SJA1000 独立的 CAN 控制器应用指南

关键词：SJA1000 独立的 CAN 控制器 CAN2.0B PeliCAN

1. 介绍

控制器局部网（CAN）是一个串行的、异步的、多主机的通讯协议。SJA1000 是一个独立的 CAN 控制器，它在汽车和普通的工业应用上有先进的特征。由于硬件和软件的兼容，它将会替代 PCA82C200。它与 PCA82C200 相比具有更先进的特征，因此特别适合于轿车内的电子模块、传感器、制动器的连接和通用工业应用中，特别是系统优化，系统诊断和系统维护时特别重要。

本文倾向于在设计 SJA1000 为基础的 CAN 节点上引导用户，同时还提供典型的应用电路图和用于编程的流程图。

2. 概述

独立的 CAN 控制器 SJA1000 有 2 个不同的操作模式：

—Basic CAN 模式（PCA82C200 兼容）；

—PeliCAN 模式。

上电时，Basic CAN 模式是默认的操作模式。因此，已经使用 PCA82C200 开发出的硬件和软件可以直接被 SJA1000 使用而不用作任何修改。

PeliCAN 模式是操作的新模式，它能够处理所有的 CAN2.0B 定义的帧类型。而且它还提供一些增强功能使 SJA1000 能应用于更宽的领域。

2.1 SJA1000 特征

SJA1000 的特征能分成 3 组：

（1）已建立好的 PCA82C200 功能

这组的特征在 PCA82C200 里已经生效。

（2）提高的 PCA82C200 功能

部份这些功能在 PCA82C200 里已经生效。但是，在 SJA1000 里，它们在速度、大小和性能方面已得到提高。

（3）在 PeliCAN 模式里的增强功能

在 PeliCAN 模式里，SJA1000 支持一些错误分析功能，如支持系统诊断、系统维护、系统优化。而且这个模式里也加入了对一般 CPU 的支持和系统自身测试的功能。

在下面的表中，SJA1000 所有的特征已被列在表里，包括它们在应用中主要的优点。

表 1 SJA1000 应用中的优点

已建立好的 PCA82C200 功能

灵活的处理器接口	允许接入大部分的微型处理器和微型控制器。
可编程的 CAN 输出驱动器	对各种物理层的分界面。
CAN 位频率高达 1Mbit/s	SJA1000 覆盖了位频率的所有范围，包括高速应用。

提高的 PCA82C200 功能

CAN2.0B(隐性的)	SJA1000 的 CAN2.0B 隐性特征允许 CAN 控制器接收带有 29 位 ID 的信息。
64 个字节接收 FIFO	高达 21 条信息能被存储在接收 FIFO 中，这延长了最大中断服务时间，避免了数据溢出。
24MHz 时钟频率	较快的处理器访问和更多的位定时选择。
接收比较器旁路	缩短间隔延迟，由于一个改进的位定时编程，产生更高的 CAN 总线长度。

在 PeliCAN 模式里强大的功能

CAN2.0B（有效）	CAN2.0B 活跃支持带有 29 位 ID 网络扩展应用。
发送缓冲器	用于带有 11 位或 29 位 ID 信息的单个信息发送缓冲器。
增强的验收滤波器	两个验收滤波器模式，支持 11 位和 29 位 ID 过滤。
可读的错误计数器	支持错误分析，在标准相位和在正常操作期间可被用于：诊断、系统维护、系统优化。
可编程的错误警告限制	
错误代码捕捉寄存器	
错误中断	
仲裁丢失捕捉中断	支持系统优化包括信息等待时间分析。
单次发送	使软件命令最小化和允许快速重载发送缓冲器。
仅听模式	SJA1000 能够作为一个隐性的 CAN 监控器操作，可以分析 CAN 总线通信或自动的位速率检测。
自我测试模式	支持全部 CAN 节点的功能性自我测试或在一个系统内的自身接收。

2.2 CAN 节点结构

一般来说，每个 CAN 模块能够被分成不同的功能块。CAN 总线的连接通常由被优化的 **CAN 收发器**建立。收发器控制逻辑电平信号从 CAN 控制器到达总线上的物理层，反之亦然。

上面一层是一个 **CAN 控制器**，它执行在 CAN 规约里定义的 CAN 协议。它通常用于信息缓冲和验收滤波。

而所有这些 CAN 功能都被一个**模块控制器**控制，它用于执行功能性的应用。例如，控制调节器，读传感器和处理人—机接口（MMI）。

如图 1 所示，SJA1000 独立的 CAN 控制器总是位于微型控制器和收发器之间，在一般情况下，这个控制器是一个集成电路。

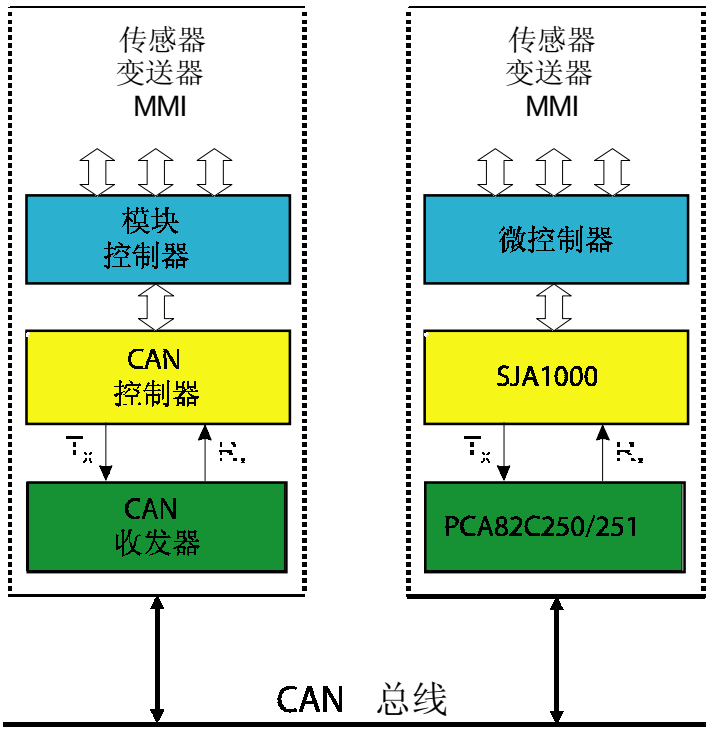

图 1 CAN 模块装置

2.3 方块图

下图是 SJA1000 的方块图。

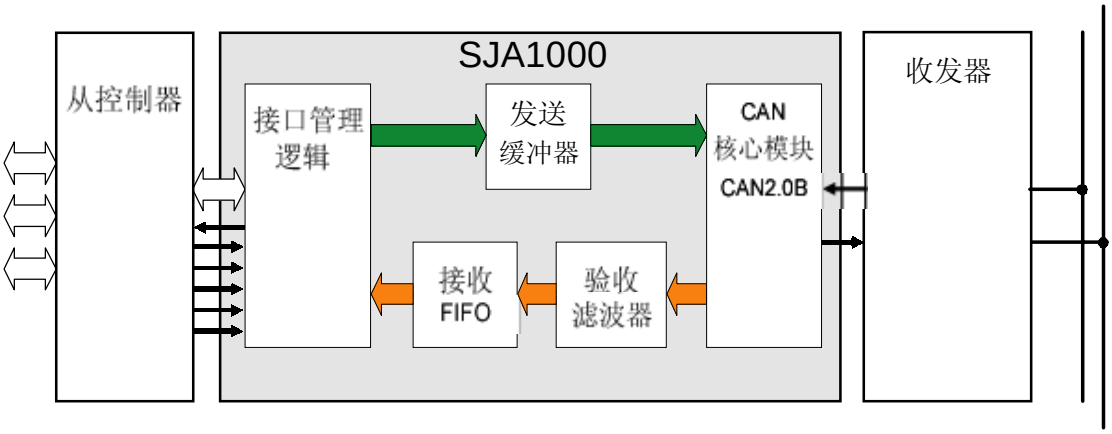

图 2　SJA1000 的方块图

根据 CAN 规约，**CAN 核心模块**控制 CAN 帧的发送和接收。

接口管理逻辑完成对外部主控制器的连接，该控制器能可以是微型控制器或其他器件。经过 SJA1000 复用的地址/数据总线访问寄存器和控制读/写选通信号都在这里处理。另外除了 PCA82C200 已有的 BasicCAN 功能，还加入了一个新的 PeliCAN 功能。因此，附加的寄存器和逻辑电路主要在这块里生效。

SJA1000 的**发送缓冲器**能够存储一个完整的信息（扩展的或标准的）。无论什么时候主控制器初始化发送，接口管理逻辑会迫使 CAN 核心块从发送缓冲器读 CAN 信息。

当收到一个信息时，CAN 核心块将串行位流转换成用于**验收滤波器的**并行数据。通过这个可编程的滤波器，SJA1000 能确定哪些信息实际上被主控制器收到。

所有收到的信息由验收滤波器接收并存储在**接收 FIFO**。储存信息的多少由工作模式决定，而最多能存储 32 个信息。因为数据溢出的可能性被大大降低，这使用户能更灵活地指定中断服务和中断优先级。

3. 系统

为了连接到主控制器，SJA1000 提供一个复用的地址/数据总线和附加的读/写控制信号。SJA1000 能被看作外围存储器并为主控制器映射 I/O 设备。

3.1 SJA1000 应用

SJA1000 的寄存器和管脚配置允许它使用于各种各样的集成的或分立的 CAN 收发器。这使不同微控制器之间的接口能够被灵活运用。

一个包括 80C51 微型控制器和 PCA82C251 收发器的典型 SJA1000 应用图如图 3 所示。CAN 控制器功能作为一个时钟源，复位信号由外部复位电路产生。在这个例子里，SJA1000 的片选由微控制器的 P2.7 口控制。否则，这个片选输入必须接到 VSS。也可以通过地址解码控制，例如，当地址/数据总线用于其他外围器件。

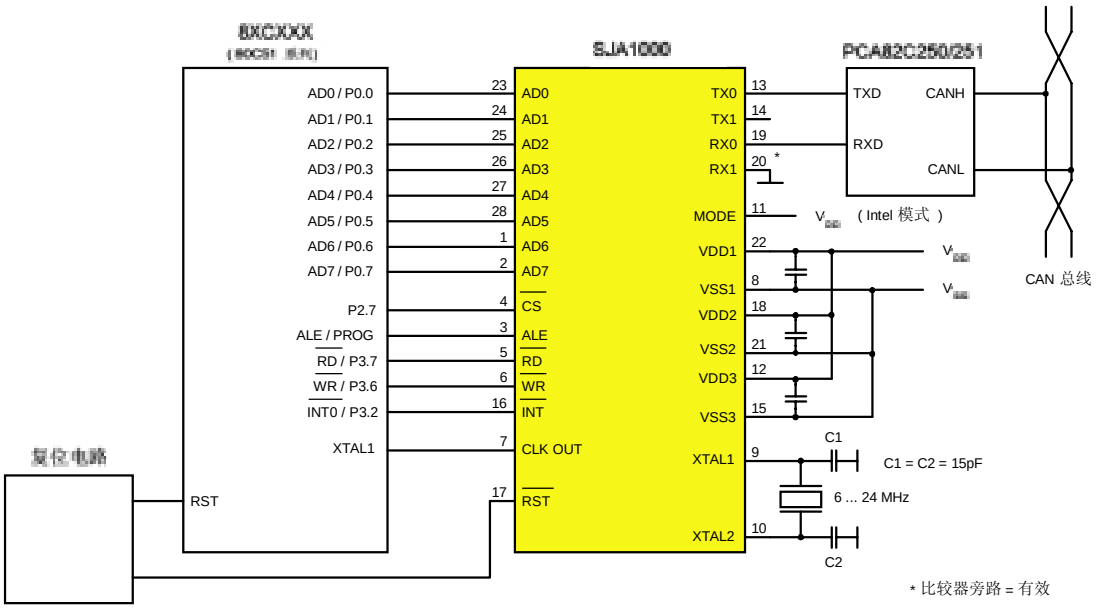

图 3　典型的 SJA1000 应用

3.2　电源

SJA1000 有三组电源引脚，用于 CAN 控制器内部不同的数字和模拟模块。

VDD1/VSS1：	内部逻辑	（数字）
VDD2/VSS2：	输入比较器	（模拟）
VDD3/VSS3：	输出驱动器	（模拟）

为了更好 EME，电源应该分开。例如用于比较器的 VDD2 可一个 RC 滤波器解耦来抑制噪音。

3.3　复位

为了得到一个恰当的复位，一个稳定的振荡器时钟必须接在在 CAN 控制器的 XTAL1 管脚上（见 3.4 章）。引脚 17 上的外部复位需要被同步并被内部延长到 15 个 t_{XTAL}。这保证了 SJA1000 所有的寄存器正确的复位（见[1]）。要注意的是上电时必须要考虑振荡器的起振时间。

3.4　振荡器和时钟策略

SJA1000 能使用片内振荡器或带有片外时钟源工作。另外 CLK OUT 管脚可被使能，为主控制器输出时钟频率。图 4 显示了四个不同的计时原理。如果不需要 CLK OUT 信号，可以通过置位被时钟寄存器（clock Off＝1）关掉。这将提高 CAN 节点的 EME 性能。

CLK OUT 信号的频率可通过时钟分频寄存器改变：

$f_{CLKOUT}=f_{XTAL}$/时钟分频因子（1，2，4，6，8，10，12，14）。

上电时或硬件复位，时钟分频因子的默认值取决于所选的接口模式（引脚 11）。如果使用 16MHz 的晶振，在 Intel 模式下，CLK OUT 的频率是 8 MHz。在 Motorola 模式，复位之后的时钟分频因子是 12，这种情况会产生 1.33MHz。

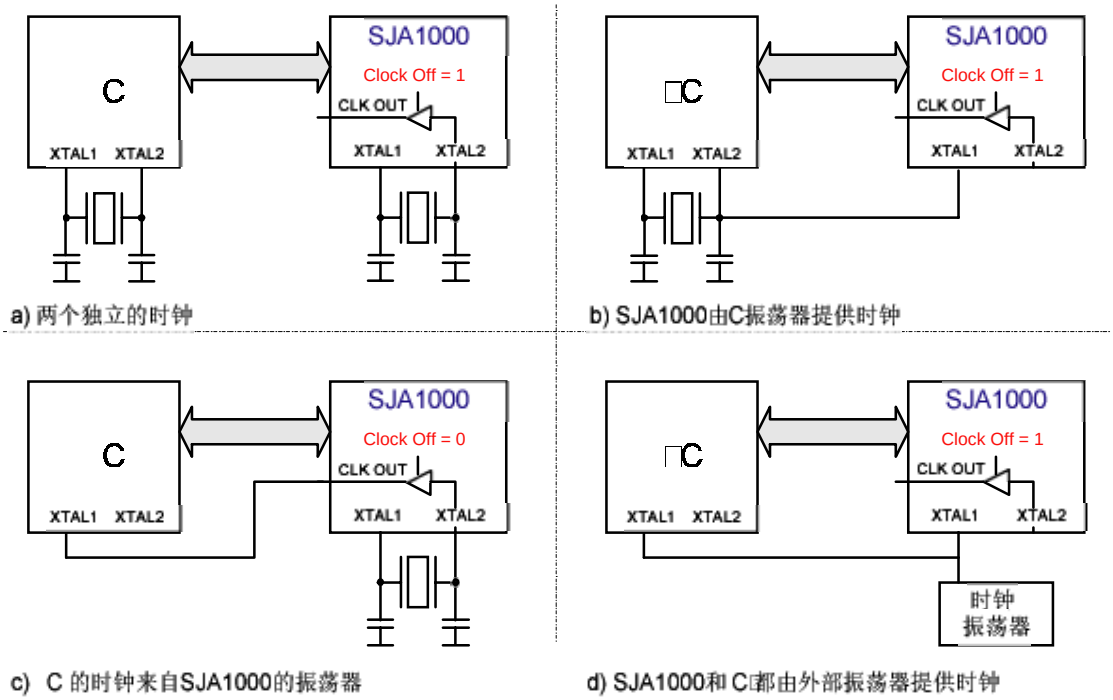

图 4 时钟方案

3.4.1 睡眠和唤醒

置位命令寄存器中的睡眠位（BasicCAN 模式）或模式寄存器（PeliCAN 模式）的睡眠模式位后，如果没有总线活动和中断等待，SJA1000 就会进入睡眠模式。振荡器保持运行直到已过了 15 个 CAN 位。此时，允许微型控制器与 CLK OUT 频率同步来进入低功耗模式。

如果三个唤醒条件之一[1]发生，振荡器会再次启动并产生一个唤醒中断。振荡器稳定下来后时，CLK OUT 频率被激活。

3.5 CPU 接口

SJA1000 支持对两个著名的微型控制器系列的直接连接：80C51，68xx。通过 SJA1000 的 MODE 引脚，可选择接口模式：

Intel 模式： MODE＝高
Motorola 模式： MODE＝低

在 Intel 模式和 Motorola 模式里，地址/数据总线和读/写控制信号的连接如图 5 所示。飞利浦基于 80C51 系列 8 位微控制器和带有 XA 结构的 16 位微型控制器，都使用 Intel 模式。

