Application Brief

How to Debug Interrupt Abnormalities

Ryan Ma, Shaoxing Ke

Introduction

Supporting real-time tasks on a CPU requires the use of interrupts. If an external sensor senses a fault, the CPU needs to be interrupted or halted to perform a subroutine that is able to handle the fault. In this example, timing of the interrupt of when the signal reaches the CPU matters. Interrupts are hardware or software-driven signals that cause the CPU to suspend the current program sequence and execute a subroutine. Interrupts often handle time critical loops and control algorithms that are critical to the application and need to execute in timely fashion. Most of the case interrupts can happen periodically with a known frequency. However, when designing the software architecture, have you ever seen an interrupt waveform oscillate incorrectly, as shown in Figure 1?

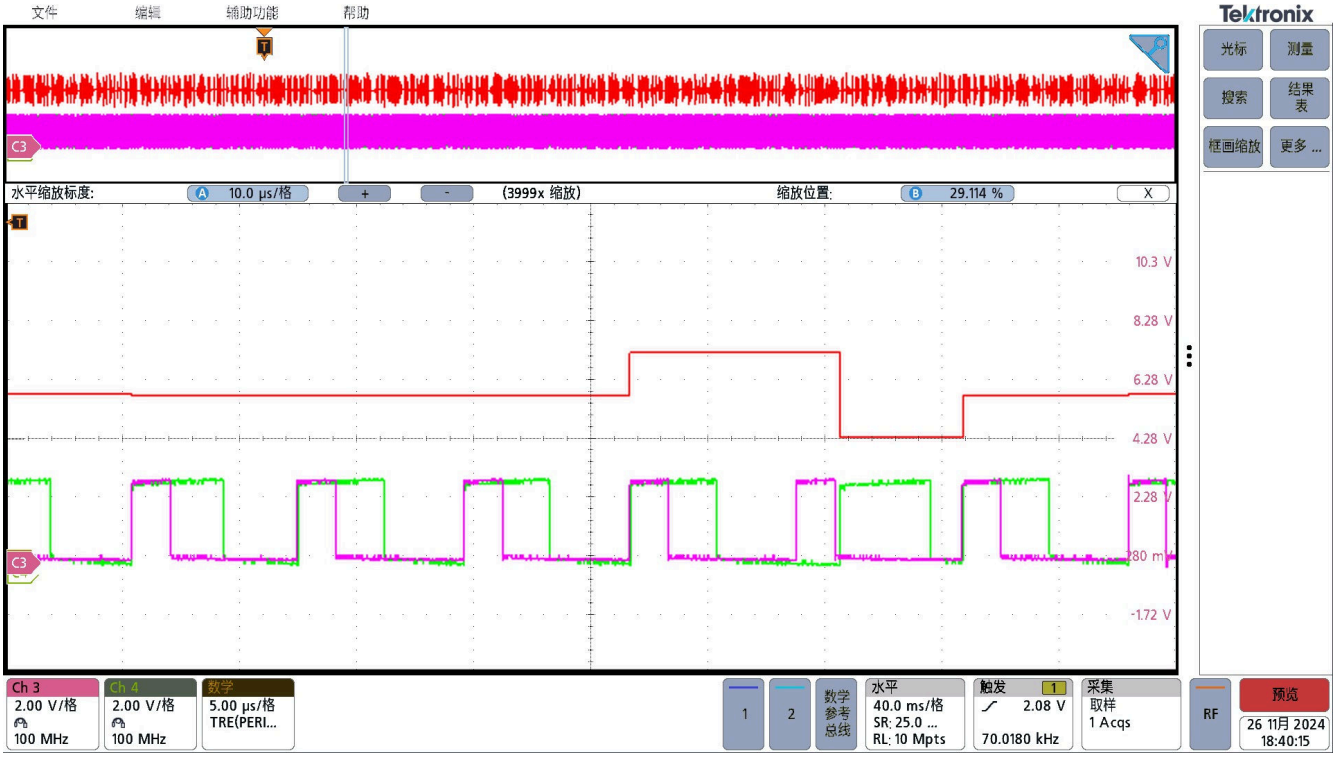

Figure 1. Abnormal Interrupt Oscillation (Ch4: Interrupt with GPIO toggle; Ch3: Interrupt trigger on ePWM ZRO event, Red signal: Frequency trend measurement from oscilloscope)

Interrupt Propagation Path and Interrupt Timing

First, there are two concepts to focus on with interrupt latency that are interrupt propagation path and interrupt timing. The interrupt propagation path is the time from an interrupt request triggering to the beginning of the interrupt service function. Second, confirm if there are any interference factors during an interrupt request triggering or with normal interrupt execution. Third, interrupt latency is maintained to be executed normally by setting interrupt priority reasonably (such as interrupt nesting and register stack restore/protect) and shielding others interrupt interference source.

Interrupt propagation path on C28x handles interrupts in four main phases:

1. Receive the interrupt request. Suspension of the current program sequence must be requested by a software interrupt (from program code) or a hardware interrupt (from a pin or an on-chip device), as described in Figure 2.
2. Approve the interrupt. The C28x must approve the interrupt request. If the interrupt is maskable, certain conditions must be met for the C28x to approve the interrupt request. For non-maskable hardware interrupts and for software interrupts, approval is immediate, as described in Figure 3.
3. Prepare for the interrupt service routine and save register values, as described in Figure 4.
4. Execute the interrupt service routine. This is interrupt loop processing entry, call ISR.

Most programmers only pay attention to first two phases, and know less about stack protection or recovery and interrupt response in the last two phases. This application brief dives deeper into phases three and four.

