QXS320F280049RevB差异手册

v1.0.3

合肥乾芯科技有限公司

版本历史

版本号	日期	备注
0.1.0	2025.08.20	初稿
1.0.0	2025.08.30	发布版本
1.0.1	2025.09.15	内容更新
1.0.2	2025.10.10	添加FSI章节说明
1.0.3	2026.03.12	更新EPWM相移PHSDIR差异

目 录

1.概述.. 7

2.CPU核.. 7

2.1.指令集及生态.. 7

3.BOOT.. 7

4.SYSCTL.. 7

4.1.电源管理.. 7

4.2.系统复位.. 8

4.3.低功耗模式.. 8

4.4.内存控制.. 8

4.5.中断控制.. 8

4.6.WD看门狗.. 10

4.7.TIMER定时器.. 10

4.8.NMI.. 12

4.9.XINT.. 12

4.10.4.10. DMA_CLA_SRC_SEL.. 12

4.11.DEV_CFG.. 12

4.12.CLK_CFG.. 13

4.13.CPU_SYS.. 13

4.14.PERIPH_AC.. 14

4.15.DCSM..14
4.16.ACCESS_PROTECTION...14
4.17.MEMORY_ERROR..14
4.18.FLASH_CTRL...14
4.19.FLASH_ECC..14
4.20.FLASH_OTP..15
4.21.UID...15
5.CLA..15
6.ADC..15
7.EPWM...16
7.1.TB..16
7.2.CC..17
7.3.AQ..17
7.4.DB..17
7.5.PC..17
7.6.TZ..17
7.7.ET..18
7.8.DC..18
7.9.GLD..18
7.10.LOCK...18
8.HRPWM...18
8.1.精度..18

8.2.控制模式......19
8.3.相移控制......20
9.ECAP......20
10.EQEP......20
11.CMPSS......20
12.GPIO......21
13.ANASUBSYS......21
14.SCI......21
15.SPI......22
16.I2C......22
17.CAN......22
17.1.差异对比......23
17.2.模块初始化......24
17.3.位速率配置......27
17.4.中断处理......28
17.5.发送数据......33
17.6.接收数据......35
18.DMA......45
18.1.差异对比......45
18.2.模块初始化......46
18.3.DMA触发源......47
18.4.中断处理......49

19.DAC.. 49

20.PGA.. 49

21.FSI.. 50

1. 概述

本文档是QX 280049和友商TI 280049的差异手册，目的是方便移植TI C2000 DSP应用程序到QX DSP。

2. CPU核

2.1. 指令集及生态

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. BOOT

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ram后再跳转到ram执行。因此乾芯的代码运行总是在ram进行（调试时IDE会直接将程序下载到ram来完成调试运行）。

4. SYSCTL

4.1. 电源管理

无差异。

4.2. 系统复位

无差异。

4.3. 低功耗模式

1. 乾芯280049额外支持STANDBY模式，该模式将自动关闭所有外设时钟以降低功耗。

4.4. 内存控制

1. 乾芯280049支持双核指令RAM和数据RAM空间大小的任意划分，但和TI的MEM_CFG寄存器组完成的内存控制在配置上存在差异。内存配置的具体使用方法参见参考手册相关章节描述。

4.5. 中断控制

1. 乾芯280049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. TI的PIECTRL[ENPIE]位用于开启外设的中断全局支持，乾芯280049的外设中断总是默认开启支持，无ENPIE操作位。TI的PIECTRL[PIEVECT]位表示从PIE向量表中获取的向量地址，乾芯280049不支持PIEVECT中断向量地址的获取操作。
3. 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：

```
1  __interrupt void XXXXX(void)
2  {
3         软件保存现场          （工具链）
4         返回PC压栈            （软件）
5         嵌套优先级修改        （软件）
6         EINT                  （软件）
7         用户程序              （软件）
8         DINT                  （软件）
9         返回PC出栈            （软件）
10        软件恢复现场          （工具链）
11 }
```

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 }
```

4.6. WD看门狗

无差异。

4.7. TIMER定时器

1. 乾芯280049不支持TIMER的FREE RUN模式，对应TCR[SOFT]和TCR[FREE]位域无实际功能。

4.8. NMI

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）。

4.9. XINT

1. 乾芯DMA额外支持XINT通过外部GPIO信号触发DMA功能，对应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 };
```

