QXS320F2800137差异手册

V1.0.4

合肥乾芯科技有限公司

版本历史

版本号	日期	备注
1.0.0	2025.11.21	初稿
1.0.1	2025.12.08	更新ADC章节
1.0.2	2025.12.12	删除LIN章节
1.0.3	2025.12.29	更新I2C章节
1.0.4	2026.03.12	更新中断差异描述

目　录

1. 概述

本文档是QX 2800137和TI 2800137的差异手册与性能对比表，目的是方便移植TI C2000 DSP应用程序到QX DSP。

2. CORE

2.1. 指令集及生态

表 3-1 乾芯内核的TI内核特性对比表

对比维度	QXDSP	TI C2000 DSP
指令体系	32位固定长度；统一、规范，学习曲线平缓	16/32位混合长度，理解和调试复杂
性能并行性	3槽VLIW，运算/load/store可并行执行，吞吐率高	单槽流水线，简单可靠，但缺乏并行能力
代码体积	指令固定32位，程序体积相对偏大	代码密度高，片上Flash利用更高效
开发体验	RISC风格，编译器友好，用户代码可预测性强	CISC风格
应用适配性	适合通用DSP+控制，可扩展到无人机/工业/算法场景	针对电机、电源控制深度优化
生态支持	工具链现代化，生态仍在建设中	生态成熟，配套库和示例丰富

3. SYSCTL

3.1. Power Management

无差异。

3.2. Resets

无差异。

3.3. Peripheral Interrupts

1. 乾芯2800137的中断没有ACK机制，中断中不需要任何中断的ACK操作即可正常运行。为了增加代码的可移植性，乾芯的驱动中增加了空的Interrupt_clearACKGroup函数，位域中增加了无实际作用的PIECTRL[PIEACK]寄存器，调用Interrupt_clearACKGroup();或者对PIECTRL[PIEACK]进行写操作，均没有任何副作用。
2. 乾芯280049在允许嵌套的中断处理函数中EINT和DINT必须成对出现，在软件恢复现场之前必须有DINT调用来显式关闭中断响应，否则将导致系统异常跑飞（TI无此明确要求并允许嵌套的现场恢复操作之前无DINT调用）。

以下为嵌套操伪代码的规范操作样例：

```
__interrupt void XXXXX(void)
{
    软件保存现场      （工具链）
    返回PC压栈        （软件）
    嵌套优先级修改    （软件）
    EINT              （软件）
    用户程序          （软件）
    DINT              （软件）
    返回PC出栈        （软件）
    软件恢复现场      （工具链）
}
```

以下为实际代码样例：

```
227 __interrupt void TZ1IntHandler(void)
228 {
229     uint32_t TempLR;
230     uint32_t TempPICIER;
231     uint32_t TempPIEIER2;
232     uint32_t TempPIEIER3;
233     uint32_t TempPIEIER8;
234 
235     // 返回PC保存压栈
236     TempLR = ExpRegs.EPCR;
237     //
238     // 中断使能保存以及中断优先级修改
239     // 在下面打开对应允许嵌套此中断的中断
240     //
241     TempPICIER  = PieCtrlRegs.PICIER.all;
242     TempPIEIER2 = PieCtrlRegs.PIEIER2.all;
243     TempPIEIER3 = PieCtrlRegs.PIEIER3.all;
244     TempPIEIER8 = PieCtrlRegs.PIEIER8.all;
245 
246     PieCtrlRegs.PICIER.all  = PIC_GROUP8;
247     PieCtrlRegs.PIEIER2.all = 0x0;
248     PieCtrlRegs.PIEIER3.all = 0x0;
249     PieCtrlRegs.PIEIER8.all = 0x2;
250 
251     EINT;
252 
253     DEVICE_DELAY_US(2);
254 
255     DINT;
256     //
257     // 返回PC出栈恢复，中断使能恢复
258     //
259     ExpRegs.EPCR            = TempLR;
260     PieCtrlRegs.PICIER.all  = TempPICIER;
261     PieCtrlRegs.PIEIER2.all = TempPIEIER2;
262     PieCtrlRegs.PIEIER3.all = TempPIEIER3;
263     PieCtrlRegs.PIEIER8.all = TempPIEIER8;
264 }
```

3.4. Low-Power Modes

1.乾芯2800137的io低功耗唤醒,在LPMCR寄存器中增加gpiolpsel控制gpio低功耗唤醒的使能位（TI没有该状态位，可直接io唤醒），如需使用GPIO唤醒需配置该位为1。

3.5. ACCESS_PROTECTION

1. 乾芯2800137不支持TI的ACCESS_PROTECTION特性。

3.6. CLK_CFG

1. 乾芯2800137无CLKCFGLOCK1寄存器，对应功能不支持。
2. 乾芯2800137无SYSPLLSTS寄存器无REF_LOSTS位域，对应功能不支持。
3. 乾芯2800137无MCDCR寄存器无SYSREF_LOSTS、SYSREF_LOSTSCLR、SYSREF_LOST_MCD_EN位域，对应功能不支持。
4. 乾芯2800137无XTALCR2寄存器，对应功能不支持。
5. 乾芯2800137的GPIO驱动能力三挡可调（2.5mA、5mA、10mA），默认驱动能力为5mA（对应TI驱动能力不可调，固定为4mA），如果需要调节驱动能力可以通过PIN11、PIN12、PIN13、PIN21、PIN22、PIN23六个寄存器控制。
6. 乾芯2800137外部晶振的单双端输入无需 XTALCR寄存器的SE的配置,TI需要配置SE切换单双端。

3.7. CPU_SYS

1. 乾芯2800137不存在CPUSYSLOCK1和PIEVERRADDR寄存器，同时不支持相应的功能特性。
2. TI 2800137存在FSITX、FSIRX、DCC_0模块独立的时钟开关，乾芯2800137对应没有这些模块的时钟独立开关（这些模块的时钟总是处于使能状态）。
3. 乾芯2800137不存在SIMRESET模拟复位寄存器，TI中存在。
4. 乾芯2800137中的LPMCR寄存器增加GPIOLPSET控制是否启用IO唤醒，TI中不存在，IO唤醒始终启用。
5. 乾芯2800137中增加HHRPWM寄存器，控制是否启用HHRPWM或HRPWM，TI中不存在。
6. 乾芯2800137中RESC寄存器没有SCCRESETn(DCSM RST)，TI中存在。
7. 乾芯2800137中没有 USER_REG1_SYSRSn、USER_REG2_SYSRSn、USER_REG1_XRSn、USER_REG2_XRSn、USER_REG1_PORESETn、USER_REG2_PORESETn、USER_REG3_PORESETn、USER_REG4_PORESETn寄存器，TI中存在。

3.8. CPUTIMER

无差异。

3.9. DEV_CFG

1. 乾芯2800137没有PARTIDL、PARTIDH、REVID、TRIMERRSTS、ECAPTYPE寄存器，

不支持相应的功能特性。

2. 乾芯2800137不支持TAP_STATUS寄存器，对应JTAGS的状态由一组内部寄存器管理。
3. 乾芯2800137不支持ECAPTYPE寄存器，对应ECAP写入的状态由一组内部寄存器管理。
4. 乾芯2800137额外添加了CANFD寄存器用于控制CAN模块的CANFD特性的开启和关闭。
5. 乾芯2800137额外添加了FLASHCLKDIV寄存器用于控制FLASH_CTRL模块的时钟分频。
6. 乾芯2800137没有SOFTPRES28的Flash软复位控制，TI有Flash的软复位控制。

3.10. MEM_CFG

1. 乾芯2800137没有MEM_CFG寄存器组，但乾芯2800137支持指令RAM和数据RAM空间大小的任意划分。对应内存空间的划分管理通过配置驱动库的ldscript_Memory.ld文件即可。

3.11. MEMORY_ERROR

1. 乾芯2800137暂不支持TI的MEMORY_ERROR特性。

3.12. NMI_INTRUPT

1. TI NMI的RAMUNCERR错误指的是内存出现不可纠正错误，乾芯的RAMUNCERR是指数据内存区域存在不可纠正错误。
2. 乾芯由于没有PIEVECT特性，因此不支持TI的PIEVECTERR错误类型。取而代之的是额外增加的一个指令RAM区域取址错误（IRAMUNCERR）。

3.13. PIE_CTRL

1. TI的PIECTRL[ENPIE]位用于开启外设的中断全局支持，乾芯2800137的外设中断总是默认开启支持，无ENPIE操作位。
2. TI的PIECTRL[PIEVECT]位表示从PIE向量表中获取的向量地址，乾芯2800137不支持PIEVECT中断向量地址的获取操作。