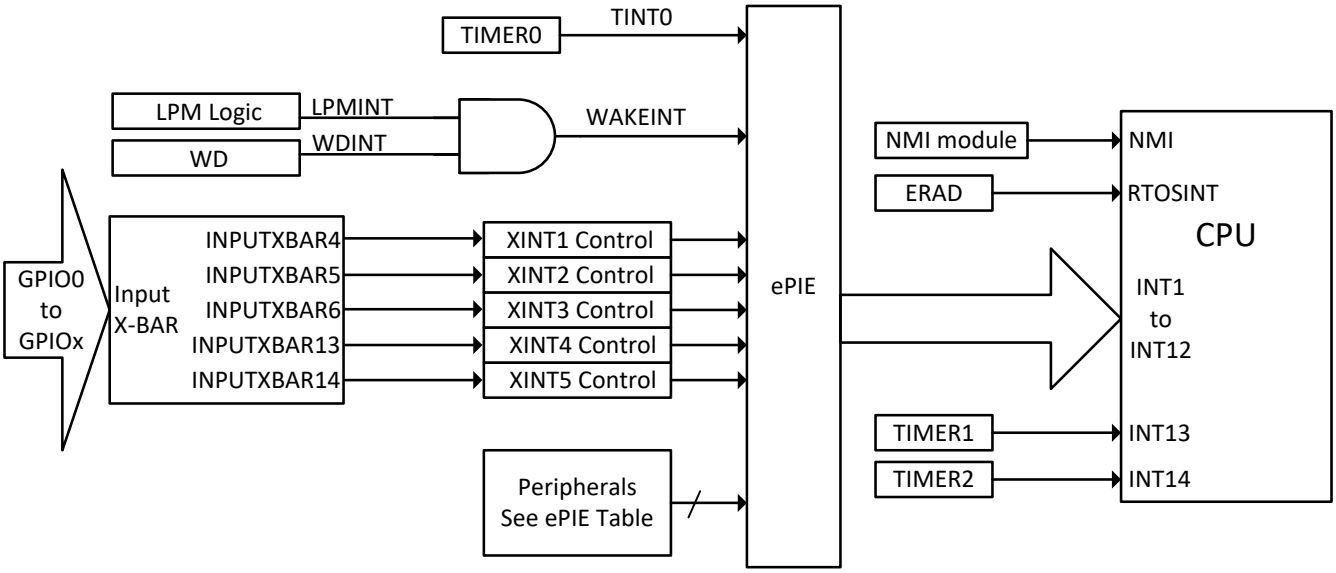

Figure 2. Interrupt Triggering Source (F28003x)

Figure 3 shows how peripheral interrupts propagate to the CPU.

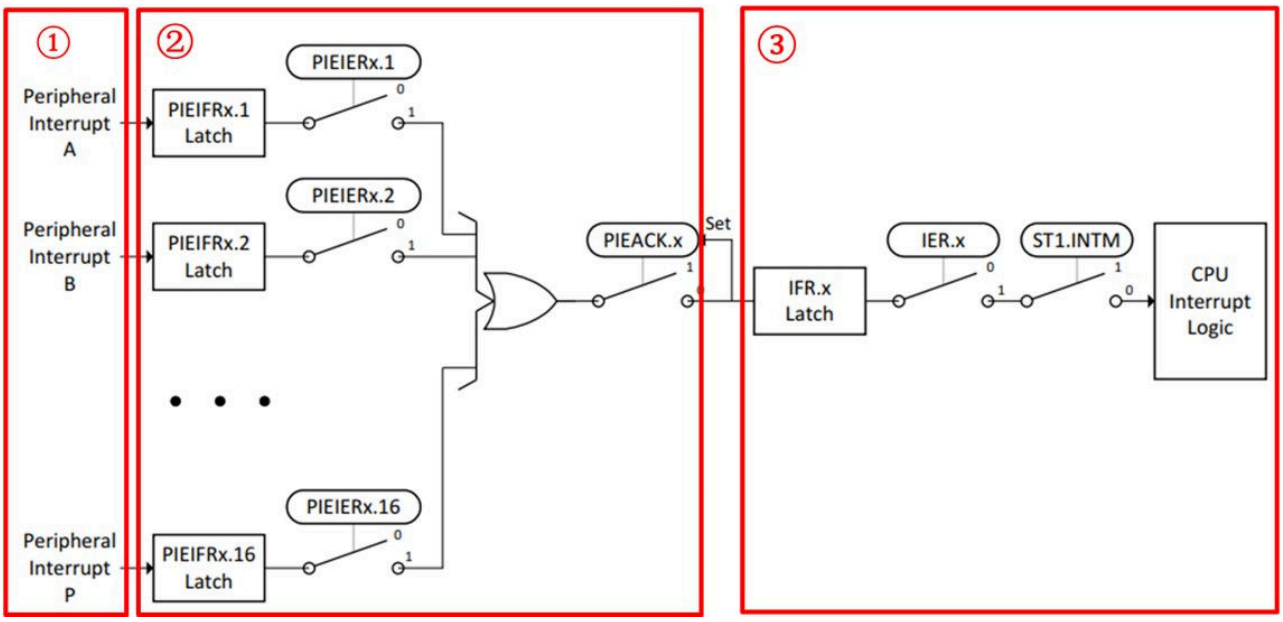

Figure 3. Interrupt Propagation Path

Figure 4 shows how C28x generates and responds to interrupt service functions.