4.10. DMA_CLA_SRC_SEL

乾芯280049无独立的CLA模块，不支持TI的DMA_CLA_SRC_SEL寄存器。

4.11. DEV_CFG

1. 乾芯280049没有PARTIDL、PARTIDH、REVID、DC21、FUSEERR寄存器，不支持相应的功能特性。
2. 乾芯280049新增一组HARDWARERESx寄存器用于控制复位和EALLOW的控制权属于CPU1还是CPU2。

```
01      uint32 rsvd1[50];                          // offset:0x13C~0x78
02      union HARDWARERES0_REG HARDWARERES0;       // offset:0x140
03      union HARDWARERES2_REG HARDWARERES2;       // offset:0x144
04      union HARDWARERES3_REG HARDWARERES3;       // offset:0x148
05      union HARDWARERES4_REG HARDWARERES4;       // offset:0x14C
06      union HARDWARERES6_REG HARDWARERES6;       // offset:0x150
07      union HARDWARERES7_REG HARDWARERES7;       // offset:0x154
08      union HARDWARERES8_REG HARDWARERES8;       // offset:0x158
09      union HARDWARERES9_REG HARDWARERES9;       // offset:0x15C
10      union HARDWARERES10_REG HARDWARERES10;     // offset:0x160
11      union HARDWARERES13_REG HARDWARERES13;     // offset:0x164
12      union HARDWARERES14_REG HARDWARERES14;     // offset:0x168
13      union HARDWARERES15_REG HARDWARERES15;     // offset:0x16C
14      union HARDWARERES16_REG HARDWARERES16;     // offset:0x170
15      union HARDWARERES19_REG HARDWARERES19;     // offset:0x174
```

乾芯280049不支持TAP_STATUS寄存器，对应JTAGS的状态由一组内部寄存器管理。

4. 乾芯280049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280049额外添加了FLASHCLKDIV寄存器用于控制FLASH_CTRL模块的时钟分频。

4.12. CLK_CFG

1. 乾芯280049的PLL倍频和分频的计算公式为：

   SYSCLK = OSC_CLK * IMULT / ODIV / SYSCLKDIVSEL

   其中OSC_CLK表示输入的时钟源时钟频率，IMULT和TI的整数倍频系数IMULT含义一致均表示PLL的输入倍频，但乾芯280049要求 OSC_CLK * IMULT 后的时钟必须在400~800Mhz之间。如果OSC_CLK为典型的10Mhz，则要求IMULT参数在40和80之间（同时需要注意IMULT的最大可配置值为63），ODIV以及SYSCLKDIVSEL则与TI含义一致。

4.13. CPU_SYS

1. 乾芯280049不存在CPUSYSLOCK1和PIEVERRADDR寄存器，同时不支持相应的功能特性。
2. 乾芯280049在架构上是双核架构，且没有独立CLA模块，因此存在额外的CpuSysRegs.PCLKCR0.CPU2位域用于控制CPU2的时钟开关。同时不存在CpuSysRegs.PCLKCR0.CLA1开关。
3. 乾芯280049在架构上是双核架构，每个CPU存在3个TIMER定时器，相对于TI 280049第二个核多出的3个TIMER的时钟开关用CpuSysRegs.PCLKCR0.CPU2TIMERx来控制。
4. TI 280049存在FSITX、FSIRX、DCC_0模块独立的时钟开关，乾芯280049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）。

5. 乾芯280049的复位源中没有SCCRESETn项。同时由于在架构上是双核，额外存在CPU2_POR、CPU2_XRSn、CPU2_WDRSn、CPU2_NMIWDRSn、CPU2_DBGRSn五个复位源用于表示第二个核对应的复位源。

4.14. PERIPH_AC

乾芯280049不支持TI的PERIPH_AC寄存器组。所有模块的寄存器默认均为可访问权限。

4.15. DCSM

1. 乾芯280049 DCSM的ZSB数量为2，对应TI位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280049无CRCLOCK功能。
3. 乾芯280049 DCSM的ZxOTP_CSMPSWD1、PSWDLOCK初始值为0xFFFFFFFF。
4. 乾芯280049 GPREG寄存器设置BOOT模式。

4.16. ACCESS_PROTECTION

乾芯280049不支持TI的ACCESS_PROTECTION特性。

4.17. MEMORY_ERROR

乾芯280049暂不支持TI的MEMORY_ERROR特性。

4.18. FLASH_CTRL

1. 乾芯280049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。

4.19. FLASH_ECC

1. 乾芯280049使用内嵌qspi flash工艺，目前仅支持对flash存储的数据做完整性的CRC校验。

4.20. FLASH_OTP

1. 乾芯280049存在两个各自独立的OTP区域（每个OTP有1024个字节空间）。第一个OTP

区用于芯片内部多个模块的出厂矫正参数的只读性存储。第二个OTP区用户可以任意使用，OTP有一个熔断机制，熔断操作之前OTP区可读可写，可以当做正常的flash区域使用。熔断操作之后则变为只读，用户可以借用这个熔断机制实现应用级别的只读参数存储等功能。

4.21. UID

乾芯280049无UID寄存器。

5. CLA

乾芯280049没有独立的CLA模块，两个独立的CPU均支持全量的CLA加速运算指令。用户可以利用第二个CPU等价实现所有CLA的运算操作。CLA和CPU之间的数据交互可以通过双核之间的IPC通信完成。

6. ADC

1. 乾芯280049的ADC多个ADC模块不可同时接入使用同一个输入信号，一个独立的ADC模块如果没有使用，建议不要将该模块的时钟开启。否则，在开启时钟的情况下，该ADC模块的默认输入复用会影响到其它用到相同输入的ADC模块。
2. 乾芯280049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：

				13h：ADCIN19
11-8	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

如果单个ADC模块用且仅使用了SOC0+SOC1作为启动转换配置，则需要额外加一行软件代码才能正常使用:

```
EALLOW;
```

AdcaRegs.ADCSOCPRICTL.bit.SOCPRIORITY = 2;