3.14. SYNC_SOC

无差异。

3.15. SYS_STATUS

无差异。

3.16. TEST_ERROR

1. 乾芯2800137不支持TI的TEST_ERROR特性。

3.17. UID

1. 乾芯2800137无UID寄存器，暂不支持相关特性。

3.18. WD

无差异。

3.19. XINT

1. 乾芯在2800137中额外添加了DMA模块，该DMA模块额外支持XINT通过外部GPIO信号触发DMA功能，对应XINT寄存器组中有一组寄存器用于控制XINT的DMA功能的使能、DMA操作溢出标志位以及溢出标志位的清除寄存器。

```
146
147 struct XINT_REGS
148 {
149     union XINT1CR_REG XINT1CR;              // 0x00 XINT1 configuration register
150     union XINT2CR_REG XINT2CR;              // 0x04 XINT2 configuration register
151     union XINT3CR_REG XINT3CR;              // 0x08 XINT3 configuration register
152     union XINT4CR_REG XINT4CR;              // 0x0C XINT4 configuration register
153     union XINT5CR_REG XINT5CR;              // 0x10 XINT5 configuration register
154     Uint32 XINT1CTR;                        // 0x14 XINT1 counter register
155     Uint32 XINT2CTR;                        // 0x18 XINT2 counter register
156     Uint32 XINT3CTR;                        // 0x1C XINT3 counter register
157     union XTINT_DMAEN_REG XINT_DMAEN;       // 0x20 XINT TINT DMA enable
158     union XTINT_OVERFCLR_REG XINT_OVERFCLR; // 0x24 XINT TINT DMA overflow flag clear
159     union XTINT_OVERF_REG XINT_OVERF;       // 0x2C XINT TINT DMA overflow flag
160 };
```

4. Booting

1. TI使用eFlash工艺，启动流程支持从Flash直接取指令执行。乾芯使用内嵌QSPI Flash工艺，启动流程上总是在BOOT阶段将Flash里的代码全部搬运到SRAM后再跳转到SRAM执行。因此乾芯的代码运行总是在SRAM进行（调试时IDE会直接将程序下载到SRAM来完成调试运行）。

5. DCSM

1. 乾芯2800137的DCSM的ZSB数量为2，对应TI为30个。
2. TI VCU CRC指令对地址进行校验，乾芯的VCU CRC指令对通用寄存器进行校验，因此乾芯2800137无CRCLOCK功能。

6. Flash

1. 乾芯2800137使用内嵌QSPI Flash工艺，flash在使用上相对于TI更加简单。使用接口上只有少量几个接口即可完成Flash的初始化和读写操作，具体操作接口参见参考手册和驱动库相关模块说明和定义。

7. DCC

无差异。

8. GPIO

1. TI 2800137 GPIO的驱动能力在4mA左右，乾芯2800137 GPIO的驱动能力为2.5mA、5mA、10mA三挡可调（默认为5mA驱动能力）。
2. 乾芯2800137的引脚复用表中CAN模块新增了CANA_STBY引脚的复用，对应TI的CAN模块没有CANA_STBY引脚。

9. X-BAR

无差异。

10. Analog Subsystem

1. 乾芯ANAREFCTL寄存器与TI不同，详情见用户参考手册。
2. 乾芯VMONCTL寄存器增加ADJ_PVD配置位，详情见用户参考手册。

11. ADC

1. 乾芯2800137的ADC多个ADC模块不可同时接入使用同一个输入信号，且ADC模块时钟开启时需将ADCA和ADCC按步骤同时开启（即便只使用了一个ADC），正确的步骤为：先开启ADCA的时钟，插入一个系统时钟周期(通过插入一个NOP指令可以实现)等待后再开启

ADCC。否则，在只单独开启一个ADC模块时钟的情况下，该ADC模块的采样率和精度会降低。下面是正确的ADC使能流程的函数调用样例：

```
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_ADCA);
__asm__("nop||");
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_ADCC);
```

2. 乾芯2800137的ADC驱动库中ADC_setVREF()和ADC_enableConverter()函数都会自动调用ADC_autoCalibration()函数来实现ADC的精度校准。若用户项目只使用比特位域库来作为低层驱动库则需要复制ADC_autoCalibration()函数实现并总是在修改ADC的参考电压配置和使能ADC之后手动调用一次该校准函数，否则将会导致ADC的无法正常使用。
3. 乾芯2800137的ADC内部参考电压支持1.65v，2.5v，3.3V，对应TI 2800137支持2.5V和3.3V两种内部参考电压。
4. 乾芯2800137的ADC采样窗口配置值的含义和TI不同，TI的ADCSOCxCTL[ACQPS]值表示采样窗口的周期数且周期按照系统时钟（SYSCLK）计算，乾芯ADC的ADCSOCxCTL[ACQPS]值表示离散的十几种采样周期数，且周期按照ADC采样时钟（ADCCLK）计算，以下是乾芯ADCSOCxCTL[ACQPS]值的具体定义列表：

ACQPS	R/W	0	SOC采集预缩放，控制此SOC的采样和保持窗口。配置的采集时间必须至少与一个ADCCLK周期一样长，才能进行正确的ADC操作。设备数据表还将指定采样和保持窗口的最小持续时间。 0h：维持1个周期 (默认) 1h：维持2个周期 2h：维持3个周期 3h：维持4个周期 4h：维持5个周期 5h：维持6个周期 6h：维持7个周期 7h：维持8个周期 8h：维持10个周期 9h：维持20个周期 Ah：维持30个周期 Bh：维持40个周期 Ch：维持50个周期 Dh：维持60个周期 Eh：维持70个周期 Fh：维持80个周期

3、乾芯2800137的ADCCTL2[PRESCALE]固定为2分频(不可修改)，如若配置ADC不同的采样率可以配置不同的采样窗口或系统时钟来实现。TI的ADCCTL2[PRESCALE]为1~8分频可配，默认1分频。

6. 乾芯2800137的提供更多的ADC SOC触发源，包括SW、CPU1_TIMER 0-2、CPU2_TIMER0-2、GPIO、EPWM1-12以及ADC INT1-2。
7. 乾芯2800137的ADC读取采样结果必须调用ADC_readResult(uint32_t resultBase, ADC_SOCNumber socNumber)函数。直接读则会导致采样码值回滚。

12. CMPSS

1. 乾芯2800137比较器提供7种迟滞电压配置，TI提供了4种。
2. 乾芯2800137提供4个全功能比较器，TI 2800137是1个全功能 CMPSS 和3个CMPSS_LITE比较器。
3. 乾芯2800137比较器使能和dac使能可单独配置。EN_COMP1代表高比较器的使能，EN_COMP2代表低比较器的使能， COMPDACE代表DAC使能。使能配置样例如下（使用CMPSS_enableModule()函数时可忽略此操作）：

```
EALLOW;
Cmpss1Regs.COMPCTL.bit.COMPDACE = 0x1;
Cmpss1Regs.COMPCTL.bit.EN_COMP1   = 0x1;
Cmpss1Regs.COMPCTL.bit.EN_COMP2   = 0x1;
EDIS;
```

13. ePWM

13.1. TB

1. 乾芯2800137在同步开启多个EPWM模块的TBCLK时钟（CpuSysRegs.PCLKCR0[TBCLKSYNC]=1）之前必须先开启所有EPWM模块的时钟（CpuSysRegs.PCLKCR2[EPWMx]），如果在开启EPWM模块时钟之前开启TBCLKSYNC将会导致多路同步启动时存在意外的相位差。

13.2. CC

1. 当开启影子寄存器时，加载方式选择为CTR=0加载时CMPx最小值不能为0，加载方式选择为CTR=PRD加载时CMPx最大值不能为PRD（当加载方式选择为CTR=0或CTR=PRD时，CMPx最小值不能为0，最大值不能PRD），TI无此限制。

13.3. GLD

1. 乾芯2800137当GLDCNT数值不为0且没有触发加载时GLD使能关闭后GLDCNT不会清除。
2. 乾芯2800137在使用全局加载时GLDCTL[GLDPRD]配置在使用过程中不支持任意改变。

13.4. HRPWM

1. 乾芯2800137的HRPWM MEP精度在110ps，对应TI在150ps。
2. 在使用SFO时，需要通过函数HRPWM_setHRPWR_CALPWRON()使能HRPWR，而TI随着SFO开启则自动使能。

3.乾芯2800137的TBPRDHR寄存器不支持全局加载。

4.当配置高精度延时边沿时，假如经过死区的POLSEL翻转，由于乾芯的高精度是在死区POLSE翻转前添加，因此当配置上升沿/下降沿延时后经过翻转延时边沿同样跟着翻转，而TI的高精度延时加在POLSEL之后，因此TI是先进性的翻转再找到下降沿/上升沿进行延时。

13.7.3 Operational Highlights for the Dead-Band Submodule

The configuration options for the dead-band submodule are shown in Figure 13-35.