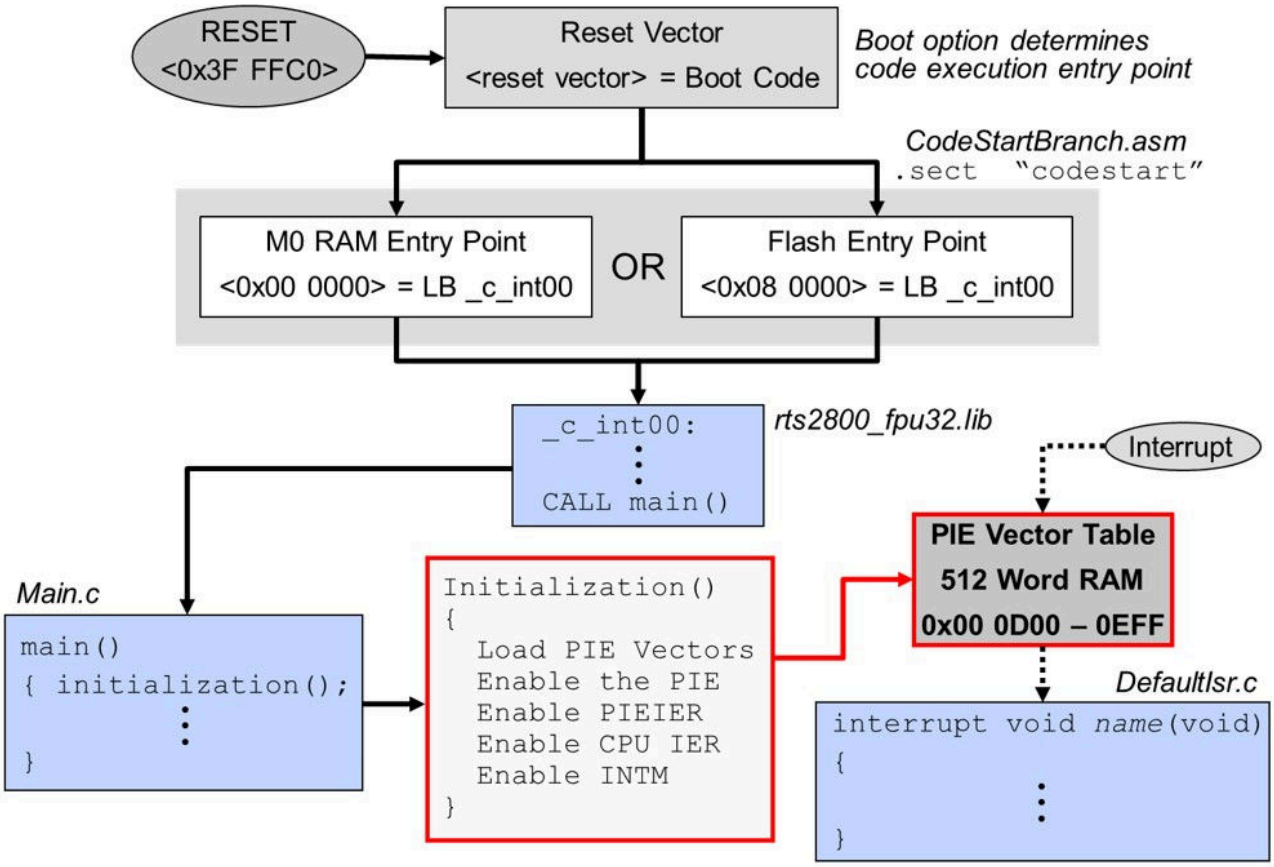

Figure 4. Interrupt PIE Initialization Code Flow

The interrupt timing from interrupt request triggering to interrupt service function ISR:

1. Minimum latency (to when real work occurs in the ISR), 14 or 16 cycles: take F280039C 120Mhz CPU for example, Minimum latency- add All Registers Save or Restored automatically On Real-Time interrupt prepared can be 40 cycles, it can be around 50 cycles/415ns latency.
2. Maximum latency: Depends on C28x handles cycles for stack protection and restoration, wait states, INTM, and not interruptible RPT Instruction, as described in Figure 5.

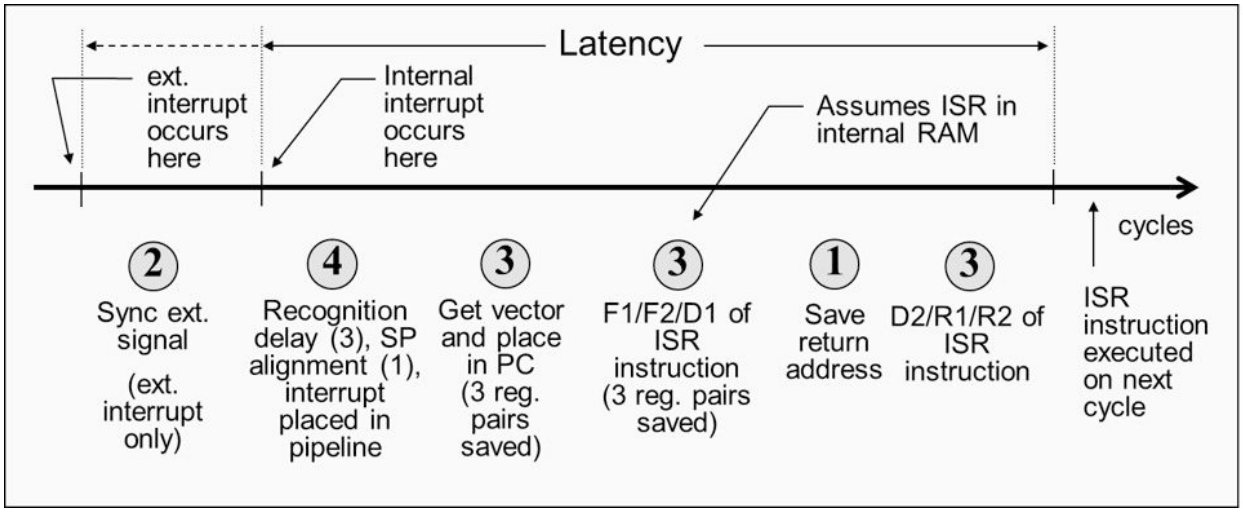

Figure 5. Interrupt Latency Flow

In addition to correct usage of interrupt request and interrupt approval operation bits (Such as INTM, IER bit), consider the following interference factors and interrupt nesting that can affect interrupts.

Interrupt Nesting and Interference Factors

- When C28x is executing an interrupt or responding to a high-priority interrupt, by default the interrupt cannot continue to respond to other low-priority interrupts. However, there are steps that can be followed to enable servicing of other interrupts within the current interrupt. This is called interrupt nesting.
- When uninterruptible instructions such as RPT instructions are being executed for too long or too frequently, the C28x CPU cannot respond to the interrupts in a timely manner.

These two points are sources that can affect how the interrupt timing can be affected.

Interrupt Nesting

When talking about interrupt nesting, interrupts are automatically prioritized by the C28x hardware. Prioritization for all interrupts can be found in the System Control guide specific to the particular device family. When the C28x CPU is responding to a low-priority interrupt, the CPU interferes with the normal response of a high-priority interrupt, as described in Figure 6.

Table 3-3. Pie Channel Mapping

	INTx.1	INTx.2	INTx.3	INTx.4	INTx.5	INTx.6	INTx.7	INTx.8	INTx.9	INTx.10	INTx.11	INTx.12	INTx.13	INTx.14	INTx.15	INTx.16
INT1.y	ADCA1	ADCB1	ADCC1	XINT1	XINT2	-	TIMER0	WAKE / WDOG	-	SYS_ERR	-	-	-	-	-	-
INT2.y	EPWM1_TZ	EPWM2_TZ	EPWM3_TZ	EPWM4_TZ	EPWM5_TZ	EPWM6_TZ	EPWM7_TZ	EPWM8_TZ	-	-	-	-	-	-	-	-
INT3.y	EPWM1	EPWM2	EPWM3	EPWM4	EPWM5	EPWM6	EPWM7	EPWM8	-	-	-	-	-	-	-	-
INT4.y	ECAP1	ECAP2	ECAP3	-	-	-	-	-	-	-	ECAP3INT2	-	-	-	-	-
INT5.y	EQEP1	EQEP2	-	-	CLB1	CLB2	CLB3	CLB4	SDFM1	SDFM2	-	-	SDFM1DR1	SDFM1DR2	SDFM1DR3	SDFM1DR4
INT6.y	SPIA_RX	SPIA_TX	SPIB_RX	SPIB_TX	-	-	-	-	-	-	-	-	SDFM2DR1	SDFM2DR2	SDFM2DR3	SDFM2DR4
INT7.y	DMA_CH1	DMA_CH2	DMA_CH3	DMA_CH4	DMA_CH5	DMA_CH6	-	-	-	-	FSITX_INT1	FSITX_INT2	FSIRX_INT1	FSIRX_INT2	-	DCC0
INT8.y	I2CA	I2CA_FIFO	I2CB	I2CB_FIFO	-	-	-	-	LINA_0	LINA_1	LINB_0	LINB_1	PMBUSA	-	-	DCC1
INT9.y	SCIA_RX	SCIA_TX	SCIB_RX	SCIB_TX	DCANA_0	DCANA_1	-	-	MCAN_0	MCAN_1	MCAN_ECC	MCAN_WAKE	BGCRC_CPU	-	-	HICA
INT10.y	ADCA_EVT	ADCA2	ADCA3	ADCA4	ADCB_EVT	ADCB2	ADCB3	ADCB4	ADCC_EVT	ADCC2	ADCC3	ADCC4	-	-	-	-
INT11.y	CLA1_1	CLA1_2	CLA1_3	CLA1_4	CLA1_5	CLA1_6	CLA1_7	CLA1_8	-	-	-	-	-	-	-	-
INT12.y	XINT3	XINT4	XINT5	MPOST	FMC	-	FPU_OVERFLOW	FPU_UNDERFLOW	-	RAM_CORR_ERR	FLASH_CORR_ERR	RAM_ACC_VIOLATION	AES_SIN_TREQ	BGCRC_CLA1	CLA_OVERFLOW	CLA_UNDERFLOW