4. 乾芯280049的ADCCTL2[PRESCALE]寄存器用于配置ADC时钟分频系数，默认值为6，表示6分频。注意该配置最低2分频（0、1、2均表示2分频），最大支持31分频。TI为1-8分频，默认1分频。
5. 乾芯280049的ADC的ADCCLK最支持最大到50Mhz。
6. 乾芯280049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280049的5个12bit ADC和3个14bit HR ADC（可配置为8,10,12,14bit），满足不同的需求。
8. 乾芯280049的ADC内部参考电压提供1.65V、2.0V、2.5V和3.3V四种可选。TI仅支持2.0V和3.3V。

7. EPWM

1. TI 280049提供8个独立的EWPM模块，乾芯280049提升到独立的12个。

7.1. TB

1. 乾芯280049在相位功能开启时（TBCTL[PHSEN]=1），在上下模式下TBCTL[PHSDIR]必须配置设置为1，否则，当TBPHS=0时EPWM不发波。TI 280049此种情况即便下TBCTL[PHSDIR]=0也可以发波。
2. 乾芯280049在用强制同步时，同步后没有相位差，TI有一个计数周期的相位差。
3. 乾芯280049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。
4. 乾芯280049支持TI type5类型EPWM的任意多通道同步特性（新增EPWMSYNCINSEL寄存器），TI 280049 EPWM为Type4类型，仅支持有限的多路组合同步类型。
5. EPWMSYNCINSEL寄存器属性为只写属性，读总是返回0。

7.2. CC

1. CMPD寄存器属性为只写属性，读总是返回0。
2. 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。

7.3. AQ

1. 乾芯280049的AQSFRC的强制翻转只支持在TBCLK不分频的情况下进行配置，TI则为任意分频下均可配置。

7.4. DB

1. 乾芯280049在使用死区半周期时，FED必须大于等于2，TI没有该限制。
2. 乾芯280049在使用死区时，如果一路信号经过延时，另一路bypass则死区输出的AB两波有一个计数周期的相位差，TI相同情况下两路波没有相位差。2.
3. 乾芯280049死区封波不支持使用延时RED和FED的方式封波，TI支持此配置方法。

7.5. PC

1. PC在配置首次脉宽时结果和理论值有一个计数周期的波动误差。

7.6. TZ

无差异。

7.7. ET

无差异。

7.8. DC

1. 乾芯280049的DC的BLANKWINDOW功能只支持BLANKWINDOW小于等于PRD，TI无此限制。
2. 乾芯280049的当DCWINDOW和OFFSET叠加使用时消隐窗口不能跨越两个周期。
3. 乾芯280049的DC滤波在使用时不能分频。
4. 乾芯280049的DC滤波个数不能配置为0，必须配置为大于等于1的值。

7.9. GLD

1. 乾芯280049的TBPRDHR寄存器不支持全局加载。

2. 乾芯280049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。
3. 乾芯280049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。

7.10. LOCK

1. 乾芯280049不支持EPWMLOCK功能。

8. HRPWM

8.1. 精度

1. 乾芯280049的HRPWM MEP精度在110ps，对应TI在150ps。
2. 乾芯280049的HPRWM的MEP的每一步单步步进精度稍差（TI精度在30ps以内，乾芯精度在100ps以内）。
3. 乾芯280049的HRPWM每一组的A路和B路之间的误差在1.5ns以内，对应TI误差在500ps以内，（注意：该误差为芯片内部走线的固定误差，不同组之间A路和B路误差总是存在且固定）。

8.2. 控制模式

1. 乾芯280049的HRPWM在开启了高精度周期控制（HRCNFG2[HRPE]=1）时仅支持上下模式（TBCTL[CTRMODE]=2），向上模式存在1个周期的抖动。
2. 乾芯280049的HRPWM在开启了高精度周期控制（HRCNFG2[HRPE]=1）时AQ动作仅支持CAU/CBU=CLEAR + CAD/CBD=SET组合，CAU/CBU=SET + CAD/CBD=CLEAR组合存在抖动。
3. 乾芯280049的每一路EPWM在启用HRPWM特性时都需要配置对应该路的HRPWR[CALPWRON]=1，TI 280049仅需要设置第一路的HRPWR[CALPWRON]=1。
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.

