另外带有一路的模拟量输入和经过 D/A 输出转换的模拟量输出。逻辑系统框架的示意图见图 2.2。

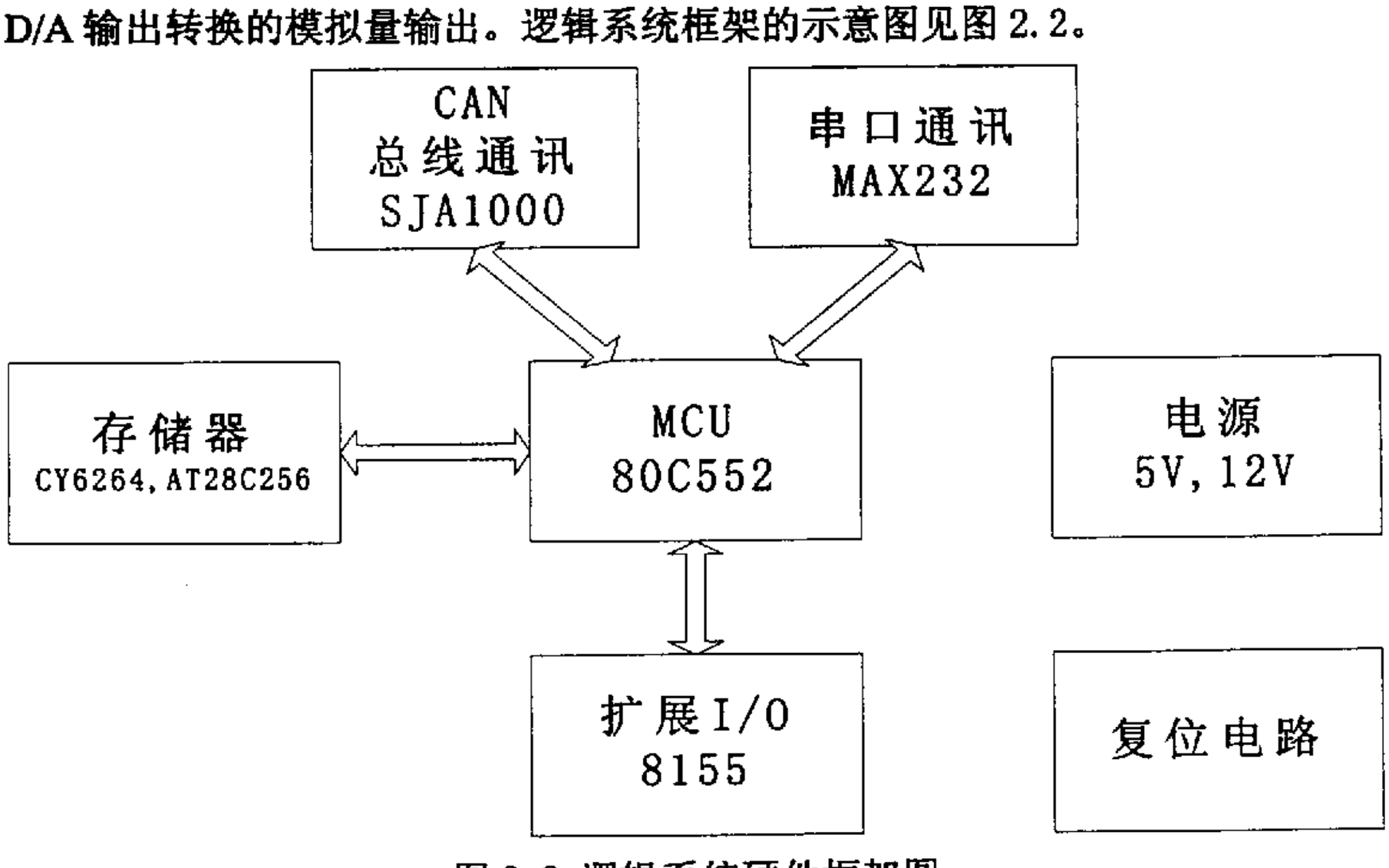


图 2.2 逻辑系统硬件框架图

2.3. 逻辑控制系统功能的总体构思

根据我的设想，系统的设计分成两个阶段，第一个阶段作为面向应用层次的

阶段，其主要功能是设计系统的电路部分，编写匹配 PLC 语句表指令的 C51 函数以适应系统逻辑控制的需要。在应用的层面，作为 PLC 的使用者，他们常用两种方式进行控制程序的编制，一是梯形图，另一个就是语句表。为适合单片机的应用环境及编程的需要，我选择语句表作为切入点。具体的指令参照欧姆龙 PLC 的相关指令。根据本人的实际工作经验，我对这些指令做了选取，将一些常用的在过程控制中会出现的语句表指令选择出来，用 C51 编写出同名的函数模块供 PLC 使用者调用。这样，一些基本的逻辑控制就可以通过调用相关函数的编程来实现。另外，为适应系统未来的网络功能的扩展，编写了一些针对 CAN 总线基本功能的函数以供系统未来扩展时调用。在这种情况下，本系统的软件环境是 KEIL C51。在基于应用的状况下，习惯于 PLC 编程语言的使用者在使用本系统时可以在编程时按照语句表的习惯写出希望的控制程序，经过 C51 编译后形成可执行的十六进制文件（HEX 文件），再通过编程器将该文件烧写在可反复使用的 EEPROM 存储器内即可使用。

第二个阶段作为提高的阶段，根据规划，最终完善的系统主要功能分为三个部分，其一是最主要的功能，即面向应用的逻辑控制的功能，也就是第一阶段的任务；其二是与上位机进行串口通讯的功能；其三是参与控制网络，具有一定的网络功能，设计上是希望它成为 CAN 网络的一个节点，以便于向 DEVICENET，CANOPEN 等流行的高层应用方面扩展。在提高阶段，作为实时控制系统的软件方面，本人引入了嵌入式操作系统 μC/OS-II，可以将控制的相关功能进行任务的分解，建立起不同的的任务，从而做到在满足实时的要求下实现所需的功能。

为何一定要引入操作系统呢？按照我的框架，如果不使用操作系统，完整的软件系统将是两个中断服务程序(CAN 通讯导致的外部中断，串口通讯中断)外加一个主程序，在中断服务程序中取得数据后在主程序中处理。显然不能在中断服务程序中处理数据，一般中断处理以最简洁、功能单一为好，这样计算机才可以实时地响应外界的随机事件。一个主程序串行地完成顺序的逻辑控制、串口处理、CAN 通讯规约的处理，假如扩展 I/O 的处理需要实时，显然这种框架无法满足要求。因此完整的软件系统应该是以嵌入式操作系统 μC/OS-II 为基础，根据上述的三个系统功能制定三个不同的任务（或者更多的任务），指定不同的优先级，通过操作系统来进行任务的切换，从而实现实时多任务的目标。

第三章:嵌入式逻辑控制系统的电路方案

系统的电路设计必然伴随着相关芯片的选择,以下将结合系统不同的功能部分对各个相关芯片的功能及其应用进行介绍。

3.1.微控制器(MCU)80C552 的功能分析

3.1.1.总体概述

根据上面的相关描述，我们要进行各个部分的芯片选择,首先是选择中心处理芯片，在此我选择了 PHILIPS 的 80C552 型的芯片，实际上，现在世界上的单片机大公司有很多家，例如 PHILIPS,INTEL,MOTOROLA, MICROCHIP, TI, ATMEL,HITACHI, SIEMENS 等等，它们的产品各有各的特点，很难说是有什么绝对的原因导致了这种选择。而用户基本上也是根据系统设计的需要及个人喜好来进行选择。

在中国，8051 单片机可以说是流行很广泛且深入人心的，事实上，80 年代初由 INTEL 发布的 51 单片机已经几乎成了一种规范，现在绝大多数的单片机产品都和 8051 单片机有着或多或少的相似之处。PHILIPS 与 INTEL 之间有一种特殊的技术互换协议。所以 PHILIPS 的 80C51 系列产品性能卓越，种类齐全，在各种 51 兼容单片机中最具代表性。

80C552 芯片的封装形式基本上是 PGA*68，在 PCB 制作时使用了 PGA 脚座，而 PROTEL 类库中的 PGA*68 的各个引脚与 80C552 的各引脚的对应关系需要一一对应地进行专门制定。

3.1.2.相对 8051 的增强功能概述

8051 单片机由于使用已久，对于它的内部结构组成在此就不再进行赘述，80C552 作为 51 系列的一个典型，它也具有 51 单片机的一般特点，除此之外它还拥有一些新增的功能与资源，对于主要的增强部分我在此做一些介绍。

并行 I/O 部分

8051 单片机一般有 4 个 8 位 I/O 口，而 80C552 具有 6 个 8 位 I/O 口，分别命名为 P0,P1,P2,P3,P4,P5。在 P0—P4 端口中，每个端口都有双向 I/O 功能，80C552 既可以从这 5 个并行 I/O 口的任意一个输出数据，也可以从它们那里输

入外部数据。P0—P4 中的每个 I/O 口内部都有一个 8 位数据输出锁存器，一个 8 位数据输出驱动器和一个 8 位数据输入缓冲器，每个数据输出锁存器与 P0—P4 同名，皆为特殊功能寄存器 SFR 中的一个，因此 80C552 内部数据从 I/O 端口输出时能得到锁存，外部数据 I/O 端口输入时能得到缓冲。

P0-P4 口都具有第二功能，在 8051 中，P1 口是没有第二功能的。

P5 口的功能与上述各端口不同，它专门用来输入 A/D 转换电路所需要的 8 位模拟量。因此 P5 电路只有一个 8 位数据输出锁存器和一个 8 位数据输入缓冲器组成，而无数据输出驱动能力。

在上述的端口中，只有 P0 口是真正的双向 I/O 口，故它具有较大的负载驱动能力，可推动 8 个 TTL 门，其他的都是准双向 I/O 口，只能推动 4 个 TTL 门。[2]

串行 I/O 端口

80C552 有两个串行 I/O 口，一个是 SIO0，称为 UART 串行 I/O 口，另一个是 SIO1，称为串行 I^2C 总线接口。80C552 的 SIO0 串行口和 8051 的 SIO 串行口完全相同，只是在标识符上略有差别。SIO0 是一个全双工的可编程串行 I/O 口，它既可以在程序中把累加器 A 送来的 8 位并行 I/O 数据变成串行数据一位一位地从发送数据线 TXD 上发送出去，也可以把从接收数据线 R_XD 上串行接收到的数据变成 8 位并行数据送给 CPU，而且这种串行发送和接收可以单独进行，也可以同时进行。

80C552 串行发送和接收利用了 P3 口的第二功能，即它利用 P3.1 引脚作为串行数据据的发送线 TXD 和 P3.0 引脚作为串行数据接收线 RXD。SIO0 的电路结构还包括串行控制寄存器 S0CON(8051 为 SCON)、电源及波特率选择寄存器 PCON 和串行数据缓冲器 S0BUF(8051 为 SBUF)等，它们都属于特殊功能寄存器 SFR。其中，S0CON/SCON 和 PCON 用于设置串行口工作方式和确定数据的发送和接收波特率，S0BUF 实际上由两个 8 位寄存器组成，一个用于存放欲发送的数据，另一个用于存放接收到的数据，起着数据缓冲作用。

串行 I^2C 总线接口（SIO1）是 80C552 的新增功能，8051 没有这个串行 I/O 口。SIO1 可以通过 P1.6 和 P1.7 引脚同外部 I^2C 总线相连，P1.6 引脚同 I^2C 总线的 SCL 线相连，P1.7 引脚同 I^2C 总线的 SDA 线相连. 由于这种连接是利用 P1.6 和 P1.7 的第二功能，因此 80C552 和接挂在 I^2C 总线上的其它器件进行串行通讯

前应预先使 P1.6 和 P1.7 端口中的相应数据输出锁存器置“1”。

80C552 串行 I^2C 总线接口是 PHILIPS 公司新一代高性能单片机的一大显著特点，这可以大大简化 I^2C 总线应用系统的硬件设计，也可以实现系统设计的模块化与傻瓜化。[2]

A/D 转换器

A/D 转换器是一种能把模拟输入电压或电流变成与它成正比的数字量，即能把被控对象的各种模拟信息变成计算机可以识别的数字信息。A/D 转换器的种类很多，但从原理上通常可以分为以下四种:计数器式，双积分式 A/D 转换器，逐次逼近式 A/D 转换器和并行 A/D 转换器。计数器式 A/D 转换器结构简单，但速度很慢；双积分式 A/D 转换器抗干扰能力强，转换精度高，但速度不理想。逐次逼近式的接口简单，转换速度也高。并行 A/D 转换器性能最高，但结构复杂，造价也很高。因而从性价比的角度来看，逐次逼近式 A/D 转换器在电路设计中的应用最为广泛。[2]

逐次逼近式 A/D 转换器也称为连续比较式 A/D 转换器。这是采用对分搜索原理来实现 A/D 转换的方法。逻辑框图如图 3.1 所示，图中 VX 为 A/D 转换器被转换的模拟输入电压；VS 是 N 位 D/A 转换网络的输出电压，其值由 N 位寄存器中的内容决定，受控制电路控制；比较器对 VX 和 VS 电压进行比较，并把比较结果送给控制电路。整个 A/D 转换是在逐次比较中形成。

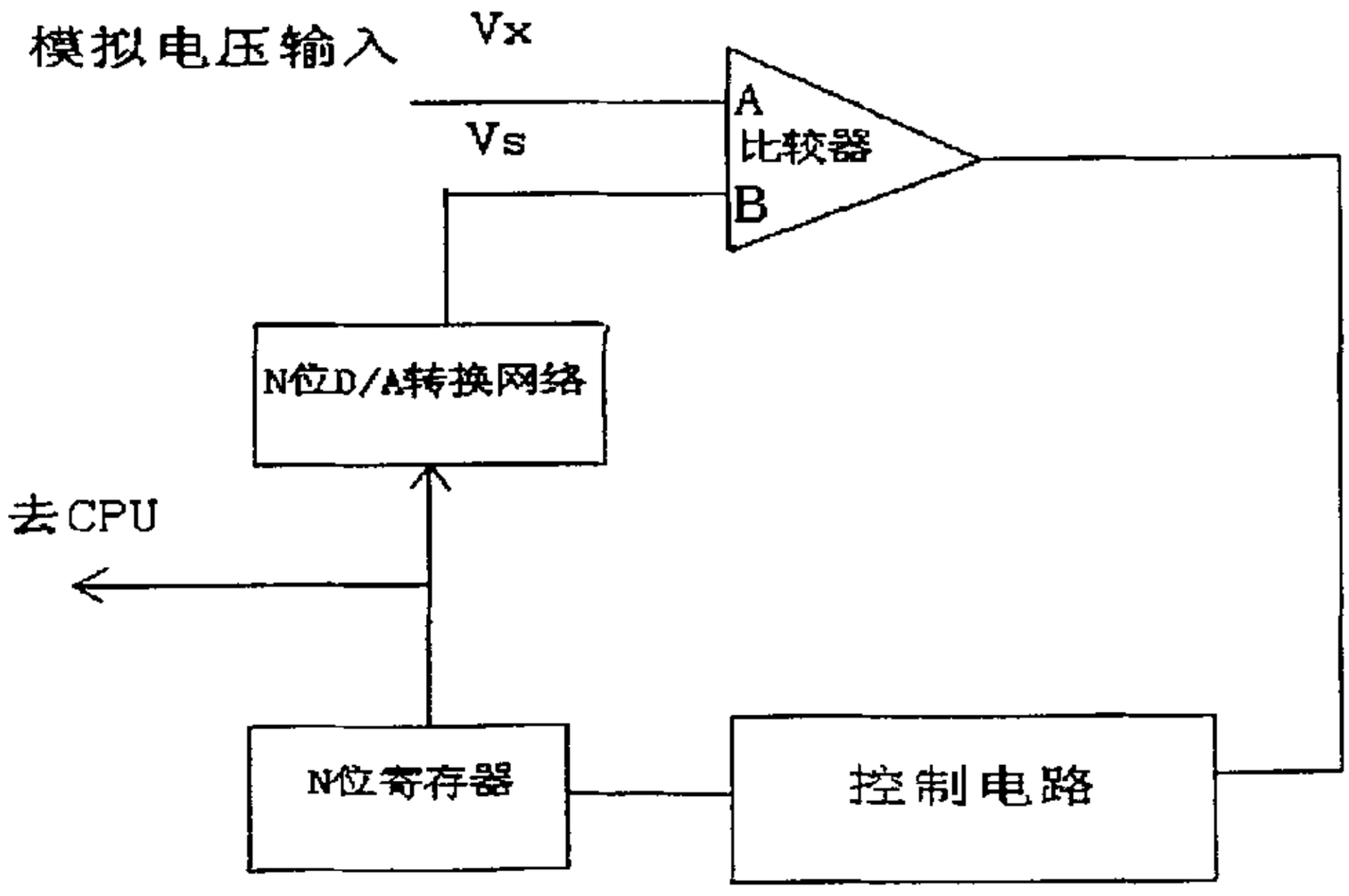


图 3.1 逐次逼近式 A/D 转换器示意图

选择80C552的原因之一在于80C552内部含有一个8路10位逐次比较型A/D转换器，若系统未来需要处理模拟量信号，可使硬件电路更为简化。通过软件方法或外部 STADC 引脚信号上升沿两种方式可以启动 A/D 转换过程。

3.2. 系统输入输出部分

该部分要考虑两个问题，一是 I/O 的扩展，另一个是关于输出负载的驱动，为了解决这个问题，需要用到两种芯片，一为 8155，一为 ULN2003。

3.2.1 I/O 扩展的实现方式

80C552 的输入输出端口通常需要扩充，以便它能够和更多的外部设备联机工作。通常 80C552 的输入输出端口的扩展方法有三种：借用外部 RAM 地址来扩展 I/O 端口；采用并行 I/O 接口芯片来扩展 I/O 端口；采用 I^2C 总线接口来扩展 I/O 端口。[16] 对于第一种方法，实施起来比较简单，所扩展的并行 I/O 端口的数量通常不限，但当外设本身没有接口能力时使用会受到限制；第三种办法和 I^2C 总线接口有关，实际上是利用串行 I/O 接口来扩展并行 I/O 接口，实现起来相对复杂，因而本人在此使用第二种方法，使用 8155 来扩展 I/O 端口。

该芯片有如下一些特点：自身容量很大，256*8 位；单五伏电源供应；完全静态操作；内部地址锁存；2 个可编程的 8 位 I/O 口；一个可编程的 6 位 I/O 口；可编程的 14 位的计数器/计时器；复用的地址与数据线。8155 的结构见图 3.2。