Figure 6. Interrupt PIE Channel Mapping

Therefore, application code needs to add simple software prioritization during low priority interrupts. This allows the CPU to respond to high-priority interrupt processing in a timely manner from the execution of low-priority interrupts. Here are the steps C28x performs interrupt nesting:

1. Set the global priority:
   a. Modify the IER register to allow CPU interrupts with a higher user priority to be serviced. (Note: at this time IER has already been saved on the stack.)
2. Set the group priority:
   a. Modify the appropriate PIEIERx register to allow group interrupts with a higher user set priority to be serviced. (Note: Do NOT clear PIEIER register bits from another group other than that being serviced by this ISR. Doing so can cause erroneous interrupts to occur.)
3. Enable interrupts: There are three steps to do this:
   a. Clear the PIEACK bits.
   b. Wait at least one cycle.
   c. Clear the INTM bit. Use the assembly statement asm(" CLRC INTM"); or TI examples use #define EINT asm(" CLRC INTM").
4. Run the main part of the ISR.
5. Set INTM to disable interrupts. Use asm(" SETC INTM"); or TI examples use #define DINT asm(" SETC INTM").
6. Restore PIEIERx (optional depending on step 2)

7. Return from ISR:
   a. This restores INTM and IER automatically. Meanwhile, the example code as below:

```
// // C28x ISR Code // // Enable nested interrupts // // ADCA1 interrupt for loop Interrput
void INT_myCPUTIMER2_ISR(void)
{
        uint16_t TempPIEIER;
        TempPIEIER = PieCtrlRegs.PIEIER1.all; // Save PIEIER register for later
        IER |= 0x001;                         // Set global priority by adjusting IER
        IER &= 0x001;
        PieCtrlRegs.PIEIER1.all &= 0x0001;    // Set group priority by adjusting PIEIER1
to //allow INT1.1 to interrupt current CPU time0 ISR
        PieCtrlRegs.PIEACK.all = 0xFFFF;      // Enable PIE interrupts
        asm("       NOP");                    // Wait one cycle
        EINT;                                 // Clear INTM to enable interrupts
        //
        // Insert ISR Code here.......
        // for now just insert a delay
        //
        //for(i = 1; i <= 10; i++) {}
        //
        // Restore registers saved:
        //
        DINT;
        PieCtrlRegs.PIEIER1.all = TempPIEIER;
}
```

Our next-generation C29x architecture F29H85x supports Hardware Interrupt Prioritization requires no software overhead and allows interrupt nesting. For C29x architecture all registers are save/restored automatically by hardware on real-time interrupt in 10 cycles when compared C28x 40 cycles.

