Application Report
SPNA219–April 2015

Nested Interrupts on Hercules™ ARM® Cortex®-R4/5-Based Microcontrollers

Christian Herget

ABSTRACT

This application report describes what nested interrupts are and how a re-entrant interrupt handler can be implemented on Hercules-based microcontrollers. A reference implementation by ARM and an adjusted example implementation suitable for Hercules-based microcontrollers will be compared and discussed.

This document assumes that you have some basic understanding of the different operating modes of a Cortex-R4/5 CPU and how interrupts are handled on Hercules based microcontrollers. For more details, see *Interrupt and Exception Handling on Hercules ARM Cortex-R4/5-Based Microcontrollers* (SPNA218).

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spna219.

NOTE: TI assumes no liability that the discussed implementation and provided code are free from faults, compliant to certain coding guidelines, nor was it developed in accordance with certain standards. The implementer has to ensure and verify that the code conforms to appropriate rules and standards, and he has the sole responsibility to ensure and verify correct functionality in his application.

Contents

List of Figures

1 Nested Interrupts

The ARM Cortex-R4/5 (ARMv7-R architecture) processor does not support interrupt nesting in hardware, as some Cortex-M (ARMv7-M architecture) processors do. Only a two level nesting is thinkable when using IRQ and FIQ, where the FIQ can interrupt the IRQ. However, interrupt nesting can be emulated in software with the help of reentrant interrupt handlers. This document describes a possible implementation for such a reentrant IRQ handler.

Pros	Cons
• Can reduce interrupt latency for higher priority interrupts	• Interrupt latency for a single interrupt is increased, due to software overhead
• Can replace a preemptive operating system	• IRQ handler implementation in assembly is necessary
• Low-priority interrupt routines may have more relaxed execution times	• The overall system becomes more complex

NOTE: The maximum nesting level (depth) should be limited to avoid situations where the stack used for the interrupt routines over-follows or the processor gets blocked by to many interrupts pending.

2 Reentrant IRQ Handler Implementation

The ARM Cortex-R4/5 core does not support more than one IRQ to be taken at a time. This is mainly because it has only one Saved Program Status Register (SPSR) and one Link Register (LR) register. If an IRQ is interrupted by another IRQ, these CPU registers would get overwritten (corrupted) and a later restoring of the processor state would not be possible anymore. Also, the nature of the ARM C implementation has to be taken into account, so that non-callee-saved registers (R0-R3 and R12) have to be preserved between function calls. To work around these limitations, the CPU registers (SPSR, LR, R0-R3 and R12) have to be preserved on the stack, by an ISR handler. Furthermore, the CPU mode has to be switched to another mode, usually System mode, as the IRQ has to be re-enabled, which causes the current LR (used by subroutines) to be overwritten when the CPU is still in IRQ mode. For more details, see the *Reentrant interrupt handlers* section in [3].

As the ISR is executed in User or System mode and not in IRQ mode the “main” stack is used and not the IRQ exclusive stack. Usually the System mode is used, as this offers privileged access similar to the IRQ mode. Before the interrupts could be enabled again, it has to be ensured that the current interrupt source is cleared or masked, so that the ISR is not immediately interrupted by “itself”.

ARM suggests a flow to implement a reentrant IRQ handler, which is described in Section 2.1. However, this flow should be optimized to get the shortest interrupt latency and has to be extended by special Vectored Interrupt Manager (VIM) handling to work on Hercules-based MCU’s. This flow is described in detail in Section 2.2.

2.1 Suggested Flow by ARM

The following is copied from the ARM Compiler Toolchain - Developing Software for ARM Processors [3]. The steps required to re-enable interrupts safely in an IRQ handler are:

1. Construct the return address and save it on the IRQ stack.
2. Save the work registers, non callee-saved registers and IRQ mode SPSR.
3. Clear the source of the interrupt.
4. Switch to System mode, keeping IRQs disabled.
5. Check that the stack is eight-byte aligned and adjust, if necessary [2].
6. Save the User mode LR and the adjustment, 0 or 4 for architectures v4 or v5TE, used on the User mode SP.
7. Enable interrupts and call the C interrupt handler function.
8. When the C interrupt handler returns, disable interrupts.
9. Restore the User mode LR and the stack adjustment value.