为了和其他控制器的地址／数据总线和控制信号匹配，必须要附加逻辑电路。但是必须确保在上电期间不产生写脉冲。另一个方法在这个时候使片选输入高电平，CAN 控制器无效。

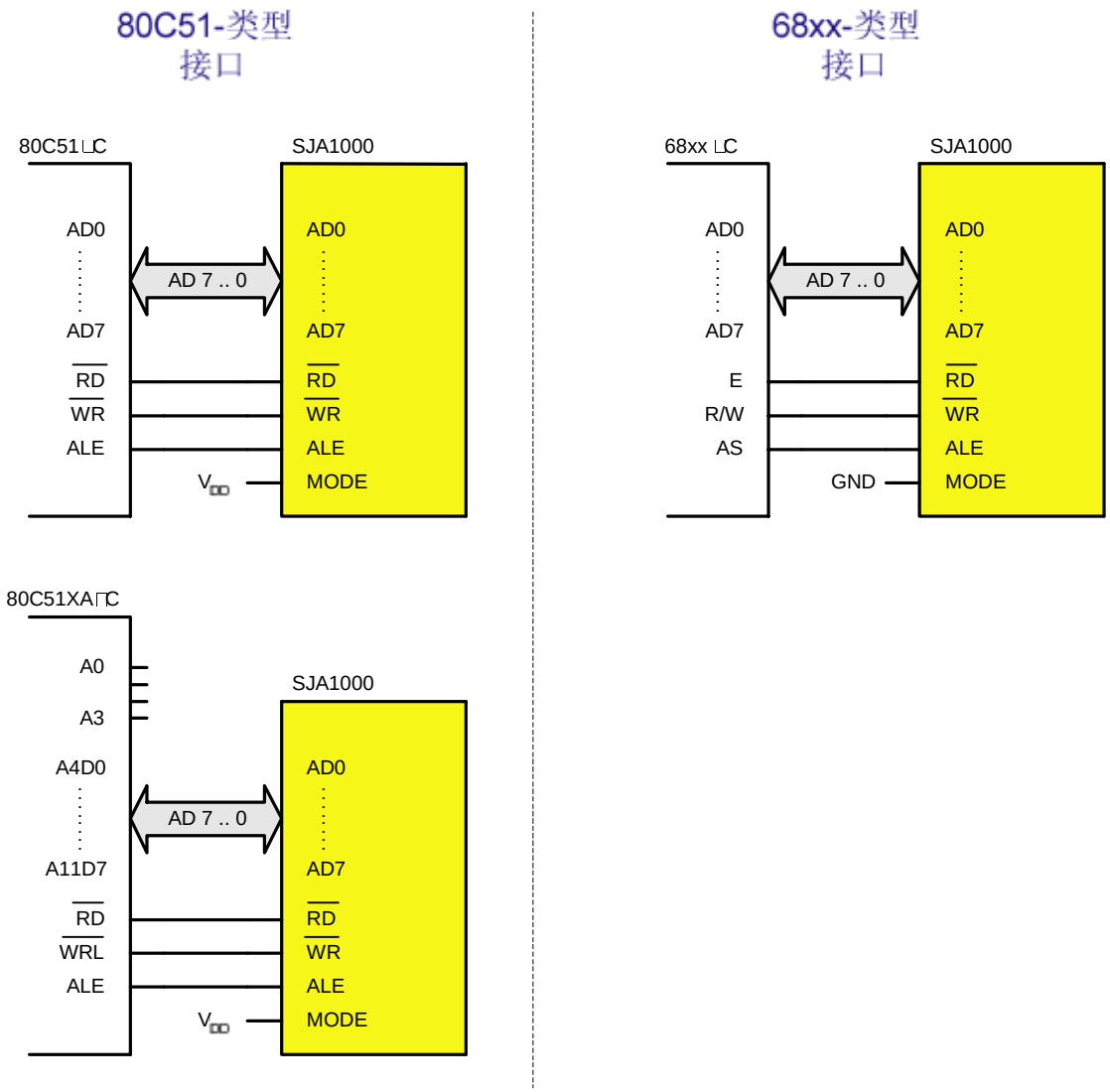

图 5　SJA1000 的 CPU 时钟接口

3.6　物理层接口

为了和 PCA82C200 兼容，SJA1000 包括一个模拟接收输入比较器电路。如果收发器的功能由分立元件实现，这个集成的比较器就能使用。

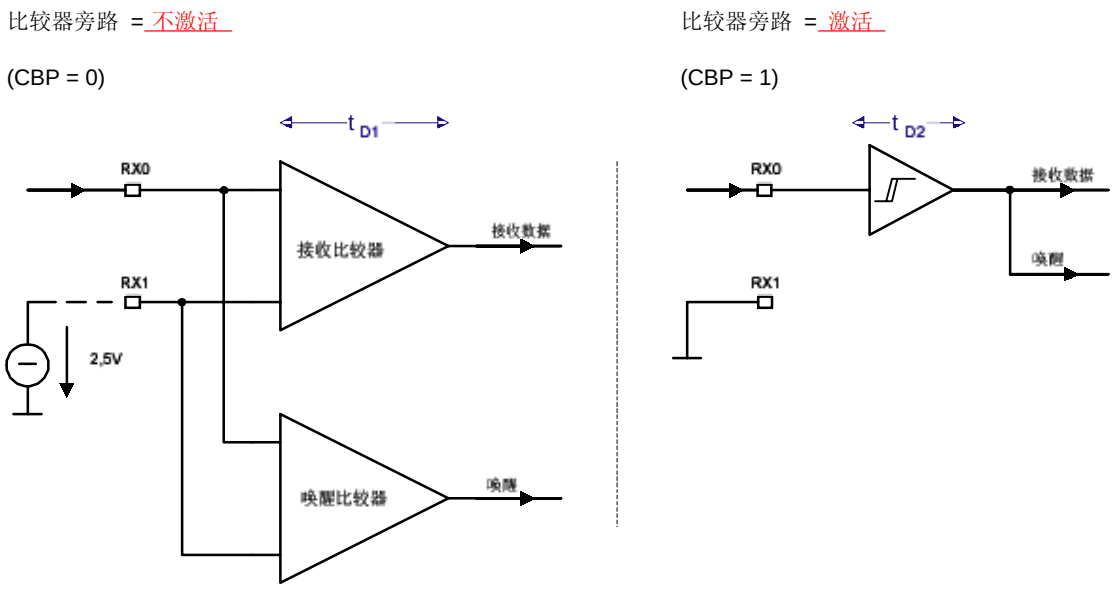

图 6　SJA1000 接收输入比较器

如果外部集成收发器电路有效，而比较器旁路功能在时钟分频寄存器里无效，RX1 必须被连接到 2.5V 的参考电压上（现存的收发器参考电压输出）。图 6 显示了两种设置的相应电路：

CBP＝激活，CBP＝非激活。另外唤醒信号的通道被引出。

一个集成的收发器电路使用所有的新应用，都建议激活 SJA1000 的比较器旁路功能（图 7）。如果这个功能被使能，施密特触发器有效，内部的传播延迟 t_{D2} 比接收比较器延迟的 t_{D1} 要小得多。这在最大总线长度[8]上有积极的影响。另外，它在休眠模式里将显著地降低电流。

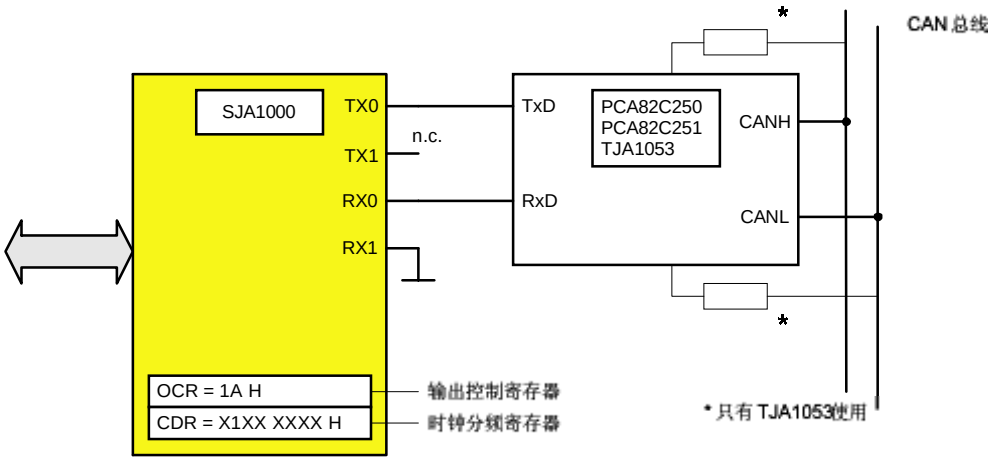

图 7　带有集成收发电路的标准应用

4. CAN 通迅的控制

4.1 控制 SJA1000 的基本功能和寄存器

SJA1000 由主控制器的程序进行功能配置和激活。因此 SJA1000 能满足不同属性的 CAN 总线系统的要求。主控制器和 SJA1000 之间的数据交换经过一系列的寄存器（控制段）和一个 RAM（信息缓冲器）完成。对于主控制器来说，构成发送和接收缓冲器的寄存器和一部份 RAM 的地址窗口，就象是外围的寄存

器。

表 2 列出了这些寄存器，并根据它们在系统中的作用分组。

注意，一些寄存器只在 PeliCAN 模式有效，控制寄存器就仅在 BasicCAN 模式里有效。而且一些寄存器是只读的或只写的，还有一些只能在复位模式时被访问。

关于寄存器的读（和 / 或）写访问，位定义和复位值的更多信息，可在数据表[1]中找到。

表 2 SJA1000 内部寄存器的分类

使用类型	寄存器名称（符号）		寄存器地址		功能
			PeliCAN 模式	BasicCAN 模式	
选择不同的操作模式的要素	模式	（MOD）	0	—	休眠—，验收滤波器—，自我检测—，仅听—，和复位模式选择。
	控制	（CR）	—	0	BasicCAN 模式里的复位模式选择。
	命令	（CMR）	—	1	BasicCAN 模式里的休眠模式命令。
	时钟分频	（CDR）	31	31	PeliCAN 模式、比较器旁路模式或 TX1（管脚 14）输出模式的 CLKOUT（管脚 7）时钟信号的设定选择
设定 CAN 通讯的要素	验收代码 屏蔽	（ACR） （AMR）	16—29 20—23	4， 5	验收滤波器位模式的选择。
	总线定时寄存器 0 1	（BTR0） （BTR1）	6 7	6 7	位时序参数的设置
	输出控制	（OCR）	8	8	输出驱动器属性的选择
	命令	（CMR）	1	1	用于自我接收、清除数据溢出、释放接收缓冲器、发送中止和发送请求的命令。
	状态	（SR）	2	2	信息缓冲器的状态，CAN 核心模块的状态。
	中断	（IR）	3	3	CAN 中断标志。
	中断使能	（IER）	4	—	在 PeliCAN 模式里中断的使能和禁能。
	控制	（CR）	—	0	在 BasicCAN 模式里中断的使能和禁能。
复杂的错误检测和分析的要素	仲裁丢失捕捉	（ALC）	11	—	显示仲裁丢失位的位置。
	错误代码捕捉	（ECC）	12	—	显示最后一次错误类型和位置。
	错误警告限制	（EWLR）	13	—	产生错误警告中断的限值选择。
	RX 错误计数	（RXERR）	14	—	反映接收错误计数器的当前值。
	TX 错误计数	（TXERR）	14，15	—	反映发送错误计数器的当前值。
	Rx 信息计数器	（RMC）	29	—	接收 FIFO 里的信息数目。
	RX 缓冲器开始地址	（RBSA）	30	—	显示在接收缓冲器里有效信息的当前内部 RAM 地址。
信息缓冲器	发送缓冲器	（TXBUF）	16—28	10—19	
	接收缓冲器	（RXBUF）	16—28	20—29	

4.1.1 发送缓冲器/接收缓冲器

要在 CAN 总线上发送的数据被载入 SJA1000 的存储区，这个存储区叫“发送缓冲器”。从 CAN 总线上收到的数据被存储在 SJA1000 的存储区，这个存储区叫“接收缓冲器”。这些缓冲器包括 2，3 或 5 个

字节的 ID 和帧信息（取决于模式和帧类型），而最多可以包含 8 个数据字节。关于数据缓冲器的各位定义和组成的更多信息，见数据表[1]。

- BasicCAN 模式： 这些缓冲器是 10 个字节长（见表 3）
  —2 个 ID 字节。
  —最高 8 个数据字节。
- PeliCAN 模式： 这些缓冲器是 13 个字节长（见表 4）
  —1 字节帧信息。
  —2 个或 4 个 ID 字节（标准帧或扩展帧）
  —最高 8 个数据字节。

表 3 在 BasicCAN 模式里 RX 和 TX 缓冲器列表

CAN 地址（十进制）	名称	组成和标注
TX—缓冲器： 10 RX—缓冲器： 20	ID 字节 1	8 个 ID 位
TX—缓冲器： 11 RX—缓冲器： 21	ID 字节 2	3 个 ID 位，1 个远程发送请求位。4 个数据长度代码位，表示数据字节的数目。
TX—缓冲器： 12—19 RX—缓冲器： 22—29	数据字节 1—8	由数据长度代码指明，最高 8 个数据字节

表 4 PeliCAN 模式里的 RX-[1.]（读访问）和 TX—缓冲器（写访问[2.]）

CAN 地址（十进制）	名称	组成和标注
16	帧信息	1 个位说明，如果信息包括一个标准的或扩展帧 1 个远程发送请求位 4 个数据长度代码位，说明数据字节的数目。
17，18	ID 字节 1，2	标准帧：11 个 ID 位 扩展帧：16 个 ID 位
19，20	ID 字节 3，4	仅扩展帧：13 个 ID
帧类型 标准：19—26 扩展：21—28	数据字节 1—8	由数据长度代码说明，最高 8 个数据字节

1. 整个 FIFO（64 个字节）能通过 CAN 地址 32—95 访问（见 5.1 章）。
2. TX—缓冲器的读访问可通过 CAN 地址 96—108 完成（也见 5.1 章）。

4.1.2 验收滤波器

独立的 CAN 控制器 SJA1000 装配了一个多功能的验收滤波器，该滤波器允许自动检查 ID 和数据字节。使用这些有效的滤波方法，对于某个节点来说，无效的信息可被防止存储在接收缓冲器里。因此降低了主控制器的处理负载。

滤波器由验收代码和屏蔽寄存器根据数据表[1]给定算法来控制。接收到的数据会和验收代码寄存器中的值进行逐位比较。接收屏蔽寄存器定义与比较相关的位的位置（0=相关，1=不相关）。只有收到信息的**相应**的位与验收代码寄存器相应的位相同，这条信息才会被接收。

BasicCAN 模式里的验收滤波

在这个模式，SJA1000 可以即插即用地取代 PCA82C200（硬件和软件）。因此验收滤波功能与 PCA82C200 的一样，也可以使用。这个滤波器是由两个 8 位寄存器控制—验收代码寄存器（ACR）和验收屏蔽寄存器（AMR）。CAN 信息的 ID 的 8 个最高位和这些寄存器里值相比较，见图 8。因此可以定义若干组的 ID 为被任何一个节点接收。

例子：

验收代码寄存器包括：

验收屏蔽寄存器包括：

带有 11 位的 ID 信息被接收。

MSB							LSB			
0	1	1	1	0	0	1	0			
0	0	1	1	1	0	0	0			
0	1	X	X	X	0	1	0	X	X	X
ID.10										ID.0

（X＝不起作用）

在验收屏蔽寄存器里包括“1”的位位置上，ID 在这位的任何值都被允许。对于三个最低位同样。因此 64 个不同的 ID 在这个例子里可被接收。其他位置的位必须等于验收代码寄存器相应位的值。

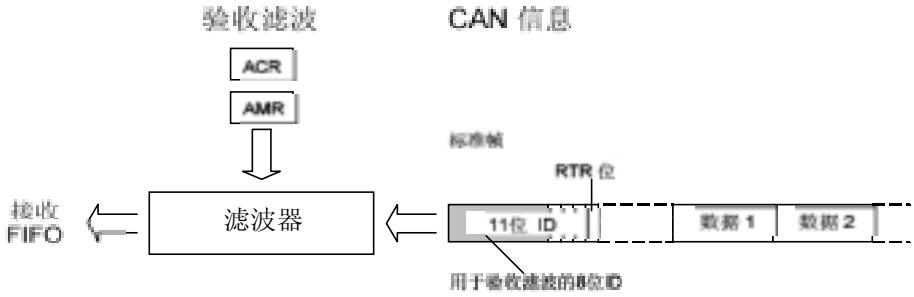

图 8　在 BasicCAN 模式里的验收滤波

Pelican 里的验收滤波

对于 Pelican 模式，验收滤波已被扩展：4 个 8 位验收代码寄存器（ACR0，ACR1，ACR2 和 ACR3），验收屏蔽寄存器（AMR0，AMR1，AMR2 和 AMR3）对于通用的信息滤波有效。这些寄存器可用于控制一个长的滤波器或两个短的滤波器，如图 9 和图 10 所示。信息的哪些位用于验收滤波，取决于收到的帧（标准的或扩展的）和选择的滤波器模式（单个或双滤波器）。表 5 给了更多关于信息的哪些位和验收代码和屏蔽位相比较的信息。从数字和表中看出，标准帧的验收滤波可以包括 RTR 位甚至数据字节。如果不需要经过验收滤波的信息位（例如信息被定义为验收），验收屏蔽寄存器必须包括一个“1”在相应的位位置上。