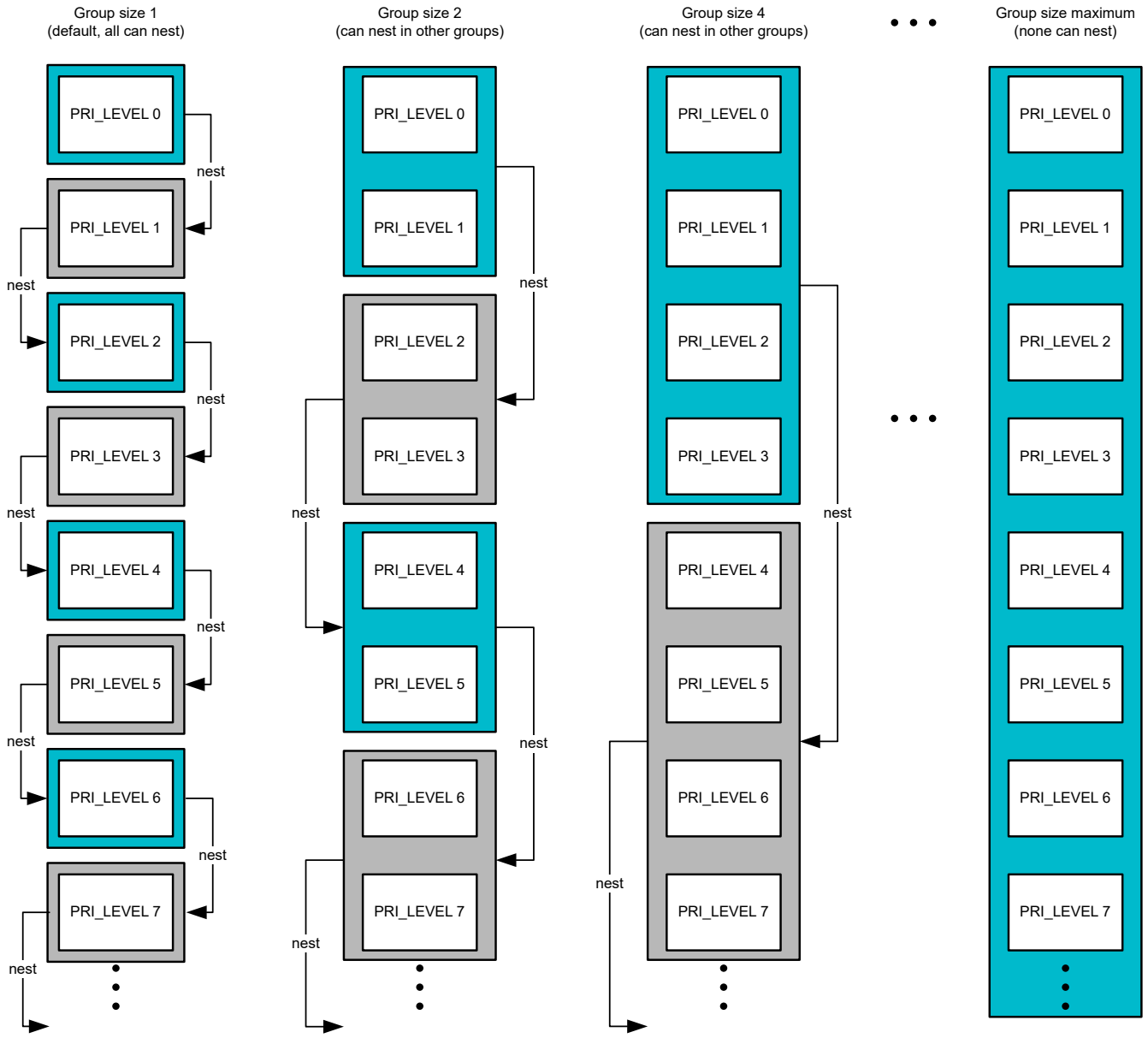

Figure 7. C29x Interrupt Grouping Overview

Nesting for INTs within the PIPE module is enabled within an Interrupt Service Routine (ISR) by setting the CPU level DSTS.INTE bit active because this bit is disabled while entering the ISR., as described in Figure 7. Here are the steps C28x performs interrupt nesting:

```
// // C29x ISR Code // // Enable nested interrupts // // ADCA1 interrupt for loop Interrput
void INT_myCPUTIMER0_ISR(void)
{
       // Set INTE to 1 to enable interrupts here.
       ENINT;
       // Insert ISR Code here.......

}
```

C28x Interrupt Nesting Test Results

The below test results are with two interrupts: EPWM interrupt at 150kHz (yellow signal) and Timer2 interrupt at 1kHz (blue signal). Timer2 interrupt has lower priority than the EPWM interrupt. Without C28x interrupt nesting enabled, the interrupt frequency of the EPWM is not be 150kHz, as shown in Figure 8. Keeping EPWM interrupt fixed at 150Khz is only possible by leveraging C28x CPU interrupt nesting, as shown in Figure 9. The test results are based on LAUNCHXL-F280039C. If interrupt nesting is not enabled by software method as described with the above code there is abnormal interrupt behavior.

With interrupt nesting enabled, the higher priority interrupts can still be entered and executed even when a lower priority interrupt has occurred. This makes sure higher priority interrupt frequencies are constant..

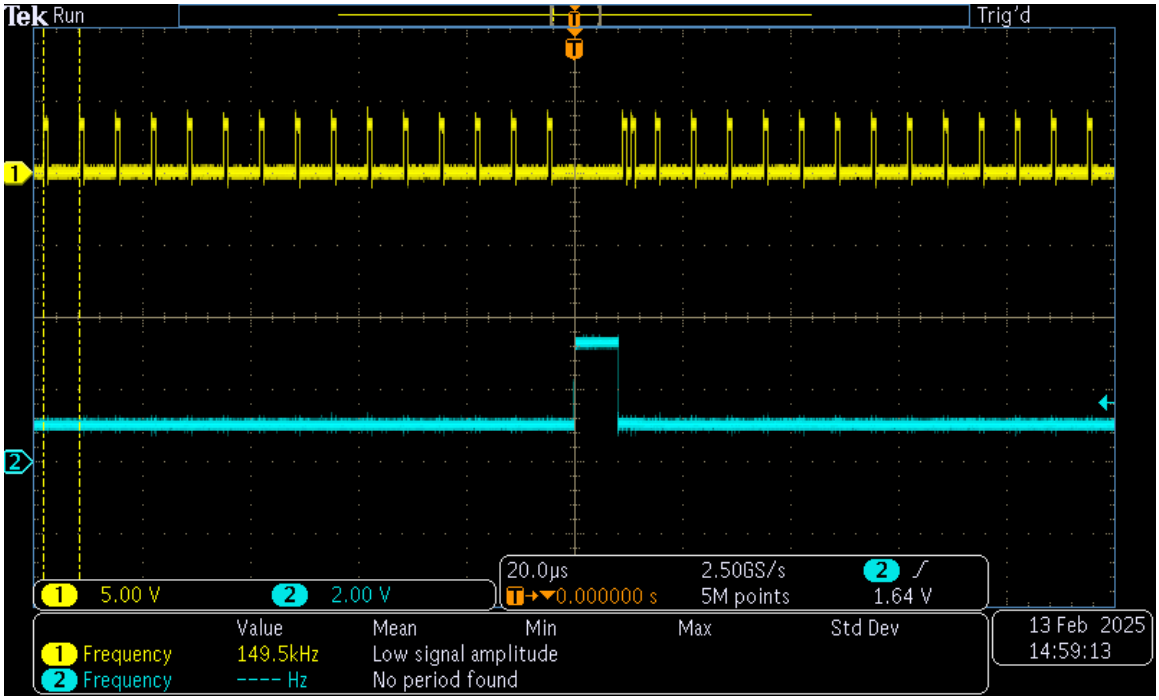

Figure 8. C28x Interrupt Nesting Disabled Test Results (CH1: EPWM interrupt, CH2: TIMER2 interrupt)

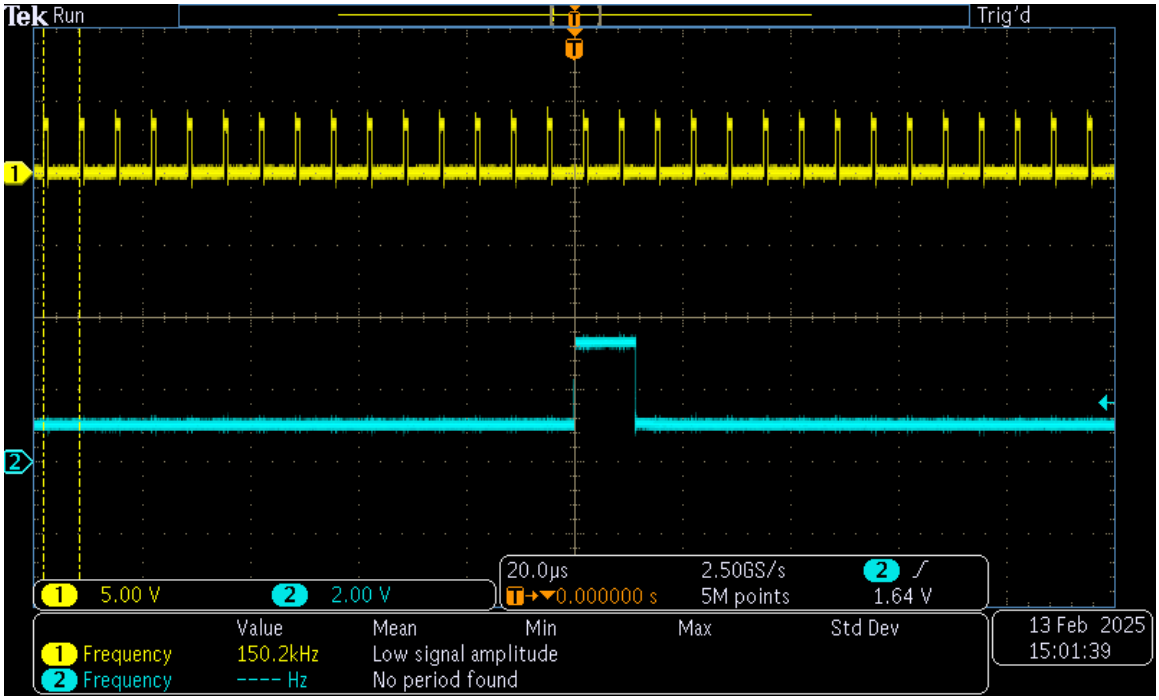

Figure 9. C28x Interrupt Nesting Enabled Test Results (CH1: EPWM interrupt, CH2: TIMER2 interrupt)