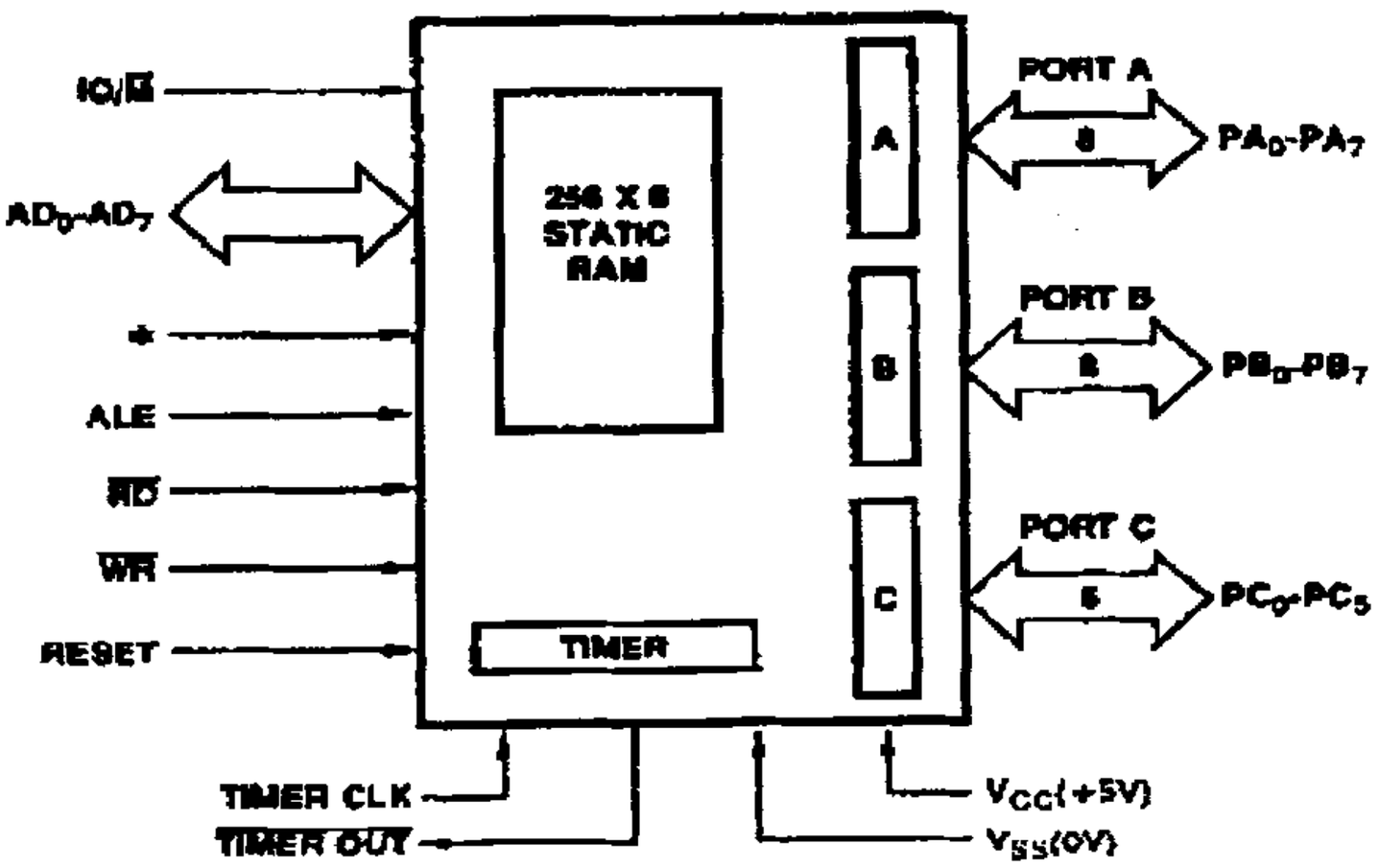


图 3.2 8155 的系统结构

在本设计中，8155 无疑扮演着重要的角色，对于 80C552 单片机来说，它自身内部共有 P0-P5 六个输入输出口。而其中 P0-P4 口皆有第二功能，P5 口用于输入 A/D 转换器的模拟量，加上 P0，P1 端口还要用来进行寻址，另外系统还可能要进行扩展，所以直接用这些输入输出口是很不明智的。

从图 3.2，我们大致可以了解 8155 的基本结构，8155 包括如下几个部分：1.RAM 存储器，容量为 256 字节，主要用于存放实时数据。存储器存储单元地址由地址缩存器输出端送来。2. I/O 寄存器，分为 A，B，C 三个端口，A 口和 B 口的 I/O 寄存器为 8 位，既可以存放外设的输出数据又可以存放外设的输入数据；C 口的 I/O 寄存器只有 6 位，用于存放 I/O 数据或命令/状态信息。8155 在某一瞬时只能选中某个 I/O 寄存器工作，这由 CPU 送给 8155 的命令字决定。3. 定时器/计数器，与 80C552 的定时器/计数器不同的是，8155 的定时器/计数器是执行减一操作的，每当计数溢出时就在 T/OUT 线上输出一个终止脉冲。

实际上 8155 内部还有几个内部控制器，分别是：1.双向数据总线缓冲器，该缓冲器是 8 位的，用于传送 CPU 对 RAM 存储器的读写数据；2.地址锁存器，也是 8 位，用于锁存 CPU 送来的 RAM 单元地址和端口地址。所以不需要外加地址锁存。3.地址译码器和读写控制器，地址译码器的三位地址由地址锁存器送过来，译码后可选择命令/状态寄存器，定时器/计数器，和 A，B，C 三个 I/O 寄存器中的某个工作。读写控制器接受 RD 和 WR 线上的信息，实现对 CPU 和 8155 间所传递信息的控制。4. 命令/状态寄存器：都是 8 位寄存器。命令寄存器存放 CPU 送来的命令，状态寄存器存放 8155 的状态字。

8155A，B，C 三口的数据传送由命令字和状态字控制。

8155 内部有 7 个寄存器，需要三位地址来加以区分，占用 A0-A7 的低三位，以 A2A1A0 为顺序，各地址分配如下：000：命令/状态寄存器；001：A 口；010：B 口；011：C 口；100：计数器低 8 位；101：计数器高 8 位；以上是 IO/M 置于高电平。当 IO/M 置于低电平时，无论 A0-A7 为何值，都选定 RAM 单元。

8155 的命令字有 8 位，用于设定 8155 的工作方式。命令字如下：

D7	D6	D5	D4	D3	D2	D1	D0
TM2	TM1	IEB	IEA	PC2	PC1	PB	PA

PA，PB：定义口 A，口 B，0 为输入，1 为输出。

IEA，IEB：口 A，口 B 中断控制，1 为允许，0 为禁止。

TM1，TM2：定时器命令，本设计中未涉及，这里略去。

PC1，PC2：定义口的工作方式。（见下表）

PC2	PC1	方　　式
0	0	口 A、口 B 为基本输入输出，口 C 输入
1	1	口 A、口 B 为基本输入输出，口 C 输出
0	1	口 A 选通输入输出，口 B 基本输入输出，口 C 控制信号。
1	0	口 A、口 B 都为选通输入输出，口 C 控制信号。

对于 8155 的定时器/计数器的具体设置以及工作方式以及存储器工作方式的具体情况，因为本设计中没有涉及，所以略去不谈。

根据本设计的实际情况，选用了两片 8155，分别用作输入与输出。根据 C51 中关于绝对地址的定义，分别对它们定义如下：

```
#define COM8155A XBYTE[0x5ff0]
#define PA8155A XBYTE[0x5ff1]
#define PB8155A XBYTE[0x5ff2]
```

这三条语句分别定义了用于输入的 8155 的命令口，输入/输出 A 口，B 口的地址为 0x5ff0，0x5ff1，0x5ff2。本系统设计中输入点为 16 点，设定的命令控制字为 0X00。

```
#define COM8155B XBYTE[0x7ff0]
#define PA8155B XBYTE[0x7ff1]
#define PB8155B XBYTE[0x7ff2]
```

这三条语句分别定义了用于输出的命令口，输入/输出 A 口，B 口的地址为 0x5ff0，0x5ff1，0x5ff2。输出点因为与 ULN2003A 的数量相关，点数设定为 14 点，其命令控制字为 0X0F。

需要补充说明的一点是，为了使用绝对地址定义的功能，在程序编写时必须要调用 C51 的库文件 absacc.h。

3.2.2. 输出驱动芯片 ULN2003A

ULA2003A 是 ULA2001 系列的一个成员，采用 DIP-16 封装，该系列都是高电

压，高电流达灵顿晶体管，每一个都包含 7 个 NPN 达灵顿管对。它的应用场合包括继电器驱动器，灯光驱动，显示驱动（LED 型和气体填充型），线形驱动等。ULN2003A 可以和 TTL 与 5V 的 CMOS 设备直接连接操作。

因为在本人的设计中，希望电路板最后的输出对象是常规的工业逻辑信号，具体的说是输出一个继电器信号，8155 输出本身的驱动能力是不够的。而要可以驱动继电器，应用类似 ULN2003A 的芯片是必须的，所以该芯片用来实现输出驱动的需要。

3.2.3. 片外扩展多个芯片的选择

在本系统中，由于扩展了两片 8155，随机存储器，加上 CAN 总线控制器 SJA1000，共有 4 个芯片需要系统进行地址译码，从而选择不同的器件进行工作。在这里，我选择了较为常见的 74HC138 芯片。

74HC138（3-8 译码器）是一种高速的，基于 CMOS 工艺的 3-8 译码器/选择器，在保持了 CMOS 器件的低功耗的优点的基础上，还具备了 TTL 器件所拥有的高速的特征（其状态真值表如表 3.1 所示）。它拥有三个选择输入端和三个使能输入端，两个低电平有效的使能输入端和一个高电平有效的使能输入端可以用做门控开关，可以使本译码器在不增加外部反相器的情况下扩充至 24 线的译码器，而增加一个反相器就可以扩充为 32 线的译码器，在需要多重译码的应用场合，一个使能输入端就可以作为一个数据输入端。

Inputs					Outputs							
Enable		Select										
G1	G2 (Note 1)	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

表 3.1 74HC138 的真值表

一个输入保护电路确保供应电压 VCC（极限最大植 0 到 7 伏，建议值 2-5.5

伏）安全地加在输入针脚上，于是这种设备可以用于 5 伏-3 伏的系统中，也可以电池作为后备的系统中，保护电路可以保护不匹配的电源供应及输入电压所造成的芯片损坏。译码器的延迟时间往往少于典型的存储器访问时间，所以这种译码器的引入不会造成系统时间的延迟。

3.3 存储器电路

3.3.1 地址锁存的必要性及实现

8051 系列芯片引脚中没有专门的地址总线和数据总线，在向外扩展存储器和接口时，由 P2 口输出地址总线的高 8 位 A15-A8，由 P0 口输出地址总线的低 8 位 A7-A0，同时对 P0 口采用了总线复用技术，P0 口又兼做 8 位双向数据总线 D7-D0，即由 P0 口分时输出低 8 位地址或输入/输出 8 位数据。因此需要在片外加用于地址锁存的芯片。这里使用的是 SN74LS373。

SN74LS373 片内集成了 8 个锁存器，提供三态总线驱动输出，可以对低阻抗负载进行完全并行地访问，对控制输入提供缓冲。其高阻三态使它们在总线系统中可以直接连接或驱动总线而不需要任何上拉元件。它们通常用来作为缓冲寄存器，输入/输出端口，双向总线驱动器以及工作寄存器。

当输出使能端为低电平且锁存使能端为高电平时，输出伴随 D 端的输入；输出使能端为低电平但锁存使能端为低电平时，输出端保持为上一状态不变；当输出使能端为高电平时，无论锁存使能及 D 端输入为何值，输出均为高阻态。

在本设计中，在控制器外扩展了 RAM 及 ROM，因为对它们进行寻址的需要，利用 SN74LS373 进行地址锁存。

3.3.2 静态随机存储器

CY6264是一种高性能的静态随机存储器，它采用的是CMOS工艺制造，它易于进行内存扩展，而且输入输出电平与TTL电平相匹配，通常封装形式为DIP-28，其状态真值表为（见表3.2）：

它的组织形式为8K*8，通过对下列信号的正确控制可以方便地进行内存扩展，这些信号分别是：低电平有效的片选信号CE1；高电平有效的片选信号CE2；低电平有效的输出选通信号OE。

$\overline{CE_1}$	CE_2	$\overline{WE}$	$\overline{OE}$	Input/Output	Mode
H	X	X	X	High Z	Deselect/Power-Down
X	L	X	X	High Z	Deselect
L	H	H	L	Data Out	Read
L	H	L	X	Data In	Write
L	H	H	H	High Z	Deselect

表3.2 CY6264的状态真值表

低电平有效的控制信号WE控制内存的读写操作，当CE1与WE同时处于低电平时且CE2处于高电平时，在八个数据输入/输出管脚（I/O0-I/O7）上的数据就被写入存储器，具体写入位置由地址管脚（A0-A12）上的数值来决定。通过选择该设备并使输出有效，也就是CE1与OE置低电平，CE2与WE同时置高电平，便可以实现该设备的读取，由地址管脚（A0-A12）上的数值所决定的存储器内相对应的数值便会出现在设备的数据输入/输出管脚上。除非芯片被选择且输出被使能，否则输入/输出管脚始终保持高阻态。

在本系统中，CY6264通过74LS138进行扩展，其基址为0Xe000，占用的地址空间为0Xe000-0Xffff。

3.3.3. 只读存储器

AT28C256 是一种高性能的电可擦除/编程的只读存储器，在不同的 IC 应用场合，它有不同芯片封装方式，TSOP，PGA，LCC，SOIC，DIP 的均有，本人的设计中选用的是封装形式为 DIP-28 的一种。

它的 256K 位的存储空间的组织方式为 32K*8，它是由 ATMEL 公司采用 CMOS 工艺制造的，它的读取速率为 150 纳秒，功耗仅为 440MW。工作电流为 50 毫安，而处于待机状态时电流仅为 200 微安。

AT28C256 的读写访问与静态随机存储器的读写访问方式相似，都不需要在外部扩展元件，AT28C256 包含一个 64 字节的页寄存器，从而允许同时写入 64 个字节，在一个写周期里，地址和 1 到 64 字节的数据都被内部缩存，地址及数据线在另一项操作时才会被释放。在写周期的初始化之后，芯片会根据内部的控制时钟自动地将缩存的数据写入。当一个写周期结束，芯片会自动检测到一个特

定的位，从而一个新的对芯片的读写访问又可以开始了。

AT28C256 的读取：与对静态随机存储器的读取相似，当 CE 与 OE 置低电平，WE 置高电平，便可以实现该设备的读取，由地址管脚上的数值所决定的存储器内相对应的数值便会出现在设备的数据输出管脚上。当 CE 或 OE 中的一个处于高电平，所有的输出管脚就会处于高阻态，这种双控制线为设计者在防止系统的总线冲突方面提供了更多的灵活性。

AT28C256 的字节写入：在 WE 或 CE 的输入端加上一个低电平脉冲，同时在 WE 或 CE 中的另一个输入端上加上低电平，在 OE 管脚上加上高电平就开始初始化一个写周期，在 WE 或 CE 输入端的低电平脉冲的下降沿，地址被锁存，而在 WE 或 CE 输入端的脉冲的第一个上升沿，数据被锁存。一旦一个字节写入开始，它就会自动记时至结束。

3.4. 与上位机的串口通讯

用于工业自动化的控制器，一般都具备一定的通讯功能。而就 PLC 而言，所有厂商的机型都具有串行通讯的功能。本设计也考虑到未来系统与上位机或其它控制器的通讯，所以设置了这个基本的功能。

8051 系列单片机片上有 UART(通用异步接收/发送)用于串行通讯。MAX232CFE 芯片是一种用于 EIA/TIA-232E 与 V.28/V.24 通讯接口的发送/接收设备。它满足所有的 EIA/TIA-232E 与 V.28 的特性。它只需要 5V 的电源供应，因而适用于±12V 不易得到的场合。在那些电池供电的系统里，这种芯片显得尤其适合，因为它的低功耗特性使它的功耗只有不到 5μW。另外它还适合那种印刷电路板空间比较紧张的应用，因为它有四路发送与接收。该种芯片通常的应用场合为移动电脑，低功率的调制解调器，接口转换，电池供电的 RS232 系统，多点的 RS232 网络。[14]

MAX232 内含三个部分：DC-DC 电压转换；RS232 驱动与 RS232 接收。

DC-DC 转换：通过两步将+5V 转化为±10V，先通过电容 C1 将+5V 输入转化为在 V+输出的电容 C3 上的+10V，第二步转化是通过电容 C2 将+10V 转化为在 V-输出的电容 C4 上的-10V。

驱动部分：当 VCC 为 5V 且负载为 5KΩ的 RS232 接收时，典型的驱动输出电

压为±8V，这种输出保证它满足 EIA/TIA-232E 与 V.28 的相关特性，而驱动输入对于 TTL 与 CMOS 都匹配，没有用到的输入可以悬空放置，因为在芯片内部各输入端都有接到 VCC 的 400KΩ上拉电阻。这些上拉电阻可以强迫未接的驱动输出为低电平。

接收部分：EIA/TIA-232E 与 V.28 定义高于 3V 的电平为逻辑 0，于是所有的接收都要反向，输入门限值被设定在 0.8V 与 2.4V，于是接受器对 TTL 电平输入的处理和 EIA/TIA-232E 与 V.28 的要求相符合。

在实际的应用中，本芯片的外部需要加上 5 个 1μF 的电容进行耦合作用，其内部的示意结构及实际应用的管脚接法如图 3.3 所示。实际上，单片机串口通讯的技术已经得到了广泛的应用，在具体的实施细节上这里就不做赘述了。

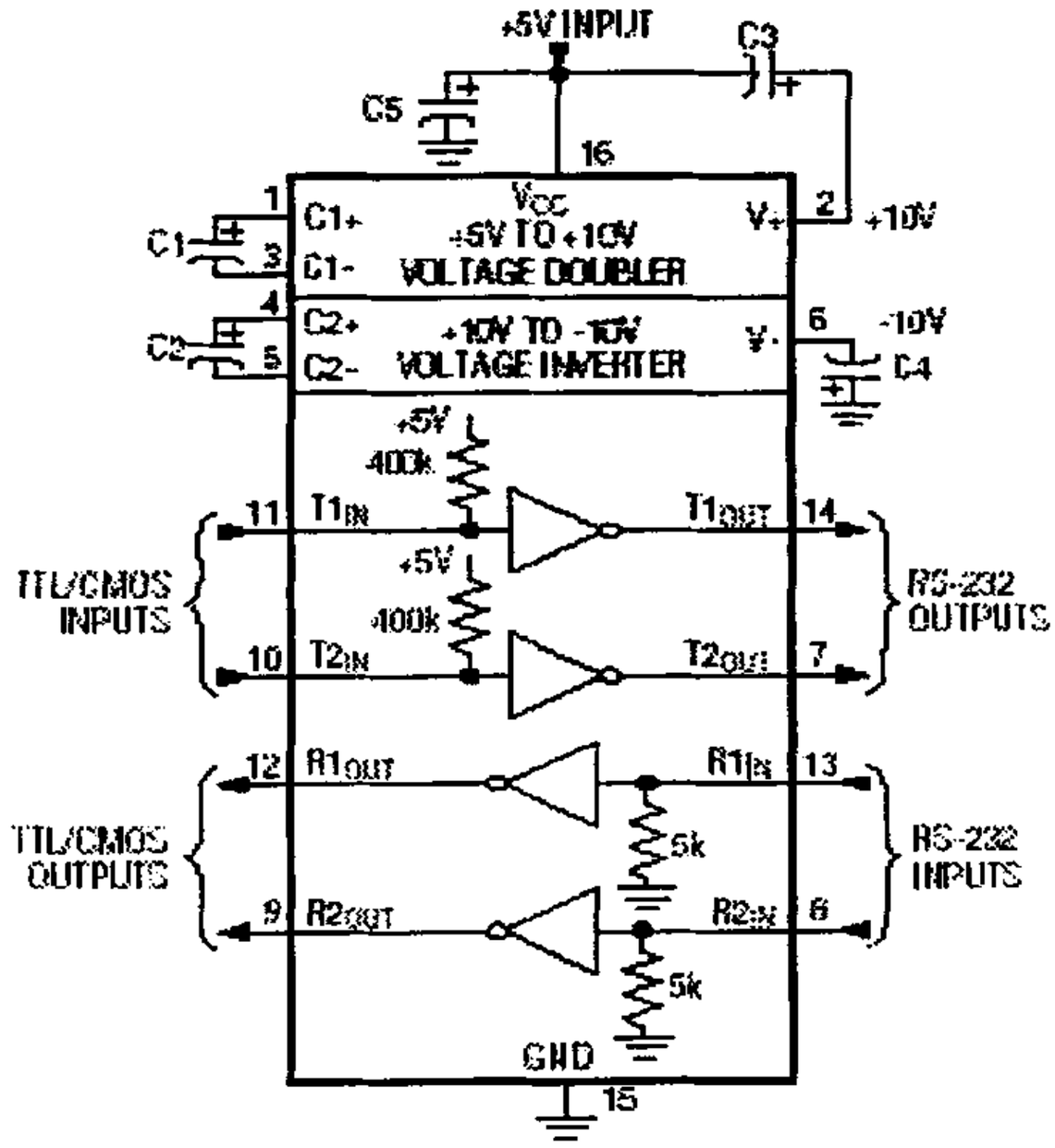


图 3.3 MAX232 的结构示意图及典型应用

3.5. CAN 网络通讯的方案

在如今的控制系统中，网络化已经非常普遍，这不仅是控制的需要，也是生产管理的需要。绝大多数的系统都具备网络通讯的功能，或是控制器自身带有网络接口，或是通过通讯模块来实现这一功能。在本人的系统规划中，是希望它能够成为 DEVICENET 网络的一个节点。DEVICENET 在工业控制和制造业领域得到了