如果一条信息不包括数据字节（例如在一个远程帧或如果数据长度代码为零）但是验收滤波器中有数据字节，信息会被接收（如果 ID 到 RTR 位有效）。

例 1：

先假定，在 18 页上描述的同样的 64 个标准帧信息要在 PeliCAN 模式里滤波。

这可以通过使用一个长滤波器完成（单滤波器模式）。

验收代码寄存器（ACRn）和验收屏蔽寄存器（AMRn）包括：

N	0	1（高四位）	2	3
ACRn	01XX　X010	XXXX	XXXX　XXXX	XXXX　XXXX
AMRn	0011　1000	1111	1111　1111	1111　1111
接收的信息（ID.28…ID.18，RTR）	01xx　x010	xxxx		

（“X”＝不相关，“x”＝无效，只使用了 ACR1 和 AMR1 的高四位）。

在验收屏蔽寄存器的包括“1”的位位置上，ID 在相应位上的任何值都被允许，如远程发送请求位和数据字节 1 和 2 的位。

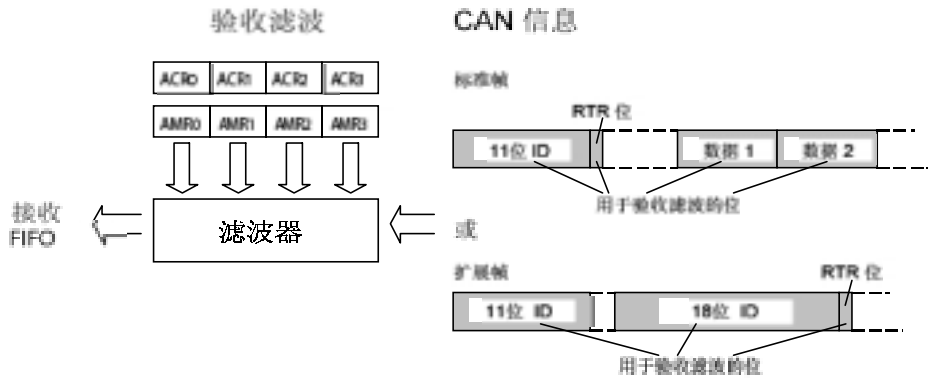

图 9　在 Pelican 模式里的验收滤波（单个滤波器模式）

例 2：

假定下面 2 条带有标准帧 ID 的信息被接收而没有经过识别位的进一步解码。数据和远程帧必须被正确接收。数据字节没有包含在验收滤波器中。

信息 1：　（ID.28）　1***0***11　***1***100　101（ID.18）
信息 2：　（ID.28）　1***1***11　***0***100　101（ID.18）

使用单滤波器模式会导致可以接收到四个信息而不仅是要求的两个：

N	0		1（高 4 位）	2		3	
ACRn	1X11	X100	101X	XXXX	XXXX	XXXX	XXXX
AMRn	0100	1000	0001	1111	1111	1111	1111
接收的信息（ID.28…ID.18，RTR）	1011	0100	101x				
	1111	0100	101x	（信息 2）			
	1011	1100	101x	（信息 1）			
	1111	1100	101x				

（“X”＝不相关，“x”＝无效，只使用了 ACR1 和 AMR1 的高四位）。

这个结果不满足不进一步解码而接收两条信息的要求。

使用双滤波器有正确的结果：

	滤波器 1					滤波器 2		
N	0		1		3 低四位	2		3 高四位
ACRn	1011	1100	101X	XXXX	… XXXX	1111	0100	101X …
AMRn	0000	0000	0001	1111	… 1111	0000	0000	0001 …
接收的信息（ID.28…ID.18，RTR）	1011	1100	101X（信息 1）			1111	0100	101X（信息 2）

（“X”＝不相关，“x”＝无效）。

信息 1 被滤波器 1 接收，信息 2 被滤波器 2 接收。如果信息至少被两个滤波器中的一个接收，信息就被存进接收 FIFO。这种方法可满足于这种要求。

例 3：

在这个例子里，使用一个长单验收滤波器滤波一组带有扩展帧 ID 的信息。

N	0		1		2		3（高六位）	
ACRn	1011	0100	1011	000X	1100	XXXX	0011	0XXX
AMRn	0000	0000	0001	0001	0000	1111	0000	0111
接收的信息（ID.28…ID.18，RTR）	1011	0100	1011	000x	1100	Xxxx	0011	0x

（“X”＝不相关，“x”＝无效，只使用了 ACR1 和 AMR1 的高六位）。

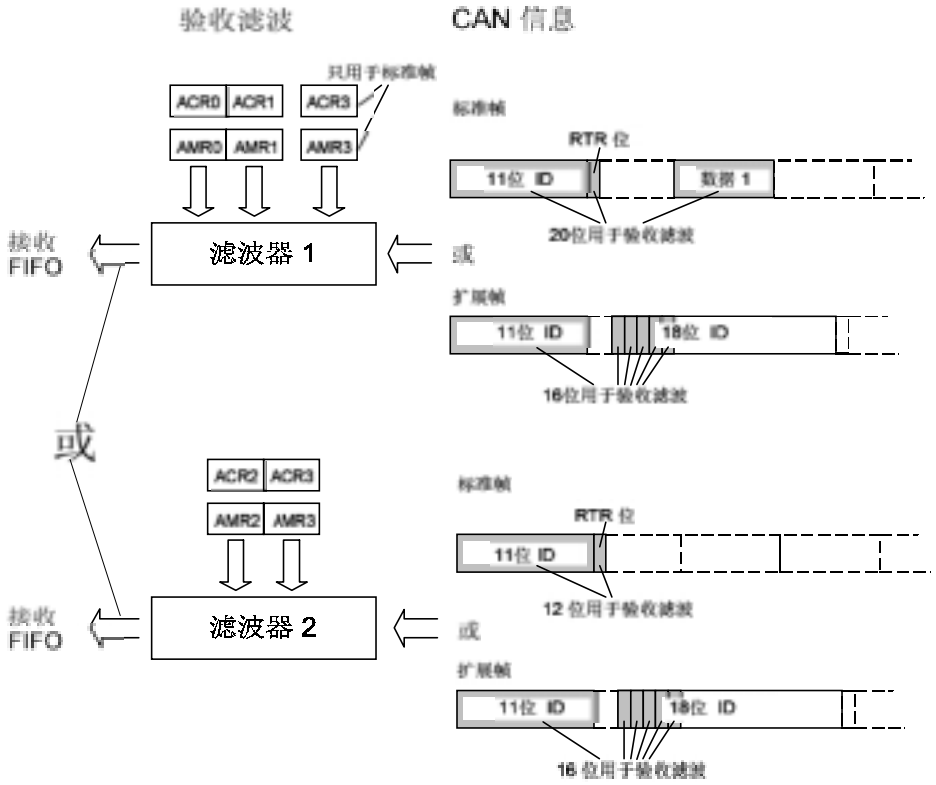

图 10　在 Pelican 模式里的验收滤波（双滤波器模式）

例 4：

有些系统仅使用标准帧，用 11 位 ID 和头两个数据字节确认信息。如果信息包括超过 8 个数据字节，就会使用像这样的协议，例如，在 Devicenet 中，将头两个数据字节定义为信息头部和使用分段存储协议。对于这个系统类型，SJA1000 在单滤波器模式里能滤波两个数据字节，在双滤波器模式里能过滤一个数据字节（除了 11 位 ID 和 RTR 位）。

使用双滤波器模式，下面的例子显示了在这样系统里信息的有效过滤：

	滤波器 1								滤波器 2			
N	0		1			3 低四位			2		3 高四位	
ACRn	1110	1011	0010	1111	…		1001		1111	0100	XXX0	…
AMRn	0000	0000	0000	0000	…		0000		0000	0000	1110	…
接收的信息	1110	1011	0010	1111	…		1001		1111	0100	xxx 0	
	ID+RTR				头一个数据字节				ID			RTR

（“X”＝不相关，“x”＝无效）。

滤波器 1 用来过滤信息，信息带有：

－ID“11101011001”

－RTR＝“0”也就是说仅数据帧和

－数据字节“11111001”（这指如 DeviceNet：一个信息的所有段都被过滤）。

滤波器 2 用来过滤一组 8 个信息，信息带有：

－ID“11110100 000”到“11110100111”和

－RTR＝“0”，也就是仅数据帧。

表 5 PeliCAN 模式里验收滤波器总结

帧类型	单滤波器模式（图 9）	双滤波器模式（图 10）
标准	验收的信息位： －11 位 ID －RTR 位 －第一个数据字节（8 位） －第二个数据字节（8 位） 使用的验收滤波器和屏蔽寄存器： －ACR0 或 ACR1/ACR2/ACR3 的高四位 －AMR0 或 AMR1/AMR2/AMR3 的高四位 （接收屏蔽寄存器的未使用的位应设为“1”）	滤波器 1 验收的信息位： －11 位 ID －RTR 位 －第一个数据字节（8 位） 使用的验收代码和屏蔽寄存器： －ACR0/ACR1 或 ACR3 的低四位 －AMR0/AMR1 或 AMR3 的低四位 滤波器 2 用于验收测试的信息位： －11 位 ID －RTR 位 使用的验收代码和标志寄存器： －ACR2/ACR3 的高四位 －AMR2/AMR3 的高四位
扩展	验收的信息位： －11 位基本的 ID －18 位扩展的 ID －RTR 位 －ACR0/ACR1/ACR2 或 ACR3 的高六位 －AMR0/AMR1/AMR2 或 AMR3 的高六位 （验收屏蔽寄存器的未使用的位应设为“1”）	滤波器 1 验收的信息位： －11 位基本 ID －扩展 ID 的 5 个最高位 使用的验收代码和屏蔽寄存器： －ACR0/ACR1 和 AMR0/AMR1 滤波器 2 用于验收测试的信息位： －11 位基本的 ID －扩展 ID 的 5 个最高位 使用的验收代码和屏蔽寄存器： －ACR2/ACR3/和 AMR2/AMR3

4.2 CAN 通讯的功能

通过 CAN 总线建立通讯的步骤是：

- 系统上电后
  - －设定主控制器连接到 SJA1000 的硬件和软件。
  - －设定用于通讯的 CAN 控制器，关于模式、验收滤波器和位时序等等的方面。－在 SJA1000 硬件复位后也要完成这些。
- 在应用的主过程中
  - －准备要发送的信息和激活 SJA1000 以发送它们。
  - －在被 CAN 控制器所接收的信息起作用。
  - －在通讯期间，发生的错误起作用。

图 11 表示了程序的总体流程。在接下来的段落里，会详细地解说那些直接控制 SJA1000 的流程。

4.2.1 初始化

如上面提到的一样，独立的 CAN 控制器 SJA1000 必须在上电后硬件复位后设定用于 CAN 通讯。在由主控制器操作期间，该主控制器可能发送一个（软件）复位请求，SJA1000 会被重新配置（再次初始化）。流程图如图 12。一个使用 80C51 微型控制器衍生的编程例子在这章里给出。

上电后，主控制器运行自己的特殊复位程序然后进入设定 SJA1000 的程序。因为图 11 的“构成控制线…”的部份被指定到使用的控制器，所以这里不作讨论。但是如这章所示，如何配置一个 80C51 衍生器件。

初始化过程的描述见图 12。假定，在上电后独立 CAN 控制器得到一个复位脉冲（低电平）在管脚 17，使它能够进入复位模式。在对 SJA1000 寄存器设定前，主控制器通过读复位模式/请求标志来检查 SJA1000 是否已达到复位模式，因为要得到配置信息的寄存器仅在复位模式能写入。

在复位模式,主控制器必须配置 SJA1000 控制段的寄存器：

- 模式寄存器（仅在 Pelican 模式里），在下面的模式中选择操作应用：
  - 验收滤波器模式
  - 自我测试模式
  - 仅听模式
- 时钟分频寄存器，定义
  - 如果使用 BasicCAN 或 PeliCAN 模式
  - 如果使能 CLKOUT 管脚
  - 如果旁路 CAN 输入比较器
  - 如果 TX1 输出被用作专门的接收中断输出
- 验收代码和验收屏蔽寄存器
  - 为收到的信息定义验收代码
  - 为和验收代码的相关位比较的信息的相关位定义验收屏蔽
- 总线定时寄存器，见[6]
  - 定义总线上的位速率
  - 定义位周期（位采样点）上的采样点
  - 定义在一个位周期里采样的数目
- 输出控制寄存器
  - 定义 CAN 总线输出管脚 TX0 和 TX1 的输出模式
    正常输出模式，时钟输出模式，双相位输出模式或测试输出模式
  - 定义 TX0 和 TX1 输出管脚配置
    悬空，下拉，上拉或推挽和极性。

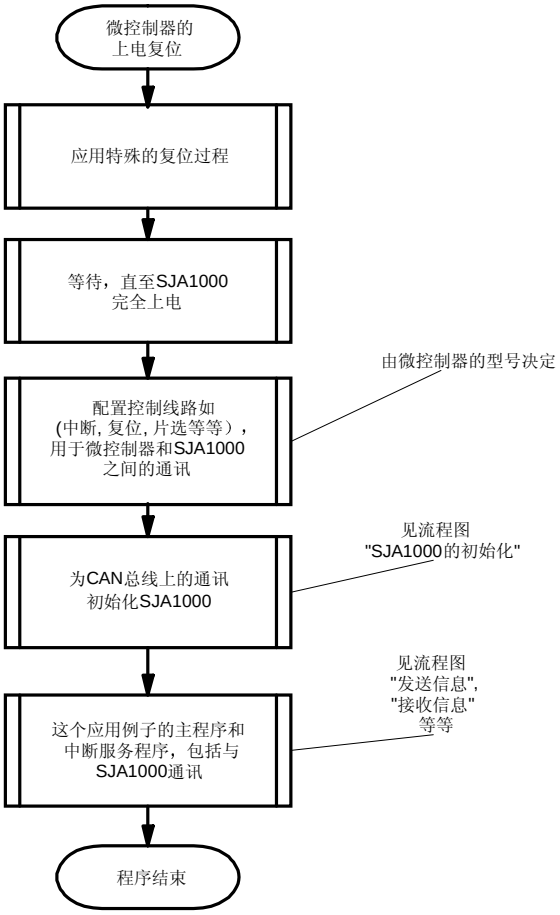
微控制器的
上电复位
应用特殊的复位过程
等待，直至SJA1000
完全上电
配置控制线路如
(中断, 复位, 片选等等)，
用于微控制器和SJA1000
之间的通讯
由微控制器的型号决定
为CAN总线上的通讯
初始化SJA1000
见流程图
"SJA1000的初始化"
这个应用例子的主程序和
中断服务程序，包括与
SJA1000通讯
见流程图
"发送信息",
"接收信息"
等等
程序结束

图 11　程序总体流程

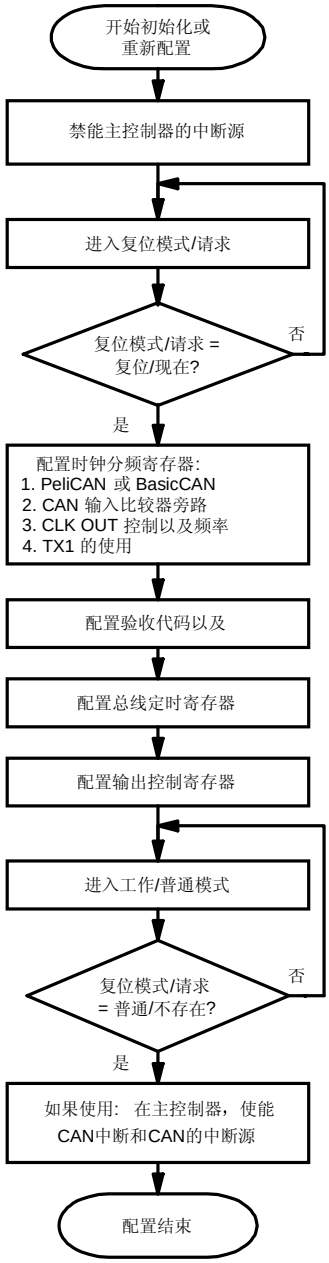

图 12 流程图“SJA1000 的初始化”

在将这个信息发送到 SJA1000 的控制段后，它会清除复位模式/请求标志，进入工作模式。要必须先检查标志是否确实被清除，才能进一步进入工作模式。这通过循环阅读标志完成。

在硬件复位等待期间（管脚 17 仍是低电平），不能清除复位模式/请求标志，因为这将迫使复位模式/请求标志变成“复位/当前”（查阅数据表可得到进一步的信息[1]）。因此这个循环是用于不断尝试清除标志和检查是否成功离开复位模式。