10. Readjust the stack, if necessary.
11. Switch to IRQ mode.
12. Restore other registers and IRQ mode SPSR.
13. Return from the IRQ.

2.2 Modified IRQ Handler Flow to Work With VIM

The flow as described by ARM needs to be extended by special handling, needed for the VIM module. Furthermore, the flow was modified to simplify the later implementation (use of System mode stack instead of IRQ stack). In this mode, all IRQ's will be interruptible as a common software interrupt dispatcher, which is used to dispatch all IRQ level interrupts, and used to handle re-entrancy.

The IRQ handler has been divided into two functions. The first is written in assembly due to its low level nature (see Section 2.2.1). The second one is written in C to avoid complex assembly code (see Section 2.2.2). shows the program flow with a common IRQ handler (grey boxes).

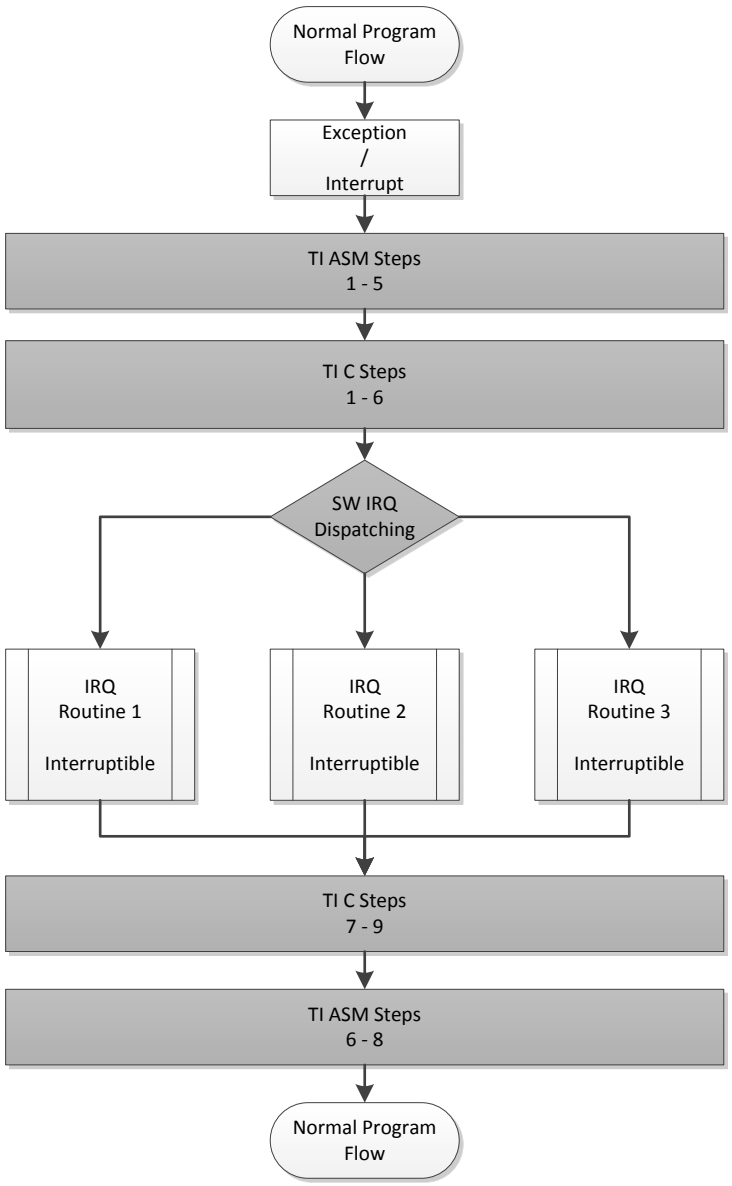

Figure 1. Program Flow With Common IRQ Handler

The discussed IRQ handler implementation does only the handling that is needed for the IRQ reentrancy. The actual interrupt prioritization is still done by the VIM.

