

PHILIPS LPC2000 系列 ARM7 微处理器

CAN 控制器驱动程序的编写与开发

广州周立功单片机发展有限公司

2004 年 8 月 25 日

目录

第 1 章	CAN 控制器和验收过滤器	4
1.1	特性	4
1.2	管脚描述	4
1.3	CAN 模块的存储器映射	4
1.4	CAN 控制器寄存器一览表	5
1.5	各控制寄存器解释	7
1.5.1	模式寄存器 (CANMOD – 0xE00x x000)	7
1.5.2	命令寄存器 (CANCMDR – 0xE00x x004)	7
1.5.3	全局状态寄存器 (CANGSR – 0xE00x x008)	8
1.5.4	中断和捕获寄存器 (CANICR – 0xE00x x00C)	9
1.5.5	中断使能寄存器 (CANIER – 0xE00x x010)	10
1.5.6	总线时序寄存器 (CANBTR – 0xE00x x014)	11
1.5.7	出错警告界限寄存器 (CANEWL – 0xE00x x018)	11
1.5.8	状态寄存器 (CANSR – 0xE00x x01C)	12
1.5.9	Rx 帧状态寄存器 (CANRFS – 0xE00x x020)	12
1.5.10	Rx 标识符寄存器 (CANRID – 0xE00x x024)	13
1.5.11	Rx 数据寄存器 A (CANRDA – 0xE00x x028)	13
1.5.12	Rx 数据寄存器 B (CANRDB – 0xE00x x02C)	14
1.5.13	Tx 帧信息寄存器 (CANTFI1,2,3 – 0xE00x x030,40,50)	14
1.5.14	Tx 标识符寄存器 (CANTID1,2,3 – 0xE00x x034,44,54)	14
1.5.15	Tx 数据寄存器 A (CANTDA1,2,3 – 0xE00x x038,48,58)	15
1.5.16	Tx 数据寄存器 B (CANTDB1,2,3 – 0xE00x x03C,4C,5C)	15
1.6	寄存器操作方法	16
1.6.1	错误处理	17
1.6.2	睡眠模式	18
1.6.3	中断	18
1.6.4	发送优先级	19
1.7	组合 CAN 寄存器	19
1.7.1	集中发送状态寄存器 (CANTxSR – 0xE004 0000)	19
1.7.2	集中接收状态寄存器 (CANRxSR – 0xE004 0004)	20
1.7.3	集中其他状态寄存器 (CANMSR – 0xE004 0008)	20
1.8	全局验收过滤器	21
1.8.1	验收过滤器寄存器	22
1.8.2	标准帧单个起始地址寄存器 (SFF_sa – 0xE003 C004)	22
1.8.3	标准帧组起始地址寄存器 (SFF_GRP_sa – 0xE003 C008)	23
1.8.4	扩展帧起始地址寄存器 (EFF_sa – 0xE003 C00C)	23
1.8.5	扩展帧组起始地址寄存器 (EFF_GRP_sa – 0xE003 C010)	23
1.8.6	AF 表格终止寄存器 (ENDofTable – 0xE003 C014)	23

1.8.7 LUT 错误地址寄存器 (LUTerrAd – 0xE003 C018)	24
1.8.8 LUT 错误寄存器 (LUTerr – 0xE003 C01C)	24
1.8.9 验收过滤器表格和 ID 索引值举例.....	24
1.8.10 FullCAN 模式.....	25
第 2 章 CAN 控制器驱动程序的编写	28
2.1 驱动程序的结构.....	28
2.2 驱动程序使用方法.....	31
2.3 驱动程序各文件介绍.....	31
2.3.1 用户配置文件.....	31
2.3.2 CAN 驱动应用接口层文件	33
2.3.3 功能函数层文件.....	41
2.3.4 硬件抽象层文件.....	44
第 3 章 CAN 驱动程序应用实例——RS232 与 CAN-bus 透明转换器	45
3.1 硬件平台.....	45
3.2 软件平台.....	46
3.3 应用注意事项.....	46
3.4 实现的功能说明.....	47
3.5 测试方法说明.....	47
3.6 简单的 RS232 数据与 CAN-bus 数据透明转换的实现.....	49
3.6.1 系统初始化.....	49
3.6.2 主循环处理.....	50
第 4 章 参考文献.....	54

第1章 CAN 控制器和验收过滤器

LPC2119/2129/2194/2290/2292/2294 微处理器包含 2 或 4 个 CAN 模块,可同时支持多个 CAN 总线的操作,使器件可用作网关、开关、工业或汽车应用中多个 CAN 总线的路由器。

1.1 特性

- 2 或 4 个 (LPC2119/2129/2292 或 LPC2290/2194/2294) CAN 控制器和总线 (64 脚封装中包含 2 条, 144 脚的封装中包含 4 条)。
- 每个总线的数据波特率可达 1Mbps;
- 可访问 32 位的寄存器和 RAM;
- 符合 CAN 规范 CAN2.0B, ISO 11898-1;
- 全局验收过滤器可识别几乎所有总线的 11 和 29 位 Rx 标识符;
- 验收过滤器为选择的标准标识符提供了 FullCAN-style 自动接收。

1.2 管脚描述

CAN 控制器输入输出管脚表 1.1 所示。

表 1.1 CAN 管脚描述

管脚名称	类型	描述
RX4、RX3 RX2、RX1	输入	串行输入。来自 CAN 收发器。 注: RX2 和 RX1 适用于所有含有 CAN 模块的器件。但 RX4 和 RX3 只可用于 144 管脚含有 CAN 模块的某些器件。
TX4、TX3 TX2、TX1	输出	串行输出。输出到 CAN 收发器。 注: TX2 和 TX1 适用于所有含有 CAN 模块的器件。但 TX4 和 TX3 只可用于 144 管脚含有 CAN 模块的某些器件。

1.3 CAN 模块的存储器映射

CAN 控制器和验收过滤器占用部分 VPB slot 空间, 表 1.2 所示。

表 1.2 CAN 模块的存储器映射

地址范围	用途
E003 8000 – 87FF	验收过滤器 RAM (2048 字节)
E003 C000 – C017	验收过滤器寄存器
E004 0000 – 000B	中央 CAN 寄存器
E004 4000 – 405F	CAN 控制器 1 寄存器
E004 8000 – 805F	CAN 控制器 2 寄存器
E004 C000 – C05F	CAN 控制器 3 寄存器
E005 0000 – 005F	CAN 控制器 4 寄存器

1.4 CAN 控制器寄存器一览表

CAN 模块使用的寄存器见表 1.3 和表 1.4 所示。详情见后面的描述。

表 1.3 CAN 验收过滤器和中央 CAN 寄存器

名称	描述	访问	复位值	地址
AFMR	验收过滤器寄存器	R/W	1	0xE003 C000
SFF_sa	标准帧单个起始地址寄存器	R/W	0	0xE003 C004
SFF_GRP_sa	标准帧组起始地址寄存器	R/W	0	0xE003 C008
EFF_sa	扩展帧起始地址寄存器	R/W	0	0xE003 C00C
EFF_GRP_sa	扩展帧组起始地址寄存器	R/W	0	0xE003 C010
ENDofTable	AF 表格终止寄存器	R/W	0	0xE003 C014
LUTerrAd	LUT 错误地址寄存器	RO	0	0xE003 C018
LUTerr	LUT 错误寄存器	RO	0	0xE003 C01C
CANTxSR	CAN 集中发送状态寄存器	RO	0x003F 3F00	0xE004 0000
CANRxSR	CAN 集中接受状态寄存器	RO	0	0xE004 0004
CANMSR	CAN 集中其他状态寄存器	RO	0	0xE004 0008

表 1.4 CAN1,CAN2,CAN3 和 CAN4 控制器寄存器映射

名称	描述	访问	CAN1 地址&名称	CAN2 地址&名称	CAN3 地址&名称	CAN4 地址&名称
CANMOD	控制 CAN 控制器的工作模式	R/W	0xE0044000 C1MOD	0xE0048000 C2MOD	0xE004C000 C3MOD	0xE0050000 C4MOD
CANCMR	影响 CAN 控制器状态的命令位	WO	0xE0044004 C1CMR	0xE0048004 C2CMR	0xE004C004 C3CMR	0xE0050004 C4CMR
CANGSR	全局控制器状态和错误计数器	RO ^a	0xE0044008 C1GSR	0xE0048008 C2GSR	0xE004C008 C3GSR	0xE0050008 C4GSR
CANICR	中断状态, 仲裁丢失捕获, 错误码捕获	RO	0xE004400C C1ICR	0xE004800C C2ICR	0xE004C00C C3ICR	0xE005000C C4ICR
CANIER	中断使能	R/W	0xE0044010 C1IER	0xE0048010 C2IER	0xE004C010 C3IER	0xE0050010 C4IER
CANBTR	总线时序	R/W ^b	0xE0044014 C1BTR	0xE0048014 C2BTR	0xE004C014 C3BTR	0xE0050014 C4BTR
CANEWL	出错警告界限	R/W ^b	0xE0044018 C1EWL	0xE0048018 C2EWL	0xE004C018 C3EWL	0xE0050018 C4EWL
CANSR	状态寄存器	RO	0xE004401C C1SR	0xE004801C C2SR	0xE004C01C C3SR	0xE005001C C4SR
CANRFS	接收帧状态	R/W ^b	0xE0044020 C1RFS	0xE0048020 C2RFS	0xE004C020 C3RFS	0xE0050020 C4RFS

续表 1.4

名称	描述	访问	CAN1 地址&名称	CAN2 地址&名称	CAN3 地址&名称	CAN4 地址&名称
CANRID	接收到的标识符	R/W ^b	0xE0044024 C1RID	0xE0048024 C2RID	0xE004C024 C3RID	0xE0050024 C4RID
CANRDA	接收到的数据字节 1-4	R/W ^b	0xE0044028 C1RDA	0xE0048028 C2RDA	0xE004C028 C3RDA	0xE005 0028 C4RDA
CANRDB	接收到的数据字节 5-8	R/W ^b	0xE004402C C1RDB	0xE004802C C2RDB	0xE004C02C C3RDB	0xE005002C C4RDB
CANTFI1	发送帧信息(1)	R/W	0xE0044030 C1TFI1	0xE0048030 C2TFI1	0xE004C030 C3TFI1	0xE0050030 C4TFI1
CANTID1	发送标识符(1)	R/W	0xE0044034 C1TID1	0xE0048034 C2TID1	0xE004C034 C3TID1	0xE0050034 C4TID1
CANTDA1	发送数据字节 1-4(1)	R/W	0xE0044038 C1TDA1	0xE0048038 C2TDA1	0xE004C038 C3TDA1	0xE0050038 C4TDA1
CANTDB1	发送数据字节 5-8(1)	R/W	0xE004403C C1TDB1	0xE004803C C2TDB1	0xE004C03C C3TDB1	0xE005003C C4TDB1
CANTFI2	发送帧信息(2)	R/W	0xE0044040 C1TFI2	0xE0048040 C2TFI2	0xE004C040 C3TFI2	0xE0050040 C4TFI2
CANTID2	发送标识符(2)	R/W	0xE0044044 C1TID2	0xE0048044 C2TID2	0xE004C044 C3TID2	0xE0050044 C4TID2
CANTDA2	发送数据字节 1-4(2)	R/W	0xE0044048 C1TDA2	0xE0048048 C2TDA2	0xE004C048 C3TDA2	0xE0050048 C4TDA2
CANTDB2	发送数据字节 5-8(2)	R/W	0xE004404C C1TDB2	0xE004804C C2TDB2	0xE004C04C C3TDB2	0xE005004C C4TDB2
CANTFI3	发送帧信息(3)	R/W	0xE0044050 C1TFI3	0xE0048050 C2TFI3	0xE004C050 C3TFI3	0xE0050050 C4TFI3
CANTID3	发送标识符(3)	R/W	0xE0044054 C1TID3	0xE0048054 C2TID3	0xE004C054 C3TID3	0xE0050054 C4TID3
CANTDA3	发送数据字节 1-4(3)	R/W	0xE0044058 C1TDA3	0xE0048058 C2TDA3	0xE004C058 C3TDA3	0xE0050058 C4TDA3
CANTDB3	发送数据字节 5-8(3)	R/W	0xE004405C C1TDB3	0xE004805C C2TDB3	0xE004C05C C3TDB3	0xE005005C C4TDB3

a. 错误计数器只能在 CANMOD 的 RM 为 1 时才能写入。

b. 这些寄存器只能在 CANMOD 的 RM 为 1 时才能写入。

下面的寄存器表中，“复位值”列表示硬件复位对每个位域的影响，“RM 置位”列指示了当 RM 位被软件置位或由于总线离线而置位时对每个位域的影响。**注意：当 RM 由于硬件复**

位而置位时，如果某一位/域对应在“复位值”列和“RM 置位”列中的值不同，使用“复位值”列的值。在“复位值”列和“RM 置位”列中，X 表示该位/域的值不变。

1.5 各控制寄存器解释

1.5.1 模式寄存器（CANMOD – 0xE00x x000）

该寄存器控制 CAN 控制器的基本工作模式。表中未列出的寄存器位读数为 0，并应向其写入 0，模式寄存器功能见表 1.5。每个 CAN 通道寄存器的地址见表 1.4。

表 1.5 CAN 模式寄存器（CANMOD – 0xE00x x000）

CAN MOD	名称	功能	复位值	RM 置位
0	RM	0: CAN 控制器正常工作，某些寄存器不能写入。 1: 复位模式——禁止 CAN，可写寄存器可以写入数据。	1	1
1	LOM	0: 当 CAN 总线成功接收到信息时，CAN 控制器产生应答。 1: 只听模式——控制器不会在 CAN 总线上产生应答，即使成功接收到信息。此时不发送信息，控制器工作在“错误被动”状态。该模式用于软件位速率检测和“热插入”。	0	X
2	STM	0: 发送的信息必须被应答才被认可。 1: 自测试模式——控制器认可没有应答的 Tx 信息传输。此状态和 CANCMR 的 SRR 位一起使用，可以达到自发自收的效果。	0	X
3	TPM	0: 3 个发送缓冲区的优先级由各自的 CAN ID 决定。 1: 3 个发送缓冲区的优先级由各自的 Tx 优先级域决定。	0	X
4	SM	0: 正常工作模式。 1: 睡眠模式——如果无中断请求发生且没有总线活动，CAN 控制器进入睡眠模式。见后面的睡眠模式描述。	0	0
5	RPM	0: 如果传输的是显性位，Rx 和 Tx 脚为低电平。 1: 翻转极性模式——如果传输的是显性位，Rx 脚为高电平。	0	0
7	TM	0: 正常工作模式。 1: 测试模式。Rx 脚的状态被记录到 Tx 脚上。	0	0

注：LOM 和 STM 位只有在 RM 位为 1 时才能被写入。

1.5.2 命令寄存器（CANCMR – 0xE00x x004）

通过写命令寄存器来启动一次动作。表中未列出的寄存器位必须写入 0。该寄存器读出总为 0，命令寄存器功能见表 1.6。

表 1.6 CAN 命令寄存器（CANCMD – 0xE00x x004）

CANCMD	名称	功能
0	TR	1: 发送请求——报文，先前写入 CANTFI 和 CANTID 的内容以及 CANTDA 和 CANTDB 寄存器的内容等待发送。
1	AT	1: 中止发送——所有等待发送请求取消，正在处理的发送请求除外。如果该位和 TR 在同一次写操作中置位，尝试执行一次单帧发送，如果既未建立错误标志仲裁也未丢失，则不再执行帧发送。

续表 1.6

CANCMR	名称	功能
2	RRB	1: 释放接收缓冲区——如果 CANRDA 和 CANRDB 寄存器的内容可用, CANRFS,CANRID 的内容释放并可由接收到的下一帧信息来代替。如果接收到的下一帧信息不可用, 写入该命令来清除 CANSR 的 RBS 位。
3	CDO	1: 清除数据溢出——清除 CANSR 的数据溢出位。
4	SRR	1: 自接收请求——报文, 先前写入 CANTFS 和 CANTID 的内容以及 CANTDA 和 CANTDB 寄存器的内容等待发送。该位与 TR 位的不同在于: 发送过程中接收器并未禁能, 接收器仍能够接收符合验收过滤器识别标识的信息。
5	STB1	1: 选择发送 Tx 缓冲区 1 的内容。
6	STB2	1: 选择发送 Tx 缓冲区 2 的内容。
7	STB3	1: 选择发送 Tx 缓冲区 3 的内容。

1.5.3 全局状态寄存器 (CANGSR – 0xE00x x008)

该寄存器是一个只读寄存器, 但当 CANMOD 寄存器的 RM 位为 1 时, 错误计数器可写入。表中未列出的寄存器位读数为 0, 并应向其写入 0, 全局状态寄存器功能见表 1.7

表 1.7 CAN 全局状态寄存器 (CANGSR – 0xE00x x008)

CANGSR	名称	功能	复位值	RM 置位
0	RBS	1: 接收缓冲区状态——如果 CANRDA 和 CANRDB 寄存器有效, CANRFS 和 CANRID 内接收到的信息可用, 只要以后再无可用的接收信息, 该位通过 CANCMR 的释放接收缓冲区命令清除。	0	0
1	DOS	1: 数据溢出状态——由于前面传输到 CAN 控制器的数据未被读出, 而接收缓冲区没有及时释放, 从而引起后面信息丢失。 0: 自从上次的清除数据溢出命令被写入 CANCMR (或自从复位后) 后就无数据溢出发生。	0	0
2	TBS	1: 发送缓冲区状态——无发送信息 (3 个 Tx 缓冲区中任何一个的信息) 等待 CAN 控制器处理, 软件可以向 CANTFI、CANTID、CANTDA 和 CANTDB 寄存器写入数据。 0: 至少有一条前面队列中的到 CAN 控制器的信息未被发送, 因此, 软件不应该向存放信息的那个 (那些) Tx 缓冲区的 CANTFI、CANTID、CANTDA 和 CANTDB 寄存器写入数据。	1	X
3	TCS	1: 发送结束状态——所有发送请求都被成功处理。 0: 至少有一个请求发送未处理。	1	0