进入工作模式后，CAN 控制器的中断可被使能（如果适合的话）。

例子：SJA1000 的配置和初始化

这个例子是基于第 11 页上的图 3 给的应用例子。在下面编程例子里，假定微型控制器 S87C654 是主

控制器。它以 SJA1000 输出的时钟作为时钟信号。在上电期间，一个复位电路为微型控制器和 CAN 控制器提供硬件复位信号。在复位[1]期间，SJA1000 的时钟分频寄存器清零。因此 CAN 控制器进入 BasicCAN 模式，且时钟输出使能，当晶振起振后，CAN 控制器能够给 S87C654 传送时钟信号。时钟的频率是 $f_{CLK}/2$，因此管脚 11 连接到 80C51 系列的控制器。收到这个时钟信号后，微控制器开始它自已的复位过程（如图 11 所示）。

在附录里，给出了不同的常数和变量等等的定义。变量在 BasicCAN 和 Pelican 模式里的表示可以不同。例如“InterruptEnReg”在 BasicCAN 模式里是指控制寄存器但在 Pelican 模式里是指中断使能寄存器。语言“C”被用作编程。

在这个例子里，假定 CAN 控制器在 PeliCAN 模式里使用之前必须被初始化。然后，很容易地就可以从上面得到 BasicCAN 模式里相应的初始化程序。

第一步必须在主控制器和 SJA1000 之间设定一个通讯连接（片选，中断等等），如图 11 的“构成控制线…”。

/*定义中断优先级和控制（电平－激活，见 4.2.5 章）		*/
PX0=PRIORITY_HIGH;	/*设 CAN 是高优先级中断	*/
IT0＝INTLEVELACT;	/*中断 0 为电平激活	*/
/*SJA1000 的通讯接口使能		*/
CS＝ENABLE_N;	/*SJA1000 接口使能	*/
/*通讯连接的定义结束－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－		*/

第二步是初始化 SJA1000 的所有内部寄存器。因为一些寄存器在仅复位模式期间可被写，所以在写入之前必须检查。上电后，SJA1000 被设定为复位模式，如果复位模式已被置位，在循环里面可以检查到。

```
/*中断禁能，如果使用（上电后不是必然的）                                  */
EA＝DISABLE;                      /*所有中断禁能                          */
SAJIntEn=DISABLE;                 /*来自 SJA100 的外部中断禁能            */

/*设定复位模式/请求位（注：上电后，SJA1000 处于 BasicCAN 模式）
在超时和出现错误信号后跳出循环 r                                          */
while((ModeControlReg & RM_RR_Bit)= =ClrByte)
{
  /*其他位而不是复位模式/请求位没有改变。                                 */
ModeControlReg = ModeControlReg｜RM_RR_Bit;
}
/*根据图 3 给定的硬件设定时钟分频寄存器
选择 PeliCAN 模式
旁路 CAN 输入比较器作为外部收发器使用
为控制器 S87C654 选择时钟                                                 */
ClockDivideReg=CANMode_Bit｜CBP_Bit｜DivBy2;
/*如果需要（在上电后总是必须的），CAN 中断禁能，
（写 SJA1000 中断使能/控制寄存器）                                        */
InterruptEnReg=ClrIntEnSJA;

/*定义验收代码和屏蔽                                                      */
AcceptCode0Reg=ClrByte;
AcceptCode1Reg=ClrByte;
AcceptCode2Reg=ClrByte;
AcceptCode3Reg=ClrByte;
AcceptMask0Reg=DontCare;          /*接收任何 ID                           */
AcceptMask1Reg=DontCare;          /*接收任何 ID                           */
AcceptMask2Reg=DontCare;          /*接收任何 ID                           */
AcceptMask3Reg=DontCare;          /*接收任何 ID                           */
```

```
/*配置总线定时                                                    */
/*位频率＝1Mbit/s@24MHz，总线被采样一次                           */
BusTiming0Reg=SJW_MB_24｜Prec_MB_24;
BusTiming1Reg=TSEG2_MB_24｜TSEG1_MB_24;

/*配置 CAN 输出：TX1 悬空，TX0 推挽
                正常输出模式                                      */
OutControlReg = Tx1Float｜Tx0PshPull｜NormalMode;

/*离开复位模式/请求，也就是转向操作模式，
S87C654 的中断使能
但 SJA1000 的 CAN 中断禁能，这可以在一个系统里面分别完成       */

/*清除复位模式位，选择双验收滤波器模式
关闭自我测试模式和仅听模式，
清除休眠模式（唤醒）                                            */
do                                  /*等待，直到 RM_RR_Bit 清零   */
/*在超时和出现错误后跳出循环                                      */
{
ModeControlReg = ClrByte;
}while((ModeControlReg&RM_RR_Bit) != ClrByte);

SJAIntEn = ENABLE;                  /*SJA1000 的外部中断使能      */
EA       = ENABLE;                  /*所有中断使能                */

/*------------------------------ SJA1000 初始化例子的结束  ----------------------------------------------------*/
```

4.2.2 发送

根据 CAN 协议规约[8]，信息的发送 CAN 控制器 SJA1000 独立完成。主控制器必须将要发送的信息传送到发送缓冲器，然后将命令寄存器里的“发送请求”标志置位。发送过程可由 SJA1000 的中断请求或控制段的轮询状态标志控制。

中断控制的发送

根据图 13 给出的控制器的主要过程，CAN 控制器的发送中断和由主控制器的外部中断中断优先级比开始发送使能（也由中断控制）高。中断使能标志是位于 BasicCAN 模式的控制寄存器和 PeliCAN 模式的中断使能寄存器（见表 2 及[1]）。

当 SJA1000 正在发送信息时，发送缓冲器被写锁定。所以在要放一条新信息到发送缓冲器中之前，主控制器必须检查状态寄存器（见[1]）的“发送缓冲器状态”标志（TBS）。

- 发送缓冲器被锁定：

  主控制器将新信息暂时存放在它自己的存储器里并设置一个标志，表示一条信息正在等待被发送。它会保存到软件设计者来处理这些临时存储信息。该存储器可设计为存储几条要被发送的信息。信息发送的开始会在中断服务程序中处理（程序在当前运行发送的末端被初始化。）

  从 CAN 控制器收到中断（见图 13 的中断处理过程）时，主控制器会检查中断类型。如果是发送中断，它会检查是否有更多的信息要被发送。一个正在等待的信息会从从临时存储器拷贝到发送缓冲器，表示要发送更多信息的标志被清除。置位命令寄存器（见[1]）的“发送请求”（TR）标志，SJA1000 将启动发送。

- 发送缓冲器被释放：

  主控制器将新信息写入发送缓冲器并置位命令寄存器（见[1]）的标志“发送请求”（TR），这将导致 SJA1000 启动发送。在发送成功结束时，CAN 控制器产生会一个发送中断。

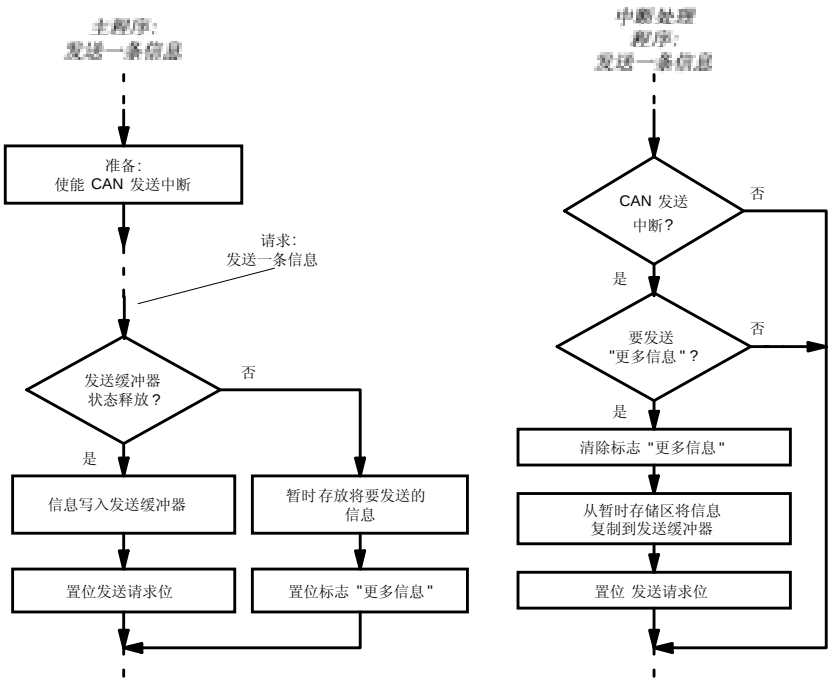

图 13 流程图“发送一条信息”（中断控制）

查询控制发送

这流程如图 14 所示，CAN 控制器的发送中断对这种发送控制无效。

只要 SJA1000 正在发送信息，发送缓冲器就被写锁定。因此在将新信息放进发送缓冲器之前，主控制器必须检查状态寄存器（见[1]）的“发送缓冲器状态”标志（TBS），。

- 发送缓冲器被锁定：

定期查询状态寄存器，主控制器等待，直到发送缓冲器被释放。

- 发送缓冲器被释放：

主控制器将新信息写入发送缓冲器并置位命令寄存器（见[1]）的“发送请求”（TR）标志，此时 SJA1000 将启动发送。

PeliCAN 模式的例子：

在附录里，给出了不同的常数和变量等等的定义。变量在 BasicCAN 和 Pelican 模式里的表示可以不同。例如“InterruptEnReg”在 BasicCAN 模式里是指控制寄存器但在 Pelican 模式里是指中断使能寄存器。语言“C”被用作编程。

根据 4.2.1 章给出的例子对 CAN 控制器初始化后，可启动正常的通讯：

```
/*等待，直到发送缓冲器被释放                                          */
Do
{
/*等待时，启动查询定时器并运行一些任务
在超时和出现错误后跳出循环                                          */
}while((statusReg & TBS_Bit) != TBS_Bit);

/*释放发送缓冲器，信息可写入缓冲器                                    */
/*在这个例子里，会发送一个标准帧信息                                  */
TxFrameInfo = 0x08;                    /*SFF(data)，DLC＝8          */
```

```
TxBuffer1    = 0xA5;                    /*ID1 ＝A5，（1010，0101）                */
TxBuffer2    = 0x20;                    /*ID2 ＝20，（0010，0000）                */
TxBuffer3    = 0x51;                    /*data1 =51                               */
•
•
TxBuffer10   = 0x58;                    /*data8 =58                               */

/*启动发送                                                                        */
   CommandReg   =TR_Bit;                /*置位发送请求位                          */
•
•
```

状态寄存器的 TS 和 RS 标志能用于检测 CAN 控制器是否已达到空闲状态。TBS－和 TCS－标志能用于检查是否成功发送。

BasicCAN 模式的例子：

在附录里，给出了不同的常数和变量等等的定义。变量在 BasicCAN 和 Pelican 模式里的表示可以不同。例如“InterruptEnReg”在 BasicCAN 模式里是指控制寄存器但在 Pelican 模式里是指中断使能寄存器。语言“C”被用作编程。

根据 4.2.1 章给出的例子对 CAN 控制器初始化后，可启动正常的通讯：

```
•
/*等待，直到发送缓冲器被释放                                                     */
Do
{
/*等待时，启动查询定时器并运行一些任务
在超时和出现错误后跳出循环                                                      */
}while((StatusReg & TBS_Bit) != TBS_Bit);

/*发送缓冲器被释放，信息可写入缓冲器                                              */
/*BasicCAN 模式里只有标准帧信息。                                                 */
TxBuffer1    = 0xA5;                    /* ID1＝A5，（1010，0101）                */
TxBuffer2    = 0x28;                    /* ID2＝28，（0010，0000）（DLC＝8）      */
TxBuffer3    = 0x51;                    /* data1=51                               */
•
TxBuffer10   = 0x58;                    /* data8=58                               */
/*启动发送                                                                        */
CommandReg = TR_Bit;                    /*置位发送请求位                          */
•
```

TBS－和 TCS－标志用于检查是否成功发送。

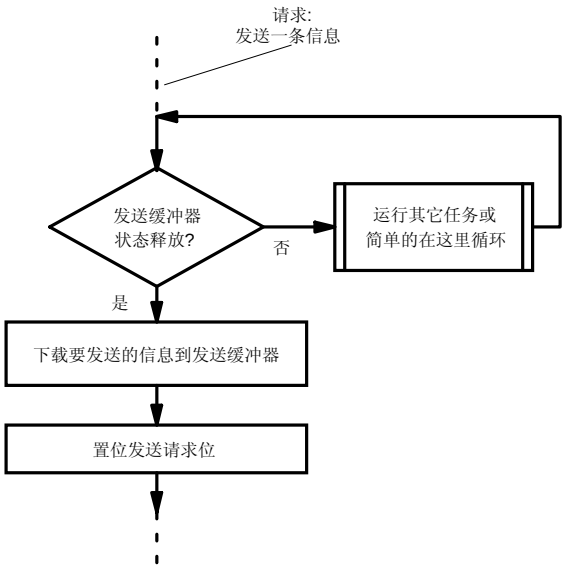

图 14　流程图“发送一条信息”（查询控制）

4.2.3　中止发送

一条请求中止发送的信息，可通过对命令寄存器[1]的相应位置位执行“中止发送”命令。这个特征可用于，例如，为了发送一个比现在信息紧急的信息，而这条信息已被提前写入发送缓冲器，但是直到现在没有被成功地发送。

图 15 显示了一个使用发送中断的流程。这个流程说明了为了发送更高优先级的信息而中止当前发送信息的情况。不同原因的中止信息要求不同的中断流程。

一个相应的流程能从查询控制发送中衍生得到。

以免信息由于不同的原因仍然等待处理，发送缓冲器会锁定（见图 15 的主流程图）。如果要求发送一个紧急信息，在命令寄存器里的中止发送位被置位。当这条等待处理信息已被成功的发送或中止，发送缓冲器被释放，同时产生一个发送中断被。在中断流程，要检查状态寄存器的发送完成标志，确定前面的发送是否成功。状态“未完成”表示发送被中止。在这种情况下，主控制器要运行一个特殊程序运行来中止发送，例如，在检查后重复发送中止的信息（如果它仍然有效的话）。

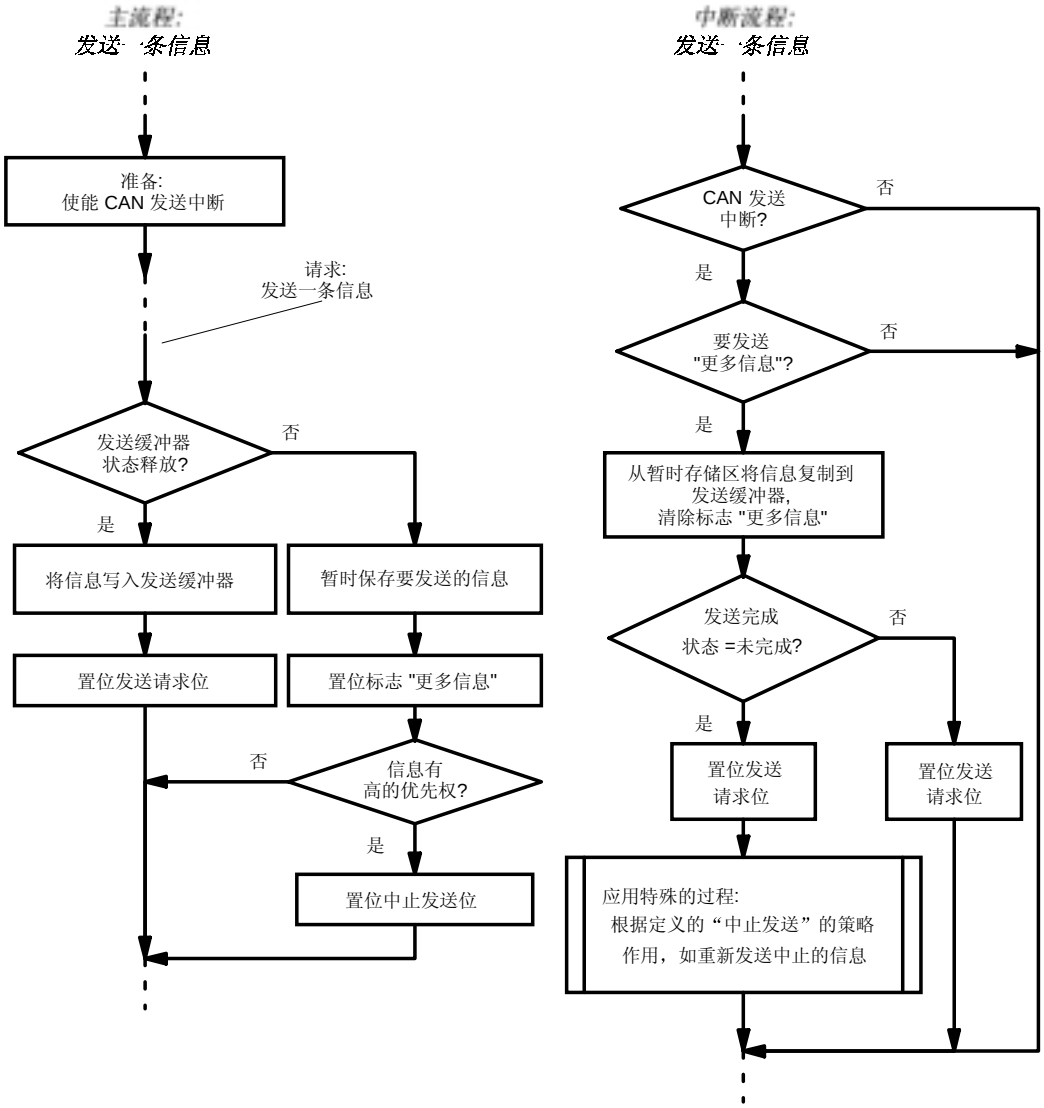

图 15　流程图“中止发送一条信息”（中断控制）

4．2．4 接收

根据 CAN 协议规约[8]信息的接收由 CAN 控制器 SJA1000 独立完成。收到的信息放到接收缓冲器(见 4.1.1 和 5.1)。可以发送给主控制器的信息，由状态寄存器的接收缓冲器状态标志“RBS”标出（见[1]）和接收中断标志“RI”（见[1]）标出（如果使能的话）。主控制器会将这条信息发送到本地的信息存储器，然后释放接收缓冲器并对信息操作。发送过程能被来自 SJA1000 的中断请求或查询 SJA1000 的控制段状态标志来控制。

查询控制接收

流程如图 16 所示，CAN 控制器在这种接收类型下接收中断禁能。

主控制器如常读 SJA1000 的状态寄存器，检查如果接收缓冲状态标志（RBS），看是否收到一个信息。这些标志的定义位于控制段的寄存器（见[1]）。

- 接收缓冲器状态标志＝“空”，也就是没有收到信息。

主控制器继续当前的任务直到收到检查接收缓冲器状态的新请求。

.