In the example code, the actual interrupt functions cannot be interrupted by an interrupt of the same priority level. For example, the RTI Compare 1 interrupt cannot be interrupted by itself or a lower priority level interrupt like RTI Compare 2, but by a higher prior interrupt like RTI Compare 0. It furthermore includes code to keep track of the nesting deep and provides mechanism to limit the depth. This is useful to avoid situations where the stack could overflow.

Figure 2 shows an example program flow with nested interrupts. The normal program flow is interrupted by two RTI compare interrupts in the example. *RTI Compare 0* has a higher priority than *RTI Compare 1* and can, thus, interrupt *RTI Compare 1*.

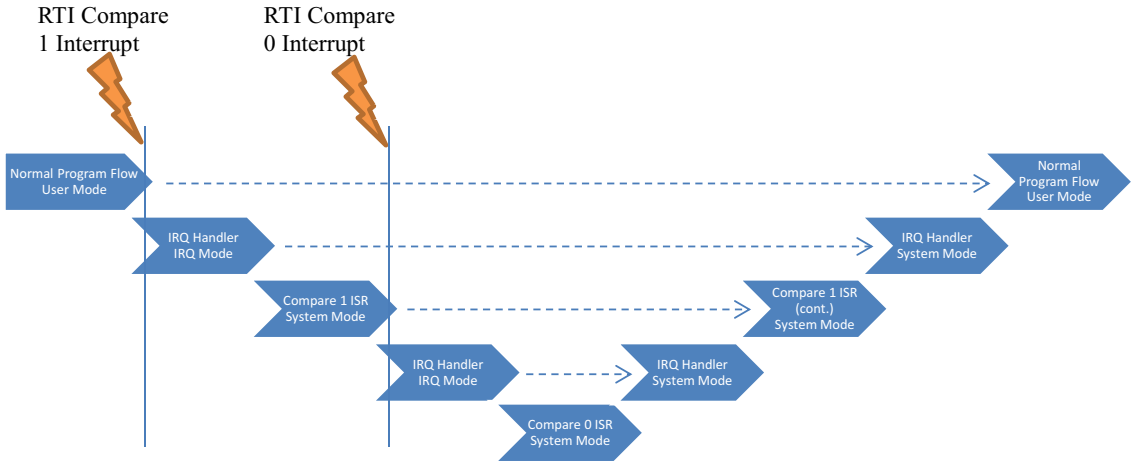

Figure 2. Example Application Program Flow

NOTE: The described implementation is only applicable for IRQ and not for FIQ, because the FIQ is implemented as a non-maskable interrupt on Hercules-based devices.

2.2.1 Assembly Level IRQ Handler Flow by TI ASM (irqDispatch_a.asm)

1. Construct the return address (related to ARM step 1), save it and the IRQ mode SPSR on the stack (related to ARM step 2).
   (a) Construct the return address.

   The return from an exception is described in the *Exception entry and exit summary* section of the *ARM Cortex-R4 and Cortex-R4F Technical Reference Manual* (ARM DDI 0363C).

   (b) Preserve the Link Register (LR) and Saved Program Status Register (SPSR). The Store Return State (SRS) onto a stack instruction saves the return state on a specific stack, in this case, the System/User Mode stack. Saving on the System Mode stack will simplify the return from the IRQ handler routine.
2. Switch to System Mode and keep IRQ disabled (related to ARM step 4). This can be done now, as the IRQ stays disabled at this point in time and, thus, the IRQ handler routine can't be interrupted by another pending IRQ.

   NOTE: This is only true for IRQ not for FIQ, as the FIQ is implemented as not mask-able on Hercules devices.
3. Save the non callee-saved registers on the stack (ARM step 2). Non callee-saved registers are R0, R1, R2, R3 and R12 they are also called AAPCS (ARM Architecture Procedure Call Standard registers). [1] This can be done after the switch to the System Mode, as these registers are not banked in IRQ mode.

NOTE: This is only true for IRQ not for FIQ.

