更新：

**01 -** 目录，更换其中‘**CAN**’字体，有宋体改为 **Arial** 字体

**02 - 37** 页，位选码已经根正

**03 - 56** 页，**sja1000.h** 头文件加入 宏定义声明

**04 - 63** 页，**Temp** 改为 **temp**

**05 - 83** 页， 更改了初始化流程图

**06 - 84** 页， 初始化函数源码时

**07 - 85** 页， 更改了发送流程图

**08 - 86** 页， 修改了源码的注释

**09 - 90~94** 页， 修改了大部分不明白的注释

**前言：**

创建该笔记的原因主要是为了总结自己学习 CAN，我承认笔记的内容记录得非常零散而且不全，不过内容已经足够让自己明白在学什么了，这本笔记来得有点匆忙，因为是在学期休假中编辑的，为期两个星期。我的性格是属于不爱说话的那种，所以在编辑笔记的时候不知不觉融入了自己的习惯，图文很多，借签内容也很多，估计读、浏览起来会有点吃力。笔记是已第五版为基础进行编辑的（也可以看成是第五版的不完全手册吧），里边也加了自己一些编写源码的习惯，写得不是很好，恰恰有参考的份量。

笔记适合那些喜欢使用 C 语言和有 STC89c5X 单片机编程基础的人，见笑了不会汇编语言（该学期才开始学汇编）。嗯，话说多了... 最后想感谢的人就是-EDN 社区里的娜娜小姐（黄娜），从国外购入学习板的时候麻烦了她很多很多，很感激她。

akuei2 上

目录

## CAN 总线活动讲座一：写在 CAN 总线开发板助学之前

大家好：

很感谢 cepark 以及 wangjin 给我们这个舞台来共同学习 **CAN 总线**。如今的电子技术日新月异，技术种类和方案层出不穷，需要电子工程师掌握的东西越来越多，你是否有点迷惑呢？在我们的“演出”正式开始之前，我想带大家简要回顾或是总结一下现在的这个领域。

电子这个领域其实很广阔，涉及的东西很多，而且我自己的能力也有限，不可能一言盖尽，所以就凭自己的粗浅认识来谈一谈，欢迎大家讨论。仅从应用角度讲，电子偏工程，所以诸如电子系、自动控制、测控、导航、机械等与电子密切相关的专业最后都被授予工学学位，与理学学位不一样。这就表示着这一大类学生的培养目标是直接面对社会工程应用领域，所以学习的课程很多也都是重工程，比如数电，模电以及单片机等。那么单就应用电子领域，我们作为一个学生应该掌握些什么呢？我们来看看

1. 基础电路知识：建立起电压，电流等基本电学概念，阻容感元件的内涵，电路基本定律。这是分析后续复杂电路的基础。

2. 模拟电路： 二极管、三极管等半导体器件的应用及性质，现在是 IC 流行的年代，已经没有人去用管子搭建复杂的电路，一是复杂，二是难调试，但是在很多场合，你就会发现一个二极管，一个三极管或是一个稳压管比 IC 是多么的有效和方便。 再有就是运算放大器的应用，这个是模拟电路（针对电子专业）的应用核心，运放的应用极为广泛：同反相放大、限幅电路、峰值电路、滤波器、IV、VI 变换、振荡器等很多关键的应用领域都有运放的身影。所以模拟电路大家一定要掌握。

3.数字电路：大家都感觉比模拟电路要简单，关键是一定要明白数字系统的电平，逻辑，传输门，时序等概念，为后续的数字高级系统打好基础。

4.单片机：单片机是目前最火的领域之一，可以说在各个重要的领域都有它，这个小小的芯片发挥着巨大的作用，它的魅力就在于“麻雀虽小，五脏俱全”，其实就是一个微型 CPU，在这个领域里，单片机的种类非常之多，差异也十分巨大，在不同的应用场合有不同的单片机来支撑，我们最为常用的就是51核的单片机，我会专门用一次讲座来说它。这里你就先建立一个感觉，那就是：单片机如果没学好，就意味着你的电子生涯不完整或者严重一点说不大合格。因此大家一定要学好单片机，它的开发其实也是最简单的，我们这回设计的 can 总线开发板，对于你来说，就一台电脑，一条串口线就够了。

5.DSP: 如果说单片机是事务驱动型的控制器，那么 DSP 就是运算驱动型，它的运算能力比单片机强很多，在涉及到雷达，声音，图像，以及高级算法实现方面，DSP 就显示出了极大的优势。它的基础说白了，就是数字信号处理，而数字信号处理的核心就是采样和数字滤波的设计，建立起一套完整的模数接口系统，这是应用的关键。

6.CPLD:cpld 偏重逻辑关系的实现，以往我们看一块 PCB，会发现很多的74**芯片，就是与非门之类的逻辑门，来处理系统中的逻辑关系，随着 GAL 等可编程逻辑阵列的出现，大大简化了设计和成本。而 CPLD 的出现更是革命性的。很多需要复杂译码的场合，尤其是 DSP 系统中，一般都会有 cpld 的出现，就是负责电平转换以及逻辑实现，既增强了系统的保密性，同时又加大了系统重组的灵活性。

7.FPGA：与 cpld 类似，但是如今的 FPGA 风头正盛，大有取代 DSP 以及单片机之势，它既具备 dsp 那种进行复杂运算的能力，同时又坚固单片机那种强大的控制能力，而且保密性和重组性又非常之强，所以越来越受到电子工程师们的重视。所以说现在的工程师如果不会 FPGA，那就明显落伍了。开发 cpld 以及 fpga 用的是硬件描述语言 VHDL 或者 Verilog，语言的描述能力很强大，足以覆盖整个信号处理领域。

8. 总线接口：这个就是我们要“表演”的，呵呵。总线非常之多，无处不在，电子也因为接口技术而精彩纷呈。看看我们周围的总线：串行 RS232，RS485总线，USB 总线，1394总线，并行接口，CAN 总线，TCP/IP 总线，以太网，单总线，IIC 总线，SPI 总线.....不胜枚举。可以说每一个 IC 器件都实际上是一种总线的体现。

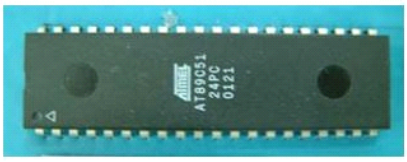
比如：大家手里的51学习板上的 DS1820温度传感器就是单总线的，USB 开发板就是 usb 接口的，AT24C02就是 IIC 的，AT93C46就是 spi 的，串口就是232的，以及我们要进行的 CAN 总线。这些都无处不在，可以说你的接口设计能力强基本就能代表你的水平了。设想一下，当你可以根据需要随便设计你的系统接口时，你的设计能力是多么高。比如开发一个仪器，觉得 USB 方便，那就直接 USB 接口，通过网络可以 internent 控制，那就以太接口，要个多点通讯的，选择 can，所以你会发现当你学会了单片机或是 FPGA 之后，总线以及接口的学习将会成为你的主要学习领域。

以上稍微讲了一些，写了现今应用广泛的几个大的领域，并不是很全面具体，但是终归给我们一个全貌。我们这次 CAN 总线助学就是针对目前应用最为广泛之一的 can 总线来和大家一起学习。在工程项目中，**大家经常采用 RS232，RS485通讯方式，是因为它们简单！但是当你学会了 CAN 总线后，就知道 CAN 总线比它们更简单！你会在以后的设计中毫不犹豫的采用 CAN 总线作为通讯方式！我们的口号是：大家一起来体会 CAN 总线的简单可靠吧！**

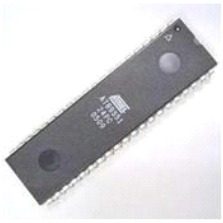
我想以讲座的形式来开展我们的活动，循序渐进的来带领大家逐步走入 CAN 总线的世界，同时也顺便讲一些单片机以及无线通讯等方面的开发。**在此之前，我们已经售出将近400套 can 总线的开发板，受到了很多网友的好评，所以请大家相信我们的质量！**我们目前已经将 CAN 开发板重新改进，增加了很多实用的功能。希望大家积极参加我们的这次助学活动，也希望通过这次活动，大家能够掌握 CAN 总线的设计！下一讲，我们先介绍一下我们这回 can 助学的讲座内容和开展的形式，欢迎大家随时提出自己的意见，并参与讨论！你的支持是我们 Cepark 前进的动力。还等什么？Just join us！

## CAN 学习讲座之二：回顾一下老朋友：单片机

首先呢。我们这个讲座开始总得有点介绍，一个课程开始都要有绪论啊，对吧？所以我们也不免这个俗，开始介绍一下，高手可以略过，初学的朋友们可以听听我们的唠叨，呵呵。 总体说来，我们这次的 can 总线学习板还是基于51单片机的学习系统，对于51我想大家再熟知不过了，几乎所有的大学中开设的课程都是以51单片机为基础来开设的，这里面的原因有很多，但是其中最最重要的一点就是51单片机的应用面广泛，而且群众基础好，架构清晰简明，容易学习，如果51能够非常熟练的应用的话，过度到其他单片机就非常容易了。在最后还会稍微介绍一下其他单片机，供大家参考，下图是到目前为止，最为普及和熟知的一些51系列的单片机，一个是以 ATMEL 公司生产的 AT 系列，还有一个就是最近异军突起的台湾宏晶公司出品的 STC 些列，我们这回开发板上用的就是这个。




AT89C系列



AT89S系列


STC89C系列


STC89C-PLCC封装系列

说到 AT 系列，勾起了多少人对往事的回忆，呵呵，记得我上大学的时候，实验室中的芯片就是 AT89C 系列，我还清晰的记得那时和几个好友参加电子竞赛的时候，用的就是这个，所以特别有“感情”，呵呵。AT89C 系列是一款低电压，高性能 CMOS 8位单片机，片内含可反复擦写的 Flash 只读程序存储器和随机存取数据存储器（RAM），兼容标准 MCS-51指令系统，片内置通用8位中央处理器和 Flash 存储单元，功能强大。先不说别的，很多初学者都不知道到底 AT89C 系列,AT89S 系列，以及 STC 系列有什么区别，呵呵，不说别的，就说一个最为大的区别，就是烧写方式的不同，如下：

- AT89C 系列，需要有专门的烧写器，当你在 IDE(集成开发环境)，开发51最为常用的就是大名顶顶 KEIL，编译通过后，会生成一个.HEX 或是.bin 的文件，这个就是下载文件，然后你要用专门的烧写器去把程序烧进去，因此这就属于离线烧写，频发的插拔器件，很不方便。目前，这位可敬“老先生”已经退休停产了。
- AT89S 些列，真是因为上面提到的 AT89C 系列的离线烧写很不方便，所以 ATMEL 公司顺应“民声”，推出了 S 系列，S 系列可以应用 ISP（在线下载）技术对芯片进行烧写，这就跨出了一大步，可以免去离线操作的麻烦，但是必须要配一个专门 ISP 电路，和烧写端口，也是麻烦，不过在那个时候这项技术一推出，有多少人为之感激涕零啊。
- STC 系列，这个由台湾宏晶生产的与 AT 系列完全兼容的小家伙，更神奇，它是通过内嵌一段代码来实现通过串口来进行下载，也就是说，isp 的电路我也不要了，你只要有串口，就 ok！那现在的单片机系统，一般都有串口啊，就算不用，一般的工程师们也愿意引出来，留着扩展或是与其他系统通信。所以串口就不但可以通信而且可以下载，这样现在对于一般的在校学生来讲，就不必去花钱买烧写器，直接连上 PC 的串口就可以自由下载自己编的程序了。是不是很爽呢？呵呵。

以上饶舌讲了讲最常用的51系列的单片机，其实目前单片机的家族庞大的很，各式各样的产品曾出不穷，

让人们真是不知道选何种为好，因此碰到一个项目，单片机的选型就是首先面临的一个问题。有经验的工程师会选的很好，给开发带来方便，下面呢，我就简单介绍几个系列的 MCU，大家可以参考一下，进一步感兴趣的呢，可以自己去 baidu 或 google 一下，这年头，大家要学会自己去找资料学习，可别总出现“跪求”，“在线等”等字眼。

- MSP430系列，这款单片机实力不能小觑！是由 TI 公司出品，具备 JTAG 功能，片上外设十分丰富！而且最最有特色的就是低功耗，因此常用在各种便携式的仪器仪表中，现在势头很猛。
- AVR 系列，ATMEL 出品，国内大名鼎鼎的“our avr”社区谁人不知？谁人不晓啊？站长阿莫是一个非常热心的人，深受广大网友的信任！因此 AVR 在中国蓬勃发展。它是增强型 RISC，内载 Flash 的单片机,芯片上的 Flash 存储器附在用户的产品中,可随时编程,再编程,使用户的产品设计容易,更新换代方便.AVR 单片机采用增强的 RISC 结构,使其具有高速处理能力,在一个时钟周期内可执行复杂的指令,每 MHz 可实现1MIPS 的处理能力.AVR 单片机工作电压为2.7~6.0V,可以实现耗电最优化.AVR 的单片机广泛应用于计算机外部设备,工业实时控制,仪器仪表,通讯设备,家用电器,宇航设备等各个领域。
- Motorola 单片机: Motorola 是世界上最大的单片机厂商.从 M6800开始,开发了广泛的品种,4位,8位,16位32位的单片机都能生产,其中典型的代表有:8位机 M6805,M68HC05系列,8位增强 M68HC11,M68HC12 , 16位机 M68HC16, 32位机 M683XX. Motorola 单片机的特点之一是在同样的速度下所用的时钟频率较 Intel 类单片机低得多,因而使得高频噪声低,抗干扰能力强,更适合于工控领域及恶劣的环境
- MicroChip 单片机: MicroChip 单片机的主要产品是 PIC16C 系列和17C 系列8位单片机,CPU 采用 RISC 结构,分别仅有33,35,58条指令,采用 Harvard 双总线结构,运行速度快,低工作电压,低功耗,较大的输入输出直接驱动能力,价格低,一次性编程,小体积. 适用于用量大,档次低,价格敏感的产品.在办公自动化设备,消费电子产品,电讯通信,智能仪器仪表,汽车电子,金融电子,工业控制不同领域都有广泛的应用,PIC 系列单片机在世界单片机市场份额排名中逐年提高.发展非常迅速.。
- 华邦单片机（Winbond）: 华邦公司的 W77,W78系列8位单片机的脚位和指令集与8051兼容, 但每个指令周期只需要4个时钟周期,速度提高了三倍,工作频率最高可达 40MHz.同时增加了 WatchDog Timer,6组外部中断源,2组 UART,2组 Data pointer 及 Wait state control pin. W741系列的4位单片机带液晶驱动,在线烧录,保密性高,低操作电压(1.2V~1.8V).
- C8051Fxxx 系列:这个系列的 MCU 是完全集成的混合信号系统级芯片，具有与8051 兼容的微控制器 内核，与 MCS-51 指令集完全兼容。除了具有标准8052 的数字外设部件之外，片内还集成了 数据采集和控制系统中常用的模拟部件和其它数字外设及功能部件，功能超级强大！很适合于数字信号处理需求强的应用领域。
- ADUC 系列：ADI 公司出品，该单片机具有高速高精度的 ADC、DAC 功能，以及独一无二的在电路可调试、可下载的特点，特别适合在各种测控系统和仪器仪表中使用。ADuC841也是目前最容易掌握、开发和应用的单片机之一。

好了，通过以上对单片机的一个简介，主要只想让大家有个总体的认识，如果说51助学是一个初学者的入门的话，那么通过 USB 助学以及 CAN 总线助学的学习之后，就应该是一个中高级开发者了（当然不是立刻就是，呵呵），因此技术知识要有一个总体的把握，不能一窝蜂似的，流行什么就学什么，自己对于目前技术路线都不清楚，那做多少年都是无用，不能有更高层次的进步。看了以上的单片机介绍，有的人会问，你介绍的这些单片机的功能比51强太多了，那我们是不是放弃51去学习这些啊，答案当然不是这样，一位老工程师和我说过，一个人用最最简单的器件如果能搭出像样的产品，那么这个产品最最可靠，而这个人也水平最高。呵呵，这个话就说明，尽管51出现的很早了，但是不不意味着它的过时，仍然有巨大的舞台，希望大家能够把单片机学习的更加好，去开发出更多更好的产品！ 下一个讲座，我们将说说51单片机系统的目前一些应用，敬请大家关注，大家有什么好的提议，可以在我们小组中提出来！

## CAN 总线活动讲座三：51单片机系统的组成

前面已经讲了关于51单片机的一些初步，我想大家肯定通过 wangjin 老师的51助学活动学到了51单片机的使用，那么这里我们来总结一下，看一下51单片机系统的构成是如何的，如果学会了这讲，我想大家以后对单片机系统的构建心中就有数了。 简单来说：只要是基于单片机的系统都可以化为如下的结构： **电源部分+单片机最小系统+外围功能部件+存储部分**

电源部分是重中之重，系统需要能源，能源都来自电源，电源可以是直流电源也可以是电池，当然对于便携式设备来说，电池就是最最主要的，比如 MP3了，电子手表等无不需要电池。电源部分不但要稳定，而且功耗也要够，这才能满足系统的需要。

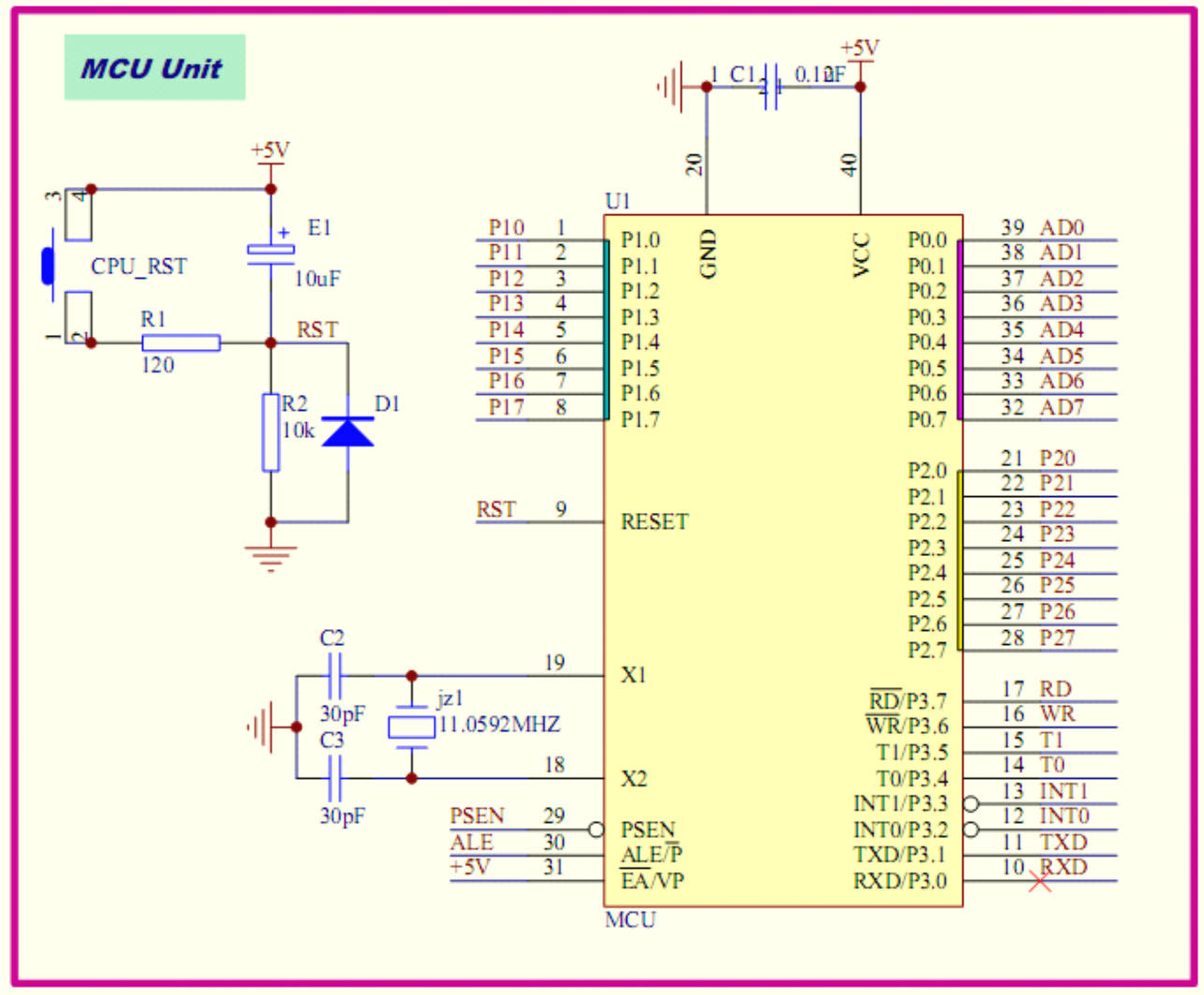
单片机最小系统，顾名思义就是单片机所构成的最小的能工作的电路，拿51单片机为例，最小系统就包括：单片机+复位电路+晶振，就基本够了，因此单片机最小系统简单但是也复杂，简单是因为电路不复杂，复杂是说要想稳定工作需要做很多电路上的处理。

外围功能电路，就是你要做什么，就用什么，比如我们的 can 总线系统，就是 can 通信器件，圈圈的 USB 系统，就是 D12构成的通信器件，采集温度，传感器系统就是功能电路，那么给大家总结一下，目前市面上的许多以51为主的开发板，大致有如下几种： 51+can 51+TCP/IP 以太网 51+无线 51+射频 51+IC 卡 51+USB 51+AD/DA 51+语音 51+cpld 51+传感器 51+其他总线 51+红外 51+电机控制 以上这些基本涵盖了51学习的所有部分，欢迎补充，呵呵，可以说只要51使用熟练了，那么以上几个部分再能熟练应用的话，单片机的应用水平就可以达到个高手的水平了。

所以我们这个 can 总线的开发板，就属于51+can，所以我们的思路就是，先按结构把我们这个板子的电路给大家剖析一下，把每一部分高清之后，再学习 can 总线的通信部分，这样学习就快了。 那么下一讲，我们将介绍一下我们即将给大家推出的 can 总线学习版的硬件电路部分。敬请期待！

## CAN 总线活动讲座四：CAN 开发板硬件-MCU 部分

首先先介绍 STC 单片机最小部分，我们给大家选的器件是 STC89C52，与 AT89S52完全兼容，最小系统部分的电路如下：

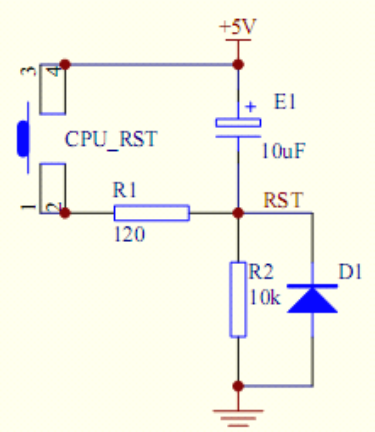



51最小系统就如前面所说，包括复位电路和晶振部分。复位电路是一般的芯片都需要的，目的简单的说就是初始化芯片，让芯片从新开始工作，具体表现就是，一旦复位，单片机就从程序 ROM 中起始位置开始读程序代码，一旦死机或是程序跑飞，就得重新复位。 我们这里的电路采用自动复位+手动复位的方式，系统上电，自动复位，复位采用简单的阻容方式，复位时间，大家可以自己通过 RC 参数计算。手动复位就是通过 CPU—RST 按键来进行，其实 STC 单片机系列内部有自动复位功能，但是为了可靠，加上也无妨，这样还有一个大的好处，就是如果换成了 AT89C 和 AT89S 系列单片机，照样可以正常工作！

晶振部分，确切的说应该叫晶体，配合2个30p 的电容实现振荡，产生电脉冲为单片机提供时钟信号，有的人问为什么选择11.0592M 呢？这个主要是从单片机的串口通信部分考虑，按照这个晶振可以没有误差的计算出通信初始值，我想这部分对于有过51开发经验的人都会明白我讲的是什么吧？如果你不知道，赶快去51助学小组去找 wangjin 老师补课哦，呵呵。 再进一步说，即使不用晶振，你也可以从外部产生一个脉冲信号直接送入 X1端也照样工作。这里注意的是31引脚我们直接接高电平，目的是让单片机上电后自动读取内部 ROM 中的程序。那么如果我接低电平会如何呢？大家可以思考一下

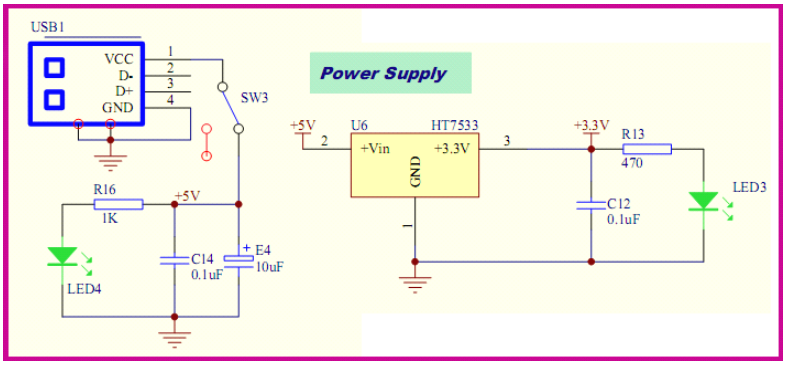
## CAN 总线活动讲座五：CAN 开发板硬件-按键和电源部分

这一讲我们讲 can 总线开发板的自由按键和电源部分。自由按键很简单，如下：




这个按键的电路很典型，按键的操作其实主要是考虑消抖，对于那种 4*4的按键来说，按键识别程序固然重要，但是对于按键的消抖处理也一样重要，这里我们板子上的按键是连接了 MCU 的中断接口，这样就可以直接以按键信号来给出中断信号，阻容的作用就是硬件消抖，也是最为经济实用的电路，当然目前已经出了一些专用消抖的芯片，但是成本无疑是增加的。

电源部分，我们采用的 USB 取电，这样做的目的是便于大家学习，因为几乎开发单片机的人都有电脑吧？前一讲我们讲过，电源简单来说，一个是考虑驱动能力，一个是考虑稳定性，鉴于手中有高精度的稳压源的人毕竟是少数，而电脑的 USB 口则是“天生”一个好直流的5v 电源，并且能提供近500mA 的电流，因此驱动我们的 can 总线学习板是足够了，电路如下：

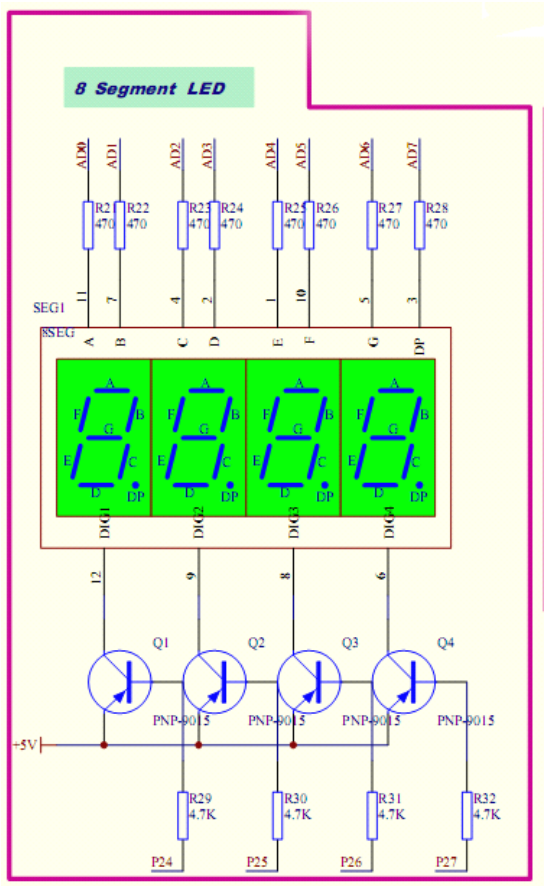



LED5灯是用来指示电源的，如果 USB 电源是5V，准确输出，那么 LED5灯就会亮。

这里有个小小的细节，那就是，有的同学会不会问，R15的阻值怎么取？这个问题其实是个基本功的问题，有的同学大学四年毕业，连个电阻的取值都不知道，这就是平时的基本功不够。这里的电阻可以叫做限流电阻，因为 LED 灯的亮其实是靠电流点亮，一般的 LED 的电流需要3-10mA,过低不会亮，过高就烧掉，所以我经常看见有的同学在实验室大叫，我的灯烧了，这就是限流电阻选的不够，那么假设我们取3mA 为工作电流,大家拿到板子后可以用电压表测量一下 LED 两端的电压，大概是1.7v 的样子，这样我们就可以得到电阻2端的压降为：5-1.7=3.3v，那么 R15的阻值就是3.3v/3mA=1.1k,我们取1k 的阻值，肯定可以满足通断的要求。所以你看电阻的取值其实大有讲究的！希望大家注意。

HT7533-3.3是一个输出3.3v 的芯片，由于我们 can 总线学习板在当初设计的时候，配有 nRF2401的无线通信芯片，需要3.3V 电源，所以选用这个。这个3.3v 的电源是在扩展口给大家引出的，以便于大家在利用我们的 can 总线开发板做扩展使用需要3.3v 的片子时用，毕竟3.3v 的器件目前很多。这一讲我们讲了电源和按键，下一讲我们将讲一讲 can 总线学习板的数码管显示部分。敬请关注！