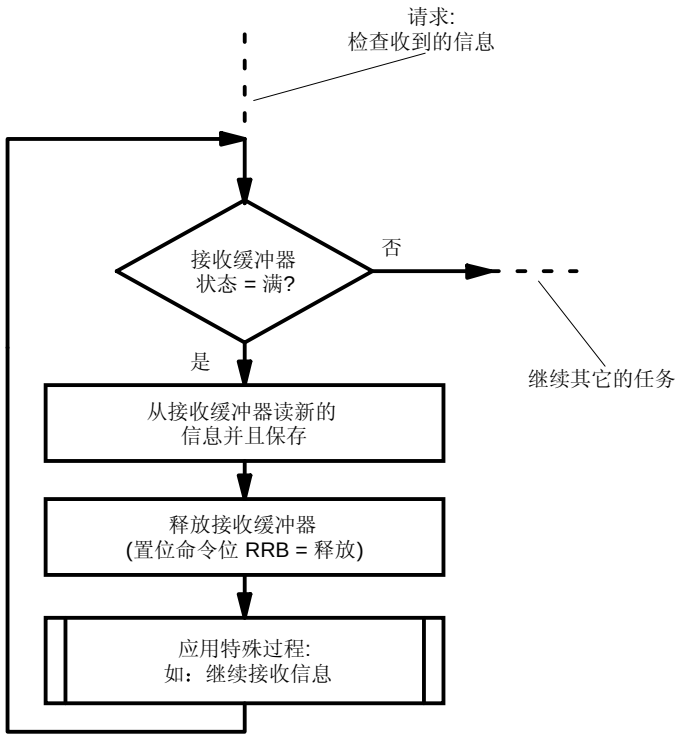

图 16　流程图“接收一条信息”（查询控制）

- 接收缓冲器状态标志＝“满”，也就是说收到一个或更多的信息：

主控制器从 SJA1000 得到第一条信息，然后通过置位命令寄存器的相应位，发送一个释放接收缓冲器命令。如图 16 所示，主控制器在检查更多信息前处理每个收到的信息。但也可以通过再次查询接收缓冲器状态位立即接收更多信息，而将所有收到的信息一起处理。在这种情况下，主控制器的信息存储器必须足够大以存储多于一条的信息。在已经发送和处理一条或所有的信息后，主控制器继续其他的任务。

中断控制接收

根据图 17 给出的控制器的主要过程，用于和 SJA1000 通迅的 CAN 控制器的接收中断和主控制器的外部中断使能而且优先级高于中断控制的信息接收。中断使能标志位于控制寄存器里（对于 BasicCAN 模式）或位于中断使能寄存器里（对于 PeliCAN 模式）－见表 2 和[1]。

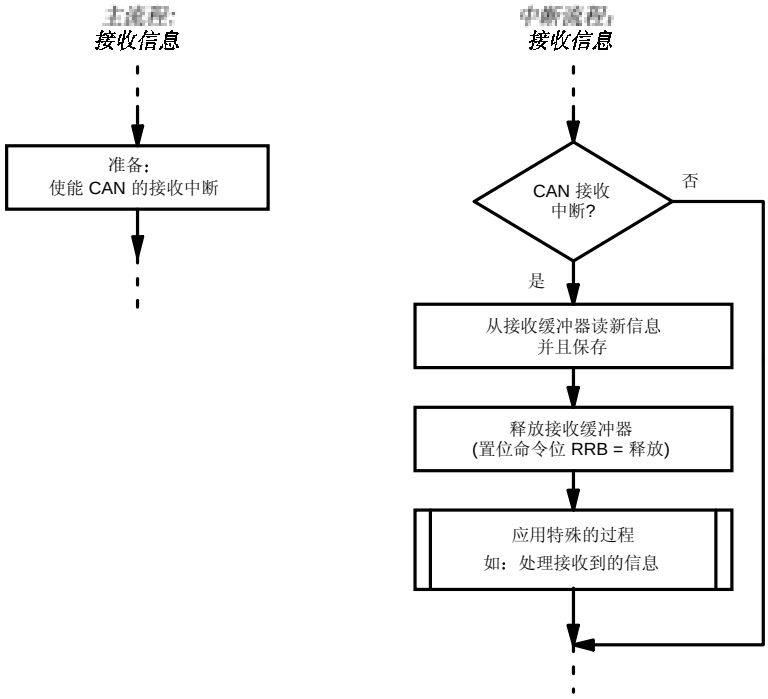

图 17　流程图“接收一条信息”（中断控制）

如果 SJA1000 已接收一条信息，这条信息已通过验收滤波器且已放在接收 FIFO 中，那么会产生一个接收中断。因此主控制器能立刻起作用，发送接收到的信息到信息存储器，然后通过对命令寄存器的相应标志“RRB”（见[1]）置位发送一个释放接收缓冲器命令。在接收 FIFO 里的更多信息将产生一个新的接收中断，因此在一个中断期间不可能将所有在接收 FIFO 中的有效信息读出。和这个方法相反，图 18 显示了一个将所有信息一次读出的过程。在释放了接收缓冲器后，SJA1000 会检查状态寄存器中接收缓冲状态（RBS）看是否有更多信息，而所有有效的信息都会被循环读出。

如图 17 所示，整个接收过程在中断程序期间完成，而且和主程序没有相互作用。如果可行的话，信息的处理甚至也可以在中断程序里完成。

例子：

在附录里，给出了不同的常数和变量等等的定义。变量在 BasicCAN 和 Pelican 模式里的表示可以不同。例如“InterruptEnReg”在 BasicCAN 模式里是指控制寄存器但在 Pelican 模式里是指中断使能寄存器。语言“C”被用作编程。

根据 4.2.1 章给出的例子对 CAN 控制器初始化后，可启动正常的通讯：

1. 部分主程序

```
•
/*接收中断使能                                                    */
InterruptEnReg＝RIE_Bit;
•
```

2. 部分中断 0 服务程序

```
•
•
/*从 SJA1000 读中断寄存器的内容并临时保存，所有中断标志被清除（在 PeliCAN 模式里，
接收中断（RI）被首先清除，当给出释放缓冲器命令时）
*/
```

```
CANInterrupt = InterruptReg;
 •
 •
/*检查接收中断和读一个或所有接收到的信息                                    */
 If     (RI_VarBit  ＝＝YES)                    /*检测到接收中断            */
{
       /*从 SJA1000 得到接收缓冲器的内容，并将它存入控制器的内部存储器，
       可以立刻对帧信息和数据长度代码解码并适当地取出。                      */
        •
        •
       /*释放接收缓冲器，接收中断标志被清除，新的信息将产生一个新中断        */
       CommandReg＝RRB_Bit;                    /*释放接收缓冲器            */
}
 •
 •
```

数据溢出处理

万一接收 FIFO 满了，但还在接收其他信息，就会通过置位状态寄存器中的数据溢出状态位（如果使能）通知主控制器有数据溢出，SJA1000 会产生一个数据溢出中断。

如果运行在数据溢出的状态下，由于主控制器没有足够的时间及时从接收缓冲器取收到的信息而变得极度超载。一个表示数据丢失的数据溢出信号，可能会导致系统矛盾。通常一个系统应该设计成：为了避免数据溢出，收到的信息要被足够快地传输和处理。如果数据溢出不能避免，那么主控制器应该执行一个特殊的处理程序来处理这些情况。

图 18 是有关的程序流程，立即处理数据溢出中断。

在已经传输这条信息后（该信息产生接收中断并释放接收缓冲器），会通过读接收缓冲器状态来检查是否在接收 FIFO 中还有有效信息。因此在继续下一步之前，所有的信息都能从接收 FIFO 取出。当然在中断期间读一条信息并且处理完它（可能的话），要比 SJA1000 接收一条新信息更快点。否则主控制器将一直在中断里读信息。

检测到数据溢出后，可以根据“数据溢出”原则启动一个异常处理。这个原则在两种情况下决定：

－数据溢出和接收中断一起发生：

信息可能已经丢失。

－数据溢出发生时，没有检测到接收中断：

信息可能已经丢失，接收中断可能禁能。

这由系统设计者确定，主控制器怎样对这些情况采取相应的动作。

信息的查询控制接收也可以使用相同的处理方法。

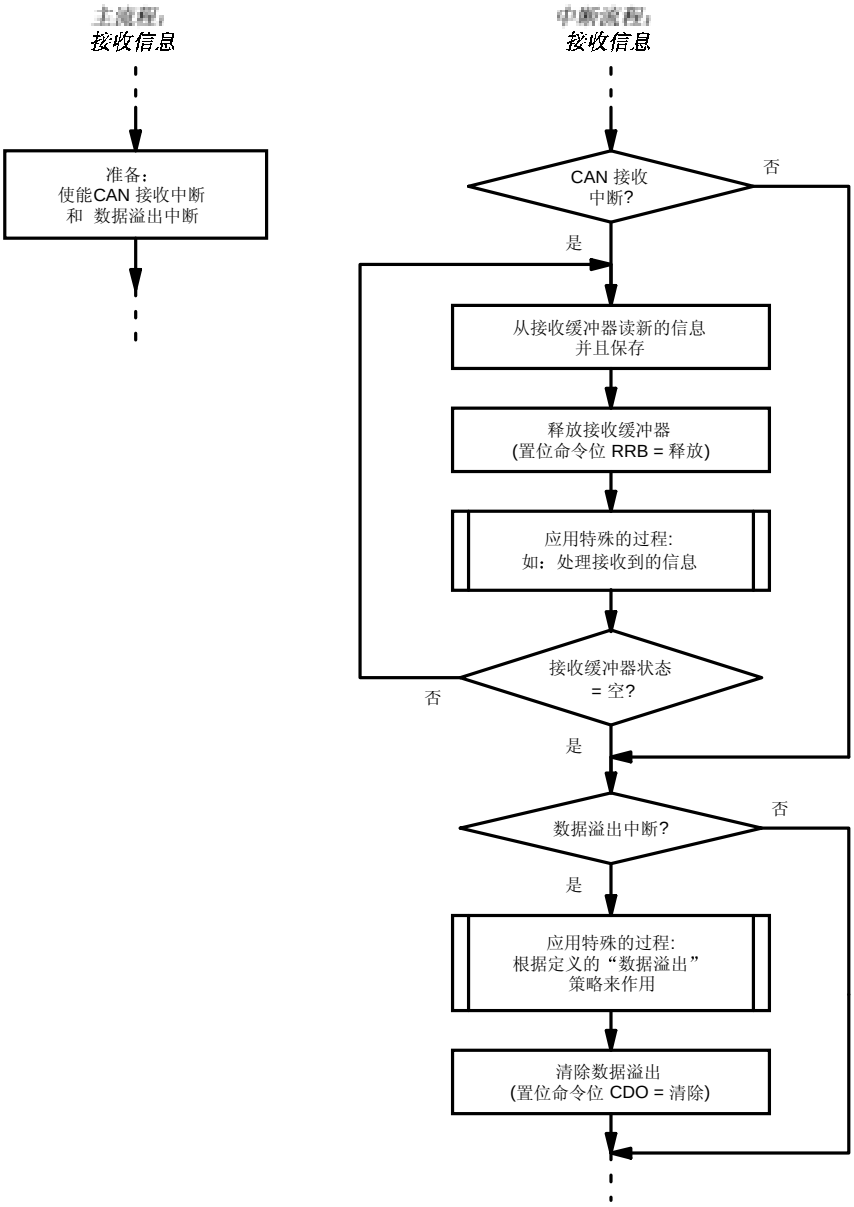

图 18　流程图“信息的接收和数据溢出”（中断控制）

4.2.5　中断

在 PeliCAN 模式里，SJA1000 有 8 个不同的中断源（在 BasicCAN 模式里仅有 5 个），这些中断可使主控制器立即动作，作用在 CAN 控制器的某些状态上。

一旦 CAN 产生中断，SJA1000 会将中断输出（管脚 16）设为低电平，直到主控制器通过读 SJA1000 的中断寄存器对中断采取相应措施；或假如在 PeliCAN 模式里，释放接收缓冲器前有接收中断。在主控制器这个动作后，SJA1000 将输出中断跳到高电平。如果这段时间有更多中断，或接收 FIFO 里有更多有效信息，SJA1000 立刻将中断输出再一次设为低电平。因此输出仅在很短的时间里保持高电平。处理中断请求的握手信号和在两个中断请求期间的高电平脉冲，主控制器的中断是由电平触发。

图 19 的流程给出所有可能中断的概述和详细的描述。在这个流程里不同的中断被处理的次序仅是种可

能解决的方法。中断被处理的次序极大地取决于系统和它所要求的行为。这必须由整个系统的设计者决定。

发送、接收和数据运行溢出中断的作用，在前几章里已作讨论。

图 20、21、22 详细地给出唤醒中断、仲裁丢失中断、和三个不同的错误中断的流程。所有的错误中断能执行系统的一个通用错误策略程序。这个策略用该完成：在开发期的系统优化和在操作期的自动系统优化和系统维护。仲裁丢失中断可被用于系统优化和维护。见下面的章节和数据表[1]可获得关于不同的错误信号、仲裁丢失处理和相关的信息的详细资料。

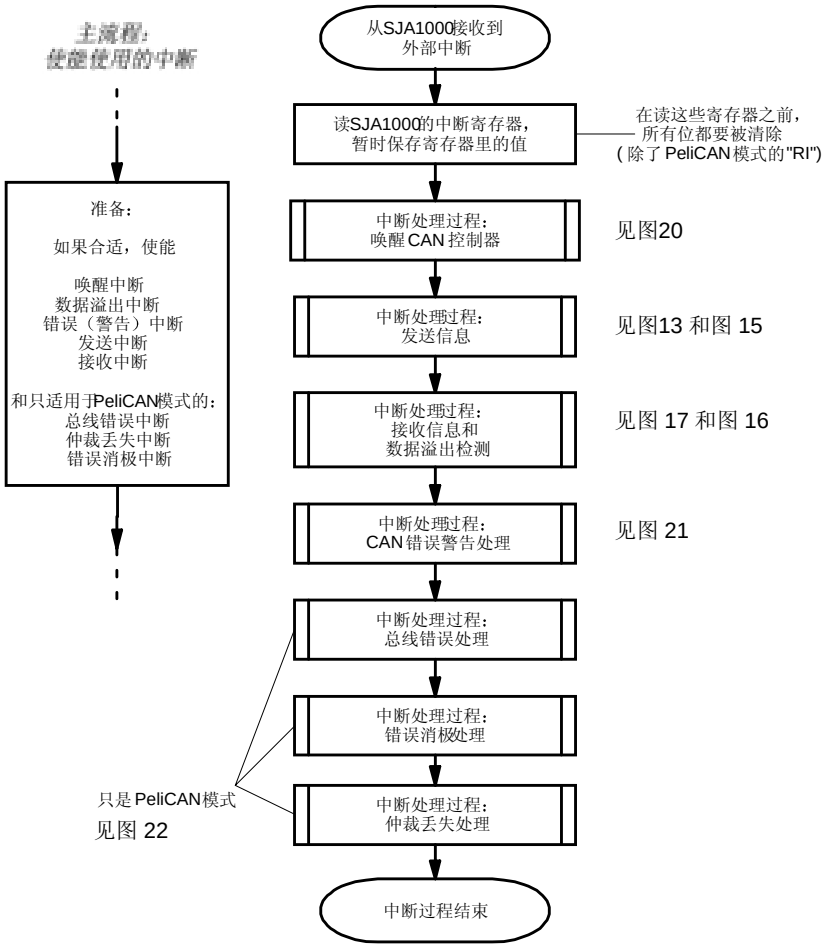

图 19　总体中断流程

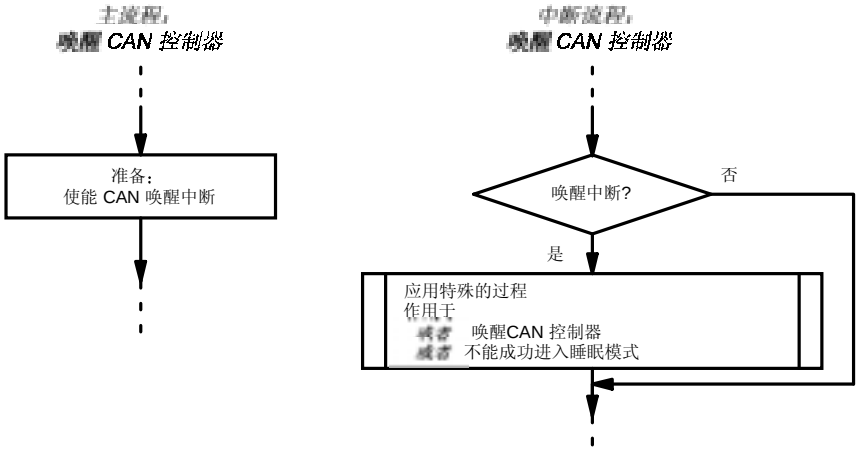

图 20　流程图“唤醒 CAN 控制器”