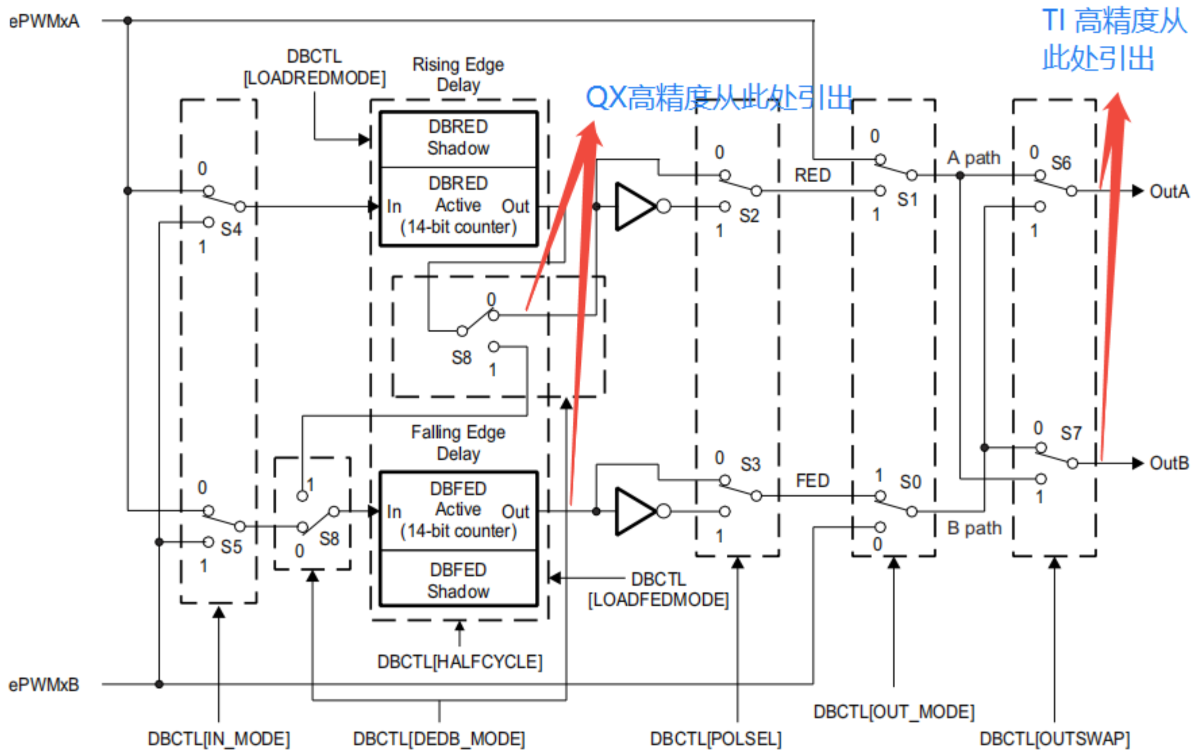


Figure 13-35. Configuration Options for the Dead-Band Submodule

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. 在使用高精度SFO时需要使能HRPWR[CALPWRON]寄存器，乾芯该寄存器的使能仅针对当前epwm通路，而TI的芯片只需要使能第一路后所有epwm通道的SFO均开启。
6. 在使用高精度周期延时时，只支持影子模式，暂不支持使用立即模式。
7. 在使用全局加载时TBPRDHR暂不支持全局加载方式。

8.3. 相移控制

1. 高进度相移控制中通过软件同步的方式(TBCTL[SWFSYNC]=1)实现多路相移同步时，TI发起同步的EPWM通道和被同步的EPWM通道之间存在一个周期的相位差，乾芯280049 不存在这一个周期的相位差。

9. ECAP

1. 乾芯HRCAP在使用连续采集模式时需在HRCAP校准中断中进行REARM重新触发控制以保证HRCAP模块连续采集。

10. EQEP

无差异。

11. CMPSS

1. 乾芯280049比较器提供7种迟滞电压配置，TI提供了4种。
2. 如下图箭头所示位置，TI使用SYSCLK作为谐波发生器的递减处理时钟。乾芯该递减时钟存在必须小于等于25Mhz的要求，因此乾芯280049存在两个额外的分频寄存器COMPDACHCTL[DACH_PRESCALE]和COMPDACLCTL[DACL_PRESCALE]分别用于高端比较器和低端比较器的DAC时钟分频配置（默认值为100），用户首先需要根据系统时钟配置合适的分频值以满足分频后不高于25Mhz的要求。假设该分频配置值为n，那么斜坡发生器的RAMPDLYS寄存器的配置需要是原来的n倍才能和TI原有逻辑功能保持一致。

GPIO

1. TI 280049 GPIO的驱动能力在4ma左右，乾芯280049 GPIO的驱动能力提升到16ma。

13. ANASUBSYS

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。
3. 乾芯DCDCCTL寄存器增加ADJ_DCDC和MD_LXDET配置位，详情见用户参考手册。

14. SCI

1. 乾芯280049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280049的SCI支持额外的总线空闲超时中断，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280049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280049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280049的LIN支持复用为第三路SCI_C，在使用SCI_C时需要注意SCI_A和SCI_B的时钟是低速外设时钟（LSP_CLK），而SCI_C模块的时钟是系统时钟（SYS_CLK）。该差异在低速分频不为1分频（ClkCfgRegs[LOSPCP] != 0）时需要注意SCI_C的波特率计算时传递时钟频率应该系统时钟而不是应该LSP低速时钟，否则将导致SCI_C的波特率计算错误。

15. SPI

1. 乾芯280049 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280049 SPI的TXBUF寄存器发送的数据默认为右对齐，TI 默认为左对齐。

16. I2C

1. 乾芯280049 I2C FIFO深度为15，对应TI FIFO深度为16。
2. 乾芯280049 I2C 不支持repeat模式，即: RM=1时，重复模式，暂不支持。

3. TI 280049 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280049此标志位和TI 280049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）。
4. 乾芯280049 I2C控制清除接收 Rx 中断标志位，操作前需先关闭FIFO才能正常清除，典型的清除中断操作流程如下：

```
I2C_disableFIFO(I2CA_BASE);
I2C_clearInterruptStatus(I2CA_BASE, I2C_INT_RXFF);
I2C_enableFIFO(I2CA_BASE);
```