广泛的应用，是为 PLC 和智能传感器设计的 。DEVICENET 是基于 CAN 总线的高层协议，也就是应用层的协议。CAN，全称为 Controller Area Network,即控制器局域网，是国际上应用最广泛的现场总线之一。在本论文阶段，重点讨论了在本系统上如何实现 CAN 网络功能以及一些 CAN 基本功能的实现方案。

完整的 CAN 网络通讯具有如下的特性：

系统灵活性：不需要改变任何节点的应用层及相关的软件或硬件，就可以在 CAN 网络中直接添加节点。

报文路由：报文的内容由识别符命名。识别符不指出报文的目的地，但解释数据的含义。因此，网络上所有的节点可以通过报文滤波确定是否应对该数据作出反应。

多播：由于引入了报文滤波的概念，任何数目的节点都可以接收该报文，同时对此报文作出反应。

数据连贯性：在 CAN 网络内，可以确保报文同时被所有的节点接收(或同时不被接收)。因此，系统数据的连贯性是通过多播和错误处理的原理实现的。

此外还有成本低，传输距离远，传输速率高及自动重发等优点。

3.5.1. CAN 总线通讯的基本特点

CAN 通讯协议主要描述设备之间的信息传递方式。对应于 OSI 的开放系统互连模型(OSI)，每一层与另一设备上相同的那一层通讯。实际的通讯发生在每一设备上相邻的两层，而设备只通过模型物理层的物理介质互连。CAN 总线规范定义了该模型的最下面两层：物理层和数据链路层。物理层规定了通讯介质的物理特性。如电气特性和信号交换的解释。数据链路层规定了在介质上传输的数据位的排列和组织。如数据校验和帧结构。[19]

物理层的作用是在不同节点之间根据所有的电气属性进行位信息的实际传输。CAN 能够使用多种物理介质，例如双绞线，光纤等。最常用的就是双绞线。

信号传输使用差分电压传送，两条信号线称为“CAN-H”和“CAN-L”，静态时均为 2.5V 左右，此时状态表示为逻辑“1”，也叫做“隐性”，用“r”表示。用 CAN-H 比 CAN-L 高表示逻辑“0”，称为“显性”，用“d”表示。此时通常的电压值为 CAN-H=3.5V 和 CAN-L=1.5V。

数据链路层主要具备报文分帧，仲裁，应答，错误检测等功能，以下介绍一下 CAN 的数据帧结构。在 CAN 组成的系统中，数据在各节点间的发送与接收是以四种不同格式的帧出现和控制。这四种帧分别为：数据帧，远程帧，出错帧和超载帧。

数据帧：用于将数据从发送器发送至接受器。数据帧由 7 个不同的位场组成：帧起始，仲裁场，控制场，数据场，CRC 场，应答场及帧结尾。用户可见的帧格式为：帧描述符(2Bytes)+数据。其中帧描述符由报文识别码 ID（11Bit），请求数据位 RTR(1BIT)和数据长度 DLC（4BIT）组成。在数据帧中请求数据位 RTR 必须为 0，数据长度不能大于 8。

远程帧：由节点发送，以请求发送具有相同报文识别码 ID 的数据帧。它没有数据部分。用户可见的帧格式为：帧描述符(2Bytes)。在远程帧中请求数据位 RTR 必须为 1。这时的数据长度 DLC 可以用来作为识别相同报文识别码 ID 的远程帧节点。

出错帧：可由任何节点发送，以检测总线错误。

超载帧：用于提供先前和后续的数据帧或远程帧之间的附加延时。[19]

3.5.2. CAN 总线控制器 SJA1000 的特点及应用

SJA1000是一个独立的CAN控制器，其结构图见图3.4。它具有灵活的微处理器接口，允许接口大多数微型处理器或微型控制器；具有可编程的CAN输出驱动器，内部具有64个字节接收FIFO，延长了最大中断服务时间。为了连接到主控制器，SJA1000提供一个复用的地址/数据总线和附加的读/写控制信号。SJA1000可以作为主控制器外围存储器映射的I/O器件。

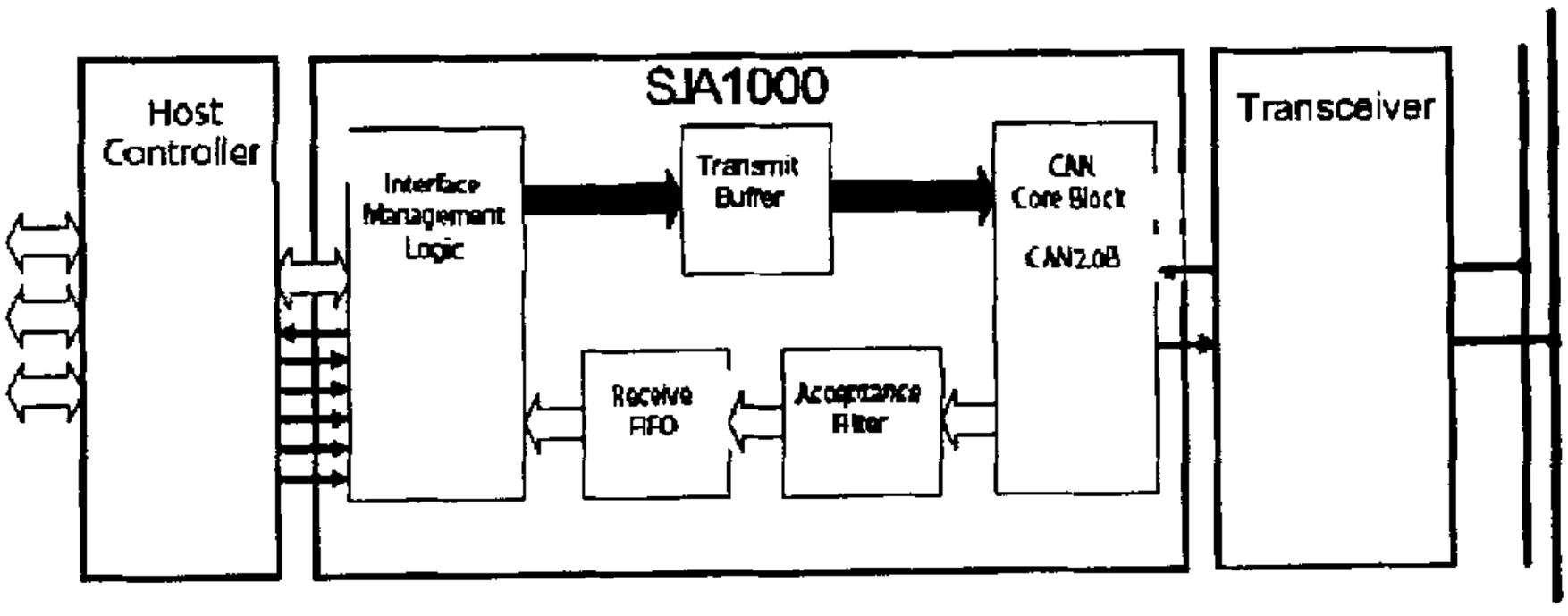


图3.4 SJA1000 的结构图

SJA1000支持直接连接到两个著名的微型控制器系列：80C51和68xx。通过SJA1000的MODE 引脚可选择接口模式：

Intel模式：MODE=高

Motorola模式：MODE=低

本系统中所用的微控制器为INTEL系列，所以MODE引脚常接高电平。SJA1000的功能配置和行为由主控制器的程序执行。因此SJA1000能满足不同属性的CAN总线系统的要求。主控制器和SJA1000之间的数据交换经过一组寄存器(控制段)和一个RAM（报文缓冲器）完成。RAM的部分寄存器和地址窗口组成了发送和接收缓冲器，对于主控制器来说就象是外围器件寄存器。

根据CAN规范，CAN核心模块控制CAN帧的发送和接收。

接口管理逻辑负责连接外部主控制器，该控制器可以是微型控制器或任何其它器件经过SJA1000复用的地址/数据总线访问寄存器和控制读/写选通信号都在这里处理。

SJA1000的发送缓冲器能够存储一个完整的报文(扩展的或标准的）。当主控制器初始化发送，接口管理逻辑会使CAN核心模块从发送缓冲器读CAN报文。

当收到一个报文时，CAN核心模块将串行位流转换成用于验收滤波器的并行数据通过这个可编程的滤波器，SJA1000能确定主控制器要接收哪些报文。

所有收到的报文由验收滤波器验收并存储在接收FIFO。储存报文的多少由工作模式决定而最多能存储32个报文。因为数据超载可能性被大大降低这使用户能更灵活地指定中断服务和中断优先级。

利用SJA1000来建立CAN通讯的主要工作就是完整地了解SJA1000内部寄存器的分布及其功能，并在程序中正确地对它们进行设置。SJA1000必须在上电或硬件复位后设置CAN通讯。在由主控制器操作期间，它可能会发送一个(软件)复位请求,SJA1000会被重新配置(再次初始化)。复位模式是SJA1000的一个重要工作模式。

SJA1000的地址区包括控制段和信息缓冲区。微控制器和SJA1000之间状态，控制和命令信号的交换都是在控制段中完成的。控制段在初始化载入是可被编程来配置通讯参数的(例如位时序）。微控制器也是通过这个段来控制CAN总线上的

通讯的。在初始化时CLKOUT 信号可以被微控制器编程指定一个值。

应发送的信息会被写入发送缓冲器。成功接收信息后，微控制器从接收缓冲器中读取接收的信息，然后释放空间以做下一步应用。

初始载入后，寄存器的验收代码，验收屏蔽，总线定时寄存器0和1以及输出控制就不能改变了。只有控制寄存器的复位位被置高时，也就是进入复位状态时才可以访问这些寄存器。

SJA1000有两种工作模式，复位模式和工作模式。在这两种不同的模式中访问寄存器是不同的。SJA1000的寄存器从功能上可以分为两个部分，不同部分的功能由几个寄存器共同作用来实现。

1：选择不同的操作模式：

控制寄存器(CR) 地址为0 用于选择复位模式；

命令寄存器(CMR) 地址为1 用于选择睡眠模式命令；

时钟分频器(CDR)地址为31 用于设置时钟信号；

2：设定CAN通讯的要素：

验收码寄存器(ACR) 地址为4

验收屏蔽寄存器(AMR) 地址为5 这两个寄存器用于验收滤波器位的模式选择；

总线定时寄存器0(BTR0) 地址为6

总线定时寄存器1(BTR1) 地址为7 这两个寄存器用于位定时参数的设置；

输出控制寄存器(OCR) 地址为8 用于输出驱动器属性的选择；

命令寄存器(CMR)地址为1 用于自接收，清除数据超载，释放接收缓冲器，中止传输和传输请求的命令；

状态寄存器(SR) 地址为2 用于报文缓冲器的状态，CAN核心模块的状态；

中断寄存器(IR) 地址为3 指示CAN中断标志；

控制寄存器(CR) 地址为0 用于使能和禁能中断事件；

各个寄存器内部的不同位有不同的作用，这里就不仔细介绍了。

发送缓冲器 地址从10到19，存放8个字节的发送数据及识别码。

接收缓冲器 地址从20到29，存放8个字节的接收数据及识别码

3.5.3. 应用 CAN 总线收发器 PCA82C250

在前面已经提到，CAN总线协议有自身的标准，这个标准的基本作用是定义了通讯链路的数据链路层与物理层。而物理层被细分成3个子层。它们分别是：物理信令，其作用为位编码定时和同步；物理媒体连接，定义驱动器和接收器特性；媒体相关接口，它通过总线连接器来实现。

Philips半导体的收发器PCA82C250是用来实现物理媒体连接子层的器件。物理信令子层和数据链路层之间的连接是通过集成的协议控制器实现的，例如上面介绍的SJA1000。而媒体相关接口负责连接传输媒体，譬如将总线节点连接到总线的连接器，这里选用PCA82C250。

PCA82C250收发器是协议控制器和物理传输线路之间的接口。它们可以用高达1Mbit/s的位速率在两条有差动电压的总线电缆上传输数据。

在实际的应用方面，协议控制器通过串行数据输出线(TX)和串行数据输入线(RX)连接到收发器。收发器通过有差动发送和接收功能的两个总线终端CANH和CANL连接到总线电缆。协议控制器输出一个串行的发送数据流到收发器的TxD引脚。内部的上拉功能将TxD 输入设置成逻辑高电平，也就是说总线输出驱动器默认是被动的。在隐性状态中，CANH和CANL 输入通过内部的接收器输入网络，偏置到2.5V的额定电压。另外，如果TxD是逻辑低电平，总线的输出级将被激活，在总线电缆上产生一个显性的信号电平。在显性状态中CAN_H的额定电压是3.5V，CAN_L是1.5V。在51单片机系统中，SJA1000与82C250配合使用的典型原理图如图3.5所示。具体的线路连接可以由附录1中看到。

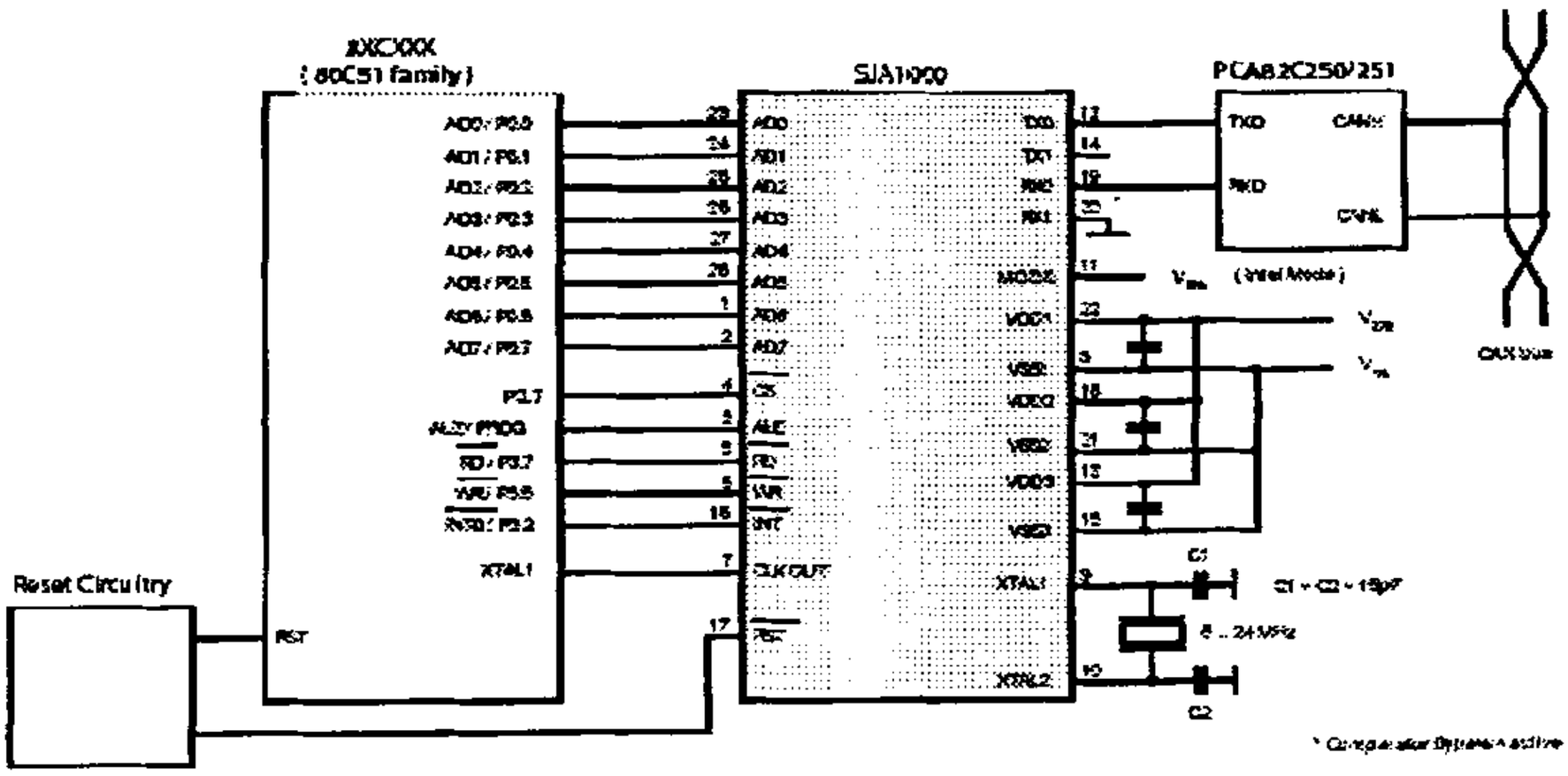


图3.5 SJA1000及82C250的典型应用

如果没有一个总线节点传输一个显性位，总线处于隐性状态，即网络中所有TxD输入是逻辑高电平。如果一个或更多的总线节点传输一个显性位，即至少一个TxD输入是逻辑低电平，则总线从隐性状态进入显性状态(通过线与功能实现)。接收器的比较器将差动的总线信号转换成逻辑信号电平。并在RxD输出。接收到的串行数据流传送到总线协议控制器译码。接收器的比较器总是活动的，也就是说当总线节点传输一个报文时，它同时也监控总线。收发器可以直接连接到协议控制器及其应用电路，一些控制器提供一个模拟的接收接口(RX0，RX1)。RX0一般需要连接到Rx0输出，RX1需要偏置到一个相应的电压电平。这可以通过VREF输出实现。

3.5.4. 如何在系统中应用 CAN 总线进行通讯

在本系统关于 CAN 总线实现的硬件设计上，采用的是微处理器控制 SJA1000 加 PCA82C250 的组合方式。

利用SJA1000通过CAN总线建立通讯的步骤可以分成两个阶段：

第一阶段是在系统上电后，此时工作为：

-根据SJA1000的硬件和软件连接设置主控制器

-根据选择的验收滤波，位定时等等设置CAN控制器的通讯，这也是在SJA1000硬件复位后进行

第二阶段是在应用的主过程中，此阶段的主要工作为：

-准备要发送的报文并激活SJA1000发送它们

-对被CAN控制器接收的报文起作用

-在通讯期间对发生的错误起作用

在程序的总体流程方面可以描述为：

微控制器的上电复位→应用程序的复位过程→等待SJA1000正确上电完毕→配置用于微控制器与SJA1000通讯的控制线(中断，复位，片选等)→初始化SJA1000→利用SJA1000进行通讯的主过程及中断服务过程→程序结束