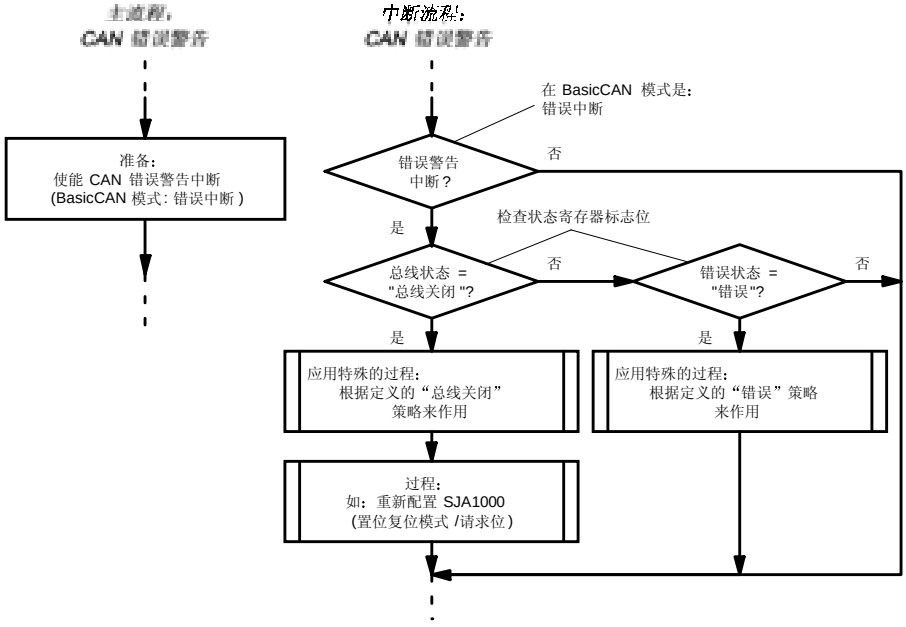

图 21　流程图“错误警告中断”

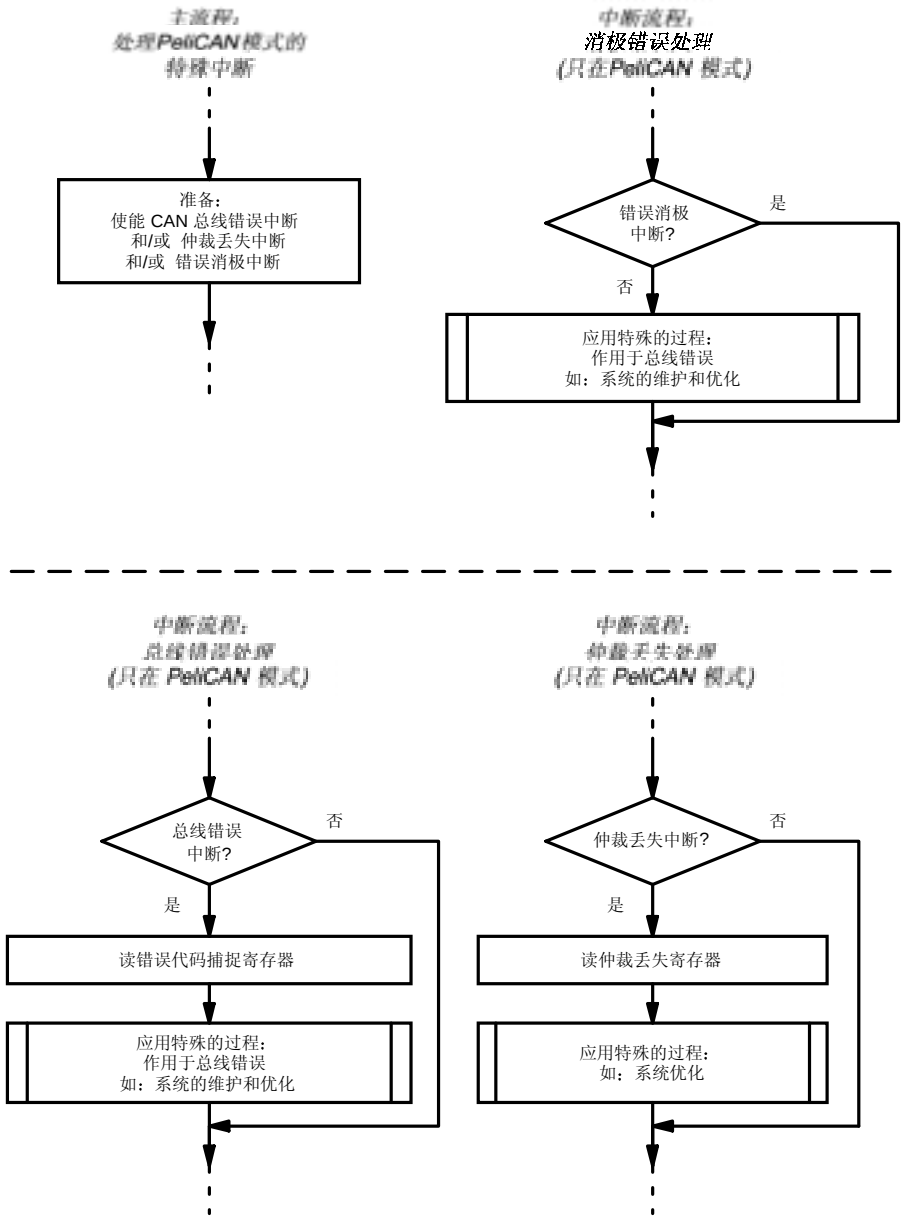

图 22　流程图“特殊 PeliCAN 中断的处理过程”

5.　PELICAN 模式功能

5.1　接收 FIFO/信息计数器/直接 RAM 访问

SJA1000 寄存器和信息缓冲器对于主控制器来说是外围寄存器，且能通过复用的地址/数据总线寻址。在不同的选择模式（操作或复位）可访问不同的寄存器是。可正常操作的地址范围是：地址 0…31。它包括用于初始化、状态和控制的寄存器。而且 CAN 信息存储器分配于地址 16 和 28 之间。在主控制器写访问时，用户能够对 CAN 控制器寻址，在读访问时，读出接收缓冲器的内容。

除了上面所说的范围外，整个接收 FIFO 映射在 CAN 地址 32 和 95 之间（见图 23）。而且，SJA1000 的发送缓冲器在 CAN 地址 96 和 108 之间可达到。

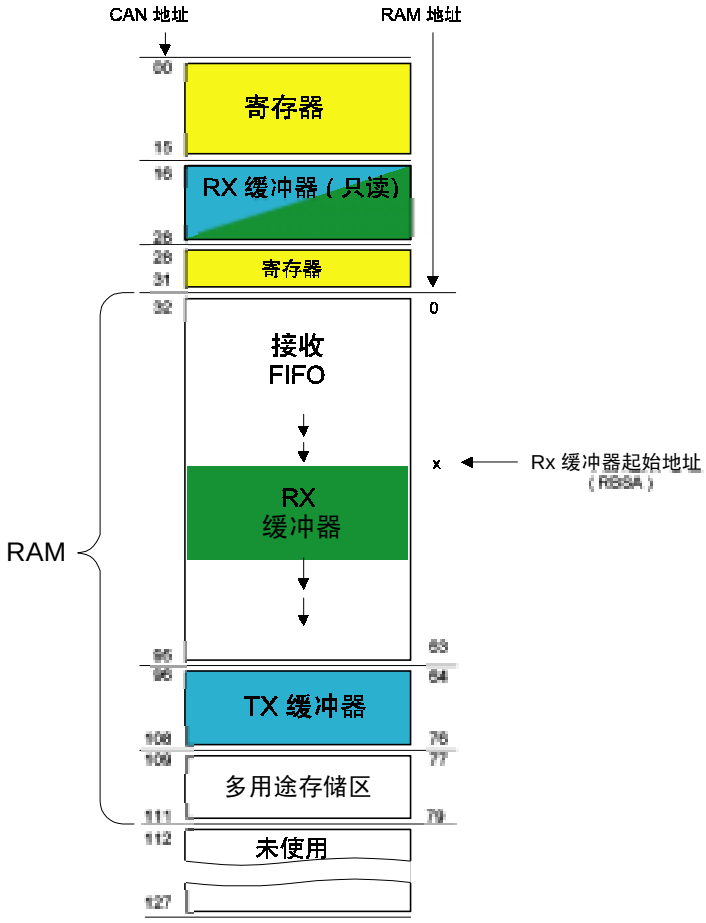

图 23　寄存器和 RAM 地址分配

在直接 RAM 访问中，可以读发送缓冲器和完整的接收 FIFO。

在 PeliCAN 模式里，接收 FIFO 能够存储高达 n=21 条信息。使用下面的等式，可以计算信息的最大数目：

$$n=\frac{64}{3+data\_length\_code}$$

接收缓冲器被定义为一个 13 字节窗口，总是包括接收 FIFO 的当前接收信息。如图 24 所示，下面的部份或全部的信息在接收缓冲器窗口有效。

但是，在命令“释放接收缓冲器”之前，接收 FIFO 里的下一个收到的信息在接收缓冲器窗口（从 CAN 地址 16 开始）将完全可视。

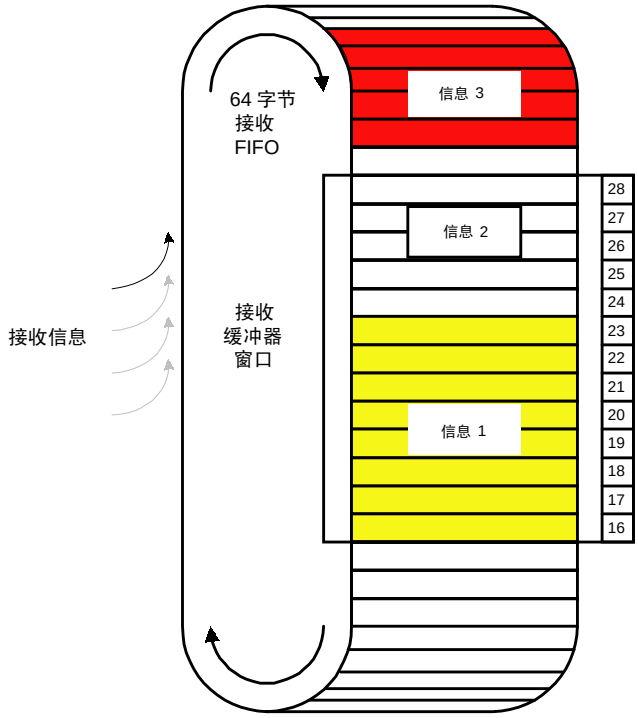

图 24　接收 FIFO

为了分析的需要，SJA1000 提供两个另外的寄存器，支持接收信息处理：

- Rx 缓冲器开始地址寄存器（RBSA）允许接收 FIFO 范围里的单个 CAN 信息的识别。
- Rx 信息计数器寄存器，包括接收 FIFO 里的当前存储的信息数目。

图 23 显示了物理 RAM 地址和 CAN 地址之间的关系。

5.2　错误分析功能

每个 CAN 控制器能够在三个错误状态之一中工作：错误激活，错误消极，或总线关闭，取决于错误计数器的值。如果两个错误计数器的值都在 0—127 之间，CAN 控制器是错误激活的。一旦有错误就会产生错误激活标志（6 个控制位）。如果错误计数器中某个位于 128—255 之间，SJA1000 是错误消极的。消极错误标志（6 个隐性位）会在检测到有错误之后产生。如果发送错误计数器高于 255，则到达总线关闭状态。在这种状态下，复位请求位自动置位，SJA1000 对总线没有影响。如图 25 所示，总线关闭只能在主控制器使用命令“复位请求＝0”退出。这将启动总线关闭恢复时间，发送错误计数器用于对 128 个总线空闲信号计数。恢复时间结束后，两个错误计数器都是 0，设备处于错误激活状态。

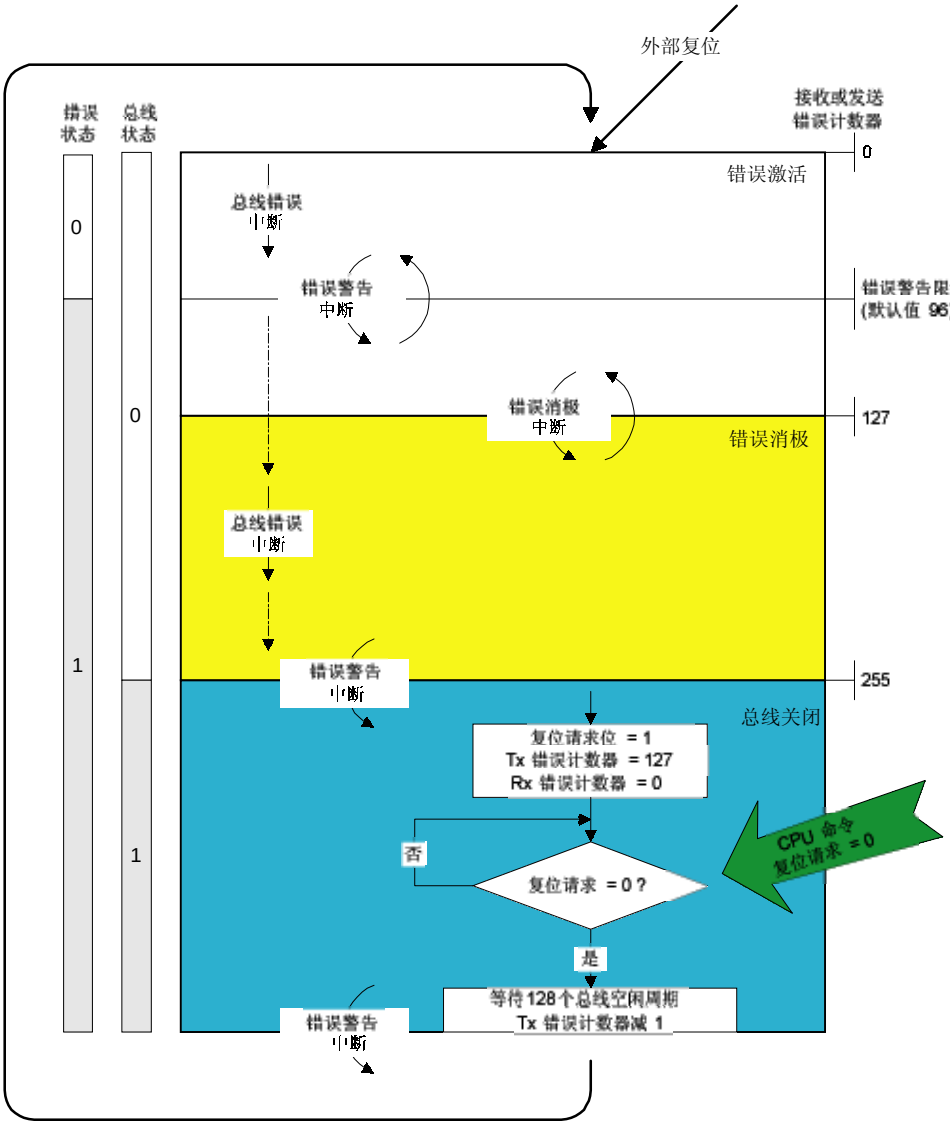

图 25 SJA1000 错误中断

而且这数字显示两个错误和总线状态在不同的错误状态下的值。

5.2.1 错误计数器

如上面描述，CAN 的错误状态直接和发送器和接收错误计数器的值有关。

为了仔细研究错误限制，和支持一个 SJA1000 的增强的错误分析，CAN 控制器提供可读的错误计数器。另外，在复位模式，允许对于两个错误计数器进行写访问。

5.2.2 错误中断

见图 25，使用了 3 个错误中断源来区分主控制器的错误状态。每个中断都能在中断使能寄存器里单独使能。

总线错误中断:

