AN205203

FR60 Family, ISR Double Execution

This application note gives some information about the behaviour of a possible Interrupt Service Routine (ISR) Double Execution.

1 Introduction

This application note gives some information about the behaviour of a possible interrupt service routine double execution.

The new FR60 family has the feature to speed up the write access to Resources connected on slower clocked busses e.g. CLKP or CANCLK by using write buffers. This could raise the problematique of double executed interrupt service routines with application and bus ratio specific origin.

This does not affect all interrupt service routines!

This behaviour can be easily prevented if it is deliberated during the interrupt service routine design process and an affected ISR can be easily redesigned.

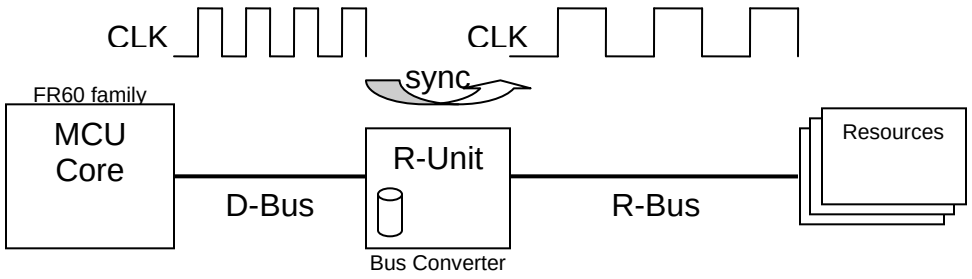
2 Origin of Double Executed ISR's

The problematique of double executed interrupt service routines can arise if there is no additional read access to the internal Resource- or Can-Bus of the FR MCU, after clearing a resource or CAN interrupt flag.

The FR60 family possesses an additionally inserted write buffers for access to the generally slower clocked R-Bus, for resources like UART's and TIMER's, and one for all CAN resource.

This write buffers give the MCU the possibility not to wait while the R-Unit or CAN syncs between CLKB and the slower Resource or CAN clock for clock ratios below or equal to 1!

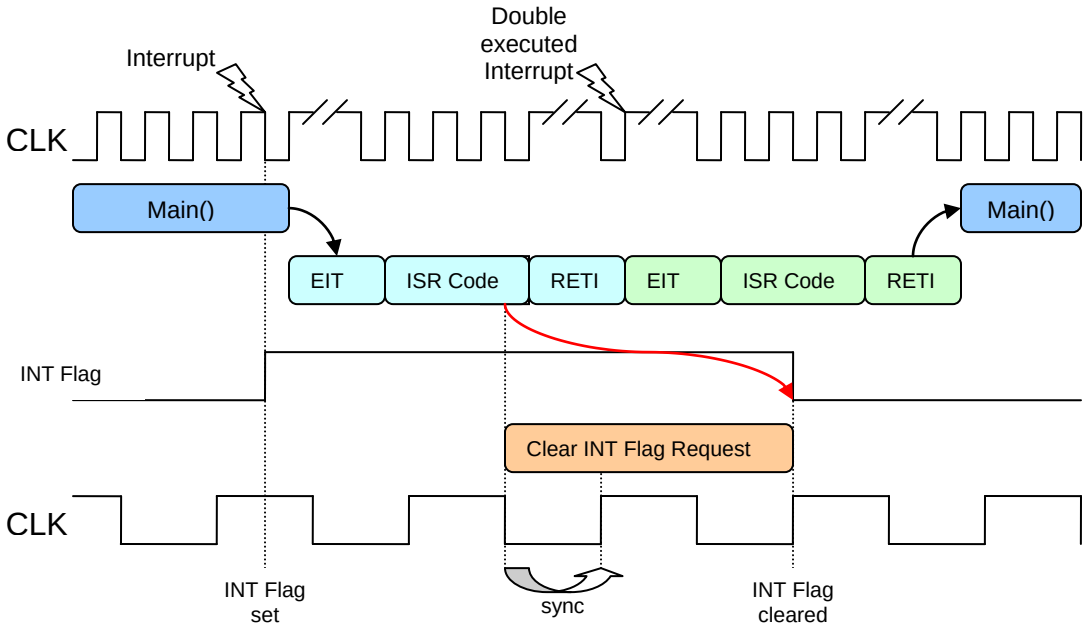
Figure 1. D- + R-Bus Clock Domains



This performance feature spawns the possible behaviour, for $\frac{CLKB}{CLKP}$ or $\frac{CLKB}{CANCLK}$ ratios lower or equal to 1, that an interrupt flag clear is not completed before the MCU core has finished the exit (RETI) from the interrupt service routine. This pushes the MCU immediately back into the interrupt service routine because the interrupt flag is still active.