C29x Interrupt Nesting Test Results

The below test results are with two interrupts: EPWM interrupt at 150kHz (pink signal) and Timer2 interrupt at 1kHz (green signal). Timer2 interrupt has lower priority than the EPWM interrupt. Without C29x interrupt nesting enabled, the interrupt frequency of the EPWM is not be 150kHz, as shown in Figure 10. Keeping EPWM interrupt fixed at 150Khz is only possible by leveraging C29x CPU interrupt nesting, as shown in Figure 11. This is tested based on the F29x devices. If interrupt nesting is not enabled by software method as described the above code there is abnormal interrupt behavior.

With interrupt nesting enabled, the higher priority interrupts can still be entered and executed even when a lower priority interrupt has occurred. This makes sure higher priority interrupt frequencies are constant.

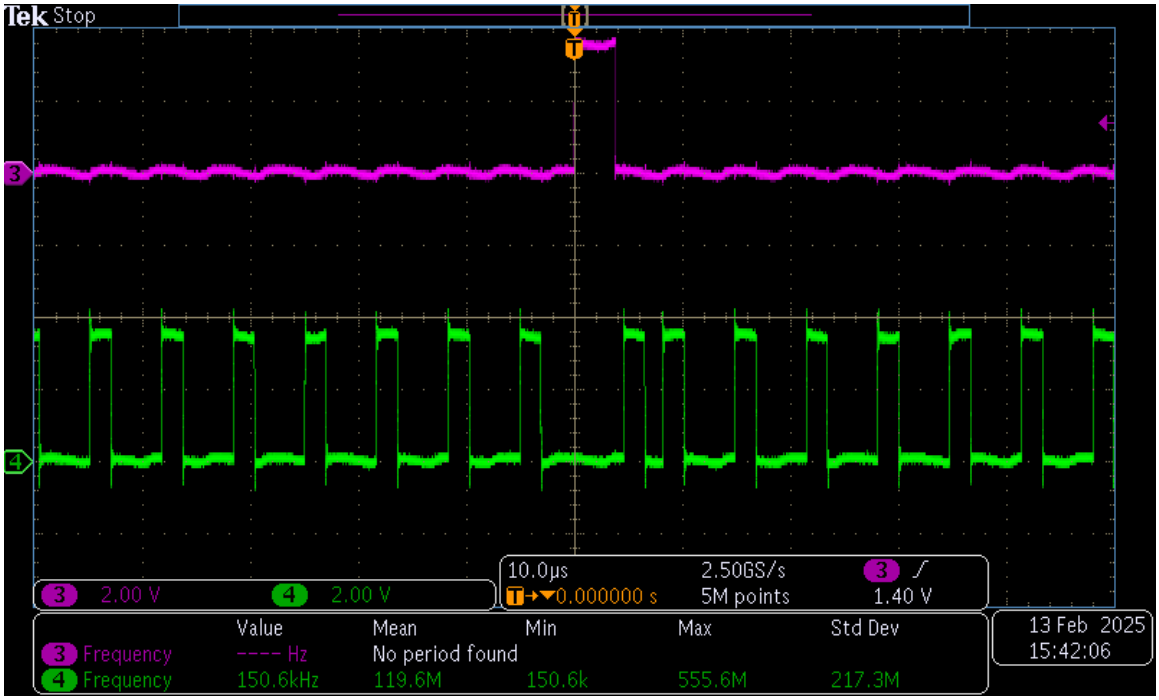

Figure 10. C29x Interrupt Nesting Disabled Test Results (CH1: EPWM interrupt, CH2: TIMER2 interrupt)

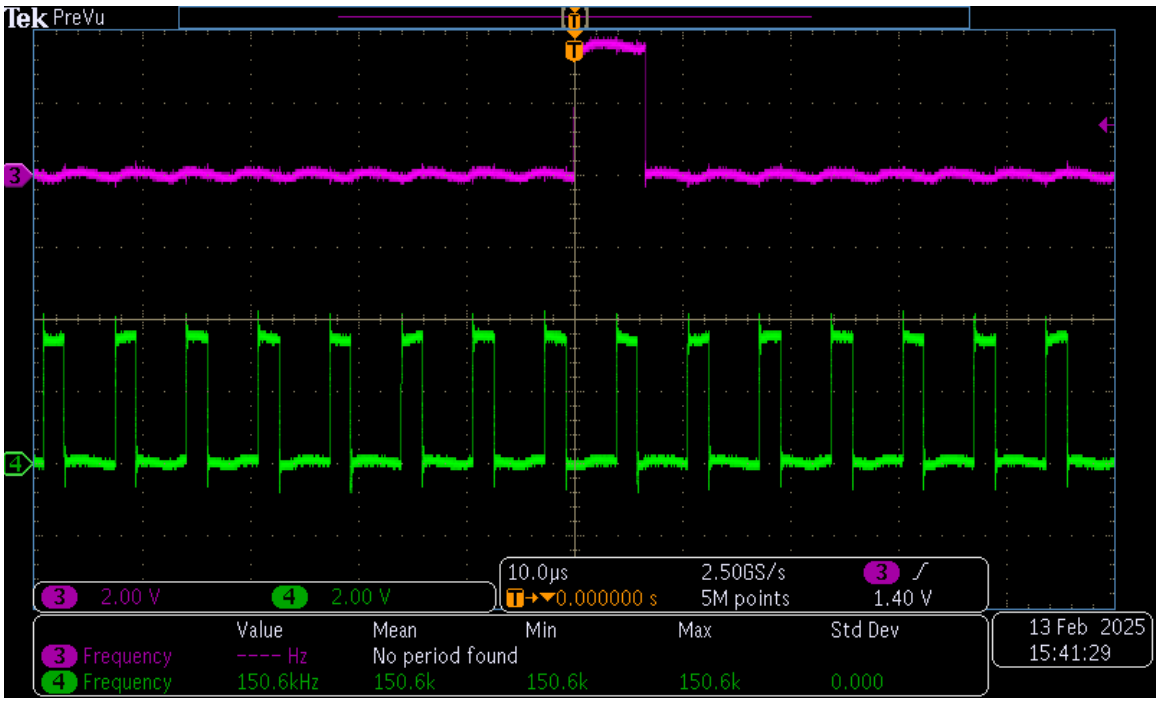

Figure 11. C29x Interrupt Nesting Enabled Test Results (CH1: EPWM interrupt, CH2: TIMER2 interrupt)

Uninterruptible Instructions Affects Interrupt Timing

When talking about uninterruptible instructions RPT, a large number of repeated global initialization variables are used in the main program or state machine such as Memcopy, for loop assigns the same array, or repeated operations are performed, the C2000 compiler automatically generates RPT instructions. The repeat (RPT) instruction allows the execution of a single instruction (N + 1) times, where N is specified as an operand of the RPT instruction. The instruction is executed once and then repeated N times. When RPT is executed, the repeat counter (RPTC) is loaded with N. RPTC is then decremented every time the repeated instruction is executed, until RPTC equals 0. For a description of RPT and a list of repeatable instructions, see the *RPT *8bit/loc16* section in the *C28x Assembly Language Instructions* chapter of the *TMS320C28x CPU and Instruction Set Reference Guide*.