续表 1.7

CANGSR	名称	功能	复位值	RM 置位
4	RS	1: 接收状态: CAN 控制器正在接收数据。	0	0
5	TS	1: 发送状态: CAN 控制器正在发送数据。	0	0
6	ES	1: 错误状态: 发送和接收错误计数器的计数值或其中一个的计数值已经达到出错警告界限寄存器中设置的阈值。	0	0
7	BS	1: 总线状态: 由于发送错误计数器的计数值已达到其阈值 255, CAN 控制器目前被禁止。	0	0
23:16	RXERR	Rx 错误计数器的当前值。	0	X
31:24	TXERR	Tx 错误计数器的当前值。	0	X

1.5.4 中断和捕获寄存器 (CANICR – 0xE00x x00C)

CANICR 寄存器的位用来表征 CAN 总线上出现的事件。它是一个只读寄存器。表中未列出的寄存器位读数为 0, 应向其写入 0。中断和捕获寄存器功能见表 1.8。

位 1-9 读出时被清零, 位 16-23 在总线出现错误时捕获。同时, 如果 CANIER 的 BEIE 位为 1, 则 CANICR 的 BEI 位置位并产生 CAN 中断。位 24-31 在 CAN 仲裁丢失时捕获。同时, 如果 CANIER 的 ALIE 位为 1, 则 CANICR 的 ALI 位置位并产生 CAN 中断。一旦上述这些字节被捕获, 它们的值将保持不变直到被读出 (该值被读取后, 将释放寄存器以捕获新的值)。

不管如何进行读取寄存器操作, 清除位 1-9 以及释放位 16-23 和 24-31 的操作都在读 CANICR 时完成。这就表明, 软件总是按字的方式来读取 CANICR, 并根据实际应用的需要对所有寄存器的位进行处理。

表 1.8 CAN 中断和捕获寄存器 (CANICR – 0xE00x x00C)

CANICR	名称	功能	复位值	RM 置位
0	RI	1: 接收中断...当 CANSR 的 RBS 位和 CANIER 的 RIE 位都为 1 时该位置位, 表示接收到一条有用信息。	0	0
1	TI1	1: 发送中断 1...当 CANSR 的 TBS1 位从 0 变为 1 且 CANIER 的 TIE1 位为 1 时该位置位, 表明发送缓冲区 1 的数据可用。	0	0
2	EI	1: 出错警告中断...每当 CANSR 的错误状态或总线状态位的状态改变 (置位或清零), 且 CANIER 的 EIE 位此时为 1 时, 该位置位。	0	X
3	DOI	1: 数据溢出中断...当 CANSR 的 DOS 位从 0 变为 1, 且 CANIE 的 DOIE 位为 1 时, 该位置位。	0	0
4	WUI	1: 唤醒中断: 当 CAN 控制器处于睡眠状态时检测到总线活动, 并且 CANIE 的 WUIE 位为 1 时, 该位置位。	0	0
5	EPI	1: 错误认可中断...当 CANIE 的 EPIE 位为 1, 且 CAN 控制器在错误认可模式和错误主动模式两者之间进行任何方向的切换时, 该位置位。	0	0

续表 1.8

CANICR	名称	功能	复位值	RM 置位
6	ALI	1: 仲裁丢失中断...当 CANIE 的 ALIE 位为 1, 且 CAN 控制器在准备发送数据时丢失了仲裁, 该位置位。	0	0
7	BEI	1: 总线错误中断...当 CANIE 的 BEIE 位为 1, 且 CAN 控制器检测到总线错误时, 该位置位。	0	X
8	IDI	1: ID 准备就绪中断...当 CANIE 的 IDIE 位为 1, 且接收到一个 CAN 标识符时, 该位置位。	0	0
9	TI2	1: 发送中断 2...当 CANSR 的 TBS2 位从 0 变为 1, 且 CANIER 的 TIE2 位为 1 时, 该位置位, 表明发送缓冲区 2 可用。	0	0
10	TI3	1: 发送中断 3...当 CANSR 的 TBS3 位从 0 变为 1, 且 CANIER 的 TIE3 位为 1 时, 该位置位, 表明发送缓冲区 3 可用。	0	0
20:16	ERRBIT	错误码捕获: 这些捕获值反映了内部状态变量, 并不表示内部状态变化的线性规律。 00011: 帧起始 00010: ID28:21 00110: ID20:18 00100: SRTR 位 00101: IDE 位 00111: ID17:13 01111: ID12:5 01110: ID4:0 01100: RTR 位 01011: DLC 01010: 数据域 01000: CRC 11000: CRC 分隔符 11001: Ack slot 11011: Ack 分隔符 11011: 帧结束 10010: 间断 10001: 主动错误标志 10110: 认可错误标志 10011: 显性 OK 位 10111: 错误间隔 11000: 超载标志 读该字节用来使能捕获另一个总线错误中断。	0	X
21	ERRDIR	当 CAN 控制器检测到总线错误时, 当前位的传输方向捕获到该位。1=接收, 0=发送。	0	X
23:22	ERRC	当 CAN 控制器检测到总线错误时, 错误类型被捕获到该域: 00=位错误, 01=格式错误, 10=填充错误, 11=其它错误。	0	X
28:24	ALCBIT	如果在向 CAN 总线发送数据时出现仲裁丢失, 数据帧的位编号被捕获到该域。0 表明仲裁丢失发生在标识符的第一位 (MS) ...31 表明仲裁丢失发生在扩展帧的 RTR 位。该字节读出后, ALI 位清零, 允许发生新的仲裁丢失中断。	0	X

1.5.5 中断使能寄存器 (CANIER – 0xE00x x010)

CANIER 是一个读/写寄存器, 它控制着 CAN 控制器出现的事件是否能产生中断。该寄存器的位 7:0 与 CANICR 寄存器的位 7:0 一一对应, 中断使能寄存器功能见表 1.9。

表 1.9 CAN 中断使能寄存器 (CANIER – 0xE00x x010)

CANIER	名称	功能	复位值	RM 置位
0	RIE	接收器中断使能	0	X
1	TIE1	发送中断使能(1)	0	X
2	EIE	出错警告中断使能	0	X
3	DOIE	数据溢出中断使能	0	X
4	WUIE	唤醒中断使能	0	X
5	EPIE	错误认可中断使能	0	X
6	ALIE	仲裁丢失中断使能	0	X
7	BEIE	总线错误中断使能	0	X
8	IDIE	ID 准备就绪中断使能	0	X
9	TIE2	发送中断使能(2)	0	X
10	TIE3	发送中断使能(3)	0	X

1.5.6 总线时序寄存器 (CANBTR – 0xE00x x014)

CANBTR 寄存器控制着由微处理器的 VPB 时钟驱动的各种 CAN 时序。该寄存器可随时读取，但只能在 CANMOD 的 RM 位为 1 时写入，总线时序寄存器功能见表 1.10。

表 1.10 CAN 总线时序寄存器 (CANBTR - 0xE00x x014)

CANBTR	名称	功能	复位值	RM 置位
0:9	BRP	波特率预分频器。VPB 时钟被（该域的值+1）分频来产生 CAN 时钟。	0	X
15:14	SJW	同步跳变宽度为（该值+1）个 CAN 时钟周期。	0	X
19:16	TSEG1	指定同步点到采样点的延时时间为（该域的值+1）个 CAN 时钟周期。	1100	X
22:20	TSEG2	采样点到下个指定同步点的延时时间为（该域的值+1）个 CAN 时钟周期。指定的 CAN 位时间为（该域的值+TSEG1 的值+3）个 CAN 时钟周期。	001	X
23	SAM	1：总线采样 3 次（建议用于低到中速的总线）； 0：总线采样 1 次（建议用于高速总线）。	0	X

1.5.7 出错警告界限寄存器 (CANEWL – 0xE00x x018)

CANEWL 寄存器设定了一个 Tx 或 Rx 出错阈值，可用于产生中断。该寄存器可随时读取，但只能在 CANMOD 的 RM 位为 1 时写入，具体功能见表 1.11。

表 1.11 CAN 出错警告界限寄存器 (CANEWL – 0xE00x x018)

CANEWL	名称	功能	复位值	RM 置位
7:0	EWL	在 CAN 操作过程中，该值与 Tx 和 Rx 错误计数器的计数器相比较，如果其中一个计数值与该值相等，CANSR 的错误状态 (ES) 位置位。	9610=0x60	X

1.5.8 状态寄存器（CANSR – 0xE00x x01C）

CANSR 寄存器含有 3 个状态字节，状态字节中与发送无关的位和全局状态寄存器中相应的位相同，与发送有关的位反映了 3 个 Tx 缓冲区的状态，具体功能见表 1.12。

表 1.12 CAN 状态寄存器（CANSR – 0xE00x x01C）

CANSR	名称	功能	复位值	RM 置位
0,8,16	RBS	这些位与 GSR 寄存器的 RBS 位相同。	0	0
1,9,17	DOS	这些位与 GSR 寄存器的 DOS 位相同。	0	0
2,10,18	TBS1,TBS2, TBS3	1: 软件可以向这个 Tx 缓冲区的 CANTFI、CANTID、CANTDA 和 CANTDB 寄存器写入数据。 0: 软件不能向这个 Tx 缓冲区的 CANTFI、CANTID、CANTDA 和 CANTDB 寄存器写入数据。	1	X
3,11,19	TCS1,TCS2, TCS3	1: 这个 Tx 缓冲区的所有发送请求都已经成功处理。 0: 这个 Tx 缓冲区的发送请求并未完全处理。	1	0
4,12,20	RS	这些位与 GSR 寄存器的 RS 位相同。	0	0
5,13,21	TS1,TS2, TS3	1: CAN 控制器正在发送 Tx 缓冲区的信息。	0	0
6,14,22	ES	这些位与 GSR 寄存器的 ES 位相同。	0	0
7,15,23	BS	这些位与 GSR 寄存器的 BS 位相同。	0	0

1.5.9 Rx 帧状态寄存器（CANRFS – 0xE00x x020）

CANRFS 寄存器用来定义当前接收到的信息的特性。正常工作模式下只能对它执行读操作，但可在 CANMOD 的 RM 位为 1 时被写入以用作测试，具体功能见表 1.13。

表 1.13 CAN Rx 帧状态寄存器（CANRFS – 0xE00x x020）

CANRFS	名称	功能	复位值	RM 置位
9:0	ID 索引	如果 BP 位（见下行）为 0，该字段的值是从零开始的查询表格 RAM 行的编号，验收过滤器利用该值来匹配接收标识符。标准表格中的禁能行也设定有编号，但不参与匹配。详见“验收过滤器表格和 ID 索引值举例”中的 ID 索引值举例。	0	X
10	BP	如果该位为 1，则当前接收的信息处于验收过滤器（AF）旁路（Bypass）模式，ID 索引字段（见上行）无意义。	0	X
19:16	DLC	该域包含了当前接收到的信息的数据长度代码（DLC）。当 RTR=0 时，它与 CANRDA 和 CANRDB 寄存器中的数据字节数有关： 0000-0111=0~7 个字节 1000-1111=8 个字节 RTR=1 时，DLC 用来指示请求发回的数据字节数，使用相同的编码方式。	0	X

续表 1.13

CANRFS	名称	功能	复位值	RM 置位
30	RTR	该位包含了当前接收到的信息的远程发送请求位。0 表明接收到一个数据帧，数据可从 CANRDA 和 CANRDB 寄存器中读出（如果 DLC 不为零）。1 表明接收到一个远程帧，这时 DLC 的值用来表示使用相同识别符的请求发送的数据字节数。	0	X
31	FF	该位为 0 时，表明当前接收到的信息包含 11 位标识符；该位为 1 时，表明当前接收到的信息包含 29 位标识符。FF 的值直接影响着下面将要描述的 CANID 寄存器的内容。	0	X

1.5.10 Rx 标识符寄存器（CANRID – 0xE00x x024）

该寄存器包含了当前接收到的信息的标识符字段。正常工作模式下只能对它执行读操作，但可在 CANMOD 的 RM 位为 1 时被写入以用作测试。CANRID 寄存器有 2 种不同格式，由 CANRFS 寄存器中 FF 位决定，标准帧格式见表 1.14，扩展帧格式见表 1.15。

表 1.14 CAN Rx 标识符寄存器（FF=0）（CANRID – 0xE00x x024）

CANRID	名称	功能	复位值	RM 置位
10:0	ID	当前接收到的信息的 11 位标识符字段。这些位在 CAN2.0A 规范被称为 ID10-0，在 CAN2.0B 中被称为 ID29-18。	0	X

表 1.15 CAN Rx 标识符寄存器（FF=1）（CANRID – 0xE00x x024）

CANRID	名称	功能	复位值	RM 置位
28:0	ID	当前接收到的信息的 29 位标识符字段。这些位在 CAN2.0B 规范中被称为 ID29-0。	0	X

1.5.11 Rx 数据寄存器 A（CANRDA – 0xE00x x028）

CANRDA 寄存器包含了当前接收到的信息的前面 1~4 个数据字节。正常工作模式下只能对它执行读操作，但可在 CANMOD 的 RM 位为 1 时被写入以用作测试，具体功能见表 1.16。

表 1.16 CAN Rx 数据寄存器 A（CANRDA – 0xE00x x028）

CANRDA	名称	功能	复位值	RM 置位
7:0	数据 1	如果 CANRFS 寄存器的 DLC 值>=0001，该域包含的是当前接收到的信息的第 1 个数据字节。	0	X
15:8	数据 2	如果 CANRFS 寄存器的 DLC 值>=0010，该域包含的是当前接收到的信息的第 2 个数据字节。	0	X
23:16	数据 3	如果 CANRFS 寄存器的 DLC 值>=0011，该域包含的是当前接收到的信息的第 3 个数据字节。	0	X
31:24	数据 4	如果 CANRFS 寄存器的 DLC 值>=0100，该域包含的是当前接收到的信息的第 4 个数据字节。	0	X

1.5.12 Rx 数据寄存器 B (CANRDB - 0xE00x x02C)

CANRDB 寄存器包含了当前接收到的信息的第 5~8 个数据字节。正常工作模式下只能对它执行读操作,但可在 CANMOD 的 RM 位为 1 时被写入以用作测试,具体功能见表 1.17。

表 1.17 CAN Rx 数据寄存器 B (CANRDB - 0xE00x x02C)

CANRDB	名称	功能	复位值	RM 置位
7:0	数据 5	如果 CANRFS 寄存器的 DLC 值>=0101, 该域包含的是当前接收到的信息的第 5 个数据字节。	0	X
15:8	数据 6	如果 CANRFS 寄存器的 DLC 值>=0110, 该域包含的是当前接收到的信息的第 6 个数据字节。	0	X
23:16	数据 7	如果 CANRFS 寄存器的 DLC 值>=0111, 该域包含的是当前接收到的信息的第 7 个数据字节。	0	X
31:24	数据 8	如果 CANRFS 寄存器的 DLC 值>=1000, 该域包含的是当前接收到的信息的第 8 个数据字节。	0	X

1.5.13 Tx 帧信息寄存器 (CANTFI1,2,3 - 0xE00x x030,40,50)

当 CANSR 的 TBSn 位为 1 时, 软件可以向 CANTFIn 写入数据, 来定义对应的 Tx 缓冲区中下一条将发送信息的格式。表中未列出的寄存器位读数为 0, 并应向其写入 0, 具体功能见表 1.18。

表 1.18 CAN Tx 帧信息寄存器 (CANTFI1,2,3 - 0xE00x x030,40,50)

CANTFI	名称	功能	复位值	RM 置位
7:0	PRI0	如果 CANMOD 寄存器的 TPM 位为 1, 根据该域的值, 使能的 Tx 缓冲区将竞争信息的发送权。PRI0 的值越小, 优先权就越高。		
19:16	DLC	该值将从下一条发送信息的 DLC 字段中发送出去。如果 RTR=0, DLC 控制着下一条发送信息的数据字节数, 发送信息来自 CANTDA 和 CANTDB 寄存器: 0000-0111=0~7 个字节 1xxx=8 个字节	0	X
30	RTR	该值将从下一条发送信息的 RTR 位中发送出去。如果该位为 0, DLC 字段调用的数据字节数将从 CANTDA 和 CANTDB 寄存器中发送出去。如果该位为 1, 发送远程帧, 要求传输的数据字节数包含在帧内。	0	X
31	FF	该位为 0 时, 发送的下一条信息包含 11 位的标识符。 该位为 1 时, 发送的下一条信息包含 29 位的标识符。	0	X

1.5.14 Tx 标识符寄存器 (CANTID1,2,3 - 0xE00x x034,44,54)

当 CANSR 的 TBSn 位为 1 时, 软件可以向 CANTFIn 写入数据来定义下一条发送信息的标识符。表中未列出的寄存器位读数为 0, 并应向其写入 0。该寄存器有 2 种不同格式, 由 CANTFIn 寄存器的 FF 位决定。标准帧格式见表 1.19, 扩展帧格式见表 1.20。

表 1.19 CAN Tx 标识符寄存器 (FF=0) (CANTID1,2,3 – 0xE00x x034,44,54)

CANTID	名称	功能	复位值	RM 置位
10:0	ID	下条发送信息包含 11 位的标识符。	0	X

表 1.20 CAN Tx 标识符寄存器 (FF=1) (CANTID1,2,3 – 0xE00x x034,44,54)

CANTID	名称	功能	复位值	RM 置位
28:0	ID	下条发送信息包含 29 位的标识符。	0	X

1.5.15 Tx 数据寄存器 A (CANTDA1,2,3 – 0xE00x x038,48,58)

当 CANSR 的 TBSn 位为 1 时，软件写入 CANTDAn 寄存器来定义下一条信息的前 1~4 个数据字节，具体功能见表 1.21。

表 1.21 CAN Tx 数据寄存器 A (CANTDA1,2,3 – 0xE00x x038,48,58)