我们在板子上为大家提供了显示单元，用的是数码管，我想对于这个大家再熟悉不过了，电路如下：




大家应锻炼自己对于电子器件的敏感程度，我一提数码管，大家立刻应该想问的是什么呢？对了，就是数码管是共阳的还是共阴的？我们这里采用的是共阳极的，下面的9015是 PNP 的三级管，既当作开关用又当作驱动电路，这个电路从使用的效率以及成本上有非常好，缺点呢，就是占用口线的使用，比如我们常常使用的 MAX7219就是专门用来驱动数码管的芯片，当然价格也是非常高！

我们来看看这个电路的工作原理，P24，P25，P26，P27 我们叫做位选线，顾名思义就是选中哪个，哪一位就亮，既然是共阳极，那就是公共端是高电平时才能点亮，因此只有位线是低电平的时候才选中相应的位亮。上面一排 AD0-AD7，就是数据线了，我们要事先建一个小的表，来存放每个数字的编码，具体的程序会在后续的课程中给大家介绍。

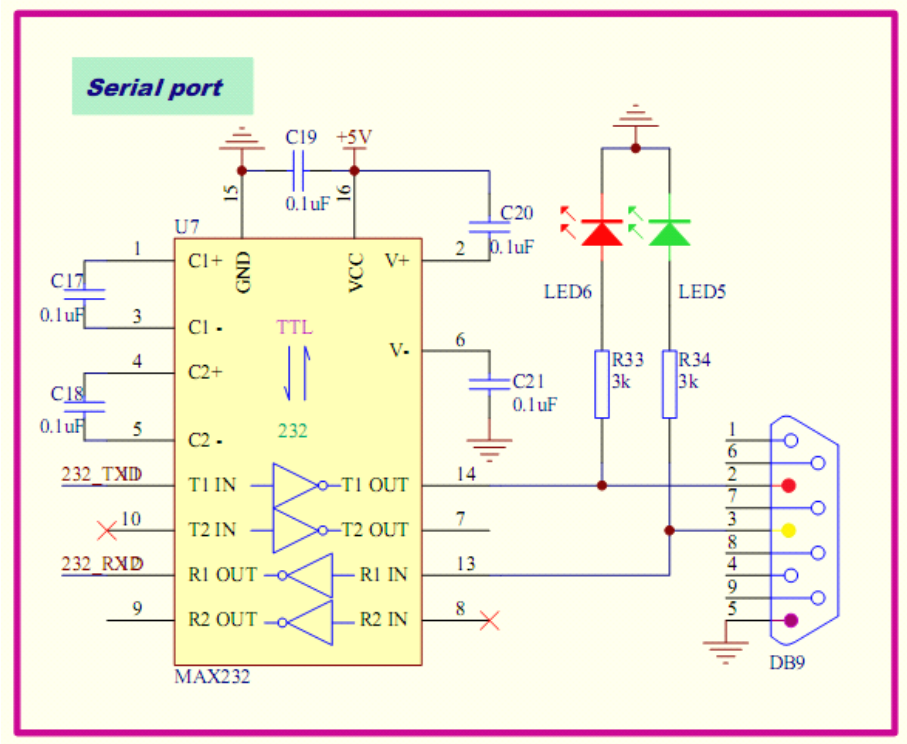
下一讲我们将简单介绍串口和扩展口部分。

## CAN 总线活动讲座七：CAN 开发板硬件-串口通讯

这一讲我们来说说串口通讯，串口是单片机的一个非常有用而且有趣的功能组件。它相当于单片机的一个“邮递员”，能够将单片机内部的信号传递出去，可以传递给另一个单片机，也可以传递给电脑。有一个很有趣的现象，现在的学生在做毕业设计的时候，几乎无一例外的将串口加入进去，他们也说不好目的，但是好像不加串口就少点什么似的，呵呵。其实串口的目的一方面是可以使单片机能与外界交流，另一方面它还可以帮我们实时了解系统中各个参数的变化。

比如我们的 can 总线，就可以通过串口实时的将状态量传递出来。其实串口的一个巨大的工作量就是上位机程序的编写，VC/VB/Delphi 等都能够完成，我本人特别钟爱使用 Labview，不知道大家对其了解与否，NI 公司的这个强大的软件目前正迅猛发展，你会发现在测控领域，无处不有它的身影。这回助学，我准备写一个小的 Labview 程序，来展示一下它的魅力。 其二，对于我们的 STC 单片机来说，它的作用更为巨大，可以让我们摆脱烧写器的束缚，直接串口将我们的程序烧写进单片机，非常容易！但是现在有的人用的笔记本电脑，没有串口，怎么办？针对这个，我们为这部分朋友特地开发了 USB 转串口的模块，可以直接通过 USB 下载，就更方便了。我会在后续的课程中介绍这个。 由于串口这部分很简单，所以我们

简要介绍，电路如下：



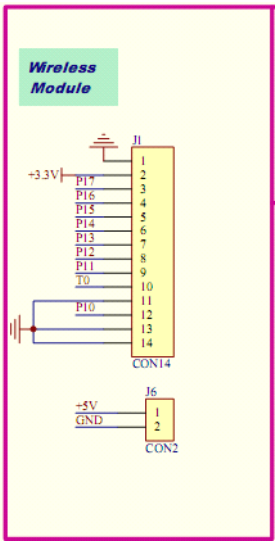

串口部分很简单，我想大家应该最为熟悉不过了，我们的板子用的是母头，一方面用来下载程序，另一方面用来与 PC 进行通讯。具体程序后续讲座中会给出。其中的 MAX232，不知道 wangjin 老师给大家讲了没有，这里我顺便提一句。它仅仅是一个电平转换芯片，因为单片机用的是 TTL 电平，电脑用的是232电平，（这部分的知识很多，尤其是现在电平种类异常繁杂，这里不详细说了），因此需要进行电平的转换。MAX232内部是通过电荷泵来进行电平提升的。我建议每个人都用你的电压表去看看 MAX232各个管脚的电平，你会发现一些有用的信息。它的电路也很简单，就记住“5电容”结构就行，一般放五个104电容，就可以解决问题！ 好了，这一讲我们就到这。在下一讲中，我们来讲一下，我们 CAN 开发板中比较有特色的一块-无线扩展接口和温度传感器部分。请大家关注！

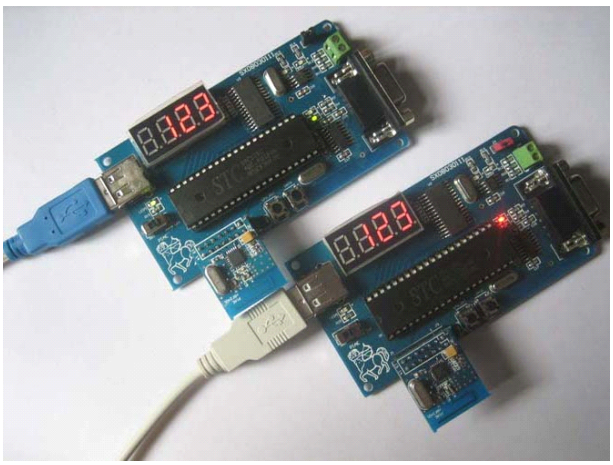
## CAN 总线活动讲座八：无线接口、温度测量以及 USB 下载配件

这一讲，我们介绍 CAN 学习板比较有特色的一部分内容，这些扩展模块，可以使我们利用 can 开发板完成更丰富的项目实例。

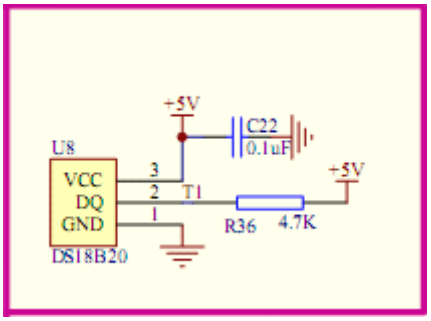
### 1.无线接口

CAN 总线学习板除了能进行 CAN 总线通信之外，我们还预留出无线的射频 NRF2401通讯模块的接口，可以方便的进行无线通讯学习，对于那些想参加竞赛以及做原型机开发的同学来说，无线等模块就更有用处了，无线通讯其实很有意思，比如我们可以利用我们的板子做一个无线抢答器，或者一个无线聊天系统，或者无线采集温度系统等等，nRF2401的功能强大，性能非常好，隔着门或是墙也能顺利的进行通讯，前一阵，我用我们的板子进行一个实验，在实验室外控制屋内的一个水样采集系统，在屋外看着屋内的水柱上升，感觉真的很神奇同时也很有成就感。下面这个图是以前的 CAN 学习板+无线模块的无线通讯试验，一发一收。我一共为大家写了3个通讯例程，包括无线同步时钟，手动无线通信和收发转换串口实验，包括控制字的配置以及发收驱动的编写，会发给购买无线模块的朋友。



### 2.DS18B20温度传感器




为了使大家能够讲 CAN 通信与实际的项目联系起来，在网友的建议下加入了一个简单的温度传感器，因为很多学生在做毕业设计的时候都喜欢用这个，wangjin 的51板子上也有对这个单总线器件的详细介绍，我想学习过的朋友都已经掌握了。那么这个 DS1820用在我们的 can 学习板上，就可以与 CAN 组成 can 的测温网络，同时也可以与无线的 nRF2401组成一个无线的测温网络。这就会使系统的功能大大丰富了。

## CAN 总线活动讲座九：CAN 总线硬件设计

这一讲我们详细介绍一下 CAN 总线通讯模块的硬件设计：CAN 总线学习板上 CAN 通讯模块的设计。包括三个部分：

（1）与 CPU 的接口；

（2）CAN 控制器 SJA1000与驱动器82C250接口及其他外围电路；

（3）82C250外围电路。

在选定了 CAN 控制器 SJA1000和 CAN 驱动器82C250后，我们肯定很想知道它的硬件电路怎么设计。其实这个比较简单，一般我们会遵循下面的步骤：

（1）在网络上搜索相关的资料，学习一下别人的设计，分析一下各自的优缺点。

（2）仔细研读 SJA1000和82C250的芯片资料，最终确定自己的硬件设计。

（3）使用 EDA 软件实施自己的设计，制作 PCB 电路板，这个过程中需要细心，耐心。

（4）焊接元器件。焊接过程中中时常检测焊接是否牢固，是否短路。

（5）接下来就是和软件配合调试了。硬件设计部分就到此为止了。

在 CAN 总线硬件设计过程中需要注意的地方有以下几点：

（1）电源的设计。这个应该是所有硬件设计的关键，所以在此也必须提醒一下！

（2）复位电路的设计。其设计方法分为三类：上电复位；手动按键复位；软件复位； 这三种复位都是可以的，但我认为如果您是初学者，选择硬件复位中的手动按键复位比较好，容易调试的过程中控制。软件复位比较灵活，您可以在程序中控制其复位。

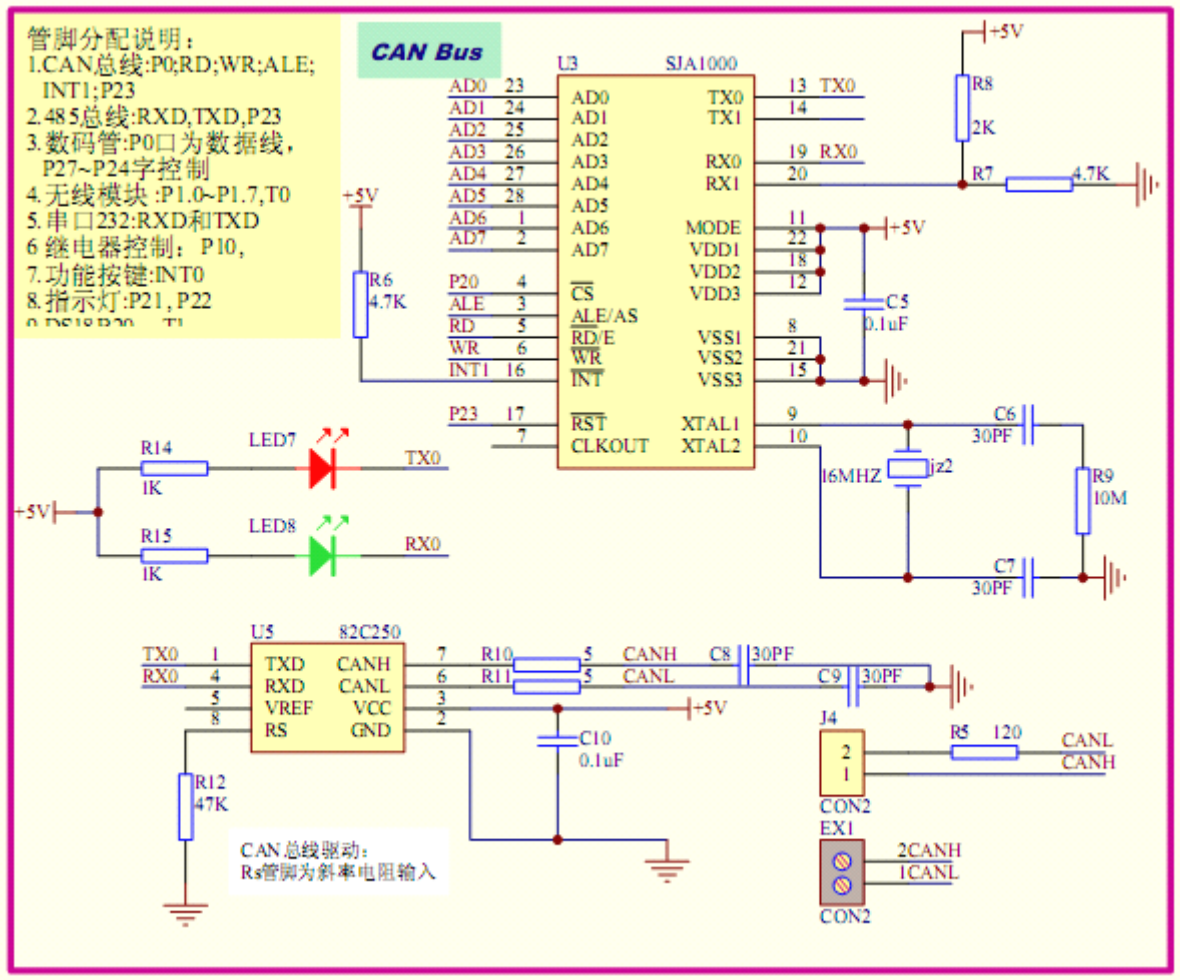
（3）时钟电路的设计。这部分是非常重要的，不过一般都有成熟的电路，所以不用担心这个。SJA1000有一个可编程的时钟输出，可以连接到 CPU 的时钟输入管脚，提供时钟信号。但是对于初学者而且对面积要求又不是很苛刻的，建议您给 CPU 一个单独的晶振电路提供时钟信号。

（4）CANH 和 CANL 管脚上最好增加电容滤波，提高抗干扰，电阻限流电路，提高电路保护。

（5）匹配电阻电路设计。

我相信大家如果考虑到了这5个方面的问题，而且搞清楚了，您设计的电路应该没有问题。下一部的工作就是准备相关的只是，进行软件设计。

电路如下：




## 1 SJA1000与 CPU 接口

我们在学习单片机原理的时候，我相信大家都学习过 RAM，ROM，I/O 口扩展。大家可以把 SJA1000看作一个外部的 RAM，扩展电路十分简单。SJA1000支持两种模式单片机的连接，我们选用的是8051系列的单片机，所以选择的是 Intel 模式。

（1）SJA1000的数据线和地址线是共用的，STC89C52的数据线和地址线也是共用的，这就更加方便了，直接连接就 OK 了。

（2）既然数据线和地址线共用，必须区分某一时刻，AD 线上传输的是地址还是数据，所以就需要连接地址锁存信号 ALE。

（3）随便使用一个单片机管脚作为 SJA1000的片选信号，我们学习板使用的是 P20。当然你也可以直接接地。

（4）读写信号直接和单片机连接就行了，就不必多说了！

（5）我们采用单片机的 IO 口线控制 SJA1000的 RST 管脚，是为了软件可以实现硬复位 SJA1000芯片。

（6）SJA1000的中断管脚连接单片机的 INT1外部中断。当收到一包数据后，通知 CPU。

## 2 SJA1000与82C250的接口及其他外围电路

（1）SJA1000有两路发送和接收管脚，CAN 总线学习板使用了第0路。与82C250的连接比较简单，直接连接就可以了。但应该数据发送和接收管脚不要接反了。而且我们增加了通讯状态指示灯，便于调试。

（2）时钟电路：SJA1000的最高时钟可达24M，我们学习板使用的是16M 的晶振。另外增加了一个启动电阻 R9（10M 欧姆）。

（3） 82C250外围电路

CANH 和 CANL 管脚增加阻容电路，滤除总线上的干扰，提高系统稳定性。

RS 管脚为斜率电阻输入。通过这个管脚来选择82C250的工作模式：高速模式（应用与对数据传输速率高的情况，通讯数据线最好是屏蔽的）；斜率模式（速度较低，通讯线可以是普通的双绞线）。准备模式（应用于对功耗要求比较高的场合）。我们的学习板采用的是斜率模式，方便大家学习。

J3是外部总线的连接口，J4是终端电阻的选择端。

到现在为止，CAN 总线学习的硬件部分就介绍完了，其实还有很多地方值得讨论，比如隔离的问题，但是为了进行助学活动，考虑成本因素，隔离暂时没有加入学习版，但是对于初学者来说已经完全够用了。相信这块 CAN 开发板能够为你的总线学习带来方便

## CAN 总线活动讲座十：CAN 总线基础扫盲讲座

CAN 总线作为一种工业界的流行总线广泛应于工业自动化、多种控制设备、交通工具、医疗仪器以及建筑、环境控制等各个行业中，它是是一种多主机局域网，所以这样一种总线的潜力是很巨大的，接下来将写几篇入门的小文章，来介绍一下 CAN 的学习和开发方法，主要是配合我们的开发板，其实入门还是很容易的，通过这几篇文章如果你能建立起一个初步的概念，能够自己搭起自己的一个节点（包括硬件和软件），那就说明你学会了 CAN 的基本开发技术，实际的工业现场 CAN 的开发不是那么简单，包括很多要考虑的因素如隔离、可靠性等，但是我们作为初学者建立起基本的概念还是很重要的，基础一旦打好，等到了实际工作中，就会如鱼得水，所以希望正要或是想开发 CAN 的朋友们能够喜欢这一系列小文章，如果某一篇甚至是某一句话能够给你带来启迪，那将是我最最高兴的，希望大家踊跃评论或是参与讨论，给我一下写下去的信心，呵呵，可以加入我们的群。 下面正题正式开始喽！第一篇，当然要介绍一下什么是“能”总线（CAN 总线）呵呵

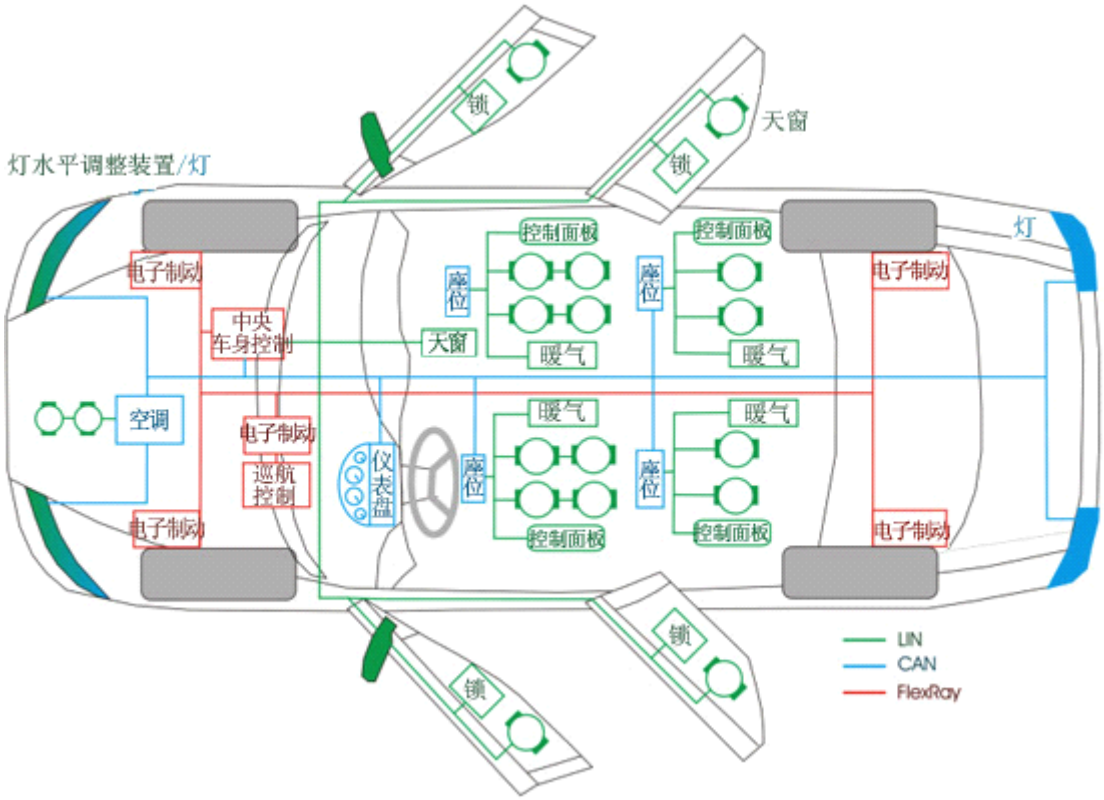



图1. 汽车中的 CAN“身影”

### 一、什么是 CAN ?

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。最初，CAN 被设计作为汽车环境中的微控制器通讯，在车载各电子控制装置 ECU 之间交换信息，形成汽车电子控制网络。比如：发动机管理系统、变速箱控制器、仪表装备、电子主干系统中，均嵌入 CAN 控制装置。 一个由 CAN 总线构成的单一网络中，理论上可以挂接无数个节点。实际应用中，节点数目受网络硬件的电气特性所限制。例如，当使用 Philips P82C250作为 CAN 收发器时，同一网络中允许挂接110个节点。CAN 可提供高达1Mbit/s 的数据传输速率，这使实时控制变得非常容易。另外，硬件的错误检定特性也增强了 CAN 的抗电磁干扰能力.

## 二、CAN 是怎样发展起来的？

CAN 最初出现在80年代末的汽车工业中，由德国 Bosch 公司最先提出。当时，由于消费者对于汽车功能的要求越来越多，而这些功能的实现大多是基于电子操作的，这就使得电子装置之间的通讯越来越复杂，同时意味着需要更多的连接信号线。提出 CAN 总线的最初动机就是为了解决现代汽车中庞大的电子控制装置之间的通讯，减少不断增加的信号线。于是，他们设计了一个单一的网络总线，所有的外围器件可以被挂接在该总线上。1993年，CAN 已成为国际标准 ISO11898(高速应用)和 ISO11519（低速应用）。 CAN 是一种多主方式的串行通讯总线，基本设计规范要求有高的位速率，高抗电磁干扰性，而且能够检测出产生的任何错误。当信号传输距离达到10Km 时，CAN 仍可提供高达50Kbit/s 的数据传输速率。 由于 CAN 总线具有很高的实时性能，因此，CAN 已经在汽车工业、航空工业、工业控制、安全防护等领域中得到了广泛应用

## 三、CAN 简介

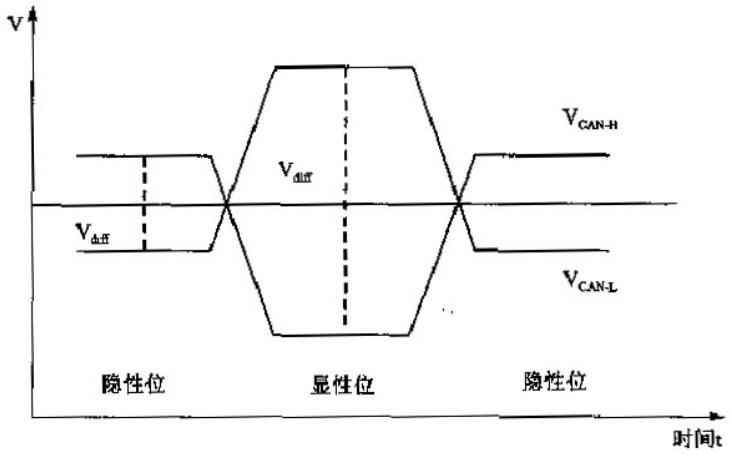
德国 Bosch 公司为解决现代车辆中众多的控制和数据交换问题,开发出一种 CAN(Controller Area Network)现场总线通信结构。CAN 总线硬件连接简单,有良好的可靠性、实时性和性能价格比。CAN 总线能够满足现代自动化通信的需要，已成为工业数据总线通信领域中最为活跃的一支。其主要特点是： ① CAN 总线为多主站总线，各节点均可在任意时刻主动向网络上的其他节点发送信息，不分主从，通信灵活；② CAN 总线采用独特的非破坏性总线仲裁技术，优先级高的节点优先传送数据，能满足实时性要求； ③ CAN 总线具有点对点、一点对多点及全局广播传送数据的功能； ④ CAN 总线上每帧有效字节数最多为8个，并有 CRC 及其他校验措施，数据出错率极低，万一某一节点出现严重错误，可自动脱离总线，总线上的其他操作不受影响； ⑤ CAN 总线只有两根导线，系统扩充时，可直接将新节点挂在总线上即可，因此走线少，系统扩充容易，改型灵活； ⑥ CAN 总线传输速度快，在传输距离小于40 m 时，最大传输速率可达1 Mb/s; ⑦ CAN 总线上的节点数主要取决于总线驱动电路,在 CAN2.0B 标准中,其报文标识符几乎不受限制。总之,CAN 总线具有实时性强、可靠性高、通信速率快、结构简单、互操作性好、总线协议具有完善的错误处理机制、灵活性高和价格低廉等特点。

## 四、CAN 总线是怎样工作的？

CAN 通讯协议主要描述设备之间的信息传递方式。CAN 层的定义与开放系统互连模型（OSI）一致。每一层与另一设备上相同的那一层通讯。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 的规范定义了模型的最下面两层：数据链路层和物理层。下表中展示了 OSI 开放式互连模型的各层。应用层协议可以由 CAN 用户定义成适合特别工业领域的任何方案。已在工业控制和制造业领域得到广泛应用的标准是 DeviceNet，这是为 PLC 和智能传感器设计的。在汽车工业，许多制造商都应用他们自己的标准。

| | | |
|---|---|---|
| 7 | 应用层 | 最高层。用户、软件、网络终端等之间用来进行信息交换。如：DeviceNet |
| 6 | 表示层 | 将两个应用不同数据格式的系统信息转化为能共同理解的格式 |
| 5 | 会话层 | 依靠低层的通信功能来进行数据的有效传递。 |
| 4 | 传输层 | 两通讯节点之间数据传输控制。操作如：数据重发，数据错误修复 |
| 3 | 网络层 | 规定了网络连接的建立、维持和拆除的协议。如：路由和寻址 |
| 2 | 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 |
| 1 | 物理层 | 规定通讯介质的物理特性。如：电气特性和信号交换的解释 |

表1 OSI 开放系统互连模型




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。信号使用差分电压传送，两条信号线被称为“CAN_H”和“CAN_L”，静态时均是2.5V 左右，此时状态表示为逻辑“1”，也可以叫做“隐性”。用 CAN_H 比 CAN_L 高表示逻辑“0”，称为“显形”，此时，通常电压值为：CAN_H = 3.5V 和 CAN_L = 1.5V。

## 五、CAN 有哪些特性？

CAN 具有十分优越的特点，使人们乐于选择。想了想，不想向教科书那样罗嗦很多，就简单概括，也许有些你还不能立刻懂，不过你只要有个印象就行，一些知识直到你用到的时候你才会真正领会！这些特性包括： 1、低成本； 2、极高的总线利用率； 3、很远的数据传输距离(长达10Km)； 4、高速的数据传输速率（高达1Mbit/s)； 5、可根据报文的 ID 决定接收或屏蔽该报文； 6、可靠的错误处理和检错机制; 7、发送的信息遭到破坏后，可自动重发； 8、节点在错误严重的情况下具有自动退出总线的功能； 9、报文不包含源地址或目标地址，仅用标志符来指示功能信息、优先级信息。

## 六、Philips 制造的 CAN 芯片有哪些？

| 类别 | 型号 | 备注 |
|---|---|---|
| CAN 微控制器 | P87C591 | 替代 P87C592 |
| CAN 独立控制器 | SJA1000 | 替代82C200 |
| CAN 收发器 | PCA82C250 | 高速 CAN 收发器 |
| | PCA82C251 | 高速 CAN 收发器 |
| | PCA82C252 | 容错 CAN 收发器 |
| | TJA1040 | 高速 CAN 收发器 |
| | TJA1041 | 高速 CAN 收发器 |
| | TJA1050 | 高速 CAN 收发器 |
| | TJA1053 | 容错 CAN 收发器 |
| | TJA1054 | 容错 CAN 收发器 |
| LIN 收发器 | TJA1020 | LIN 收发器 |