4. Check that the stack is eight-byte aligned and adjust, if necessary [2]. Save the System/User Mode LR and the adjustment, 0 or 4 for architectures v4 or v5TE, used on the System/User Mode SP (related to ARM step 5 and ARM step 6).
5. Branch to C IRQ Handler Routine, for further processing (related to ARM step 7).
6. Undo the stack alignment and restore System Mode LR (related to ARM step 9).
7. Restore AAPCS registers, from TI ASM step 3, (related to ARM step 12).
8. Return to normal program flow from IRQ Handler (related to ARM step 13).

2.2.2 C Level IRQ Handler Flow by TI C (irqDispatch_c.c)

1. Get IRQ Index Offset Vector from VIM.
2. Get IRQ Interrupt Routine Vector from VIM.
3. Save the Interrupt Request Mask 0-2 from VIM.
4. Disable the current interrupt line and all lower priority lines in the VIM (using index from TI C step 1).
   (a) This is related to ARM step 3 as it has to be ensured that the current interrupt which is still pending does not generate an interrupt again, as soon as the IRQ is enabled.
   (b) Furthermore, lower priority interrupts usually should not be able to interrupt the current one. To emulate this all lower priority interrupts have to be disabled till the current interrupt was executed.
5. Enable IRQ's (related to ARM step 7).
6. Execute IRQ Interrupt Routine (using vector from TI C step 2).
7. Disable IRQ's (related to ARM step 8).
8. Restore the Interrupt Request Mask 0-2 in the VIM (saved in TI C step 3).
9. Return to Assembly Level IRQ Handler.

NOTE: The Example has code included to limit the max nesting level, which is not described here.

To further improve interrupt latency it is possible to implement the entire IRQ Handler in assembly.

2.3 How to Make Only Specific Interrupts Interruptible

In the previously discussed flow, a special interrupt dispatcher (the so-called IRQ handler), was used to handle the re-entrance of the interrupts and the call of the corresponding interrupt routines. Contrary to this, it is possible to add a wrapper to each IRQ function, which should be interruptible. In other words, each interrupt that should be interruptible is wrapped with its own IRQ re-entrance handler. An additional advantage is that interrupts that should not be interruptible do not suffer from the increased interrupt latency introduced by the IRQ handler, as they are still dispatched by the VIM hardware.

Figure 3 shows the program flow with only some interruptible interrupt routines. In this example, IRQ Routine 1 is not interruptible, whereas, IRQ Routines 2 and 3 are. A detailed flow of an IRQ Routine with wrapper is shown on the right hand side. The implementations of the gray boxes are discussed on the following pages.