CANTDA	名称	功能	复位值	RM 置位
7:0	数据 1	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0001，该字节作为下条信息的第 1 个数据字节。	0	X
15:8	数据 2	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0010，该字节作为下条信息的第 2 个数据字节。	0	X
23:16	数据 3	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0011，该字节作为下条信息的第 3 个数据字节。	0	X
31:24	数据 4	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0100，该字节作为下条信息的第 4 个数据字节。	0	X

1.5.16 Tx 数据寄存器 B (CANTDB1,2,3 – 0xE00x x03C,4C,5C)

当 CANSR 的 TBSn 位为 1 时，软件写入 CANTDAn 寄存器来定义下一条信息的第 5~8 个数据字节，具体功能见表 1.22。

表 1.22 CAN Tx 数据寄存器 B (CANTDB1,2,3 – 0xE00x x03C,4C,5C)

CANRDB	名称	功能	复位值	RM 置位
7:0	数据 5	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0101，该字节作为下条信息的第 5 个数据字节。	0	X
15:8	数据 6	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0110，该字节作为下条信息的第 6 个数据字节。	0	X
23:16	数据 7	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=0111，该字节作为下条信息的第 7 个数据字节。	0	X
31:24	数据 8	如果 CANTFI 寄存器的 RTR=0 且 DLC 值>=1000，该字节作为下条信息的第 8 个数据字节。	0	X

1.6 寄存器操作方法

我们以 CAN 的模式寄存器的操作方法为例，介绍操作 CAN 寄存器的方法。其他的寄存器定义和使用请读者参考附带的程序。

操作详解：

模式寄存器决定了 CAN 控制器的工作模式，所以在应用中对于该寄存器的设置显得非常重要。由于该寄存器是可以读写的，所以完全可以实现不同功能位的分时操作。在操作这些功能位时，应该注意 LOM 和 STM 位只有在 RM 位为 1 时才能被写入。为了方便操作，可以将该寄存器用一个数据结构——联合的方式来定义。下面的例子说明了定义该寄存器并对该寄存器进行读写操作的过程。

1. 用数据类型对该寄存器进行描述，以便更直观、更方便的操作。因为该寄存器可以同时操作，也可以分时操作，所以可以用联合的方式来描述该寄存器的内容。这样便可以实现同时操作该寄存器的 32 个位，或只操作其中的某个功能位。如程序清单 1.1。

程序清单 1.1 CAN 寄存器的定义

```
//CAN 模式寄存器数据类型定义
typedef union _canmod_
{
    UINT32 Word;
    struct {
        UINT32 RM_BIT :1;      //定义 RM 位
        UINT32 LOM_BIT :1;     //定义 LOM 位
        UINT32 STM_BIT :1;     //定义 STM 位
        UINT32 TPM_BIT :1;     //定义 TPM 位
        UINT32 SM_BIT :1;      //定义 SM 位
        UINT32 RPM_BIT :1;     //定义 RPM 位
        UINT32 RSV_BIT1 :1;    //保留位
        UINT32 TM_BIT :1;      //定义 TM 位
        UINT32 RSV_BIT24 :24;   //保留位
    }Bits;
}uCANMod,*P_uCANMod;
```

2. 定义该寄存器在微处理器 VPB 空间的基地址。

```
//定义模式寄存器在 ARM 中 VPB 的基地址
#define CANMOD_BADR 0xE0044000
```

3. 定义 1 个统一的对所有 CAN 模块该功能寄存器进行操作的方法。

```
//定义所有 CAN 模块模式寄存器的数据类型
#define CANMOD(CanNum) (*(volatile P_uCANMod) (CANMOD_BADR+CanNum* CAN_OFFSET_ADR))
```

程序清单 1.2 描述了对 CAN 模块 4 模式寄存器中 LOM 位置位的操作。

程序清单 1.2 寄存器的操作

```
CANMOD(CAN4).Bits.RM_BIT = 1;    //在写操作 LOM 位之前，RM 必须为 1
CANMOD(CAN4).Bits.LOM_BIT = 1;    //CAN 控制器操作
```


1.6.1 错误处理

CAN 控制器根据 CAN 2.0B 规范来对发送和接收错误进行计数、处理。每当检测到一个错误，发送或接收错误计数器将加 1；当错误被释放后，发送和接收错误计数器减 1。如果发送错误计数器的值为 255 时出现一个错误，CAN 控制器强制进入总线离线状态。此状态下，以下寄存器位被置位：CANSR 的位 BS、CANIR 的位 BEI 和位 EI（如果使能）和 CANMOD 的位 RM。RM 将许多 CAN 控制器功能复位和禁止。此时发送错误计数器设置成 127，接收错误计数器清零。软件必须在下一步清零 RM 位。这样，发送错误计数器将递减计数总线释放条件（11 个连续的隐性位）的第 128 个事件。软件可通过读取 Tx 错误计数器对计数器递减计数的情况进行监测。当递减计数结束后，CAN 控制器清除 CANSR 的位 BS 和 ES，并置位 CANICR 的位 EI（如果 IER 的 EIE 为 1）。

如果 CANMOD 的 RM 位为 1，Tx 和 Rx 错误计数器可被写入。向 Tx 错误计数器写入 255 将使 CAN 控制器强制进入总线离线状态。如果总线离线（CANSR 的 BS 位）位为 1，向 Tx 错误计数器写入 0~254 中的任何一个值都可清除总线离线。当通过软件清除 CANMOD 的 RM 后，只需要一个总线释放条件（11 个连续的隐性位）就可恢复操作。

重要提示：

在一般情况下，只有当总线发生严重故障的情况下才有可能发生总线脱离。这些现象有 TX、RX 与收发器之间的连接断路、收发器的 CAN_H、CAN_L 之间短路等。所以 CAN 总线上的错误,仅仅当发生总线离线时才有必要去处理，而其他的错误可以不用处理。如果不是硬件复位，而是软件将 RM 清零，在总线离线清除后，这时 CANSR 寄存器的 TCS（发送成功）位仍然为 0 来表示上次发送不成功。除非再次启动发送，数据发送成功后，TCS 位才会标志为 1。

程序清单 1.3 是该错误处理的操作示例。

程序清单 1.3 错误处理的操作

```
// 方法 1，等待 CAN 控制器清除总线离线
UINT32 i;
.....
if(0 != CANGSR(CAN1).Bits.BS)           // 总线脱离
{
    CANMOD(CAN1).Bits.RM = 0;           // 清除 RM 位
    for(i=0;i<2000000;i++)               // 如果总线故障没解决，超时退出
    {
        if(CANGSR(CAN1).Bits.TXERR == 0) // 清除总线脱离
        {
            break;
        }
    }
}
.....
// 方法 2，快速清除总线离线
if(0 != CANGSR(CAN1).Bits.BS)           // 总线脱离
{
    CANGSR(CAN1).Bits.TXERR = 0;         // 手动清零错误计数器
    CANMOD(CAN1).Bits.RM = 0;           // 清除 RM 位
}
.....
```

1.6.2 睡眠模式

如果 CAN 模式寄存器的 SM 位为 1 且无中断等待和 CAN 总线活动，CAN 控制器将进入睡眠模式。当 CAN 模式寄存器的 RM 为 0 时，软件只能置位 SM；也可通过软件来置位 CAN 中断使能寄存器的 WUIE 位以使能唤醒中断。

CAN 控制器唤醒（并置位 CAN 中断寄存器的 WUI 位，只要 CAN 中断使能寄存器的 WUIE 位为 1）来响应 a) CAN 总线出现一个显性位，或 b) CAN 模式寄存器的 SM 位通过软件清零。为了响应总线活动而被唤醒的休眠 CAN 控制器并不能接收初始信息，直到它检测到总线释放条件（11 个连续的隐性位）。如果在软件置位 SM 时中断挂起或 CAN 总线激活，则 CAN 控制器立刻被唤醒。

关于睡眠模式的注意事项：
在 LPC2119/2129 芯片错误报告（版本 V1.1，发布日期为 04 年 6 月 1 日）中，“A”版 LPC2119/2129 芯片的 CAN 模块在测试中发现睡眠模式存在以下问题。

CAN.2 No wake up from CAN sleep mode using SM bit

Introduction: The CAN Controller will enter sleep mode if the SM bit in the CAN Mode register is 1, no CAN interrupt is pending, and there is no activity on the CAN bus. The CAN Controller wakes up (and sets WUI in the CAN Interrupt register if WUIE in the CAN Interrupt Enable register is 1), in response to a dominant bit on the CAN bus or software clearing the SM bit in the CAN Mode register.

Problem: Clearing the SM bit does not cause the CAN module to wakeup from CAN sleep mode.

work-around: None, the SM bit cannot be used as a source of CAN wakeup.

解释：
处于睡眠模式的 CAN 模块不能够通过软件清除 SM 位（CAN 模式寄存器）来唤醒这个 CAN 模块，但可以由外部 CAN 报文唤醒。因此，在需要主动发送 CAN 报文的应用场合，不可以置位 SM 位，导致这个 CAN 模块进入睡眠模式。

1.6.3 中断

每个 CAN 控制器可产生 3 种中断请求：接收、发送和“其它状态”。发送中断是 3 个 Tx 缓冲区发送中断“或”的结果。每个控制器的各个接收和发送中断请求在向量中断控制器（VIC）中分配有不同的通路并拥有各自的中断服务程序。所有 CAN 控制器的“其它状态”和验收过滤器 LUTerr 条件相或到一个 VIC 通道。根据“LPC2119/2129/2194/2292/2294 USER MANUAL”中关于 CAN 中断通道的描述如表 1.23。

表 1.23 CAN 中断通道的描述（原文）

Block	Flag(s)	VIC 通道号
CAN	CAN and Acceptance Filter (1 ORed CAN, LUTerr int)	19
	CAN1 Tx	20
	CAN2 Tx	21
	CAN3 Tx (仅支持 LPC2194/2294 芯片)	22
	CAN4 Tx (仅支持 LPC2194/2294 芯片)	23
	CAN1 Rx	26
	CAN2 Rx	27
	CAN3 Tx (仅支持 LPC2194/2294 芯片)	28
	CAN4 Tx (仅支持 LPC2194/2294 芯片)	29

经过在 EasyARM2119 开发板上测试，各中断正常。
测试程序如程序清单 1.4。

程序清单 1.4 中断测试（初始化部分）

```
VICVectAddr8 = (INT32U)CANIntPrg;    //中断服务程序 CANIntPrg
VICVectCntl8 = (0x20|26);            //26 通道中断为向量 IRQ
VICIntEnable |= (1<<(26+ CanNum));    //使能 26 通道中断
CANIER(CanNum).Word= 0x01;           //只开放接收中断
```

1.6.4 发送优先级

如果 CANMOD 寄存器的 TPM 位为 0，多个使能的 Tx 缓冲区根据各自的 CAN 标识符（TID）来竞争信息发送权。如果 TPM 为 1，则它们根据各自的 CANTFS 寄存器位 7:0 的 PRIO 字段值来竞争。两种情况下都是最小的二进制值拥有优先权。如果 2（或 3）个发送使能的缓冲区具有同样的最小值，则先发送编号小的缓冲区的数据。CAN 控制器先对多个使能的 Tx 缓冲区进行动态选择，再发送信息。

1.7 组合 CAN 寄存器

3 个只读寄存器将 CAN 控制器的状态寄存器的所有位组合在一起便于共同访问。如果器件定义了更多或更少的 CAN 控制器，寄存器中有效字节的位数也随之改变。下列寄存器中每个定义的字节的 LS 位都是对应于一个 CAN 控制寄存器的特定状态位。

1.7.1 集中发送状态寄存器（CANTxSR – 0xE004 0000）

集中发送状态寄存器具体功能见表 1.24。

表 1.24 CAN 集中发送状态寄存器（CANTxSR – 0xE004 0000）

CANTxSR	名称	功能	复位值
3:0	TS4:1	1: CAN 控制器正在发送信息（同于 CANGSR 的 TS） TS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
11:8	TBS4:1	1: 3 个 Tx 缓冲区均可供 CPU 使用（同于 CANGSR 的 TBS） TBS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	全为 1
15:12	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
19:16	TCS4:1	1: 所有请求发送都已成功处理（同于 CANGSR 的 TCS） TCS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	全为 1
31:20	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

1.7.2 集中接收状态寄存器（CANRxSR – 0xE004 0004）

集中接收状态寄存器具体功能见表 1.25。

表 1.25 CAN 集中接收状态寄存器（CANRxSR – 0xE004 0004）

CANRxSR	名称	功能	复位值
3:0	RS4:1	1: CAN 控制器正在接收数据（同于 CANGSR 的 RS） RS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
11:8	RBS4:1	1: CAN 控制器接收到可用的信息（同于 CANGSR 的 RBS） RBS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
15:12	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
19:16	DOS4:1	1: 由于前面传输到 CAN 控制器的数据未被尽快读出而引起信息丢失（同于 CANGSR 的 DOS） DOS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
31:20	保留	保留。用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

1.7.3 集中其他状态寄存器（CANMSR – 0xE004 0008）

集中其他状态寄存器具体功能见表 1.26。

表 1.26 CAN 集中其他状态寄存器（CANMSR – 0xE004 0008）

CANMSR	名称	功能	复位值
3:0	ES4:1	1: Tx 和 Rx 错误计数器两者或其中之一的计数值达到 EWL 寄存器设定的阈值（同于 CANGSR 的 ES） ES4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
7:4	保留	保留。用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
11:8	BS4:1	1: CAN 控制器正在处理总线活动（同于 CANGSR 的 BS） BS4:3 仅用于 LPC2294。它们在其它器件中是保留位。	0
31:12	保留	保留。用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

1.8 全局验收过滤器

该模块为所有 CAN 控制器提供了接收标识符的查询功能(CAN 术语称之为验收过滤)。它包含一个 512×32bit (2k 字节) 的 RAM, 通过软件处理, 可在 RAM 中存放 1~5 个标识符表格。整个 RAM 可容纳 1024 个标准标识符或 512 个扩展标识符或两种类型混合的标识符。

如果在应用中使用的是标准标识符 (11 位), 则 3 个表格中至少有一个必定不为空。如果“fullCAN 模式”选项使能, 第 1 个表格存放标准标识符, 用于 fullCAN 模式中接收的处理。第 2 个表格存放单个标准标识符, 第 3 个表格存放标准标识符的范围, 使信息通过 CAN 控制器接收。FullCAN 的单个标准标识符表格必须按升序排列, 每半字为 1 个标识符, 每个字包含 2 个标识符。由于每个 CAN 总线都有自身的地址映射, 因此表格的每个条目都必须包含使用的 CAN 控制器的编号 (001—110)。FullCAN 下单个标准帧标识符表格单元结构如图 1.1 所示。

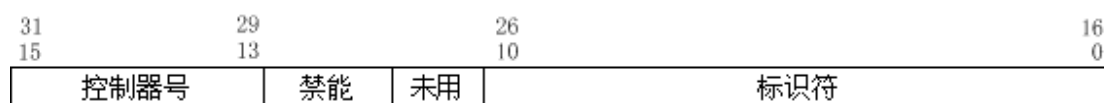


图 1.1 单个标准标识符表单元结构

标识符范围表格包含标识符的一对上下边界 (边界值包含在内), 一对范围占用一个字。按升序排列, 其结构如图 1.2 所示。

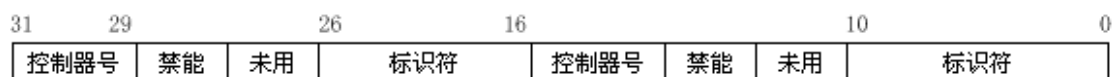


图 1.2 标准标识符范围表格单元结构

标准标识符单元的禁能位提供了一种开/关特定 CAN 标识符或范围标识符响应的方法。当验收过滤器功能被使能, 验收过滤器 RAM 中只有禁能位可通过软件来改变。通过向 RAM 中相应的字内写入 32 位 0 来使能对标准地址范围的响应, 并且, 通过写入 32 位 1 (0xFFFF FFFF) 来关闭该响应。在整个过程中只有禁能位被改变。

如果在应用中使用的是扩展标识符 (29 位), 验收过滤器 RAM 其它 2 个表格中至少有 1 个必定不为空, 两个表格一个用来存放单个扩展标识符, 一个用来存放扩展标识符的范围。单个扩展标识符表格必须按升序排列。图 1.3 为单个扩展标识符表格单元结构。

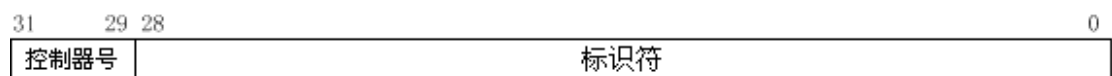


图 1.3 扩展标识符表格单元结构

扩展标识符范围表格必须含有偶数个行, 其格式与单个扩展标识符表格相同。和单个扩展标识符的表格一样, 扩展范围必须按升序排列。表中的第 1 和第 2 行 (第 3 和第 4 行...) 形成一对, 作为包括边界在内的扩展地址范围。这样, 包含在地址范围内 (含边界) 的地址就一定能接收到。要求编写的软件必须保证表格由一对一对的字组成。

使用 fullCAN 方法来接收扩展标识符信息实现起来并不容易。

有 5 个地址寄存器用来指向验收过滤器 RAM 中的表格: fullCAN 标准地址、标准单个地址、标准地址范围、扩展单个地址和扩展地址范围。这些表格在存储器中必须是连续的。后 4 个表格的起始地址分别是各自相邻的前一个表格的终止地址。扩展地址范围表格的终止地址在表格终止寄存器中给出。如果一个表格的起始地址等于下一个表格的起始地址或表格终止寄存器的值, 则该表格为空。