根据以上部分的描述，关于SJA1000的使用有以下几点需要注意：

1：在设计微处理器与SJA1000的接口电路时，首先选择好SJA1000的接口模

式，其次要注意SJA1000的片选地址与其它的外部存储器没有冲突，还应注意其复位电路为低电平有效。

2：微处理器对SJA1000的访问控制是以外部存储器的方式来访问SJA1000的内部存储器，所以应该正确定义SJA1000的内部存储器的访问地址。

3：微处理器可以通过中断或轮询的方式来访问SJA1000。

4：SJA1000有两种不同的访问模式，工作模式和复位模式。对SJA1000的初始化只能在复位模式下进行。初始化包括设置验收滤波器，总线定时器，输出控制，时钟分频的控制等。而且设置后最好进行校验。

5：向SJA1000的发送缓冲区写入数据时，一定要检查发送缓冲区是否处于锁定状态，如果锁定，数据将丢失。

3.6. 电源器件及复位电路

在本设计中用到了两种电压，分别是 5V 与 12V，为了得到这两种电压，我选用了 L7800 系列的两种器件，7805 与 7812，下面对它们做些简要的介绍。

该系列的变压器都有多种包装方式，TO-3，D^2PAK，TO-220，TO220FP 等，7805 固定输出为 5V，7812 固定输出为 12V，多种的包装方式使该系列器件在广泛的应用场合都适用，这些变压器能提供卡上的稳压，可以清除单点稳压所带来的问题，每种类型都有内部的电流限制，过热中断与安全保护。如果有适当的散热措施，它可以承受 1A 的工作电流。尽管设计为固定的电压调节器，但这些设备在与其他器件共同工作时可以得到可调节的电压与电流。它们都有三个引脚，其中引脚 1 接输入电压，引脚 2 接输出电压，引脚 3 接地。接入的输入电压最大可以到达 35V。为了保持电压的稳定性，消除电压波动，应该在输入与输出脚上分别接一个电容到地。在实际的应用过程中，不加散热片的 7805 与 7812 非常容易发热，过热一方面会导致芯片的损坏，也会使电压的输出出现波动。实际上，在本人的实践过程中已经出现了这方面的问题。

在上电复位电路方面，采用的是电阻电容串连充放电的方式，由于 SJA1000 的复位端脚是低电位有效的方式，而 80C552 的复位端则是高电位有效。所以在实际线路上略有不同，SJA1000 的复位回路中，电容接地，这样在上电时可以得到瞬间的低电平。而在 80C552 的复位回路中，电阻接地，在上电时可以得到瞬时的高电平。此外需要注意的是电阻电容的参数选择。由于复位信号的要求是持

续时间在 24 个时钟周期以上，在本系统中应用的是 11.0592M 的晶振，经计算得出，需求的复位电平应至少持续 2.2 微秒。根据这一原则，本系统中最终确定的复位电阻值为 10K，而电容则定为 2.2 微法。

针对微控制器，我还设置了按键复位，两个相关电阻分别为 1K 与 10K。经实验表明，这两种复位电路在实际的应用中都工作正常。

3.7. 系统的抗干扰与优化

3.7.1. 问题的发生及解决

在本系统的设计制作过程中，因为缺乏经验，曾经出现过包括设计在内的各种问题，包括：电源芯片 7805，7812 的管脚分布与 PROTEL 提供的三脚器件元件图不匹配；因为对元件研读不够深入而导致控制线路错误，例如 AT28C256 的 CE 引脚应该固定接在电源地上而不是参与选通；因为对单片机的时序研究不彻底而未对 P0 口添加地址锁存，从而导致程序不能正常地执行；板上的晶振因为放置位置与 MCU 的接入引脚，耦合电容都有一定的距离而造成晶振不工作；电源芯片散热考虑不周而导致芯片损坏等等。另外在程序调试的过程中也发现了一些设计上的新问题，例如在最初的设计中是将扩展 RAM 的地址定义为 0X0000－0X1FFF，但在调试μC/OS-II 的时候，由于仿真堆栈的引入，扩展 RAM 的地址必须改为 0XE000－0XFFFF，所以地址译码的线路就需要调整。主要的错误如晶振布置，地址锁存芯片的应用等问题都在重新制板的过程中得到了解决，一些后期发现的问题及调整则通过跳线的方式来解决。在发现和解决问题的过程中，本人对单片机系统及相关的芯片的理解也得到了很大的加深。

在实验室环境下开发一个满足要求的控制系统往往只是开发工作的一部分，在实验室环境（理想环境）下安装和调试好的样机在投入工业现场进行实际运行时，经常不能正常工作，时好时坏，究其原因，主要在于工业环境有强大的干扰，一般情况下，控制系统没有采取抗干扰措施或者措施不力。需要反复修改硬件设计和软件设计，增加一系列措施之后，系统才能够适应现场环境。往往为抗干扰而进行的工作比前期实验室的设计工作要多。

3.7.2 印刷电路板的防干扰设计

作为一个单片机系统，首先要考虑印刷电路板的设计状况。印刷电路板是电源线，信号线和元器件的高度结合体，它们在电气上会相互影响。诸多的实践证明，印刷线路的设计与制作对电子线路的抗干扰性能有很大的关系。即使同一个原理图，线路的设计与制作不同，其抗干扰性能有很大的区别。因此，印刷电路板的设计必须符合抗干扰的原则，以抑制大部分干扰，对软硬件的调试也非常重要。通常在设计印刷电路板的时候应该遵循以下几个原则：

{1}电源线布线原则

在电路板上，电源线的布线方法应注意三点：一是要根据电流的大小尽量加宽导线；二是电源线与地线的走向应同数据线的传递方向一致；三是印刷电路板的电源输入端应该接去耦电容，稳压电源最好单独做在一块电路板上。

{2}地线布线原则

通常，印刷电路板上的地线有数字地和模拟地两类。数字地是数字电路的地线，模拟地是模拟电路的地线。它们的布置也要遵循以下原则：一是数字地与模拟地要分开走，并分别与各自的电源地线相连；二是地线要加粗；三是接地线应注意构成闭合回路，以减少地线上的电位差，提高系统的抗干扰能力。

{3}信号线的分类走线

印刷电路板上的走线类型比较多，应该减少各类线间的相互干扰，例如交流线要和信号线分开布置，驱动线也要同信号线分开走线。

{4}去耦电容的配置

为了提高系统的综合抗干扰能力，印刷电路板上各关键部位都应改配置去耦电容。需要配置去耦电容的部位有：电路板的电源进线端；每块集成电路芯片的电源引脚到地，单片机的复位端到地。

此外还要根据元器件及走线的数量综合确定电路板的大小。

3.7.3 干扰源分析及相关对策

干扰可以沿各种线路侵入控制系统，也可以通过场的形式从空间侵入。供电线路是电网中各种浪涌电压入侵的主要途径，系统接地装置不良或不合理，也是引入干扰的重要途径。各类传感器，输入输出线路的绝缘效果不良，均有可能引

入干扰，以场的形式入侵的干扰主要发生在高电压，大电流，电磁场附近，它们可以通过静电感应，电磁干扰等方式在控制系统中形成干扰。

干扰对控制系统的作用可以分成三个部分：

第一个部分是输入系统，它使模拟信号失真，数字信号出错。控制系统根据这种输入信息作出的反应必然是错误的。

第二个部分是输出系统，它使各种模拟及数字信号混乱，不能正常反应控制系统的真实输出量，从而导致一系列的严重后果。

第三个部位是控制系统的内核，使三总线上数字信号错乱，CPU 得到错误的数据信息，使运算操作数失真，导致结果出错。并将这个错误一直传递下去，形成一系列错误。CPU 得到错误的地址信息后引起程序计数器 PC 出错，使程序运行离开正常轨道，导致程序失控，中断混乱，甚至可能使程序进入死循环，从而使系统完全瘫痪。迫使系统进行复位操作才能重新工作。

数字信号输入/输出中的抗干扰措施：

在单片机应用系统中，信号是作为脉冲信号在线路上传播的，为了确保信息在传播过程中的可靠性，主要通过以下的方法来预防干扰问题。

首先，在条件许可的情况下应该采用光电隔离，这样实现了一定的电气隔离，可以防止被控对象对单片机造成的危害。

对于数字信号的输入而言，因为数字信号主要来自各种开关型状态传感器，如限位开关和操作按钮等，它的干扰信号多呈毛刺状，作用时间短，对于这一特点，可以采用多次采集的方法，直到连续两次或多次的采集结果相同才视为有效。对于数字信号的输出而言，外部干扰可能使输出装置得到错误的数据，为此需要采取一定措施，最有效的方法就是重复输出同一个数据，而且重复周期要尽可能的短。总而言之，要根据不同的状况来选择不同的对策才能保证系统的稳定运行。

第四章:嵌入式逻辑控制系统的功能

4.1. 面向应用的逻辑控制功能的设计

本论文的一个重要的工作就是针对过程控制中所用到的逻辑控制命令编写相关的模块以便于 PLC 用户的调用，在此本人以欧姆龙 PLC 的语句表为参考编写了一些 C51 的函数供用户按照自身的需要来进行调用，在以后的章节里将对相关的函数作一些说明。

PLC 的指令一般的可以分为以下几个部分：位操作指令，数据传送指令，定时器与计数器指令，逻辑指令等。[29] 根据前文的分析，目前应用最广阔的是微型控制器。所以实际上大量的指令在工程中并没有得到应用。因此我根据既往的 PLC 编程经验，选取了一些常用的指令进行了 C51 函数的编写。PLC 是在继电器控制的基础上发展起来的，它的主要工作基本上是逻辑控制，所以我所选取的指令参照也以位逻辑控制的相关指令为主，包括：LD；LDNOT；OR；AND；ANDNOT；ORLD；DIFU；DIFD；KEEP；SET；RESET；TIM；CNT；OUT，END，INPUT 等。函数名基本上与 OMRON 的 PLC 相关指令同名，只是 PLC 指令中的操作数在本人的设计中变成了 C51 函数的实参。在 PLC 系统中每一次指令的扫描和执行对应于本系统就是各个相关函数的调用与执行。

4.1.1. 利用 C51 函数实现逻辑控制的总体思想

本系统实际上是 PLC 技术与单片机技术的结合，为适应这种结合，必然在某些方面有一些特殊的变化及特定的设置。

首先，需要特别指出的是在 PLC 编程中大量的操作数是 BIT 量，也就是 C51 中的位变量，但是在 51 单片机中可用的位变量只有 256 个，并且包括 128 个用于特殊寄存器的，在总量上是不能满足系统需求的，并且位变量在操作的过程中不能进行基于寻址的操作，这些都给系统编程带来了很多的不便。所以本人在设计系统时除了少量的位变量之外，大部分原该属于 BIT 型的变量定义为无符号整形变量(unsigned char)。

第二，在 PLC 的逻辑操作过程中经常会产生中继的逻辑结果，这个结果随着系统的运行在不断地改变，并且可能并没有作用于某一个数或变量，但是又直接对下一步的操作产生影响，因此我特别的为这个中间结果定义了一个变量

SIGNAL，SIGNAL 在初始化的时候定义为“1”。我将所有的位逻辑控制指令分成两大部分，一部分为输入型指令，一部分为输出型指令。一个完整的逻辑往往是由如 LD()之类的输入性指令开始，中间经过其它的逻辑指令后，由输出性指令如 OUT(),DIFD()作为结束。在刚开始处理如 LD()之类的输入性指令时 SIGNAL 应该为“1”。在中间过程中 SIGNAL 不断变化，而在输出性的函数调用即将结束的时候要将 SIGNAL 置为“1”。绝大多数的指令/函数都以 SIGNAL 为核心，首先对该值进行判断，然后再决定下一步的动作。

第三，定时器在逻辑控制中可能会有大量的应用。在 80C552 单片机中只有四个定时器，其中 T3 用于软件看门狗的定时，T1 用于在串口通讯中充当波特率发生器。在实际的逻辑控制程序中，定时器是远远超过两个的，而意图在片外扩展定时器电路来满足编程的需求则是不可思议的，所以我在系统中采用软件的方法在定时器 T0 的基础上设定了多个定时器，而且依照同样的方法可以任意将定时器的总数进行扩展。

第四，在系统中利用 8155 作为输入输出的扩展，按照 PLC 编程的习惯，将作为输入用的 8155 的各输入端口分别定义为 IX，“X”从 1 增加到 16。将作为输出用的变量定义为 OUTX，“X”从 1 增加到 14。之后通过 C51 语言函数的按位相或的操作，来对作为输出扩展的 8155 的对应端口进行赋值。这些是利用 C51 函数中绝对寻址的功能来实现的。

4.1.2. 应用于逻辑控制功能的函数

在本节里我将对所涉及的语句表指令及其相关的函数实现做简要的介绍。

TIM 指令为定时器指令，在逻辑控制的相关指令中最为重要。在本系统中的函数原型为 void TIMX(unsigned int X)，参数是针对 0.1 秒时基的计时值。当其输入或前方指令的中间结果保持为逻辑高时，TIM 进行持续的记时，若到达预定的时间，则其相关的标志位(我在函数实现中将其定义为 TIMXXX)将会被置“1”，当然此标志位在逻辑控制中会是一个重要的参考量。无论记时是否到达预定的时间，一旦输入或前方指令的中间结果变为逻辑低，则该定时器内的所有相关值全部清零，直到下次输入或前方指令的中间结果成为高电平再重新开始记时。本系统在定时器 T0 的基础上通过软件的方法来扩展定时器的数量。按照

OMRON PLC 中语句表的格式，定时器的命令为 TIMX，“X”的值随着定时器的使用而不断增加。所以我为每个定时器各自指定了不同的函数，命名为 TIMX(XX)。(形参就是定时器定时基数的倍数)。当然各个函数除了变量相关的数字不同，其余的全部相同。使用不同的定时器就是调用不同的函数。在硬件方面，本系统使用的是 11.0592M 的晶振，在此晶振下，T0 的溢出中断定时最大为 0.06 秒。所以本系统中，TH=0X4c,TL=0X00，在这种设置下，系统每 0.05 秒产生一次 T0 溢出中断，在中断服务程序中，针对各个定时器函数的具体情况做一些判断与基数的增加。定时器函数的相关变量基本上是全局变量，它们在初始化之后，它们值的变化是主函数与中断服务函数综合作用的结果。在主程序中，调用定时器函数主要就是根据输入或前方指令的中间结果的状态以及累积记时的值作一些综合判断，从而对标志位 TIMXXX 作出置“0”或置“1”的处理。进而对控制的进程产生影响。以 TIM1 为例，该函数实现的过程框图如图 4.1 所示。

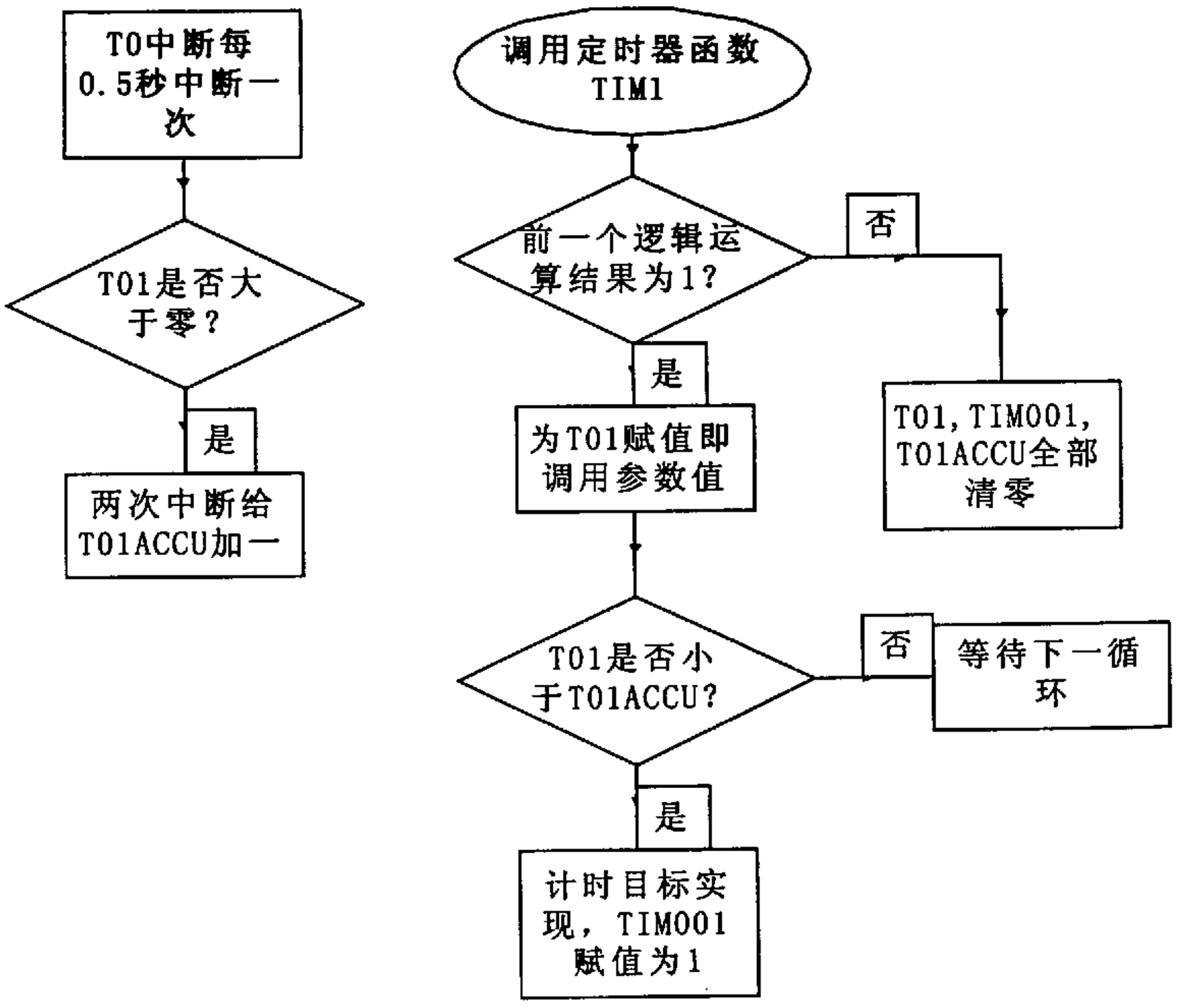
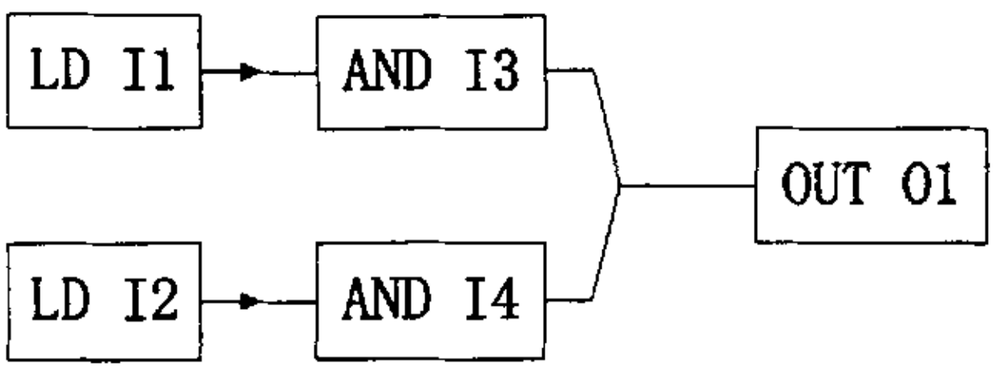