Due to this RPT instruction being uninterruptible, it does not have the context saving stack protection or restore function. So, the PC pointer stays in RPT at this time and it may not be able to respond to the interrupt request in time, as described in Figure 12.

RPT #8bit/loc16 ***Repeat Next Instruction***

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
RPT #8bit	1111 0110 CCCC CCCC	X	–	1
RPT loc16	1111 0111 LLLL LLLL	X	–	4

Operands

#8bit – 8-bit constant immediate value (0 to 255 range)

loc16 – Addressing mode (see Chapter 5)

Description

Repeat the next instruction. An internal repeat counter (RPTC) is loaded with a value N that is either the specified #8bit constant value or the content of the location pointed to by the "loc16" addressing mode. After the instruction that follows the RPT is executed once, it is repeated N times; that is, the instruction following the RPT executes N + 1 times. Because the RPTC cannot be saved during a context switch, repeat loops are regarded as multicycle instructions and are not interruptible.

Note on syntax: Parallel bars (||) before the repeated instruction are used as a reminder that the instruction is repeated and is not interruptable. When writing inline assembly, use the syntax

```
asm(|| RPT #8bt/ loc16 || instruction");
```

Not all instructions are repeatable. If an instruction that is not repeatable follows the RPT instruction, the RPTC counter is reset to 0 and the instruction only executes once. The 28x Assembly Language tools check for this condition and issue warnings.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Copy the number of elements specified in VarA from Array1
; to Array2:
; int16 Array1[N]; // Located in high 64K of program space
; int16 Array2[N]; // Located in data space
; for(i=0; i < VarA; i++)
;    Array2[i] = Array1[i];
  MOVL XAR2,#Array2  ; XAR2 = pointer to Array2
  RPT @VarA          ; Repeat next instruction
                     ; [VarA] + 1 times
|| XPREAD            ; Array2[i] = Array1[i],
*XAR2++,*(Array1)    ; i++
```

Figure 12. RPT Instructions Introduction

Therefore, you can go into the C2000 compilers using the correct settings or you can avoid generated C usage notes. Regarding the C2000 compilers, you can change the Project Properties -> C2000 Compiler -> Advanced Options -> Runtime Model Options -> Enable “Don’t generate RPT instructions, as described in Figure 13.

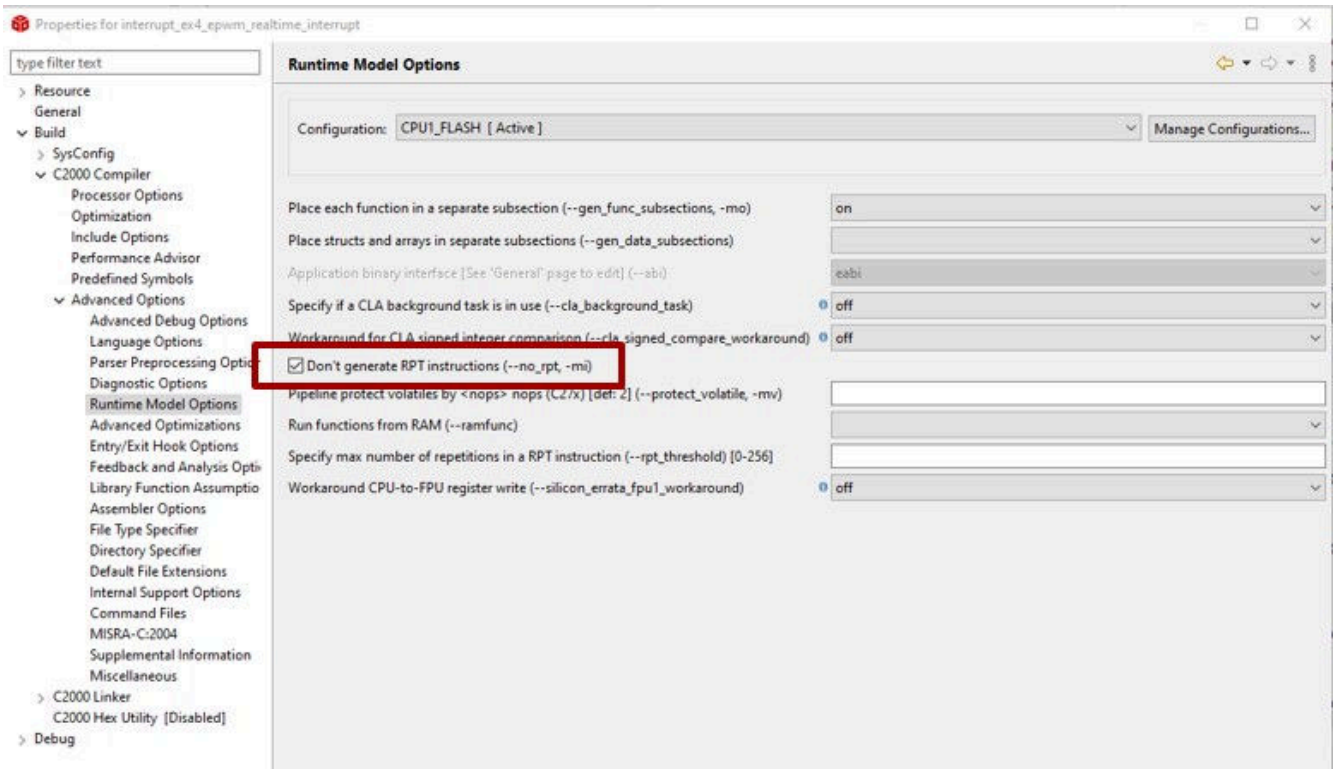

Figure 13. C2000 Compiler Setting About RPT Instructions

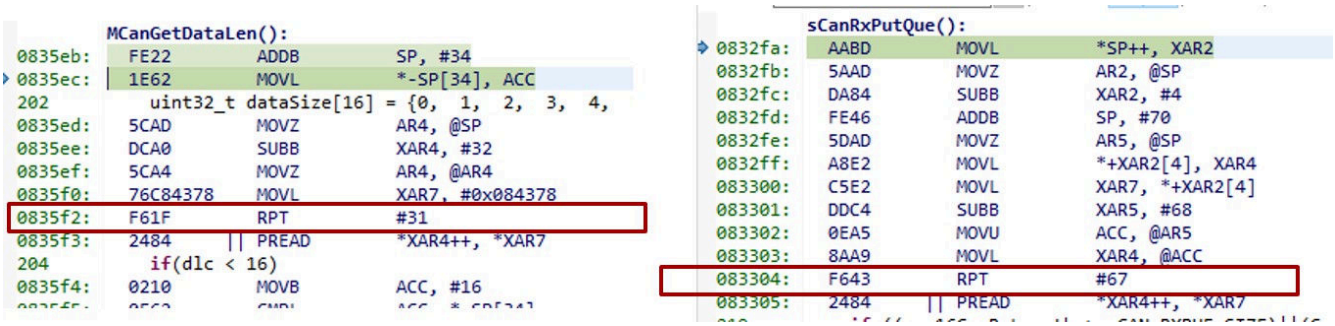

Figure 14. RPT Instructions Generated By Above Functions