当 CAN 控制器的接收端已接收到一个完整的标识符，它将通知验收过滤器。验收过滤器响应这个信号，并读出控制器编号、标识符尺寸，以及来自控制器本身的标识符。然后，验收过滤器搜索 RAM 中的表格，以决定接收或放弃这一帧信息。

如果 fullCAN 模式使能且 CAN 控制器告知当前信息包含一个标准标识符，则验收过滤器首先查询标识符表格，以便接收可在 fullCAN 模式下处理。另外，如果 AF 未在 fullCAN 表格中发现合适的匹配，它将接着查询单个标识符表格，以获取 CAN 控制器给出的标识符长度。只要发现相等的匹配，AF 就通知 CAN 控制器保存信息，并提供一个 ID 索引值使之保存到接收帧状态寄存器（CANRFS）中。

如果验收过滤器未在单个标识符表格中找到相等的匹配，它便接着查询标识符范围表格，以获取 CAN 控制器给出的标识符长度。如果 AF 在表格的范围内发现了合适的匹配，它就通知 CAN 控制器保存信息，并提供一个 ID 索引值使之保存到接收帧状态寄存器（CANRFS）中。如果在单个标识符表格或范围表格中，AF 都未找到与接收到的标识符长度相匹配的值，AF 便通知 CAN 控制器丢弃/忽略接收到的信息。

1.8.1 验收过滤器寄存器

验收过滤寄存器的功能如表 1.27。

表 1.27 验收过滤器模式寄存器（AFMR – 0xE003 C000）

AFMR	名称	功能	复位值
0	AccOff	1: 如果 AccBP 为 0，验收过滤器不工作。所有 CAN 总线上的 Rx 信息均被忽略。	1
1	AccBP	1: 所有 Rx 信息被使能的 CAN 控制器接收。软件必须先置位该位，再修改以下各个寄存器的内容和查询表格 RAM 的内容（但不能置位或清零标准标识符行的禁能位）。如果该位和 AccOff 都为 0，验收过滤器就将接收到的 CAN 标识符屏蔽。	0
2	eFCAN	1: 由验收过滤器自身来处理所选 CAN 总线上指定标准 ID 值的信息的接收和保存。见“FullCAN 模式”。 0: 通过软件从接收 CAN 控制器中读出所有使能 CAN 总线上的 ID 被使能的全部信息。	0

1.8.2 标准帧单个起始地址寄存器（SFF_sa – 0xE003 C004）

标准帧单个起始地址寄存器功能如表 1.28。

表 1.28 标准帧起始地址寄存器（SFF_sa – 0xE003 C004）

SFF_sa	名称	功能	复位值
10:2		AF 查询 RAM 中单个标准标识符表格的起始地址。如果表格为空，该寄存器和下面的 SFF_GRP_sa 寄存器必须写入相同的值。为了兼容未来的器件，该寄存器的位 31:11 和 1:0 应当写入 0。如果 AFMR 的 eFCAN 位为 1，SFF_sa 的值就指明了标准 ID 表格的规格，这也是验收过滤器将要查询的值，一旦发现该值，验收过滤器自动将接收信息存放到验收过滤器 RAM 中。	0

1.8.3 标准帧组起始地址寄存器 (SFF_GRP_sa – 0xE003 C008)

标准帧组起始地址寄存器功能如表 1.29。

表 1.29 标准帧组起始地址寄存器 (SFF_GRP_sa – 0xE003 C008)

SFF_GRP_sa	名称	功能	复位值
11:2		AF 查询 RAM 中标准标识符组表格的起始地址。如果表格为空，该寄存器和下面的 EFF_sa 寄存器必须写入相同的值。当只有标准单个标识符被使用并且 AF 查询表格 RAM 的最后一个字（地址 0x7FC）被使用时，该寄存器允许写入的最大值是 0x800。为了兼容未来的器件，该寄存器的位 31:12 和 1:0 应当写入 0。	0

1.8.4 扩展帧起始地址寄存器 (EFF_sa – 0xE003 C00C)

扩展帧起始地址寄存器功能如表 1.30。

表 1.30 扩展帧起始地址寄存器 EFF_sa – 0xE003 C00C)

EFF_sa	名称	功能	复位值
10:2		AF 查询 RAM 中单个扩展标准标识符表格的起始地址。如果表格为空，该寄存器和下面的 EFF_GRP_sa 寄存器必须写入相同的值。当扩展表格为空并且 AF 查询表格 RAM 的最后一个字（地址 0x7FC）被使用时，该寄存器允许写入的最大值是 0x800。为了兼容未来的器件，该寄存器的位 31:11 和 1:0 应当写入 0。	0

1.8.5 扩展帧组起始地址寄存器 (EFF_GRP_sa – 0xE003 C010)

扩展帧组起始地址寄存器功能如表 1.31。

表 1.31 扩展帧组起始地址寄存器 (EFF_GRP_sa – 0xE003 C010)

EFF_GRP_sa	名称	功能	复位值
11:2		AF 查询 RAM 中扩展标识符组表格的起始地址。如果表格为空，该寄存器和下面的 ENDofTable 寄存器必须写入相同的值。当该表格为空并且 AF 查询表格 RAM 的最后一个字（地址 0x7FC）被使用时，该寄存器允许写入的最大值是 0x800。为了兼容未来的器件，该寄存器的位 31:12 和 1:0 应当写入 0。	0

1.8.6 AF 表格终止寄存器 (ENDofTable – 0xE003 C014)

AF 表格终止寄存器功能如表 1.32。

表 1.32 AF 表格终止寄存器 (ENDofTable – 0xE003 C014)

ENDofTable	名称	功能	复位值
11:2		<p>最后一个有效 AF 表格中最后有效的地址的上个地址。为了兼容未来的器件，该寄存器的位 31:12 和 1:0 应当写入 0。</p> <p>如果 AFMR 的 eFCAN 位为 0，该寄存器允许写入的最大值是 0x800，使得 AF 查询表格 RAM 的最后一个字（地址 0x7FC）可被使用。</p> <p>如果 AFMR 的 eFCAN 位为 1，该值是用来标注收滤波器 RAM 区的起始位置，在该区域内，验收过滤器将自动接收所选 CAN 总线上被指定 ID 的信息。这种情况下，该寄存器被写入的最大值为 0x800 – (6 × SFF_sa 的值)。使该地址和验收过滤器 RAM 终止地址之间允许保存 12 个字节的信息，每个标准 ID 被指定在验收过滤器 RAM 起始地址和下一个有效 AF 表格之间。</p>	0

1.8.7 LUT 错误地址寄存器（LUTerrAd – 0xE003 C018）

LUT 错误地址寄存器功能如表 1.33。

表 1.33 LUT 错误地址寄存器（LUTerrAd – 0xE003 C018）

LUTerrAd	名称	功能	复位值
10:2		如果 LUT 错误位（下面）为 1，该只读字段包含一个 AF 查询表格 RAM 的地址，在该地址处验收过滤器遇到表格内容错误。	0

1.8.8 LUT 错误寄存器（LUTerr – 0xE003 C01C）

LUT 错误寄存器功能如表 1.34。

表 1.34 LUT 错误寄存器（LUTerr – 0xE003 C01C）

LUTerr	名称	功能	复位值
0		当验收过滤器遇到 AF RAM 表格内容错误时该只读位置位。该位通过软件读取 LUTerrAd 寄存器清零。验收过滤器 LUTerr 条件与 CAN 控制器的“其它 CAN”中断相或来产生一个 VIC 通道的中断。	0

1.8.9 验收过滤器表格和 ID 索引值举例

假设 5 个验收过滤器地址寄存器包含的值在下表 1.35 的第 3 列中给出。表格的第 4 和第 5 列分别是字和行的编号（十进制表示）。如果 CAN 信息的标识符与表中行的编号相匹配，CANRFS 的 ID 索引字段将返回表格最右边一列的值。

表 1.35 验收过滤器表格和 ID 索引值举例

表格	寄存器	值	# 字	# 行	ID 索引
标准单个标识符	SFF_sa	0x040	810	1610	0—1510
标准标识符组	SFF_GRP_sa	0x060	410	410	16—1910
扩展单个标识符	EFF_sa	0x070	3610	3610	20—5510
扩展标识符组	EFF_GRP_sa	0x100	410	210	56—5710
	ENDofTable	0x110			

图 1.4 是地址寄存器、表格分布和 ID 索引值的详细图例。它列出了：

- 标准单个标识符表格，位于验收过滤器 RAM 起始位置，包含 26 个标识符；
- 标准标识符组表格，包含 12 组标识符；
- 扩展单个标识符表格，包含 3 个标识符；
- 扩展标识符组表格，包含 2 组标识符。

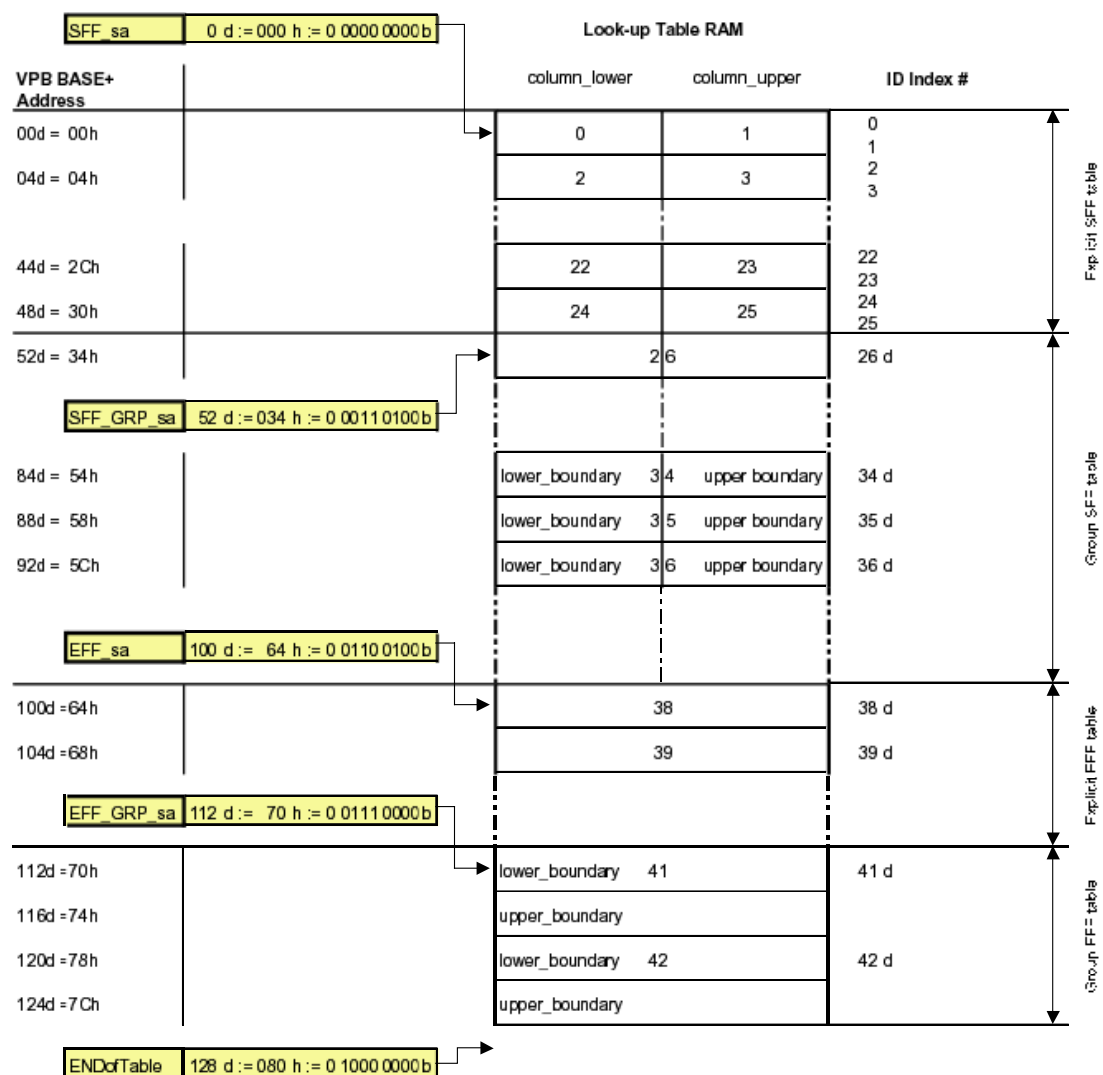


图 1.4 验收过滤器表格和 ID 索引值详细举例

1.8.10 FullCAN 模式

当 FullCAN 模式使能时，验收过滤器以“FullCAN”控制器的身份来处理所选 CAN 总线上指定标准帧 ID 值的信息的接收和保存。为了置位 eFCAN 位和使用 FullCAN 模式，验收过滤器 RAM 和指针的内容必须满足其它 2 个条件：

1. 标准帧单个起始地址寄存器 (SFF_sa) 的内容必须大于或等于 ID 值, 以便自动执行接收保存操作, SFF_sa 一般是 2 的倍数。如果必要的话, SFF_sa 还可为 4 的倍数。
2. EndOfTable 寄存器必须小于或等于 $0x800 - 6 \times \text{SFF_sa}$, 允许每个 ID 保存 12 字节的信息, 以便自动执行接收保存操作。

当上述条件都满足且 eFCAN 被置位时:

- 验收过滤器 RAM 和 SFF_sa 地址之间的区域用作单个标准 ID 表格和 CAN 控制器/总线的识别，这部分区域按升序排列，格式与单个标准 ID 表格相同（见图 1.2）。它的行和其它标准表格一样可以标识为“禁能”。如果存在奇数个“FullCAN”ID。则表格中至少有一行被标识为“禁能”。
- 第一个 $(SFF_sa) / 2$ 索引值分配给这些自动存储 ID。这也就是说，当 eFCAN 为 0 时，ID 不能再在 FullCAN 模式下处理时，Rx 帧状态寄存器的索引值可在原值的基础上增加 $(SFF_sa) / 2$ 。

- 当接收到标准 ID 时，验收过滤器在标准单个标识符和标识符组表格前查询相应的表格。
- 当在表格中查询到接收信息的控制器编号和 ID 值，验收过滤器将信息从 CAN 控制器中读出并存放到验收过滤器 RAM 中（起始地址为（EndOfTable）+它的 ID 索引*12）。

信息的格式见表 1.36。

表 1.36 自动保存 Rx 信息的格式

地址	31							24	23					16	15					10		8	7						0
0	FF	RTR						0000	SEM					0000	DLC					00000									
+4								Rx Data 4						Rx Data 3						Rx Data 2									Rx Data 1
+8								Rx Data 8						Rx Data 7						Rx Data 6									Rx Data 5

表中 FF, RTR 和 DLC 字段的描述见表 1.13。当信息更新时硬件将 SEM 字段设定为 01，信息更新结束后 SEM 字段被设定为 11。对信息进行访问时，软件应当清零 SEM 字段。软件必须以特定的方式来访问一条信息中的三个字以确保它们都来自同一条接收信息。图 1.5 所示为软件是如何使用 SEM 字段来确保访问的字来自同一信息的。

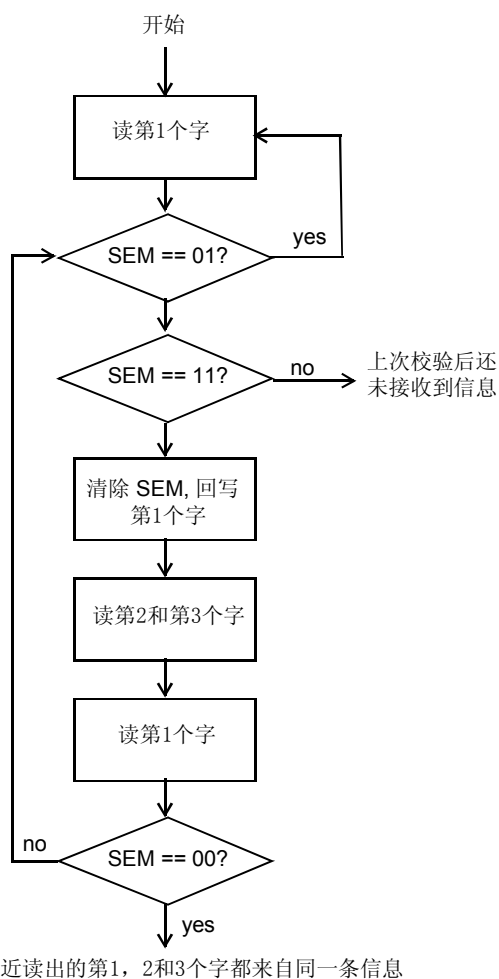


图 1.5 FullCAN 自动存储帧的读取

关于 FullCAN 验收过滤功能的问题:

在 LPC2119/2129 芯片错误报告（版本 V1.1，发布日期为 04 年 6 月 1 日）中，“A” 版 LPC2119/2129 芯片的 CAN 模块在测试中发现 FullCAN 验收过滤器存在以下问题。

CAN.3 FullCAN Mode Inoperative

Introduction: In FullCAN mode CAN messages are automatically compared to the acceptance filter look-up table and, if the message ID is found, automatically stored.

Problem: A conflict exists such that the ARM core and the CAN peripheral state machine may try to access the message ID look-up table at the same time. In this situation the semaphore bits in the message may be corrupted and the application code may fail. Alternatively the application may continue to run yet CAN messages will be missed.

work-around: None, do not use FullCAN mode.