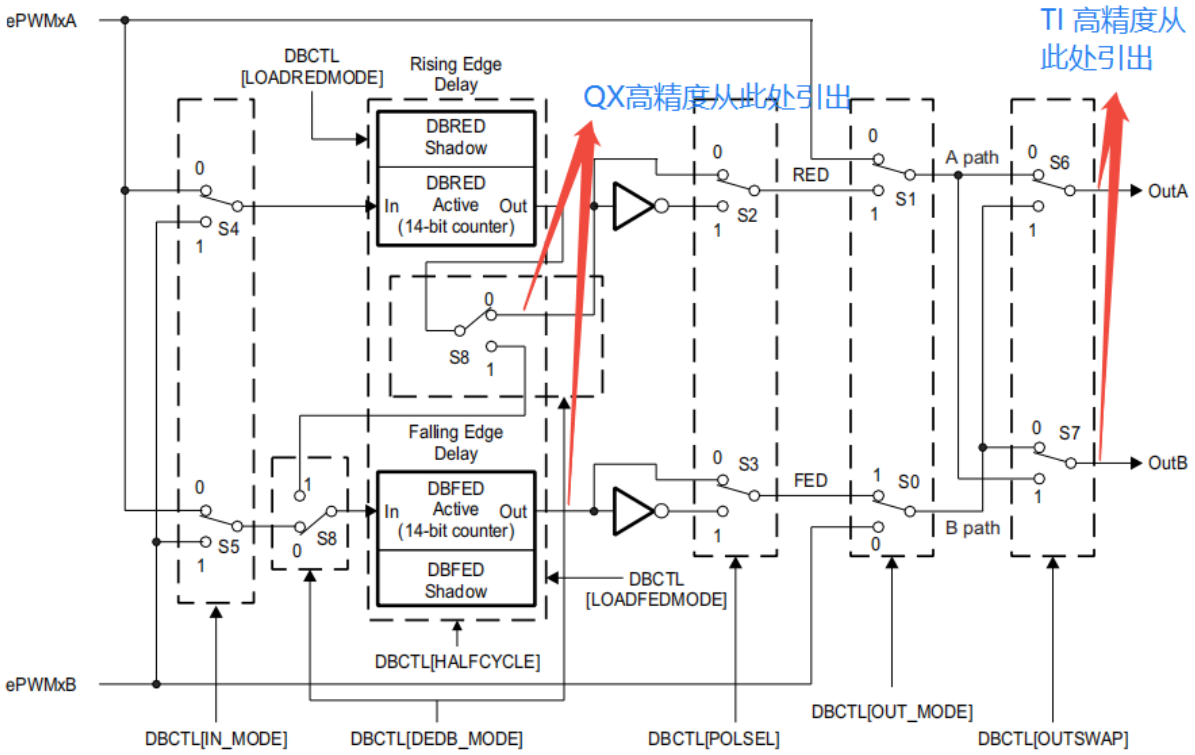


Figure 13-35. Configuration Options for the Dead-Band Submodule

例子：假如此时A路开启了高精度占空比延时，且为下降沿延时，当S2从0接到1时，波形发生反转反转后的延时则变为上升沿延时，而TI仍为下降沿延时。

软件修改方式：当需要配置下降沿延时时，需要经过POLSEL的情况下改为上升沿延时即可。

5.当使用高精度周期延时时，如果要用到整数和小数的立即模式，不能直接将PRDLD写1，可以使用全局加载oneshot的方式实现立即生效.

14. eCAP

无差异。

15. eQEP

无差异。

16. CAN

QX全系列芯片现已支持CANFD，TI C2000系列某些芯片只支持传统CAN，本文列出了QX CANFD模块和TI传统CAN模块之间的差异，旨在指导用户将TI传统CAN工程迁移到QX芯片上。

16.1. 差异对比

与传统CAN相比，QX CANFD具有两个显著优势：

• 数据段的比特率更快，提高了整体吞吐量。应用可以通过设置 RBUF.BRS = 1 选择以不同的比特率发送整个帧。仲裁段以传统比特率发送，而数据段以快速比特率发送。

• 与传统 CAN（多达 8 字节）相比，有效负载大小更高（多达 64 字节）

表2-1 重点介绍了QX CANFD 和 TI CAN的模块之间的主要差异。

表 16-1 QX CANFD 和 TI CAN的模块之间的主要差异

功能	QX CANFD	TI CAN
比特率	可以使用两种比特率：用于仲裁段的较慢比特率和用于数据段的较快比特率。	整个帧使用固定比特率。
每帧发送的字节数（有效负载能力）	除0至8字节外，还可以发送 12/16/20/24/32/48/64 数据字节	可以发送从0到8的任意数量的字节
数据存储元素	数据存储在与过滤器元素相关的缓冲器中	数据存储在消息对象中。消息对象有时也称为邮箱。
发送器延迟补偿	在数据段实现更快的比特率时需要	不需要
位速率配置	需配置快慢两种速率（CANFD） 整个位时间 = 段1 + 段2 段1的实际值 = 寄存器值 + 2 段2的实际值 = 寄存器值 + 1	只需配置一种速率 整个位时间 = 段1 + 段2 + 1 段1的实际值 = 寄存器值 + 1 段2的实际值 = 寄存器值 + 1
滤波器配置	16组滤波器组，通过ID和掩码寄存器确定要接收的CAN报文	每个消息对象都可配置一个对应的滤波器，同样通过ID和掩码寄存器确定要接收的CAN报文
数据发送	有两个发送缓冲区，一个主发送缓冲区（PTB），一个次要发送	32个消息对象，这些消息对象可配置为发送或者接收，根据消息

	缓冲区（STB），PTB只能存储一帧，PTB的优先级比STB高，STB有16个消息槽，可工作在FIFO或者优先级模式	对象标识符确定发送顺序
数据接收	8个接收消息槽，工作在FIFO模式	32个消息对象，这些消息对象可配置为发送或者接收，根据消息对象标识符确定读取顺序

16.2. 模块初始化

16.2.1. TI CAN的初始化

1. 调用CAN_initModule初始化消息 RAM
2. 调用CAN_setBitTiming配置位速率
3. 调用CAN_setupMessageObject配置消息对象（接收和发送配置）。

16.2.2. QX CANFD的初始化

1. 首先配置CANFD模块进入reset模式（CFG_STAT.RESET=1）以支持波特率、过滤器等寄存器的设置。
2. 调用CAN_initModule初始化CAN模式(传统CAN 或 CANFD)、发送延迟补偿使能以及补偿的数值。
3. 调用CAN_setAcceptFilter配置接收过滤器，共有16个过滤器可配置，可通过此函数选择要使能的过滤器以及过滤器所要过滤的CAN报文类型。
4. 调用CAN_setBitTimingSlow和CANFD_setBitTimingFast配置快慢位速率。
5. 配置CANFD模块进入常规模式（CFG_STAT.RESET=0）

16.2.3. 模块初始化代码片段

图3-1展示了TI CAN模块的初始化步骤，图3-2展示了QX CANFD的初始化步骤

```
}

void myCAN0_init(){
    CAN_initModule(myCAN0_BASE);
    //
    // Refer to the Driver Library User Guide for information on how to set
    // tighter timing control. Additionally, consult the device data sheet
    // for more information about the CAN module clocking.
    //
    CAN_setBitTiming(myCAN0_BASE, 15, 0, 15, 7, 3);
    //
    // Enable CAN test mode
    //
    CAN_enableTestMode(myCAN0_BASE, CAN_TEST_EXL);
    //
    // Initialize the transmit message object used for sending CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 1
    //      Message Identifier: 660
    //      Message Frame: CAN_MSG_FRAME_STD
    //      Message Type: CAN_MSG_OBJ_TYPE_TX
    //      Message ID Mask: 0
    //      Message Object Flags:
    //      Message Data Length: 2 Bytes
    //
    CAN_setupMessageObject(myCAN0_BASE, 1, myCAN0_MessageObj1_ID,
                           CAN_MSG_FRAME_STD,CAN_MSG_OBJ_TYPE_TX, 0, 0,2);
    //
    // Initialize the transmit message object used for sending CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 2
    //      Message Identifier: 660
    //      Message Frame: CAN_MSG_FRAME_STD
    //      Message Type: CAN_MSG_OBJ_TYPE_RX
    //      Message ID Mask: 0
    //      Message Object Flags:
    //      Message Data Length: 2 Bytes
    //
    CAN_setupMessageObject(myCAN0_BASE, 2, myCAN0_MessageObj2_ID,
                           CAN_MSG_FRAME_STD,CAN_MSG_OBJ_TYPE_RX, 0, 0,2);
    //
    // Start CAN module operations
    //
    CAN_startModule(myCAN0_BASE);
}