17. 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。

17.1. 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2-1 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）	只需配置一种速率

	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	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	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

17.2. 模块初始化

17.2.1. 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）。

17.2.2. 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）

17.2.3. 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

```c
}

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

```
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

17.3. 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。

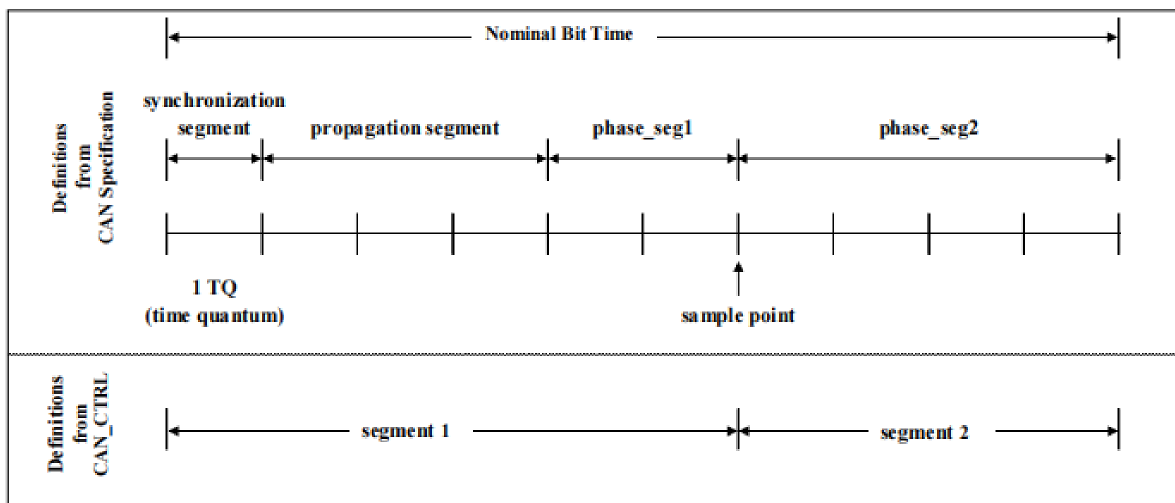


图3-3 CAN位时间定义图

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);

17.4. 中断处理

17.4.1. 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 5-1 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5-2 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

17.4.2. 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。

```
can_ex1_loopback.c   board.c   Resource Explorer   can_ex2_loopback_interrupts.c   interrupt
    //
    // Initialize PIE and clear PIE registers. Disables CPU interrupts.
    //
    Interrupt_initModule();

    //
    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    //
    Interrupt_initVectorTable();

    //
    // Enable Global Interrupt (INTM) and realtime interrupt (DBGM)
    //
    EINT;
    ERTM;

    //
    // Interrupts that are used in this example are re-mapped to
    // ISR functions found within this file.
    // This registers the interrupt handler in PIE vector table.
    //
    Interrupt_register(INT_CANA0, &canISR);

    //
    // Enable the CAN interrupt signal
    //
    Interrupt_enable(INT_CANA0);
    CAN_enableGlobalInterrupt(CANA_BASE, CAN_GLOBAL_INT_CANINT0);

    //
    // Enable CAN test mode with external loopback
    //
    CAN_enableTestMode(CANA_BASE, CAN_TEST_EXL);

    //
    // Initialize the transmit message object used for sending CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 1
    //      Message Identifier: 0x1
    //      Message Frame: Standard
    //      Message Type: Transmit
    //      Message ID Mask: 0x0
    //      Message Object Flags: Transmit Interrupt
```

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。

```
57 //
58 // CAN A ISR - The interrupt service routine called when a CAN interrupt is
59 //             triggered on CAN module A.
60 //
61 __interrupt void
62 canaISR(void)
63 {
64     uint32_t status;
65 
66     // Read the CAN-A interrupt status to find the cause of the interrupt
67 
68     status = CAN_getInterruptCause(CANA_BASE);
69 
70     // If the cause is a controller status interrupt, then get the status
71 
72     if(status == CAN_INT_INT0ID_STATUS)     // Read the controller status.
73     {
74         status = CAN_getStatus(CANA_BASE);
75 
76         // Check to see if an error occurred.
77 
78         if(((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_MSK) &&
79            ((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_NONE))
80         {
81             errorFlag = 1;        // Set a flag to indicate some errors may have occurred.
82         }
83     }
84     else if(status == TX_MSG_OBJ_ID)
85     {
86         // Transmit Message handling will go here
87 
88         CAN_clearInterruptStatus(CANA_BASE, TX_MSG_OBJ_ID);  // Clear the message object interrupt
89     }
90     else if(status == RX_MSG_OBJ_ID)
91     {
92         // Receive message handling will go here
93 
94         CAN_clearInterruptStatus(CANA_BASE, RX_MSG_OBJ_ID);  // Clear the message object interrupt
95     }
96
97     else
98     {
99         //
00         // Spurious interrupt handling can go here.
01         //
02     }
03 
04     //
05     // Clear the global interrupt flag for the CAN interrupt line
06     //
07     CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0);
08 
09     //
10     // Acknowledge this interrupt located in group 9
11     //
12     Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);
13 }
```