解释：由于 ARM 内核与 CAN 外围状态机存在冲突，在 FullCAN 模块大数据量过滤接收 CAN 报文时，FullCAN 模块会将已过滤的报文转存到查询表(AF RAM)的起始 3 个字节空间。

因此，在出现错误的“A”版芯片中，不可以使用 FullCAN 模块，以避免发生此类情况。除 FullCAN 验收过滤功能外，ARM-CAN 的其他验收过滤功能可以正常工作。

“CANstarter-II开发套件”中有一份文档详细说明了LPC2000芯片的验收过滤功能，请查看《LPC2000系列32位ARM微控制器的CAN接收过滤设置》。

第2章 CAN 控制器驱动程序的编写

为了快速利用内嵌 CAN 控制器的 ARM7 微处理器 LPC2xxx 来设计你自己的 CAN 产品，我们提供一个 LPC2xxx 中 CAN 控制器的驱动程序。驱动程序的源代码以及如何利用该驱动程序的示例，可以使你快速而方便对这些源代码进行修改，以适合特别的需要。但是在进行驱动程序开始之前，我们假设用户已经对 PHILIPS 的 ARM7 微处理器有相当的了解，并对基于 ARM7 微处理器的开发流程有一定的基础。

本驱动采用中断方式完成，如果在不需要中断的场合，稍加修改即可成为查询方式，使您的应用更加方便。如果你要立即使用该驱动，也不想修改驱动的结构，可以参考我们的示例程序：一个简单的 RS232 数据与 CAN 数据的透明转换器。

接下来将对该驱动程序的结构和源代码进行详细的介绍。

2.1 驱动程序的结构

对于微处理器来说，CAN 控制器完全是基于事件触发的，即 CAN 控制器会在本身状态发生改变时，会将状态变化的结果告诉微处理器。所以微处理器处理 CAN 控制器时，可以采用中断的方式，也可以采用轮询查看 CAN 控制器状态的方式来对 CAN 控制器作出相应的处理。图 2.1 是一个 CAN 控制器的状态图，通过该状态图我们可以很容易的理解这种基于事件触发的驱动程序的结构。

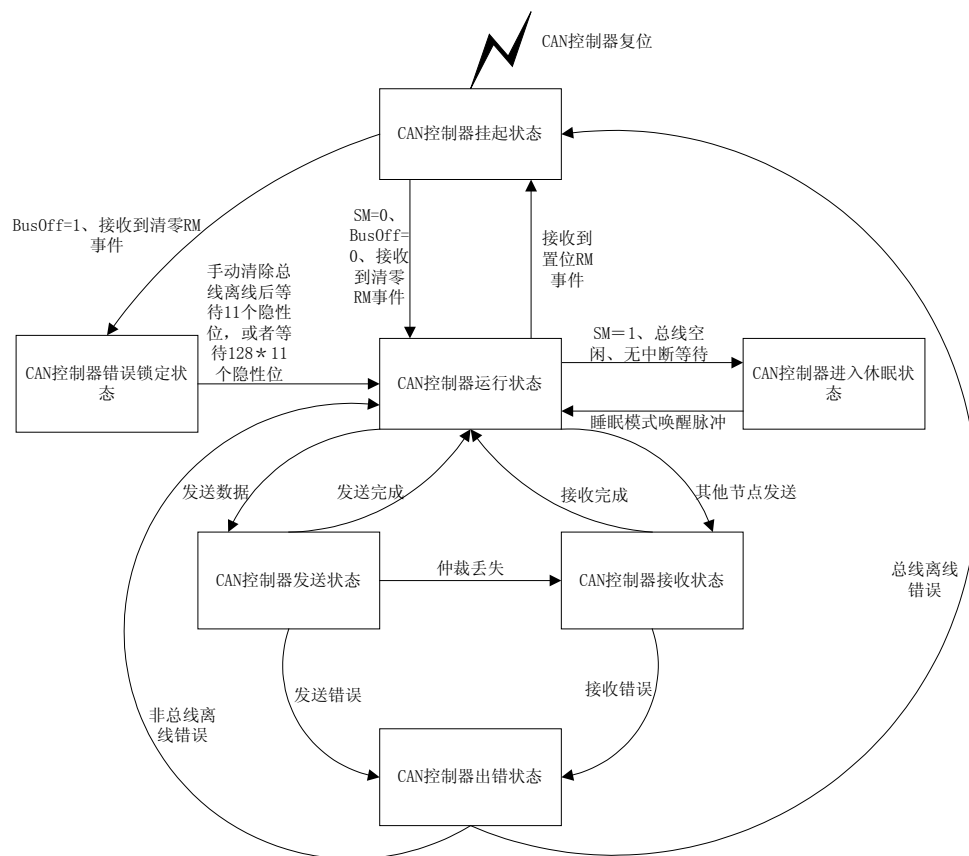


图 2.1 CAN 控制器状态图

事实上，微处理器对于 CAN 控制器的操作是非常简单的，但是为了更明白的说明问题，我们还是来看一下整个 CAN 数据通讯的流程。图 2.2 描述了 CAN 节点结构，通过该图可以看到驱动程序所在的位置以及所具有的功能。

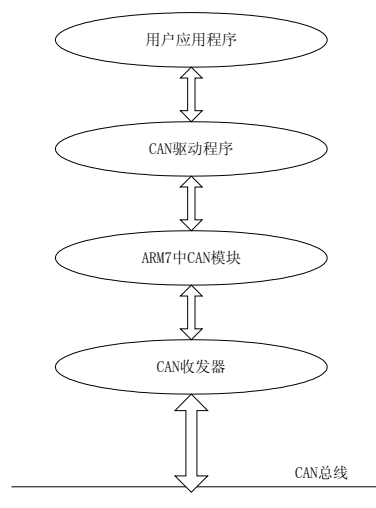


图 2.2 CAN 固件编程层次结构图

对于用户程序来说,不需要关心驱动代码是如何对 CAN 模块进行操作的。用户程序只需要通过调用驱动程序提供的接口,来实现数据的发送和接收即可。但是由于微处理器中,CAN 模块提供了非常灵活的操作方式,所以要实现一个统一的兼顾一切的方法是不可能的。为此在驱动程序中专门留出了可以由用户自主配置的选项,以更好的适应不同的应用领域。

根据 CAN 控制器的特性、微处理器的特性我们将 CAN 的驱动程序分为 3 个层次:

- 硬件抽象层: 将 CAN 控制器的硬件特性用数据类型进行抽象,并提供 CAN 控制器硬件操作的接口。在该层有 2 个文件 CANREG.H 和 CANIORAW.H,在这 2 个文件中分别定义了 CAN 控制器寄存器的数据类型和对寄存器的操作方法。
- 功能函数层: CAN 控制器各种功能的实现函数,该层的函数利用硬件抽象层中对寄存器操作的接口来访问 CAN 控制器来实现各种 CAN 控制器所能提供的功能。在该层中包含 2 个文件 CANFUNC.C 和 CANFUNC.H。
- 应用程序接口层: 在该层中的主要提供给用户 3 个函数,初始化 CAN 控制器、发送数据、接收数据。该层的文件有 CANAPP.H 和 CANAPP.C。

由于在用户实际的应用中,真正关心的是应用程序接口层部分,所以在这里也重点的介绍接口层。

对于用户来讲向 CAN 总线发送数据具有主动权,而接收数据是被动的,所以在此使用一个环形缓冲区来保存 CAN 接收的数据。对于发送数据,在驱动程序里不作干涉而是将发送数据的权力完全交给用户处理,当然为了方便用户,仍然提供给用户一个发送数据的函数。对于发送函数怎样调用,这完全由用户来处理。而对于接收数据则驱动程序直接将 CAN 的数据放到接收缓冲区里,同时提供给用户 1 个应用接收数据的函数。这样整个驱动便非常的明晰和简单。因为底层的用户可以不用关心,只是调用这 3 个简单的接口函数即可。程序清单 2.1 描述了 CAN 接收环形缓冲区的数据结构,图 2.3 描述了用户应用 CAN 驱动程序的简单过程。

程序清单 2.1 stcRcvCANCyBuf

```

/*****
**                                     **
**          驱动程序占用用户 RAM——环形缓冲区          **
**                                     **
*****/
typedef struct _RcvCANDataCycleBuf_
{

```

```

UINT32      WritePoint1   :8;          (1)
UINT32      WritePoint2   :8;          (2)
UINT32      WritePoint3   :8;          (3)
UINT32      WritePoint4   :8;          (4)

UINT32      ReadPoint1    :8;          (5)
UINT32      ReadPoint2    :8;          (6)
UINT32      ReadPoint3    :8;          (7)
UINT32      ReadPoint4    :8;          (8)

UINT32      FullFlag1     :8;          (9)
UINT32      FullFlag2     :8;          (10)
UINT32      FullFlag3     :8;          (11)
UINT32      FullFlag4     :8;          (12)

stcRxBUF  RcvBuf[CAN_MAX_NUM][USE_CAN_RCV_BUF_SIZE];
}stcRcvCANCyBuf,*P_stcRcvCANCyBuf;
  
```

注：stcRcvCANCyBuf(1)~(4) 环型缓冲区写指针。

stcRcvCANCyBuf(5)~(8) 环型缓冲区读指针。

stcRcvCANCyBuf(9)~(12) 环型缓冲区满标志。当写指针追上读指针此时表示环形缓冲区已满，此时相应的满标志置位。

以上的指针、标志的操作不需要用户干预。用户仅仅只需调用接口函数即可。

stcRcvCANCyBuf(13) 环型缓冲区定义。

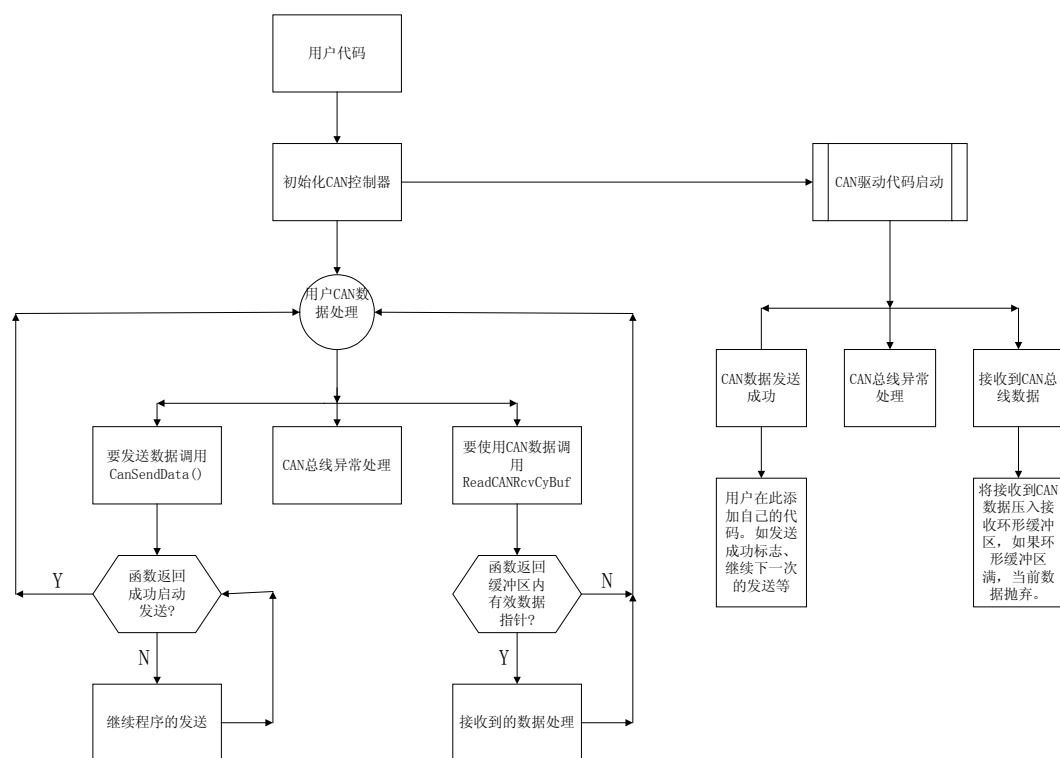


图 2.3 DriverApp

2.2 驱动程序使用方法

对于本驱动程序的使用,我们尽可能的考虑到如何让用户用最小的改动来适应不同的应用范围。因此在驱动程序中除了上面所说的与 CAN 控制器直接相关的文件外,还额外的提供一个用户配置文件 Custom.H,其中有一些选项需要用户根据自己的需要进行配置。另外还有与编译器有关的数据类型转换头文件 DATATYPETRS.H 和整合在一起的引用 CAN 驱动代码文件 IncludeCAN.H 文件。在 DATATYPETRS.H 中重新定义了许多与编译器有关的数据类型,如: int、unsigned char 等数据类型,如果用户编译器的这些数据类型的范围不同,请根据实际情况修改。在 IncludeCAN 头文件中将 CAN 驱动代码中所有的头文件集合在了一起,这样用户在自己的程序中,只需引用一个 IncludeCAN.H 文件即可。

但是对于用户来说,真正关心的是如何利用 CAN 控制器来与 CAN 网络上的其它节点进行数据通讯。而正常的通讯无非是向总线上发送数据和从总线上接收数据。但是在正常通讯之前必须要对 CAN 控制器进行初始化,以便让 CAN 控制器的工作方式、总线波特率、验收过滤器等各个功能与实际的工作相符。因此本驱动程序在使用时应该按下面的步骤:

1. 首先检查自己编译器数据类型是否与 DATATYPETRS.H 中定义的数据类型相符合,如果不符合请修改 DataTypeTrs.H,以使数据类型与编译器一致。
2. 根据自己的需要,修改 Custom.H 文件中的各项定义,使 CAN 控制器工作在自己需要的工作模式、波特率、验收过滤方式、中断使能等条件下。
3. 由于 CAN 控制器的中断分为 CAN 发送、CAN 接收、CAN 异常 3 大类,如果在具有 4 个 CAN 控制器的微处理器中将这中断独立出来则会占用 9 个中断资源。所以在本驱动中将 CAN 的中断全部统一在一起,构成一个微处理器非向量方式的中断。如果用户有更好的中断结构请根据实际的需要修改 CANAPP.C 文件中的 CAN 中断处理函数。
4. 在使用 CAN 控制器发送、接收数据之前,请先调用 CANAPP.C 文件中的 InitCAN 函数。
5. 如果要发送数据请调用 CANSendData 函数。
6. 如果要使用 CAN 接收到的数据请调用 ReadCANRcvCyBuf 函数。
7. 默认情况下,对于异常的处理本驱动已经有了默认处理,但如果用户要特别处理还请修改相关的异常处理函数。
8. 在使用 CAN 驱动的文件中引用“IncludeCAN.H”,并将 CANAPP.C, CANFUNC.C 文件添加到自己的工程里面。
9. 在编译器中指定 CAN 驱动文件的路径,与自己的项目文件一起进行编译即可。
10. 在调用了初始化 CAN 控制器函数后,用户不得关闭 CAN 中断。

更快捷的方法是用用户直接参考驱动程序应用示例的程序结构,只要在 Castom.H 中修改定义一些相关的参数即可。

2.3 驱动程序各文件介绍

2.3.1 用户配置文件

用户在应用本驱动程序之前,应该首先查阅配置文件中的某些配置选项是否符合自己特别的需要。程序清单 2.2 描述了用户需要配置哪些选项,以及应该怎样配置这些选项。

程序清单 2.2 Custom.H

```

/*****
**                                     **
**          用户配置文件                **
**                                     **
*****/
#endif  _CUSTOM_H_

```

```

#define    _CUSTOM_H_
//当 VPB 时钟为 11059200Hz 时，常用波特率与总线时序器对照表,如果 VPB 时钟不等，请自己计算出总线
//时序器的值
#define    BPS_250K                0x0017C003           //CAN 的波特率 250Kbps 时总线时序器的值
//全局应用定义
#define    CAN_MAX_NUM              4                      (1)
#define    CAN_OFFSET_ADR           0x4000                //CAN 各模块寄存器之间的线性差异 (2)
#define    USE_CAN_RCV_BUF_SIZE     0x10                  (3)
//各路 CAN 独立应用定义
//工作模式定义，正常方式=0；只听模式=1；
#define    USE_MODE_CAN1            0x00                  (4)
#define    USE_MODE_CAN2            0x00                  (5)
#define    USE_MODE_CAN3            0x00                  (6)
#define    USE_MODE_CAN4            0x00                  (7)
//错误报警寄存器的值
#define    USE_EWL_CAN1             0x60                  (8)
#define    USE_EWL_CAN2             0x60                  (9)
#define    USE_EWL_CAN3             0x60                  (10)
#define    USE_EWL_CAN4             0x60                  (11)
//中断使能定义
#define    USE_INT_CAN1             0x7FF                 (12)
#define    USE_INT_CAN2             0x7FF                 (13)
#define    USE_INT_CAN3             0x7FF                 (14)
#define    USE_INT_CAN4             0x7FF                 (15)
//发送优先级模式定义
#define    USE_TPM_CAN1             0x01                  (16)
#define    USE_TPM_CAN2             0x01                  (17)
#define    USE_TPM_CAN3             0x01                  (18)
#define    USE_TPM_CAN4             0x01                  (19)
//总线波特率定义
#define    USE_BTR_CAN1             BPS_250K             (20)
#define    USE_BTR_CAN2             BPS_250K             (21)
#define    USE_BTR_CAN3             BPS_250K             (22)
#define    USE_BTR_CAN4             BPS_250K             (23)
#endif

```

各项说明：

- CASTOM.H(1)~(3) 用来定义驱动程序所有 CAN 模块的应用定义。
- CASTOM.H(1) 用来定义微处理器中所包含的 CAN 模块的数目。如果微处理器有 2 个 CAN 模块则此处应定义为 2，如果微处理器有 4 个 CAN 模块则此处应定义为 4。
- CASTOM.H(2) CAN_OFFSET_ADR，该值不需要修改。
- CASTOM.H(3) 该处用来定义驱动程序接收环形缓冲区的大小。对于 CAN 总线数据的接收由驱动程序自动接收到该环形缓冲区。用户如果要使用接收的 CAN 数据，直接访问该缓冲区即可。
- CASTOM.H(4)~(23) 用来定义驱动程序中独立的各个 CAN 模块的应用定义。