```

图3-1 TI CAN初始化

```c
void CAN_init()
{

    CAN_Init_Config canInit;
    CANFD_Config canFd;

    //
    // enable module clock
    //
    SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANB);

    //
    //Initializes CAN parameters.
    //
    CAN_structInit(&canInit);
    CANFD_structInit(&canFd);
    canInit.canFDControl = CAN_FD_ENABLE;
    canFd.TDC = CAN_FD_TDC_ENABLE;
    canFd.SSPOffset = 16;
    canInit.ptrCanFd = &canFd;

    //
    // Software reset.Set the bit rate and filter need the can in software reset mode.
    //
    CAN_enableStatReset(CANFDB_BASE);
    //
    //Initialize CAN module
    //
    CAN_initModule(CANFDB_BASE, &canInit);

    //
    //Set CAN filter
    //Accept frames with extended ID 0x121314x5.
    //
    CAN_setAcceptFilter(CANFDB_BASE, CAN_ACF1, CAN_ID_EXT, 0x12131415UL, 0x000000F0UL);

    //
    // 500K bundrate when can pclk = 100Mhz, sample point = 80%
    // (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
    //
    CAN_setBitTimingSlow(CANFDB_BASE, 3, 38, 9, 9);
    //
    // 2000K bundrate when can pclk = 100Mhz, sample point = 80%
    // (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
    //
    CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);

    //
    // CAN enters normal communication mode.
    //
    CAN_disableStatReset(CANFDB_BASE);
    //
    // Enable CAN Interrupt.
    //
    CAN_enableInterrupt(CANFDB_BASE, CAN_INT_ALL);

    //
    // delay for config ready
    //
    simple_delay(100);
}
```

图3-2 QX CANFD初始化

16.3. 位速率配置

传统 CAN 和 CAN FD 之间的位时序配置不同。在传统 CAN 中，该过程相对更简单，因为整个帧的比特率是相同的。在 CAN FD 中，可以使用两种不同的比特率：较慢的“标称”比特率和较快的“数据”比特率。在 QX CANFD 模块中，可通过在模块初始化期间分别调用CAN_setBitTimingSlow和 CANFD_setBitTimingFast来配置这两种比特率。 应用可以选择仅利用在 CAN FD 中发送的每帧包含更多的数据字节数，并对整个帧使用相同的比特率。数据段采用快速比特率时，应设定合适的发送器延迟补偿（TDC），如果没有该补偿，可能发生位错误。下面是一个计算位速率参数的示例：

示例1

设定CAN时钟频率为100MHZ，标称比特率 = 500kbps，数据比特率 = 2Mbps

图3-3给出CAN位时间定义图，虚线上部分为CAN协议规定的位时间，虚线下部分为QX CANFD控制器CAN-CTRL定义的位时间。其中segment1和segment2可以通过寄存器 SBT和FBT设定。SBT 寄存器和FBT寄存器只能在CFG_STAT.RESET=1即 CAN 软件复位时设定。SBT 寄存器用于CAN2.0和CAN FD的仲裁段，FBT寄存器用于CAN FD数据段。

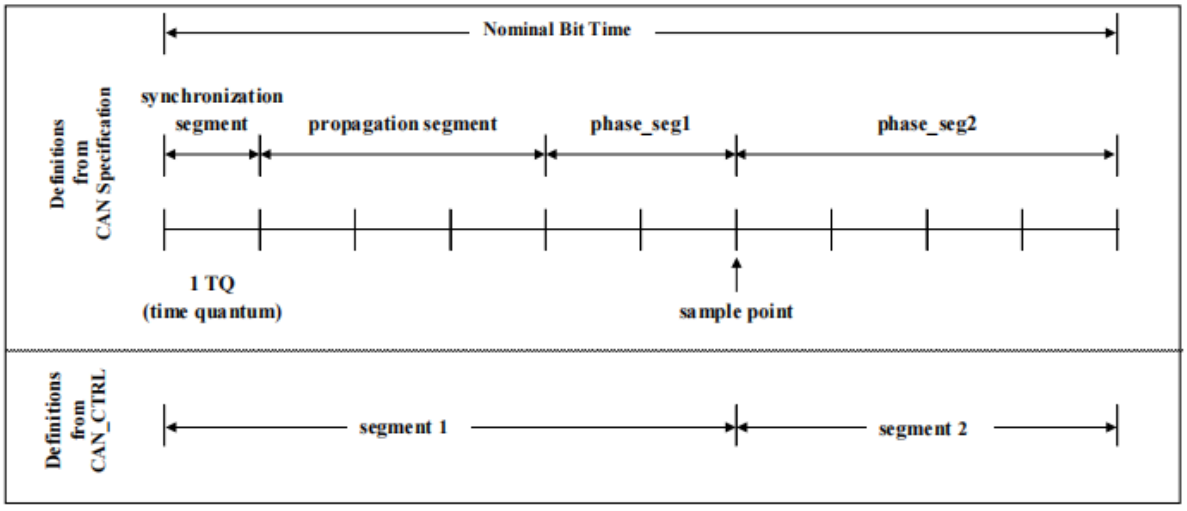


图3-3 CAN位时间定义图

根据位时间定义图可得出计算比特率的公式如下：

对于500kbps的标称比特率，可以选择prescaler（预分频器）的值为4，bittime（位时间）为50个TQ，因此设定慢速比特率分频寄存器S_PRESC的值为3，通过图3-3可知bittime = segment1 + segment2，通过segment1 和segment2的不同组合可产生不同的采样点，这里设定segment1 为40，segment2为10，SP为80%，注意：segment1 和segment2设定需满足段1必须略大于段2。因此慢速比特率S_Seg_1寄存器写入38，S_Seg_2寄存器写入9，除了

段1和段2寄存器之外还需设置同步跳跃宽度SJW的值，建议SJW的值设定为和段1相同，因此慢速比特率S_SJW寄存器写入9。对于函数调用来说：

CAN_setBitTimingSlow(CANFDB_BASE, 3, 38, 9, 9);

同理可设置2Mbps的快速比特率，可以选择prescaler（预分频器）的值为2，bittime（位时间）为25个TQ，因此设定快速比特率分频寄存器F_PRESC的值为1，这里设定segment1 为20，segment2为5，SP为80%，因此快速比特率F_Seg_1寄存器写入18，F_Seg_2寄存器写入4，快速比特率F_SJW寄存器写入4。对于函数调用来说：

CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);

16.4. 中断处理

16.4.1. QX CANFD中断源

从 CPU 级别（PIE、IFR 和 INTM）来看，QX CANFD 和 TI CAN 之间的中断处理是相同的。但是中断处理在模块级别存在差异。表 5-1 总结了 TI CAN 和 QX CANFD 模块之间中断处理的基本差异：

表 16-2 TI CAN 和 QX CANFD 模块之间中断处理的基本差异

类别	TI CAN	QX CANFD
中断源	与每个消息对象相对应的错误、状态和发送/接收中断	14个内部中断源（在下表中指定）
全局中断寄存器	用于启用、读取和清除存在的全局中断的寄存器	不存在
配置接收中断	可以根据需要，通过设置每个消息对象中的RxIE位来单独启用接收中断	启用之后，对于任何接收到的新消息都会产生中断
确定接收中断的源	从寄存器 CAN_INT 读取的值对应于已接收到消息的消息对象编号	中断仅表示 Rx 缓冲器中已接收到新消息。
配置发送中断	可以根据需要，通过设置每个消息对象中的TxIE位来单独启用发送中断	可以通过配置寄存器 RTIE.TPIE和RTIE.TSIE来启用发送中断
确定发送中断的源	从寄存器 CAN_INT 读取的值对应于已发送消息的消息对象编号	中断仅表示发送已完成。