图 4.1 TIM1 的流程框图

AND 指令是通常所说的“与”命令，在 PLC 的梯形图指令形式中，它与 LD

的外形一致，只是串联在其后。函数原型为 Void AND(unsigned int X)。它将前一个中间结果与直接的操作数相与，得出它的逻辑结果。SIGNAL 的作用在此可以得到一定的显现。

LD 是位读取指令，函数原型为 Void LD(unsigned char X)。从功能上讲，在本指令执行时，系统读取操作数/实参，根据其逻辑值的高低将标志位(SIGNAL)置相应的值，如操作数/实参为“1”，则标志位为“1”，若操作数/实参为“0”，则标志位为“0”。标志位的值将直接影响下一步的操作结果。所以本函数只需对操作数/实参做出判断然后相应地对 SIGNAL 赋值即可。实际上，在一个逻辑中，LD()可能会多次出现，如下面所示的逻辑：



这个逻辑在逻辑控制中是最常见的，用指令表形式写出来就是：

```
LD I1;        AND I3;        LD I2;        AND I4;
ORLD;         OUT O1
```

ORLD 指令并不是一个独立的操作指令，它应用于在梯形图上先有串连再有并联的情况。函数原型为 Void ORLD()。在具体的实现上，我引入了与 LD 指令相关的 SIGNALPRE 变量用于存储第一个并联分支的中间逻辑结果。

在这个逻辑里面如果 LD()简单地读取操作数并对 SIGNAL 进行赋值必将导致逻辑的紊乱，因此还引入了表征是否是逻辑开始的变量 STARTID。根据 STARTID 的值不同，LD 的处理步骤也不同。该函数清单如下：

```
LD(unsigned char X)
{if(STARTID==0)
   {STARTID=1;
if(X!=0) SIGNAL=1;
if(X==0) SIGNAL=0;}
else
{if(SIGNAL==1)
```

```
{SIGNALPRE=1;}
if(SIGNAL==0)
{SIGNALPRE=0;}
if(X!=0) SIGNAL=1;
if(X==0) SIGNAL=0;}
}
```

LDNOT 也是位读取指令，函数原型为 Void LDNOT(unsigned char X)。其函数结构与 LD 相同，与 LD 不同的是，LDNOT 对标志位 SIGNAL 所置的结果恰与 LD 相反，既操作数/实参为“1”时，置 SIGNAL 为“0”。其对应函数也与 LD 的进行相反的操作。

OR 指令也就是通常所说的“或”命令，函数原型为 Void OR(unsigned char X)。它将前一个中间结果与直接的操作数相或，得出它的逻辑结果。

ANDNOT 指令是“与非”命令，函数原型为 Void ANDNOT(unsigned char X)。它将前一个中间结果与当前操作数的取反值相与得到该指令/函数的结果。它的操作对象也是作为中间结果的 SIGNAL 信号。

DIFU 指令是一个微分指令，在一个逻辑控制的实例中，它出现的次数并不多，但它往往在比较特殊的时候出现。函数原型为 Void DIFU(unsigned char* X)。其意义是当其前方所串联的指令的中间结果或输入值出现一个上升沿，即前一个扫描周期为逻辑低而当前扫描结果为逻辑高时，将对其操作数置“1”。这也是一个常用的位逻辑控制命令。在对它的函数实现中，第一次调用时先将 SIGNALPRE 变量初始化为“0”，在函数将结束时，再将当前的 SIGNAL 值赋给 SIGNALPRE。然后进行是否是上升沿的判断。此外，由于 DIFU 可能在程序中多次出现，仅靠上述方法必然会造成混乱，还另设了一个数组分别依次存放每个 DIFU 函数中的 SIGNALPRE，这样就可以避免上述问题的发生。函数清单如下所示：

```
DIFU(unsigned char* X)
{STARTID=0;
   DIFUB[DIFUID]=SIGNAL;
if((DIFUB[DIFUID]==1)&&(DIFUA[DIFUID]==0))
{*X=1;
```

```
}
if((DIFUB[DIFUID]==1)&&(DIFUA[DIFUID]==1))
{*X=0;
   }
if(DIFUB[DIFUID]==0)
{*X=0;}
   DIFUA[DIFUID]=DIFUB[DIFUID];
   DIFUID++;
   }
```

DIFD 指令也是微分指令，其功能与 DIFU 相似。函数原型为 Void DIFD(unsigned char* X)。它也要比较前方所串联的指令的中间结果或输入值的变化，与 DIFD 相反，若前一个扫描周期为逻辑高而当前扫描结果为逻辑低时，将对其直接操作数置“1”。在其函数实现上，采用与 DIFD 相似的方法，也设置了专门的数组 DIFDA[10]及 DIFDB[10]用于存放每次函数调用时的 SIGNALPRE。

SET 指令是置位指令，函数原型为 Void SET(unsigned char* X)。当指令的输入或前方指令的中间结果为逻辑高时，函数将对直接操作数进行置“1”的操作。该操作数一旦置“1”，其值不随输入或前方指令的中间结果的变化而变化，若要将其置“0”，则必须调用 RESET 指令来实现。

RESET 指令是与 SET 相对应的置位指令，函数原型为 Void RESET(unsigned char* X)。当指令的输入或前方指令的中间结果为逻辑高时，函数将对直接操作数进行置“0”的操作。该操作数一旦置“0”，其值不随输入或前方指令的中间结果的变化而变化，若要将其置回“1”，则必须调用 SET 指令来实现。

KEEP 指令是将 SET 与 RESET 相结合的指令，其作用就是两种指令的结合。在梯形图的形式上它有两个输入端，一个用于 SET 端，一个用于 RESET 端。在语句表的形式中，它分为两句，本人的函数实现与语句表一致，由 KEEP(unsigned char* X)与 RSTKEEP(unsigned char* X)两部分组成。在实际的应用中，它们应该成对出现，而且调用的参数都是 KEEP 指令的操作数，应保持一致。

CNT 指令为计数器指令，其主体函数原型为 void CNTX(unsigned int X)。参数是预定的计数值。当其输入或前方指令的中间结果出现一个上升沿时，CNT

指令就在内部进行加一操作。若上升沿总数到达预定的数量，则其相关的标志位(我在函数实现中将其定义为 CNTXXX)将会被置“1”，此标志位在逻辑控制中非常重要。在梯形图的形式中，CNT 是两端输入一端输出的形式。CNT 指令还有一个复位端的输入端，无论 CNT 目前的记数值达到多少，一旦复位端的输入成为高电平，则 CNT 指令/函数内相关的变量全部清“0”。所以函数 void RSTCNTX()必须与主体函数结合使用来共同实现一个完整的计数器指令。以 CNT1 为例，其程序清单如下所示：

```
    void CNT1(unsigned int X)
    {STARTID=0;
       if(CNT1WI==0)
       {CNT1WI=1;
       CNT1TEMP=0;
       CNT1N=X;
       CNT1PRE=0;
       CNT1NOW=SIGNAL;
       if((CNT1TEMP<CNT1N)&&(CNT1PRE==0)&&(CNT1NOW==1))
       {CNT1TEMP++;
       }
       }
  else
  { CNT1NOW=SIGNAL;
     if((CNT1TEMP<CNT1N)&&(CNT1PRE==0)&&(CNT1NOW==1))
      {CNT1TEMP++;
      }
       CNT1PRE=CNT1NOW;
  }
  if(CNT1TEMP>=CNT1N)
       {CNT1TEMP=CNT1N;
         CNT001=1;
```

```
        }
    }
```

OUT 为逻辑输出控制指令，在本系统中的函数原型为 OUT(unsigned int* X)。参数为即将操作的对象。在 PLC 中，它根据前方指令的中间结果向中间继电器或输出点输出相应的逻辑值，前方指令的中间结果为“1”时，它将操作数置为“1”。前方指令的中间结果为“0”时，将操作数置为“0”。在系统对它的函数实现上，主要也是控制中间变量和作为输出扩展的 8155 端口的逻辑值。

END 在 PLC 系统中作为程序编制的结束标志。其函数原型为 void END()。在本系统中，我编制了这个函数的作用不是结束的标志，而有一些特殊的作用。主要是将 OUTX 相对应的 8155 端口赋以相对应的逻辑电平。以 O1 为例，在 END() 中，

```
if (O1) PA8155B=PA8155B|0x01;
    if (O1==0) PA8155B=PA8155B&0xfe;
```

另外还要将 DIFUID 与 DIFDID 重新清零。

在 PLC 中，系统为用户提供秒脉冲供编程使用，我也编制了函数 SECPULSE() 为编程提供了秒脉冲。通过调用变量 SEC_PULSE 可以得到这个脉冲。

根据 PLC 的执行顺序，PLC 在执行控制程序之前，首先进行变量及程序的扫描，它会在开始阶段将所有的输入值及中间变量记录到内存，在本设计中，我也借鉴了这一思路，编写了针对输入信号的扫描程序，其函数原型为 void INPUT()。在主函数的起始阶段对其调用，将 8155 各个输入位存入从 I1 到 I16 中。以 I1 到 I8 为例其函数清单为：

```
        I8=PA8155A/128;
        K8=PA8155A%128;
        I7=K8/64;
        K7=K8%64;
        I6=K7/32;
        K6=K7%32;
        I5=K6/16;
        K5=K6%16;
        I4=K5/8;
```

```
K4=K5%8;
I3=K4/4;
K3=K4%4;
I2=K3/2;
K2=K3%2;
I1=K2;
```

4.2.嵌入式实时操作系统μC/OS-II 的特性

嵌入式系统在当今工业领域中非常普及，能够实现各种功能并且不断优化。商业实时内核的嵌入式系统一般都很昂贵。μC/OS-II 具有高度的灵活性，又是免费的，已经在各个领域取得了广泛的应用。本人的工作目标就是，将高效实时操作系统μC/OS-II 移植到 80C552 中去，使 80C552 能够在引入μC/OS-II 后正常运行。并且对于本人选定的控制对象及目的，进行任务的分解与创建，从而真正实现多任务的调度，并实现总体的目标。

4.2.1. μC/OS-II 实时操作系统的特点

μC/OS-II 是一个针对嵌入式系统而开发的实时操作系统，源代码全部公开，并且具有如下特点。

可移植性：只要在用户的应用程序中（用#define constants 语句）定义那些μC/OS-II 的功能是应用程序需要的就可以了。

占先式：μC/OS-II 完全是占先式的实时内核。这意味着μC/OS-II 总是运行就绪条件下优先级最高的任务。

多任务：μC/OS-II 可以管理 64 个任务，其中保留 8 个给系统。因此应用程序最多可以有 56 个任务。赋予每个任务的优先级必须是不同的。

可确定性：全部μC/OS-II 的函数调用与服务的执行时间具有其可确定性。也就是说，全部μC/OS-II 的函数调用与服务器的执行时间是可知的。进而言之，μC/OS 系统服务的执行时间不依赖于应用程序任务的多少。

任务栈：μC/OS-II 每个任务有自己单独的栈，及不同的栈空间。

系统服务：μC/OS-II 提供很多系统服务。例如邮箱、消息队列、信号量、块大小固定的内存的申请与释放、时间相关函数等。

中断管理：中断可以使正在执行的任务暂时挂起。如果优先级更高的任务被该中断唤醒，则更高优先级的任务在中断嵌套全部退出后立即执行。[1]

4.2.2. μC/OS-II 内核结构

任务的概念

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间. 一个任务通常是一个无限的循环。它有函数返回类型，有形式参数变量。为了使μC/OS-II 能管理用户任务，用户必须将任务的起始地址与其他参数一起传给下面两个函数中的一个：OSTaskCreat()或 OSTaskCreatExt()。选择这两个函数中的一个来建立任务。

任务的状态

图 4.2 是μC/OS-II控制下的任务状态转换图。在任一给定的时刻，任务的状态一定是休眠态(DORMANT)，就绪态(READY)、运行态(RUNNING)、挂起态(WAITING)(等待某一事件发生)和被中断态(ISR)。

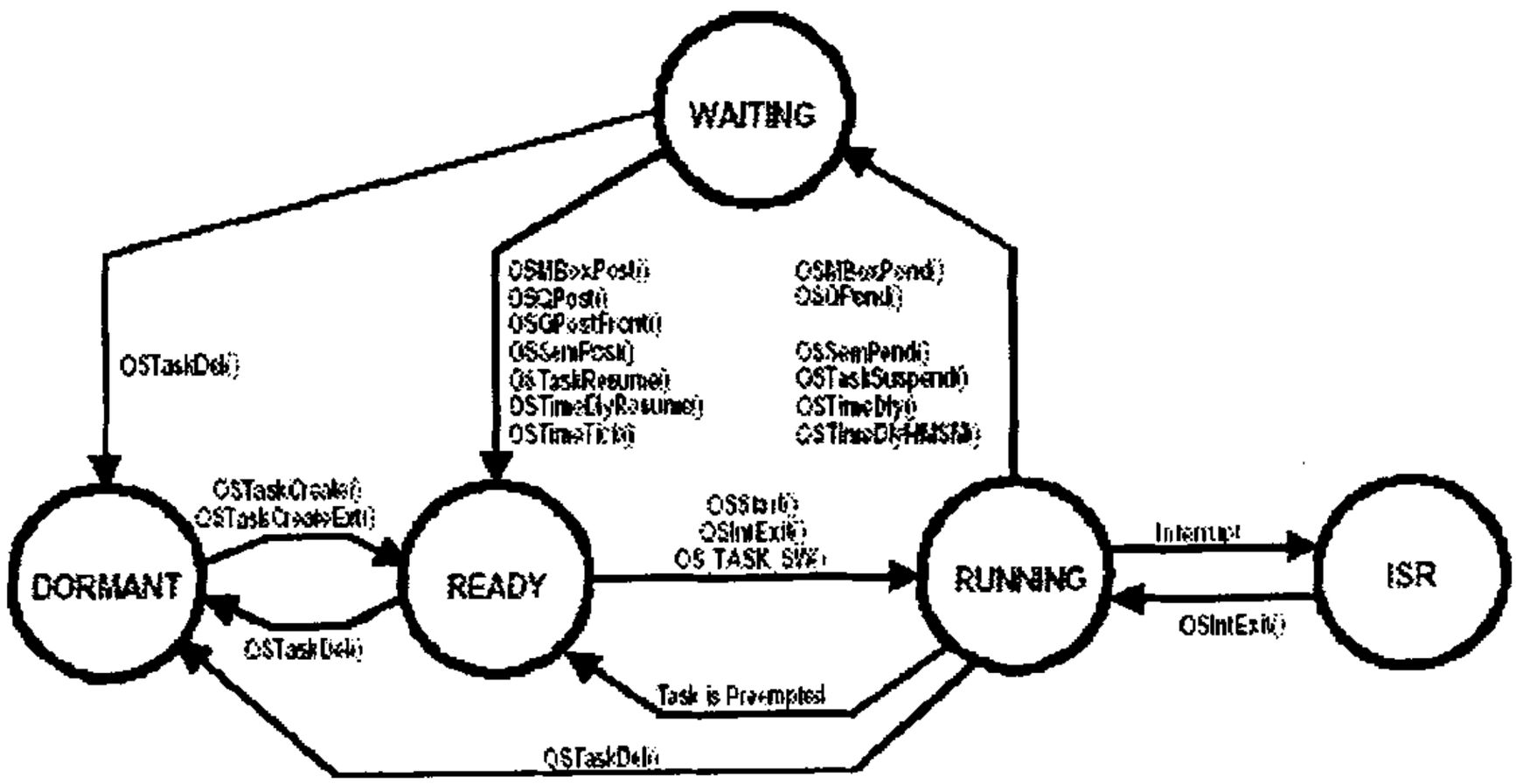


图 4.2μC/OS-II控制下的任务状态

休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的

优先级低，还暂时不能运行。运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中。挂起状态也可以叫做等待事件态 WAITING，指该任务在等待，等待某一事件的发生，（例如等待某外设的 I/O 操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。

调用 OSStart()可以启动多任务。OSStart()函数运行进入就绪态的优先级最高的任务。就绪的任务只有当所有优先级高于这个任务的任务转为等待状态，或者是被删除了，才能进入运行态。

正在运行的任务可以通过调用两个函数（OSTimeDly()或 OSTimeDlyHMSM()）之一将自身延迟一段时间，这个任务于是进入等待状态，等待这段时间过去，下一个优先级最高的、并进入了就绪态的任务立刻被赋予了 CPU 的控制权。等待的时间过去以后，系统服务函数 OSTimeTick()使延迟了的任务进入就绪态。

正在运行的任务期待某一事件的发生时也要等待，手段是调用以下 3 个函数之一：OSSemPend()，OSMboxPend()，或 OSQPend()。调用后任务进入了等待状态（WAITING）。当任务因等待事件被挂起（Pend），下一个优先级最高的任务立即得到了 CPU 的控制权。当事件发生了，被挂起的任务进入就绪态。

正在运行的任务是可以被中断的，被中断了的任务就进入了中断服务态（ISR）。响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权。中断服务子程序可能会使一个或多个任务进入就绪态。从中断服务子程序返回之前，μC/OS-II 要判定，被中断的任务是否还是就绪态任务中优先级最高的。如果中断服务子程序使一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务将得以运行，否则是中断了的任务继续运行。[1]

操作系统的核心：任务控制块

任务控制块是一个数据结构，它可以说是μC/OS-II 中最重要的核心，所有的任务操作都是要围绕它来进行的。当任务的 CPU 使用权被剥夺时，μC/OS-II 用它来保存该任务的状态。当任务重新得到 CPU 使用权时，任务控制块能确保任务从当时被中断的那一点丝毫不差地继续执行。OS_TCBs 全部驻留在 RAM 中。

应用程序中可以有的最多任务数（OS_MAX_TASKS）也是μC/OS-II 分配给用

户程序的最多任务控制块 OS_TCBs 的数目。将 OS_MAX_TASKS 的数目设置为用户应用程序实际需要的任务数可以减小 RAM 的需求量。所有的任务控制块 OS_TCBs 都是放在任务控制块列表数组 OSTCBTbl[]中的。[1]