- CASTOM.H(4)~(7)该处定义 CAN 控制器工作时的工作方式,如果使用正常工作模式则 USE_CAN_MODE 的值定义为 0。如果使用只听工作模式则 USE_CAN_MODE 的值定义为 1。
- CASTOM.H(8)~(11)该处定义错误报警的界限值,当 CAN 控制器的错误计数器达到 USE_EWL_VAL 时,如果报警中断使能则会产生报警中断信号,同时错误状态寄存器中的 ES 置位。
- CASTOM.H(12) ~ (15)该处用来设置用户使用 CAN 的中断值,默认情况下 CAN 中断全部使能。
- CASTOM.H(16)~(19)该处用来设置 CAN 控制器中 3 个发送缓冲区发送数据时的发送优先级。当 USE_TPM_MODE 定义为 0 时 3 个发送缓冲区的发送优先级按帧 ID 的优先级;当 USE_TPM_MODE 定义为 1 时 3 个发送缓冲区的发送优先级按 TPIO 的优先级。
- CASTOM.H(20)~(23)定义总线时序寄存器的值,按照上面的介绍计算出相应的波特率的值。

2.3.2 CAN 驱动应用接口层文件

应用接口层包括 CANAPP.H 和 CANAPP.C 两个文件。在文件中主要是定义了用户可以直接使用或修改的初始化 CAN 控制器函数 InitCAN、发送数据函数 CANSendData、应用接收数据函数 ReadCANRcvCyBuf 以及中断处理函数 CANIntPrg。

2.3.2.1 初始化 CAN 模块函数 InitCAN

CAN 控制器初始化函数主要用来实现 CAN 工作时的参数设置,如果 CAN 控制器不经过初始化是不能进行工作的。这些初始化的内容包括,硬件使能 CAN、设置 CAN 报警界限、设置总线波特率、设置中断工作方式、设置 CAN 验收过滤器的工作方式、设置 CAN 控制器的工作模式等。程序清单 2.3 和图 2.4 描述了初始化 CAN 控制器的过程。

程序清单 2.3 InitCAN

```

/*
*****
**函数原型      :    void InitCAN(eCANNUM CanNum)
**参数说明      :    CanNum  --> CAN 控制器, 值不能大于 CAN_MAX_NUM 规定的值
**返回值       :    无
**说 明        :    本函数用于初始化 CAN 控制器
*****/
void InitCAN(eCANNUM CanNum)
{
    HwEnCAN(CanNum);                                     (1)
    SoftRstCAN(CanNum);                                  (2)
    CANEWL(CanNum).Bits.EWL_BIT = USE_EWL_CAN[CanNum];  (3)
    //初始化波特率
    CANBTR(CanNum).Word = USE_BTR_CAN[CanNum];          (4)
    //初始化中断为非向量中断
    VICDefVectAddr =(UINT32)CANIntPrg;                   (5)
    VICIntEnable |= (1<<19)|(1<<(20+ CanNum))|(1<<(26+ CanNum)); (6)
    CANIER(CanNum).Word= USE_INT_CAN[CanNum];           (7)
    //配置验收滤波器(旁路状态)

```

```

CANAFMR.Bits.AccBP_BIT =1; (8)
//初始化模式
CANMOD(CanNum).Bits.TPM_BIT = USE_TPM_CAN[CanNum]; (9)
CANMOD(CanNum).Bits.LOM_BIT = USE_MOD_CAN[CanNum]; (10)
//初始化接收环形缓冲区
CANRcvBufApp.FullFlag1=CANRcvBufApp.FullFlag2=
CANRcvBufApp.FullFlag3=CANRcvBufApp.FullFlag4=0; (11)
CANRcvBufApp.ReadPoint1=CANRcvBufApp.ReadPoint2=
CANRcvBufApp.ReadPoint3=CANRcvBufApp.ReadPoint4=0; (12)
CANRcvBufApp.WritePoint1=CANRcvBufApp.WritePoint2=
CANRcvBufApp.WritePoint3=CANRcvBufApp.WritePoint4=0; (13)
//启动 CAN
SoftEnCAN(CanNum); (14)
}

```

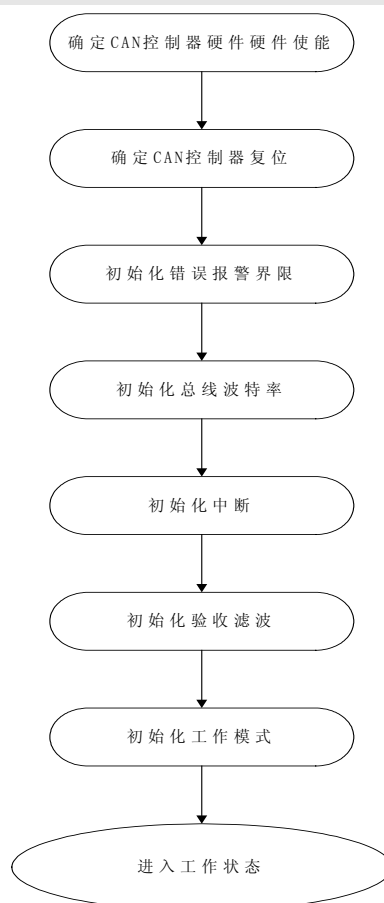


图 2.4 InitCAN

- InitCAN(1) 硬件使能 CAN 控制器，该函数在 CAN 功能层文件 CANFUNC.C 中定义。
- InitCAN(2) 软件复位 CAN 控制器，该函数在 CAN 功能层文件 CANFUNC.C 中定义。
- InitCAN(3) 初始化 CAN 控制器的错误报警界限，如果采用默认值为 0x60。
- InitCAN(4) 初始化 CAN 总线时序寄存器。
- InitCAN(5)~(7)进行 CAN 中断处理的初始化操作。

- InitCAN(8)设置模验收过滤器（设置为旁路状态）。
- InitCAN(9)~(10)配置工作模式。
- InitCAN(11)~(13)初始化接收环形缓冲区
- InitCAN(14)退出初始化前要使 CAN 进入工作状态。向 CAN 总线发送数据函数 CANSendData

在用户将本地数据经过打包成符合 CAN 发送帧格式的数据后，可以调用本函数进行数据的发送。程序清单 2.4 和图 2.5 描述了 CAN 发送数据函数。

程序清单 2.4 CANSendData

```

/*****
**函数原型   :   UINT32      CANSendData(eCANNUM CanNum,UINT32 Cmd,P_stcTxBUF Buf)
**参数说明   :   CanNum    < CAN_MAX_NUM
**           Cmd      发送命令字
**           Buf      要发送的一帧 CAN 数据
**返回值    :   =0; 成功写入缓冲区并成功启动发送; ！=0, 写发送缓冲区失败。
**说 明     :   CAN 发送数据函数,该函数的返回值只能表示数据是否已写入 CAN 发送缓冲区,并启动发送,但是否发送成功,并不知道.要想明确的知道是否发送成功请在调用该函数后
                查询 TCS 状态位, 或配合发送成功中断来判断。
*****/
UINT32      CANSendData(eCANNUM CanNum,UINT32 Cmd,P_stcTxBUF Buf)
{
    UINT32 i,status=0;
    if(0 != CANSR(CanNum).Bits.TBS1_BIT)                                (1)
    {
        i=SEND_TX_BUF1;
    }
    else if(0 != CANSR(CanNum).Bits.TBS2_BIT)                            (2)
    {
        i=SEND_TX_BUF2;
    }
    else if(0 != CANSR(CanNum).Bits.TBS3_BIT)                            (3)
    {
        i=SEND_TX_BUF3;
    }
    else
    {
        i=0xFF;
    }
    status=WriteCanTxBuf(CanNum,i, USE_TPM_MODE, Buf);                    (4)
    if(status == 0)
    {
        if(CANMOD(CanNum).Bits.SM_BIT != 0)                             (5)
        {
            CanQuitSM(CanNum);
        }
    }
}

```

```

CanSendCmd(CanNum,Cmd,i);
}
return (status);
}

```

(6)

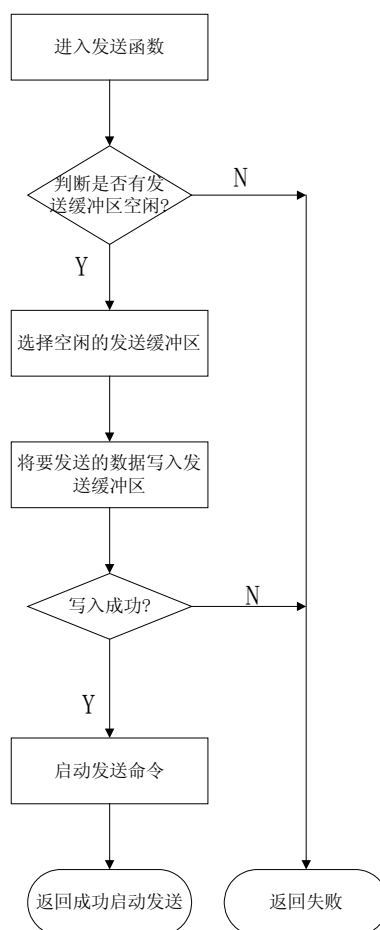


图 2.5 CanSendData

- CANSendData(1)~(3)判断是否有空闲的发送缓冲区 TxBUF。
- CANSendData(4) 将要发送的数据写入选定的发送缓冲区。
- CANSendData(5) 启动 CAN 数据发送。发送命令可以选择。

在使用发送函数时有一点必须注意，因为在启动发送数据的命令后，CAN 控制器要将缓冲区内的数据发送完毕后，才会将该帧数据是否发送成功的状态返回。这样如果在函数里一直等待数据发送完毕，会使整个微处理器的性能下降，所以为了避免这种情况，该函数在启动发送后便立即返回，如果用户想要得到成功发送的事件，请用户配合发送中断或利用查询 TCS 状态位的方法来处理。

2.3.2.2 使用 CAN 驱动接收到的 CAN 数据函数 ReadCANRcvCyBuf

用户可以在自己的程序中调用该函数，对从 CAN 总线上接收的数据进行特殊的处理。程序清单 2.5 和图 2.6 描述了用户如何使用由驱动接收的 CAN 总线数据。

程序清单 2.5 ReadCANRcvCyBuf

```

/*****
**函数原型   :   UINT32   ReadCANRcvCyBuf(eCANNUM CanNum, stcRxBUF *Buf)
**参数说明   :   CanNum       < CAN_MAX_NUM
                  Buf         将环形缓冲区中接收的 1 帧数据放到 Buf 里
**返回值     :   !=0       Buf 数据无效
                  = 0       Buf 数据有效
**说 明      :   该函数用来使用环形缓冲区中 1 帧有用的数据
*****/
UINT32   ReadCANRcvCyBuf(eCANNUM CanNum,stcRxBUF *Buf)
{
    UINT32   status=0;
    switch(CanNum)
    {
        case CAN1:
            if((0 != CANRcvBufApp.FullFlag1) ||
                (CANRcvBufApp.ReadPoint1 != CANRcvBufApp.WritePoint1))           (1)
            {
                *Buf=CANRcvBufApp.RcvBuf[CAN1][CANRcvBufApp.ReadPoint1];        (2)
                if(++CANRcvBufApp.ReadPoint1 >= USE_CAN_RCV_BUF_SIZE)            (3)
                {
                    CANRcvBufApp.ReadPoint1 =0;
                }
                CANRcvBufApp.FullFlag1=0;
            }
            else
            {
                status=1;                                                         (4)
            }
            break;
        case CAN2:
            if((0 != CANRcvBufApp.FullFlag2) ||
                (CANRcvBufApp.ReadPoint2 != CANRcvBufApp.WritePoint2))           (5)
            {
                *Buf=CANRcvBufApp.RcvBuf[CAN2][CANRcvBufApp.ReadPoint2];        (6)
                if(++CANRcvBufApp.ReadPoint2 >= USE_CAN_RCV_BUF_SIZE)            (7)
                {
                    CANRcvBufApp.ReadPoint2 =0;
                }
                CANRcvBufApp.FullFlag2=0;
            }
            else
            {
                status=1;                                                         (8)
            }
    }
}

```

```

    }
    break;
case CAN3:
    if((0 != CANRcvBufApp.FullFlag3) ||
        (CANRcvBufApp.ReadPoint3 != CANRcvBufApp.WritePoint3))          (9)
    {
        *Buf=CANRcvBufApp.RcvBuf[CAN3][CANRcvBufApp.ReadPoint3];        (10)
        if(++CANRcvBufApp.ReadPoint3 >= USE_CAN_RCV_BUF_SIZE)            (11)
        {
            CANRcvBufApp.ReadPoint3 =0;
        }
        CANRcvBufApp.FullFlag3=0;
    }
    else
    {
        status=1;                                                          (12)
    }
    break;
case CAN4:
    if((0 != CANRcvBufApp.FullFlag4) ||
        (CANRcvBufApp.ReadPoint4 != CANRcvBufApp.WritePoint4))          (13)
    {
        *Buf=CANRcvBufApp.RcvBuf[CAN4][CANRcvBufApp.ReadPoint4];        (14)
        if(++CANRcvBufApp.ReadPoint4 >= USE_CAN_RCV_BUF_SIZE)            (15)
        {
            CANRcvBufApp.ReadPoint4 =0;
        }
        CANRcvBufApp.FullFlag4=0;
    }
    else
    {
        status=1;                                                          (16)
    }
    break;
default:
    status=1;
    break;
}
return status;
}

```

- ReadCANRcvCyBuf(1)(5)(9)(13) 判断环形缓冲区是否有数据。
- ReadCANRcvCyBuf(2)(6)(10)(14) 返回有效的缓冲区数据。
- ReadCANRcvCyBuf(3)(7)(11)(15) 环形缓冲区读操作处理。
- ReadCANRcvCyBuf(4)(8)(12)(16) 标志环形缓冲区无可用的数据。

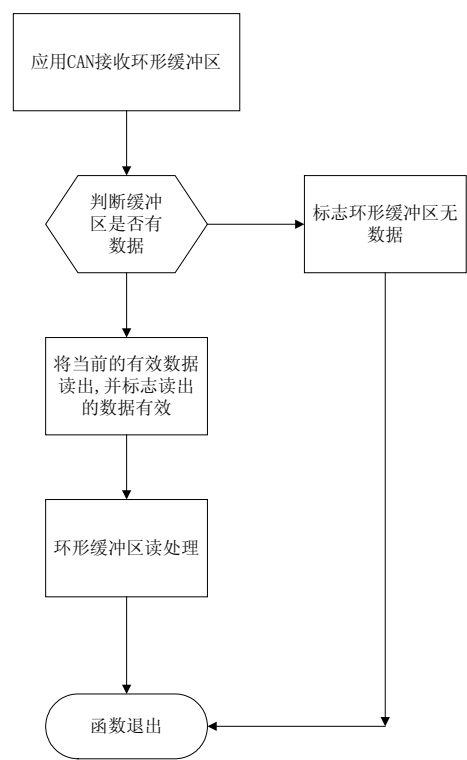


图 2.6 ReadCANRcvCyBuf 驱动程序中断处理过程

该程序主要是对 CAN 所有状态变化进行监控，但该程序的运行还需要用户在开始前进行中断的设置使能。如果中断不使能则该程序不会工作，整个驱动程序便不能使用。程序清单 2.6 和图 2.7 描述了 CAN 中断处理过程。

程序清单 2.6 CANIntPrg

```

/*****
**函数原型      :   void CANIntPrg(void)
**参数说明      :   无
**返回值        :   无
**说 明        :   CAN 中断处理函数，所有 CAN 处理均在中断中完成
*****/
void __irq      CANIntPrg(void)
{
    UINT32  j;
    uCANICR k;
    if(CANLUTerr.Word != 0 )
    {
        //可在此修改添加 LUT 错误处理代码
        j=CANLUTerrAd.Word;
    }
    for(j=0;j<CAN_MAX_NUM;j++)
    {
        k=CANICR(j);
        if(k.Bits.RI_BIT != 0)
    }
}

```

```

    {
        //可在此修改添加 CAN 接收数据处理代码
        WriteCANRcvCyBuf(j);
    }
    if(k.Bits.TI1_BIT != 0) (3)
    {
        //可在此修改添加 CANTxBUF1 发送完成数据处理代码
    }
    if(k.Bits.TI2_BIT != 0) (4)
    {
        //可在此修改添加 CANTxBUF1 发送完成数据处理代码
    }
    if(k.Bits.TI3_BIT != 0) (5)
    {
        //可在此修改添加 CANTxBUF1 发送完成数据处理代码
    }
    if(k.Bits.BEI_BIT != 0) (6)
    {
        //可在此修改添加总线错误处理代码
        CanBufOffLinePrg(j);
    }
    if(k.Bits.ALI_BIT != 0) (7)
    {
        //可在此修改添加仲裁丢失处理代码
    }
    if(k.Bits.EPI_BIT != 0) (8)
    {
        //可在此修改添加错误报警处理代码
    }
    if(k.Bits.WUI_BIT != 0) (9)
    {
        //可在此修改添加总线唤醒处理代码
    }
    if(k.Bits.DOI_BIT != 0) (10)
    {
        //可在此修改添加数据溢出处理代码
        ClrCanDataOver(j);
    }
}
VICVectAddr = 0;
}