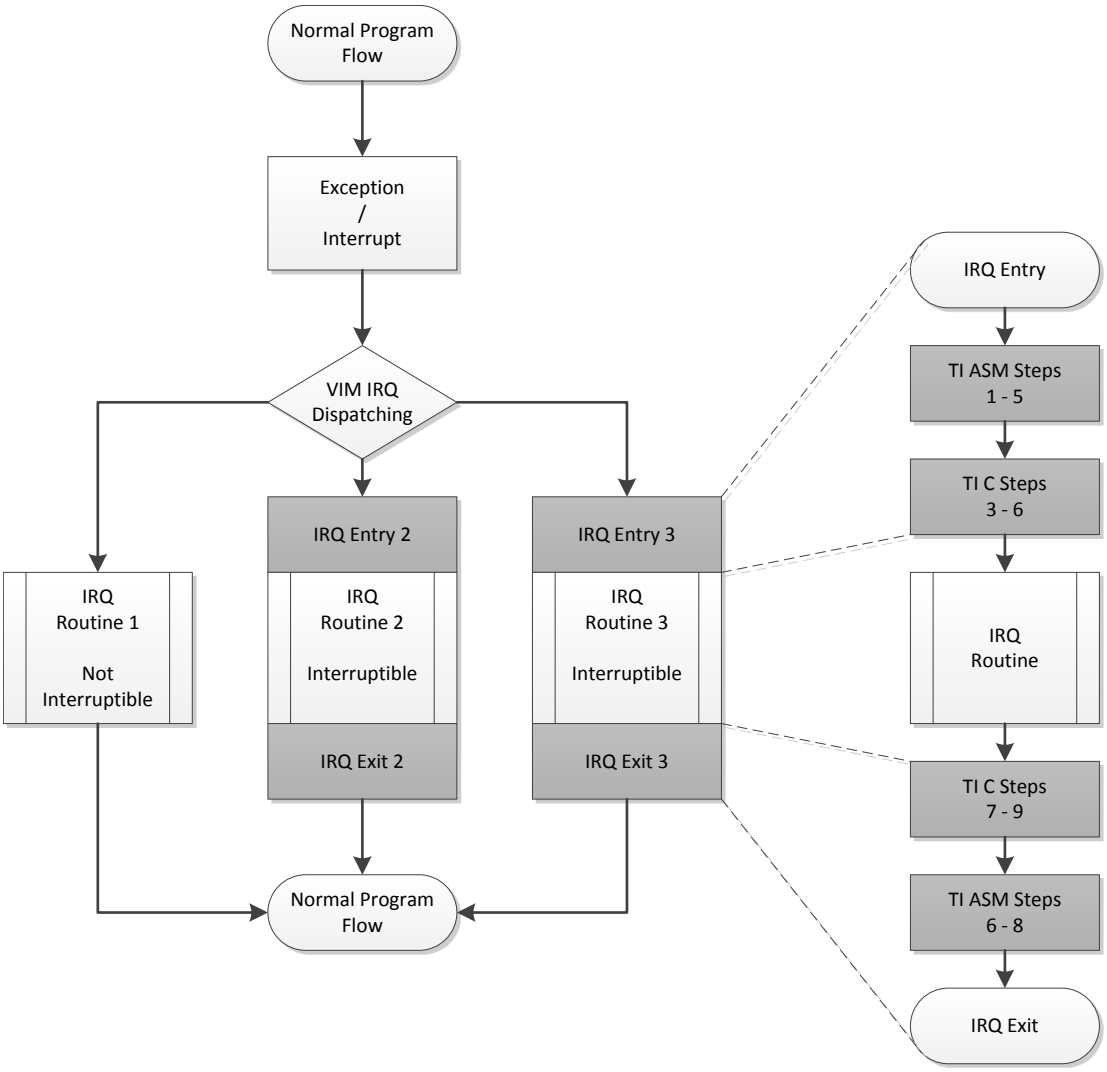

Figure 3. Program Flow With selective IRQ Wrappers

Comparable to the previously discussed flow, the re-entrance handler is divided into two parts, one which is written in assembly language and one which is written in C language. The assembly level handler is identical for each interruptible IRQ; however, it has to be added to each function individually and is potentially held several times in the program memory. The C level handler has to be written individually for each interruptible IRQ, to allow reconfiguring the interrupt line mask registers in the VIM. This is necessary to mask out lower priority interrupts and prevent them from interrupting the current one. In the previously discussed flow, it was possible to write a common routine to mask the interrupts because, all IRQ interrupt's shared the same handler for dispatching the interrupts. In this implementation, this is not possible as the pending IRQ could change in-between branching to the IRQ handler and reading the highest priority pending IRQ index. Looking back to the example in Figure 2, it would only be necessary to add the wrapper to *RTI Compare 1*, as it is the only interrupt that needs to be interruptible.

2.3.1 C Level IRQ Handler Flow for Use With Vectored Interrupt Mode

Changes to the flow discussed in Section 2.2.2 are highlighted to help to compare both.

1. ~~Get IRQ Index Offset Vector from VIM.~~
   This is not needed anymore as VIM does the dispatching.
2. ~~Get IRQ Interrupt Routine Vector from VIM.~~
   This is not needed anymore as VIM does the dispatching.