This condition only comes up if there is no further read access to any Resource or any CAN resource after clearing the interrupt flag within the interrupt service routine.

Figure 2. Double Executed Resource Interrupt Service Routine



3 Analysis of interrupt service routines

The following chapter describes how to analyze and redesign existing interrupt service routines if the application shows a double execution of any Resource or CAN interrupt service routine.

3.1 Unaffected Resource or CAN ISR

Resource or CAN ISR's with an additional read access to the respective memory space after clearing the interrupt flag are not affected in this double executed interrupt service routine problematique.

Figure 3. Unaffected Interrupt Service Routine

```
__interrupt void IRQHandler (void) {
       /* clear ext. interrupt flag */
       EIRRy = 0;

       /* interrupt service code */
       /* with read access to resource */
       /* memory */
       a = EIRRy;
}
```

3.2 Affected Resource or CAN ISR

The affected interrupt service routine as it can be found in common applications is shown in the figure below.

Figure 4. Affected Interrupt Service Routine

```
__interrupt void IRQHandler (void) {
       /* clear ext. interrupt flags */
       EIRRy = 0;

       /* no add. resource  read  */
       /* access within interrupt */
      /* service code */
       …
}
```

Note: For the following $\frac{CLKB}{CLKP}$ ratios it is possible that these interrupt service routine structure spawns the above described double execution condition.

$\frac{CLKB}{CLKP}$ <= 1 or

$\frac{CLKB}{CANCLK}$ <= 1

3.2.1 Redesigned Resource ISR

The Resource interrupt service routine shown in the figure below inhibits the double execution by adding the macro call RB_SYNC to the end of the interrupt service routine. This forces an additional read access to the Resource memory space which waits until the interrupt clear instruction on the respective Resource register is completed and the write buffer is empty.

Figure 5. Redesigned Resource Interrupt Service Routine

```
__interrupt void IRQHandler (void) {
       /* clear ext. interrupt flag */
       EIRRy = 0;

       /* interrupt service code */
       …

       /* Synchronization with R-Bus */
       RB_SYNC;
}
```

RB_SYNC macro

The RB_SYNC macro is a simple assembler read access to a dummy register which is located within the Resource memory space.

This macro is defined in the respective derivates header file in C code.

- #define RB_SYNC if(RBSYNC)

The compiler produces two different assembler codes depending on the selected optimization level.
Optimization level = NONE:

'if(RBSYNC)'

```
DMOVB   @03A,R13
MOV       R13,R0
```

Optimization level = SPEED

'if(RBSYNC)'

```
DMOVB   @03A,R13
```

3.2.2 Redesigned CAN ISR

The interrupt service routine template shown in the figure below is one possibility to inhibit the double execution. This forces an additional read access to the CAN memory space which waits until the interrupt clear instruction on the respective CAN register is completed and the write buffer is empty. This is done by adding a macro call at the end of each CAN ISR.

Figure 6. Redesigned CAN Interrupt Service Routine

```
__interrupt void CANIRQHandler (void) {
      /* CAN interrupt service code */
      …
      /* Synchronization with CAN-Bus */
      CB_SYNC;
}
```

CB_SYNC macro

The C_SYNC macro is a simple assembler read access to a dummy register which is located within the CAN memory space.

This macro is defined in the respective derivates header file in C code.

- #define CBSYNC if(CBSYNC)

The compiler produces two different assembler codes depending on the selected optimization level.

Optimization level = NONE:

'if(CBSYNC)'

```
DMOVB   @C00E,R13
MOV     R13,R0
```

Optimization level = SPEED

'if(CBSYNC)'

```
DMOVB   @C00E,R13
```

4 Behaviour of the LIN-USART and RTC module

The following chapter describes the unique behaviour of the LIN-USART and RTC (real time clock) module with respect to handling of the double IRQ execution issue.

4.1 Explanation of behaviour

Unlike other resources on the microcontroller the LIN-USART and RTC module output the interrupt request signal to CPU delayed by additional clock cycles (CLKP) on clearing the interrupt flags.