```

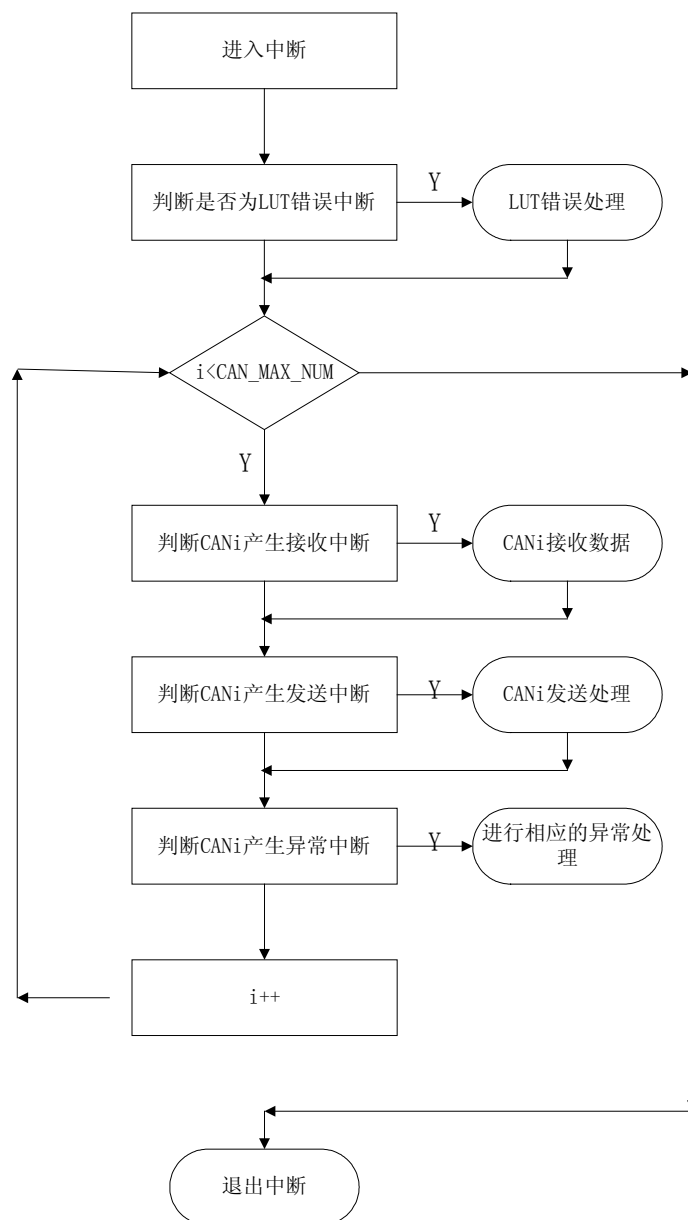



图 2.7 CANIntPrg

- CANIntPrg(1) CAN 模块 LUT 错误。用户可以添加或修改 LUT 错误处理代码。
- CANIntPrg(2) CAN 模块接收数据处理，如果用户不想使用默认的处理方式，可以添加自己的代码。
- CANIntPrg(3)~(5)对于发送完成处理，完全交给用户处理，用户可在此添加自己的代码。
- CANIntPrg(6)~(10)为 CAN 模块异常处理，必须要处理的为总线离线错误和数据溢出错误，其他的异常可以不用关心，但如果用户要有特别的用途，如用于诊断、调试等可以加入自己的程序代码。

2.3.3 功能函数层文件

在该层主要是定义了 CAN 控制器所应用到的功能函数。该层函数如无特别的必要，可以不用修改。这些函数包括：复位 CAN 控制器、使能 CAN 控制器、CAN 控制器发送命令、CAN 控制器写发送缓冲区、读 CAN 控制器接收缓冲区、CAN 控制器异常处理等。

这些函数包含在 CANFUNC.C 和 CANFUNC.H 这 2 个文件中。

为了更清楚的说明这些函数，将这些函数以表格的形式罗列出来供用户参考。其源代码见源程序。

2.3.2.3 复位 CAN 控制器功能函数

为了更好的控制 CAN 控制器的工作模式切换，复位 CAN 控制器分为硬件复位和软件复位。硬件复位 CAN 控制器与微处理器的硬件结构密切相关，应用该函数主要是为了在适当的场合禁止 CAN 控制器工作，例如为降低功耗。软件复位主要应用在 CAN 控制器软件应用过程中，修改一些只有在复位下才能修改的寄存器的值。

表 2.1 描述了 CAN 复位函数的结构。

表 2.1 CANRST

函数名称	函数原型	入口参数	出口参数	功能描述
硬件复位 CAN 控制器	Void HWRstCAN(eCANNUM CanNum)	CAN 通道号，值应 该小于 CAN_MAX_NUM	无	该函数用于从 硬件上禁止所 选择的 CAN 通道工作
软件复位 CAN 控制器	Void SoftRstCAN(eCANNUM CanNum)	CAN 通道号，值应 该小于 CAN_MAX_NUM	=0: 复位成 功 !=0: 复位 失败	通过置位 CAN 模式寄存器的 RM 位来，复 位 CAN

2.3.2.4 使能 CAN 控制器功能函数

CAN 控制器使能函数也分为硬件使能和软件使能。CAN 控制器使能函数应该与 CAN 复位函数配对使用。如果在程序中调用了硬件复位 CAN 控制器函数将 CAN 控制器禁止后，想重新使用 CAN 控制器则必须调用硬件使能 CAN 函数。如果要使处于软复位模式的 CAN 控制器进入工作模式，则必须调用 CAN 软件使能函数。

表 2.2 描述了 CAN 使能函数的结构。

表 2.2 CANEN

函数名称	函数原型	入口参数	出口参数	功能描述
硬件使能 CAN 控制器	Void HwEnCAN (eCANNUM CanNum)	CAN 通道号，值应 该小于 CAN_MAX_NUM	无	该函数用于将 掉电的 CAN 控 制器模块重新 上电
软件使能 CAN 控制器	UINT32 SoftRstCAN (eCANNUM CanNum)	CAN 通道号，值应 该小于 CAN_MAX_NUM	=0: 成功进入 工作模式 !=0: 进入工 作模式失败	通过清零 CAN 模式寄存器的 RM 位来使 CAN 进入工作 状态

2.3.2.5 CAN 控制器各类操作命令功能函数

微处理器对 CAN 控制器的一些特定操作是通过写 CAN 模块的命令寄存器来实现的，将这些命令归类后，有 3 种命令处理情况：发送命令、接收数据相关命令、异常处理命令。

发送命令包括发送缓冲区加发送方式，发送缓冲区和发送方式在前面有介绍，详见（CAN 控制器命令寄存器节）。接收数据相关的命令为释放接收缓冲区。异常处理命令为清除数据溢出状态。

表 2.3 描述了 CAN 的命令函数结构。

表 2.3 CANCMD

函数名称	函数原型	入口参数	出口参数	功能描述
发送命令函数	void CanSendCmd(eCANNUM CanNum,UINT32 Cmd,UINT32 TxBuf)	① CAN 通道号，值应该小于 CAN_MAX_NUM。 ② 命令字，发送命令方式。 ③ 发送缓冲区选择	无	该函数用于启动各类发送命令
释放接收缓冲区	RelCanRecBuf(CanNum)	CAN 通道号，值应该小于 CAN_MAX_NUM	无	这是一个宏定义函数用来实现接收缓冲区的释放
清除数据溢出	ClrCanDataOver(CanNum)	CAN 通道号，值应该小于 CAN_MAX_NUM	无	这是一个宏定义函数用来实现数据溢出状态的清除

2.3.2.6 CAN 控制器睡眠状态软件切换功能函数

在一些特别的应用场合用户可能会使 CAN 控制器进入睡眠状态，但是要使 CAN 控制器进入睡眠状态必须符合 2 个条件：没有中断、CAN 总线空闲。所以如果这时 CAN 总线在活动状态、或者正在有 CAN 控制器中断那么是不会成功进入到睡眠状态的。如果想从睡眠状态唤醒有 2 种方式：软件清零 SM 位、检测到总线释放条件。

表 2.4 描述了 CAN 睡眠状态切换的函数结构。

表 2.4 CANSN

函数名称	函数原型	入口参数	出口参数	功能描述
进入睡眠状态	UINT32 CanEntrySM(eCANNUM CanNum)	CAN 通道号，值应该小于 CAN_MAX_NUM	=0：成功 !=0：失败	函数用于使 CAN 进入睡眠模式
软件睡眠唤醒	UINT32 CanQuitSM(eCANNUM CanNum)	CAN 通道号，值应该小于 CAN_MAX_NUM	=0：成功 !=0：失败	用于将处于睡眠状态的 CAN 控制器唤醒

2.3.2.7 写 CAN 控制器发送缓冲区

将要发送到 CAN 总线上的数据经过封装处理后，使要发送的数据符合数据类型 stcTxBUF。然后调用该函数将数据写入 CAN 发送 BUF，便可以使用发送命令进行数据的发

送了。

表 2.5 描述了写 CAN 控制器发送缓冲区的函数结构

表 2.5 CANWRITEBUF

函数名称	函数原型	入口参数	出口参数	功能描述
写发送缓冲区	UINT32 WriteCanTxBuf(eCANNUM CanNum,UINT32 TxBufNum,UINT32 TPM,P_stcTxBUF Buf)	① CAN 通道号，值应 该小于 CAN_MAX_NUM ② 选择发送缓冲区 ③ 发送优先级模式 ④ 要发送的数据帧	=0，成功将 数据写入 TxBUF。 !=0，写缓 冲区失败。	该函数用于 将要发送的 1 帧数据写入 发送缓冲区

注：以上所用函数源代码，在源代码部分文件 CANFUNC.C 和 CANFUNC.H 中描述。

2.3.4 硬件抽象层文件

硬件抽象层文件主要包括各 CAN 控制器寄存器数据格式及读写访问方法的定义，在前面的章节中已经作了详细的定义，在这里不再重复详细的介绍。详尽的程序见源代码部分文件 CANREG.H 和 CANIORAW.H。

第3章 CAN 驱动程序应用实例——RS232 与 CAN-bus 透明转换器

在本实例开始之前,先把本实例的硬件平台、软件平台、应用注意事项、以及实例要实现的功能解释清楚。以使用户更好的了解和利用该实例,并通过该实例对 CAN 驱动的使用方法,找到适合自己特定应用的 CAN 驱动移植方法。

3.1 硬件平台

- ZLG LPC22xx ARM 开发板
- Philips ARM7 微处理器 LPC2294
- LPC22xx ARM 开发板 CAN 接口板
- EasyJTAG ARM 仿真器
- ZLG 任何一款 CAN 接口卡
- DC9V 电源
- RS232 标准 9 针延长接口线
- ZLG DB9-OPEN5 CAN-bus 连接线

详细的硬件信息请登陆周立功单片机网站 www.zlgmcu.com。

图 3.1 描述了该实例的硬件平台连接图,图中的图形不代表任何意义,一切应以实物为准。

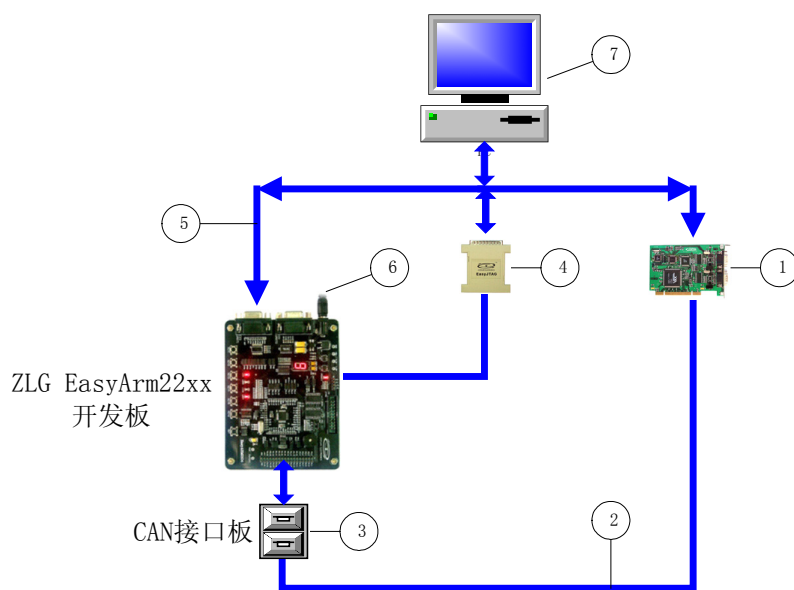


图 3.1 HwWorkSpace

①为 ZLG 任何接口的 CAN 接口卡,这些接口卡有:PCI 接口、ISA 接口、RS232 接口、USB 接口等。

②为 ZLG DB9-OPEN5 格式的 CAN 网络连接线,用于将③CAN 接口板的 CAN 接口与①的 CAN 接口相连构成 CAN 网络。

③EasyArm22xx 的 CAN 接口板,内有 CAN 收发器用来连接选择微处理器 LPC22xx 的 CAN 通道。

④EasyJTAG ARM7 仿真器,用来开发基于 ARM7 平台的应用程序仿真。

⑤RS232 9 针延长线,将 EasyArm22xx 的 RS232 接口与 PC 机的 RS232 接口连接起来。

⑥DC9V 电源,给开发板提供电源

⑦开发宿主机 PC 机。开发调试平台。

各类硬件的使用，请参考相关的硬件数据手册。

3.2 软件平台

- windows 操作系统
- ARM7 开发软件 ADS1.2
- 串口调试助手 ZLGCOMVIEW
- ZLGCAN 测试软件
- ZLGCAN 接口卡的驱动程序
- EasyJTAG 仿真器的驱动程序

在准备好硬件开发平台后，在 PC 机中应该最少具有以上 6 个软件。对于 ARM7 的开发工具可以从相关的网站下载、而串口调试工具、CAN 测试软件以及驱动程序可以到 www.zlgmcu.com 网站下载。各软件的使用方法见相关软件的使用手册，在此不作说明。

3.3 应用注意事项

由于 RS232 的数据格式与 CAN-bus 的数据格式不同，要实现 CAN 数据与 RS-232 数据的相互转换需要一定的通讯协议。为此我们在此制定了一个简单的通讯协议的数据格式。但是这个通讯协议仅仅是用来适合该实例的，并不能用于用户特定的应用。RS232 数据到 CAN 数据的转换，须遵循下面的数据格式：

- 帧起始（SOF）+CAN 通道号（CanNum）+CAN 帧信息（CFI）+CAN 报文 ID（CID）+要发送到 CAN 总线的数据（nDATA）+CAN 发送命令（CMD）+CRC 校验字节
- 该帧数据中所有的数据为 16 进制格式。
- SOF 为 1 字节表示要向 CAN 总线发送一帧数据的开始字节，SOF 恒等于 0x12。
- CanNum 为 1 字节表示 CAN 的通道号，CAN 的通道与选择的微处理器有关，但该值不能大于微处理器所含有的最大 CAN 数目。如：LPC2294 有 4 路 CAN，则 CanNum 的值应该小于 4。
- CMD 为 1 字节表示 CAN 向总线发送数据的发送方式。CanNum=1，表示单次发送；=2，表示自发自收；=3，表示单次自发自收；=其他值为正常发送。
- CFI 为 1 字节表示发送往 CAN 总线的数据信息。该字节的结构为：
 - bit7=1，表示该帧数据为扩展帧，=0 该帧数据为标准帧；
 - bit6=1，表示该帧为远程帧，=0 表示该帧为数据帧；
 - bit4~bit0 表示当前帧携带的数据长度为多少有效字节。
- CID 为 4 字节，帧 ID 的顺序按从左向右的顺序排列，如帧 ID 为 0x12345678，则 CID 的值为 12 34 56 78。这种格式同样适合标准帧，如标准帧的 ID 为 0x12，则 CID 的值为 00 00 00 12。
- nDATA 为 8 字节，数据的顺序为从左向右，如果不足 8 各数据则后面的数据补 0。如：帧信息中的数据长度为 6，要发送的数据为“0x11 0x22 0x33 0x44 0x55 0x66”，则 nDATA 的值为“0x11 0x22 0x33 0x44 0x55 0x66 0x00 0x00”。
- CRC 校验字节，用户可以自己设计 CRC 算法。在本应用中值恒定为 0。主要是为了方便用户调试。
- CAN 数据发送到 RS232 的数据格式为：

CAN: xx

FIF: xx

FID: xxxx

Dat: xx xx xx xx xx xx xx xx

- 以上的数据均为 ASCII 码格式。

其中 CAN 的值表示由哪路 CAN 接收到的数据。FIF 的值表示接收到的 1 帧数据的帧信息。FID 的表示接收到数据的报文 ID。Dat 的表示接收到的有效数据，如果该帧数据为 0，或者该帧数据为远程帧，则 Dat:后无数据。

3.4 实现的功能说明

- ① 当微处理器 LPC22xx 接收到串口的一帧数据后，会判断该帧数据是否符合上面描述的通讯协议。如果符合则程序对数据进行解包处理，然后打包成符合 CAN 发送数据格式的帧数据，并将该帧数据发送到通讯协议指定的 CAN 通道所在的 CAN 总线。
- ② 当微处理器 LPC22xx 接收到来自 CAN 网络的一帧数据时，会将该帧数据打包成 CAN 数据到 RS232 的数据格式，并将该数据发送到串口。
- ③ 该实例只实现了 Rs232 与 CAN 控制器 1 的互相通信，剩下的功能如 CAN 通道选择，发送命令选择和校验功能用户可参照实例自己完成。实例详见“ARM_CAN232”。

3.5 测试方法说明

- ① 测设 RS232 数据到 CAN-bus。在上位机中打开 ZLGCOMView 软件，并将该软件的接收窗口设为文本显示模式，发送格式设置为 HEX 格式发送。假设 PC 机的 COM1 口与 EasyArm22xx 的串口相连，并且 RS232 的通讯波特率为 57600bps。设置后图 3.2 所示。



图 3.2 ZLGCOMView-1

将符合 RS232 数据到 CAN 格式的一帧数据填入 ZLGCOMView 软件的发送字符串窗口，如图 3.3 所示。



图 3.3 ZLGCView-2

然后点击 ZLGCView 的“发送”按钮，则此时 PC 机会通过 COM 口将该帧数据发送出去。

打开 ZLGCANTest 软件，选择接口卡类型，假设这里选用的是 CAN232 接口卡，选择好与该卡连接的 PC 的 COM 口，COM 口的波特率、以及 CAN-bus 的通讯波特率。CAN-bus 的通讯波特率应该与 EasyArm22xx 开发板中 CAN 程序的设置一致。如果各方面设置都正确则在 ZLGCANTest 软件中会接收到该帧数据，如图 3.4 所示。