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
    OS_EVENT        *OSTCBEventPtr;
    Void            *OSTCBMsg;
    INT16U           OSTCBDly;
    INT8U            OSTCBStat;
    INT8U            OSTCBPrio;
    INT8U            OSTCBX;
    INT8U            OSTCBY;
    INT8U            OSTCBBitX;
    INT8U            OSTCBBitY;
    BOOLEAN          OSTCBDelReq;
} OS_TCB;
```

.OSTCBStkPtr 是指向当前任务栈顶的指针。

.OSTCBNext 和.OSTCBPrev 用于任务控制块 OS_TCBs 的双重链接，该链表用于在时钟节拍函数 OSTimeTick()中刷新各个任务的任务延迟变量.OSTCBDly。

.OSTCBEventPtr 是指向事件控制块的指针。

.OSTCBMsg 是指向传给任务的消息的指针。

.OSTCBDly 当需要把任务延时若干时钟节拍时要用到这个变量，或者需要把任务挂起一段时间以等待某事件的发生，这种等待是有超时限制的。在这种情况下，这个变量保存的是任务允许等待事件发生的最多时钟节拍数。

.OSTCBStat 是任务的状态字。当.OSTCBStat 为 0，任务进入就绪态。

.OSTCBPrio 是任务优先级。高优先级任务的.OSTCBPrio 值小。也就是说，这个值越小，任务的优先级越高。

.OSTCBX, .OSTCBY, OSTCBBitX 和 .OSTCBBitY 用于加速任务进入就绪态的过程或进入等待事件发生状态的过程。这些值是在任务建立时算好的。

.OSTCBDelReq 是一个布尔量，用于表示该任务是否需要删除。

任务调度的关键：就绪表

每个任务被赋予不同的优先级等级，从 0 级到最低优先级 OS_LOWEST_PRIO，每个任务的就绪态标志都放在就绪表中，就绪表中有两个变量 OSRdyGrp 和 OSRdyTbl[]。在 OSRdyGrp 中，任务按优先级分组，8 个任务为一组。OSRdyGrp 中的每一位表示 8 组任务中每一组中是否有进入就绪态的任务。任务进入就绪态时，就绪表 OSRdyTbl[]中的相应元素的相应位也置位。就绪表 OSRdyTbl[]数组的大小取决于 OS_LOWEST_PRIO(见文件 OS_CFG.H)。当用户的应用程序中任务数目比较少时，减少 OS_LOWEST_PRIO 的值可以降低 μC/OS-II 对 RAM（数据空间）的需求量。如何处理优先级，OSRdyGrp 和 OSRdyTbl[]之间的关系牵涉到 μC/OS 进行任务调度的核心算法，要对该操作系统进行应用，应该掌握这个核心算法。关于这方面的内容我将在接下来的部分进行分析。

任务调度的实现方法

μC/OS-II 总是运行进入就绪态任务中优先级最高的那一个。当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态，并开始下一个任务的运行。这个过程叫做任务切换。为实现任务切换，OSTCBHighRdy 必须指向优先级最高的那个任务控制块 OS_TCB。

任务切换实际上很简单，由以下两步完成，将被挂起任务的微处理器寄存器推入堆栈，然后将较高优先级的任务的寄存器值从栈中恢复到寄存器中。在 μC/OS-II 中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有微处理器的寄存器都保存在栈中。换句话说，μC/OS-II 运行就绪态的任务所要做的一切，只是恢复所有的 CPU 寄存器并运行中断返回指令。为了做任务切换，运行 OS_TASK_SW(),人为模仿了一次中断。[1]

μC/OS-II 的中断处理

μC/OS-II 的中断服务子程序示意代码如下：

保存全部 CPU 寄存器；

调用 OSIntEnter 或 OSInetNesting 直接加 1；

执行用户代码做中断服务；

调用 OSIntExit()；

恢复所有 CPU 寄存器；

执行中断返回指令；

中断服务子程序先保存 CPU 寄存器，完后调用 OSIntEnter()或者给 OSIntNesting 直接加 1。接着用户中断服务代码开始执行。用户中断服务中做的事要尽可能地少，要把大部分工作留给任务去做。中断服务子程序调用相关函数通知某任务去执行，用户中断服务完成以后，调用 OSIntExit()来退出中断。

处理临界区代码的方法

代码的临界段也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。

和其他内核一样，μC/OS-II 为了处理临界区（critical section）代码需要关中断，处理完毕后再开中断。这使μC/OS-II 能够避免同时有其他任务或者中断进入临界区代码。μC/OS-II 定义两个宏（macro）来关中断和开中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()。

关于时钟节拍

μC/OS-II 需要用户提供周期性信号源，用于实现时间延时和确认超时。节拍率应在每秒 10 次到 100 次之间，或者说 10 到 100Hz。时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于用户应用程序的精度。时钟节拍源最好是专门的硬件定时器。

μC/OS-II 中的时钟节拍服务是通过在中断服务子程序中调用 OSTimeTick()实现的。OSTimetick()中量大的工作是给每个用户任务控制块 OS_TCB 中的时间延时项 OSTCBDly 减 1（如果该项不为零的话）。当某任务的任务控制块中的时间延时项 OSTCBDly 减到了零，这个任务就进入了就绪态。

4.2.3. 任务管理机制

任务管理包括建立任务、删除任务、改变任务的优先级、挂起和恢复任务，以及获得有关任务的信息。μC/OS-II 可以管理多达 64 个任务，并从中保留了四个最高优先级和四个最低优先级的任务供自己使用，所以用户可以使用的只有

56 个任务。任务的优先级越高，反映优先级的值越低。

建立任务

在开始多任务调度（即调用 OSStart()）前，用户必须建立至少一个任务。可以通过传递任务地址和其他参数到以下两个函数之一来建立任务：OSTaskCreate()或 OSTaskCreateExt()。任务可以在多任务调度开始前建立，也可以在其它任务的执行过程中被建立。任务不能由中断服务程序(ISR)来建立。一旦 OSTaskStkInit()函数完成了建立堆栈的任务，OSTaskCreate()就调用 OSTCBInit()，从空闲的 OS_TCB 池中获得并初始化一个 OS_TCB。

任务堆栈及其检验

每个任务都有自己的堆栈空间。用户可以静态分配堆栈空间（在编译的时候分配）也可以动态地分配堆栈空间（在运行地时候分配）。

有时候决定任务实际所需的堆栈空间大小是很有必要的。因为这样用户就可以避免为任务分配过多的堆栈空间，从而减少自己的应用程序代码所需的 RAM（内存）数量。μC/OS-II 提供的 OSTaskStkChk()函数可以为用户提供这种有价值的信息。

删除任务的请求及实现

有时候，如果任务 A 拥有内存缓冲区或信号量之类的资源，而任务 B 想删除该任务。这些资源就可能由于没有被释放而丢失。在这种情况下，用户可以想法让拥有这些资源的任务在使用完资源后，先释放资源，再删除自己。用户可以通过 OSTaskDelReq()函数来完成该功能。

删除任务，是说任务将返回并除以休眠状代，并不是说任务的代码被删除了，只是任务的代码不再被 μC/OS-II 调用。通过调用 OSTaskDel()就可以完成删除任务的功能。OSTaskDel()不允许在 ISR 例程中去试图删除一个任务，任务删除时，OS_TCB 就会从所有可能的μC/OS-II 的数据结构中移除。最后 OSTaskDel()置任务的.OSTCBStat 标志为 OS_STAT_RDY。并且从 TCB 链中解开 OS_TCB，并将 OS_TCB 返回到空闲 OS_TCB 表中。

挂起和恢复任务

有时候挂起任务是很有用的。挂起任务可通过调用 OSTaskSuspend()函数来完成。被挂起的任务只能通过调用 OSTaskResume()函数来恢复。

需要说明如果任务在被挂起的同时也在等待延时的期满，那么，挂起操作需要被取消，而任务继续等待延时期满，并转入就绪状态。

被挂起的任务只有通过调用 OSTaskResume()才能恢复。OSTaskResume()是通过清除 OSTCBStat 域中的 OS_STAT_SUSPEND 位来取消挂起的，要使任务处于就绪状态，OS_TCBDly 域必须为 0。只有当以上两个条件都满足的时候，任务才处于就绪状态。最后，任务调度程序会检查被恢复的任务拥有的优先级是否比调用本函数的任务的优先级高。如果高的话，将会有任务的切换调度。

μC/OS-II 要求用户提供定时中断来实现延时与超时控制等功能。这个定时中断叫做时钟节拍，时钟节拍的频率越高，系统的负荷就越重。

任务延时函数

对任务的延时可以通过 OSTimeDly()及 OSTimeDlyHMSM()来实现，而且 OSTimeDlyHMSM()是按时分秒的延时函数。

*μ*C/OS-II 提供了这样一个系统服务：申请该服务的任务可以延时一段时间，这段时间的长短是用时钟节拍的数目来确定的。调用任务延时函数 OSTimeDly()时，用户的应用程序是通过提供延时的时钟节拍数——一个 1 到 65535 之间的数，来调用该函数的。每隔一个时钟节拍就减少一个延时节拍数。调用该函数会使*μ*C/OS-Ⅱ进行一次任务调度，并且执行下一个优先级最高的就绪态任务。

用户也通过用小时、分、秒和毫秒指定延时来调用 OSTimeDlyHMSM()。实际上还是通过时钟节拍来实现延时，而节拍数是从指定的时间中计算出来的。

让处在延时期的任务结束延时

μC/OS-II 允许用户结束正处于延时期的任务而不必等待延时期满。这可以通过调用 OSTimeDlyResume()和指定要恢复的任务的优先级来取消延时，使指定任务处于就绪态。OSTimeDlyResume()会检验 OS_TCB 域中的 OSTCBDly 是否包含非 0 值以确定任务是否在等待延时期满。延时就可以通过强制命令 OSTCBDly 为 0 来取消。而且 OSTimeDlyResume()会调用任务调度程序来看被恢复的任务是否拥有比当前任务更高的优先级。这将会导致任务的切换。[1]

4.2.4. 任务之间的通讯与同步

在 *μ*C/OS-II 中，有多种方法可以保护任务之间的共享数据和提供任务之间

的通信。一是利用宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来关闭中断和打开中断。当两个任务或者一个任务和一个中断服务子程序共享某些数据时，可以采用这种方法。二是利用函数 OSSchedLock()和 OSSchedUnlock()对 μC/OS-II 中的任务调度函数上锁和开锁。另外三种用于数据共享和任务通信的方法是：信号量、邮箱和消息队列。

关于事件与 ECB

一个任务或者中断服务子程序可以通过事件控制块 ECB（Event Control Blocks）来向另外的任务发信号。这里，所有的信号都被看成是事件（Event）。一个任务还可以等待另一个任务或中断服务子程序给它发送信号。这里要注意的是，只有任务可以等待事件发生，中断服务子程序是不能这样做的。对于处于等待状态的任务，还可以给它指定一个最长等待时间，以此来防止因为等待的事件没有发生而无限期地等下去。多个任务可以同时等待同一个事件的发生。在这种情况下，当该事件发生后，所有等待该事件的任务中，优先级最高的任务得到了该事件并进入就绪状态，准备执行。

事件控制块 ECB 的数据结构

μC/OS-II 通过 μCOS_II.H 中定义的 OS_EVENT 数据结构来维护一个事件控制块的所有信息。也就是事件控制块 ECB。该结构中除了包含了事件本身的定义，如用于信号量的计数器，用于指向邮箱的指针，以及指向消息队列的指针数组等，还定义了等待该事件的所有任务的列表。[1]

ECB数据结构

```
Typedef struct {
    void    *OSEventPtr;              /* 指向消息或者消息队列的指针 */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE]; /* 等待任务列表        */
    INT16U   OSEventCnt;               /* 计数器(当事件是信号量时) */
    INT8U    OSEventType;                  /* 时间类型   */
    INT8U    OSEventGrp;                /* 等待任务所在的组   */
} OS_EVENT;
```

.OSEventPtr 指针，只有在所定义的事件是邮箱或者消息队列时才使用。

.OSEventTbl[] 和 .OSEventGrp 包含的是等待某事件的任务。

.OSEventCnt 当事件是一个信号量时，用于信号量的计数器。

.OSEventType 定义了事件的具体类型。它可以是信号量（OS_EVENT_SEM）、邮箱（OS_EVENT_TYPE_MBOX）或消息队列（OS_EVENT_TYPE_Q）中的一种。

每个等待事件发生的任务都被加入到该事件控制块中的等待任务列表中，该列表包括.OSEventGrp 和.OSEventTbl[]两个域。

等待任务列表的结构与相关的算法与就绪表的相关结构及算法相似。

在μC/OS-II 中，事件控制块的总数由用户所需要的信号量、邮箱和消息队列的总数决定。在调用 OSInit()时，（μC/OS-II 的初始化），所有事件控制块被链接成一个单向链表——空闲事件控制块链表。每当建立一个信号量、邮箱或者消息队列时，就从该链表中取出一个空闲事件控制块，并对它进行初始化。

对于事件控制块进行的一些通用操作包括：初始化一个事件控制块；使任务进入就绪态；使任务进入等待该事件的状态；因为等待超时而使任务进入就绪态。

μC/OS-II 将上面的操作用 4 个系统函数实现，它们是：OSEventWaitListInit()，OSEventTaskRdy()，OSEventWait()和 OSEventTO()

OSEventWaitListInit()用来初始化一个事件控制块。当建立一个信号量、邮箱或者消息队列时，相关的建立函数通过调用 OSEventWaitListInit()对事件控制块中的等待任务列表进行初始化。

OSEventTaskRdy()用来使一个任务进入就绪态，当发生了某个事件，该事件等待任务列表中的最高优先级任务要置于就绪态时，该事件对应的函数调用该函数实现此项操作。

OSEventTaskWait()使一个任务进入等待某事件发生状态，当某个任务要等待一个事件的发生时，相应事件的函数会调用该函数将当前任务从就绪任务表中删除，并放到相应事件的事件控制块的等待任务表中。

OSEventTO()应等待超时而将任务置为就绪态：当在预先指定的时间内任务等待的事件没有发生时，OSTimeTick()函数会因为等待超时而将任务的状态置为就绪。此时系统会调用 OSEventTO()来完成这项工作。它负责从事件控制块中的等待任务列表里将任务删除，并把它置为就绪态。

信号量机制

μC/OS-II 中的信号量由两部分组成：一个是信号量的计数值，它是一个 16 位的无符号整数；另一个是由等待该信号量的任务组成的等待任务表。μC/OS-II 提供了 5 个对信号量进行操作的函数。它们是：OSSemCreate()，OSSemPend()，OSSemPost()，OSSemAccept()和 OSSemQuery()函数。

建立信号量，也即调用函数 OSSemCreate()；该函数从空闲任务控制块链表中得到一个事件控制块，将该事件控制块设置成信号量。

等待一个信号量则需要调用 OSSemPend()；如果信号量当前是可用的，将信号量的计数值减 1，然后函数将“无错”错误代码返回给它的调用函数。如果此时信号量无效，则调用 OSSemPend()函数的任务要进入睡眠状态。

发送一个信号量，则是 OSSemPost()；如果事件控制块中的.OSEventGrp 域不是 0，说明有任务正在等待该信号量。这时，就要调用函数 OSEventTaskRdy()，把其中的最高优先级任务从等待任务列表中删除并使它进入就绪状态。

无等待地请求一个信号量，OSSemAccept()：当一个任务请求一个信号量时，如果该信号量暂时无效，也可以让该任务简单地返回，而不是进入睡眠等待状态。

OSSemQuery()查询一个信号量的当前状态。

邮箱机制

邮箱是μC/OS-II 中另一种通讯机制，它可以使一个任务或者中断服务子程序向另一个任务发送指向一个包含了特定“消息”的数据结构的指针。

μC/OS-II 提供了 5 种函数来对邮箱进行操作：

OSMboxCreate()建立一个邮箱，将事件控制块设置成 OS_EVENT_TYPE_MBOX。

OSMboxPend()等待一个邮箱中的消息，如果邮箱中没有可用的消息，OSMboxPend()的调用任务就被挂起，直到邮箱中有了消息或者等待超时；

OSMboxPost()发送一个消息到邮箱，如果有任务在等待该消息。就调用 OSEventTaskRdy()将使其中的最高优先级任务进入就绪状态。

OSMboxAccept()无等待地从邮箱中得到一个消息，中断服务子程序在试图得到一个消息时，应该使用 OSMboxAccept()函数，而不能使用 OSMboxPend()函数。

OSMboxQuery()函数使应用程序可以随时查询一个邮箱的当前状态。[1]

4.3. μC/OS-II 在逻辑控制系统中的应用

根据一段时间的反复研读，我认为如果要在自己的系统中将 μC/OS-II 应用起来，我们需要理解和解决的问题可以归结为三个，分别是：

（1）理解 μC/OS-II 工作原理；

（2）实现 μC/OS-II 的移植；

（3）用户程序如何嵌入到操作系统中运行。

4.3.1 μC/OS-II 工作原理的分析

μCOSII 工作核心原理是：近似地让最高优先级的就绪任务处于运行状态。

操作系统将在下面情况中进行任务调度：调用 API 函数(用户主动调用)；中断(系统占用的时间片中断 OsTimeTick(),用户使用的中断)。

在调用 API 函数时，如果系统 API 函数察觉到运行条件不满足，需要切换就调用 OSSched()调度函数，这个过程是系统自动完成的，用户没有参与。OSSched()判断是否切换，如果需要切换，则此函数调用 OS_TASK_SW()。这个函数模拟一次中断。好象程序被中断打断了，目的是为了任务切换。既然是中断，那么返回地址(即紧邻 OS_TASK_SW()的下一条汇编指令的 PC 地址)就被自动压入堆栈，接着在中断程序里保存 CPU 寄存器(PUSHALL)……。堆栈结构不是任意的，而是严格按照 μCOS-II 规范处理。OS 每次切换都会保存和恢复全部现场信息(POPALL)，然后用 RETI 回到任务断点继续执行。这个断点就是 OSSched()函数里的紧邻 OS_TASK_SW()的下一条汇编指令的 PC 地址。切换的整个过程就是，用户任务程序调用系统 API 函数，API 调用 OSSched()，OSSched()调用软中断 OS_TASK_SW()即 OSCtxSw，返回地址(PC 值)压栈，进入 OSCtxSw 中断处理子程序内部。反之，切换程序调用 RETI 返回紧邻 OS_TASK_SW()的下一条汇编指令的 PC 地址，进而返回 OSSched()下一句，再返回 API 下一句，即用户程序断点。因此，如果任务从运行到就绪再到运行，它是从调度前的断点处运行。

中断会引发条件变化，在退出前必须进行任务调度。μCOSII 要求中断的堆栈结构符合规范，以便正确协调中断退出和任务切换。前面已经提到任务切换实际是模拟一次中断事件，而在真正的中断里只要规定中断堆栈结构和 μCOSII 模拟的堆栈结构一样，就能保证在中断里进行正确的切换。任务切换发生在中断退出前，此时还没有返回中断断点。仔细观察中断程序和切换程序最后两句，它们

是一模一样的，POPALL+RETI。即要么直接从中断程序退出，返回断点；要么先保存现场到 TCB，等到恢复现场时再从切换函数返回原来的中断断点，由于中断和切换函数遵循共同的堆栈结构，所以退出操作相同，效果也相同。用户编写的中断子程序必须按照 μ COSII 规范书写。

4.3.2. 多任务调度算法分析

在前文我曾提到就绪表是任务调度的核心，任务调度究竟是如何实现的？图 4.3 就是 μ C/OS-II 的任务就绪表，根据此图我将对任务调度的具体算法做一个详细的分析。

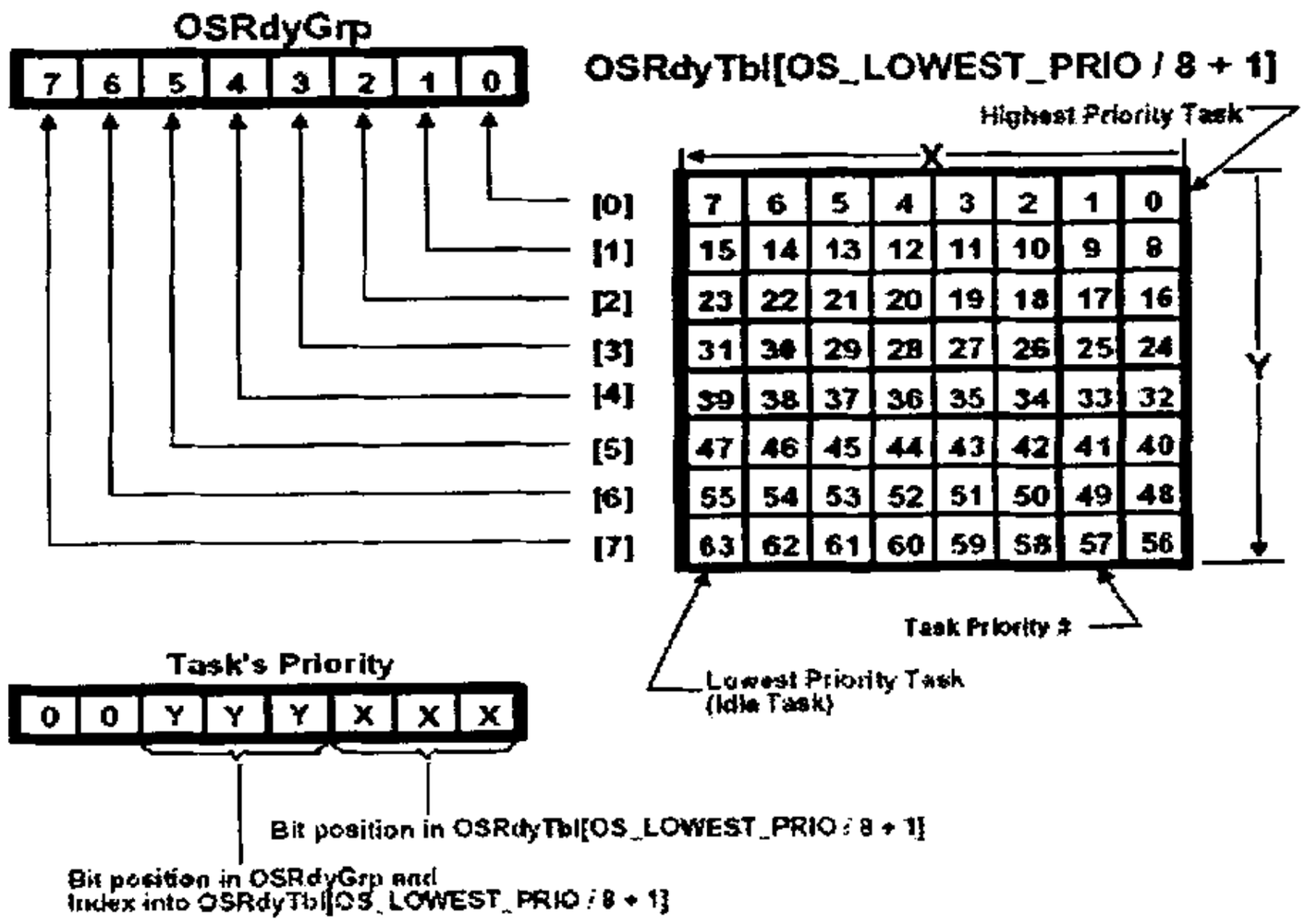


Figure 3-3, μ C/OS-II's Ready List

图 4.3 μ C/OS-II 的就绪表

我们可以看到，任务优先级的低三位用于确定任务在总就绪表 OSRdyTbl[] 中的所在位。接下去的三位用于确定是在 OSRdyTbl[]数组的第几个元素。OSRdyGrp 和 OSRdyTbl[]之间的关系是：当 OSRdyTbl[0]中的任何一位是 1 时，OSRdyGrp 的第 0 位置 1，当 OSRdyTbl[1]中的任何一位是 1 时，OSRdyGrp 的第 1 位置 1，以下依此类推直到 OSRdyTbl[7]与 OSRdyGrp 的第 7 位。

任务调度在就绪表中的相关操作共有 3 项，分别是将任务放入就绪表，从就绪表中删除任务，找出进入就绪态的优先级最高的任务。

1. 将任务放入就绪表，也就是使任务进入就绪态是通过如下两条语句：