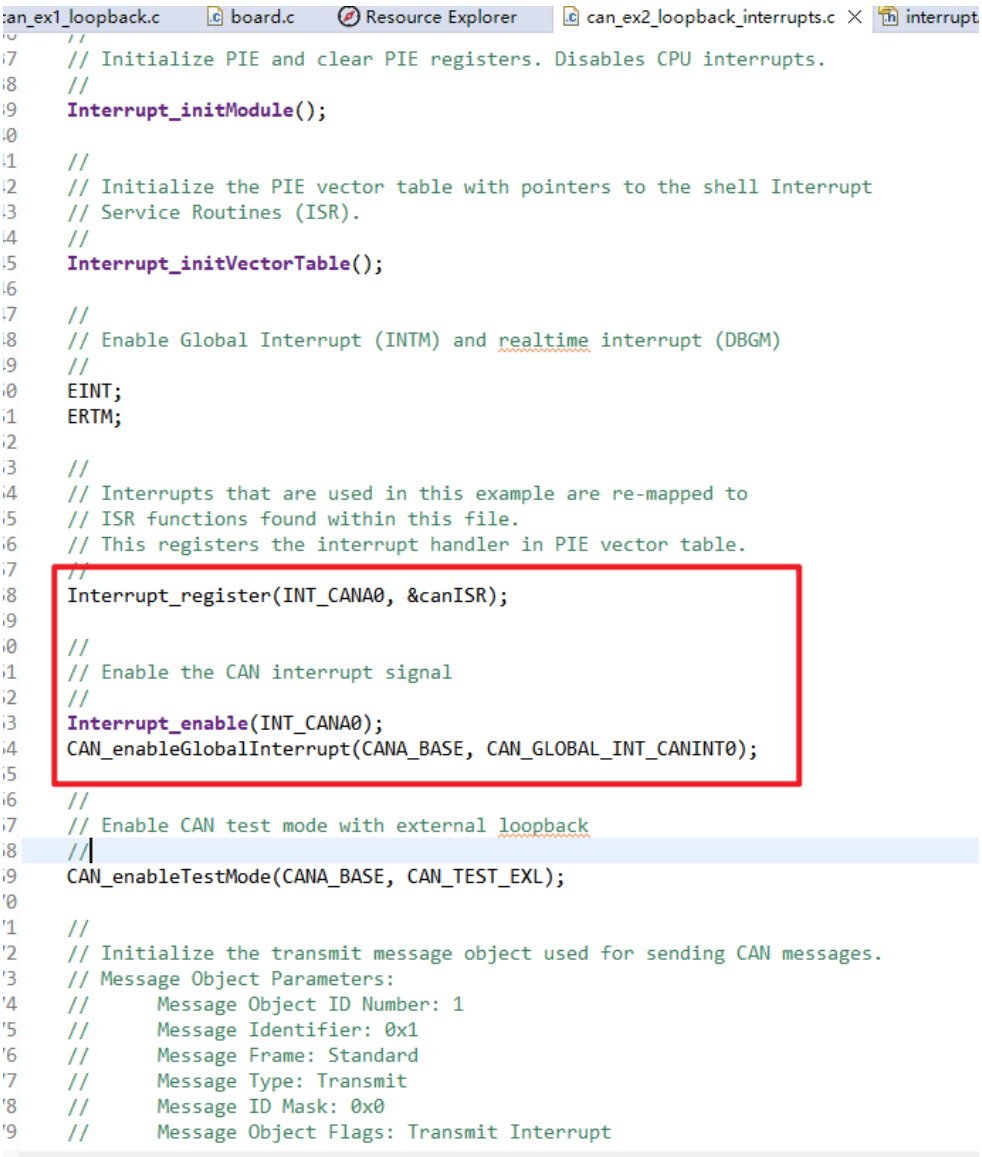
表 16-3 QX CANFD中断源

中断标志	描述
RIF	接收中断
ROIF	接收上溢中断
RFIF	接收BUF满中断
RAFIF	接收BUF将满中断
TPIF	PTB发送中断
TSIF	STB发送中断
EIF	错误中断
AIF	取消发送中断
EPIF	错误被动中断
ALIF	仲裁失败中断
BEIF	总线错误中断
WTIF	触发看门中断
TEIF	触发错误中断
TTIF	时间触发中断

16.4.2. TI CAN中断处理

器件级中断配置：

1. 初始化 PIE 和 PIE 向量表。
2. 在 PIE 向量表中配置中断处理程序。在中断控制器中启用中断。



模块级中断配置

1. 使用 CAN 控制寄存器 (CAN_CTL) 启用错误和状态中断。在单独设置消息对象时启用消息对象中断。
2. 选择要使用寄存器 (CAN_IP_MUX21) 路由每个消息对象中断的中断线路，其中每个位对应于单个消息对象。
3. 中断服务例程 (ISR)：读取中断寄存器 (CAN_INT) 以确定中断源（状态/错误/特定消息对象）。通过写入CAN错误和状态寄存器 (CAN_ES) 或通过清除相应消息对象中的 IntPnd 位来清除中断。清除相应中断线路的全局中断标志。

4. 通过 PIEACK 响应中断。

```
//
// CAN A ISR - The interrupt service routine called when a CAN interrupt is
//             triggered on CAN module A.
//
__interrupt void
canaISR(void)
{
    uint32_t status;

    // Read the CAN-A interrupt status to find the cause of the interrupt

    status = CAN_getInterruptCause(CANA_BASE);

    // If the cause is a controller status interrupt, then get the status

    if(status == CAN_INT_INT0ID_STATUS)     // Read the controller status.
    {
        status = CAN_getStatus(CANA_BASE);

        // Check to see if an error occurred.

        if(((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_MSK) &&
           ((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_NONE))
        {
            errorFlag = 1;       // Set a flag to indicate some errors may have occurred.
        }
    }
    else if(status == TX_MSG_OBJ_ID)
    {
        // Transmit Message handling will go here

        CAN_clearInterruptStatus(CANA_BASE, TX_MSG_OBJ_ID);  // Clear the message object interrupt
    }
    else if(status == RX_MSG_OBJ_ID)
    {
        // Receive message handling will go here

        CAN_clearInterruptStatus(CANA_BASE, RX_MSG_OBJ_ID);  // Clear the message object interrupt
    }

    else
    {
        //
        // Spurious interrupt handling can go here.
        //
    }

    //
    // Clear the global interrupt flag for the CAN interrupt line
    //
    CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0);

    //
    // Acknowledge this interrupt located in group 9
    //
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);
}
```

16.4.3. QX CANFD中断处理

器件级中断配置：

1. 初始化 PIE 和 PIE 向量表。
2. 在 PIE 向量表中配置中断处理程序。在中断控制器中启用中断。

```
137
138
139 //*****************************************************************************
140 //
141 // INTERRUPT Configurations
142 //
143 //*****************************************************************************
144 void INTERRUPT_init()
145 {
146
147     //
148     // Interrupt Setings for INT_CANB_BASE
149     //
150     Interrupt_register(INT_CANB, &canaISR);
151     Interrupt_enable(INT_CANB);
152 }
153
```

模块级中断配置

1. 使用 CANFD模块中断控制寄存器 (RTIE和ERRINT) 启用错误和状态中断。
2. 中断服务例程 (ISR)：读取中断标志寄存器(RTIF和ERRINT)以确定中断源（状态/错误）。通过写入CAN中断标志寄存器 (RTIF和ERRINT)来清除中断。

```
90      CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);
91
92      //
93      // CAN enters normal communication mode.
94      //
95      CAN_disableStatReset(CANFDB_BASE);
96      //
97      // Enable CAN Interrupt.
98      //
99      CAN_enableInterrupt(CANFDB_BASE, CAN_INT_ALL);
100
101     //
102     // delay for config ready
103     //
104     simple_delay(100);
105 }
106
107
```

16.5. 发送数据

QX CANFD和TI CAN模块的结构设计有差异，发送数据的机制存在不同，TI CAN模块中包含32个消息对象，配置好消息对象后，通过IFx寄存器写入配置好的消息对象，每个消息对象有标识符，多个消息对象可组成FIFO，标识符最小的优先级最高。QX CANFD模块发送缓冲区有两部分，一个PTB（主发送缓冲区）和一个STB（次要发送缓冲区，具有16个消息槽），PTB具有最高的优先级，但只能缓冲1帧，STB比PTB优先级低，可以缓冲16帧数据，STB中的数据帧的传输可以工作在FIFO模式下或者优先级模式下（根据CAN ID判优）QX CANFD 的数据帧可以更长并采用两种不同的比特率。经过适当配置后，该模块就会负责比特率切换和处理更大的有效负载。