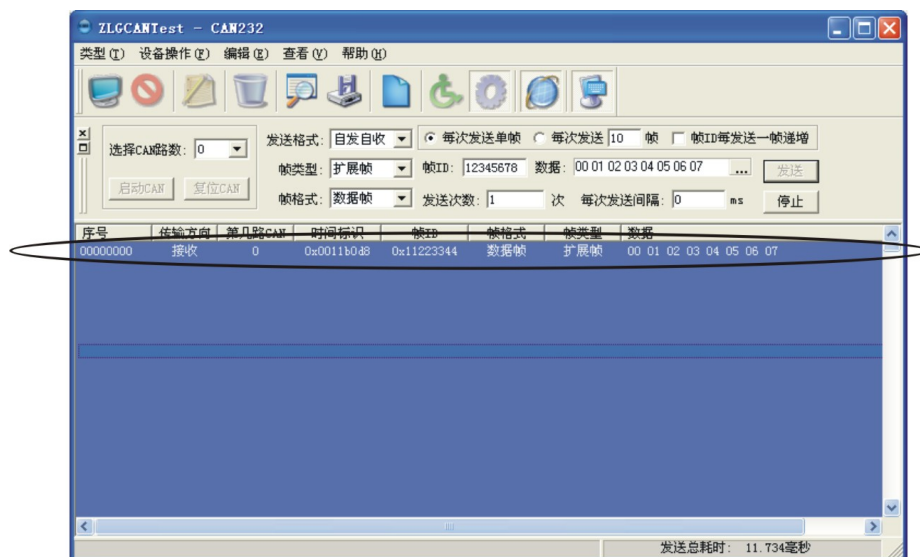


图 3.4 ZLGCANTest

- ② 测试 CAN 数据到 RS-232。同样在打开的 ZLGCANTest 软件中，填入要发送的数据、帧 ID、选择帧类型后启动发送，则在 ZLGCView 中应该会接收到该帧数据。如图 3.5 所示：

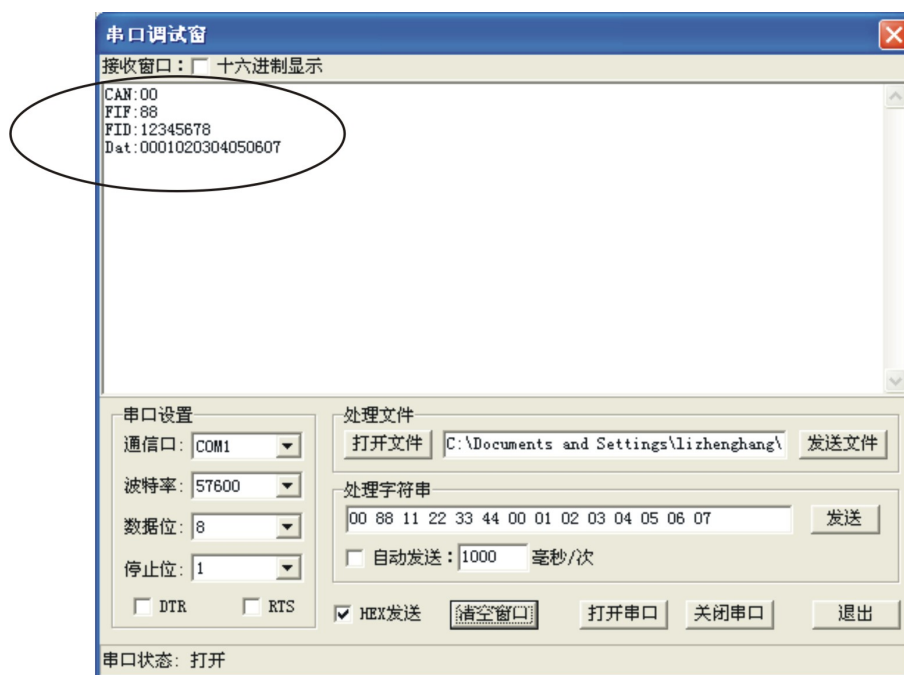


图 3.5 ZLGCView-3

当以上 2 步测试通过后, 便可认为该 RS232 与 CAN-buss 数据转换器的基本功能得到实现。下面详细的解释该实例实现的步骤。

3.6 简单的 RS232 数据与 CAN-bus 数据透明转换的实现

由于本文的主要内容是讲述 CAN 驱动代码的应用, 所以对于微处理器的 UART 部分不会详细的介绍, 关于 UART 部分的资料请参考 Philips LPC22xx 的数据手册。

图 3.6 是根据转换器功能特性描述出的 RS232 与 CAN 数据透明转换的功能流程图。从流程图也可以看出, CAN 驱动程序的使用非常简单只需要初始化 CAN、发送数据到 CAN、从 CAN 接收数据而已。

3.6.1 系统初始化

- 系统初始化中包括微处理器外设的初始化、UART 口初始化、CAN 模块初始化。微处理器的初始化主要包括堆栈初始化, PLL 锁相环初始化用来确定微处理器的系统时钟、VPB 时钟, 中断清零等。这些请参照 ZLG EasyArm22xx 的工程模板文件部分。
- UART0 口初始化包括波特率的设置、工作模式的设置等。在这里将波特率设置为 57600bps, 工作模式设置为 1 位起始、8 位数据、1 位停止、无奇偶校验。具体的设置过程参考微处理器使用手册。
- CAN 模块的初始化包括 CAN 模块的选择、CAN 工作模式的设定、CAN 波特率的选择、FullCAN 验收过滤器的使用、CAN 中断的使用等。这些设置全部可以在用户定义头文件 Custom.h 中修改定义。在这里我们选择 CAN 波特率 250Kbps、工作模式为正常模式、所有中断使能、不使用睡眠模式。

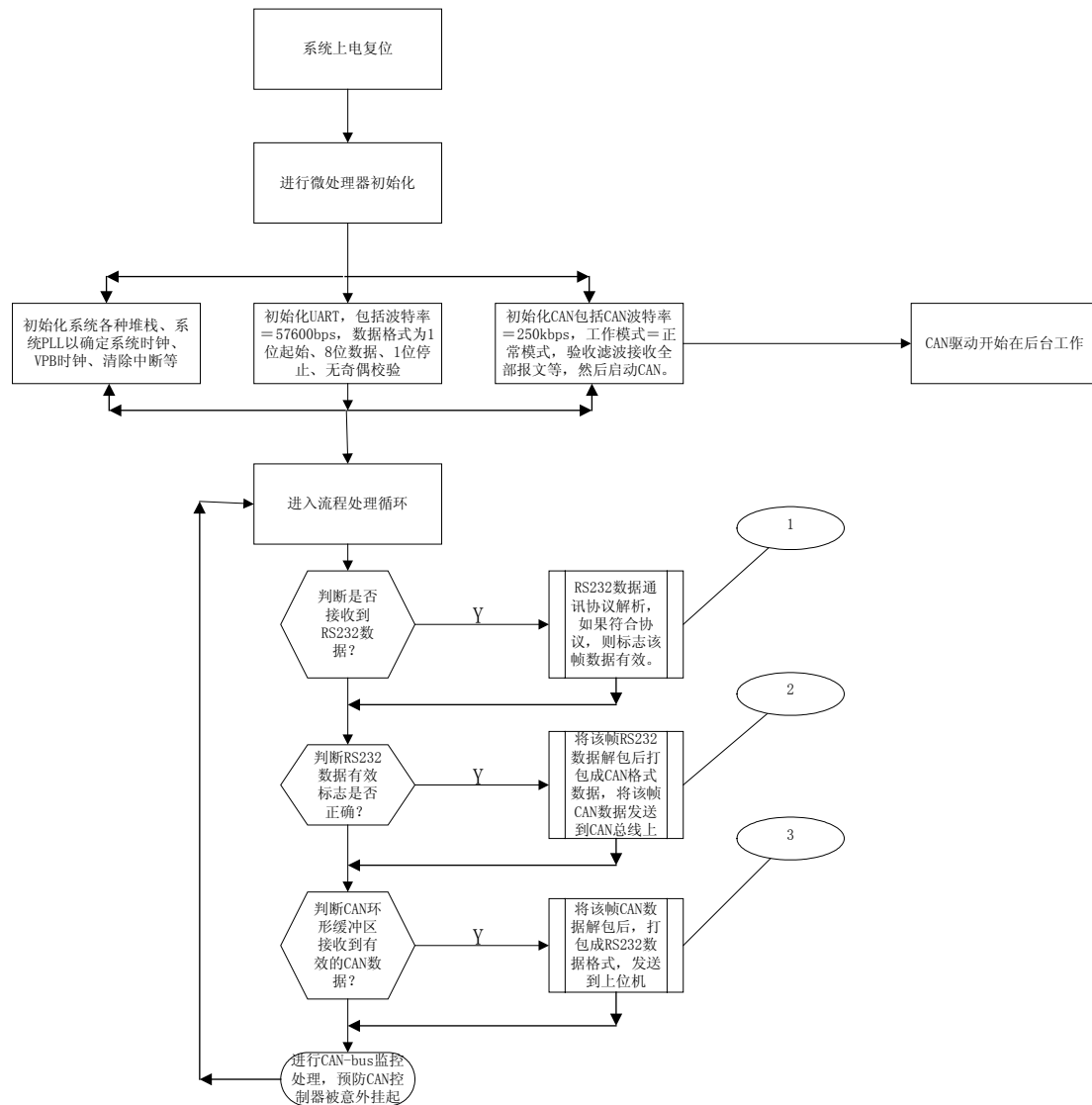


图 3.6 RS232-CAN

3.6.2 主循环处理

系统中 RS232 数据的接收处理采用查询的方式, 一旦检测从串口接收到 1 个 RS232 数据。便会进入 RS232 数据协议判断处理过程, 进行 1 帧数据的验证, 如果该帧数据符合协议, 则会置位 RS232 帧数据有效标志。其处理过程如图 3.7 所示。

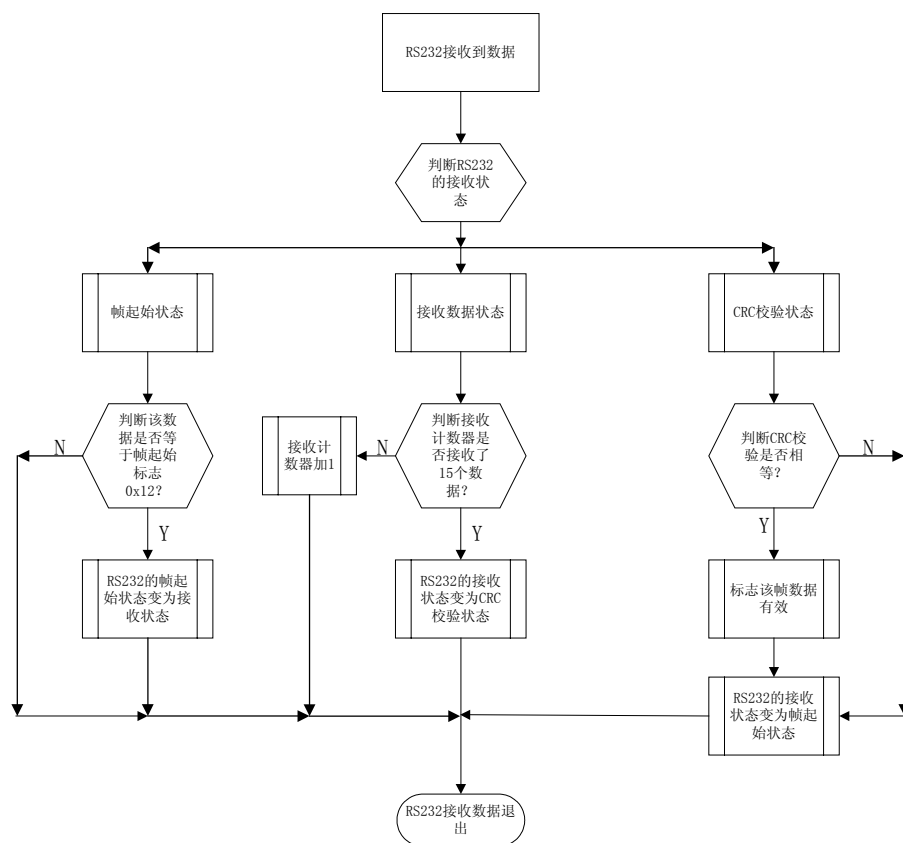


图 3.7 RS232-COM

当系统检测到 RS232 帧数据有效的标志后，便进入解析该帧 RS232 数据到 CAN-bus 的处理流程。该过程是 RS232 数据解包、CAN-bus 数据打包的过程。其过程如图 3.8 所示。

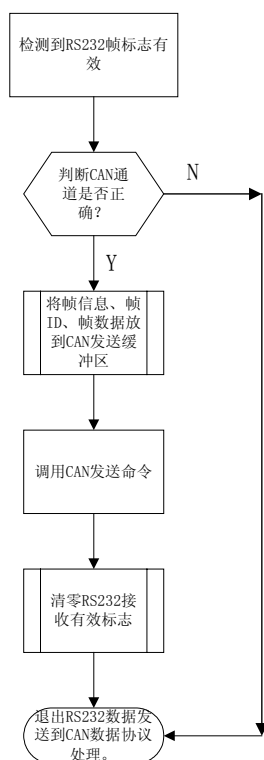


图 3.8 SEND-RS232-CAN

而对于 CAN 接收数据是通过在驱动程序中自动完成的，只需检查环形缓冲区即可。但是当环形缓冲区满了，而用户没有来得及处理这些数据时，后面来自 CAN 总线的数据会被抛弃。如果检测到环形缓冲区有数据，则进行 CAN 数据的解包，然后打包成 RS232 数据，并将该帧数据发往 UART 口。其工作流程如图 3.9 所示。

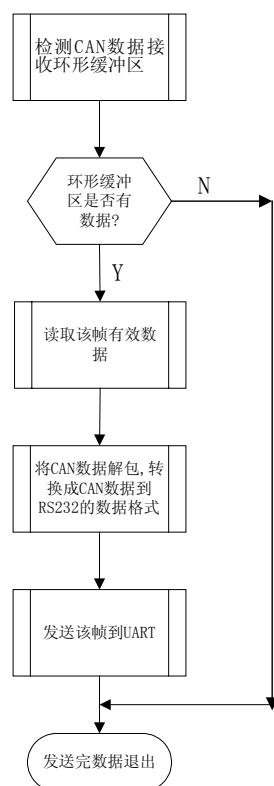


图 3.9 Send-CAN-RS232

有了上面实例架构的分析，便可以很轻松的由源代码中分析出如何使用 CAN 的驱动程序,从而举一反三可以将 CAN 驱动用到其他特定的场合。下面主要分析一下实例中用到的 CAN 驱动部分的源代码，其他部分的代码见实例项目文件。

程序清单 3.1 是 MAIN.c 的程序代码。

程序清单 3.1 MAIN.C

```

/*****
**                                CANDriver Demo :RS232—CAN 主程序 MAIN.C                                **
*****/

int  main(void)
{
    UINT32  i;
    TargetResetInit();
    InitUart0(57600);
    memcpy(STBuf,HelloArmCan,sizeof(HelloArmCan));
    Uart0Send(sizeof(HelloArmCan),STBuf);
    for(i=0;i<CAN_MAX_NUM;i++)
    {    //初始化 CAN
        InitCAN(i);
    }
}

```

(1)

```

while(1)
{
    //循环处理事件
    if((U0LSR & 0x01) !=0)
    {
        //串口接收处理数据
        UartComPol();
    }
    if(0 != CanSendFlag)
    {
        //RS232 数据转发 CAN
        if(0 != RS232DataToCan())
        {
            memcpy(STBuf,CanErr,sizeof(CanErr));
            Uart0Send(sizeof(CanErr),STBuf);
        }
        CanSendFlag=0;
    }
    for(i=0;i<4;i++)
    {
        if(ReadCANRcvCyBuf(i,&CRBuf) == 0)
        {
            // CAN 数据转发 RS232
            memcpy(STBuf,cCANChunl,sizeof(cCANChunl));
            CanHexToRs232ASCII(&STBuf[sizeof(cCANChunl)],i);
            STBuf[sizeof(cCANChunl)+2]=0x0D;STBuf[sizeof(cCANChunl)+3]=0x0A;
            TransCounter = sizeof(cCANChunl)+4;
            CanDataToRs232();
        }
    }
}
return 0;
}

```

- MAIN.C (1) CAN 控制器初始化函数 InitCAN，因为在本实例中应用了微处理器的全部 CAN 模块，所以可以进行适当的修改以适合特定的需要。
- MAIN.C (2) 在 RS232 数据到 CAN 的函数中调用了发送数据到 CAN 函数 CANSendData。
- MAIN.C (3) 利用驱动接收到的 CAN 数据，调用了函数 ReadCANRcvCyBuf。

所以对于用户程序来说如果不想修改大的程序结构便可利用 CAN 驱动，则也可以应用该主程序的结构，在上面添加自己的代码即可。如果要在自己的程序结构中使用 CAN 驱动，则可以在适当的地方灵活的利用这 3 个用户接口函数

第4章 参考文献

- [1] CAN Specification Version 2.0, Parts A and B, Philips Semiconductors, 1992
- [2] Philips Semiconductors, Data Sheet SJA 1000, April 1997
- [3] Philips Semiconductors, Data Sheet PCx82C200, November 1992
- [4] Philips Semiconductors, Data Handbook IC18, Semiconductors for In-car Electronics 1996
- [5] Philips Semiconductors, Data Handbook IC20, 80C51-Based 8-bit Microcontrollers 1997
- [6] Dietmayer, K.: CAN Bit Timing, Philips Semiconductors, Technical Report HAI/TR9708, 1997
- [7] Dietmayer, K.; Overberg, K. W.: CAN Bit Timing Requirements, SAE Paper 970295, 1997
- [8] Application Note PCA82C250/251 CAN Transceiver, AN96116, Philips Semiconductors, 1996
- [9] Philips LPC22xx Data Sheet