```
OSRdyGrp|= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

OSMapTbl[]是μC/OS-II定义的一个查表函数，用于限制 OSRdyTbl[]数组的元素下标在0到7之间，其查表值定义为：当输入参数为0时，输出为00000001；当输入参数为1时，输出为00000010；以下类推至当输入参数为7时，输出为10000000。

2. 从就绪表中删除任务的实现代码为：

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

以上代码将就绪任务表数组 OSRdyTbl[]中相应元素的相应位清零，而对于OSRdyGrp，只有当被删除任务所在任务组中全组任务一个都没有进入就绪态时，才将相应位清零。

3. 找出进入就绪态的优先级最高的任务：

为了找到那个进入就绪态的优先级最高的任务，并不需要从OSRdyTbl[0]开始扫描整个就绪任务表，只需要查另外一张表，即优先级判定表OSUnMapTbl([256])。OSRdyTbl[]中的8位代表这一组的8个任务中的哪些进入就绪态了，低位的优先级高于高位。利用这个字节为下标来查OSUnMapTbl这张表，返回的字节就是该组任务中就绪态任务中优先级最高的那个任务所在的位置。这个返回值在0到7之间。确定进入就绪态的优先级最高的任务是用以下代码完成的。

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

例如，如果OSRdyGrp的值为二进制01101000，查OSUnMapTbl[OSRdyGrp]得到的值是3，它相应于OSRdyGrp中的第3位bit3，这里假设最右边的一位是第0位bit0。类似地，如果OSRdyTbl[3]的值是二进制11100100，则OSUnMapTbl[OSRdyTbc[3]]的值是2，即第2位。于是任务的优先级Prio就等于26（3*8+2）。利用这个优先级的值。查任务控制块优先级表 OSTCBPrioTbl[]，

得到指向相应任务的任务控制块 OS_TCB 的工作就完成了。

4.3.3. μC/OS-II 的移植与应用

如果用户理解了处理器和 C 编译器的技术细节，移植μC/OS-II 的工作实际上并不是特别困难。前提是所使用的处理器和编译器满足了μC/OS-II 的要求。移植工作包括以下几个内容：[1]

用#define 设置一个常量的值(OS_CPU.H)

声明 10 个数据类型(OS_CPU.H)

用#define 声明三个宏(OS_CPU.H)

用 C 语言编写六个函数(OS_CPU_C.C)

编写四个汇编语言函数(OS_CPU_A.ASM)

OS_CPU.H 的相关内容

OS_CPU.H 包括了用#defines 定义的与处理器相关的常量，宏和类型定义。

```
*                                   (与编译器相关)
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;       /* 无符号8位整数     */
typedef signed   char  INT8S;       /* 有符号8位整数     */
typedef unsigned int   INT16U;      /* 无符号16位整数    */
typedef signed   int   INT16S;      /* 有符号16位整数    */
typedef unsigned long  INT32U;      /* 无符号32位整数    */
typedef signed   long  INT32S;      /* 有符号32位整数    */
typedef float          FP32;        /* 单精度浮点数      */
typedef double         FP64;        /* 双精度浮点数      */
typedef unsigned int   OS_STK;      /* 堆栈入口宽度为8位 */
```

实际上这种类型定义可以说不是必须的，它只是帮助用户在进一步的开发时保持变量定义的一致性。

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

与所有的实时内核一样，μC/OS-II 需要先禁止中断再访问代码的临界段，并且在访问完毕后重新允许中断。

定义关中断的宏：#define OS_ENTER_CRITICAL()

定义开中断的宏：#define OS_EXIT_CRITICAL()

在此利用 EA=0 关中断；EA=1 开中断。这样定义在实现基本功能的情况下可以减少程序行数。

OS_STK_GROWTH

置 OS_STK_GROWTH 为 0 表示堆栈从下往上长。

置 OS_STK_GROWTH 为 1 表示堆栈从上往下长。

定义堆栈的增长方向：#define OS_STK_GROWTH

MCS-51 的堆栈是从下往上增长，所以 OS_STK_GROWTH 定义为 0。

OS_TASK_SW()

OS_TASK_SW()是一个宏，它是在μC/OS-II从低优先级任务切换到最高优先级任务时被调用的。OS_TASK_SW()总是在任务级代码中被调用的。另一个函数 OSIntExit()被用来在 ISR 使得更高优先级任务处于就绪状态时，执行任务切换功能。任务切换只是简单的将处理器寄存器保存到将被挂起的任务的堆栈中，并且将更高优先级的任务从堆栈中恢复出来。定义任务切换函数 OS_TASK_SW() 为 OSCtxSw()，OSCtxSw()要在 OS_CPU_A.ASM 中实现。

OS_CPU_C.C 的功能分析

在涉及具体的内容之前，首先讨论一下关于可重入的问题。可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数或者只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以保护。而对于不同的任务，将要使用的变量总数是不确定的，因此各个任务自身的栈空间大小应由用户来进行配置。就μC/OS-II操作系统本身的设计而言，是希望在其基础上进行开发的人员能够少用全局变量，尽量使用局部变量。TCB 结构体中 OSTCBStkPtr 总是指向用户堆栈最低地址，该地址空间内存放用户堆栈长度，其上空间存放系统堆栈映像，即：用户堆栈空间大小=系统堆栈空间大小+1。用户任务栈不能直接采用系统硬件堆栈的形式，图 4.4 是本系统中用户堆栈及系统硬件堆栈的示意图。

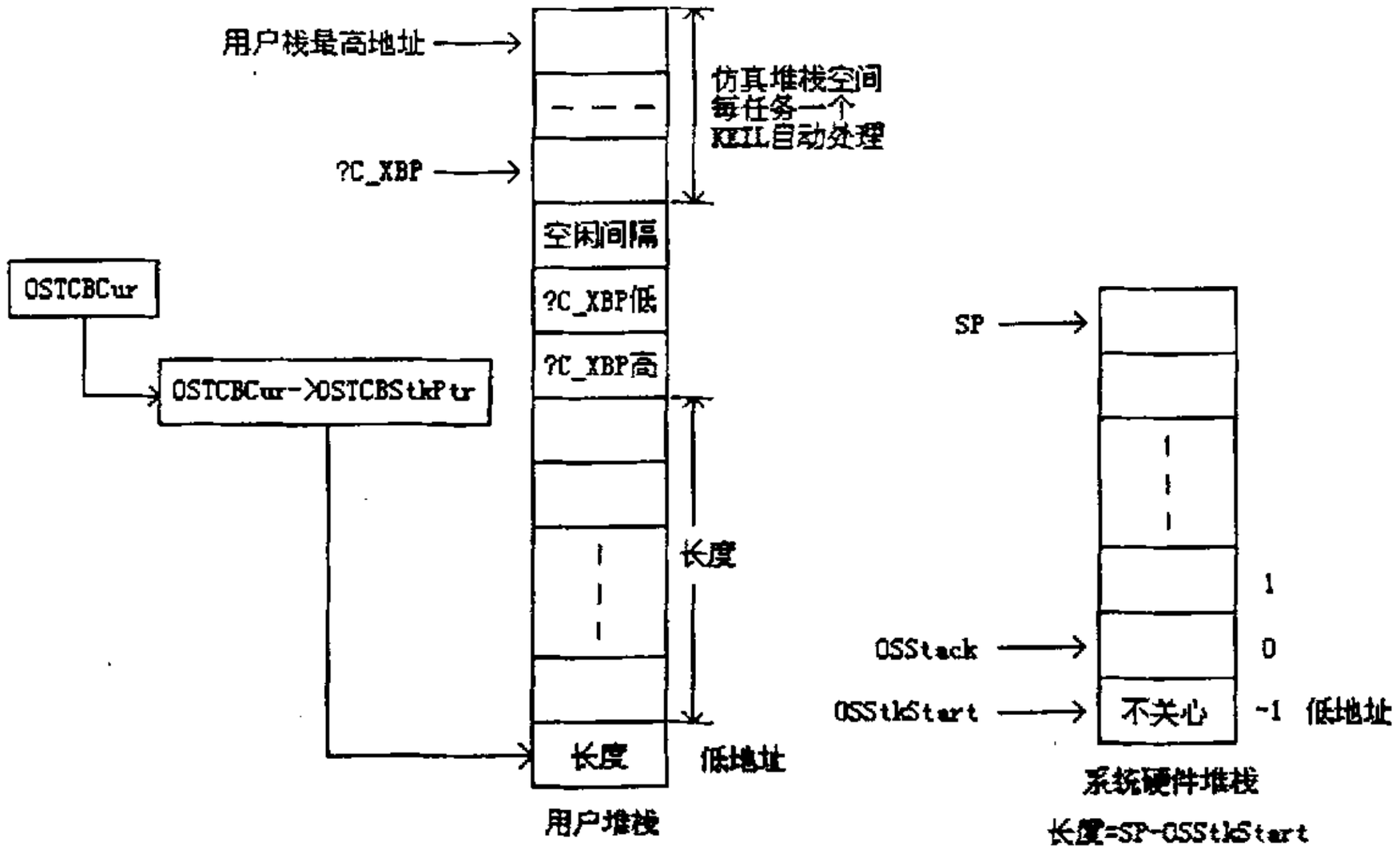


图 4.4 用户堆栈及系统硬件堆栈的示意图

在这里解释一下?C_XBP 的意思，由于 51 硬件堆栈太小，利用 KEIL 编译器时，KEIL 将在相应内存空间仿真堆栈。函数返回地址保存在硬件堆栈里，形参和局部变量放在仿真堆栈中，栈指针为?C_XBP，这些是由编译器自动完成的。[5]

OS_CPU_C.C 的主要任务就是进行用户堆栈的初始化(编写 OSTaskStkInit())以及几个钩子函数(在以下几种情况发生时调用，分别是任务创建；任务删除；执行任务切换；每秒钟被 uC/OS-II 统计任务调用；每一时钟节拍)的编写。若用户在上述几种情况下没有特别的工作，只需将空函数放置在里面就行了。至于堆栈初始化的工作，就是在得到用户堆栈最低有效地址后，采用指针依次递加的方式将用户的堆栈长度，任务地址低 8 位，高 8 位，寄存器 PSW,ACC,B,DPL,DPH,R0,R1,R2，R3,R4,R5,R6,R7 存放在连续的内存空间里，之后再将?C_XBP 的高 8 位与低 8 位存入。而用户堆栈最低有效地址作为函数 OSTaskStkInit()的返回值。

除此之外，操作系统 tick 时钟我使用了 51 单片机的 T0 定时器，它的初始化代码也写在了本文件中。在本人的系统中，我设定的时钟节拍为每秒 20 拍，根据相关的计算，T0 定时器的初始化设置为 TH=4C,TL=00。

任务切换的具体实现:OS_CPU_A.ASM

在确定了任务的堆栈形式及内容之后，OS_CPU_A.ASM 中则要对任务的切换

进行具体的实现。这里指的是相关的数据保存与恢复。

μC/OS-II的移植实例要求用户编写四个汇编语言函数，它们是：

OSStartHighRdy() ；OSCtxSw()；OSIntCtxSw() ；OSTickISR()

关于这4个函数的具体代码，这里就不具体列出了，在此作一个大致的说明。在OS_CPU_A.ASM中利用汇编函数写出上述几个函数的主要目的就是实现在任务切换时的相关堆栈操作。前面已经说明在用户进行相关的API函数调用及中断发生时要进行任务的切换，此时相关的任务数据必须得到正确地处理，这些就要通过对系统栈以及用户栈的操作来实现。

首先要做的是根据51单片机的堆栈特性定义相关的堆栈操作宏，POPALL和PUSHALL，分别对用户的寄存器PSW,ACC, B, DPL,DPH, R0,R1,R2,R3,R4,R5,R6,R7进行弹出及压栈操作。这两种操作在任务切换的过程中会多次出现。

任务的切换过程中，堆栈操作包括，首先将当前的任务下的相关寄存器的值进行压栈操作，然后将其拷贝到用户的任务堆栈中，接下来再将最高优先级的任务堆栈拷贝到系统堆栈中，再进行出栈操作。

由于51在进行数据入栈操作时，SP的内容总是先加1作为本次进栈的地址指针，然后再存入数据，因此，SP初始时指向系统堆栈起始地址(OSStack)减1处(OSStkStart)。因此系统堆栈存储空间大小=SP-OSStkStart。任务切换时，先保存当前任务堆栈内容，具体方法是：用SP-OSStkStart得出保存字节数，将其写入用户堆栈最低地址内，以用户堆栈最低地址为起址，以OSStkStart为系统堆栈起址，由系统栈向用户栈拷贝数据，循环SP-OSStkStart次，每次拷贝前先将各自栈指针增1。

其次，恢复最高优先级任务系统堆栈。方法是：获得最高优先级任务用户堆栈最低地址，从中取出“长度”，以最高优先级任务用户堆栈最低地址为起址，以OSStkStart为系统堆栈起址，由用户栈向系统栈拷贝数据，循环“长度”数值指示的次数，每次拷贝前先将各自栈指针增1。根据任务堆栈初始化的实际情况，用户栈的长度在此定义为15。

有一点需要注意的是在发生中断的时候，由于中断子程序的原因，例如发出了信号量，对某些任务实施了延时操作等，将使中断服务结束后，被中断的任务不再是最高优先级的任务。系统要求用户的中断服务子程序在结束时调用

OSIntExit()来实施退出，但该函数会发现有了更高优先级的任务，OSIntExit()会调用 OSIntCtxSw_in()来实施任务的切换，这些调用的过程中，会有一些返回地址的信息进入系统堆栈，而这些信息是用户保存上一个任务堆栈所不需要的，因此堆栈指针需要进行调整，在这里采用的是 SP=SP-4 的操作。这是在中断发生后的任务切换中所需要注意的地方。

OSTickISR()是一个重要的中断服务程序，以前提到了μC/OS-II的时钟节拍，时钟节拍的实现是借助于单片机的 T0 中断，所以每次中断都会进入 T0 中断服务子程序，在这里就是 OSTickISR()。它会去调用函数 OSTIMETick()，而 OSTIMETick()中有钩子函数 OSTimeTickHook()，本人利用软件方式扩展定时器就是在这个钩子函数内进行编程的。

4.3.4. μC/OS-II 在 80C552 上的应用示例

经过上面对μC/OS-II的分析及移植说明，我在这里举出一个μC/OS-II在 80C552 上的应用例子。以下是应用例子的简单说明。

在主函数中我建立了四个任务，建立任务的代码如下：

```
OSTaskCreate(TaskStartyy1, (void *)0, &TaskStartStkyy1[0],1);
OSTaskCreate(TaskStartyy2, (void *)0, &TaskStartStkyy2[0],2);
OSTaskCreate(TaskStartyy3, (void *)0, &TaskStartStkyy3[0],3);
OSTaskCreate(TaskStartyy4, (void *)0, &TaskStartStkyy4[0],4);
```

任务一的优先级为 1,它的工作是等待一个信号量,得到信号量后输出 TASK1 到上位机，然后延时 10 个节拍后再将信号量发出。同时处理一些简单的逻辑控制任务。任务二的优先级为 2，它等待信号量，在收到之后，输出 TASK2 到上位机，再将信号量发出。任务三的优先级为 3，它循环地发送字符“A”到“D”到邮箱之中，发送后就输出 TASK3 到上位机，并等待接收邮箱的消息，收到消息再继续发送。一个周期结束后将任务延时 5 个节拍。任务四的优先级为 4，它等待邮箱的消息，收到邮箱信号后输出 TASK4 到上位机，再发送信息到邮箱。各任务的代码如下：

```
void TaskStartyy1(void *yydata) reentrant
{yydata=yydata;
```

```
    for(;;){OSSemPend(MySem,0,&err);
   INPUT();
   LD(INN1);
   CNT1(5);
   LD(INN2);
   TIM1(2);
   TIM2(6);
   LD(CNT001);
   OUT(&02);
   LD(TIM001);
   LDNOT(TIM002);
   OUT(&01);
   if(INN1==1) {INN1=0;}
   else {INN1=1;}
   END();
   PrintStr("TASK1.\n");
     OSTimeDly(10);
      OSSemPost(MySem); }
     }
void TaskStartyy2(void *yydata) reentrant
{yydata=yydata;
    for(;;){
       OSSemPend(MySem,0,&err);
        PrintStr("TASK2.\n");
        OSSemPost(MySem);
       OSTimeDly(100);
    }
}
void TaskStartyy4(void *yydata) reentrant
```

```
{    unsigned int j;
     char *rxmsg=0;
     yydata=yydata;
     for(;;)
     {rxmsg=(char*)OSMboxPend(TxMbox,0,&err);
        PrintStr("TASK4.\n");
        for (j=0;j<12000;j++);
                OSMboxPost(AckMbox,(void *)1);
          }
}
void TaskStartyy3(void *yydata) reentrant
{  char txmsg='A';
     yydata=yydata;
     for(;;)
     {while(txmsg<='D')
       {
         OSMboxPost(TxMbox,(void *)&txmsg);
         PrintStr("TASK3.\n");
         for (j=0;j<12000;j++);
             OSMboxPend(AckMbox,0,&err);
         txmsg++;
       }
      OSTimeDly(5);
        txmsg='A';
     }
}
```