表2 CAN 芯片一览表

## 七、CAN 总线如何进行位仲裁？

CSMA/CD 是“载波侦听多路访问/冲突检测”（Carrier Sense Multiple Access with Collision Detect）的缩写。　　利用 CSMA 访问总线，可对总线上信号进行检测，只有当总线处于空闲状态时，才允许发送。利用这种方法，可以允许多个节点挂接到同一网络上。当检测到一个冲突位时，所有节点重新回到‘监听’总线状态，直到该冲突时间过后，才开始发送。在总线超载的情况下，这种技术可能会造成发送信号经过许多延迟。为了避免发送时延，可利用 CSMA/CD 方式访问总线。当总线上有两个节点同时进行发送时，必须通过“无损的逐位仲裁”方法来使有最高优先权的的报文优先发送。在 CAN 总线上发送的每一条报文都具有唯一的一个11位或29位数字的 ID。CAN 总线状态取决于二进制数‘0’而不是‘1’，所以 ID 号越小，则该报文拥有越高的优先权。因此一个为全‘0’标志符的报文具有总线上的最高级优先权。可用另外的方法来解释：在消息冲突的位置，第一个节点发送0而另外的节点发送1，那么发送0的节点将取得总线的控制权，并且能够成功的发送出它的信息。

## 八、CAN 的高层协议

CAN 的高层协议（也可理解为应用层协议）是一种在现有的底层协议（物理层和数据链路层）之上实现的协议。高层协议是在 CAN 规范的基础上发展起来的应用层。许多系统（像汽车工业）中，可以特别制定一个合适的应用层，但对于许多的行业来说，这种方法是不经济的。一些组织已经研究并开放了应用层标准，以使系统的综合应用变得十分容易。　　一些可使用的 CAN 高层协议有：

1、制定组织主要高层协议　2、CiA CAL 协议　　3、CiA CANOpen 协议　4、ODVA DeviceNet 协议　5、Honeywell SDS 协议　6、Kvaser CANKingdom 协议

## 九、什么是标准格式 CAN 和扩展格式 CAN?

标准 CAN 的标志符长度是11位,而扩展格式 CAN 的标志符长度可达29位 。CAN 协议的2.0A 版本规定 CAN 控制器必须有一个11位的标志符。同时，在2.0B 版本中规定，CAN 控制器的标志符长度可以是11位或29位 。遵循 CAN2.0B 协议的 CAN 控制器可以发送和接收11位标识符的标准格式报文或29位标识符的扩展格式报文。如果禁止 CAN2.0B,则 CAN 控制器只能发送和接收11位标识符的标准格式报文，而忽略扩展格式的报文结构，但不会出现错误。　　目前，Philips 公司主要推广的 CAN 独立控制器均支持 CAN2.0B 协议，即支持29位标识符的扩展格式报文结构。

## CAN 总线活动讲座十一：CAN 总线硬件设计 CAN 总线与485总线比较

上一讲我们介绍了 CAN 总线的基础知识，那么有人会问，现在的总线格式很多，CAN 相对于其他的总线有什么特点啊？有什么特别的优势，让我们必须选择这种总线呢？这个问题问的好，所以我想与其它总线做一下比较，首先呢，就比较一下大家耳熟能详的485总线吧。其实485总线与232差不多，它们只定义了物理层，规定了电平标准。

下面我们进行详细的对比： CAN(Controller Area Network)属于现场总线的范畴，它是一种有效支持分布式控制或实时控制的串行通信网络。较之目前 RS-485基于 R 线构建的分布式控制系统而言， 基于 CAN 总线的分布式控制系统在以下方面具有明显的优越性：

1） CAN 控制器工作于多主方式，网络中的各节点都可根据总线访问优先权(取决于报文标识符)采用无损结构的逐位仲裁方式竞争向总线发送数据，且 CAN 协议废除了站地址编码，而代之以对通信数据进行编码，这可使不同的节点同时接收到相同的数据，这些特点使得 CAN 总线构成的网络各节点之间的数据通信实时性强，并且容易构成冗余结构，提高系统的可靠性和系统的灵活性。而利用 RS-485只能构成主从式结构系统，通信方式也只能以主站轮询的方式进行，系统的实时性、可靠性较差。

2） CAN 总线通过 CAN 控制器接口芯片82C250的两个输出端 CANH 和 CANL 与物理总线相连，而 CANH 端的状态只能是高电平或悬浮状态，CANL 端只能是低电平或悬浮状态。这就保证不会出现象在 RS-485 网络中，当系统有错误，出现多节点同时向总线发送数据时，导致总线呈现短路，从而损坏某些节点的现象。而且 CAN 节点在错误严重的情况下具有自动关闭输出功能，以使总线上其他节点的操作不受影响，从而保证不会出现象在网络中，因个别节点出现问题，使得总线处于“死锁”状态。

3） CAN 具有完善的通信协议，可由 CAN 控制器芯片及其接口芯片来实现，从而大大降低了系统的开发难度，缩短了开发周期，这些是只仅仅有电气协议的 RS-485所无法比拟的。

| 特性 | RS-485 | CAN-bus |
|---|---|---|
| 单点成本 | 低廉 | 稍高 |
| 系统成本 | 高 | 较低 |
| 总线利用率 | 低 | 高 |
| 网络特性 | 单主网络 | 多主网络 |
| 数据传输率 | 低 | 高 |
| 容错机制 | 无 | 可靠的错误处理和检错机制 |
| 通讯失败率 | 高 | 极低 |
| 节点错误的影响 | 导致整个网络的瘫痪 | 无任何影响 |
| 通讯距离 | <1.5km | 可达10km（5kbps） |
| 网络调试 | 困难 | 非常容易 |
| 开发难度 | 标准 Modbus 协议 | 标准 CAN-bus 协议 |
| 后期维护成本 | 高 | 低 |

## CAN 总线活动讲座十二：CAN 控制器的选择

在进行 CAN 总线开发前，首先要选择好 CAN 总线控制器。下面就比较一些控制器的特点。

一些主要的 CAN 总线器件产品

| 制造商 | 产品型号 | 器件功能及特点 |
|---|---|---|
| Intel | 82526 82527<br>8XC196CA/CB | CAN 通信控制器，<br>符合 CAN2.0A CAN 通信控制器，<br>符合 CAN2.0B 扩展的8XC196+CAN 通信控制器，<br>符合 CAN2.0A |
| Philips | 82C200 SJA1000<br>82C250 TJA1040<br>TJA1054 8XC592<br>8XCE598 P51XA-C3 | CAN 通信控制器，<br>符合 CAN2.0A CAN 通信控制器,<br>82C200的替代品, 符合 CAN2.0B 通用 CAN 总线收发器,<br>高速 CAN 总线收发器<br>容错的 CAN 总线收发器<br>8XC552+CAN 通信控制器,<br>去掉 IIC 符合 CAN2.0A 提高了电磁兼容性的8XC592<br>16位微控制器+CAN 通信控制器,<br>符合 CAN2.0B |
| Motorola | 68HC05X4系列 | 68HC05微控制器+CAN 通信控制器，<br>符合 CAN2.0A |
| Siemens | 81C90/91 C167C | CAN 通信控制器，<br>符合 CAN2.0B 微控制器+CAN 通信控制器，<br>符合 CAN2.0A/B |

因为 SJA1000比较简单，资料丰富，对于初学者非常适用，所以我们采用 SJA1000作为我们总线开发板的 CAN 控制器，让大家更容易入门。 SJA1000是一种独立的 CAN 控制器，主要用于移动目标和一般工业环境中的区域网络控制。它是 Philips 公司 PCA82C200控制器的替代产品，除了 PCA82C200的 BasicCAN 操作模式以外，还增加了一种新的操作模式——PeliCAN，这种模式支持具有很多新特性的 CAN2. 0B 协议。SJA1000的基本特征如下：

1)引脚电气参数与 PCA82C200兼容；

2)具有 PCA82C200模式(即默认的 BasicCAN 模式)，支持 CAN2.0A 和 CAN2.0B；

3)有扩展的接收缓冲器64字节，先进先出(FIFO)；

4)支持11位和29位标识码，通信速率可达1Mbps；

5)其 PeliCAN 模式的扩展功能包括：可读写的错误计数器，可编程的错误报警限额寄存器，最近一次错误代码寄存器，对每一个总线错误的中断，有具体位表示的仲裁丢失中断，单次发送(无重发)，支持热拔插，可扩展的验收滤波器，可接收自身报文(自请求接收)。

## CAN 总线活动讲座十三：CAN 协议简单介绍

硬件设计好了，在进行软件编程之前我们最好对 CAN 总线的协议有个了解。

### 一 ：报文传送由以下四种帧类型

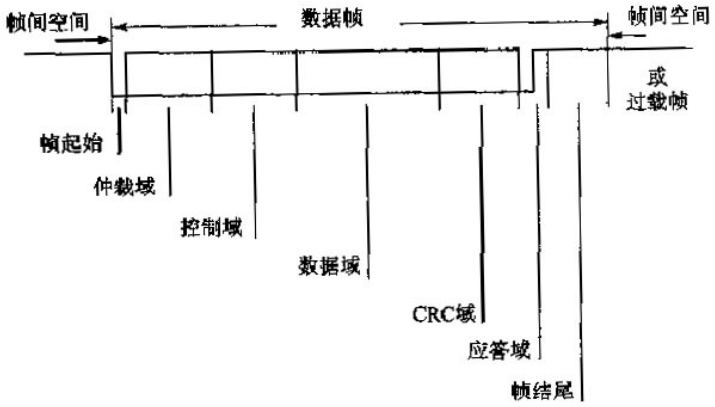
**数据帧：** 数据帧携带数据从发送器至接收器。

**远程帧：** 总线单元发出远程帧，请求发送具有同一识别符的数据帧。

**错误帧：** 任何单元检测到一总线错误时就发送错误帧。

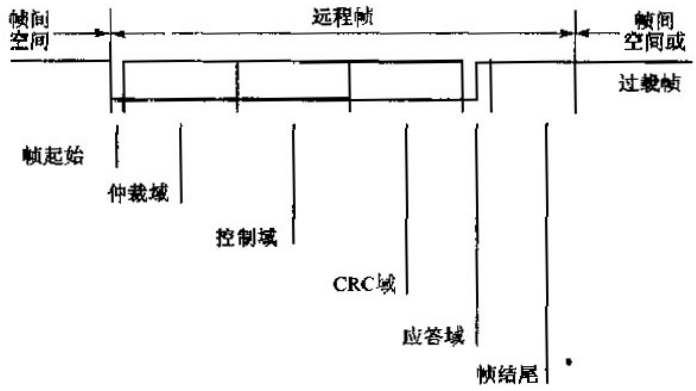
**超载帧：** 用来在先行的和后续的数据帧（或远程帧）之间提供一附加的延时。

### 二 ：帧格式介绍



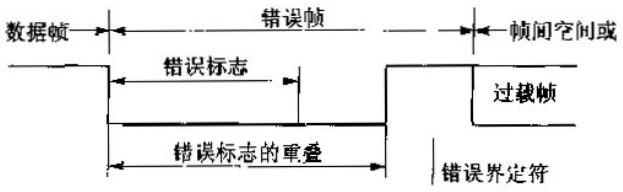

报文的数据帧结构

**数据帧** ： 数据帧由7个不同的位场组成，即帧起始、仲裁场、控制场、数据场、CRC 场、应答场、帧结束



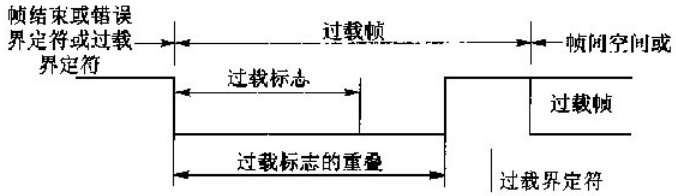

远程帧结构

**远程帧** ： 远程帧由6个不同的位场组成，即帧起始、仲裁场、控制场、CRC 场、应答场、帧结束。




**错误帧结构**

**错误帧** ：错误帧由两个不同的场组成。第一个场是错误标志，用做为不同站提供错误标志的叠加；第二个场是错误界定符




**过载帧结构**

**超载帧** ：超载帧包括两个位场：超载标志和超载界定符。

## 三 ： 帧格式中重点部分介绍

**帧起始** ：帧起始标志数据帧和远程帧的起始，由一个单独的“显性”位组成。由控制芯片完成。

**仲裁场** ：仲裁场包括标识符和远程发送请求位（RTR）。 对于 CAN2.0A 标准，标识符的长度为11位。RTR 位在数据帧中必须是显性位，而在远程帧必须为隐性位。 对于 CAN2.0，标准格式和扩展格式的仲裁场不同。在标准格式中，仲裁场由11位标识符和远程发送请求位组成。在扩展格式中，仲裁场由29位标识符和替代远程请求位（SRR） 、标志位（IDE）和远程发送请求位组成。 仲裁场的作用之一是说明数据帧或远程帧发送目的地；之二是指出数据帧或远程帧。仲裁场的数据由软件编程配置 SJA1000完成。

**控制场** ：控制场由6个位组成，说明数据帧中有效数据的长度。控制场的数据由软件编程配置 SJA1000完成。

**数据场** ：数据场由数据帧中的发送数据组成。它可以为0~8个字节。数据场的数据由软件编程配置 SJA1000 完成。

**CRC 场** ：CRC 场包括 CRC 序列，这部分由 SJA1000控制芯片完成。

**应答场** ：应答场长度为两个位，包括应答间隙和应答界定符。由 SJA1000控制芯片自动完成。

**帧结束** ：每一个数据帧和远程帧均由一标志序列界定，这个标志序列由7个“隐性”位组成。这部分由 SJA1000控制芯片自动完成。 总之，仲裁场、控制场、数据场由软件编程配置 SJA1000完成；帧起始、CRC 场、应答场、帧结束由 CAN 总线控制芯片 SJA1000自动完成。

其中要重点理解的就是数据帧和远程帧以及组成帧的重要部分。所以初学者首先把这两种帧格式掌握了，学习 CAN 总线应该就可以入门了。

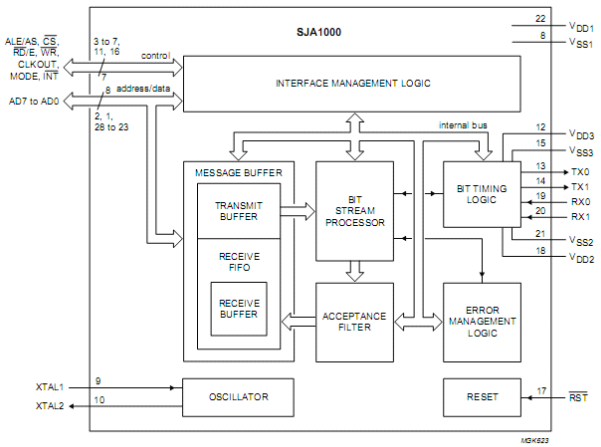
## CAN 总线活动讲座十三（插入篇一）：SJA1000芯片的概述

SJA1000是一种独立的 CAN 控制器，主要用于移动目标和一般工业环境中的区域网络控制。它是 Philips 半导体公司 PCA82C200 CAN 控制器(BasicCAN)Mlt 代产品，而且它增加了一种新的操作模式-PeIiCAN.这种模式支持具有很多新特性的 CAN 2.013协议

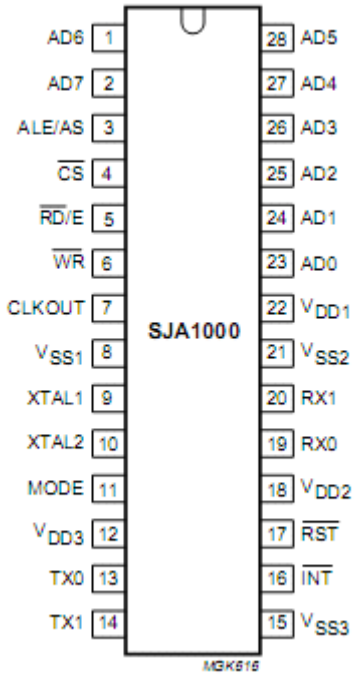
### SJA1000的基本特性如下

- 引脚与 PCA82C200独立 CAN 控制器兼容
- 电气参数与 PCA82C200独立 CAN 控制器兼容。
- 具有 PCA82C200模式（即欲认的 BasicCAN IM.
- 有扩展的接收缓冲器“字节，先进先出(FIFO)
- 支持 CAN 2.0A 和 CAN 2.0B 协议.
- 支持11位和29位标识码.
- 通信位速率可达1 Mbps
- PeIiCAN 模式的扩展功能有如下：
  - 可读 / 写访问的错误计数寄存器
  - 可编程的错误报警限额寄存器钾
  - 最近一次错误代码寄存器
  - 对每一个 CAN 总线错误的中断苍
  - 有异体位表示的仲裁丢失中断
  - 单次发送〔无重发）
  - 只听模式〔无确认、无激活的错误标志）
  - 支持热插拔〔软件进行位速率检测）
  - 验收滤波器的扩展 C4字节的验收代码,4字节的屏蔽〕
  - 接收自身报文（自接收请求）
- -24 MHz 时钟频率.
- -可与木同的徽处理器接口.
- -可编程的 CAN 输出驱动器配置.
- -温度适应范围大（-40℃～＋125℃）口

### SJA0000内部结构放宽图



## 芯片引脚排列与名称

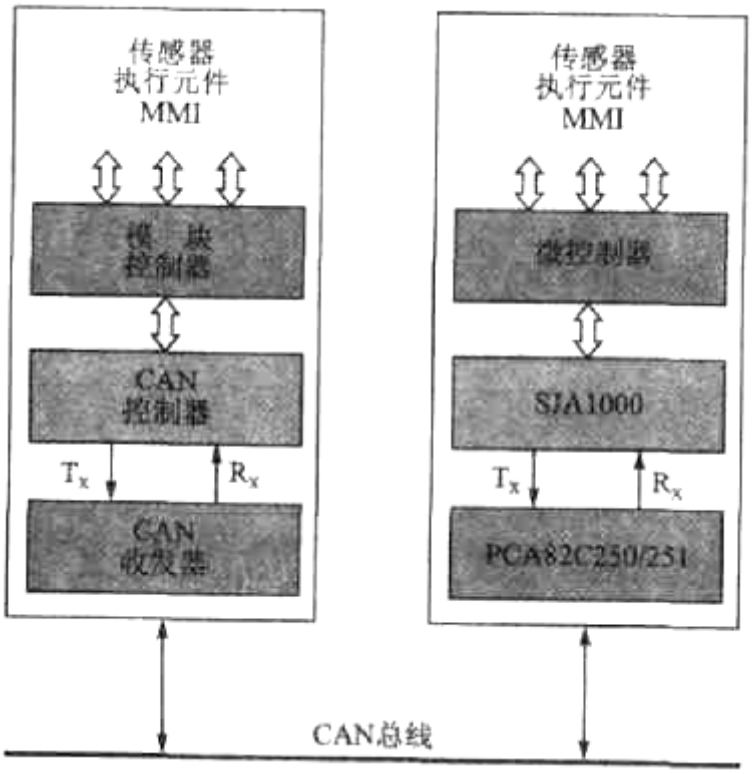



## 引脚定义

| 名称符号 | 引脚号 | 功能描述 |
|---|---|---|
| AD7～AD0 | 2,1,28～23 | 地址/数据复合总线 |
| ALE/AS | 3 | ALE 输入信号(Intel 模式),AS 输入信号(Motorola 模式) |
| $\overline{CS}$ | 4 | 片选信号输入,低电平允许访问 SJA1000 |
| ($\overline{RD}$)/E | 5 | 微控制器的$\overline{RD}$信号(Intel 模式)或 E 使能信号(Motorola 模式) |
| $\overline{WR}$ | 6 | 微控制器的$\overline{WR}$信号(Intel 模式)或 RD/$\overline{WR}$信号(Motorola 模式) |
| CLKOUT | 7 | SJA1000 产生的提供给微控制器的时钟输出信号,它来自内部振荡器且通过编程分频;时钟分频寄存器的时钟关闭位可禁止该引脚输出 |
| $V_{SS1}$ | 8 | 接地 |
| XTAL1 | 9 | 输入到振荡器放大电路;外部振荡信号由此输入[1] |
| XTAL2 | 10 | 振荡放大电路输出;使用外部振荡信号时漏极开路输出[1] |
| MODE | 11 | 模式选择输入;1＝Intel 模式;0＝ Motorola 模式 |
| $V_{DD3}$ | 12 | 输出驱动的 5 V 电源 |
| TX0 | 13 | 从 CAN 输出驱动器 0 输出到物理线路上 |

| 名称符号 | 引脚号 | 功能描述 |
| --- | --- | --- |
| TX1 | 14 | 从 CAN 输出驱动器 1 输出到物理线路上 |
| $V_{SS3}$ | 15 | 输出驱动器接地 |
| $\overline{INT}$ | 16 | 中断输出，用于中断微控制器；在内部中断寄存器的任一位置 1 时，$\overline{INT}$ 低电平有效；开漏输出，且与系统中的其他 $\overline{INT}$ 输出是线性关系。此引脚上的低电平可以把该控制器从睡眠模式中激活 |
| $\overline{RST}$ | 17 | 复位输入，用于复位 CAN 接口(低电平有效)；把 $\overline{RST}$ 引脚通过电容连到 $V_{SS}$，通过电阻连到 $V_{DD}$，可自动上电复位(例如：C=1 μF；R=50 kΩ) |
| $V_{DD2}$ | 18 | 输入比较器的 5 V 电源 |
| RX0，RX1 | 19，20 | 从物理的 CAN 总线输入到 SJA1000 输入比较器；显性电平将唤醒 SJA100 的睡眠模式；如果 RX1 电平比 RX0 的高，就读显性电平，反之读隐性电平；如果时钟分频寄存器的 CBP 位被置 1，CAN 输入比较器被旁路以减少内部延时；当 SJA1000 连有外部收发电路时，只有 RX0 被激活，隐性电平被认为是逻辑高而显性电平被认为是逻辑低 |
| $V_{SS2}$ | 21 | 输入比较器的接地端 |
| $V_{DD1}$ | 22 | 逻辑电路的 5 V 电源 |

**CAN 控制器 SJA1000在系统中的位置**



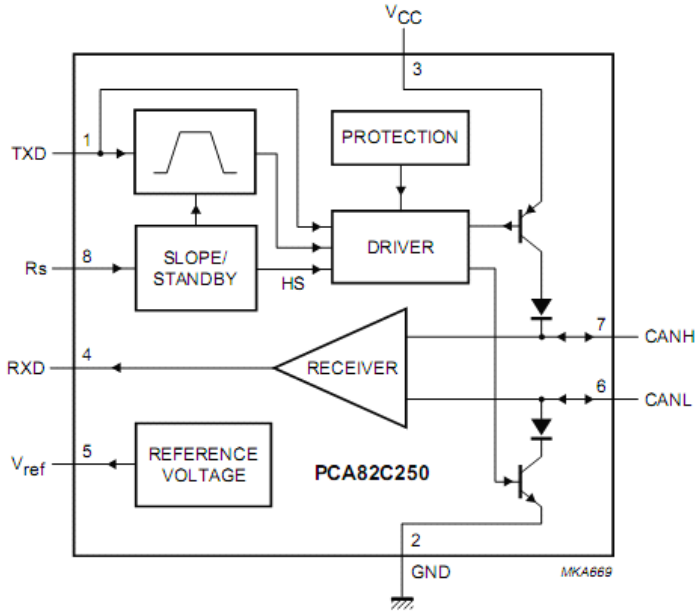
## CAN 总线活动讲座十三（插入篇二）：CAN 总线驱动器82C250的概述

总迷

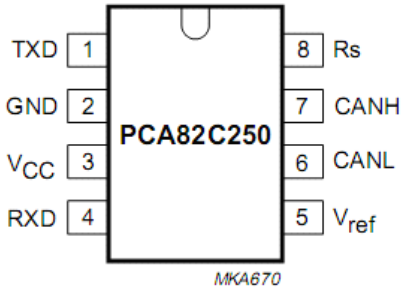
82C250是 CAN 控洲器与物理总烧之问的接口它最初是为汽车中的高速应用〔达 1 Mbps 而设计的．器件可以提供对总朋的差动发遇和接收功幽 ft2f15(1的主要掩性如下

- 与 ISO11898标准兜金旅容－
- 高速率（最高可达1 Mbps)
- 具有抗汽车环境下的瞬间干扰．
- 保护总线能力
- 采用斜率控翻 (Slope Control)
- 降低射倾干扰（RFI）
- 过热保护
- 总能与电源及地之间的短路保护
- 低电流待机模式
- 未上电节点不会干找总线
- 总线至少可连接110个节点．

**82C250内部结构放宽图**



## 82C250引脚输出




## 82C250引脚定义

| SYMBOL | PIN | DESCRIPTION |
|---|---|---|
| TXD | 1 | transmit data input |
| GND | 2 | ground |
| $V_{CC}$ | 3 | supply voltage |
| RXD | 4 | receive data output |
| $V_{ref}$ | 5 | reference voltage output |
| CANL | 6 | LOW-level CAN voltage input/output |
| CANH | 7 | HIGH-level CAN voltage input/output |
| Rs | 8 | slope resistor input |

## 82C250基本性能参数

| 符　号 | 参　数 | 条　件 | 最小值 | 典型值 | 最大值 | 单　位 |
|---|---|---|---|---|---|---|
| $V_{CC}$ | 电源电压 | | 4.5 | — | 5.5 | V |
| $I_{CC}$ | 电源电流 | 显性位，V1＝1 V | — | — | 70 | mA |
| | | 隐性位，V1＝4 V | — | — | 14 | mA |
| | | 待机模式 | — | 100 | 170 | μA |

| 符　号 | 参　数 | 条　件 | 最小值 | 典型值 | 最大值 | 单　位 |
|---|---|---|---|---|---|---|
| $V_{CAN}$ | CANH，CANL 脚直流电压 | 0 V＜$V_{CC}$＜5.5 V | −8 | — | +18 | V |
| ΔV | 差动总线电压 | V1＝1V | 1.5 | — | 3.0 | V |
| $V_{dif(r)}$ | 差动输入电压(隐性位) | 非待机模式 | −1.0 | — | 0.4 | V |
| $V_{dif(d)}$ | 差动输入电压(显性位) | 非待机模式 | 1.0 | — | 5.0 | V |
| td | 传播延迟 | 高速模式 | — | — | 50 | ns |
| $T_{amb}$ | 工作环境温度 | | −40 | — | +125 | ℃ |

## CAN 总线活动讲座十四：SJA1000重要的寄存器

要编写 CAN 总线通讯程序，只要了解 CAN 总线协议，熟悉 SJA1000寄存器的配置，就可以完成 CAN 总线通讯。所以我们首先必须要详细了解 SJA1000的寄存器。

### SJA1000的工作模式

SJA1000的两个工作模式(Basic 和 Peli)所使用的寄存器数目不同,功能也不尽相同。Basic CAN 有从0-31 共32 个寄存器可用，Peli CAN 有从0-127 共128 个寄存器可用。要实现 CAN 通讯，主要就是怎么配置这些寄存器。

### 要掌握的重要寄存器

要掌握的重要寄存器又如下：模式寄存器；命令寄存器；状态寄存器；中断寄存器；中断使能寄存器；总线定时器0，总线定时器1；输出控制寄存器；时钟分频寄存器；屏蔽寄存器0-3；验收代码寄存器0-3。而以上寄存器的基本左右如下：

- 模式寄存器的作用：控制 SJA1000的状态模式有如：睡眠模式；自检测模式；复位模式；只听模式。
- 命令寄存器的作用：启动发送或自发送；释放接收寄存器；中止发送
- 状态寄存器的作用：指示 SJA1000的状态，以判断是否可以进行下一步操作。
- 中断寄存器的作用：当发生中断后，读其值可以判断是什么原因引起的中断。
- 中断使能寄存器的作用：打开相应的中断。
- 总线定时器的作用：设置通讯的速率。
- 输出控制寄存器的作用：控制输出模式
- 时钟分频寄存器的作用：控制 CAN 总线采用那种模式。
- 验收代码寄存器和屏蔽寄存的作用：决定接收哪类标志码的数据。注意验收滤波器的设置。

### 复位模式（追加）

当 SJA1000检测有复位请求后将终止当前就收/发送的报文而进入如复位模式，复位模式可以是硬件复位或者软件复位，而当 SJA1000进入复位模式，所有的寄存器都会有“复位值"，详细的资料就查看手册吧

### 关于地址分配 （追加）

地址分配又分 BasicCAN 地址分配，与 PeiliCAN 地址分配

### 不同的工作模式下，可用的寄存器也不同 （追加）

在不同的工作模式下如在 Peli 的模式下模式寄存器可用，而 Basic 的工作模式下控制寄存器可用。

### 关于同一个寄存器在不同的工作模式下 （追加）

即使是同一个寄存器：如中断寄存器，在不同的工作模式下（Basic 或者 Peli）明显都有不同的设置方法，详细的资料还是参考手册吧，一一介绍会死人

## CAN 总线活动讲座十五：CAN 总线硬件调试及软件编程

在软硬件联调的时候，必须首先要确保硬件是否工作正常。 硬件正常是整个调试工作的基础，在进行软件调试之前首先需要仔细检查硬件连接。保证每一个连接是正确的，没有虚焊。而在所有连接中 CPU 与 CAN 控制器的连接又是最重要的。所以我们采用软件方法对 CAN 控制器与 CPU 的连接接口进行了检测测试。检测步骤如下：