17.4.3. 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
```

17.5. 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。

17.5.1. 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

17.5.2. 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.
    //
```


17.6. 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。

17.6.1. 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         //
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
```

通过中断的方式接收数据

17.6.2. 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
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.
```

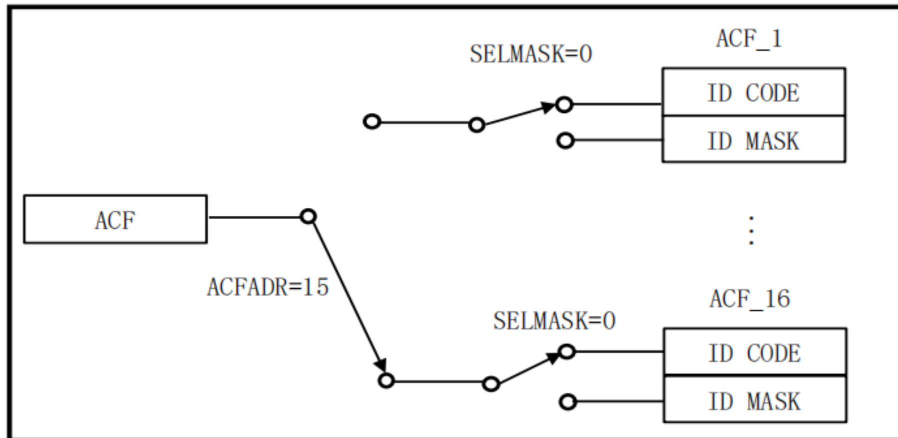
初始化中设置滤波器组

```
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

17.6.3. 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

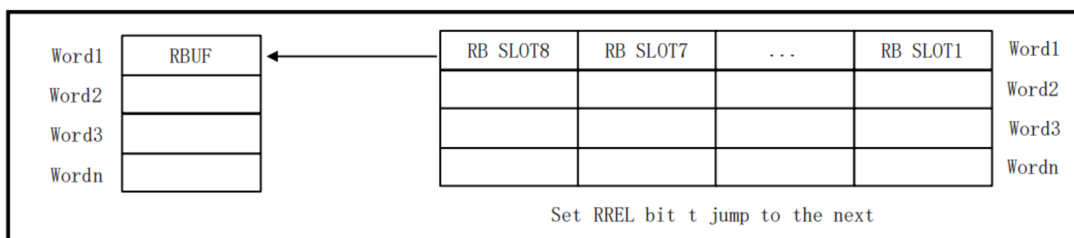
```
// enable module clock
//
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANB);