16.5.1. 使用TI CAN模块发送数据

1、调用CAN_setupMessageObject设置消息对象，需指定要配置的消息对象，以及配置CAN数据帧的帧类型、配置消息对象为发送动作、消息对象的数据负载大小等。

2、调用CAN_sendMessage写入 IFx 寄存器，而这些寄存器依次将消息 ID (ARBID)、DLC 和数据写入消息对象

3. CAN_sendMessage最后会设置 IFx 寄存器 (CAN_IFxCMD) 中的 TXRQST 位，以表示消息对象已准备好发送。

4. 当总线空闲时，消息处理程序解析准备好发送的消息对象，并发送可用的最高优先级消息。

```
191     //      Message Frame: Standard
192     //      Message Type: Receive
193     //      Message ID Mask: 0x0
194     //      Message Object Flags: Receive Interrupt
195     //      Message Data Length: 4 Bytes (Note that DLC field is a "don't care"
196     //      for a Receive mailbox
197     //
198     CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID, 0x1, CAN_MSG_FRAME_STD,
199                            CAN_MSG_OBJ_TYPE_RX, 0, CAN_MSG_OBJ_RX_INT_ENABLE,
200                            MSG_DATA_LENGTH);
201
202     //
203     // Initialize the transmit message object data buffer to be sent
204     //
205     txMsgData[0] = 0x12;
206     txMsgData[1] = 0x34;
207     txMsgData[2] = 0x56;
208     txMsgData[3] = 0x78;
209
210     //
211     // Start CAN module operations
212     //
213     CAN_startModule(CANA_BASE);
214
215     //
216     // Loop Forever - A new message will be sent once per second.
217     //
218     for(;;)
219     {
220         //
221         // Check the error flag to see if errors occurred
222         //
223         if(errorFlag)
224         {
225             asm("   ESTOP0");
226         }
227
228         //
229         // Verify that the number of transmitted messages equal the number of
230         // messages received before sending a new message
231         //
232         if(txMsgCount == rxMsgCount)
233         {
234             CAN_sendMessage(CANA_BASE, TX_MSG_OBJ_ID, MSG_DATA_LENGTH,
235                             txMsgData);
236         }
237         else
238         {
239             errorFlag = 1;
```

配置消息对象

填充数据并启动发送

16.5.2. 使用QX CANFD模块发送数据

1、初始化发送BUF控制位，对发送帧的各个参数进行配置，包括配置是否启用快慢速率切换、使用can2.0还是canfd通讯、标准帧还是扩展帧、有效数据负载大小、数据帧还是远程帧

2、调用CAN_fillMessage将发送BUF控制位写入发送缓冲区，并将要发送的数据填充到发送缓冲区，这里要选择是填充到PTB还是STB。

3.调用CAN_startTx启动对应的缓冲区的数据发送。

4. 当总线空闲时，消息处理程序解析准备好发送的消息对象，并发送可用的最高优先级消息。

```
//
// Set can TBUF control.
//
CAN_TBUF_Ctrl tx1Ctrl;
tx1Ctrl.TTSEN = CAN_TTSEN_DISABLE;
tx1Ctrl.BRS = CANFD_BRS_SLOW;
tx1Ctrl.DLC = CAN_DLC8;
tx1Ctrl.FDF = CAN_FDF_CAN20;
tx1Ctrl.IDE = CAN_IDE_STANDARD;
tx1Ctrl.RTR = CAN_RTR_DATA;

// Transmit messages from CAN-A.
//
for(i = 0; i < MSGCOUNT; i++)
{

    //
    // Transmit the message.
    //
    CAN_fillMessage(CANFDA_BASE, CAN_TX_BUF_PTB, 0x88UL, &tx1Ctrl, txMsgData);
    CAN_startTx(CANFDA_BASE, CAN_TX_REQ_PTB);

    //
    // Delay 0.25 second before continuing
    //
    DEVICE_DELAY_US(250000);

    //while(txMsgSuccessful);
    //
    // Increment the value in the transmitted message data.
    //
```

初始化发送BUF控制位

填充数据

启动发送

16.6. 接收数据

在 TI CAN 的消息 RAM 中，有 32 个可配置的消息对象可用于接收。接收消息对象用于存储接收到的数据。如果应用需要，可以为一个或多个消息对象启用接收过滤。CPU 对消息 RAM 的读写访问通过三个接口寄存器 (IFx) 来完成。

QX CANFD模块接收缓冲器共有8个消息槽，这些槽工作在FIFO模式下，RB SLOT 通过 RBUF 寄存器来读取接收到的数据，总是最先读取最早接收到的数据， 并通过 RCTRL 寄存器的 RREL 设置为 1 释放已经读取的 RB SLOT，并指向下一个 RB SLOT。共有16组过滤器可以启用。

16.6.1. 使用TI CAN模块接收数据

1. 配置接收消息对象：这涉及写入消息 ID (ARBID)，并在需要时屏蔽要接收的帧。
2. 对于每个接收到的帧，模块将按升序对照接收消息对象进行检查。当第一次匹配时，帧存储在相应的消息对象中。
3. 通过轮询或使用中断，确定新数据的接收。对于轮询，寄存器 CAN_NDAT_21 中的每个接收消息对象都有一个对应的位。对于使用中断，相应章节中已概述了该过程。
4. 使用其中一个 IFx 寄存器从接收的帧中读取数据。

```
164     //
165     CAN_setupMessageObject(CANA_BASE, TX_MSG_OBJ_ID, 0x15555555,
166                            CAN_MSG_FRAME_EXT, CAN_MSG_OBJ_TYPE_TX, 0,
167                            CAN_MSG_OBJ_TX_INT_ENABLE, MSG_DATA_LENGTH);
168     //
169     // Initialize the transmit message object data buffer to be sent
170     //
171     txMsgData[0] = 0x12;
172     txMsgData[1] = 0x34;
173     txMsgData[2] = 0x56;
174     txMsgData[3] = 0x78;
175 #else
176     //
177     // Initialize the receive message object used for receiving CAN messages.
178     // Message Object Parameters:
179     //      CAN Module: A
180     //      Message Object ID Number: 1
181     //      Message Identifier: 0x15555555
182     //      Message Frame: Extended
183     //      Message Type: Receive
184     //      Message ID Mask: 0x0
185     //      Message Object Flags: Receive Interrupt
186     //      Message Data Length: 4 Bytes (Note that DLC field is a "don't care"
187     //      for a Receive mailbox
188     //
189     CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID, 0x15555555,
190                            CAN_MSG_FRAME_EXT, CAN_MSG_OBJ_TYPE_RX, 0,
191                            CAN_MSG_OBJ_RX_INT_ENABLE, MSG_DATA_LENGTH);
192 #endif
193
194     //
195     // Start CAN module A operations
196     //
197     CAN_startModule(CANA_BASE);
198
199 #ifdef TRANSMIT
200     //
201     // Transmit messages from CAN-A.
202     //
203     for(i = 0; i < MSGCOUNT; i++)
204     {
205         //
206         // Check the error flag to see if errors occurred
207         //
208         if(errorFlag)
209         {
210             asm("   ESTOP0");
211         }
```

配置消息对象为接收

```
339
340         //
341         // Clear the message transmitted successful Flag.
342         //
343         txMsgSuccessful  = 0;
344     }
345 #else
346     else if(status == RX_MSG_OBJ_ID)
347     {
348         //                                  通过中断的方式接收数据
349         // Get the received message
350         //
351         CAN_readMessage(CANA_BASE, RX_MSG_OBJ_ID, rxMsgData);
352
353         //
354         // Getting to this point means that the RX interrupt occurred on
355         // message object 1, and the message RX is complete.  Clear the
356         // message object interrupt.
357         //
358         CAN_clearInterruptStatus(CANA_BASE, RX_MSG_OBJ_ID);
359
360         //
361         // Decrement the counter after a message has been received.
362         //
363         rxMsgCount--;
364
365         //
366         // Since the message was received, clear any error flags.
367         //
368         errorFlag = 0;
369     }
370 #endif
371     //
372     // If something unexpected caused the interrupt, this would handle it.
373     //
374     else
375     {
376         //
377         // Spurious interrupt handling can go here.
378         //
379     }
380
```