1) CAN 节点上电复位后，检测 SJA1000的复位管脚电平应为高电平，反之说明 SJA1000的复位电路不正常 。
2) 向 SJA1000的测试寄存器写入 AAH，再读 SJA1000的测试寄存器，结果应该是 AAH，如果不是，说明数据线，地址线，控制线的连接有问题。
3) 向 SJA1000的测试寄存器写入55H，再读 SJA1000的测试寄存器，结果应该是55H，如果不是，说明数据线，地址线，控制线的连接有问题。
4) 在 CAN 总线驱动器的总线端接上负载电阻120欧姆，软件强制 SAJ1000进入工作模式，看其是否真正进入工作模式。若已进入工作模式，说明 CAN 控制器与 CPU 连接正常。

在编写 CAN 总线通讯程序时，主要编写函数由初始化函数，发送函数和接收函数组成。 其中发送函数一般写为主动发送函数，接收函数一般采用中断接收。下面就简单介绍一下三个函数的编写。

### 1. SJA1000寄存器初始化配置顺序

（1）进入复位模式，进行配置
（2）配置时钟分频寄存器，决定 Peil 模式还是 Basic 模式。
（3）配置总线定时寄存器，确定波特率。
（4）配置中断使能寄存器，决定使用那几个中断
（5）配置输出控制寄存器
（6）配置验收码和屏蔽码，决定接收哪一类节点的数据
（7）退出复位模式，进入正常工作模式。

### 2. 发送数据顺序

（1）查询状态寄存器，判断是否正在接收，是否正在发送，是否数据缓冲区被锁。
（2）配置发送缓冲区。
（3）配置命令寄存器，启动发送。

### 3. 接收数据顺序

（1）采用中断接收，关 CPU 中断。
（2）判断是不是接收中断。
（3）判断是远程帧还是数据帧
（4）取数据
（5）开中断

## CAN 总线活动讲座十七：STC 单片机程序下载软件的使用

我想对于 STC 单片机程序的下载烧写，很多朋友都很熟悉了，无论是 wangjin 的51开发板还是圈圈的 USB 开发板，都用的是 STC 单片机，它的特点和优点在前面的课程中已经说过了，这里仅简单的把下载的过程提一下。 接下来我们就要开始学习如何利用板子来学习，我们还是按照以往一样，一个程序一个程序的来。这一节主要介绍一些相关软件的使用方法。我们这里主要用到2个软件，一个是 KeilC51,一个是 STC 单片机的下载软件。前者是开发环境，后者的烧录程序。当然用 wave 开发软件也可以。

下载地址：
http://www.mcu-memory.com

## CAN 总线活动讲座十八：数码管显示实验

估计很多朋友们已经或者快要拿到板子了。最近我们也把程序整理的差不多了。接下来逐步公布出来，供大家学习探讨交流！ 你也许是个高手，也许是个单片机的初学者，但我相信，在拿到板子的一开始的工作都是熟悉硬件资源，熟悉每部分的电路，怎么编程控制。所以我们就从 CAN 总线学习板上比较简单的开始，公布实验程序，一帮助大家熟悉硬件，二有助于逐渐熟悉我们的程序。那就先开始第一个实验——数码管显示实验。

### 实验内容

数码管显示数字，而数字从 N 开始移动到第四个数码管。

### 实验目的

熟悉 CAN 总线学习板的数码管显示电路；熟悉数码管显示编程。能够熟练应用板子上的数码管，为后续的实验打好基础！

### 相关点评

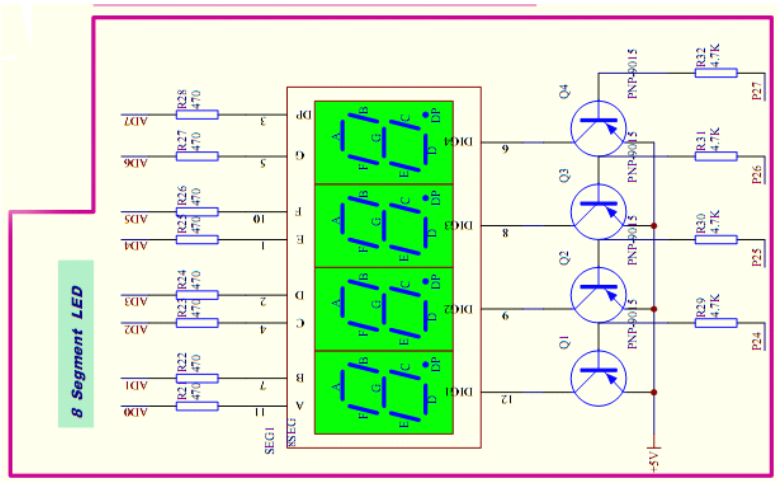
采用三极管反向驱动，降低成本，减轻大家的负担。
数码管采用共阳极数码管，P0口为数码管的字形控制，P24~P27为数码管的字位控制。
当 P24~P27位为低电平，数码管的字形控制也为低电平时，对应的里面的 LED 灯亮。

### 实验步骤和实验现象与结果

（1）检查电路，判断电路是否短路。
（2）打开电源开关，下载程序。下载程序时观察软件上显示是否下载成功。
（3）下载完毕后，数码管显示255。

### 实验注意事项

实验前一定要检查电路，防止短路。




| 数码管码(低电平) | 0xc0 | 0xf9 | 0xa4 | 0xb0 | 0x99 |
|---|---|---|---|---|---|
| 符号 | 0 | 1 | 2 | 3 | 4 |
| 数码管码(低电平) | 0x92 | 0x82 | 0xf8 | 0x80 | 0x90 |
| 符号 | 5 | 6 | 7 | 8 | 9 |

0xff 为无显示

源码： （声明：程式源 PIAE 修改致 29-09-09 akuei2）

```
#include <reg52.h>
#include <intrins.h>
#define uchar unsigned char                                    //对管脚分配进行了定义
#define uint  unsigned int                                     //对子函数进行了声明
#define NOPS  {_nop_();_nop_();_nop_();_nop_();_nop_();} //延时5us

//编码规则是 hgfedcba ,h 亦是 dp，控制小数点，这里都设为1，不亮，eg：9==0b10010000；
uchar code led[] = {0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0xff};
uchar Show_Data = 0xff;//数码管要显示的数据

sbit      LED_RED     =  P2^1;//红色指示灯,作为接收指示灯
sbit      LED_GRE     =  P2^2;//绿色指示灯,作为发送指示灯

void Delay(uchar delay_time); //延时程序声明
void led_seg7(uchar from,uchar number); //数码管显示程序声明

//主函数
void main(void)
{
    _nop_();
    while(1)
    {
        NOPS;
        led_seg7(1,Show_Data);
        LED_RED = !LED_RED;
```

```
        LED_GRE = !LED_GRE;
    }
}

//延时程序
void Delay(uchar delay_time)
{
    while(delay_time--);
}

//from(1_4)：数码管显示起始位置（从右到左），number：显示的数 ,eg: leg_seg7(1,255)
void led_seg7(uchar from,uchar number)
{
     uchar digit,temp_l;
     uchar temp_h=0x7f;                  //0b0111111
     temp_h = _cror_(temp_h,from-1);    //确定从哪一位开始显示，即确定高四位,eg:第二位开始，temp_h==0b01111111
     temp_h = temp_h & 0xf0;            //取高四位， temp_h==0b01110000
     temp_l = P2 & 0x0f;                //取 P2的低四位（c51默认下都是高平）,temp_l==0b00001111
     P2 = temp_h | temp_l;              //设定 P2口 P2==0b0111111|0b00001111,P2==0b01111111，0xbf，第二位打开；

    if(number==0)                        //如果 number==0的话， 立即发送0的数码管码
    {
        P0 = led[0];
        Delay(10);
        P0 = 0xff;
    }
    else
    {
        while(number)                     //如果数字式大于0， 就永远死循环， eg:number==255
        {
         digit = number%10 ;              //eg:digit=255%10,digit==5;
              number /= 10;               //eg:number=255/10,number==25;
              P0 = led[digit] ;           //送数码管码， eg:5亦是0x92
              Delay(10);
              temp_h = P2 & 0xf0;                 //取 P2的高四位， eg:temp_h=0=0b0111000；
              temp_h = temp_h | 0x0f;             //拼装 temp_h，进行位选， eg:temp_h==0b0111111
              temp_h = _cror_(temp_h,1);           //eg:temp_h==0b10111111；
              temp_h = temp_h & 0xf0;             //取高四位， eg:temp_h==0b10110000
              temp_l = P2 & 0x0f;                 //取 P2的低四位
                                                  //eg:temp_l==0b011111&0b00001111,temp_l==0b00001111;
              P0 = 0xff;                    //数码管清除
              P2 = temp_h | temp_l;    //设定 P2口，eg：P2==0b10110000|0b00001111 P2==0b10111111；
        }
    }
}
```

## 实验内容

经典的定时器实验。主要利用定时器0，将每秒累加的数目显示在数码管上;

源码

```
// 02-定时器.c
// 29-09-09 akuei2;

#include "reg52.h"
#define uchar unsigned char

//定义数码管码
uchar code Led_Code[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0xff};

//定义位选码
//0b11101111,0b11011111,0b10111111,0b01111111
uchar code Led_Select[]={0xef,0xdf,0xbf,0x7};

//定义变量
int Sec,t0;

//50微秒延迟函数
void Delay_50us(int t)
{
	uchar j;
	t--;
	for(;t>0;t--)
		for(j=19;j>0;j--);

}

//数码管显示函数
void Display(void)
{
	int Digit[4],i;
	Digit[3]=Sec/1000;		//第一个数码管取千位
	Digit[2]=Sec%1000/100;	//第二个数码管取百位
	Digit[1]=Sec%100/10;	//第三个数码管取十位
	Digit[0]=Sec%10;		//第四个数码管取个位
	for(i=0;i<4;i++)
	{
		P0=Led_Code[Digit[3-i]];	//送数码管码
		P2=Led_Select[i];		//送位选码
		Delay_50us(20);			//延迟1微秒
		P0=P2=0xff;			//消影
```

```
    }
}

//初始化函数
void Init(void)
{
    TMOD=0x01;                  //定时0，工作模式1
    TH0=(65536-(50000))/256;    //定时器0，高四位赋值
    TL0=(65536-(50000))%256;    //定时器0，第四位赋值，时间大约是50毫秒
    EA=1;                       //全能中断使能
    ET0=1;                      //定时器0中断使能
    TR0=1;                      //定时0启动
}

//主函数
void main(void)
{
    Init();                     //初始化函数
    while(1)
    {
        Display();              //显示函数
    }
}

//定时器0 中断函数
void Ir_t0(void) interrupt 1
{
    TH0=(65536-(50000))/256;  //重新赋值
    TL0=(65536-(50000))%256; //重新赋值
    t0++;
    if(t0>=20)                //大约1秒
    {
        t0=0; Sec++;          //Sec 增值
        if(Sec>=10000) Sec=0; //当 Sec 大于等于10000，Sec 重新赋值
    }


}
```

## 实验内容

INT0按键为外部中断计数按键，将计数结果在数码管上显示。

## 实验目的

熟悉总线学习板上的硬件资源，熟悉外部中断计数程序和显示程序编程。

## 相关点评

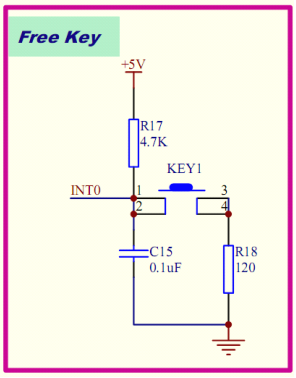
按键增加硬件消抖，使计数准确，不容易出现按一下按键，计多个数值的情况。

## 实验步骤和实验现象与结果

（1）检查电路，判断电路是否短路。
（2）打开电源开关，下载程序。下载程序时观察软件上显示是否下载成功
（3）下载完毕后，数码管显示0
（4）按 INT0按键，计数增加1，数码管显示增加1（十进制显示）。

## 实验注意事项

（1）实验前一定要检查电路，防止短路




原理图：硬件消抖

源程式：（无修改）
ext_int0_def.h 文件

```
//对管脚分配进行了定义
//对子函数进行了声明

# define uchar unsigned char
# define uint    unsigned int

# define NOPS       {_nop_();_nop_();_nop_();_nop_();_nop_();}/*延时5us*/
uchar code led[] = {0xc0,0xf9,0xa4,0xb0,    // 0, 1, 2, 3
                    0x99,0x92,0x82,0xf8,0x80,0x90, 0xff};// 4, 5, 6, 7, 8, 9, off
                     //编码规则是 gfedcba ,其中 g 为小数点，控制 dp，这里都设为1，不亮
```

```
uchar           Show_Data = 0x00;       //数码管要显示的数据，也是要发送的数据。

/*IO  口分配*/
sbit        LED_RED       =   P2^1;    //红色指示灯,作为接收指示灯
sbit        LED_GRE       =   P2^2;    //绿色指示灯,作为发送指示灯
sbit      SJA1000_CS    = P2^0;         //SJA1000片选管脚. 低电平有效

void Delay(uchar delay_time);           //延时程序
void CPU_init(void);                    //CPU 初始化子函数
void led_seg7(uchar from,uchar number);
```

==========

ext_int0.C 文件

```
#include <reg52.h>
#include <intrins.h>
#include <ext_int0_def.h>

void INT0_Counter( void ) interrupt 0 using 1
{//INT0为计数按键
    EA = 0;
    Show_Data++;                /计数单元加1
    EA = 1;
}

void main(void)
{
    CPU_init();
    _nop_();
    while(1)
    {
        NOPS;
//////////////////////数码管显示程序////////////////////////////////
        led_seg7(1,Show_Data);
        LED_RED = !LED_RED;
        LED_GRE = !LED_GRE;

//////////////////////数码管显示程序////////////////////////////////
    }//while  结束
}//main 结束


void Delay(uchar delay_time)
{//延时程序
    while(delay_time--)
    {}
}
```

```
void CPU_init(void)
{//初始化 CPU
    IT0 = 1;                    //外部中断0负边沿触发
    EX0 = 1;                    //打开外部中断0
    EA = 1;                     //打开总中断
    SJA1000_CS = 1;             //片选无效
}

void led_seg7(uchar from,uchar number)   //from(1_4)：数码管显示起始位置（从右到左），number：显示的数
{
    uchar digit,temp_l;
    uchar temp_h=0x7f;
    temp_h = _cror_(temp_h,from-1);         //确定从哪一位开始显示，即确定高四位
    temp_h = temp_h & 0xf0;                 //取高四位
    temp_l = P2 & 0x0f;                     //取 P2的低四位
    P2 = temp_h | temp_l;                   //设定 P2口

    if(number==0)
    {
        P0 = led[0];
        Delay(5);
        P0 = 0xff;
    }
    else
    {
        while(number)
    {
            digit = number%10 ;
        number /= 10;
            P0 = led[digit] ;
        Delay(5);
            temp_h = P2 & 0xf0;                 //取 P2的高四位
            temp_h = temp_h | 0x0f;             //拼装 temp_h，进行位选
        temp_h = _cror_(temp_h,1);
        temp_h = temp_h & 0xf0;                 //取高四位
            temp_l = P2 & 0x0f;                 //取 P2的低四位
            P0 = 0xff;
            P2 = temp_h | temp_l;                //设定 P2口
        }//while 结束
    }//else 结束

}
```

加了自己练习用的程式。

源码：

```
//     01-外部中断.c
//     记录外部中断的次数（外部中断的次数）,然后显示在数码管
//     29-09-09 akuei2

#include "reg52.h"
#define uchar unsigned char

sbit       SJA1000_CS    = P2^0;    //SJA1000片选管脚. 低电平有效

//定义数码管码
uchar code Led_Code[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0xff};

//定义位选码
//0b111011111,0b11011111,0b10111111,0b01111111
uchar code Led_Select[]={0xEF,0xDF,0xBF,0x7F};

int Click;

//50微秒延迟函数
void Delay_50us(int t)
{
       uchar j;
       t--;
       for(;t>0;t--)
              for(j=19;j>0;j--);

}

//1毫秒延迟函数
void Delay_1ms(int t)
{
       uchar j;
       for(;t>0;t--)
              for(j=112;j>0;j--);

}

//数码管显示函数
void Display(void)
{
       int Digit[4],i;
       Digit[3]=Click/1000;                    //第一个数码管取千位
```

```
    Digit[2]=Click%1000/100;        //第二个数码管取百位
    Digit[1]=Click%100/10;          //第三个数码管取十位
    Digit[0]=Click%10;              //第四个数码管取个位
    for(i=0;i<4;i++)
    {
        P0=Led_Code[Digit[3-i]];        //送数码管码
        P2=Led_Select[i];               //送位选码
        Delay_50us(20);                 //延迟1微秒
        P0=P2=0xff;                     //消影
    }
}

//初始化函数
void Init(void)
{
    IT0 = 1;                //外部中断0负边沿触发
    EX0 = 1;                //打开外部中断0
    EA = 1;                 //打开总中断
    SJA1000_CS = 1;         //片选无效
}

//主函数
void main(void)
{
    Init();
    while(1)
    {
        Display();
    }
}

//外部0中断函数
void Ir_ex0(void) interrupt 0
{
    EX0=0;EA=0;                     //关闭外部中断0，关闭全局中断使能
    Delay_1ms(10);                  //消抖
    Click++;                        //记录按键次数
    if(Click>=10000) Click=0;       //如果案件超过一万次就回归0
    EX0=1;EA=1;                     //开启外部中断0，开启全局中断使能
}
```

备注：

既然已经设置了硬件消抖，可能是出于习惯吧，总是加了软件消抖才安心，啊哈哈！哪行有点多此一举，可以无视！

## CAN 总线活动讲座二十：串口通讯实验

### 实验内容

INT0按键为外部中断计数按键，每按一次，数码管1-2（从右至左）显示数据加1，通过串口发送当前计数结果给 PC 机； PC 机发送一个字节的数据，在数码管3-4上显示。

### 实验目的

熟悉总线学习板上的硬件资源，熟悉中断计数程序和显示程序，学习串口通讯编程。

### 相关点评

检查 RS232芯片的好坏。芯片正常时，2脚电平在9V 左右，6脚在-9V 左右。

### 实验步骤和实验现象与结果

（1）检查电路，判断电路是否短路。

（2）打开电源开关，下载程序。下载程序时观察软件上显示是否下载成功。

（3）下载完程序后，数码管1-2显示0，数码管3-4可能显示不是0。

（4）按一下复位键，让程序重新运行。数码管1-2显示0，数码管3-4显示0。

（5）关闭下载程序软件，打开串口调试精灵。

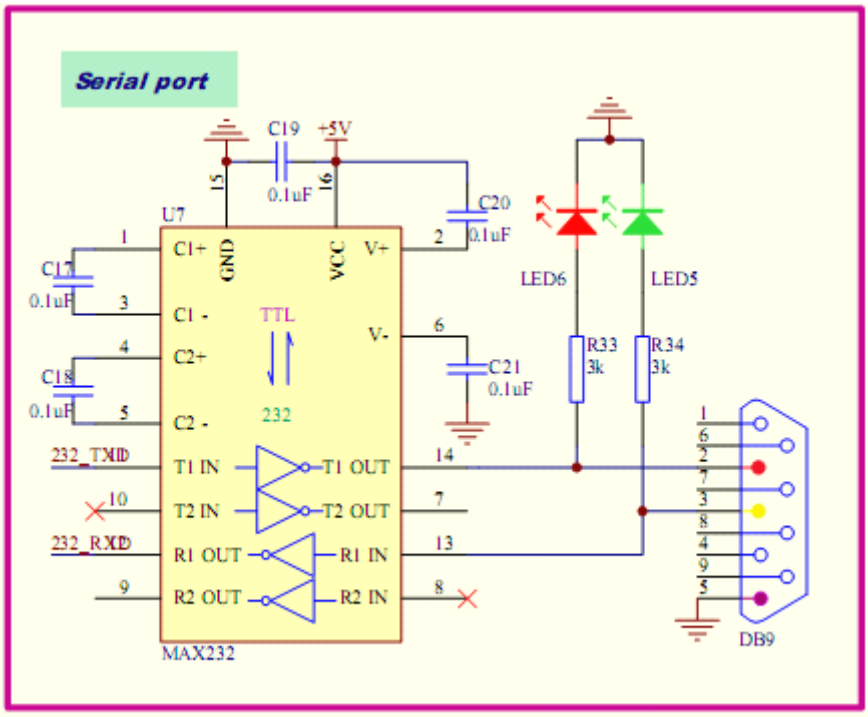
（6）按 INT0按键一次，数码管1-2显示加1（十进制显示），自动发送数据给 PC 机。

（7）通过串口调试精灵发送一个字节的16进制数，数码管3-4显示改变。

### 实验注意事项

（1）实验前一定要检查电路，防止短路

（2）程序下载程序和串口调试精灵不能同时打开，避免串口冲突。




原理图：串口的硬件连接

先来最基本的串口测试程式

源程式：

```
//     00-串口测试程式(发送).c
//     不停的发送1
//     29-09-09 akuei2

#include "stdio.h"
#include "reg52.h"
#define uchar unsigned char

//1毫秒延迟函数
void Delay_1ms(int x)
{
      int j;
      for(;x>0;x--)
            for(j=112;j>0;j--);
}

//串口发送函数
void Txd(void)
{
      ES=0;                   // 关闭串口中断，采用查询发送方式
      TI=1;                   // 发送结束标志位置一
      while(TI)
      {
            /*
            T1=0;             // 如果不喜欢 printf 函数，可以注释掉它
            SBUF=0x31;        // 然后去除 ** 就可以使用典型的发送 eg: 0x31 ， "1" 的 ascii 码
            */
            printf("1");      // 使用 printf 函数，发送数据
            while(!TI);       // 等待发送
            TI=0;
      }
      ES=1;                   // 打开串口中断
}

//初始化函数
void Init(void)
{
      TMOD=0x20;                    //   T1工作模式2,T0工作模式1
      SM0=0;SM1=1;                  // 设置串口的工作模式
      TH1=0xfd;                     // 给 TH1赋值,决定赋给 TL1的值
      TL1=0xfd;                     // 给 TL1赋值决定波特率9600kb/s
```

```
    TR0=1;          // 定时器0启动
    ET0=1;
    TR1=1;          // 定时器1启动

    IT0 = 1;        //外部中断0负边沿触发
    EX0 = 1;        //打开外部中断0

    REN=0;          // 不应许串口接收数据
    EA=1;           // 开启中断
    ES=1;           // 串口中断应许

}

//主函数
void main(void)
{
    Init();                     // 调用初始化函数
    while(1)
    {
        Txd();                  //不停的发送
        Delay_1ms(500);         //延迟500毫秒
    }
}
```

源程式：

```
//    01-串口测试程式(接收).c
//    接收来至串口的数据，对应每一个 ASCII 码，然后显示相对应的数字
//    29-09-09 akuei2

#include "stdio.h"
#include "reg52.h"
#define uchar unsigned char

//定义数码管码
uchar code Led_Code[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0xff};

//定义位选码
//0b111011111,0b11011111,0b10111111,0b01111111
uchar code Led_Select[]={0xEF,0xDF,0xBF,0x7F};

uchar Buffer_RXD,Number,N,Temp;

//50微秒延迟函数
void Delay_50us(int t)
{
```

```
    uchar j;
    t--;
    for(;t>0;t--)
        for(j=19;j>0;j--);
}

//数码管显示函数
void Display(void)
{
    P0=Led_Code[Number];                // 送数码管码
    P2=Led_Select[3];                   // 送位选码
    Delay_50us(20);                     // 延迟1微秒
    P0=P2=0xff;                         // 消影
}

//接收数据对应数字函数
void Edit_Number(void)
{
    if(Temp!=Buffer_RXD)                // 如果接受的数据与前一个数据不一样...
    {
        switch(Buffer_RXD)
        {
            case '0': Number=0; break;
            case '1': Number=1; break;
            case '2': Number=2; break;
            case '3': Number=3; break;
            case '4': Number=4; break;
            case '5': Number=5; break;
            case '6': Number=6; break;
            case '7': Number=7; break;
            case '8': Number=8; break;
            case '9': Number=9; break;
        }
    }
}

//初始化函数
void Init(void)
{
    TMOD=0x20;                      // T1工作模式2
    SM0=0;SM1=1;                    // 设置串口的工作模式
    TH1=0xfd;                       // 给 TH1赋值,决定赋给 TL1的值
    TL1=0xfd;                       // 给 TL1赋值决定波特率9600kb/s

    TR0=1;      // 定时器0启动
    ET0=1;
```

```
    TR1=1;          // 定时器1启动

    IT0 = 1;        // 外部中断0负边沿触发
    EX0 = 1;        // 打开外部中断0

    REN=1;          // 不应许串口接收数据
    EA=1;           // 开启中断
    ES=1;           // 串口中断应许

    Number=0;
    Buffer_RXD=0;

}

//主函数
void main(void)
{
    Init();                 // 调用初始化函数
    while(1)
    {
        Display();      // 数码管不停的显示
    }
}

//串口接收中断
void Ir_RXD(void) interrupt 4
{
    ES = 0;                         // 关闭串口
    if(RI==1)
    {
        RI = 0;
        Buffer_RXD = SBUF;
        Temp=Buffer_RXD;

    }
    ES = 1;                         // 打开串口
    Edit_Number();                  // 调用函数

}
```

这个是 PIAE 的源程式又接受和输出

源程式： （无修改）

RS232_def.h 文件

```
//对管脚分配进行了定义
//对子函数进行了声明

# define uchar unsigned char
# define uint   unsigned int

# define NOPS        {_nop_();_nop_();_nop_();_nop_();_nop_();}/*延时5us*/
uchar code led[]   = {0xc0,0xf9,0xa4,0xb0,  // 0, 1, 2, 3
                      0x99,0x92,0x82,0xf8,0x80,0x90, 0xff};// 4, 5, 6, 7, 8, 9, off
                          //编码规则是 gfedcba ,其中 g 为小数点，控制 dp，这里都设为1，不亮


bit             RXD_flag = 0;             //收到数据标志； 0 无  ， 1 有
bit             TXD_flag = 0;             //需要发送数据标志； 0 不  ， 1 要
uchar           RX_data   = 0x00;         //接收的数据
uchar           TX_data   = 0x00;         //发送的数据
uchar           Show_TX_Data = 0x00;      //数码管要显示的发送数据，在数码管1-2上显示（从右至左）。
uchar           Show_RX_Data = 0x00;      //数码管要显示的接收数据，在数码管3-4上显示（从右至左）。

/*IO  口分配*/
sbit        LED_RED        =   P2^1;          /红色指示灯,作为接收指示灯
sbit        LED_GRE        =   P2^2;          //绿色指示灯,作为发送指示灯
sbit       SJA1000_CS    = P2^0;             //SJA1000片选管脚. 低电平有效

void Delay(uchar delay_time);                 //延时程序
void CPU_init(void);                          //CPU 初始化子函数
void RS232_TXD( void );                       //IRDA 发送子函数
void led_seg7(uchar from,uchar number);       //数码管显示子程序
```

==========

RS232_com.c 文件

```
#include <reg52.h>
#include <intrins.h>
#include <RS232_def.h>



void INT0_Counter( void ) interrupt 0 using 1
{//INT0按键为计数按键
      EA = 0;
      Show_TX_Data++;             //计数单元加1
```

```
    TXD_flag = 1;                   //置要发送标志位
    EA = 1;
}


void RS232_RXD( void ) interrupt 4      using 1
{//接收数据函数，在中断服务程序中调用
    EA = 0;                         //关闭所有中断
    ES = 0;                         //关闭串口
    if(RI==1)
    {
        RI = 0;
        RX_data = SBUF;
        RXD_flag = 1;       //置接收到数据标志
    }
    _nop_();
    ES = 1;//打开串口
    EA = 1;//打开中断
}
//******************************************************

void main(void)
{
    CPU_init();
    _nop_();
    while(1)
    {
        NOPS;
////////////////////////  接收处理程序////////////////////////////////
        if( RXD_flag )
        {
            EA = 0;                         //关闭 CPU 中断
            LED_RED = !LED_RED;     //指示灯状态变化
            RXD_flag = 0;                   //清除标志
            Show_RX_Data = RX_data;  //接收对方发送的数据
            EA = 1;
        }
//////////////////////发送处理程序////////////////////////////////
        if( TXD_flag )
        {
            EA = 0;
            LED_GRE = !LED_GRE;//指示灯状态变化
            TXD_flag = 0;
            TX_data    = Show_TX_Data;//给发送的数据赋值
            RS232_TXD();
            _nop_();
```

```
                EA = 1;
            }
//////////////////////数码管显示程序////////////////////////////////
            led_seg7(1,Show_TX_Data);
            led_seg7(3,Show_RX_Data);
//////////////////////数码管显示程序////////////////////////////////
      }//while  结束
}//main 结束


void Delay(uchar delay_time)
{//延时程序
      while(delay_time--)
      {}
}


void CPU_init(void)
{//初始化 CPU
      SCON = 0x50;      //串口方式1
      PCON = 0x80;      //串口波特率加速
      TMOD = 0x21;
      TH1=0xFD;         //19200bps
      TL1=0xFD;
      TR1 = 1;
      TI = 0;
      RI = 0;
      PS = 1;           //串口中断的优先级设为最高

      IT0 = 1;          //外部中断0负边沿触发
      EX0 = 1;          //打开外部中断0

      ES  = 1;          //打开串口中断
      EA = 1;           //打开总中断
      SJA1000_CS = 1;   //片选无效
}

void RS232_TXD( void )
{
      EA = 0;       //关闭所有中断
      ES = 0;       //关闭串口中断，采用查询发送方式
      TI = 1;
      while(TI)
      {
            TI = 0;    //一字节发送完后清除标志位
            SBUF = TX_data;
```

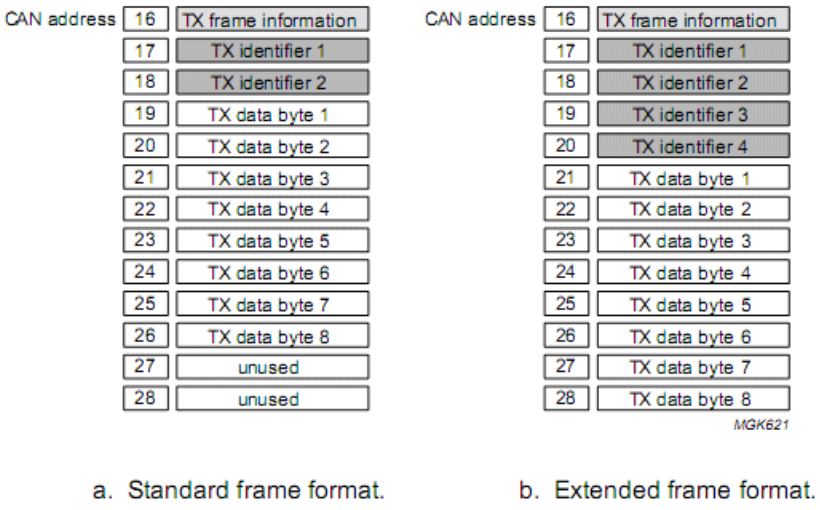
```
        while(!TI)
        {}//等待发送
        TI = 0;
    }
    _nop_();
    ES = 1;   //打开串口中断
    EA = 1;   //打开中断
}

void led_seg7(uchar from,uchar number)   //from(1_4)：数码管显示起始位置（从右到左），number：显示的数
{
    uchar digit,temp_l;
    uchar temp_h=0x7f;
    temp_h = _cror_(temp_h,from-1);     //确定从哪一位开始显示，即确定高四位
    temp_h = temp_h & 0xf0;               //取高四位
    temp_l = P2 & 0x0f;                   //取 P2的低四位
    P2 = temp_h | temp_l;                 //设定 P2口

    if(number==0)
    {
        P0 = led[0];
        Delay(5);
        P0 = 0xff;
    }
    else
    {
        while(number)
    {
            digit = number%10 ;
        number /= 10;
            P0 = led[digit] ;
        Delay(5);
            temp_h = P2 & 0xf0;                 //取 P2的高四位
            temp_h = temp_h | 0x0f;             //拼装 temp_h，进行位选
        temp_h = _cror_(temp_h,1);
        temp_h = temp_h & 0xf0;                 //取高四位
            temp_l = P2 & 0x0f;                 //取 P2的低四位
            P0 = 0xff;
            P2 = temp_h | temp_l;               //设定 P2口
        }//while 结束
    }//else 结束

}
```

**前言：**

如前面在第十四讲座那里的介绍 SJA1000，独立控制器支持 CAN2.0B 协议，换句话说同样也支持 PeliCAN 的工作模式，在 PIAE 的实验中都是以 PeliCAN 模式为基础，而且 PeliCAN 模式也兼容 BasicCAN 工作模式， 先了解 PeliCAN 的工作模式就足够了，明白了 PeliCAN 以后再明白 BasicCAN 吧。

**PeliCAN 工作模式特征简单介绍：**

在 BasicCAN 模式中（工作模式），报文的长度是11位，而 PeliCAN 模式下的报文是27位，前者是 SFF-标准帧格式，后者是 EFF-扩展帧格式，而一个报文的格式是由 TX 帧信息来决定的（发送帧信息）。以下是 SSF 和 EFF 的基本配置 。(注：TX Frame Information ，TX 帧信息)




a. Standard frame format. b. Extended frame format.