3. Save the Interrupt Request Mask 0-2 from VIM.
4. Disable the current interrupt line and all lower priority lines in the VIM ~~(using index from TI C step 1)~~. **(This part changes between the different interrupts as each has to know its own priority level).**
   (a) This is related to ARM step 3: as it has to be ensured that the current interrupt that is still pending does not generate an interrupt again, as soon as the IRQ is enabled.
   (b) Furthermore, lower priority interrupts usually should not be able to interrupt the current one. To emulate this, all lower priority interrupts have to be disabled until the current interrupt was executed.
5. Enable IRQ's (related to ARM step 7).
6. Execute IRQ Interrupt Routine ~~(Using vector from TI C Step 2)~~.
7. Disable IRQ's (related to ARM step 8).
8. Restore the Interrupt Request Mask 0-2 in the VIM (saved in TI C step 3).
9. Return to the Assembly Level IRQ Handler.

2.3.2 Proposed Implementation in C Source

As described previously, this method requires adding a special assembly wrapper around each IRQ function that should be interruptible. To avoid duplicating and modifying the assembly source for each interrupt function, the C preprocessor can be used along with inline assembly. The following C preprocessor macro can be used to declare an interruptible IRQ function and add the necessary assembly wrapper:

```
#define __REENTRANT_IRQ__(FuncName) \
__asm(  " .def " #FuncName "              ; Export Symbol\n" \
        " .arm                            ; Assemble in ARM UAL Syntax\n" \
        " .align 4                        ; Aligne to word size\n" \
        " .armfunc " #FuncName "          ; Declare an ASM function\n" \
        "                                 ; Start ASM Handler to manage IRQ Nesting\n" \
        #FuncName":                       ; Symbol\n" \
                                          ; TI ASM Step 1\n" \
        " SUB      LR, LR, #4             ; construct the return address\n" \
        " SRSDB    #31!                   ; Save LR_irq and SPSR_irq to System mode stack\n" \
        "                                 ; TI ASM Step 2:\n" \
        " CPS      #31                    ; Switch to System mode\n" \
        "                                 ; TI ASM Step 3:\n" \
        " PUSH     {R0-R3, R12}           ; Store AAPCS registers\n" \
        " .if __TI_VFPV3D16_SUPPORT__ = 1 ; If VFPV3D16 is used\n" \
        " FMRX      R12, FPSCR\n" \
        " STMFD     SP!, {R12}\n" \
        " FMRX      R12, FPEXC\n" \
        " STMFD     SP!, {R12}\n" \
        " FSTMDBD   SP!, {D0-D7}\n" \
        " .endif\n" \
        "                                 ; TI ASM Step 4\n" \
        "                                 ; Align stack to a 64 Bit boundary\n" \
        " AND      R3, SP, #4             ; Calculate Stack adjustment to 64bit boundary\n" \
        " SUB      SP, SP, R3             ; Adjust System Stack\n" \
        " PUSH     {R3, LR}               ; Put Stack adjustment and System Mode LR on Stack\n"
\
        "                                 ; TI ASM Step 5\n" \
        " BL   __CLevel"#FuncName"        ; Branch to C Level Interrupt Function\n" \
        "                                 ; TI ASM Step 6\n" \
        "                                 ; Undo stack alignment\n" \
        " POP    {R3, LR}b                ; Get Stack adjustment and System Mode LR from
Stack\n" \
        " ADD      SP, SP, R3             ; Undo System Stack adjustment\n" \
        "                                 ; TI ASM Step 7\n" \
        " .if __TI_VFPV3D16_SUPPORT__ = 1 ; If VFPV3D16 is used\n" \
        " FLDMIAD   SP!, {D0-D7}   \n" \
        " LDMFD     SP!, {R12}     \n" \
```

```
        " FMXR      FPEXC, R12     \n" \
        " LDMFD     SP!, {R12}     \n" \
        " FMXR      FPSCR, R12     \n" \
        " .endif                   \n" \
        " POP      {R0-R3, R12}           ; Restore AAPCS registers\n" \
        "                                 ; TI ASM Step 7\n" \
        " RFEIA    SP!                    ; Return using RFE from System mode stack\n" \
        "\n") ;                           /* End of ASM Function */ \
/* Modify Function name for C Level Interrupt Function */ \
```