在 CAN 总线上任何一个错误都会产生中断。

错误警告中断：

如果超过错误警告限制，产生错误警告中断被。而且它被产生如果 CAN 控制器进入总线关闭状态和在此之前再一次进入错误激活状态。SJA1000 的错误警告限制在复位模式是可编程的。复位后的默认值是

96。

错误消极中断：

如果错误状态从错误激活变到错误消极或相反，产生错误消极中断。

5.2.3 错误代码捕捉

如前几部分描述，SJA1000 完成在 CAN2.0B 规约[8]指定的所有的错误限制。在每个 CAN 控制器，处理错误的整个过程是完全自动的。但是，为了提供用户某个错误的详细信息，SJA1000 包括错误代码捕捉功能。无论什么时候发生 CAN 总线错误，会强制产生相应的总线错误中断。同时，当前位的位置被捕捉进错误代码捕捉寄存器。捕捉的数据会被保存直到主控制器将它读出。之后捕捉机构再次激活。寄存器可以内容区分四种错误类型：格式，一填充，一位和其他错误。如图 26 所示，寄存器还另外表明在信息的接收或发送期间是否发生错误。在这个寄存器的五个位表明 CAN 帧内错误的位位置，见下面的表和数据表获得更多的信息。

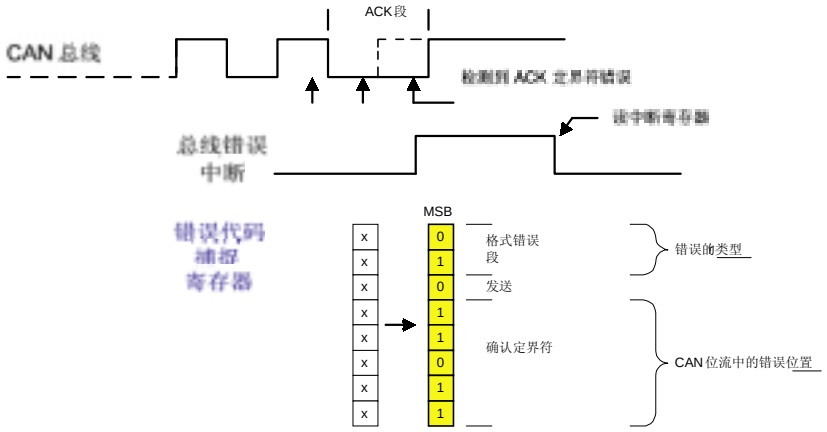

图 26 错误捕捉功能的举例

如 CAN 规格定义，CAN 总线上的每个位有特殊的错误类型。下面两张表显示了在 CAN 信息发送和接收期间所有可能的错误。左边这部份包括位置和错误的类型，这些由错误代码捕捉寄存器捕捉。每张表的右边部分是翻译成上层错误描述而且可以直接从寄存器内容中得到。通过使用这些表格，能得到有关错误计数器改变和在器件发送和接收管脚的错误状态的更多信息。使用这些表时，例如在错误分析软件里，可以详细地分析每一个错误状态。关于 CAN 错误类型和位置的信息能用于错误统计和系统维护或在系统优化期间进行纠正。

表 6 接收期间可能出现的错误

错误代码捕捉	
在 CAN 位流里的错误的位置	错 误类型
ID SRR，IDE 和 RTR 位 保留位 数据长度代码 数据段 CRC 序列	填充
CRC 定界符	格式 填充
确认位	位
确认定界符 [1]	格式

RX 错误计数	描述	
+1	相同级别下收到超过 5 个连续的位	－－
+1 +1	Rx＝控制 相同级别下收到超过 5 个连续的位	位必须是隐性的
+1	TX＝控制，但 RX＝隐性	不能写控制位
+1	RX＝控制 或 检测到 CRC 错误 [1]	紧急的总线定时或总线长度 CRC 序列不正确

帧结束	格式 其他	+1 ±0	RX＝在头六位里控制 RX＝在最后一位里控制	—— 作用：发出过载标志，如果发送器开始重新发送，数据可能重复。
中止	其他	±0	RX＝控制	作用：接收器发出过载标志
激活错误标志	位	+8	TX＝控制，但 RX＝隐性	不能写控制位
容许控制位	其他	+8	RX＝在错误标志的第一位控制 RX＝控制超过 7 位的错误或过载标志	
错误定界符	格式 其他	+1 ±0	RX＝在头七位里控制 RX＝在定界符的最后一位控制	—— 过载标志将被发送
超载标志	位	+8	TX＝控制，但 RX＝隐性	不能写控制位

[1] 如果 CRC 不 o.k，则错误会在确认定界符内处理，并产生“格式错误”。

表 7　在发送期间的可能的错误

错误代码捕捉				
在 CAN 位流里的一个错误位置	错误类型	TX 错误计数	描述	
帧开始	位	+8	Tx＝控制，但 Rx＝隐性	不能写控制位
ID	位 填充	+8 ±0	Tx＝控制，但 Rx＝隐性 Tx＝隐性，但 Rx＝控制的	不能写控制位 ——
SRR 位	位 填充	+8 ±0	Tx＝控制，但 Rx＝隐性 Tx＝隐性，但 Rx＝控制	不能写控制位 ——
IDE 和 RTR 位	位 填充	+8 +8	Tx＝控制，但 Rx＝隐性 Tx＝隐性，但 Rx＝控制	不能写控制位 ——
保留位 数据长度代码 数据段 CRC 序列	位	+8	Tx＝控制，但 Rx＝隐性	不能写控制位
CRC 定界符	格式	+8	Rx＝控制	位必须为隐性
确认位	其他 其他	+8 ±0	Rx＝隐性（错误激活） Rx＝隐性（错误消极）	没有确认 没有确认，节点可能单独在总线上
确认定界符	格式	+8	Rx＝控制的	紧急总线定时或总线长度
帧结束	格式 其他	+8 +8	Rx＝控制头六个位 Rx＝控制最后一位	—— 帧已经被一些节点接收，再次发送可能导致接收器里数据重复。
中止	其他	±0	Rx＝控制的	来自于“旧”CAN 控制器的过载标志
激活的错误标志 过载标志	位	+8	Tx＝控制的，但 Rx＝隐性	不能写控制位
允许控制位	格式	+8	Rx＝控制的，在激活错误标志或过载标志后超过 7 个位时间	——
错误定界符	格式 其他	+8 ±0	Rx＝头七位控制 Rx＝控制定界符的最后一位	—— 来自于“旧”CAN 控制器的过载标志
消极错误标志	其他	+8	Rx＝控制的（错误消极）	没有确认收到，节点不单独在总线上。

5.3　仲裁丢失捕捉

SJA1000 能够确定仲裁丢失的确切的 CAN 位流位置。这个“仲裁丢失中断”立即产生。而且，这个

位号被捕捉进仲裁丢失捕捉寄存器里。主控制器读出这个寄存器的内容后，仲裁丢失捕捉功能被再次激活。

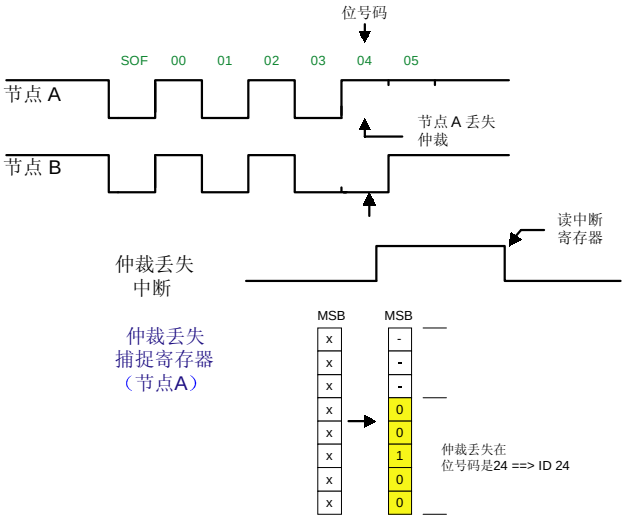

图 27　仲裁丢失捕捉功能

在这个功能的帮助下，SJA1000 能够监控每个 CAN 总线的访问。诊断或在系统配置期间，可以识别仲裁失败的所有位置。

下面这个例子显示了钟裁丢失功能怎样使用。

首先使能中断使能寄存器里的仲裁丢失中断。中断后，中断寄存器的内容就被保存起来。如果仲裁丢失中断标志被置位，仲裁丢失捕捉寄存器的内容就会被分析。

例子：**仲裁丢失**

```
…
InterruptEnReg＝ALIE_Bit;                    /*仲裁丢失中断使能                    */
…
/*---------------------------------------中断服务程序-------------------------------------------------------------- */
…
int_reg_copy＝InterruptReg;                  /*保存中断寄存器内容                  */
…
if     (int_reg_copy & ALIE_Bit)
       candat = ArbLostCapReg;               /*读仲裁丢失捕捉寄存器                */
…
…
```

5.4　单次发送

在一些应用中，CAN 信息的自动重发送没有意义：它造成一个节点几次仲裁和数据变得无效。

为了提前请求一个“单次发送”，CAN 控制器必须完成下面的步骤：

1.　发送请求

2.　等待发送状态

3.　中止发送

处理的这个软件能被最小化到单个命令，带有“单个 shot 发送”选项，通过对命令位 CMR.0 和 CMR.1 同时置位来初始化，。

在这种情况下，没有必要查询状态位，主控制器能集中于其他任务上。“单次发送”功能能与 SJA1000 的仲裁丢失和错误代码捕捉功能完美结合。

如果仲裁丢失或发生错误， SJA1000 不会重新发送信息。在状态寄存器里，发送状态位一被置位，内部发送请求位就自动清除。

来自于两个捕捉寄存器的额外信息，一条信息是否被重新发送是由用户控制。

如在 5.7 章里描述的，单次发送能和自我测试模式一起使用。

5.5 仅听模式

在仅听模式里，SJA1000 在 CAN 总线上不能写控制位。激活错误标志或过载标志不能被写，成功接受之后的确认信息也不会给出。

错误的处理就像在错误被动模式里。错误分析功能如，错误代码捕捉和错误中断就像在正常操作模式一样可以工作。

但是，错误计数器的状态被冻结。

可以接收信息，但不可以发送。因此，这个模式可用于自动位频率检测，见 5.6 章和其他带有监控功能的应用。

注意，在进入仅听模式之前，必须进入复位模式。

例子：**仅听模式**

```
…
ModeControlReg＝RM_RR_Bit;                /*进入复位模式                */
ClockDivideReg ＝CANMode_ Bit;            /*PeliCAN 模式                */
…
ModeControlReg＝LOM_Bit;                  /*进入仅听模式                */
                                          /*离开复位模式                */
…
```

5.6 自动位率检测

对于自动位频率来说，现存的主要缺陷是 CAN 错误帧的产生不被接受的。SJA1000 支持 PeliCAN 模式的自动位频率检测的请求。这部份简短地描述了一个不影响网络上的运行操作应用例子。

在仅听模式，SJA1000 不能发送信息也不能产生错误帧。这个模式里只能接收信息。在软件里预先定义的表格包含所有可能的位频率以及它们的位时序参数。在启动用最高位率信息接收之前，SJA1000 接收和错误中断都使能。

如果在 CAN 总线上产生了错误，软件会转向下一个较低的位频率。

在一条信息的成功地接收后，SJA1000 已经检测到正确的位率而且能转向正常工作模式。从现在起，这个节点上能够象系统其他激活的 CAN 节点一样工作。

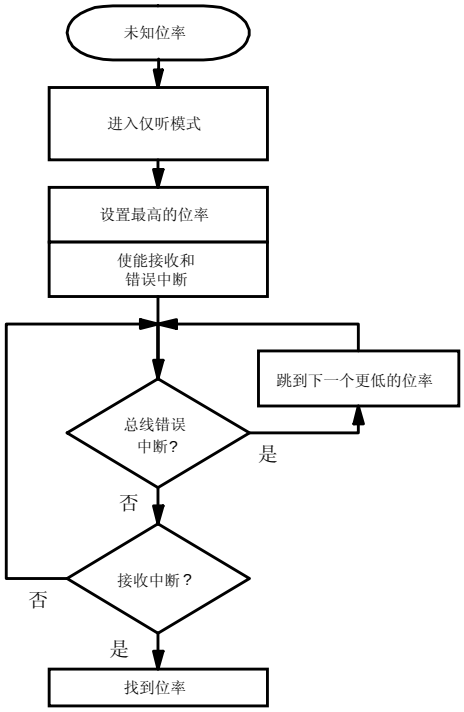

图 28　检测位率的算法

5.7　CAN 自我测试

SJA1000 支持两种不同属性的自我测试：

- 局部自我测试
- 总体自我测试

局部自我测试，例如能被完美地用于单个节点测试，因为它不需要来自于其他节点的确认。如果这样的话，SJA1000 必须处于“自我测试模式”（模式寄存器），命令“自我接收请求”被给出。

对于总体自我测试，在操作模式下 SJA1000 执行同样的命令。但是，在一个正在运行的系统中，需要一个 CAN 的确认。

注意在这两种情况下，一个物理层接口必须是有效，包括一个带有终端的 CAN 总线。发送或自我接收通过对命令寄存器的相应位的置位进行初始化，。

SJA1000 提供三个命令位用于 CAN 发送和自我接收的初始化。表 8 显示了所有可能的连接（取决于所选的工作模式）。

表 8　CAN 发送请求命令

命令	CMR=	在成功操作上的中断	自我测试模式	操作模式
自我接收请求	0x10	RX 和 TX	局部自我测试	总体自我测试
发送请求	0x01	TX	正常发送 [1]	正常发送
单次发送	0x03	TX	没有重新发送的发送 [1]	没有重新发送的发送
单次发送和自我接收请求	0x12	RX 和 TX	没有重新发送的局部自我测试	没有重新发送的总体自我测试

下面的例子表示了用于局部自我测试初始化的基本编程元素。

例：局部自我测试

```
…
        ModeControlReg   =RM_RR_Bit;          /*进入复位模式          */
         ClockDivideReg  =CANMode_Bit;        /*PeliCAN 模式          */
        ModeControlReg   =STM_Bit;           /*进入自我测试模式      */
                                             /*离开复位模式          */
             TxFramInfo  =0x03;              /*填满发送缓冲器        */
              TxBuffer1  =0x53;              /*                      */
…
              TxBuffer5  =0xAA;              /*最后一个发送的字节    */

            CommandReg   =SRR_Bit;           /*自我接收请求          */
……
if   (RxBuffer1 != TxBufferRd1)   comparison    =false;
if   (RxBuffer2 != TxBufferRd2)   comparison    =false;
```

[1] 带有或不带重新发送的正常发送通常在操作模式里完成。

5.8 接收同步脉冲的产生

在一条信息成功的接收时，SJA1000 允许 TX1 管脚上产生一个脉冲（如果信息被完全存储在接收 FIFO 里）。这个脉冲在时钟分频驱动寄存器里使能和在第 6 位的“帧结束”持续期间激活。

因此，它能被用于通用的事件触发任务，例如，作为一个专门触发中断源或用于一个分布式系统内的总时钟同步。

在分布式系统内，使一个不带额外同步信号的系统时钟起作用很困难。所有连接到总线上的节点都有局部时钟（带有时钟相移）。假定 CAN 网络内的一个节点作为“主”时钟操作，网络里其余的时钟被同步到主时钟的值。

自身接收请求特征和每个 SJA1000 在接收到信息后的一定时间内产生一个脉冲，能在分布式系统里支持系统时钟同步。

在图 29，一个系统主机发送“自身接收信息”到 CAN 总线上。信息接收后，每个节点，包括主机，都产生一个接收同步脉冲。在使用的每个从节点，使定时器复位。同时主节点使用这个脉冲去捕捉主时钟值 t_M。

在下一步，t_M 值被主机作为一个“参考定时信息”发送到所有从机。在每个从机的简单的加法程序（包括重载所有定时器）同步了整个系统时钟。

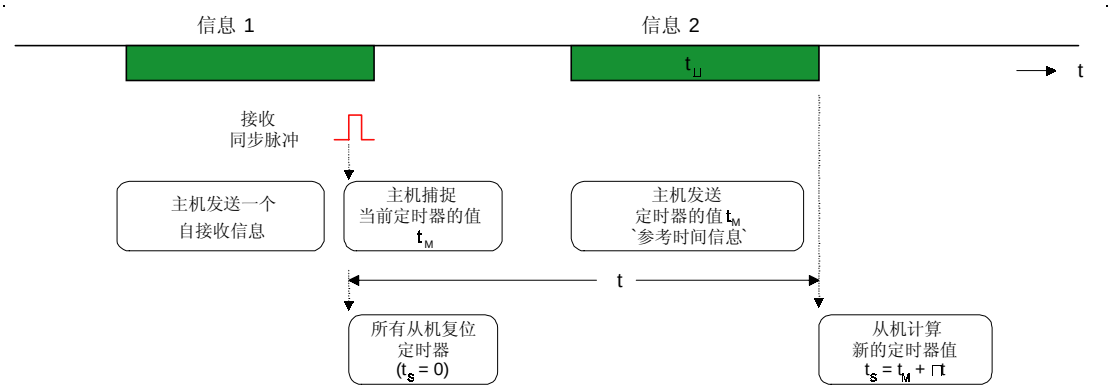

图 29 系统同步的时序图

这种思想的主要好处是简化复杂的时间标志处理。由于关键路径是由硬件控制确定，所以没有必要使用软件循环计数。而且它独立于网络参数。中断事件可能会在整个时期发生，但不影响同步过程。