为什么说 TX 帧信息决定了报文的格式？让我们来了解 TX 帧信息的结构

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| FF(1) | RTR(2) | X(3) | X(3) | DLC.3(4) | DLC.2(4) | DLC.1(4) | DLC.0(4) |

在 TX 帧信息的 MSB，也就是 BIT 7 是 FF，FF 的设置取决了报文的格式，当：

| | |
|---|---|
| 0 | 为标准格式 |
| 1 | 为扩展格式 |

这样就明白了吗？至于 RTR，DLC.0~3，是什么东西？RTR 是远程请求，而 DLC.0~3决定了数据的长度，这个后面会慢慢的介绍的，不要着急！学习是慢慢来的。

好了，休息一会儿吧，好让大脑吸收，我也是用了几天的时间才渐渐的消化！

休息好了，继续吧！

**什么是标示符，数据与数据长度？**

表示符：

表示符好比是一个电子邮件的地址，如你要给 PIAE 发一封电邮，电邮的格式是 TX 帧信息来决定，是普通邮件，垃圾邮件，还是 SPAM 邮件，这些都是 TX 帧信息的工作！那么电邮的地址，如 PIAE2008@163.com 就是标示符了，标示符的定义是决定“这个报文是给谁的？”（由于 PIAE 的实验都是以 PeliCAN 为基础，BasicCAN 的报文格式就不介绍了，毕竟 PeliCAN 可以兼容 BasicCAN）

那么先让我们来了解标示符的结构吧

**Table 29** TX identifier 1 (EFF); CAN address 17; note 1

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| ID.28 | ID.27 | ID.26 | ID.25 | ID.24 | ID.23 | ID.22 | ID.21 |

**Table 30** TX identifier 2 (EFF); CAN address 18; note 1

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| ID.20 | ID.19 | ID.18 | ID.17 | ID.16 | ID.15 | ID.14 | ID.13 |

**Table 31** TX identifier 3 (EFF); CAN address 19; note 1

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| ID.12 | ID.11 | ID.10 | ID.9 | ID.8 | ID.7 | ID.6 | ID.5 |

**Table 32** TX identifier 4 (EFF); CAN address 20; note 1

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| ID.4 | ID.3 | ID.2 | ID.1 | ID.0 | $X^{(2)}$ | $X^{(3)}$ | $X^{(3)}$ |

从地址17~20，其中有32位，而其中3位是无用的状态就是 BIT0~BIT2，从 BIT3开始是 ID.0，到地址17的 BIT 7 是 ID.28。如果你问我地址是什么用的，呵呵好戏在后头，在这里你只要先理解 SSF 报文的标示符是又四个字节组成，而其中有29为被用到。这就是为什么说 SSF-扩展格式中，有27位，但是从我的理解上似乎少了一个为，如果你问我为什么，呵呵我也是知道个大概而已，总之有奶的就是娘了，就先这样理解吧。

数据与数据长度：

数据的定义报文中要发送的信息，如电邮里边的内容，至于数据长度是告诉别人“你的数据有多长”，那么数据的长度是如何设置的？依然是浏览 TX 帧信息的结构

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| $FF^{(1)}$ | $RTR^{(2)}$ | $X^{(3)}$ | $X^{(3)}$ | $DLC.3^{(4)}$ | $DLC.2^{(4)}$ | $DLC.1^{(4)}$ | $DLC.0^{(4)}$ |

TX 帧信息中的低四位也就是 BIT0~BIT3，它们决定了报文中数据的长度。报文的长度是又公式计算的

$$DataByteCount = 8 \times DLC.3 + 4 \times DLC.2 + 2 \times DLC.1 + DLC.0$$

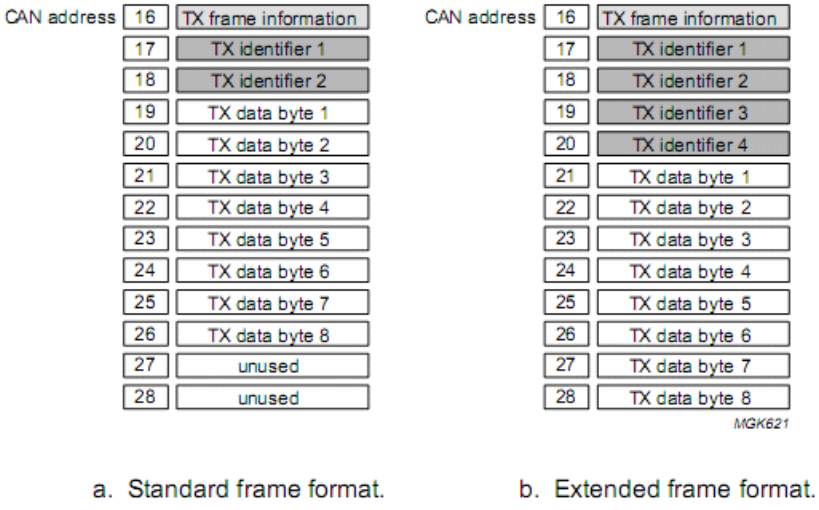
但是有一个很遗憾的事实，目前大于8的数据长度代码是不可用，如果数据的长度大于8，那么依然是以8个字节来计算，换句话说就是前八个字节数据有效，第八个字节以后的数据都无视了

问题来了：“设置数据的长度有什么用？”，见笑了，小的不才以目前的理解程度，我只知道，当有两个 CAN 控制器同时以同样的标示符来发送报文，那么唯一可以区分的就是数据长度的预先设置，这个可以进一步的保障，接收方不会混乱于要接收哪一个拥有同样标示符报文呢？如此，可以避免总线错误.

Eg:假设 A 同时间给 PIAE 发送两封电邮，由于不同内容的电邮有不同的大小，那么 A 只要事先通知 PIAE 说：“我电邮给你的邮件是有包含了黏贴物”，那么他就会知道，A 要 PIAE 就收的邮件就是那个内容比较大的电邮了”

**总结：**

总结一下




a. Standard frame format.　　b. Extended frame format.

这回我们明白了，SSF 和 EFF 到底有什么不一样，而 TX 帧又是干什么用的了。我们来了解一个列子吧：

Eg：TBSR=0x88

TBSR 是报文的第一个字节，也就是 TX 帧信息，而0x88，就是将高四位的最高位，和低四位的最高位设置为1，也就是 BIT7=1，BIT3=1。

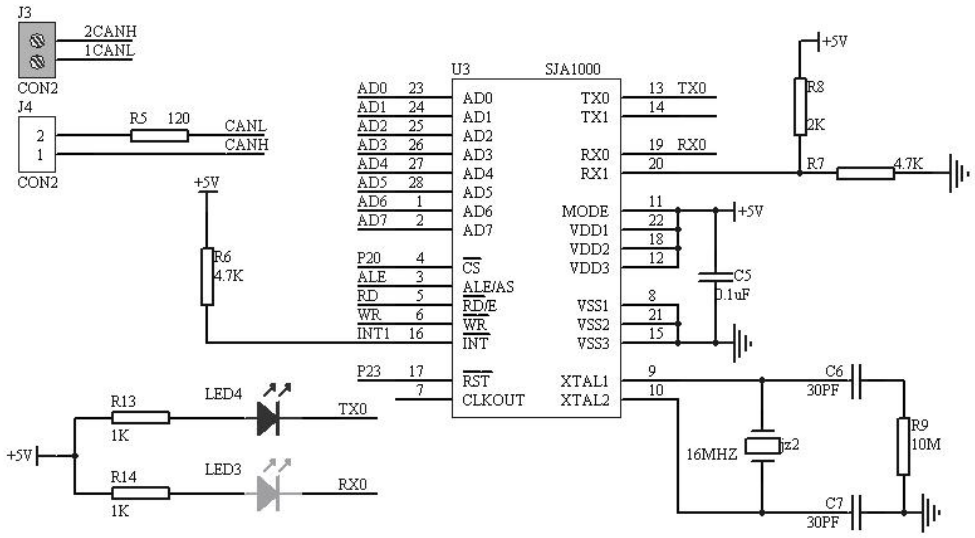
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FF=1 | RTR=0 | X | X | DLC.3=1 | DLC.2=0 | DLC.1=0 | DLC.0=0 |

这个报文就是一个“EFF-扩展帧格式”，“非远程请求”，“数据长度为8个字节”

看到这里基本上你已经掌非常非常基本的概念了，再休息一回吧，好让大脑消化，
是时候我肚子也饿了，用餐过后过后再继续吧。

## 个人秀零二：sja1000.h 头文件简介

要开始写程式之前还是先了解 sja1000.h 都文件的定义， 由于 sja1000是独立的 CAN 控制，所以内部的寄存器都是以 CPU 从外部读取的，回顾一下原理图：




也就是 AD0~7直接连接向89c5X 的 P0口。了解 CPU 如何访问 SJA1000的寄存器后，先来看看 sja1000.h 头文件：

Sja1000.h 头文件

```
/*CAN 总线 SJA1000寄存器地址定义（用的是 PeliCAN 模式，扩展帧 EFF 模式）*/
#define uchar unsigned char

uchar   xdata   MODR    _at_   0xFE00;      // 模式寄存器
uchar   xdata   CMR     _at_   0xFE01;      // 命令寄存器
uchar   xdata   SR      _at_   0xFE02;      // 状态寄存器
uchar   xdata   IR      _at_   0xFE03;      // 中断寄存器
uchar   xdata   IER     _at_   0xFE04;      // 中断使能寄存器
uchar   xdata   BTR0    _at_   0xFE06;      // 总线定时寄存器0 ； 总线波特率的选择
uchar   xdata   BTR1    _at_   0xFE07;      // 总线定时寄存器1 ； 总线波特率的选择
uchar   xdata   OCR     _at_   0xFE08;      // 输出控制寄存器
uchar   xdata   ACR     _at_   0xFE10;//16;
uchar   xdata   ACR1    _at_   0xFE11;//17;
uchar   xdata   ACR2    _at_   0xFE12;//18;
uchar   xdata   ACR3    _at_   0xFE13;//19;      // 接收代码（0x16_0x19）；接收过滤位选择*复位模式
uchar   xdata   AMR     _at_   0xFE14;//20;
uchar   xdata   AMR1    _at_   0xFE15;//21;
uchar   xdata   AMR2    _at_   0xFE16;//22;
uchar   xdata   AMR3    _at_   0xFE17;//23;      // 掩码（0x20_0x23）；接收过滤的选择*复位模式

uchar   xdata   CDR   _at_   0xFE1F;//31;      // 时钟分频器
uchar   xdata   ALC   _at_   0xFE0B;//11;      // 仲裁丢失捕捉寄存器
uchar   xdata   ECC   _at_   0xFE0C;//12;      // 误码捕捉寄存器
```

```
uchar   xdata   TBSR     _at_   0xFE10;//16; //TX  帧信息(标准帧、扩展帧）寄存器
uchar   xdata   TBSR1    _at_   0xFE11;//17; //TX  帧信息(标准帧数据1、扩展帧识别码1）寄存器
uchar   xdata   TBSR2    _at_   0xFE12;//18; //TX  帧信息(标准帧数据1、扩展帧识别码2）寄存器
uchar   xdata   TBSR3    _at_   0xFE13;//19; //TX  帧信息(标准帧数据1、扩展帧识别码3）寄存器
uchar   xdata   TBSR4    _at_   0xFE14;//20; //TX  帧信息(标准帧数据2、扩展帧识别码4）寄存器
uchar   xdata   TBSR5    _at_   0xFE15;//21; //TX  帧信息(标准帧数据3、扩展帧数据1）寄存器
uchar   xdata   TBSR6    _at_   0xFE16;//22; //TX  帧信息(标准帧数据4、扩展帧数据2）寄存器
uchar   xdata   TBSR7    _at_   0xFE17;//23; //TX  帧信息(标准帧数据5、扩展帧数据3）寄存器
uchar   xdata   TBSR8    _at_   0xFE18;//24; //TX  帧信息(标准帧数据6、扩展帧数据4）寄存器
uchar   xdata   TBSR9    _at_   0xFE19;//25; //TX  帧信息(标准帧数据7、扩展帧数据5）寄存器
uchar   xdata   TBSR10   _at_   0xFE1A;//26; //TX  帧信息(标准帧数据8、扩展帧数据6）寄存器
uchar   xdata   TBSR11   _at_   0xFE1B;//27; //TX  帧信息                (扩展帧数据7）寄存器
uchar   xdata   TBSR12   _at_   0xFE1C;//28; //TX  帧信息                (扩展帧数据8）寄存器

uchar   xdata   RBSR     _at_   0xFE10;//16; //RX  帧信息(标准帧、        扩展帧）寄存器
uchar   xdata   RBSR1    _at_   0xFE11;//17; //RX  识别码(标准帧、        扩展帧）寄存器1
uchar   xdata   RBSR2    _at_   0xFE12;//18; //RX  帧信息(标准帧、        扩展帧）识别码2寄存器
uchar   xdata   RBSR3    _at_   0xFE13;//19; //RX  帧信息(标准帧数据1、扩展帧识别码3）寄存器
uchar   xdata   RBSR4    _at_   0xFE14;//20; //RX  帧信息(标准帧数据2、扩展帧识别码4）寄存器
uchar   xdata   RBSR5    _at_   0xFE15;//21; //RX  帧信息(标准帧数据3、扩展帧数据1）寄存器
uchar   xdata   RBSR6    _at_   0xFE16;//22; //RX  帧信息(标准帧数据4、扩展帧数据2）寄存器
uchar   xdata   RBSR7    _at_   0xFE17;//23; //RX  帧信息(标准帧数据5、扩展帧数据3）寄存器
uchar   xdata   RBSR8    _at_   0xFE18;//24; //RX  帧信息(标准帧数据6、扩展帧数据4）寄存器
uchar   xdata   RBSR9    _at_   0xFE19;//25; //RX  帧信息(标准帧数据7、扩展帧数据5）寄存器
uchar   xdata   RBSR10   _at_   0xFE1A;//26; //RX  帧信息(标准帧数据8、扩展帧数据6）寄存器
uchar   xdata   RBSR11   _at_   0xFE1B;//27; //RX  帧信息                (扩展帧数据7）寄存器
uchar   xdata   RBSR12   _at_   0xFE1C;//28; //RX  帧信息                (扩展帧数据8）寄存器
```

在这里先声明一下 uchar 的定义是 unsigned char ，无符号位的字符型。以上的头文件是从 PIAE 源代码里剪贴过来的，内容里的寄存器都是以以下的格式声明

类型　　储存空间　　定义别名　　关键字_at_　　寄存器的地址

声明的类型全部都是无符号位的字符型，而储存空间都是 xdata，在这里先说一句吐糟的话，就是 xdata 的定义是什么，至于现在我还是不是很明白，从网上搜了一些答案，可是我就是看不懂，相信有很多同病相怜的志同道合。

这是网络上的答案

data ---> 可寻址片内 ram
bdata ---> 可位寻址的片内 ram
idata ---> 可寻址片内 ram，允许访问全部内部 ram
pdata ---> 分页寻址片外 ram　(256 BYTE/页)
xdata ---> 可寻址片外 ram (64k 地址范围)
code ---> 程序存储区 (64k 地址范围)

从内容上来看，至于什么跟什么我也不知道个大概，不过以自己的话来理解，越小的空间访问的速度越快，越大的空间读取速度越慢，前者是 data，后者是 code，这个好比是不同重量等级的动物，如猴子虽然很轻，动作很敏捷但是体积非常有限，大象虽然体积很大，但是却很笨重，移动起来也是懒洋洋的。而外什么会是 xdata 而不是 code 呢，个人见解 xdata 可能比 code 快一些，毕竟空间的等级上，谁者空间又大速度又快，这个非 xdata 莫属。（这个纯粹是个人见解，见笑了小弟不才）

再接下来就是定义别名了，头文件似乎是参考了数据手册来定义的，名字短而意思突出，如果你看了协成电子写得头文件，你的眼睛绝对会脱窗，呵呵，开玩笑而已。

关键字 _at_ ,它的定义在网络的解释是“绝对定义地址”，这个好像是说 “什么什么别名在那里那里”那样的意思

最后就是个个寄存器在 SJA1000里面的地址，至于这个不可能以词汇来介绍了，但是多少还是可以解释一点点，就是外什么地址的定义都是以0xFExx 为开头的呢？聪明的朋友如果注意到 sja1000.h 头文件最上面的注释的话，大概已经明白了，那就是单片机使用片外寻址方式。

更详细的内容还是看看下面的图吧，

**PeliCAN 工作模式的寄存器地址：**

**Table 10** PeliCAN address allocation; note 1

| CAN ADDRESS | OPERATING MODE | | RESET MODE | |
|---|---|---|---|---|
| | READ | WRITE | READ | WRITE |
| 0 | mode | mode | mode | mode |
| 1 | (00H) | command | (00H) | command |
| 2 | status | – | status | – |
| 3 | interrupt | – | interrupt | – |
| 4 | interrupt enable | interrupt enable | interrupt enable | interrupt enable |
| 5 | reserved (00H) | – | reserved (00H) | – |
| 6 | bus timing 0 | – | bus timing 0 | bus timing 0 |
| 7 | bus timing 1 | – | bus timing 1 | bus timing 1 |
| 8 | output control | – | output control | output control |
| 9 | test | test; note 2 | test | test; note 2 |
| 10 | reserved (00H) | – | reserved (00H) | – |
| 11 | arbitration lost capture | – | arbitration lost capture | – |
| 12 | error code capture | – | error code capture | – |
| 13 | error warning limit | – | error warning limit | error warning limit |
| 14 | RX error counter | – | RX error counter | RX error counter |
| 15 | TX error counter | – | TX error counter | TX error counter |

| | | | | | | |
|---|---|---|---|---|---|---|
| 16 | RX frame information SFF; note 3 | RX frame information EFF; note 4 | TX frame information SFF; note 3 | TX frame information EFF; note 4 | acceptance code 0 | acceptance code 0 |
| 17 | RX identifier 1 | RX identifier 1 | TX identifier 1 | TX identifier 1 | acceptance code 1 | acceptance code 1 |
| 18 | RX identifier 2 | RX identifier 2 | TX identifier 2 | TX identifier 2 | acceptance code 2 | acceptance code 2 |
| 19 | RX data 1 | RX identifier 3 | TX data 1 | TX identifier 3 | acceptance code 3 | acceptance code 3 |
| 20 | RX data 2 | RX identifier 4 | TX data 2 | TX identifier 4 | acceptance mask 0 | acceptance mask 0 |
| 21 | RX data 3 | RX data 1 | TX data 3 | TX data 1 | acceptance mask 1 | acceptance mask 1 |
| 22 | RX data 4 | RX data 2 | TX data 4 | TX data 2 | acceptance mask 2 | acceptance mask 2 |
| 23 | RX data 5 | RX data 3 | TX data 5 | TX data 3 | acceptance mask 3 | acceptance mask 3 |
| 24 | RX data 6 | RX data 4 | TX data 6 | TX data 4 | reserved (00H) | – |
| 25 | RX data 7 | RX data 5 | TX data 7 | TX data 5 | reserved (00H) | – |
| 26 | RX data 8 | RX data 6 | TX data 8 | TX data 6 | reserved (00H) | – |

| CAN ADDRESS | OPERATING MODE | | | | RESET MODE | |
|---|---|---|---|---|---|---|
| | READ | | WRITE | | READ | WRITE |
| 27 | (FIFO RAM); note 5 | RX data 7 | – | TX data 7 | reserved (00H) | – |
| 28 | (FIFO RAM); note 5 | RX data 8 | – | TX data 8 | reserved (00H) | – |
| 29 | RX message counter | | – | | RX message counter | – |
| 30 | RX buffer start address | | – | | RX buffer start address | RX buffer start address |
| 31 | clock divider | | clock divider; note 6 | | clock divider | clock divider |
| 32 | internal RAM address 0 (FIFO) | | – | | internal RAM address 0 | internal RAM address 0 |
| 33 | internal RAM address 1 (FIFO) | | – | | internal RAM address 1 | internal RAM address 1 |
| ↓ | ↓ | | ↓ | | ↓ | ↓ |
| 95 | internal RAM address 63 (FIFO) | | – | | internal RAM address 63 | internal RAM address 63 |
| 96 | internal RAM address 64 (TX buffer) | | – | | internal RAM address 64 | internal RAM address 64 |
| ↓ | ↓ | | ↓ | | ↓ | ↓ |
| 108 | internal RAM address 76 (TX buffer) | | – | | internal RAM address 76 | internal RAM address 76 |
| 109 | internal RAM address 77 (free) | | – | | internal RAM address 77 | internal RAM address 77 |
| 110 | internal RAM address 78 (free) | | – | | internal RAM address 78 | internal RAM address 78 |
| 111 | internal RAM address 79 (free) | | – | | internal RAM address 79 | internal RAM address 79 |
| 112 | (00H) | | – | | (00H) | – |
| ↓ | ↓ | | ↓ | | ↓ | ↓ |
| 127 | (00H) | | – | | (00H) | – |

在头文件中个个寄存器的定义地址是不是和以上的图表一样？在这里你可能有一个问题，就是 ACR 与 AMR，TXFIFO，RXFIFO 它们所在的地址是一样的，至于这个问题还是放在后面再说吧，因为目前的我还是不是很理解，还有寄存器们的功能也不介绍了，不知道你们有没有听过郭老师说过的一句话，“先知道一个大概，要使用的时候再去了解"，当然结论还是老话，要更详细的资料还是参考数据手册吧。

一些题外话：

说老实话，以上的 Sja1000.h 头文件，基本上已经很足够了，有人会说，为什么网上还有很多不同方式编写的 sja1000.h 的头文件，个人一向的习惯，“够用就好"，至于他人是如何，呵呵见仁见智呀。

**总结：**

头文件先了解，有个概念先吧。休息一会儿吧，让大脑休息一下...

## 个人秀零三：简单的认识SJA1000的寄存器

在还没有开始写程式先，还要做功课那就是认识寄存器，无疑驱动 SJA1000 就是如何操作这些寄存器。那么有哪些寄存器用在下一个实验中呢？
我们一一来认识：

### 模式寄存器：

**Table 12** Bit interpretation of the mode register (MOD); CAN address '0'

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| MOD.7 | – | – | – | reserved |
| MOD.6 | – | – | – | reserved |
| MOD.5 | – | – | – | reserved |
| MOD.4 | SM | Sleep Mode; note 1 | 1 | sleep; the CAN controller enters sleep mode if no CAN interrupt is pending and if there is no bus activity |
| | | | 0 | wake-up; the CAN controller wakes up if sleeping |
| MOD.3 | AFM | Acceptance Filter Mode; note 2 | 1 | single; the single acceptance filter option is enabled (one filter with the length of 32 bit is active) |
| | | | 0 | dual; the dual acceptance filter option is enabled (two filters, each with the length of 16 bit are active) |
| MOD.2 | STM | Self Test Mode; note 2 | 1 | self test; in this mode a full node test is possible without any other active node on the bus using the self reception request command; the CAN controller will perform a successful transmission, even if there is no acknowledge received |
| | | | 0 | normal; an acknowledge is required for successful transmission |

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| MOD.1 | LOM | Listen Only Mode; notes 2 and 3 | 1 | listen only; in this mode the CAN controller would give no acknowledge to the CAN-bus, even if a message is received successfully; the error counters are stopped at the current value |
| | | | 0 | normal |
| MOD.0 | RM | Reset Mode; note 4 | 1 | reset; detection of a set reset mode bit results in aborting the current transmission/reception of a message and entering the reset mode |
| | | | 0 | normal; on the '1-to-0' transition of the reset mode bit, the CAN controller returns to the operating mode |

模式寄存器说得简单一点就是设置 SJA1000的状态模式 它位于地址0，而在头文件里它定义为 MODR，绝对地址值是0xFE00；

The contents of the mode register are used to change the behaviour of the CAN controller. Bits may be set or reset by the CPU which uses the control register as a read/write memory. Reserved bits are read as logic 0.

再来看看以上的一句英语短文，他说到该寄存器是可读写，而前3位亦是 BIT5~7是无用的

简单的列子：

| | |
|---|---|
| Eg: MODR=0x01 | MOD.0=1,也就是软件进入复位模式 |
| Eg: MODR= 0x04 | MOD.3=1,也就是自动检测模式 |

**命令寄存器：**

6.4.4 COMMAND REGISTER (CMR)

A command bit initiates an action within the transfer layer of the CAN controller. This register is write only, all bits will return a logic 0 when being read. Between two commands at least one internal clock cycle is needed in order to proceed. The internal clock is half of the external oscillator frequency.