//
//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);
```

17.6.4. 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。

18. DMA

本章节介绍如何从典型的TI C2000系列的DMA模块移植代码到乾芯 280049系列芯片的DMA模块的移植过程与注意事项。本文档列出了 TI DMA和 乾芯DMA模块之间的主要差异。然后，本文档继续着重介绍如何在这两个模块中执行常见操作。

18.1. 差异对比

TI DMA 和 乾芯DMA模块 功能描述对比

功能	TI DMA	乾芯 DMA
传输位宽	TI DMA通过DmaChxRegs. MODE. DATASIZE设置单次DMA传输的位宽（0=16bit，1=32bit）	乾芯DMA中通过DmaChxRegs.CTL_L.SRC_WIDTH和DmaChxRegs.CTL_L.DST_WIDTH 可分别设置源和目的地的单次传输位宽（0=8bit，1=16bit，2=32bit，对应DMA_TransferWidth枚举定义）
Brust长度	Burst是一次DMA传输中的最小触发单元。每当DMA通道触发时进行的数据传送。这个数量称为Burst长度，通过BURST_SIZE寄存器配置（Burst长度=BURST_SIZE+1），支持1-16任意长度配置。	乾芯DMA中通过 DmaChxRegs.CTL_L.SRC_MSIZE和 DmaChxRegs.CTL_L.DST_MSIZE 两个寄存器配置源和目的地址的Burst长度，乾芯DMA的Bust长度不可任意配置，仅支持下列长度配置：1、4 、8 、16 、32 、64 、128 、256、512（对应DMA_BurstLength枚举定义）
Brust步进	在burst传输过程中，每传输一个字，源地址和目的都会增加一个Burst步长（BURST_STEP）。这个步长是个16位的有符号整数，也就是说，步长既可以是正数，也可以是负数，范围为-4096~+4095. 源地址步长和目的地址步长是分别可设的，对应着2个寄存器：SRC_BURST_STEP和DST_BURST_STEP。	乾芯DMA暂不支持Burst步进配置，一个Burst传输中的多个单次传输之间的步进只能为0或1（0表示地址不增长，1表示地址连续增长）。
Transfer长度	一次完整的DMA数据传输称为transfer，可以包含多个burst. 所包含的burst数量称为transfer数量，通过TRANSFER_SIZE配置，TRANSFER_SIZE可以配置为任意16bit位宽数值。	乾芯DMA通过DmaChxRegs.BLOCK_TS配置Transfer长度，最大长度支持到512（BLOCK_TS=Transfer长度-1），如果需要超过512长度的传输需要通过DMA完成中断中再次启动DMA通道来完成。
Transfer步进	与Burst类似，Transfer也有传输步长，每传输一次Burst，源地址和目的都会增加一个TRANSFER步长（TRANSFER_STEP），Transfer步进通过（SRC_TRANSFER_STEP和DST_TRANSFER_STEP）完成配置。	乾芯DMA暂不支持Transfer步进配置，多个Burst传输之间的步进仅只能为0或1（0表示地址不增长，1表示地址连续增长）。

CONTINUOUS模式	禁止连续模式时（MODE.CONTINUOUS=0），当启动DMA传输后，只完成一次transfer。完成后RUNSTS自动清零。要开启下一次transfer，需要再次启动DMA传输（RUN写入1）。 使能连续模式时（MODE.CONTINUOUS=1），一旦启动DMA，则RUNSTS一直为1，DMA通道一直在运行（也要等待外设事件触发才会进行数据传输），直到HALT写入1才停止运行。	乾芯DMA目前仅支持单块传输模式，一个Block块等同于TI的一个Transfer，因此当BLOCK_TS配置的总长度传输完成后，DMA通道将关闭（产生DMA_DONE完成中断）。如果希望实现TI DMA的连续传输模式，需要在DMA完成中断处理函数中适当的配置当前DMA通道的源和目的地址并再次启动当前DMA通道。
ONESHOT模式	禁止单次模式时（MODE.ONESHOT=0），每一次burst都需要外设事件触发。如果transfer中包含N个burst，则需要N个外设事件的触发脉冲，才能完成一次DMA传输。使能单次模式时（MODE.ONESHOT=1），只要一个外设事件的触发脉冲，就可以完成一次transfer中的N个burst。	乾芯DMA没有单独的ONESHOT模式配置。如果希望一个DMA触发事件完成所有传输，可以选择将SRC_MSIZE和DST_MSIZE长度和BLOCK_TS配置为相同长度。
WARP回绕	TI DMA提供了一种回绕机制。源地址和目的地址可以分别配置WRAP_SIZE和WRAP_STEP。WRAP_SIZE对应的是Burst数量。当完成WRAP_SIZE个Burst传输时，就发生地址WRAP。此时，在起始地址的基础上增加WRAP_STEP，重新使用该地址作为下一次Burst传输的地址。	乾芯DMA没有单独的WARP模式配置。如果希望完成类似效果，需要在DMA完成中断处理函数中适当的配置当前DMA通道的源和目的地址并再次启动当前DMA通道。
SHADOW影子寄存器	TI DMA通过在程序运行过程中动态的配置SRC_ADDR_SHADOW和DST_ADDR_SHADOW可以实现DMA多个Transfer之间源和目的地址的动态修改。	乾芯DMA暂不是支持影子寄存器特性。

18.2. 模块初始化

TI DMA和乾芯 DMA在寄存器布局上存在很大差异，为了方便客户移植，驱动层面乾芯DMA和TI DMA做了一些相似的接口。因此，建议客户通过driverlib驱动库中的DMA模块函数实现业务流程的控制以获得更加方便的移植操作。TI DMA模块的初始化接口主要是下函数接口：

static inline void DMA_initController(void)；

void DMA_configChannel(uint32_t base, const DMA_ConfigParams *config)；

初始化时一般会定义一个DMA_ConfigParams结构体并通过对该结构体初始化参数来实现初始化，下面是典型的初始化流程调用并做了相应注释（DMA_ConfigParams在名称上和TI一致，但目前支持的配置比TI要少）：

```
void DMA_init()
{
    DMA_ConfigParams dconfig = { 0 };

    SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_DMA); // 启动DMA模块时钟

    DMA_initController(); // DMA模块初始化
    // DMA channel 1 配置
    dconfig.srcAddr         = (u32)send_frame;
    dconfig.destAddr        = (u32)&SciaRegs.THR;
    dconfig.srcWidth        = DMA_TRANS_WIDTH_8;  // 设置DMA源数据位宽为8bit
    dconfig.destWidth       = DMA_TRANS_WIDTH_8;  // 设置DMA目的数据位宽为8bit
    dconfig.srcMsize        = DMA_MSIZE_1;        // 设置DMA一次传输请求传输一个字节
    dconfig.destMsize       = DMA_MSIZE_1;        // 设置DMA一次传输请求传输一个字节
    dconfig.blockSize       = 99;                 // 一次DMA 块传输总长度(总长度=blockSize+1)
    dconfig.srcHSInterface  = DMA_TRIGGER_MEM;    // 设DMA源触发器
    dconfig.destHSInterface = DMA_TRIGGER_SCIATX; // 设DMA目的触发器
    DMA_configChannel(DMA_CH1_BASE, &dconfig);
    DMA_enableInterrupt(DMA_CH1_BASE, DMA_IRQ_BLOCK_TRF); // 开启dma done中断
    DMA_startChannel(1); // 启动DMA通道
}
```

18.3. DMA触发源

乾芯在DMA的触发源列表上和TI 基本保持一致，触发器通过DMA_ConfigParams结构体的srcHSInterface和destHSInterface传递参数并配置。值得注意的是乾芯DMA实际上除了触发源需要配置外还需要配置触发源的类型以及地址的是否增长。以下是DMA_configMode函数的封装截图：