16.6.2. 使用QX CANFD模块接收数据

1. 在初始化时通过CAN_setAcceptFilte配置滤波器组

2.调用CAN_readMessage从RBUF中读取有效数据，还可以调用CAN_readMessageWithID读取帧ID和有效数据。

3. 通过轮询或使用中断，确定新数据的接收。对于轮询，可通过判断寄存器RCTRL.RSTAT标注位确定是否接收到新数据，对于中断可启用RTIE.RIE接收中断并在中断服务函数中将数据读出。

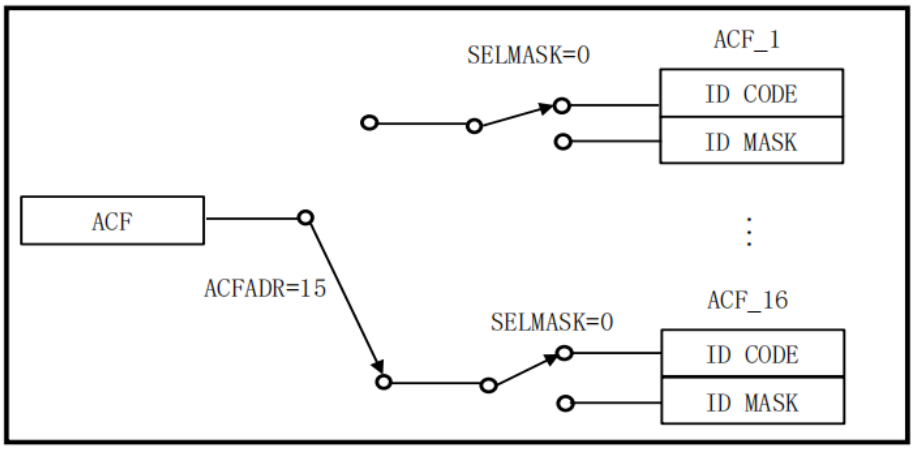
```
62      SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANA);
63
64      //
65      //Initializes CAN parameters.
66      //
67      CAN_structInit(&canInit);
68      CANFD_structInit(&canFd);
69      canInit.canFDControl = CAN_FD_DISABLE;
70  //    canFd.TDC = CAN_FD_TDC_DISABLE;
71  //    canFd.SSPOffset = 0;
72      canInit.ptrCanFd = &canFd;
73      //
74      // Software reset.Set the bit rate and filter need the can in software reset mode.
75      //
76      CAN_enableStatReset(CANFDA_BASE);
77      //
78      //Initialize CAN module
79      //
80      CAN_initModule(CANFDA_BASE, &canInit);
81
82      //
83      //Set CAN filter                                    初始化中设置滤波器组
84      //Accept frames with extended ID 0x121314x5.
85      //
86      CAN_setAcceptFilter(CANFDA_BASE, CAN_ACF1, CAN_ID_STD, 0x34UL, 0x00000000UL);
87      CAN_setAcceptFilter(CANFDA_BASE, CAN_ACF1, CAN_ID_STD, 0x35UL, 0x00000000UL);
88      //
89      // 500K bundrate when can pclk = 144Mhz, sample point = 80%
90      // (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1)) =144/2/=72     =72M/72=1Mbps
91      //
92
93      volatile uint8_t seg1 = (uint8_t)(TQ_NUM*0.8) - 2;
94      volatile uint8_t seg2 = (uint8_t)(TQ_NUM*(1.0f-ps)) - 1;
95      volatile uint8_t sjw = (seg2 > 4)?3:seg2 ;
96      volatile uint8_t fdiv = 2 -1;
97      CAN_setBitTimingSlow(CANFDA_BASE,3, 55, 14, 3);
98      //
99      // 2000K bundrate when can pclk = 100Mhz, sample point = 80%
00      // (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
01      //
02      //CANFD_setBitTimingFast(CANFDA_BASE,9, 62, 15, 15);
03
04      //
05      // CAN enters normal communication mode.
```

```
165 //              triggered on CAN module A.
166 //
167 __interrupt void canaISR(void)
168 {
169
170 #ifdef TRANSMIT
171     if (CAN_getStatus(CANFDA_BASE, CAN_FLAG_PTB_TX) == SET)
172     {
173         txMsgSuccessful = 0;
174         //
175         // Increment a counter to keep track of how many messages have been
176         // transmitted. In a real application this could be used to set flags
177         // indicate when a message is transmitted.
178         //
179         txMsgCount++;
180         //
181         // Since the message was transmitted, clear any error flags.
182         //
183         errorFlag = 0;
184     }
185 #else
186     if (CAN_getStatus(CANFDA_BASE, CAN_FLAG_RX))
187     {
188         CAN_readMessage(CANFDA_BASE, rxMsgData);
189         //
190         // Decrement the counter after a message has been received.
191         //
192         rxMsgCount--;
193
194         //
195         // Since the message was received, clear any error flags.
196         //
197         errorFlag = 0;
198     }
199 #endif
200
201     //
202     // check error.
203     //
204     if ((CAN_getStatus(CANFDA_BASE, CAN_FLAG_ERR_INT) == SET)       || \
205         (CAN_getStatus(CANFDA_BASE, CAN_FLAG_ERR_PASSIVE) == SET)  || \
206         (CAN_getStatus(CANFDA_BASE, CAN_FLAG_ARBITR_LOST) == SET)  || \
207         (CAN_getStatus(CANFDA_BASE, CAN_FLAG_BUS_ERR) == SET))
```

响应接收中断并将数据读出

16.6.3. 过滤器

QX CANFD提供 16 组 32 位筛选器用于过滤接收到的数据从而降低 CPU 负荷，筛选器可以支持标准格式 11 位 ID 或者扩展格式 29 位 ID。每组筛选器有一个 32 位 ID CODE 寄存器和一个 32 位 ID MASK 寄存器，ID CODE 寄存器用于比较接收到 CAN ID，而 ID MASK 寄存器用于选择比较的 CAN ID 位。对应的 ID MASK 位为 1 时，不比较该位的 ID CODE。 接收到的数据只要通过 16 组筛选器的任意一组，则被接收，接收到的数据存储在 RB 中，否则数据不被接收，也不被存储。每组筛选器通过 ACFEN 寄存器使能或者禁止。ID CODE 和 ID MASK 通过 ACFCTRL 寄存器的 SELMASK 位设定，SELMASK=0 时，指向 ID CODE，SELMASK=0 时，指向 ID MASK。筛选器通过 ACFCTRL 寄存器的 ACFADR 位选择。ID CODE 和 ID MASK 通过ACF寄存器访问且只能在 CFG_STAT.RESET=1 即 CAN 软件复位时设定。ACF 寄存访问筛选寄存器组的方式请参考下图。



设置滤波器1接收帧ID为0x701的标准帧，配置如下

(1) 指定要设置的过滤器 ：ACFCTRL.ACFADR = 0

(2) 指向IDCODE 寄存器：ACFCTRL.SELMASK = 0

(3) 设置IDCODE寄存器的值：ACF.ACODE = 0x701UL

(4) 指向IDMASK寄存器：ACFCTRL.SELMASK = 1

(5) 设置IDMASK 寄存器的值：ACF.AMASK= 0x0UL, ACF.AIDEE = 1, ACF.AIDE = 0

(6) 使能滤波器1：ACF_EN_0.AE_0 = 1

函数调用如下图所示

```
// enable module clock
//
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANB);

//
//Initializes CAN parameters.
//
CAN_structInit(&canInit);
CANFD_structInit(&canFd);
canInit.canFDControl = CAN_FD_ENABLE;
canFd.TDC = CAN_FD_TDC_ENABLE;
canFd.SSPOffset = 16;
canInit.ptrCanFd = &canFd;

//
// Software reset.Set the bit rate and filter need the can in software reset mode.
//
CAN_enableStatReset(CANFDB_BASE);
//
//Initialize CAN module
//
CAN_initModule(CANFDB_BASE, &canInit);

//
//Set CAN filter
//Accept frames with standard ID 0x701.
//
CAN_setAcceptFilter(CANFDB_BASE, CAN_ACF1, CAN_ID_STD, 0x701UL, 0x0UL);

//
// 500K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CAN_setBitTimingSlow(CANFDB_BASE, 3, 38, 9, 9);
//
// 2000K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);

//
// CAN enters normal communication mode.
//
CAN_disableStatReset(CANFDB_BASE);
```