The C code of the *rtiCompare1Interrupt* IRQ handler function as generated by HALCoGen, has to be modified as following:

```
__REENTRANT_IRQ__(rtiCompare1Interrupt)
{
    /* TI C Step 3 */
    /* Save VIM REQENASET[0,1,2] Registers for later restore */
    uint32 SaveREQMASKSET0 = vimREG->REQMASKSET0;
    uint32 SaveREQMASKSET1 = vimREG->REQMASKSET1;
    uint32 SaveREQMASKSET2 = vimREG->REQMASKSET2;

    /* TI C Step 4 */
    /* Mask out lower priority IRQ's and the current IRQ itself, keep FIQ's enabled */
    vimREG->REQMASKCLR0 = ((0xFFFFFFFFU << (4ul  - 1ul)) & (~vimREG->FIRQPR0));
    vimREG->REQMASKCLR1 = ( 0xFFFFFFFFU                  & (~vimREG->FIRQPR1));
    vimREG->REQMASKCLR2 = ( 0xFFFFFFFFU                  & (~vimREG->FIRQPR2));
    vimREG->REQMASKCLR;
                          /* Read back Mask to ensure that the previous write was finished
                             before enabling interrupts again */

    /* TI C Step 5 */
    /* Enable IRQ, to allow preemption of IRQ routine */
    _enable_IRQ();

    /* TI C Step 6 */
    /* Branch to the interrupt handler */
    rtiREG1->INTFLAG = 2U;
    rtiNotification(rtiNOTIFICATION_COMPARE1);

    /* TI C Step 7 */
    /* Disable IRQ, to protect the remainder of the dispatcher from preemption */
    _disable_IRQ();

    /* TI C Step 8 */
    /* Restore VIM REQENASET[0,1,2] Registers */
    vimREG->REQMASKSET0 = SaveREQMASKSET0;
    vimREG->REQMASKSET1 = SaveREQMASKSET1;
    vimREG->REQMASKSET2 = SaveREQMASKSET2;

    /* TI C Step 9 */
    /* Return to normal program flow from IRQ Handler Registers */
    return;
}
```

Please note the use of the *__REENTRANT_IRQ__* macro in the function declaration. Furthermore, it is visible that static code was added to store, manipulate and restore the IRQ request mask.

3 Example Projects

Code Composer Studio™ (CCS) project examples with code are included in http://www.ti.com/lit/zip/spna219 for the following devices out of the Hercules family and development kits:

- TMS570
  - TMS570LS0432 → LAUNCHXL-TMS57004
  - TMS570LS1227 → TMDS570LS12HDK
  - TMS570LS3137 → TMDS570LS31HDK
  - TMS570LC4357 → TMDX570LC43HDK
- RM4x/5x →
  - RM42L432 → LAUNCHXL-RM42
  - RM46L852 → TMDXRM46HDK
  - RM48L952 → TMDSRM48HDK
  - RM57L843 → TMDXRM57LHDK

The projects where tested with CCS v6.0.1 using TI ARM C/C++ Compiler v5.1.8.

3.1 Project Folder Structure

Each project for a certain device and development kit is in a separate folder, with the same name as the development kit. In this folder, the CCS and HALCoGen project files and generated code are included. The HALCoGen C source files *sys_main.c (HL_sys_main.c)* and *notification.c (HL_notification.c)* are the only files were User Code was added.

The IRQ dispatcher routine as discussed in Section 2.2 is included in the folder *IRQ Dispatcher*, which is used within every example CCS project. It consists of two files: *irqDispatch_a.asm* with the assembly code in it and *irqDispatch_c.c* with the C code in it.

There is one additional example project for the implementation discussed in Section 2.3, it located in the *TMDSRM48HDK_HWVecMode* folder.

3.2 Description of the Example Project Behavior

Two RTI timer interrupts are setup in every project (RTI Compare 0 and 1). Both interrupts are mapped to the CPU's IRQ line, Compare 0 has a higher priority level and interrupt frequency compared to Compare 1 (Compare 0: 1Hz / 1s vs. Compare 1: 0.25Hz / 4s).