**Table 13** Bit interpretation of the command register (CMR); CAN address 1

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| CMR.7 | – | reserved | – | – |
| CMR.6 | – | reserved | – | – |
| CMR.5 | – | reserved | – | – |
| CMR.4 | SRR | Self Reception Request; notes 1 and 2 | 1 | present; a message shall be transmitted and received simultaneously |
| | | | 0 | – (absent) |
| CMR.3 | CDO | Clear Data Overrun; note 3 | 1 | clear; the data overrun status bit is cleared |
| | | | 0 | – (no action) |
| CMR.2 | RRB | Release Receive Buffer; note 4 | 1 | released; the receive buffer, representing the message memory space in the RXFIFO is released |
| | | | 0 | – (no action) |
| CMR.1 | AT | Abort Transmission; notes 5 and 2 | 1 | present; if not already in progress, a pending transmission request is cancelled |
| | | | 0 | – (absent) |
| CMR.0 | TR | Transmission Request; notes 6 and 2 | 1 | present; a message shall be transmitted |
| | | | 0 | – (absent) |

这家伙就是给 SJA1000下命令的，在最上面的英文短文中说到，它只是可写的家伙。在头文件的定义中他是 CMR 而绝对地址值是0xfe01，接下来再看看几个列子吧：

Eg：CMR=0x10； 该表示 CMR.4=1,亦是自动发送请求|接收请求
Eg：CMR=0x04； 该表示 CMR.3=1，亦是释放 RXFIFO 的空间

问题来了？可能到目前你有疑问，为什么是 CMR=0x10，还是 CMR=0x04，那么他们又有什么目的~朋友慢慢来学习是不可心急的，对于自动发送|接收请求，和什么是释放 RXFIFO 空间，这个在后面会慢慢明了，这里只要先有个概念，命令寄存器就像经理，他服从上司的命令要员工们如何进行工作，而 CPU 就是老板了~

**状态寄存器：**

**Table 14** Bit interpretation of the status register (SR); CAN address 2

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| SR.7 | BS | Bus Status; note 1 | 1 | bus-off; the CAN controller is not involved in bus activities |
| | | | 0 | bus-on; the CAN controller is involved in bus activities |
| SR.6 | ES | Error Status; note 2 | 1 | error; at least one of the error counters has reached or exceeded the CPU warning limit defined by the Error Warning Limit Register (EWLR) |
| | | | 0 | ok; both error counters are below the warning limit |
| SR.5 | TS | Transmit Status; note 3 | 1 | transmit; the CAN controller is transmitting a message |
| | | | 0 | idle |
| SR.4 | RS | Receive Status; note 3 | 1 | receive; the CAN controller is receiving a message |
| | | | 0 | idle |
| SR.3 | TCS | Transmission Complete Status; note 4 | 1 | complete; last requested transmission has been successfully completed |
| | | | 0 | incomplete; previously requested transmission is not yet completed |
| SR.2 | TBS | Transmit Buffer Status; note 5 | 1 | released; the CPU may write a message into the transmit buffer |
| | | | 0 | locked; the CPU cannot access the transmit buffer; a message is either waiting for transmission or is in the process of being transmitted |

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| SR.1 | DOS | Data Overrun Status; note 6 | 1 | overrun; a message was lost because there was not enough space for that message in the RXFIFO |
| | | | 0 | absent; no data overrun has occurred since the last clear data overrun command was given |
| SR.0 | RBS | Receive Buffer Status; note 7 | 1 | full; one or more complete messages are available in the RXFIFO |
| | | | 0 | empty; no message is available |

状态寄存器它放映了目前该节点的状态，所以呢它仅可读而已。假设：读取状态寄存器的值是0x10
那么我们可以分析到，该位被置一的是 SR.4，而它放映了该节点正在接受报文。

那么再来个列子：
Eg：

```
unsigned char temp;
temp=SR;
If(temp&0x10) printf("Receiving status");
```

以上列子解释道，当我用一个临时变量 temp，读取命令寄存器的值，然后再将 temp 的值和0x10进行“与”运算，如果结构是大于1，那么这个 if 语句就成立，那么就表示该节点目前的状态时读取状态。

**中断寄存器：**

6.4.6 INTERRUPT REGISTER (IR)

The interrupt register allows the identification of an interrupt source. When one or more bits of this register are set, a CAN interrupt will be indicated to the CPU. After this register is read by the CPU all bits are reset except for the receive interrupt bit.

The interrupt register appears to the CPU as a read only memory.

**Table 15** Bit interpretation of the interrupt register (IR); CAN address 3

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| IR.7 | BEI | Bus Error Interrupt | 1 | set; this bit is set when the CAN controller detects an error on the CAN-bus and the BEIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.6 | ALI | Arbitration Lost Interrupt | 1 | set; this bit is set when the CAN controller lost the arbitration and becomes a receiver and the ALIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.5 | EPI | Error Passive Interrupt | 1 | set; this bit is set whenever the CAN controller has reached the error passive status (at least one error counter exceeds the protocol-defined level of 127) or if the CAN controller is in the error passive status and enters the error active status again and the EPIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.4 | WUI | Wake-Up Interrupt; note 1 | 1 | set; this bit is set when the CAN controller is sleeping and bus activity is detected and the WUIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.3 | DOI | Data Overrun Interrupt | 1 | set; this bit is set on a '0-to-1' transition of the data overrun status bit and the DOIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.2 | EI | Error Warning Interrupt | 1 | set; this bit is set on every change (set and clear) of either the error status or bus status bits and the EIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.1 | TI | Transmit Interrupt | 1 | set; this bit is set whenever the transmit buffer status changes from '0-to-1' (released) and the TIE bit is set within the interrupt enable register |
| | | | 0 | reset |
| IR.0 | RI | Receive Interrupt; note 2 | 1 | set; this bit is set while the receive FIFO is not empty and the RIE bit is set within the interrupt enable register |
| | | | 0 | reset; no more message is available within the RXFIFO |

中断寄存器吗..通过读取它可各种各样的中断信息，不过前提是先要在中断使能寄存器中，使能相关的中断服务，在头文件中他的定义是 IR，而绝对地址值是 0xfe03。中断寄存器一但被读取，所有位为自动复位，RI 除外，RI 是根据 RIE 的使能情况，而且只要 RXFIFO 不空缺，RI 就一直置一，反之当 RXFIFO 被释放后，如果不存在着报文，那么 RI 就在复位状态。

假设一个案例, RIE=1 接受中断使能，然后分析以下的代码：

```
Unsigned char temp;
  temp=IR;
if( Judge & 0x01)
{
       printf("Now Receiving Data...");
       RX_Buffer[0]=RBSR；
       ......
       ......
       ......
}
```

变量 temp 从 IR 中读取值，然后 temp 变量中的值与0x01进行“与”运算，如果结果大于1，if 条件成立，打印出“接受中...”，下文就是一一读取 RXFIFO 中的值。

中断使能寄存器：

**Table 16** Bit interpretation of the interrupt enable register (IER); CAN address 4

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| IER.7 | BEIE | Bus Error Interrupt Enable | 1 | enabled; if an bus error has been detected, the CAN controller requests the respective interrupt |
| | | | 0 | disabled |
| IER.6 | ALIE | Arbitration Lost Interrupt Enable | 1 | enabled; if the CAN controller has lost arbitration, the respective interrupt is requested |
| | | | 0 | disabled |
| IER.5 | EPIE | Error Passive Interrupt Enable | 1 | enabled; if the error status of the CAN controller changes from error active to error passive or vice versa, the respective interrupt is requested |
| | | | 0 | disabled |
| IER.4 | WUIE | Wake-Up Interrupt Enable | 1 | enabled; if the sleeping CAN controller wakes up, the respective interrupt is requested |
| | | | 0 | disabled |
| IER.3 | DOIE | Data Overrun Interrupt Enable | 1 | enabled; if the data overrun status bit is set (see status register; Table 14), the CAN controller requests the respective interrupt |
| | | | 0 | disabled |
| IER.2 | EIE | Error Warning Interrupt Enable | 1 | enabled; if the error or bus status change (see status register; Table 14), the CAN controller requests the respective interrupt |
| | | | 0 | disabled |
| IER.1 | TIE | Transmit Interrupt Enable | 1 | enabled; when a message has been successfully transmitted or the transmit buffer is accessible again (e.g. after an abort transmission command), the CAN controller requests the respective interrupt |
| | | | 0 | disabled |
| IER.0 | RIE | Receive Interrupt Enable; note 1 | 1 | enabled; when the receive buffer status is 'full' the CAN controller requests the respective interrupt |
| | | | 0 | disabled |

中断使能寄存器，通过单片机的设置可以使各种中断源使能。它对于单片机的访问时可读写。在头文件中他的定义是 IER 而绝对地址值是0xfe04。

在这里要注意一下，RIE 这个中断使能直接影响了 SJA1000 $\overline{INT}$ 引脚


当 RXFIFO 接受报文为饱和状态时，如果 RIE=1，那么 RI 就会被置一，然后呢 $\overline{INT}$ 会一会儿功夫从低电平变为高电平，在硬件连接中，单片机的 $\overline{INT}1$ 直接与 SJA1000 的 $\overline{INT}$ 相连接，SJA1000 $\overline{INT}$ 的低电平会引起单片机外部中断1的跳变沿式触发外部中断，那么只要在源程式中加入相应的外部中断处理函数，如下：

```
Void Ir_EX1(void) interrupt 2
{
    RX_buffer[0] =  RBSR;
    RX_buffer[1] =  RBSR1;
    .....
```

**仲裁丢失捕捉寄存器：**

6.4.8 ARBITRATION LOST CAPTURE REGISTER (ALC)

This register contains information about the bit position of losing arbitration. The arbitration lost capture register appears to the CPU as a read only memory. Reserved bits are read as logic 0.

**Table 17** Bit interpretation of the arbitration lost capture register (ALC); CAN address 11

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| ALC.7 to ALC.5 | – | reserved | For value and function see Table 18 | |
| ALC.4 | BITNO4 | bit number 4 | | |
| ALC.3 | BITNO3 | bit number 3 | | |
| ALC.2 | BITNO2 | bit number 2 | | |
| ALC.1 | BITNO1 | bit number 1 | | |
| ALC.0 | BITNO0 | bit number 0 | | |

在还没有理解什么是仲裁丢失捕捉寄存器开始之前，先了解什么是仲裁，为什么需要仲裁，那么又如何仲裁？在网络上的定义中，仲裁是一种调停或调解的方式。争议的双方或几方授权调停方作出判决、解决争议，那么 CAN 们要竞争什么呢？没错，CAN 节点们竞争的就是总线的使用权。然而 CAN 的节点们都是以 CSMA/CD 的方式来进行仲裁。

想象一下，总线就象是一个单行桥，而报文就如行人，可是不一样的是，每一个行人（报文）都有自己的 ID 号（标示符），而 CSMA/CD 就好像是管理员。当有两个行人要同时要使用单行桥，管理员就会询问他们的 ID 号，而越小的 ID 号有更高的使用权，假设这连个行人，一个 ID 是0，另一个 ID 是1，理所当然的那位拥有 0 号的行人获得，当前单行桥的使用权。

那么 CSMA/CD 如何进行仲裁？CSNM/CD 媒体（CAN 的节点）访问控制方法的工作原理，可以概括如下：

先听后说，边听边说；
节点先监听总线，总线是否空闲，如果总线是出于空闲状态，那么节点可获得当前总线的使用权，然后将报文发在上面，当报文在总线上传输时，该节点还要继续跟踪该报文，确保报文传输的过称重没有发生错误，直到报文发送结束为止。

一旦冲突，立即停说；
如果在报文传输的途中，节点检测到自己发送的报文发生错误，则立即停止传输报文；

等待时机，然后再说；
当节点因传输报文失败后，会禁止一段时间期，然后继续重新“先听先说”；

（听，即监听、检测之意；说，即发送数据之意。）

说到这里我们得出几个问题：第一-什么是错误呢？第二-一个 CAN 节点何谓关闭总线？第三-CAN 节点错误后会静止多长的时间期呢？

这些问题先不讨论，这里主要了解什么是仲裁，然后以什么方式仲裁而已。接下来来理解仲裁丢失捕捉寄存器如何工作了。

ALC 中对应了有效的前五位也就是 ALC.0~4,而 ALC.5~7则是无用的状态。就是这五个有效的位一一放映了不同的仲裁丢失。在这里我们先了解仲裁域的概念。看看以下的图：

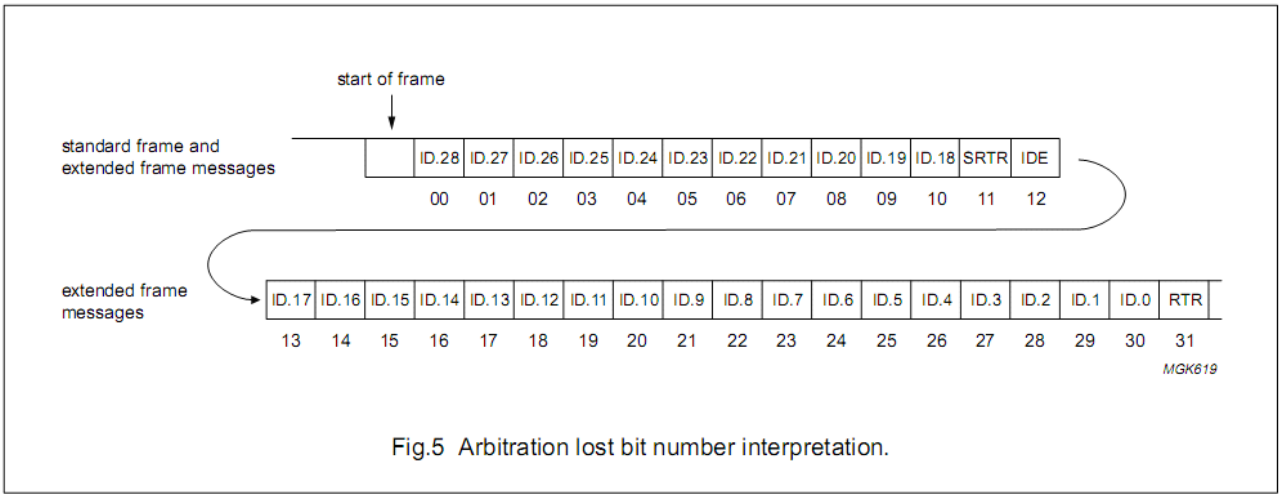



Fig.5 Arbitration lost bit number interpretation.

该图说明 SFF 与 EFF 的仲裁域，SFF 的仲裁域从 ID.28位到 IDE 位，而 EFF 的仲裁域从 ID.28位到 RTR 位再来参考一下其他的图源：

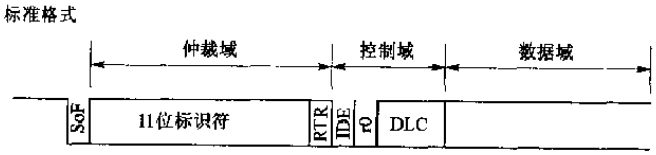



图 2.3 数据帧标准格式中的仲裁域结构

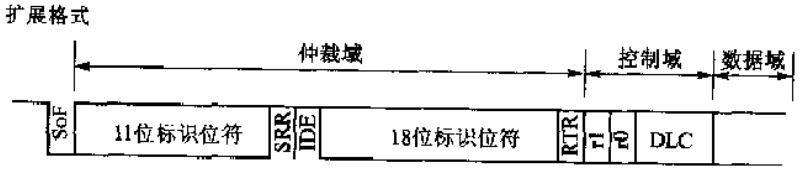



图 2.4 数据帧扩展格式中的仲裁域结构

是不是清晰了许多？那么讨论回到这里，ALC.0~4,是如何放映不同的仲裁丢失呢，这个问题就要好好的参考一小的图表了：

**Table 18** Function of bits 4 to 0 of the arbitration lost capture register

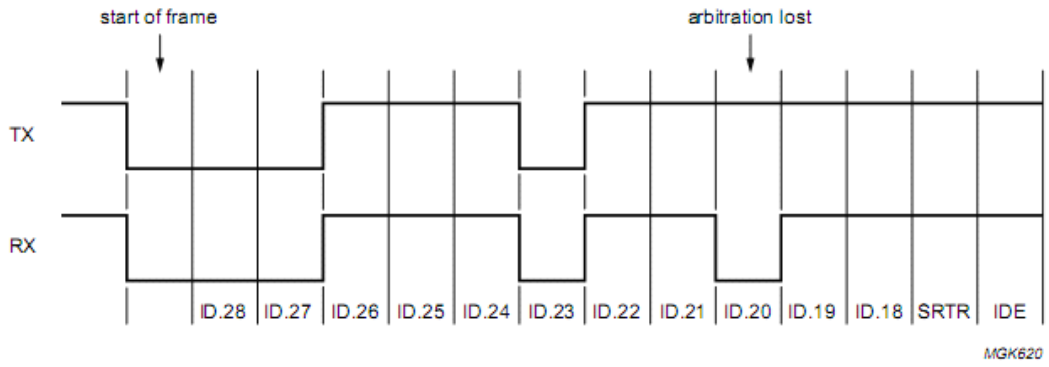
| BITS[1] | | | | | DECIMAL VALUE | FUNCTION |
|---|---|---|---|---|---|---|
| ALC.4 | ALC.3 | ALC.2 | ALC.1 | ALC.0 | | |
| 0 | 0 | 0 | 0 | 0 | 00 | arbitration lost in bit 1 of identifier |
| 0 | 0 | 0 | 0 | 1 | 01 | arbitration lost in bit 2 of identifier |
| 0 | 0 | 0 | 1 | 0 | 02 | arbitration lost in bit 3 of identifier |
| 0 | 0 | 0 | 1 | 1 | 03 | arbitration lost in bit 4 of identifier |
| 0 | 0 | 1 | 0 | 0 | 04 | arbitration lost in bit 5 of identifier |
| 0 | 0 | 1 | 0 | 1 | 05 | arbitration lost in bit 6 of identifier |
| 0 | 0 | 1 | 1 | 0 | 06 | arbitration lost in bit 7 of identifier |
| 0 | 0 | 1 | 1 | 1 | 07 | arbitration lost in bit 8 of identifier |
| 0 | 1 | 0 | 0 | 0 | 08 | arbitration lost in bit 9 of identifier |
| 0 | 1 | 0 | 0 | 1 | 09 | arbitration lost in bit 10 of identifier |
| 0 | 1 | 0 | 1 | 0 | 10 | arbitration lost in bit 11 of identifier |
| 0 | 1 | 0 | 1 | 1 | 11 | arbitration lost in bit SRTR; note 2 |
| 0 | 1 | 1 | 0 | 0 | 12 | arbitration lost in bit IDE |
| 0 | 1 | 1 | 0 | 1 | 13 | arbitration lost in bit 12 of identifier; note 3 |
| 0 | 1 | 1 | 1 | 0 | 14 | arbitration lost in bit 13 of identifier; note 3 |
| 0 | 1 | 1 | 1 | 1 | 15 | arbitration lost in bit 14 of identifier; note 3 |
| 1 | 0 | 0 | 0 | 0 | 16 | arbitration lost in bit 15 of identifier; note 3 |
| 1 | 0 | 0 | 0 | 1 | 17 | arbitration lost in bit 16 of identifier; note 3 |
| 1 | 0 | 0 | 1 | 0 | 18 | arbitration lost in bit 17 of identifier; note 3 |
| 1 | 0 | 0 | 1 | 1 | 19 | arbitration lost in bit 18 of identifier; note 3 |
| 1 | 0 | 1 | 0 | 0 | 20 | arbitration lost in bit 19 of identifier; note 3 |
| 1 | 0 | 1 | 0 | 1 | 21 | arbitration lost in bit 20 of identifier; note 3 |
| 1 | 0 | 1 | 1 | 0 | 22 | arbitration lost in bit 21 of identifier; note 3 |
| 1 | 0 | 1 | 1 | 1 | 23 | arbitration lost in bit 22 of identifier; note 3 |
| 1 | 1 | 0 | 0 | 0 | 24 | arbitration lost in bit 23 of identifier; note 3 |
| 1 | 1 | 0 | 0 | 1 | 25 | arbitration lost in bit 24 of identifier; note 3 |
| 1 | 1 | 0 | 1 | 0 | 26 | arbitration lost in bit 25 of identifier; note 3 |
| 1 | 1 | 0 | 1 | 1 | 27 | arbitration lost in bit 26 of identifier; note 3 |
| 1 | 1 | 1 | 0 | 0 | 28 | arbitration lost in bit 27 of identifier; note 3 |
| 1 | 1 | 1 | 0 | 1 | 29 | arbitration lost in bit 28 of identifier; note 3 |
| 1 | 1 | 1 | 1 | 0 | 30 | arbitration lost in bit 29 of identifier; note 3 |
| 1 | 1 | 1 | 1 | 1 | 31 | arbitration lost in bit RTR; note 3 |

好了！让我们来假设一个情况，一个 SFF 的报文的仲裁域信息发送时是如下：

| 起始帧 | ID. 28 | ID. 27 | ID. 26 | ID. 25 | ID. 24 | ID. 23 | ID. 22 | ID. 21 | ID. 20 | ID. 19 | ID. 18 | SRTR | IDE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

在接受后产生下列的变化，ID.20位为0：

| 起始帧 | ID. 28 | ID. 27 | ID. 26 | ID. 25 | ID. 24 | ID. 23 | ID. 22 | ID. 21 | ID. 20 | ID. 19 | ID. 18 | SRTR | IDE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |




图像的效果如以上，如果我们以变量 unsigned char temp 读取当前的值，然后再以0x1f 来进行“与运算"，那么 temp 值得结果会是 “0x10”，亦是 AC08 （AC08表示了从仲裁丢失捕捉寄存器中捕捉到的仲裁丢失错误，而对应该错误的十进制值就是 AC08）。 如果换成是 C 语言，编写方式有点像这样:

```
unsigned char temp；
temp=ACR;
temp=temp&0x1f;
switch(temp)
{
    ......
    case 0x10: printf("Error code AC08，ID20");
    ......
}
```

最后还有几个要值得注意的是，一旦冲裁丢失捕捉寄存器捕捉到仲裁错误，如果仲裁丢失捕捉寄存器没有给与复位，那么它就不会进行下一次的仲裁错误捕捉。复位的办法很简单，就是直接读取仲裁丢失捕捉寄存器的值，那么它就会自动复位。

```
temp=ACR;
```

写到这里辛苦了，辛苦了~老实说仲裁丢失捕捉寄存器有什么用处，呵呵呵，答案自己搜索吧！让大脑休息一会儿吧。

## 错误代码捕捉寄存器：

This register contains information about the type and location of errors on the bus. The error code capture register appears to the CPU as a read only memory.

**Table 19** Bit interpretation of the error code capture register (ECC); CAN address 12

| BIT | SYMBOL | NAME | VALUE | FUNCTION |
|---|---|---|---|---|
| ECC.7[1] | ERRC1 | Error Code 1 | – | – |
| ECC.6[1] | ERRC0 | Error Code 0 | – | – |
| ECC.5 | DIR | Direction | 1 | RX; error occurred during reception |
| | | | 0 | TX; error occurred during transmission |
| ECC.4[2] | SEG4 | Segment 4 | – | – |
| ECC.3[2] | SEG3 | Segment 3 | – | – |
| ECC.2[2] | SEG2 | Segment 2 | – | – |
| ECC.1[2] | SEG1 | Segment 1 | – | – |
| ECC.0[2] | SEG0 | Segment 0 | – | – |

有时候报文在总线传输中往往会因为总线错误而导致报文丢失或者报文错误错误代码捕捉寄存器的工作就是正对报文的失误。错误代码捕捉寄存器在头文件里的绝对地址值是0xfe0c，而定义别名是 ECC。该寄存器后针对错误种类，错误方向，错误段进行种种的捕捉。

最高两位亦即 ECC.6~7，针对了错误的种类而捕捉失误报文。以下是对应的错误种类

| BIT ECC.7 | BIT ECC.6 | FUNCTION |
|---|---|---|
| 0 | 0 | bit error |
| 0 | 1 | form error |
| 1 | 0 | stuff error |
| 1 | 1 | other type of error |

除此之外，位5 ECC.5放映了错误方向。该位逻辑1表示接收报文时发生错误，逻辑0表示发送报文时发生错误。那么 ECC.0~ECC4,纷纷对应了不同段之间的错误，该分类如的图：

**Table 21** Bit interpretation of bits ECC.4 to ECC.0; note 1

| BIT ECC.4 | BIT ECC.3 | BIT ECC.2 | BIT ECC.1 | BIT ECC.0 | FUNCTION |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | start of frame |
| 0 | 0 | 0 | 1 | 0 | ID.28 to ID.21 |
| 0 | 0 | 1 | 1 | 0 | ID.20 to ID.18 |
| 0 | 0 | 1 | 0 | 0 | bit SRTR |
| 0 | 0 | 1 | 0 | 1 | bit IDE |
| 0 | 0 | 1 | 1 | 1 | ID.17 to ID.13 |
| 0 | 1 | 1 | 1 | 1 | ID.12 to ID.5 |
| 0 | 1 | 1 | 1 | 0 | ID.4 to ID.0 |
| 0 | 1 | 1 | 0 | 0 | bit RTR |
| 0 | 1 | 1 | 0 | 1 | reserved bit 1 |
| 0 | 1 | 0 | 0 | 1 | reserved bit 0 |
| 0 | 1 | 0 | 1 | 1 | data length code |
| 0 | 1 | 0 | 1 | 0 | data field |
| 0 | 1 | 0 | 0 | 0 | CRC sequence |
| 1 | 1 | 0 | 0 | 0 | CRC delimiter |
| 1 | 1 | 0 | 0 | 1 | acknowledge slot |
| 1 | 1 | 0 | 1 | 1 | acknowledge delimiter |
| 1 | 1 | 0 | 1 | 0 | end of frame |
| 1 | 0 | 0 | 1 | 0 | intermission |
| 1 | 0 | 0 | 0 | 1 | active error flag |
| 1 | 0 | 1 | 1 | 0 | passive error flag |
| 1 | 0 | 0 | 1 | 1 | tolerate dominant bits |
| 1 | 0 | 1 | 1 | 1 | error delimiter |
| 1 | 1 | 1 | 0 | 0 | overload flag |

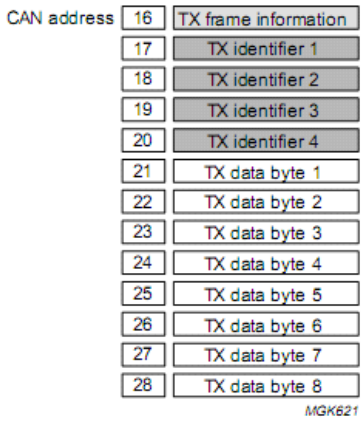
ALC 与 ECC 都一样都是自读寄存器，要释放它们也就单纯的读取它们就完事了。

## 发送缓冲器 TXFIFO 与接收缓冲器 RXFIFO

### 发送和缓冲器：

接下来要探讨的就是 TXFIFO 和 RXFIFO 也就是发送缓冲寄存器和接受缓冲寄存器。在个人秀零一-PeliCAN 简介中，对 PeliCAN 的报文结构有了相关的介绍，不过要真正要搞明白 PIAE 里的第一个 CAN 实验还是要了解这两个东西，废话还是少说。那么开始吧：

那么回顾一下 EFF 的报文结构，第一个字节是 TX 帧信息，第二个字节第四个字节是标示符，而最后八个字节就用来存放数据的，在扩展格式的报文中 FF 必须置一，而 DLC 决定了数据的长度。



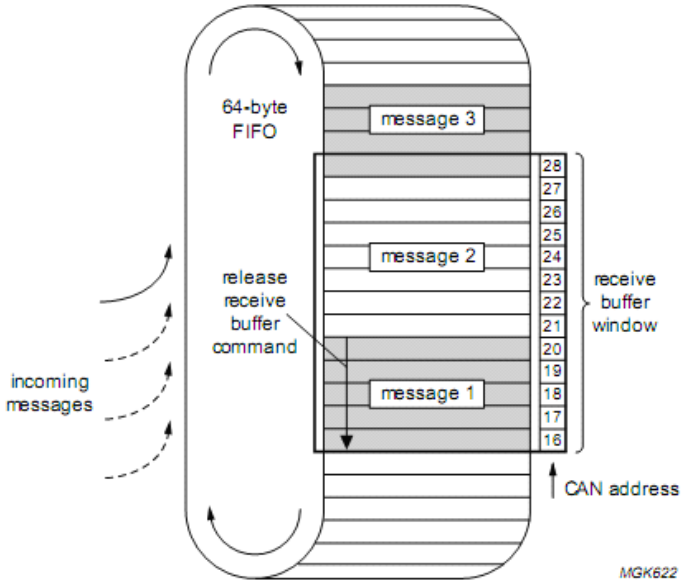

发送缓冲器在头文件里的绝对地址值是从0xfe10~0xfe1c，换句话说是 CAN 地址的16到28，TX 帧信息定义别名是 TBSR，标示符则是 TBSR1~TBSR4，数据是 TBSR5~12。CAN 控制每当要发送信息的时候，必须进行以下的步骤：

1）将 TX 帧信息写入 TBSR；
2）将标示符写入 TBSR1~TBSR4；
3）将数据写入 TBSR5~TBSR12；

最后就是向命令寄存器写入自动收发请求。C 语言的编写感觉像这样：

```
TBSR=0x88;          //TX 帧信息
TBSR1=0;            //最高优先级的标示符
TBSR2=0;
TBSR3=0;
TBSR4=0;
TBSR5=TX_1B;        //数据
TBSR6=TX_2B;
TBSR7=TX_3B;
TBSR8=TX_4B;
TBSR9=TX_5B;
TBSR10=TX_6B;
TBSR11=TX_7B;
TBSR12=TX_8B;
```

**接收缓冲器：**




以上是接受缓冲器的概念图，当有 SFF 格式的报文接收进来的时候，先进入的是 TX 帧信息，而数据随后。接受缓冲器在头文件里的绝对地址值是也是从0xfe10到0xfe1c，CAN 地址是16到28，定义别名则是 RBSR~RBSR12， RBSR 保存 TX 帧信息，RBSR1~4保存标示符，而 RBSR5~12则保存数据。