图 4.5 多任务调度的运行结果

在任务建立并运行之后借助串口调试软件我们得到了图 4.5 所示的结果。并且在施以一定的输入后在输出口上检测到了相关的结果。从图中我们可以看到，系统运行的结果和我们的期望结果是基本一致的。

4.4. 用于 CAN 总线通讯基本功能的函数

为了利用 SJA1000 实现的 CAN 基本功能，在具体的实现方面主要分为两个部分，首先是根据 SJA1000 的结构特点，建立一个专门定义其基本寄存器及功能位的头文件 SJA_BCANCONF.H。

根据相关的电路设计，SJA1000 的片选信号通过 74LS138 的第五个输出信号得到，所以定义 SJA1000 的基地址为 0X9000。在此基础上，分别定义了内部控制寄存器 CR(0X9000)，命令寄存器 CMR(0X9001)，状态寄存器 SR(0X9002)，中断寄存器 IR(0X9003)，验收代码寄存器 ACR(0X9004)，验收屏蔽寄存器 AMR(0X9005)，总线定时寄存器 0BTR0(0X9006)，总线定时寄存器 1BTR1(0X9007)，输出控制寄存器 OC(0X9008)，测试寄存器(0X9009)。接下来，从 0X900A 到 0X9013 定义了 10 个发送缓冲区寄存器，从 0X9014 到 0X901D 定义了 10 个接收缓冲区寄存器。

此外针对 SJA1000 在收发信息中出现的各种情况定义了若干命令字。

除了头文件，在 SJA_BCANFUNC.C 中编写了若干用于 CAN 基本功能的函数。以下对这些函数作一些具体介绍。

用于检测 CAN 控制器的接口是否正常的函数： bit BCAN_CREATE _ COMMUNATION(void) 该函数没有调用参数，其返回值为：0 表示 SJA1000 接口正常；1:表示 SJA1000 与处理器接口不正常。其主要步骤就是读写 SJA1000 的测试寄存器，

从而判断其接口是否正常。

使 CAN 控制器进入复位工作模式的函数：bit BCAN_ENTER_RETMODEL(void) 该函数没有调用参数，其返回值为:0:表示成功进入复位工作模式;1:表示不能进入复位工作模式。它通过对内部控制寄存器的复位请求位置“1”来实现。

使 CAN 控制器退出复位工作模式的函数：bit BCAN_QUIT_RETMODEL(void) 没有调用参数，其返回值为:0：表示成功退出复位工作模式；1：表示不能退出复位工作模式 。它通过对内部控制寄存器的复位请求位置“0”来实现。

设置 CAN 控制器 SJA1000 通讯波特率的函数:bit BCAN_SET_BANDRATE (unsigned .char CAN_ByteRate) 其调用参数为 CAN_ByteRate。返回值为：0:波特率设置成功；1:波特率设置失败。

根据 CAN 总线的相关要求，需要对 CAN 总线通讯设置不同的波特率，SJA1000 中总线定时寄存器有两个：总线定时寄存器 0 与总线定时寄存器 1，系统中所有的节点对于这两个总线定时器的设置必须为相同的值，否则系统可能无法通讯。这两个寄存器只能在复位模式下访问。

总线定时寄存器 0 定义了波特率预设值(BRP)与同步跳转宽度(SJW)的值。总线定时寄存器 0 的位分布为：

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0

波特率的预设值(BRP)为：32* BRP.5＋16* BRP.4＋8* BRP.3＋4* BRP.2＋2* BRP.1＋BRP.0

CAN 系统时钟为 $T_{scl}=2*T_{clk}*(BRP+1)$

T_{clk}=SJA1000 的晶振频率周期

总线定时寄存器 1 定义了每个位周期的长度，采样点的位置与每个采样点的采样数目。总线定时寄存器 1 的位分布为：

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

$T_{syncseg}=1*T_{scl}$

$T_{tseg1}=T_{scl}*(8*TSEG1.3+4*TSEG1.2+2*TSEG1.1+TSEG1.0+1)$

$T_{tseg2}=T_{scl}*(4*TSEG2.2+2*TSEG2.1+*TSEG2.0+1)$

1个位周期（Tbit）=（$T_{syncseg}$ +T_{tseg1} +T_{tseg2}）

波特率为=1/ Tbit

经计算得出了12MHZ晶振下的各个波特率所对应得BTR0与BTR1的初值。

CAN_ByteRate	波特率（Kbit/s）	BTR0	BTR1
0	5	0EFH	0FFH
1	10	0D7H	0FFH
2	20	0CBH	0FFH
3	40	0C5H	0FFH
4	50	0C9H	0A7H
5	80	0C2H	0FFH
6	100	04H	0A7H
7	200	02H	025H
8	250	01H	045H
9	300	01H	025H
10	500	01H	012H
11	600	00H	025H
12	1000	00H	012H

在实际应用中，先建立储存BTR0及BTR1值的数组，函数根据调用参数将数组中的对应值赋予总线定时寄存器0与总线定时寄存器1。

对CAN节点的通讯对象进行设置的函数:bit BCAN_SET_OBJECT(unsigned char ACR, unsigned char AMR) 本程序只能用于复位模式，其参数为:ACR:存放验收代码寄存器（ACR）的参数设置;AMR:存放接收屏蔽寄存器（AMR）的参数设置。返回值:0:通信对象设置成功; 1:通信对象设置失败。

允许接收的报文,是由AMR和ACR共同决定的。在验收滤波器的帮助下，CAN控制器能够允许RXFIFO只接受识别码和验收滤波器中的预设值相一致的信息。验收滤波器通过AMR和ACR来定义。能够被接收的信息必须满足如下条件：验收代码位(AC.7-AC.0)和信息识别码的高8位（ID.10-ID.3）相等，与验收屏蔽位(AM.7-AM.0)的相应位相或为1。用公式描述为：

[（ID.10-ID.3）=（AC.7-AC.0）]V（AM.7-AM.0）=11111111

调用本函数时，应根据接受报文的需求，正确设置验收代码寄存器及接收屏蔽寄存器的参数。

设置SJA1000的输出模式和时钟分频的函数：bit BCAN_SET_OUTCLK (unsigned char Out_Control, unsigned char Clock_Out) 其参数为：Out_Control:存放输出控制寄存器（OC）的参数设置；Clock_Out:存放时钟分频寄存器(CDR)的参数设置。返回值为：0：设置成功；1：设置失败。该子程序只能用于复位模式。

SJA1000 初始化的函数：bit BCAN_HW_INIT(unsigned char BCAN_ACR, unsigned char BCAN_AMR, unsigned char Bus_Timing0, unsigned char Bus_Timing1, unsigned char Out_Control, unsigned char Clock_Out)

其参数为：BCAN_ACR:存放验收代码寄存器（ACR）的参数设置

BCAN_AMR:存放接收屏蔽寄存器（AMR）的参数设置

Bus_Timing0:存放总线定时 0 寄存器(BTR0)的参数设置

Bus_Timing1:存放总线定时 1 寄存器(BTR1)的参数设置

Out_Control:存放输出控制寄存器（OC）的参数设置

Clock_Out:存放时钟分频寄存器（CDR）的参数设置

返回值为：0：表示初始化成功；1：表示初始化失败

说明:CAN 控制器的初始化只能在复位模式下才能完成。初始化操作之前要先进入到复位操作模式,并将要初始化的各参数的值按数据手册中的计算方法,正确的按实际需要计算好参数后，将参数写入对应的寄存器 。本系统中晶体频率 12MHZ，节点只接收 ID(标志符)高八位为'10101010'的消息，系统波特率为 50Kbit/s,关闭 CLKOUT 输出。

则根据数据手册中的计算方法的计算参数如下：

ACR==0AAH, AMR==00H, BTR0==0C9H, BTR1==0C9H, OC==0FFH, CDR=48H

则调用本函数为：BCAN_HW_INIT(0xaa, 0x00, 0xc9, 0xc9, 0xff, 0x48)

数据发送函数:bit BCAN_DATA_WRITE(unsigned char *SendDataBuf)

参数说明:特定帧各式的数据? 返回值：0：表示将数据成功的送至发送缓冲区 ；1：表示上一次的数据正在发送；2：表示发送缓冲区被锁定，不能写入数据；3：表示写入数据错误。本函数将待发送特定帧各式的数据，送入 SJA1000

发送缓存区中，然后启动 SJA1000 发送。本函数的返回值指示将数据正确写入 SJA1000 发送缓存区中与否。

CAN 总线帧格式为:开始的两个字节存放‘描述符’,以后的为数据内容。描述符包括 11 位长的 ID(标志符)\1 位 RTR\4 位描述数据长度的 DLC 共 16 位。

本函数的在执行时，先做几项判断：上一次的数据是否正在发送；发送缓冲区是否被锁定;写入数据是否有错误。当这几项判断都获得通过后统计帧的长度，并调用 memcpy 函数将参数指定地址内的数据拷贝到发送缓冲区。

接收数据的函数:bit BCAN_DATA_RECEIVE(unsigned char *RcvDataBuf)

其参数为:RcvDataBuf,存放微处理器保存数据缓冲区。返回值为:0：接收成功；1：接收失败。

函数先判断接受到的帧的类型,若为数据帧则计算报文中数据的个数后调用 memcpy 函数将接收缓冲区内的数据拷贝到参数指定地址。

SJA1000 命令运行函数 unsigned char BCAN_CMD_PRG(unsigned char cmd)

其参数为 cmd，是 sja1000 运行的命令字(在 SJA_BCANCONF.H 中定义)，分别为 01:发送请求；02：中止发送；04：释放接收缓冲区；08：清除超载状态；0x10：进入睡眠状态。函数返回值为:0：表示命令执行成功；1：表示命令执行失败。通过不同的参数进行函数调用可以使 SJA1000 进入不同的工作状态，从而控制 CAN 网络的通讯状况。

4.5. 软件抗干扰

在实际的运行过程中,正常执行的程序会因干扰而导致程序指针跳到不可预料的地址上，使总线上数字信号错乱，CPU 得到错误的数据信息，从而导致程序跑飞，为了防止这种情况的发生，除了在硬件上采取抗干扰措施外，在程序上也可以利用软件采取一定的措施来消除干扰后的影响。

WATCHDOG 有如下的特征：本身就独立工作，基本上不依赖于 CPU。CPU 在一个固定的时间间隔内和该系统打一次交道，以表明系统目前正常。当 CPU 掉入死循环中，可以及时发觉并使系统复位。

如果要真正达到 WATCHDOG 的真正目的，控制系统应该包括完全独立于 CPU 之外的硬件电路。有时为了简化硬件电路，也可以采用纯软件的 WATCHDOG 系统。

当硬件电路的设计未采用 WATCHDOG，则软件 WATCHDOG 是一个比较好的补救措施。在 80C552 中，系统就新增了定时器 T3，当 T3 出现溢出中断时，单片机自身就会进行复位操作。这为实现软件 WATCHDOG 提供了很好的资源。

第五章:总结与工作展望

5.1.本文开展的工作

本文针对那种只需要小型逻辑控制系统的场合，结合自身的工作经历，工业逻辑控制以及当前流行的嵌入式技术，提出了利用单片机来实现低成本嵌入式逻辑控制系统的方案。并进行了系统的电路及软件方面的设计。力图将新的嵌入式技术与已有的工业控制技术相结合。

本论文的主要工作包括以下几个方面：

1.提出了相对于 PLC 更加简化实用的低成本嵌入式逻辑控制器的总体方案。在分析了工业控制器的应用需求及 PLC 的体系结构之后，提出了结合传统逻辑控制以及嵌入式技术的嵌入式逻辑控制器的构思及方案。这一方案是对已有的 PLC 系统功能的一种精简，它更加适应实际的需求。

2.根据课题的具体设想，进行了逻辑控制系统电路的设计与相关芯片的选型，利用 PROTEL99SE 绘制了系统的电路原理图及印刷电路板图，并进行了电路板的制作。

3.根据 PLC 使用者的编程习惯，结合 PLC 编程中语句表的指令形式，以 OMRON PLC 语句表为指令格式参照，选取了常用的逻辑控制指令，编写了一些应用于逻辑控制的 C51 函数模块以及应用于逻辑控制系统的多个定时器及计数器。一方面，经试验证实通过这些函数可以实现逻辑控制的功能。从另一方面来说，实验室内实现的逻辑控制功能在工业现场是否能够正常工作仍有待于进一步的实践。

4.针对控制系统中多个任务同时执行的状况，引入了嵌入式实时操作系统 μC/OS-II 并进行了深入的研读，分析了该操作系统的工作原理及任务调度的核心算法。进行了 μC/OS-II 在 80C552 上的移植并进行了应用。对比于现在的 PLC 系统，利用嵌入式操作系统 μC/OS-II 来进行多任务调度可以简化电路，为未来将本设计应用于控制现场奠定了良好的基础。μC/OS-II 应用于 80C552 只是在仿真器条件下得到了完整的实现，而完全脱离仿真器应用于本电路时仍遇到了一定的问题。

5. 在系统电路中设置了 CAN 总线通讯所需的电路部分并且根据相关芯片的特性编制了一些用于 CAN 总线基本通讯功能的函数供扩展通讯时调用。但是在网

络通讯方面还有待将 CAN 总线通讯协议进一步的完善。

6. 结合以上的实践内容，做了一个模拟板试验了基本的逻辑控制及与上位机进行通讯显示相关变量信息的功能。

5.2. 后续工作展望

随着工业自动化技术的发展，控制器相关的功能也会不断的发展。作为一个面向应用的系统，本设计利用 C51 实现了常用的逻辑控制功能，并利用软件方法实现了多个定时器及计数器，实现了与上位机的串口通讯，进行了 μC/OS-II 在 80C552 上的移植并进行了应用。为了将本设计真正应用于工业现场，仍有许多需要进一步完善的工作。首先，为使其成为一个真正成熟的系统，在与上位机的通讯环节上应开发一个通用的操作界面，以完善上位机与本系统的变量信息传递及控制的功能。此外应对 CAN 总线予以完善，使之真正成为一个完善的控制网络。这些都将是本系统进一步发展的着重点。

参考文献

【1】：《μC/OS-Ⅱ—源码公开的实时嵌入式操作系统》JEAN J.LABROSSE著，邵贝贝译

【2】：《单片机原理及系统设计》胡汉才著，清华大学出版社

【3】：单片机的C语言应用程序设计（修订版）超星版

【4】：单片机开发与典型应用设计 超星版

【5】：Cx51编译器用户手册（C51.pdf） 网友jxlxh翻译

【6】：嵌入式系统及单片机国际学术交流会论文集，2001

【7】：《学习RTOS与使用RTOS》，邵贝贝，嵌入式系统及单片机国际学术交流会论文集，2001

【8】：《嵌入式系统设计与实例开发》，王田苗，清华大学出版社，2002

【9】：《嵌入式系统的硬件/软件协同设计》，姚放吾，微计算机信息，2001.3

【10】：《嵌入式系统编程源代码解析Programming for Embedded System》Dreamtech 软件开发组 著，王 勇 盖江南 阎文丽 等译，电子工业出版社，2002

【11】：单片机开发应用十例/ 李兰友，王勇才，傅景义主编电子工业出版社 1994

【12】：PC 机及单片机数据通信技术/ 李朝青编著 北京航空航天大学出版社 2000

【13】：单片机应用技术教程/ 张洪润，蓝清华编著

【14】：MCS-51 系列单片机实用接口技术/ 李华主编 北京航空航天大学出版社 1993

【15】：微型计算机原理 李广军...[等]编 电子科技大学出版社 2001

【16】：单片机接口技术 王修才，刘祖望编著 复旦大学出版社 1995

【17】：单片微机及其外围集成电路技术手册 郑子礼主编 光明日报出版社 1989

【18】：最新集成电路互换手册 楼铁军主编 江西科学技术出版社 1997

【19】：现场总线技术及其应用 阳宪惠主编 清华大学出版社 1999

【20】：PC 接口技术内幕 PC Ph.D.: inside PC interfacing/ Myke Predko 著

【21】：集成电路妙用巧用 300 例 陈有卿等编著 人民邮电出版社 1999

【22】：Protel 99 原理图与 PCB 设计 清源计算机工作室编著 机械工业出版社

2000

【23】: Protel PCB 99 设计与应用技巧/ 张义和编著 科学出版社 2000

【24】: 微型计算机数据传输基础与实践/（日）宫崎诚一著 人民邮电出版社 1990

【25】: 《嵌入式系统构件》，袁勤勇，机械工业出版社。2002

【26】: 《C Programming for Embedded System》，[加]Kirk Zurell著，艾克武 张剑波 艾克文 译， 机械工业出版社，2002

【27】: 《嵌入式系统设计》，Arnold Berger著，吕俊 译，电子工业出版社，2002

【28】: 《MC68332 单片机实时多任务操作系统的开发与应用》，常久鹏，卓斌，

【29】: 欧姆龙 C200HX/C200HG/C200HE 编程手册

【30】: www.c51bbs.com

【31】: www.hjhj.com

【32】: www.laogu.com

【33】: www.zlgmcu.com

【34】: 《Embedded Software》, Edward A. Lee, eal@eecs.berkeley.edu, 2001.11

【35】: Alexander Wolfe. Embedded ICs: Expanding the Possibilities. Website: http://embedded.com/2000/0011/0011feat6.htm

【36】: Zorner, W., Andreae, K.-H., Emshoff, H. and Muller, H., Diagnostics system for monitoring the operation of steam turbine generator sets. VGB Kraftwerkstechnik 72, Heft 6, 1992, pp487-496.

攻读硕士学位期间完成的和已发表的论文

韦奕，蒋式勤，软件方法实现欧姆龙 PLC 双机热备份，《微计算机信息》，2003 年 10 期

致谢

本文从选题到最后定稿自始至终都是在导师蒋式勤教授的严格要求和悉心指导下进行的。蒋老师为本文的完成倾注了大量的心血，她严谨治学的科学态度和渊博的知识将使我终身受益。在此对蒋式勤教授表示衷心的感谢！

感谢我的父母，他们将我抚育成人，并在我选择攻读研究生时给予了我巨大的物质支持与精神鼓励。感谢我的女友于淼，在我遭遇困境时给了我很大的抚慰与支持。

作者在本文的研究工作中，得到了从未谋面的师兄胡国四以及网友杨屹给予的有益的启发和许多帮助，作者在此向他们表示由衷的感谢。

作者在论文工作过程中，得到了同实验室的白连军、梁吉、凌空、查继来等师弟以及 00 级王磊同学的帮助和关心，在此表示衷心的感谢。另外，作者还要感谢在两年半的硕士研究生的生活中给我许多帮助和关心的学校领导、老师和同学。正是在所有的人的关爱之下，我才能顺利完成硕士阶段的课程学习及论文。