设置滤波器2接收帧ID为0x121314x5的扩展帧，配置如下

(7) 指定要设置的过滤器 ：ACFCTRL.ACFADR = 1

(8) 指向IDCODE 寄存器：ACFCTRL.SELMASK = 0

(9) 设置IDCODE寄存器的值：ACF.ACODE = 0x12131415UL

(10) 指向IDMASK寄存器：ACFCTRL.SELMASK = 1

(11) 设置IDMASK 寄存器的值：ACF.AMASK= 0x000000F0UL, ACF.AIDEE = 1, ACF.AIDE = 1

(12) 使能滤波器2：ACF_EN_0.AE_1 = 1

函数调用如下图所示

```
// enable module clock
//
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANB);

//
//Initializes CAN parameters.
//
CAN_structInit(&canInit);
CANFD_structInit(&canFd);
canInit.canFDControl = CAN_FD_ENABLE;
canFd.TDC = CAN_FD_TDC_ENABLE;
canFd.SSPOffset = 16;
canInit.ptrCanFd = &canFd;

//
// Software reset.Set the bit rate and filter need the can in software reset mode.
//
CAN_enableStatReset(CANFDB_BASE);
//
//Initialize CAN module
//
CAN_initModule(CANFDB_BASE, &canInit);

//
//Set CAN filter
//Accept frames with extended ID 0x121314x5.
//
CAN_setAcceptFilter(CANFDB_BASE, CAN_ACF2, CAN_ID_EXT, 0x12131415UL, 0x000000F0UL);

//
// 500K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CAN_setBitTimingSlow(CANFDB_BASE, 3, 38, 9, 9);
//
// 2000K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);

//
// CAN enters normal communication mode.
//
CAN_disableStatReset(CANFDB_BASE);
```

设置滤波器3接收帧ID为0x1A1B1C1x的扩展帧和0x41x的标准帧，配置如下

(1) 指定要设置的过滤器 ：ACFCTRL.ACFADR = 2

(2) 指向IDCODE 寄存器：ACFCTRL.SELMASK = 0

(3) 设置IDCODE寄存器的值：ACF.ACODE = 0x1A1B1C1DUL

(4) 指向IDMASK寄存器：ACFCTRL.SELMASK = 1

(5) 设置IDMASK 寄存器的值：ACF.AMASK= 0x0000000FUL, ACF.AIDEE = 0, ACF.AIDE = 0

(6) 使能滤波器3：ACF_EN_0.AE_2 = 1.

(7) 注：ACODE 和AMASK寄存器对于标准帧0~10位有效，扩展帧0~28位有效

函数调用如下图所示

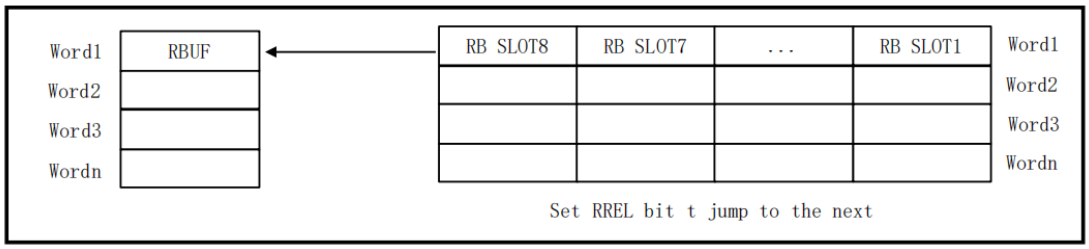
```
// enable module clock
//
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANB);

//
//Initializes CAN parameters.
//
CAN_structInit(&canInit);
CANFD_structInit(&canFd);
canInit.canFDControl = CAN_FD_ENABLE;
canFd.TDC = CAN_FD_TDC_ENABLE;
canFd.SSPOffset = 16;
canInit.ptrCanFd = &canFd;

//
// Software reset.Set the bit rate and filter need the can in software reset mode.
//
CAN_enableStatReset(CANFDB_BASE);
//
//Initialize CAN module
//
CAN_initModule(CANFDB_BASE, &canInit);

//
//Set CAN filter
//Accept frames with extended ID 0x1A1B1C1x and standard ID 0x41x.
//
CAN_setAcceptFilter(CANFDB_BASE, CAN_ACF3, CAN_ID_STD_EXT, 0x1A1B1C1DUL, 0x0000000FUL);

//
// 500K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CAN_setBitTimingSlow(CANFDB_BASE, 3, 38, 9, 9);
//
// 2000K bundrate when can pclk = 100Mhz, sample point = 80%
// (bundrate = pclk/s_prescale/(s_seg1+2 + s_seg2+1))
//
CANFD_setBitTimingFast(CANFDB_BASE, 1, 18, 4, 4);

//
// CAN enters normal communication mode.
//
CAN_disableStatReset(CANFDB_BASE);
```

16.6.4. Rx FIFO

QX CANFD和TI CAN的接收FIFO的工作机制大致相同，主要差别在于TI支持最大FIFO深度为32，QX CANFD的FIFO深度为8。

QX CANFD模块提供了提供 8 个 SLOT 的接收缓冲器用于存储接收到的数据，该 8 SLOT 的接收缓冲器工作在 FIFO 模式。RB SLOT 通过 RBUF 寄存器来读取接收到的数据，总是最先读取最早接收到的数据，并通过 RCTRL 寄存器的 RREL 设置为 1 释放已经读取的 RB SLOT，并指向下一个RB SLOT。

通过 RBUF 读取 RB SLOT 示意图如下。



FIFO 有两种模式，可根据 FIFO 已满时接收到新消息时的行为进行区分。第一种是 FIFO 阻塞模式，这种模式下当Rx FIFO 已满时，Rx FIFO 不再接收新消息，除非当前存储的至少一条消息已被应用读取。第二种是 FIFO 覆盖模式，当Rx FIFO已满时，下一条接受的消息将覆

盖最早的 FIFO 消息。这两种模式下，若出现BUF上溢，RCTRL.ROV标志位会置起，并且可设置RTIE.ROIE位来使能BUF上溢中断。

为避免由于 FIFO 已满而导致丢失数据，可以通过设置LIMIT.AFWL位来设置水线，并且可设置RTIE.RAFIE位来使能BUF达到水线中断。

17. I2C

1. TI 2800137的I2C模块不支持DMA传输，乾芯2800137额外新增了DMA模块并支持I2C作为触发源，I2C的DMA使用方法详见参考手册I2C和DMA章节。
2. TI 2800137 XRDY标志位为1表示发送准备已完成（为0表发送准备未完成）。乾芯2800137此标志位和TI 2800137含义相反，为1表示发送准备未完成（为0表发送准备已完成）。

18. SCI

1. 乾芯2800137的SCI波特率配置支持增强的小数配置，在一些速度较高且希望降低误码率的场景下可以选择启用小数波特率配置以提高通信的稳定性。
2. 乾芯2800137的SCI支持额外的水位接收超时中断，该中断可以简化通信的控制逻辑。用户还可以选择关闭TI原生的WAKEUP中断以简化串口的接收逻辑。
3. 乾芯2800137的SCI支持DMA传输，该特性可以简化SCI通信的控制逻辑并降低CPU的控制负担。
4. 乾芯2800137的SCI模块波特率配置寄存器SCILBAUD最小有效值为3（可正常覆盖常用波特率范围）。
5. 乾芯2800137的SCI，在使用RXST寄存器时要注意，当OE、PE、FE、BRK同时触发时只能触发其中的2个标志位，TI可以全触发对应的错误标志位。

19. SPI

1. 乾芯2800137 SPI的波特率最高支持到LSPCLK时钟的8分频速率，TI 最高支持到LSPCLK的4分频。
2. 乾芯2800137 SPI的TXBUF寄存器发送的数据默认为右对齐，TI 默认为左对齐。

20. EPG

1. 乾芯2800137的EPG在PRD寄存器分频值为0时实际分频值为2，TI此时为1。
2. 乾芯2800137中EPG的GCTRL0寄存器，其中的EPGOUT0SEL~EPGOUT7SEL控制位，写时和TI一致是8~15bit，读操作时EPGOUT0SEL~EPGOUT7SEL对应为1~8bit（TI对应读时对应8~15bit），该差异在使用bit位域操作时需格外注意。

21. DMA

TI 2800137无DMA模块，乾芯2800137添加了2个通道的DMA模块支持，DMA的使用方法详见参考手册DMA章节。