6．附录

在应用指南里的例子，C 语言（Keil C）用于描述 SJA1000 的编程流程。在这些例子里，目标控制器是 S87C654，其他 80C51 系列的器件也可以使用。务必确保在主程里包括对目标器件寄存器正确的说明。

SJA1000 的寄存器和位定义

```
/*定义直接对 8051 的存储区访问                                                    */

#define   XBYTE（（unsigned char volatile xdata  *）0）

/*模式和控制寄存器的地址和位定义                                                  */

#define   ModeControlReg     XBYTE[10]

#define   RM_RR_Bit          0x01          /*复位模式（请求）位                  */
#if defined   (PeliCANMode)
#define     LOM_Bit          0x02          /*仅听模式位                          */
#define     STM_Bit          0x04          /*自我测试模式位                      */
#define     AFM_Bit          0x08          /*验收滤波器模式位                    */
#define     SM_Bit           0x10          /*进入休眠模式位                      */
#endif

/*中断使能和控制寄存器的地址和位定义                                              */
#if defined   (PeliCANMode)
#define     InterruptEnReg   XBYTE[4]      /* PeliCAN 模式                       */
#define     RIE_Bit          0x01          /*接收中断使能位                      */
#define     TIE_Bit          0x02          /*发送中断使能位                      */
#define     EIE_Bit          0x04          /*错误警告中断使能位                  */
#define     DOIE_Bit         0x08          /*数据溢出中断使能位                  */
#define     WUIE_Bit         0x10          /*唤醒中断使能位                      */
#define     EPIE_Bit         0x20          /*错误消极中断使能位                  */
#define     ALIE_Bit         0x40          /*仲裁丢失中断使能位                  */
#define     BEIE_Bit         0x80          /*总线错误中断使能位
#else       /*BasicCAN 模式    */
#define     InterruptEnReg   XBYTE[0]      /*  控制寄存器                        */

#define     RIE_Bit          0x02          /*接收中断使能位                      */
#define     TIE_Bit          0x04          /*发送中断使能位                      */
#define     EIE_Bit          0x08          /*错误中断使能位                      */
#define     DOIE_Bit         0x10          /*溢出中断使能位                      */
#endif


/*命令寄存器的地址和位定义                                                        */

#define     CommandReg       XBYTE[1]

#define     TR_Bit           0x01          /*发送请求位                          */
#define     AT_Bit           0x02          /*中止发送位                          */
#define     RRB_Bit          0x04          /*释放接收缓冲器位                    */
#define     CDO_Bit          0x08          /*清除数据溢出位                      */
#if defined   (PeliCANMode)
#define     SRR_Bit          0x10          /*自身接收请求位                      */
#else       /*BasicCAN 模式    */
#define GTS_Bit              0x10          /*进入睡眠模式位                      */
#endif

/*状态寄存器的地址和位定义
#define     StatusReg        XBYTE[2]                                           */

#define     RBS_Bit          0x01        /*接收缓冲器状态位                    */
```

```
#define     DOS_Bit          0x02      /*数据溢出状态位                    */
#define     TBS_Bit          0x04      /*发送缓冲器状态位                  */
#define     TCS_Bit          0x08      /*发送完成状态位                    */
#define     RS_Bit           0x10      /*接收状态位                        */
#define     TS_Bit           0x20      /*发送状态位                        */
#define     ES_Bit           0x40      /*错误状态位                        */
#define     BS_Bit           0x80      /*总线状态位                        */

/*中断寄存器的地址和位定义                                                 */
#define     InterruptReg     XBYTE[3]

#define     RI_Bit           0x01      /*接收中断位                        */
#define     TI_Bit           0x02      /*发送中断位                        */
#define     EI_Bit           0x04      /*错误警告中断位                    */
#define     DOI_Bit          0x08      /*数据溢出中断位                    */
#define     WUI_Bit          0x10      /*唤醒中断位                        */
#if defined   (PeliCANMode)
#define     EPI_Bit          0x20      /*错误消极中断位                    */
#define     ALI_Bit          0x40      /*仲裁丢失中断位                    */
#define     BEI_Bit          0x80      /*总线错误中断位                    */
#endif

/*总线定时寄存器的地址和位定义                                             */

#define     BusTiming0Reg    XBYTE[6]
#define     BusTiming1Reg    XBYTE[7]

#define     SAM_Bit          0x80      /*采样模式位
                                       1==总线被采样三次
                                       0==总线被采样一次                 */

/*输出控制寄存器的地址和位定义                                             */

#define     OutControlReg    XBYTE[8]
                                       /*OCMODE1，OCMODE0                  */
#define     BiPhaseMode      0x00      /*双相输出模式                      */
#define     NormalMode       0x02      /*正常输出模式                      */
#define     ClkOutMode       0x03      /*时钟输出模式                      */

                                       /*TX1 的输出管脚配置                */
#define     OCPOL1_Bit       0x20      /*输出极性控制位                    */
#define     Tx1Float         0x00      /*配置为悬空                        */
#define     Tx1PullDn        0x40      /*配置为下拉                        */
#define     Tx1PullUp        0x80      /*配置为上拉                        */
#define     Tx1PshPull       0Xc0      /*配置为推挽                        */
                                       /*TX0 的输出管脚配置                */
#define     OCPOL0_Bit       0x04      /*输出极性控制位                    */
#define     Tx0Float         0x00      /*配置为悬空                        */
#define     Tx0PullDn        0x08      /*配置为下拉                        */
#define     Tx0PullUp        0x10      /*配置为上拉                        */
#define     Tx0PshPull       0X18      /*配置为推挽                        */


/*验收代码和屏蔽寄存器的地址定义                                           */
#if defined   (PeliCANMode)
#define     AcceptCode0Reg   XBYTE[16]
#define     AcceptCode1Reg   XBYTE[17]
#define     AcceptCode2Reg   XBYTE[18]
#define     AcceptCode3Reg   XBYTE[19]
#define     AccepMask0 Reg   XBYTE[20]
#define     AccepMask1 Reg   XBYTE[21]
#define     AccepMask2 Reg   XBYTE[22]
```

```
#define     AccepMask3Reg     XBYTE[23]
#else       /*BasicCAN 模式
#define AcceptCodeReg         XBYTE[4]
#define AcceptMaskReg         XBYTE[5]
#endif

/*Rx-缓冲器的地址定义                                                    */
#if defined   (PeliCANMode)
#define     RxFramInFo        XBYTE[16]
#define     RxBuffer1         XBYTE[17]
#define     RxBuffer2         XBYTE[18]
#define     RxBuffer3         XBYTE[19]
#define     RxBuffer4         XBYTE[20]
#define     RxBuffer5         XBYTE[21]
#define     RxBuffer6         XBYTE[22]
#define     RxBuffer7         XBYTE[23]
#define     RxBuffer8         XBYTE[24]
#define     RxBuffer9         XBYTE[25]
#define     RxBuffer10        XBYTE[26]
#define     RxBuffer11        XBYTE[27]
#define     RxBuffer12        XBYTE[28]
#else       /*BasicCAN 模式
#define     RxBuffer1         XBYTE[20]
#define     RxBuffer2         XBYTE[21]
#define     RxBuffer3         XBYTE[22]
#define     RxBuffer4         XBYTE[23]
#define     RxBuffer5         XBYTE[24]
#define     RxBuffer6         XBYTE[25]
#define     RxBuffer7         XBYTE[26]
#define     RxBuffer8         XBYTE[27]
#define     RxBuffer9         XBYTE[28]
#define     RxBuffer10        XBYTE[29]
#endif

/*Tx 缓冲器的地址定义                                                    */
#if defined   (PeliCANMode)
/*仅写地址                                                              */
#define     TxFramInFo        XBYTE[16]
#define     TxBuffer1         XBYTE[17]
#define     TxBuffer2         XBYTE[18]
#define     TxBuffer3         XBYTE[19]
#define     TxBuffer4         XBYTE[20]
#define     TxBuffer5         XBYTE[21]
#define     TxBuffer6         XBYTE[22]
#define     TxBuffer7         XBYTE[23]
#define     TxBuffer8         XBYTE[24]
#define     TxBuffer9         XBYTE[25]
#define     TxBuffer10        XBYTE[26]
#define     TxBuffer11        XBYTE[27]
#define     TxBuffer12        XBYTE[28]
/*仅读地址
#define     TxFramInFoRd      XBYTE[96]
#define     TxBufferRd1       XBYTE[97]
#define     TxBufferRd2       XBYTE[98]
#define     TxBufferRd3       XBYTE[99]
#define     TxBufferRd4       XBYTE[100]
#define     TxBufferRd5       XBYTE[101]
#define     TxBufferRd6       XBYTE[102]
#define     TxBufferRd7       XBYTE[103]
#define     TxBufferRd8       XBYTE[104]
#define     TxBufferRd9       XBYTE[105]
#define     TxBufferRd10      XBYTE[106]
#define     TxBufferRd11      XBYTE[107]
#define     TxBufferRd12      XBYTE[108]
#else       /*BasicCAN 模式
#define     TxBuffer1         XBYTE[10]
```

```
#define     TxBuffer2           XBYTE[11]
#define     TxBuffer3           XBYTE[12]
#define     TxBuffer4           XBYTE[13]
#define     TxBuffer5           XBYTE[14]
#define     TxBuffer6           XBYTE[15]
#define     TxBuffer7           XBYTE[16]
#define     TxBuffer8           XBYTE[17]
#define     TxBuffer9           XBYTE[18]
#define     TxBuffer10          XBYTE[19]
#endif

/*其他寄存器的地址定义
#if defined   (PeliCANMode)
#define     ArbLostCapReg       XBYTE[11]
#define     ErrCodeCapReg       XBYTE[12]
#define     ErrWarnLimitReg     XBYTE[13]
#define     RxErrCountReg       XBYTE[14]
#define     TxErrCountReg       XBYTE[15]
#define     RxMsgCountReg       XBYTE[29]
#define     RxBufstartAdr       XBYTE[30]
#endif

/*时钟分频寄存器的地址和位定义                                                    */

#define     ClockDivideReg      XBYTE[31]

#define     DivBy1              0x07        /*CLKOUT=振荡器频率                  */
#define     DivBy2              0x00        /*CLKOUT=1/2 振荡器频率              */

#define     CLKOff_Bit          0x08        /*时钟关闭位，时钟输出管脚控制位     */
#define     RXINTEN_Bit         0x20        /*用于接收中断的管脚 TX1             */
#define     CBP_Bit             0x40        /*CAN 比较器旁路控制位               */
#define     CANMode_Bit         0x80        /*CAN 模式控制位                     */


S87C654 的寄存器和位定义
/*端口 2 寄存器“P2”                                                              */
sfr   P2        =0xA0;

Sbit   P2_7     =0xA7;                      /*端口 2 的 MSB，用于 SJA1000 的片选 */
 •
/*端口 3   P3 的复用功能                                                         */
sfr   P3        =0xB0;
 •
Sbit   int0     =0xB2;
 •
/*定时控制寄存器“TCON” */
sfr    TCON     =0x88;

 •
Sbit   IE0      =0x89;                      /*外部中断 0 边缘标志                */
Sbit   IT0      =0x88;                      /*中断 0 类型控制位（边缘或低电平触发）*/
 •

/*中断使能寄存器“IE”                                                             */
sfr    IE       =0xA8;

Sbit   EA       =0xAF;                      /*所有中断使能/禁能标志              */
 •
Sbit   EX0      =0xA8;                      /*外部中断 0 的使能或禁能位          */
 •

/*中断优先级寄存器“IP” */
```

```
sfr     IP      =0xB8;
 •
sbit    PX0     =0xB8;                  /*外部中断 0 优先级控制                */
 •
```

例子里的变量和常量的定义

```
/*-硬件/软件连接的定义----------------------------------------------------------------------------------------------*/
/*控制器：S87C654；CAN 控制器：SJA1000（见 11 页上的图 3）                                                        */
#define     CS              P2_7        /*SJA1000 的片选                        */
#define     SJAIntInp       Int0        /*SJA1000 的外部中断 0                  */
#define     SJAIntEn        EX0         /*外部中断 0 使能标志                   */

/*一使用的常量定义---------------------------------------------------------------------------------------------*/

#define     YES             1
#define     NO              0

#define     ENABLE          1
#define     DISABLE         0
#define     ENABLE_N        0
#define     DISABLE_N       1

#define     INTLEVELACT     0
#define     INTEDGEACT      1

#define     PRIORITY_LOW    0
#define     PRIORITY_HIGH   1

/*寄存器内容默认（复位）值，清除寄存器                                                                            */
#define     ClrByte         0x00

/*常量：清除中断使能寄存器                                                                                        */
#if defined   (PeliCANMode)
#define     ClrIntEnSJA     ClrByte
#else
#define     ClrIntEnSJA     ClrByte | RM_RR_Bit      /*保留复位请求            */
#endif

/*验收代码和屏蔽寄存器的定义                                                                                      */
#define     DontCare        0xFF

/*不同例子的总线定时值的定义----------------------------------------------------------------------------------*/

/*例子 AN97076 里总线定时值
    一位率                  ：25kbit/s
    一振荡器频率            ：24MHz，1，0%
    一最大允许传播延迟      ：1630ns
    一最小要求传播延迟      ：120ns
#define     PrescExample    0x02        /*波特率预分频    ：3                   */
#define     SJWExample      0xc0        /*SJW             ：4                   */
#define     TSEG1Example    0x0A        /*TSEG1           ：11                  */
#define     TSEG2Example    0x30        /*TSEG2           ：4                   */

/*总线定时值：
    一位率                  ：1MBit/s
    一振荡器频率            ：24MHz，1，0%
    一最大允许传播延迟      ：747ns
    一最小要求传播延迟      ：45ns
#define     Prec_MB_24      0x00        /*波特率预分频器  ：1                   */
#define     SJW_MB_24       0x00        /*SJW             ：1                   */
```

```
#define     TSEG1_MB_24         0x08            /*TSEG1          : 9          */
#define     TSEG2_MB_24         0x10            /*TSEG2          : 2          */

/*总线定时值：
    ―位率                      ：100kBit/s
    ―振荡器频率                ：24MHz，1，0%
    ―最大允许传播延迟          ：4250ns
    ―最小要求传播延迟          ：100ns
#define     Prec_kB_24          0x07            /*波特率预分频器  : 8          */
#define     SJW_kB_24           0xc0            /*SJW            : 4          */
#define     TSEG1_kB_24         0x09            /*TSEG1          : 10         */
#define     TSEG2_kB_24         0x30            /*TSEG2          : 4          */

/*总线定时值：
    ―位率                      ：1MBit/s
    ―振荡器频率                ：16MHz，1，0%
    ―最大允许传播延迟          ：623ns
    ―最小要求传播延迟          ：23ns
#define     Prec_MB_16          0x00            /*波特率预分频器  : 1          */
#define     SJW_MB_16           0x00            /*SJW            : 1          */
#define     TSEG1_MB_16         0x08            /*TSEG1          : 5          */
#define     TSEG2_MB_16         0x10            /*TSEG2          : 2          */

/*总线定时值：
    ―位率                      ：100kBit/s
    ―振荡器频率                ：16MHz，1，0%
    ―最大允许传播延迟          ：4450ns
    ―最小要求传播延迟          ：500ns
#define     Prec_kB_16          0x04            /*波特率预分频器  : 5          */
#define     SJW_kB_16           0xC0            /*SJW            : 4          */
#define     TSEG1_kB_16         0x0A            /*TSEG1          : 11         */
#define     TSEG2_kB_16         0x30            /*TSEG2          : 4          */

/*总线定时值的结束----------------------------------------------------------------------------------------------*/

/*使用过的变量的定义--------------------------------------------------------------------------------------------*/
/*中断寄存器内容的中间存储内容----------------------------------------------------------------------------------*/
BYTE bdata CANInterrupt;                        /*位可寻址的字节                */
sbit    RI_BitVar       = CANInterrupt  ^   0;
sbit    TI_ BitVar      = CANInterrupt  ^   1;
sbit    EI_ BitVar      = CANInterrupt  ^   2;
sbit    DOI_ BitVar     = CANInterrupt  ^   3;
sbit    WUI_ BitVar     = CANInterrupt  ^   4;
sbit    EPI_ BitVar     = CANInterrupt  ^   5;
sbit    ALI_ BitVar     = CANInterrupt  ^   6;
sbit    BEI_ BitVar     = CANInterrupt  ^   7;
```