接收报文的过程与发送报文过程相比，比较麻烦一点,抽象的步骤如下：

1）当 RXFIFO（接收缓冲器）接收报文而处于饱和状态下，如果 RIE 使能，那么 RI 中断源就是置一;

2）PIN 16 ,也就是 $\overline{INT}$ 引脚，会产生跳变沿的变化而触发单片机的外部中断1；

3）在单片机的源程式中写入相关的处理函数；

4）该处理函数包括，先将 RBSR 读取 TX 帧信息；

5）然后从 RBSR1~RBSR4，读取标示符；

6）最后就是读数据了，而数据在 RXFIFO 存放的是 RBSR5~RBSR12；

7）释放 RXFIFO 空间;

过程大致上时这样，当然你也可以直从 RBSR5~RBSR12读取数据的，然后释放 RXFIFO 空间。这里就列出相关的 C 语言编写例子了，因为编程方式太多了。

**最后要稍微强调一下，以上提出的步骤或者建议绝对不是实实在在的方法，毕竟编程的方式和编程的目的有太多不一样了，这里只是提出一个概念而已，目的是要大家明白大致的操作过程而已，为了就是有一个抽象概念，好让编程工作更容易理解。**

**接收代码寄存器和接收屏蔽寄存器：**

如果说到接收报文，那么不得不搞明白接收代码寄存器和接受屏蔽寄存器，接收代码这个好比是“接收地址”，而接受屏蔽则“无视的地址位”，那么打个比方：假设我又一个电邮要寄个PIAE，而 PIAE2008@163.com 就是接受代码，视觉化的感觉如下：

| 发送的邮件==> | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 字符15 | 字符14 | 字符13 | 字符12 | 字符11 | 字符10 | 字符9 | 字符8 | 字符7 | 字符6 | 字符5 | 字符4 | 字符3 | 字符2 | 字符1 | 字符0 |
| P | I | A | E | 2 | 0 | 0 | 8 | @ | 1 | 6 | 3 | . | c | o | m |

如果163电邮服务器屏蔽功能最长为16个字符，那么假设163邮件服务器屏蔽了第15个字符，所以呢第15字符无论是 P 还是其他的字符，PIAE 也接收到邮件，eg：BIAE2008@163.com。视觉化的效果如下：

| ==>163邮箱服务器接收 | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 字符15 | 字符14 | 字符13 | 字符12 | 字符11 | 字符10 | 字符9 | 字符8 | 字符7 | 字符6 | 字符5 | 字符4 | 字符3 | 字符2 | 字符1 | 字符0 | 状态 |
| P | I | A | E | 2 | 0 | 0 | 8 | @ | 1 | 6 | 3 | . | c | o | m | 接收 |
| B | I | A | E | 2 | 0 | 0 | 8 | @ | 1 | 6 | 3 | . | c | o | m | 接收 |

那么再来假设一个状况，如果163服务器不屏蔽任何字符呢？效果会怎么样呢？答案是肯定的， 如果发送的邮件地址不是 PIEA2008@163.com, PIAE 是不可能收到的。视觉化得效果如下：

| ==>163邮箱服务器接收 | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 字符15 | 字符14 | 字符13 | 字符12 | 字符11 | 字符10 | 字符9 | 字符8 | 字符7 | 字符6 | 字符5 | 字符4 | 字符3 | 字符2 | 字符1 | 字符0 | 状态 |
| P | I | A | E | 2 | 0 | 0 | 8 | @ | 1 | 6 | 3 | . | c | o | m | 接收 |
| B | I | A | E | 2 | 0 | 0 | 8 | @ | 1 | 6 | 3 | . | c | o | m | 拒绝 |

明白了吗？换另一句话说，屏蔽位可以看成是“无视”的功能。那么以上的假设又跟 CAN 的接受代码寄存器和接收屏蔽寄存器有什么关系？在某种定义上，这两个寄存器是为 RXFIFO 也就是接受缓冲寄存器所准备的，当多个报文在总线传输时，能进入到节点的 RXFIFO 只有相对应标示符的报文，那么该节点就要事先将接受的标示符写入接受代码寄存器，过后进行屏蔽动作，这一过程就是验收过滤。

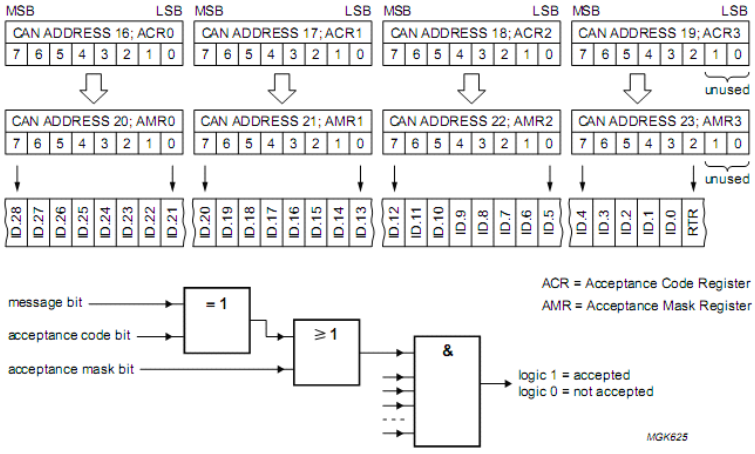
这样说很模糊，假设一个实例：

A 节点发送一个到 B 节点为目的地的报文，而 A 节点发送的报文中的标示符是0x00,0x11,0x22,0x33。如果 B 节点要接受 A 节点的报文，那么 B 节点要事先将0x00,0x11,0x22,0x33写入接受代码寄存器，同时也要将屏蔽位写入接受屏蔽寄存器，而一般都是0xf000x00,0x00,0x00（只接收相对应的标示符，不进行任何屏蔽动作）。那么验收过滤发生时，A 节点发送的报文就会成功被 B 节点接受到 RXFIFO 里面。

接受代码寄存器在头文件中的绝对地址值是0xFE10~0xFE13，而相对的定义别名是 ACR，ACR1，ACR2，ACR3。屏蔽代码寄存器的绝对地址值是0xfe14~0xfe17,则对应的别名是 AMR，AMR1，AMR2，AMR3。事先说明一下，验收过滤有两种模式，单验收过滤和双验收过滤，这里就介绍单验收过滤模式，而且也是 EFF 帧格式的报文而已。那么设置验收过滤模式是在模式寄存器中的 MOD.3 (AFM)进行。

| | | | | |
|---|---|---|---|---|
| MOD.3 | AFM | Acceptance Filter Mode; note 2 | 1 | single; the single acceptance filter option is enabled (one filter with the length of 32 bit is active) |
| | | | 0 | dual; the dual acceptance filter option is enabled (two filters, each with the length of 16 bit are active) |

话到这里先看看单验收滤波模式，如何过滤 EFF 格式的报文




图形果然更能说话，一张简单的图表就解释了一切，可是还是要介绍一下，ACR0进行 ID.21~28的标示符验收，而 AMR0对 ID21~28进行屏蔽动作，接下了的 ACR1~2，AMR1~2也是依此类推，最后就是 ACR3 和 AMR3，在这里你还记得 SSF 帧格式的报文的结构吗？在第四字节中只有前六个位被用到：

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| ID.4 | ID.3 | ID.2 | ID.1 | ID.0 | RTR(2) | 0 | 0 |

位1和位0都是被保留，而在数据手册里提到，相关的“无用位”理应在 AMR3中设置为逻辑1，以表示“无关”的状态。

It should be noted that the 2 least significant bits of AMR3 and ACR3 are not used. In order to be compatible with future products these bits should be programmed to be "dont care" by setting AMR3.1 and AMR3.0 to logic 1.

在 C 语言中给接受代码寄存器和屏蔽代码寄存器的编程方式感觉像这样（在给该寄存器赋初值之前，必须先设置验收滤波模式和复位模式状态）：

```
MODR = 0x09;      //设置为单验收滤波，和复位状态

ACR  = 0x00;      //初始化标示码
ACR1 = 0x11;
ACR2 = 0x22;
ACR3 = 0x33;

AMR  = 0x00;      //初始化掩码，无屏蔽动作
AMR1 = 0x00;
AMR2 = 0x00;
AMR3 = 0x00;
```

越说越糊涂起来了，那么还是假设一个实例吧：

有一个 EFF 报文包含标示符为

| EFF 报文的标示符 | | | |
|---|---|---|---|
| 第一字节 | 第二字节 | 第三字节 | 第四字节 |
| 000 10001 | 0001 0001 | 0001 0001 | 0001 0101 |

而某节点使用单验收滤波为模式，并且在 AMR3设置了屏蔽代码0x01

| AMR0 | AMR1 | AMR2 | AMR3 |
|---|---|---|---|
| 0x00（无屏蔽） | 0x00（无屏蔽） | 0x00（无屏蔽） | 0000 0001 |

那么如此的情况，只要 EFF 报文的第四字节表示的标示位无论是0001 0101 或者 0001 0100 都有效的被该节点接受，换句话说屏蔽位使得标示位为“无关系”的状态。

再深一层设想，如果该节点奖所有屏蔽位都设置的话，那么该节点什么报文它都会接收。

| AMR0 | AMR1 | AMR2 | AMR3 |
|---|---|---|---|
| 0xff | 0xff | 0xff | 0xff |

| | 第一字节 | 第二字节 | 第三字节 | 第四字节 |
|---|---|---|---|---|
| 报文标示符 | 0001 0001 | 0001 0001 | 0001 0001 | 0001 0001 |
| 屏蔽代码 | 1111 1111 | 1111 1111 | 1111 1111 | 1111 1111 |
| 对应关系 | **** **** | **** **** | **** **** | **** **** |

至于其他模式的验收滤波，可以参考一下前辈的作品，这里就借签了

## CAN 总线学习笔记三：验收滤波

了解 CAN 总线的人都知道，CAN 总线在的帧数据在总线上传送时，其它的 CAN 控制器是通过验收滤波来决定总线上的数据帧的 ID 是否和本节点相吻合，如果与本节点吻合，那么总线上的数据就被存入总线控制器的相应寄存器里，否则就抛弃该数据，从而也能够减轻总线控制器的工作量。换句话说，总线上数据帧的 ID 通过待接收节点的验收滤波后是吻合的，是可以被接收的。

那么，总线控制器是如何进行验收滤波的呢？验收滤波分单滤波和双滤波。标准帧和扩展帧由于 ID 长度不同，它们的两种滤波也有所区别。这里我只重点举一个例子，因为只要理解了一种滤波方式，其它的滤波方式都是类似的，也很容易就理解了。这里就说扩展帧的双滤波方式。所谓双滤波，就是有两次的滤波，但并非两次滤波都需要通过才双通过，两次滤波只要有一次滤波成功那么就默认滤波通过，可以接收数据了。

| ACR0 | ACR1 | ACR2 | ACR3 |
|---|---|---|---|
| AMR0 | AMR1 | AMR2 | AMR3 |
| ID.28-ID.21 | ID.20-ID.13 | ID.28-ID.21 | ID.20-ID.13 |

如上表所示，ACR 寄存器是接收代码寄存器，AMR 是接收屏蔽寄存器。ACR 一般是需要与对应的 ID 相吻合的，但是如果 AMR 的相应位上设置为1的时候，ID 的那一位数据可以不和 AMR 的相应位一样，也就是起到屏蔽的作用。

举个例子。如果 ACR0=11101111，AMR0=00000000，那么要想通过验收滤波，必须 ID.28-ID.21=ACR0=11101111。如果 AMR0=00010000，那么 ID.28-ID.21=11111111时，也可以通过验收滤波，因为此时 AMR0的第五位为1，也就是屏蔽了 ACR0的第五位。所以 ID 的相应位可以不合 ACR0一致。

在扩展帧的双滤波方式下，ACR0\ACR1分别对应 ID.28-ID.13，ACR2\ACR3分别也对应 ID.28-ID.13，这就达到了两次滤波的效果。另外要说明的一点是：通过验收滤波后符合节点要求的数据就存储到节点的相应寄存器里，其它的帧信息并不做存储。

**总线时序寄存器0和总线时序寄存器1：**

说到总线时序的话虽然在字面上还不是很理解，不过可以将这两个个寄存器看着是用来设定波特率的。总线时序寄存器0，1在头文件中的绝对地址值是0xfe06，0xfe07，而且定义别名是 BRP0，BRP1；如果把 CAN 总线时序看得简单一点的话就是：

BRP0决定了 CAN 系统时钟而 BRP1决定了不同的时间段，如：同步时间段，时间段1和时间段2。波特率就是由这些：CAN 系统时钟，头部时间段，时间段1，时间段2 组成的。

至于 CAN 的系统时钟和各各时间段都是什么东西？这个要慢慢的介绍了。先看看 BRP0的结构：

**总线时序寄存器0:**

**Table 44** Bit interpretation of bus timing register 0 (BTR0); CAN address 6

| 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 |

如表中的那样 BRP.0~5,使得 CAN 系统时钟可编程,那么接下来就是讨人厌的公式了：

$$t_{scl} = 2 \times t_{CLK} \times (32 \times BRP.5 + 16 \times BRP.4 + 8 \times BRP.3 + 4 \times BRP.2 + 2 \times BRP.1 + BRP.0 + 1)$$

$$\text{where } t_{CLK} = \text{time period of the XTAL frequency} = \frac{1}{f_{XTAL}}$$

Tscl 就是 CAN 系统时钟，而 Tclk 求出的方法就是 1/Fxtal 。这样用几句话来解释一切可能不行的，我们就来一个简单的实例吧：

假设我购入了 CAN 学习板第五版，而 SJA1000使用了独立的振荡晶体为16MHz，那么请求出该 CAN 的系统时钟:（注：BRP0赋值为0x31 , BRP.5 BRP.4 BRP.0 都置一，其余都是0）

由公式我们可以求出：

Tclk=1/16Mhz=62.5纳秒

而系统时钟是:

Tscl=2*62.5纳秒 * ( 32 * 1 + 16 * 1 + 1 + 1 )

=125纳秒 * ( 32+16+1+1)

=125纳秒 * (50)

=6.25微秒；

**总线时序寄存器1：**

明白了如何求出 CAN 系统时钟以后，接下来我们要了解如何计算各各时间段了。先看看 BRP1的内部结构

| 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 |

BIT 0~BIT 3 亦是控制时间段1的 Tseg1.0~Tseg1.3; 而 BIT 4 ~ BIT 5 亦是控制时间段2 的 Tseg2.0~Tseg2.1 。那么注意一下的公式：

Tsyncseg=Tscl * 1

Tseg1=Tscl * ( 8 * Tseg1.3 + 4 * Tseg1.2 + 2 * Tseg1.1 + Tseg1.0 + 1)

Tseg2=Tscl * ( 4 * Tseg2.2 + 2 * Tseg2.1 + Tseg2.0 + 1 )

BRP1就先了解到这里为止，至于他们有什么用？可以尝试这样理解，同步时间段，时间段1，和时间段2是为了求出 CAN 控制的位时间，而 CAN 的位时间就是决定波特率最重要的东西。

**波特率的的计算：**

让我们看看 CAN 总线时序里波特率的公式：

波特率=1/TBit ( TBit 就是位时间 )

而 TBit 求得的公式是：

TBit= Tsyncseg + Tseg1 + Tseg2

而 Tsyncseg，Tseg1，Tseg2 求得的公式又是：

Tsyncseg=Tscl * 1

Tseg1=Tscl * ( 8 * Tseg1.3 + 4 * Tseg1.2 + 2 * Tseg1.1 + Tseg1.0 + 1)

Tseg2=Tscl * ( 4 * Tseg2.2 + 2 * Tseg2.1 + Tseg2.0 + 1 )

当然也可以看成像以下那样，不过还是典型的公式好，以下的公式是为了更容易明白而已，作为参考的份 。

TBit= (1个同步时间段 + n 个时间段1 + n 个时间段2 ) * Tscl

=(1+n+n)*Tscl

如果 (1+ n + n) 等于 x , Tscl 等于 y

Tbit = x * y

**在这里我给出一个简单的计算题：**

某天我打开了 PIAE 其中一个实验源码，从中我看到了 BRP0的赋初值是0x31，BRP1的赋初值是0x1c，况且我所佩戴的 CAN 学习板第五版是使用了振荡晶体，而频率是16Mhz，那么我该如何求出，PIAE 在这次的实验中，它的总线时序是设置那个波特率？

(Fxtal= 16Mhz, BRP0=0x31, BRP1=0x1c)

先求出系统时钟 Tscl：

Tclk= 1/16Mhz = 62.6纳秒

Tscl=Tscl=2*62.5纳秒 * ( 32 * 1 + 16 * 1 + 1 + 1 )

=6.25微秒

求得 CAN 系统时钟后，再求出 CAN 控制的位时间 Tbit,也就是求出同步时间段，时间段1和2的综合：

Tsyncseg=Tscl * 1

Tsyncseg=1 * 6.25微秒 = 6.25微秒

Tseg1= Tscl * ( 8 * Tseg1.3 + 4 * Tseg1.2 + 2 * Tseg1.1 + Tseg1.0 + 1)

=6.25微秒 * (12+1)

=6.25微秒*13

=81.25微秒

Tseg2=Tscl * ( 4 * Tseg2.2 + 2 * Tseg2.1 + Tseg2.0 + 1 )

=6.25微秒 * (1+1)

=6.25微秒 * (2)

=12.5微秒

TBit= Tsyncseg + Tseg1 + Tseg2

=（6.25+81.25+12.5）微秒

=100微秒

求出比特率：

比特率=1/TBit = 1/100微秒 = 10000 bps =10 Kbps

在上一个问题中我们知道了 PIAE 在这次的实验中使用了10Kbps 的波特率。

其实我之间一直有一个问题，为什么 sja1000佩戴的晶体是16MHz，就如当初89c52单片机一样，我也搞不明白，因为一般情况下89c52单片机佩戴的晶体大多数为11.0592， 后来发现了，原来是为了在串口通讯的时候使得错误率为0 。 可能 SJA1000 也是一样吗 ？ 为了使 波特率更容易编程...

在网上已经早有好心人将不同的波特率的结果计算出来了，计算基础的晶体的频率为16Mhz，我们可以直接将值赋给 BRP0 和 BRP1 ，那样不就更直接吗？ 呵呵，这个不是懒惰，而是“善用资源”啊哈哈哈

| CAN_ByteRate | 波特率（Kbit/s） | BRP0 | BRP1 |
|---|---|---|---|
| 0 | 5 | 0XBFH | 0XFFH |
| 1 | 10 | 0X67H | 0X2FH |
| 2 | 20 | 0X53H | 0X2FH |
| 3 | 40 | 0X87H | 0XFFH |
| 4 | 50 | 0X47H | 0X2FH |
| 5 | 80 | 0X83H | 0XFFH |
| 6 | 100 | 0X43H | 0X2FH |
| 7 | 125 | 0X03H | 0X1CH |
| 8 | 200 | 0X81H | 0XFAH |
| 9 | 250 | 0X01H | 0X1CH |
| 10 | 400 | 0X80H | 0XFAH |
| 11 | 500 | 0X00H | 0X1CH |
| 12 | 666 | 0X80H | 0XB6H |
| 13 | 800 | 0X00H | 0X16H |
| 14 | 1000 | 0X00H | 0X14H |

好了，介绍先到这里为止了~这个时间估计是晚饭时间了，至于什么是 SJW 和 SAM，这个现在估计介绍不上了，还是先歇一会儿吧，让大脑消化和吸收。

**输出分频寄存器：**

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| CAN mode | CBP | RXINTEN | (0)[1] | clock off | CD.2 | CD.1 | CD.0 |

时钟分频寄存器在头文件里决定地址值是0xfe1f，而定义别名是 CDR。时钟分频寄存器，这个寄存器呀主要是控制输出给单片机的 CLKOUT 频率，换另一句话说输出分频寄存器是用来设置 CLKOUT 引脚 CLKOUT 7，原来 CLKOUT 引脚也挺大炮的，还有专属的寄存器。

基本上，在 PIAE 设计的 CAN 学习版中，单片机有专属的振荡晶体，这个 CLKOUT 引脚也失去意义了，不过在节点初始化的过程中，还是要设置到它，那么再这里就来个简单的介绍吧。

看到 BIT 0~2吗？亦是 CD.0到 CD.2，他们是用来设置 CLKOUT 引脚输出的频率，具体的情况还是观察以下的图表：

**Table 50** CLKOUT frequency selection; note 1

| CD.2 | CD.1 | CD.0 | CLKOUT FREQUENCY |
|---|---|---|---|
| 0 | 0 | 0 | $\frac{f_{osc}}{2}$ |
| 0 | 0 | 1 | $\frac{f_{osc}}{4}$ |
| 0 | 1 | 0 | $\frac{f_{osc}}{6}$ |
| 0 | 1 | 1 | $\frac{f_{osc}}{8}$ |
| 1 | 0 | 0 | $\frac{f_{osc}}{10}$ |
| 1 | 0 | 1 | $\frac{f_{osc}}{12}$ |
| 1 | 1 | 0 | $\frac{f_{osc}}{14}$ |
| 1 | 1 | 1 | $f_{osc}$ |

Fosc 就是 SJA1000佩戴晶体的振荡频率，我们的学习板使用的是16Mhz，假设一个情况，当我设置 CD.0~2 都是0的时候，那么 CLKOUT 输出引脚的频率是 SJA1000晶体频率的一半，也就是8Mhz。

接下来要讨论的还有 Bit 3，CLOCK OFF 和 Bit 7的 CAN mode。CLOCKOFF 位主要是使能 CLKOUT 引脚有效还是无效，设置为1时 CLKOUT 引脚无效，亦即没有频率输出，如果设置为0, CLKOUT 引脚有效。至于 CANmode 亦是 BIT 7 ，这个在数据手册中没有详细的解释，不过大通一点就是用什么模式就设定什么模式（PeliCAN 还是 BasicCAN），逻辑0是 BasicCAN 模式 ，逻辑1则是 PeliCAN 模式。

| BIT 3 Clock Off | |
|---|---|
| 0 | CLKOUT 引脚有效 |
| 1 | CLKOUT 引脚无效 |
| BIT 7 CAN Mode | |
| 0 | BasicCAN 模式 |
| 1 | PeliCAN 模式 |

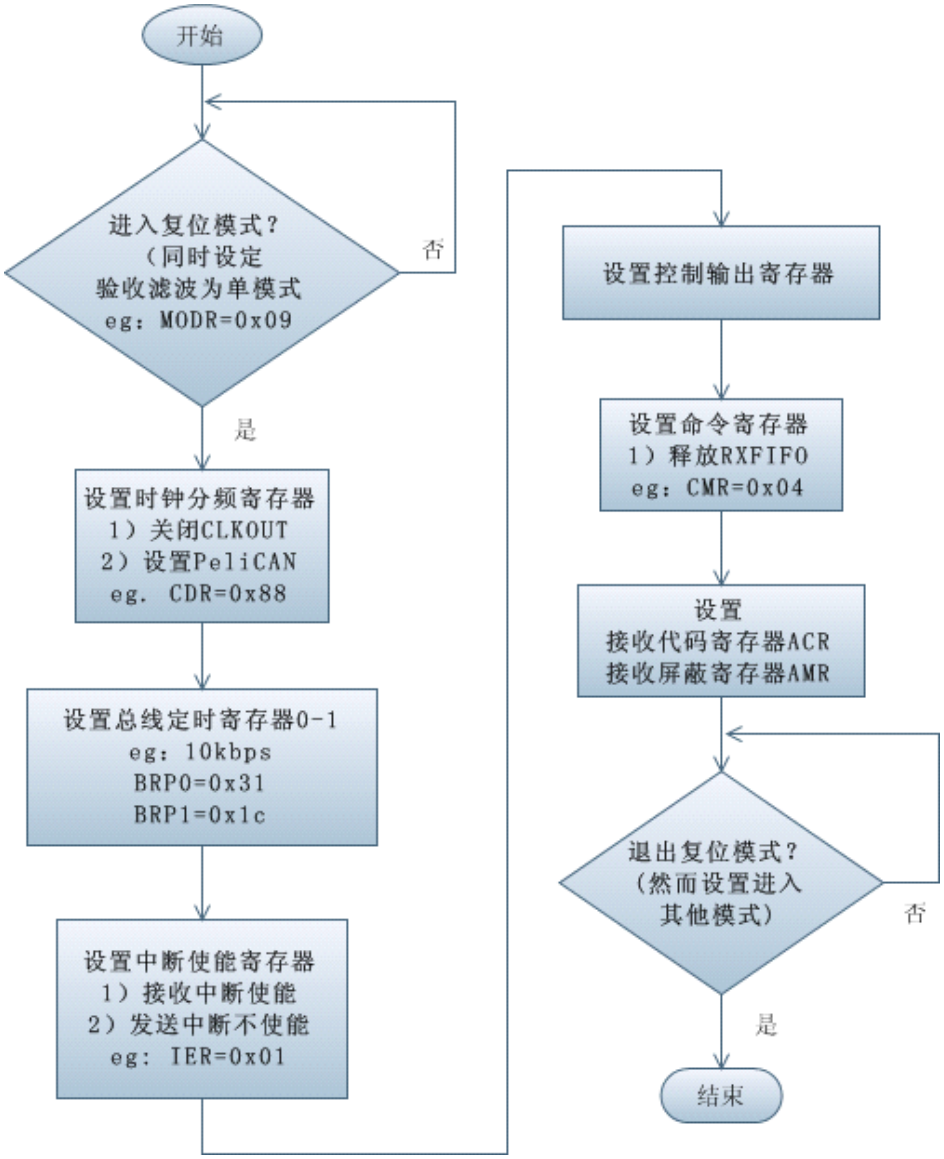
最后，BIT5和 BIT6现在就不介绍了，目前的我很理解还是有限，怕误人子弟。

总结：到目前为止认识了很多相关的寄存器，那么用来明白的最小份量已经达到了，虽然还有很多很多的寄存器没有介绍到，不过有缘的话会慢慢的认识他们。

**个人秀零四：编写节点初始化函数**

节点的初始化应该算是写程式的前奏，开始直接建议先复习之前的几章个人秀好让脑里建立一个总结的概念。如果准备好了，那么就开始吧：

初始化节点无疑有如以下流程图：




初始化过程中牵扯到得寄存器有模式寄存器，时钟分频寄存器，总线时钟寄存器，中断使能寄存器，控制输出寄存器，命令寄存器，接收代码寄存器，接收屏蔽寄存器。但是这里还是保持了一个疑问，就是控制输出寄存器，我实在搞不明白它是什么，在编写源码的时候只能借签 PIAE 了，呵呵见笑了小的不才。废文就不多写了，流程图已经解释到很清楚了，那么还是开始编写代码吧：

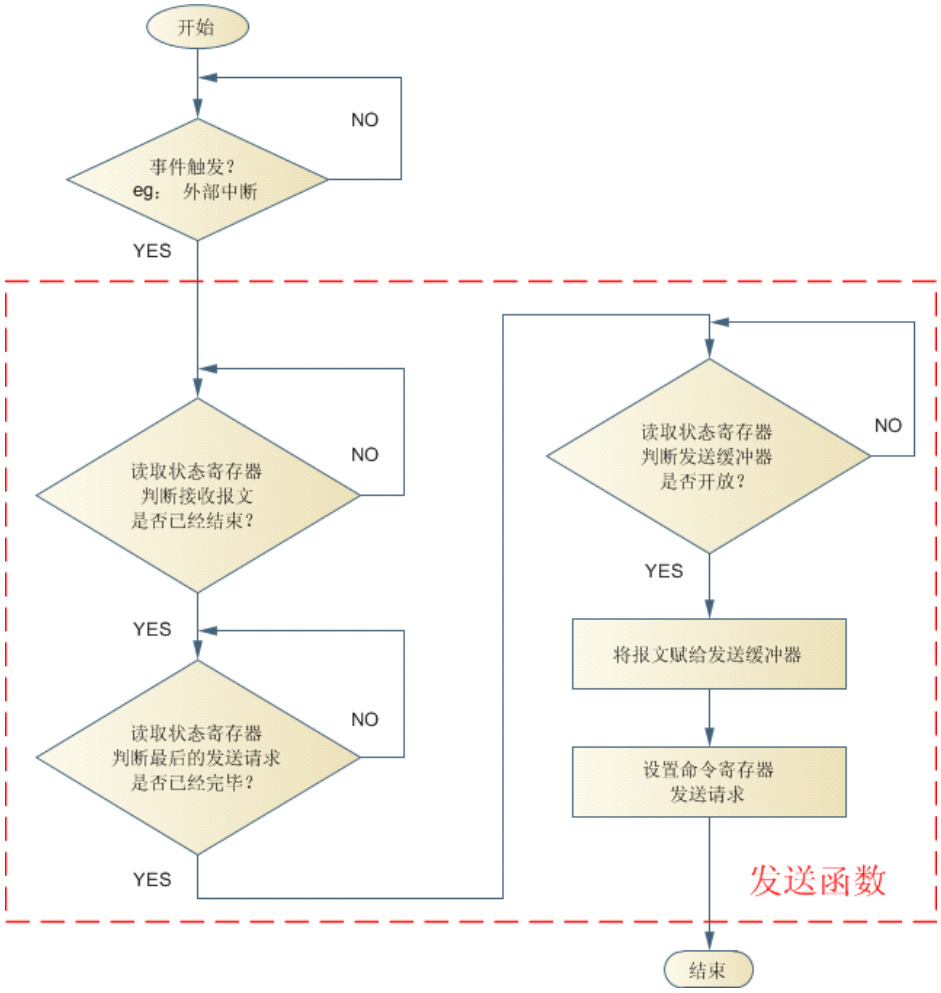
编写 CAN 初始化函数：

```
//CAN 初始化函数
void Init_CAN(void)
{
    unsigned char bdata temp;      //建立存储在 bdata 中的临时变量 temp

    do
    {
        MODR=0x09; temp=MODR;   //模式寄存器-设置复位模式而且单验收滤波模式
    }
    while(!(temp&0x01));   //判断

    CDR=0x88;         //时钟分频寄存器-设置 PeliCAN 模式，Clock off 有效
    BTR0=0x31;        //总线时序寄存器-波特率为10kbps
    BTR1=0x1c;
    IER=0x01;         //中断使能寄存器-接收中断使能
    OCR=0xaa;         //输出控制寄存器-借签...
    CMR=0x04;         //命令寄存器-释放 RXFIF0

    ACR0='C';         //接收代码寄存器-本节点地址位 CAN0
    ACR1='A';
    ACR2='N';
    ACR3=0x00;

    AMR0=0x00;        //接收屏蔽寄存器-无任何屏蔽
    AMR1=0x00;
    AMR2=0x00;
    AMR3=0x03;     //最后两位为无用位，必须设置为逻辑1

    /*根据想要设置的模式而进行设置，这里设置了自检测模式*/
    do
    {
        MODR=0x08; temp=MODR;     //模式寄存器-推出复位模式，保持单验收滤波模式
    }
    while(temp&0x01);     //判断...
}
```