The following table shows the related interrupt bits:

Table 1. Interrupt Bits with Additional Clock Cycles

Resource	Register	Bit	Delay [CLKP cycle]	Comment
LIN-USART	SSRx	RDRF	1	RDRF is cleared by reading RDRx.
	SCRx	CRE	1	Bit that clears error flag (PE,ORE,FRE)
	SMRx	UPCL or SRST	1	Reset bit of LIN-USART that clears all flags (TDRE,RDRF,LBD,PE,ORE,FRE)
RTC	WTCER	INT4	2	0.5 second interrupt cause bit
	WTCR	INT3	2	1day interrupt cause bit
	WTCR	INT2	2	1hour interrupt cause bit
	WTCR	INT1	2	1minute interrupt cause bit
	WTCR	INT0	2	1second interrupt cause bit

Due to this behaviour it cannot be guaranteed that the signalled interrupt request is already inactive when leaving the ISR (i.e. additional clock cycles have elapsed when leaving the ISR).

- **LIN-USART:** Double IRQ execution doesn't occur if the description of using single RB_SYNC is followed (the additional 1xCLKP cycle is covered by the method of preventing double IRQ execution caused by the write buffer).
- **RTC:** Double IRQ execution may occur even if the description of using single RB_SYNC is followed (the additional 2xCLKP cycles are not covered by the method of preventing double IRQ execution caused by the write buffer). I.e. for the RTC ISR a particular handling is necessary and explained in following sections.

4.2 Unaffected RTC ISR

RTC ISR with two additional read accesses to the respective memory space after clearing the interrupt flag are not affected in this double executed interrupt service routine issue.

Figure 7. Unaffected RTC Interrupt Service Routine

```
__interrupt void IRQHandler (void) {
       /* clear ext. interrupt flag */
       WTCR_INT0 = 0;

       /* interrupt service code */
       /* with read access to res. */
       /* memory */
       a = WTCR_INT0;
       b = WTBR0;
}
```

4.3 Affected RTC ISR

There are two cases which involve the RTC ISR into the double IRQ issue.

4.3.1 No additional read

The affected interrupt service routine as it can be found in common applications is shown in the figure below.

Figure 8. Affected RTC Interrupt Service Routine (No Read)

```
__interrupt void IRQHandler (void) {
       /* clear ext. interrupt flags */
       WTCR_INT0 = 0;

       /* no add. resource read  */
       /* access within interrupt */
        /* service code */
       …
}
```

4.3.2 Single additional read

The affected interrupt service routine as it can be found in common applications is shown in the figure below.

Figure 9. Affected RTC Interrupt Service Routine (Single Read)

```
__interrupt void IRQHandler (void) {
              /* clear ext. interrupt flags */
              WTCR_INT0 = 0;

              /* interrupt service code */
              /* with single read access to */
              /* resource memory */
              a = WTCR_INT0;
}
```

4.4 Software adaption of ISR handling

The following software adaption of the RTC ISR will fully resolve the issue of double IRQ execution and should be implemented wherever applicable.

4.4.1 Redesigned RTC ISR

The RTC interrupt service routine shown in the figure below inhibits the double execution by adding the macro call RB_SYNC:

ONCE to the end of the interrupt service routine if there is already a single read:

Figure 10. Redesigned RTC Interrupt Service Routine (Single Sync)

```
__interrupt void IRQHandler (void) {
       /* clear RTC interrupt flag */
       WTCR_INT0 = 0;

       /* interrupt service code */
       /* with single read access to */
       /* resource memory */
       a = WTCR_INT0;

       /* Synchronization with R-Bus ONCE*/
       RB_SYNC;
}
```

TWICE to the end of the interrupt service routine if there is no read:

Figure 11. Redesigned RTC Interrupt Service Routine (Double Sync)

```
__interrupt void IRQHandler (void) {
        /* clear RTC interrupt flag */
        WTCR_INT0 = 0;

        /* interrupt service code */
        …

        /* Synchronization with R-Bus TWICE*/
        RB_SYNC;
        RB_SYNC;
}
```

That forces additional read accesses to the Resource memory space which waits until the interrupt clear instruction on RTC WTCR/WTCER registers is completed, the write buffer is empty and the additional clock cycles have elapsed.

5 Document History

Document Title: AN205203 - FR60 Family, ISR Double Execution

Document Number: 002-05203

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	03/14/2006	Initial release
			03/22/2006	Several corrections
			06/23/2008	Added LIN-USART and RTC behaviour
*A	5085471	NOFL	01/14/2016	Migrated Spansion Application Note from MCU-AN-300025-E-V12 to Cypress format
*B	5840860	AESATP12	08/01/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709