```
202 uint32_t MCanGetDataLen(uint32_t dlc)
203 {
204     uint32_t dataSize[16] = {0,  1,  2,  3,  4,  5,  6, 7, 8,
205                              12, 16, 20, 24, 32, 48, 64};
206     if(dlc < 16)
207     {
208         return(dataSize[dlc]);
209     }
210     else
211     {
212         return 0;
213     }
214
215 }
```

```
217 void sCanRxPutQue(CANFRAME CanRx)
218 {
219     //uint16_t u16UDSRxDataFlag = 0;
220
221     if ((s_u16CanRxLength >= CAN_RXBUF_SIZE)||(CanRx.uBLen !=8))
222     {
223         return;
224     }
225
226     *s_pCanRxIn = CanRx;
227     if(s_pCanRxIn->CanId.all!= 0)
228     {
229
230         if (s_pCanRxIn == &s_aCanRxBuf[CAN_RXBUF_SIZE - 1])
231         {
232             s_pCanRxIn = s_aCanRxBuf;
233         }
234         else
235         {
236             s_pCanRxIn++;
237         }
238         s_u16CanRxLength++;
239     }
240 }
```

Figure 15. Source Code

Finally, follow the above C2000 compilers settings and the EPWM ISR works normally, as described in Figure 16.

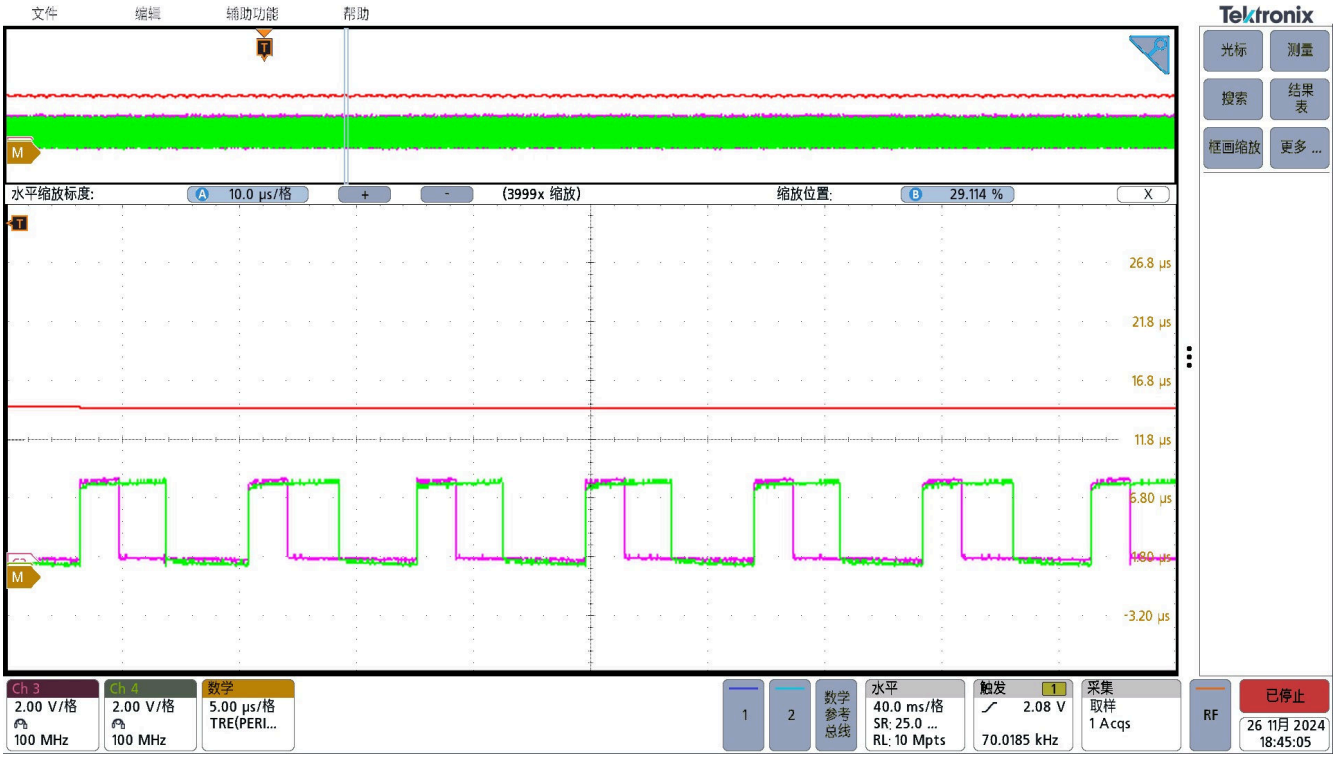

Figure 16. Abnormal Interrupt Oscillation Fixed (Ch4: Interrupt with GPIO toggle; Ch3: Interrupt trigger on ePWM ZRO event, Red signal: Frequency trend measurement from oscilloscope)

Summary

The interrupt is executed normally by observing the minimum interrupt delay. However, the factors that affect the normal execution of the interrupt are:

- Whether the interrupt request is triggered normally
- Whether the INTM or IER enable bit is enabled, and whether the IFG flag is set normally
- Whether the interrupt propagation path is likely to be blocked
- Whether the interrupt response is likely to be disturbed when saving the register, such as the non-interruptible instruction RPT
- Whether the interrupt is nested, when the low-priority interrupt is responding, the high-priority interrupt is blocked, affecting the timeliness of the interrupt response

This technical article tells you how to locate and troubleshoot the factors affecting the interrupts.

Trademarks

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265