以上是一个简单的 CAN 初始化函数，大体上和 PIAE 大同小异,可能是个人习惯不同，在这里接受屏蔽寄存器我设置为“无任何屏蔽”，相反的 PIAE 是“全部屏蔽”。其实初始化函数也没有什么故事好说的，最重要是多参考别人写的代码，多实践就 okay 了。

**个人秀零五：节点发送函数**

学习完初始化函数，那么现在我们就学习发送函数基本的概念。还是依然看流程图：




发送函数要编程起来比较麻烦一点，大体上要触发事件才会启动发送函数，触发的事件可以是-外部中断，或者其他的。当触发事件后，就会开始调用发送函数，最先判断 SR.4-接收报文动作是否已经结束，如果读取的值是1的话，那么就会不停的循环，当 SR.4==0, 该表示目前接受报文动作是闲置状态，就会进入下一个动作。

这个动作会判断，发送状态为 SR.3 最后发送请求是否已经结束？如果 SR.3 是逻辑，那么表示最后发送请求还在执行中，相反的如果是逻辑0，就表示最后发送请求已经结束，进入下一个动作。（其实在这个动作之前，还可以先判断发送状态为 SR.5.，虽然这个动作有点多此一举，不过可以更好的保障流程的顺利性。）

在写入发送缓冲寄存器之前要先判断发送缓冲寄存器是否已经那个开放？如果发送缓冲当前被锁着，SR2 就会放映逻辑0，相反的则是逻辑1。

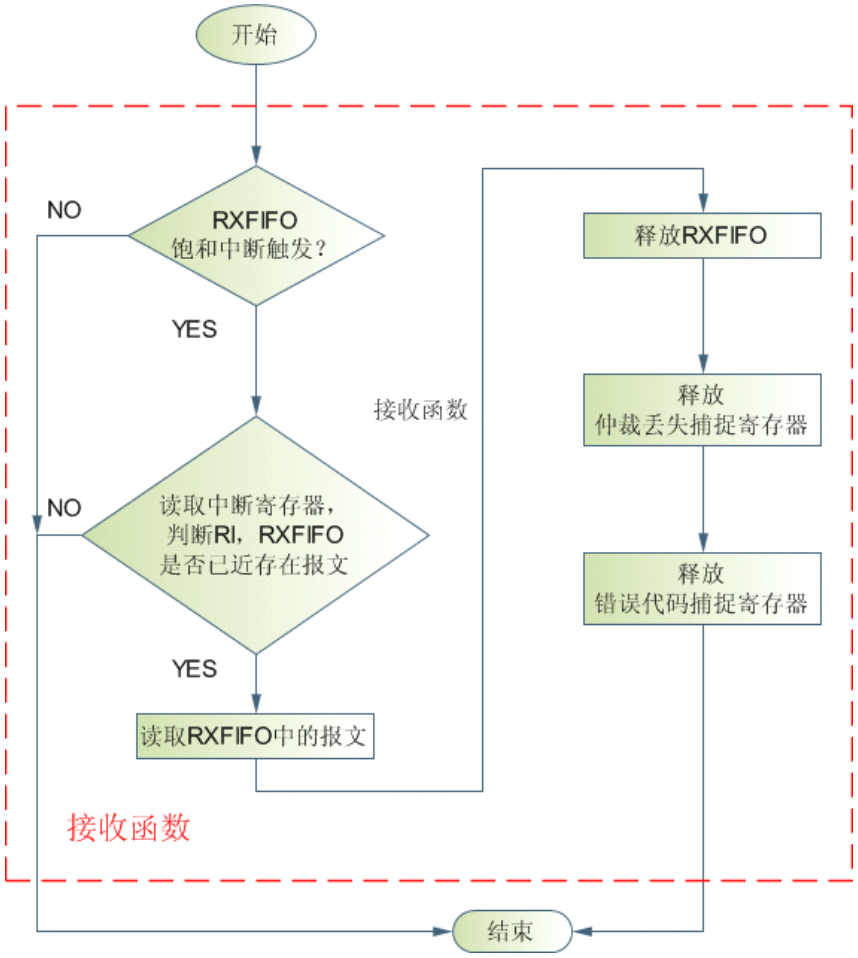
最后呢，将报文写入发送缓冲寄存器，并且设置命令寄存器，告诉他发送报文已经准备好，设置发送请求 。

节点发送函数：

```
#include "reg52.h"
#include "sja1000.h"
unsigned char Txd_data;

//CAN 节点发送函数
void CAN_TXD(void)
{
    unsigned char bdata temp;
    do
    {
        temp=SR;              //判断报文接收完毕？
    }
    while(temp&0x10);       //SR.4=0  发送闲置，SR.4=1  发送状态中

    do
    {
        temp=SR;              //判断最后报文请求？
    }
    while(!(temp&0x08));    //SR.3=0  没有余报文请求，SR.3=1  还存在报文请求

    do
    {
        temp=SR;              //判断 TXFIFO 是否锁定？
    }
    while(!(temp&0x04));    //SR.2=0  锁定，SR.2=1  开放

    TBSR0=0x88;           //ff=1 SFF 格式，DLC.3=1  数据长度8
    TBSR1='C';            //目标节点地址  CAN0
    TBSR2='A';
    TBSR3='N';
    TBSR4=0x00;
    TBSR5=Txd_data;     //数据
    TBSR6=0x00;
    TBSR7=0x00;
    TBSR8=0x00;
    TBSR9=0x00;
    TBSR10=0x00;
    TBSR11=0x00;
    TBSR12=0x00;

    CMR=0x01;           //命令寄存器-发送请求

}
```

**个人秀零六：节点接收函数**

这秀主要介绍关于接收函数，接收函数与发送函数有些不同，接收函数的事件则是当“报文接收”时触发。不过前提就是 RIE（接收中断使能）必须为逻辑1，因为 报文接受与引起接收中断而该效果关系性的直接影响 SJA1000 的使 INT 引脚产生电平变化会而触发单片机的外部中断。




接收函数触发开始直接必须完成几个条件；一 RIE 必须使能，二硬件的链接。RIE 使能-这个条件估计这个是很容易理解，但是为什么要符合硬件连接的条件？这个答案之前一直重复了，因为 SJA1000 的 INT 引脚与单片机的 INT1引脚是直接相连的（参考原理图），而我们编程的方法是由 SJA1000 RXFIFO 饱和中断，引起 SJA1000引脚的电平产生变化而触发单片机的外部中断服务，就是这个外部中断源通知单片机进行“从 RXFIFO 读取报文”的函数。这样讲可以理解了吗？

接收函数大体上比较简单，就是先判断 RI，RXFIFO 空间是否“不空"，而“不空”表示接收的报文已经存在，那么接下来的动作就是读取报文，然后给命令寄存器设置释放 RXFIFO，过后呢释放仲裁丢失捕捉寄存器和错误代码捕捉寄存器（等于读取仲裁丢失捕捉寄存器和错误代码捕捉寄存器）就结束了。
ECC（错误代码捕捉寄存器）就不介绍了，自己看看数据手册。

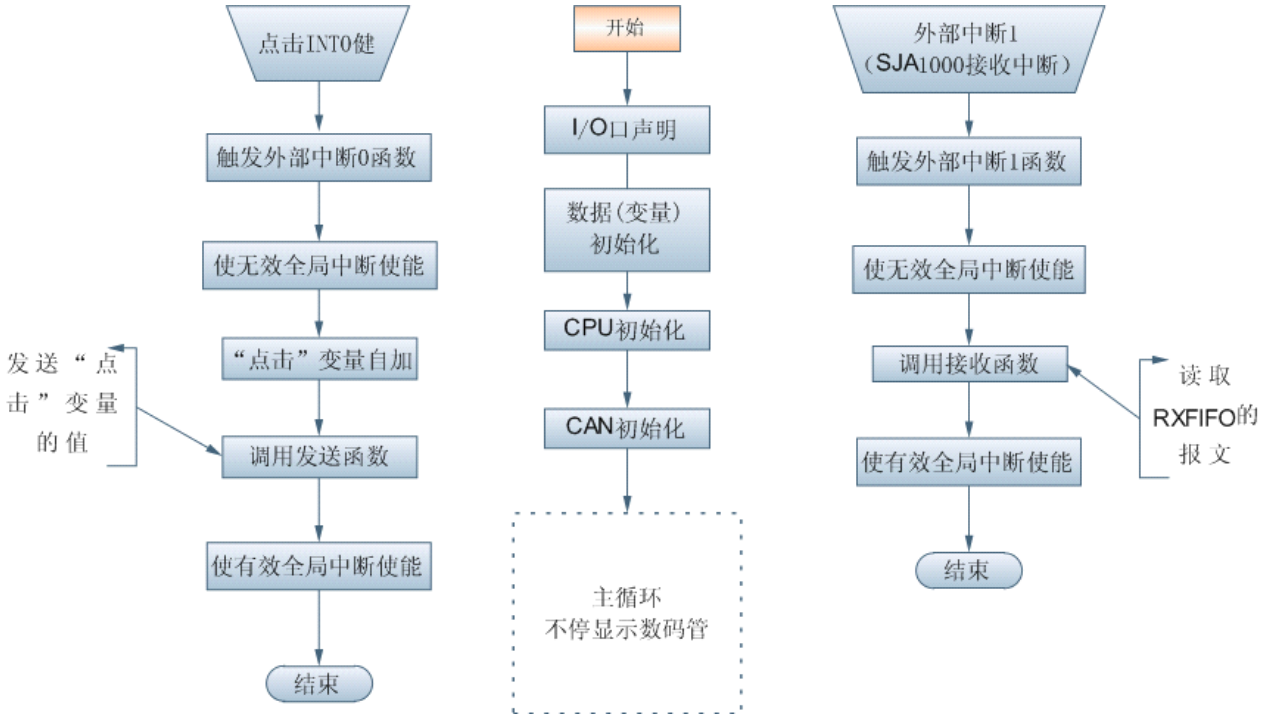
在这里要提醒一下：

如果你认真读过数据手册，会被里边的内容使混乱。我就有这样的经验，不过一旦把事情简化就很简单：“当接受到报文，如果 RIE 使能就产生接收中断 RI，如果 RIE 不使能就不产生接收中断 RI”就这样而已。

节点接收函数：

```
#include"reg52.h"
#include "sja1000.h"

unsigned char bdata temp;
unsigned char RX_buffer[13];

//节点接收函数
void CAN_RXD(void)
{
    temp = IR;
    if( temp & 0x01)                        //判断是否接收中断
    {
        RX_buffer[0] =   RBSR;              //读取 RXFIFO
        RX_buffer[1] =   RBSR1;
        RX_buffer[2] =   RBSR2;
        RX_buffer[3] =   RBSR3;
        RX_buffer[4] =   RBSR4;
        RX_buffer[5] =   RBSR5;
        RX_buffer[6] =   RBSR6;
        RX_buffer[7] =   RBSR7;
        RX_buffer[8] =   RBSR8;
        RX_buffer[9] =   RBSR9;
        RX_buffer[10] =   RBSR10;
        RX_buffer[11] =   RBSR11;
        RX_buffer[12] =   RBSR12;
        CMR = 0X04;                         //释放 RXFIFO 中的空间
        temp = ALC;                         //释放仲裁随时捕捉寄存器
        temp = ECC;                         //释放错误代码捕捉寄存器
     }
}

//单片机外部中断1
void Ir_EX1( void ) interrupt 2
{
    void CAN_RXD(void)；                    //调用接收函数
}
```

接下来就要开始编辑一个利用硬件资源（数码管，INT0建）的驱动程式，




上面流程图是编程思维的视觉化效果。一旦程式运行就会先初始化数据然后初始化硬件，最终就会陷入“不停显示数码管”的死循环中。而 SJA1000发送报文和接收报文基本上是由中断服务而触发的。该实验的操作方式基本上与 PIAE 的点对点实验大同小异。

在指示方面仅数码管而已，第一二个数码管是显示发送源也就是点击变量的值，第三四个数码管则显示接收的报文的值。每当点击一下中断健，点击变量的值就会累加，而点击的次数会在本节点上显示，同时间还会将点击变量的值传送出去。当接受到报文时，报文中的数据也会显示在数码管上。


程式主要用来测试两个节点之间的通讯（如图），而程式只对节点1进行修改。节点2编程的方法大体上与节点1是一样的，不过只是修改“接收代码”和“报文标示符”而已。

这章包含之前所学习的基础，编程的过程中可能会与流程图出现出入，不过基本上编程思维都是一样的。

00-CAN 驱动程式.c 文件

```
//程式是利用 CAN 学习版上的硬件资源（数码管，中断健）
//建立的点对点的驱动程式
//akuei2上  10-10-09

#include "reg52.h"
#include "sja1000.h"
#define uchar unsigned char

//i0脚定义
sbit SJACS=P2^0;
sbit SJARST=P2^3;

//数码管码，与位选码
uchar code Led_Code[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0xff};
uchar code Led_Select[]={0xef,0xdf,0xbf,0x7f};

//接收代码变量,接收屏蔽变量,发送缓冲变量,接收缓冲变量,发送数据,接收数据,点击变量
uchar ACR[4],AMR[4],TXD_Buffer[13],RXD_Buffer[13],TXD_Data,RXD_Data,Click;

//函数声明
void Display(void);
void CAN_TXD(void);
void CAN_RXD(void);

//1毫秒延迟函数
void Delay_1ms(int x)
{
	int j;
	for(;x>0;x--)
		for(j=125;j>0;j--);
}

//50微秒延迟函数
void Delay_50us(int t)
{
	uchar j;
	t--;
	for(;t>0;t--)
	for(j=19;j>0;j--);
}
```

```
//变量初始化
void Init_Data(void)
{
    int i;

    ACR[0]='C';                          //源节点地址为 CAN1
    ACR[1]='A';
    ACR[2]='N';
    ACR[3]='1';                          //全部标示符无屏蔽
    AMR[0]=0x00;
    AMR[1]=0x00;
    AMR[2]=0x00;
    AMR[3]=0x03;

    TXD_Buffer[0]=0x88;                  //EFF 格式，数据长度8 的 TX 帧信息
    TXD_Buffer[1]='C';                   //目的标示符为 CAN2
    TXD_Buffer[2]='A';
    TXD_Buffer[3]='N';
    TXD_Buffer[4]='2';
    TXD_Buffer[5]=TXD_Data;              //发送数据
    TXD_Buffer[6]=0x00;
    TXD_Buffer[7]=0x00;
    TXD_Buffer[8]=0x00;
    TXD_Buffer[9]=0x00;
    TXD_Buffer[10]=0x00;
    TXD_Buffer[11]=0x00;
    TXD_Buffer[12]=0x00;

    for(i=0;i<13;i++)                    //接收缓冲变量初始化
    {
        RXD_Buffer[i]=0x00;
    }

    TXD_Data=0x00;                       //发送数据初始化
    RXD_Data=0x00;                       //接收数据初始化
}

//CPU 初始化
void Init_CPU(void)
{
    SJACS=1;                 // 使片选无效
    SJARST=1;                // 硬件复位无效

    IT0=1;                   // 外部中断0为电平触发方式
    IT1=0;                   // 外部中断1为跳电沿触发方式
```

```
    EX0=1;              // 外部中断0使能
    EX1=1;             // 外部中断1使能
    EA=1;              // 全局中断使能
}

//CAN 初始化
void Init_CAN(void)
{
    unsigned char bdata temp;     //建立存储在 bdata 中的临时变量 temp

    do
    {
        MODR=0x09;
        temp=MODR;   //模式寄存器-设置复位模式而且单验收滤波模式
    }
    while(!(temp&0x01));   //判断

    CDR=0x88;           //时钟分频寄存器-设置 PeliCAN 模式，Clock off 有效
    BTR0=0x31;           //总线时序寄存器-波特率为10kbps
    BTR1=0x1c;
    IER=0x01;            //中断使能寄存器-接收中断使能
    OCR=0xaa;           //输出控制寄存器-借签...
    CMR=0x04;            //命令寄存器-释放 RXFIF0

    ACR0=ACR[0];        //接收代码寄存器-本节点地址位 CAN1
    ACR1=ACR[1];
    ACR2=ACR[2];
    ACR3=ACR[3];

    AMR0=AMR[0];       //接收屏蔽寄存器-无任何屏蔽
    AMR1=AMR[1];
    AMR2=AMR[2];
    AMR3=AMR[3];       //最后两位为无用位，必须设置为逻辑1

    do
    {
        MODR=0x08;
        temp=MODR;      //模式寄存器-推出复位模式，保持单验收滤波模式
    }
    while(temp&0x01);     //判断...
}
```

```
//主函数
void main(void)
{
    Init_Data();
    Init_CPU();
    Init_CAN();
    while(1)
    {
        Display();
    }
}

//数码管显示函数
void Display(void)
{
    int Digit[4],i;
    Digit[3]=TXD_Data/10;                                   //第一个数码管显示发送源十位
    Digit[2]=TXD_Data%10;                                   //第二个数码管显示发送源个位
    Digit[1]=RXD_Data/10;                                   //第三个数码管显示接收源十位
    Digit[0]=RXD_Data%10;                                   //第四个数码管显示接受原个位
    for(i=0;i<4;i++)
    {
        P0=Led_Code[Digit[3-i]];                            //送数码管码
        P2=Led_Select[i];                                   //送位选码
        Delay_50us(20);                                     //延迟1微秒
    }
}

//CAN 发送函数
void CAN_TXD(void)
{
    unsigned char bdata temp;

    do
    {
        temp=SR;            //判断报文接收完毕？
    }
    while(temp&0x10);       //SR.4=0  发送闲置，SR.4=1  发送状态中

    do
    {
        temp=SR;            //判断最后报文请求？
    }
    while(!(temp&0x08));    //SR.3=0  没有余报文请求，SR.3=1  还存在报文请求
```

```
    do
    {
        temp=SR;              //判断 TXFIFO 是否锁定？
    }
    while(!(temp&0x04));   //SR.2=0 锁定，SR.2=1 开放

    TBSR0=TXD_Buffer[0];
    TBSR1=TXD_Buffer[1];
    TBSR2=TXD_Buffer[2];
    TBSR3=TXD_Buffer[3];
    TBSR4=TXD_Buffer[4];
    TBSR5=TXD_Buffer[5];
    TBSR6=TXD_Buffer[6];
    TBSR7=TXD_Buffer[7];
    TBSR8=TXD_Buffer[8];
    TBSR9=TXD_Buffer[9];
    TBSR10=TXD_Buffer[10];
    TBSR11=TXD_Buffer[11];
    TBSR12=TXD_Buffer[12];

    CMR=0x01;           //命令寄存器-发送请求
}

//CAN 接收函数
void CAN_RXD(void)
{
    unsigned char temp;

    temp = IR;
    if( temp & 0x01)                    //判断是否接收中断
    {
        RXD_Buffer[0]=RBSR0;            //读取 RXFIFO
        RXD_Buffer[1]=RBSR1;
        RXD_Buffer[2]=RBSR2;
        RXD_Buffer[3]=RBSR3;
        RXD_Buffer[4]=RBSR4;
        RXD_Buffer[5]=RBSR5;
        RXD_Buffer[6]=RBSR6;
        RXD_Buffer[7]=RBSR7;
        RXD_Buffer[8]=RBSR8;
        RXD_Buffer[9]=RBSR9;
        RXD_Buffer[10]=RBSR10;
        RXD_Buffer[11]=RBSR11;
        RXD_Buffer[12]=RBSR12;
```

```
        CMR = 0X04;                    //释放 RXFIFO 中的空间
        temp = AL                      //释放仲裁随时捕捉寄存器
        temp = ECC;                    //释放错误代码捕捉寄存器
    }
}

//外部中断0函数-点击中断
void Ir_EX0(void) interrupt 0
{
    Delay_1ms(10);                                          //软件消抖

    EA=0;                                                   //关闭全局使能
    EX0=0;                                                  //外部中断0使无效
    Click++;
    if(Click>=100) Click=0;
    TXD_Data=Click;                                         //发送数据赋值
    TXD_Buffer[5]=TXD_Data;
    CAN_TXD();                                              //调用发送函数
    EX0=1;                                                  //外部中断0使能
    EA=1;                                                   //全局中断使能

}

//外部中断1函数-RXFIFO 饱和中断
void Ir_EX1(void) interrupt 2
{
    EA=0;                                                   //关闭全局使能
    EX1=0;                                                  //外部中断1使无效
    CAN_RXD();                                              //调用接收函数
    RXD_Data=RXD_Buffer[5];                                 //接收数据变量赋值
    EX1=1;                                                  //外部中断1使能
    EA=1;                                                   //全局中断使能
}
```

==========

sja1000.h 文件

```
/*CAN 总线 SJA1000寄存器地址定义（用的是 PeliCAN 模式，扩展帧 EFF 模式）*/

unsigned    char    xdata    MODR    _at_    0xFE00;        // 模式寄存器
unsigned    char    xdata    CMR     _at_    0xFE01;        // 命令寄存器
unsigned    char    xdata    SR      _at_    0xFE02;        // 状态寄存器
unsigned    char    xdata    IR      _at_    0xFE03;        // 中断寄存器
unsigned    char    xdata    IER     _at_    0xFE04;        // 中断使能寄存器
unsigned    char    xdata    BTR0    _at_    0xFE06;        // 总线定时寄存器0 ； 总线波特率的选择
```

```
unsigned    char    xdata    BTR1    _at_    0xFE07;          // 总线定时寄存器1 ； 总线波特率的选择
unsigned    char    xdata    OCR     _at_    0xFE08;          // 输出控制寄存器

unsigned    char    xdata    ACR0    _at_    0xFE10;//16;
unsigned    char    xdata    ACR1    _at_    0xFE11;//17;
unsigned    char    xdata    ACR2    _at_    0xFE12;//18;
unsigned    char    xdata    ACR3    _at_    0xFE13;//19;   // 接收代码（0x16_0x19）;
unsigned    char    xdata    AMR0    _at_    0xFE14;//20;
unsigned    char    xdata    AMR1    _at_    0xFE15;//21;
unsigned    char    xdata    AMR2    _at_    0xFE16;//22;
unsigned    char    xdata    AMR3    _at_    0xFE17;//23;   // 掩码（0x20_0x23）;

unsigned    char    xdata    CDR     _at_    0xFE1F;//31;   // 时钟分频器
unsigned    char    xdata    ALC     _at_    0xFE0B;//11;   // 仲裁丢失捕捉寄存器
unsigned    char    xdata    ECC     _at_    0xFE0C;//12;   // 误码捕捉寄存器

unsigned    char    xdata    TBSR0    _at_    0xFE10;//16; //TX 帧信息(标准帧、扩展帧）寄存器
unsigned    char    xdata    TBSR1    _at_    0xFE11;//17; //TX 帧信息(标准帧数据1、扩展帧识别码1）寄存器
unsigned    char    xdata    TBSR2    _at_    0xFE12;//18; //TX 帧信息(标准帧数据1、扩展帧识别码2）寄存器
unsigned    char    xdata    TBSR3    _at_    0xFE13;//19; //TX 帧信息(标准帧数据1、扩展帧识别码3）寄存器
unsigned    char    xdata    TBSR4    _at_    0xFE14;//20; //TX 帧信息(标准帧数据2、扩展帧识别码4）寄存器
unsigned    char    xdata    TBSR5    _at_    0xFE15;//21; //TX 帧信息(标准帧数据3、扩展帧数据1）寄存器
unsigned    char    xdata    TBSR6    _at_    0xFE16;//22; //TX 帧信息(标准帧数据4、扩展帧数据2）寄存器
unsigned    char    xdata    TBSR7    _at_    0xFE17;//23; //TX 帧信息(标准帧数据5、扩展帧数据3）寄存器
unsigned    char    xdata    TBSR8    _at_    0xFE18;//24; //TX 帧信息(标准帧数据6、扩展帧数据4）寄存器
unsigned    char    xdata    TBSR9    _at_    0xFE19;//25; //TX 帧信息(标准帧数据7、扩展帧数据5）寄存器
unsigned    char    xdata    TBSR10   _at_     0xFE1A;//26; //TX 帧信息(标准帧数据8、扩展帧数据6）寄存器
unsigned    char    xdata    TBSR11   _at_     0xFE1B;//27; //TX 帧信息                (扩展帧数据7）寄存器
unsigned    char    xdata    TBSR12   _at_     0xFE1C;//28; //TX 帧信息                (扩展帧数据8）寄存器

unsigned    char    xdata    RBSR0    _at_    0xFE10;//16; //RX 帧信息(标准帧、扩展帧）寄存器
unsigned    char    xdata    RBSR1    _at_    0xFE11;//17; //RX 识别码(标准帧、扩展帧）寄存器1
unsigned    char    xdata    RBSR2    _at_    0xFE12;//18; //RX 帧信息(标准帧、扩展帧）识别码2寄存器
unsigned    char    xdata    RBSR3    _at_    0xFE13;//19; //RX 帧信息(标准帧数据1、扩展帧识别码3）寄存
器
unsigned    char    xdata    RBSR4    _at_    0xFE14;//20; //RX 帧信息(标准帧数据2、扩展帧识别码4）寄存
器
unsigned    char    xdata    RBSR5    _at_    0xFE15;//21; //RX 帧信息(标准帧数据3、扩展帧数据1）寄存器
unsigned    char    xdata    RBSR6    _at_    0xFE16;//22; //RX 帧信息(标准帧数据4、扩展帧数据2）寄存器
unsigned    char    xdata    RBSR7    _at_    0xFE17;//23; //RX 帧信息(标准帧数据5、扩展帧数据3）寄存器
unsigned    char    xdata    RBSR8    _at_    0xFE18;//24; //RX 帧信息(标准帧数据6、扩展帧数据4）寄存器
unsigned    char    xdata    RBSR9    _at_    0xFE19;//25; //RX 帧信息(标准帧数据7、扩展帧数据5）寄存器
unsigned    char    xdata    RBSR10   _at_    0xFE1A;//26; //RX 帧信息(标准帧数据8、扩展帧数据6）寄存器
unsigned    char    xdata    RBSR11   _at_    0xFE1B;//27; //RX 帧信息                (扩展帧数据7）寄存器
unsigned    char    xdata    RBSR12   _at_    0xFE1C;//28; //RX 帧信息                (扩展帧数据8）寄存器
```

程式主要是借签 PIAE 的点对点源程式，不过已经本本人习惯化了，说到底还是换汤不换药呀。见笑了。如果要编辑节点2，那么仅修改接收代码寄存器与发送报文的表示符就好了。那么修改源节点的地方法如下：

在程式中的的变量初始化函数中：

```
void Init_Data ( void )
{
    ...
    ...
    ACR[0]='C';
    ACR[1]='A';
    ACR[2]='N';
    ACR[3]='1';
  ...
  ...
    TXD_Buffer[1]='C';
    TXD_Buffer[2]='A';
    TXD_Buffer[3]='N';
    TXD_Buffer[4]='2';
}
```

以上该节点的接收代码和报文标示符都使用了字符赋值是，感觉上比起数字，字母比较容易记忆吧。 经 void Init_CAN ( void ) 执行后，接收代码寄存器和报文的发送表示符就会被初始化了。还有一点，就是屏蔽代码寄存器在程式中已经设置为无屏蔽，也就是说只有接收代码与相对应的报文有等于的标示符，否则报文无法存入 RXFIFO。

```
void Init_Data ( void )
{
    ...
    AMR[0]=0x00;
    AMR[1]=0x00;
    AMR[2]=0x00;
    AMR[3]=0x03;
    ...
}
```

除此之外，每当触发发生中断服务，应该习惯的将中断使能使无效，这个是为了万一，函数执行完以后将中断使能使能回去。