```
void DMA_configMode(uint32_t base, DMA_Trigger srcTrigger, DMA_Trigger destTrigger)
{
    volatile struct DMA_CH_REGS *ch_reg = (volatile struct DMA_CH_REGS *)base;

    ch_reg->CFG2_H.bit.HS_SEL_SRC = 0;
    ch_reg->CFG2_L.bit.SRC_PER    = srcTrigger;
    ch_reg->CFG2_H.bit.HS_SEL_DST = 0;
    ch_reg->CFG2_L.bit.DST_PER    = destTrigger;

    if ((srcTrigger == 0) && (destTrigger == 0))
    {
        ch_reg->CFG2_H.bit.TT_FC = DMA_M2M_DMA;
        ch_reg->CTL_L.bit.SINC   = 0;
        ch_reg->CTL_L.bit.DINC   = 0;
    }
    else if ((srcTrigger == 0) && (destTrigger != 0))
    {
        ch_reg->CFG2_H.bit.TT_FC = DMA_M2P_DMA;
        ch_reg->CTL_L.bit.SINC   = 0;
        ch_reg->CTL_L.bit.DINC   = 1;
    }
    else if ((srcTrigger != 0) && (destTrigger == 0))
    {
        ch_reg->CFG2_H.bit.TT_FC = DMA_P2M_DMA;
        ch_reg->CTL_L.bit.SINC   = 1;
        ch_reg->CTL_L.bit.DINC   = 0;
    }
    else
    {
        ch_reg->CFG2_H.bit.TT_FC = DMA_P2P_DMA;
        ch_reg->CTL_L.bit.SINC   = 1;
        ch_reg->CTL_L.bit.DINC   = 1;
    }
```

其中CFG2_H.TT_FC用于配置DMA的操作类型，DMA触发器列表的第一个触发源固定用于内存，其它都为外设，因此这里通过判断触发源是否为0来自动配置TT_FC参数。在乾芯DMA中内存是一个特殊的触发器，他被定义为无需DMA握手总是准备好的一种触发器，因此如果TT_FC被设置为DMA_M2M_DMA，则当通道开启时，DMA操作将立刻执行，该效果和TI的DMA_TRIGGER_SOFTWARE在功能上是一致的。

另外还需要特别注意的是DMA操作的源或目的触发器和DMA搬运的源和目的地址可以是不同的模块（地址是内存时触发器需要总是为DMA_TRIGGER_MEM）。例如，一个触发器为TIMER的DMA传输，他的源和目的可以是任意其它模块的地址，这样就可以实现通过TIMER完成任意模块之间的数据搬运工作。

CTL_L.SINC和CTL_L.DINC用于配置源地址和目的地址是否递增，0表示递增，1表示不递增。一般情况下，内存的地址总是需要递增的，而外设地址一般不需要递增，因此这里默认根据触发源和目的地址的类型做了自动化配置。用户如果有特殊需求，可以额外对CTL_L.SINC和CTL_L.DINC做单独的配置。

18.4. 中断处理

乾芯 DMA的中断类型一般情况下我们关注最多的是DMA完成中断和DMA的各各种错误中断。由于目前版本的DMA中仅支持单块传输模式，因此如果希望DMA完成超过512个字的搬运或持续不断的搬运操作，则需要借助DMA的完成中断实现接续。下面是典型的DMA在完成中断处理函数中重新配置当前DMA源和目的地址后重新启动DMA通道的操作：

```
__interrupt void dmach1ISR(void)
{
    u32 status = DmaCh1Regs.INTSTATUS;
    if (status & DMA_IRQ_DMA_TRF)
    {
        DmaCh1Regs.DAR = (u32)recv_frame;
        DMA_startChannel(1);
    }
}
```

19. DAC

1. 乾芯280049 DAC通过DACCTL[PRESCALE]寄存器提供1~255时钟分频，TI无该配置。
2. 乾芯280049 DAC的DACCTL[MODE]值为2时表示BYPASS模式，TI无BYPASS模式。
3. 乾芯280049 DAC存在DACCTL[VREF_MD]配置，TI无该配置。
4. 乾芯280049 DAC的DACCTL[DACREFSEL]为2bit，值为2或3时表示参考电压为高阻态，TI对应该位为1bit，无高阻态配置。

20. PGA

1. 乾芯280049支持最大48倍放大倍数，TI最大支持到24倍。
2. 乾芯280049通过PGACTL[FILTREEN]寄存器实现PGA滤波功能的开关，TI则通过设置PGACTL[FILTRESSEL]=0来关闭滤波功能。
3. 乾芯280049的PGACTL[FILTRESSEL]配置的滤波电阻大小的计算公式为 滤波电阻=800 Ω/FILTRESSEL)。TI 280049的PGACTL[FILTRESSEL]配置在含义上是一组离散的固定电阻值。

21. FSI

乾芯280049RevB版本不支持FSI。该模块功能处于在研状态，后续乾芯280049芯片迭代版本

有计划添加FSI支持。