Within both interrupts dedicated board level LED's are switched on when entering and switched off when leaving the associated interrupt routine. In the routine itself, there are delay loops implemented introducing a delay of half the period time of the RTI timer interrupts (500 ms and 2s delay). The following list shows the mappings between the LED's and interrupt routines:

- HDK's
  - LED D4 ← Compare 0
  - LED D5 ← Compare 1
- LaunchPad's
  - LED D12 ← Compare 0
  - LED D11 ← Compare 1

With this setup, it is possible to visualize the time the CPU is executing which interrupt routine, as the LED's are only on during the time the CPU is executing the corresponding interrupt routine. On a system without nested interrupts, both LED's could never be switched on at the same time, but with nested interrupts it is visible that there are time slots where both are switched on simultaneously. This behavior is visualized in Figure 4.

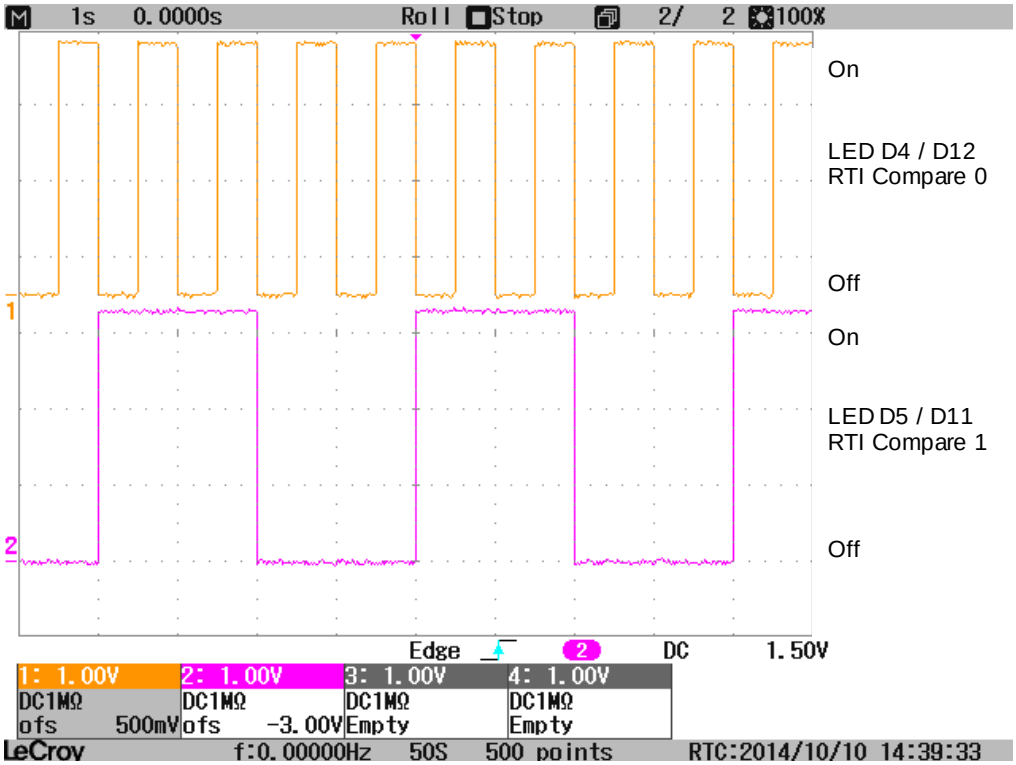

Figure 4. Oscilloscope Shot

4 References

1. *Procedure Call Standard for the ARM® Architecture* (ARM IHI 0042)
2. *ABI for the ARM® Architecture Advisory Note – SP must be 8-byte aligned on entry to AAPCS-conforming functions* (ARM IHI 0046)
3. *ARM® Compiler Toolchain - Developing Software for ARM® Processors* (ARM DUI 0471)
4. *ARM Cortex-R4 and Cortex-R4F Technical Reference Manual* (ARM DDI 0363C)
5. *ARM Cortex-R5 and Cortex-R5F Technical Reference Manual* (ARM DDI 0460D)
6. *Interrupt and Exception Handling on Hercules ARM Cortex-R4/5-Based Microcontrollers* (SPNA218)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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