Semiconductors

AN10381

Nesting of interrupts in the LPC2000

Rev. 01 — 6 June 2005 Application note

Document information

Info	Content
Keywords	nested, reentrant, interrupts, LPC2000
Abstract	Details on reentrant interrupt handlers and code examples for the same is provided

PHILIPS

Revision history

Rev	Date	Description
01	20050606	Initial version

Contact information

For additional information, please visit: **http://www.semiconductors.philips.com**

For sales office addresses, please send an email to: **sales.addresses@www.semiconductors.philips.com**

1. Introduction

This application note provides code examples for effectively handling reentrant interrupt handlers in the LPC2000 devices. The following organization is been adopted for this application note.

1. Interrupt handling overview
2. Nesting of Interrupts
3. Code examples

It is assumed for this application note, that the user is familiar with the ARM7TDMI-S architecture. Also, for code examples relating to simple interrupt handling using FIQ and IRQ, please refer to the AN10254_1 application note available online. All the code examples provided in this application note were built on the Keil MicroVision3 ARM compiler (an evaluation version of which is free for download at www.keil.com)

2. Interrupt handling overview

2.1 Interrupt levels in the ARM7 core

The ARM7 processor supports two levels of interrupts: Interrupt Request (IRQ) and Fast Interrupt Request (FIQ). ARM recommends only one interrupt source to be classified as an FIQ. All other interrupt sources could be classified as IRQ's and they could be programmed for reentrancy. Also, IRQ's could be vectored or non-vectored (In this application note, we are only considering vectored IRQ's).

2.2 Vectored Interrupt Controller (VIC)

The VIC takes all the interrupt requests from different sources and assigns them into three categories, FIQ, vectored IRQ and non-vectored IRQ. FIQ has the highest priority. Vectored IRQ's can take sixteen interrupt requests and prioritize them within sixteen vectored IRQ slots among which slot zero has the highest priority and slot fifteen has the lowest priority. Non-vectored IRQ's have the lowest priority. The VIC OR's the request from both the vectored and non-vectored IRQ's and generates the IRQ signal to the ARM7 core.

2.3 ARM7 core's response to an IRQ/FIQ

The processor does the following on an IRQ or FIQ interrupt:

1. Copies the CPSR into the SPSR for the mode in which the exception is to be handled.
2. Changes the CPSR mode bits in order to:
   a. Change to the appropriate mode and map in the appropriate banked registers for that mode.
   b. Disable interrupts, IRQ's are disabled when any exception occurs. FIQ's are disabled when an FIQ occurs and on reset.
3. Sets the LR_mode to the return address.
4. Sets the Program Counter (PC) to the vector address of the exception. This forces a branch to the appropriate exception handler.

An illustration of the Core's response to an IRQ interrupt is shown below.

1. LR_irq= address of the next instruction to be executed + 4.
2. SPSR_irq=CPSR.
3. CPSR[4:0]=10010 (Mode bits for entering IRQ mode).
4. CPSR[5]=0 (Execute in ARM state).
5. CPSR[7]= 1 (Disable interrupts).
6. PC=0x18 (Exception vector for IRQ).

During application development, the user need not worry about the ARM7 core's response to an IRQ or FIQ interrupt. Please refer to the next section on the guidelines to be followed for simple interrupt handling.

2.4 Simple interrupt handling in the LPC2000

Some of the important aspects to be considered while programming interrupts for the LPC2000 are mentioned below. For code examples on the below points, please refer to the simple interrupt handling using FIQ and IRQ (AN10254_1) application note available online.

1. On reset, the ARM7 core has interrupts disabled. Interrupts have to be enabled in the CPSR. This is usually handled by the startup assembly file which is accompanied with the example projects in the Keil environment. Most compilers provide a startup file in ARM assembly which handles the basic startup code.
2. Exception vectors should be linked and programmed correctly. This is usually managed by the linker. Also appropriate handlers need to be programmed at the respective locations. For instance at the IRQ vector (0x18) the following instruction should exist if the ISR address is read directly from the VIC Vector Address Register (Register location- 0xFFFF030).

   ```
   LDR PC[PC,#-0xFF0]
   ```
3. Stack pointers should be programmed correctly for FIQ and IRQ.
4. The VIC is programmed correctly with the ISR address. This needs to be handled in the application.
5. Compiler supported keywords are used for the Interrupt handlers. For instance in Keil, an ISR function could have the following form.

   ```
   void IRQ_Handler()__irq
   ```

   More details on compiler keywords is provided in the next section.

2.5 Compiler support for writing ISR's

ARM compilers provide keywords for both FIQ and IRQ which could be applied to a C function thereby writing the complete ISR in C. Below is the typical example from the Keil C ARM compiler:

```
void IRQ_Handler __irq{

// Clear the source of the interrupt

// Additional statements
```

```
// Update the VIC by writing to VIC Vector Address Register

}
```

By using the __irq keyword, the Compiler generates the following code for the above function

1. On function entry, the work registers (including all ARM-Thumb Procedure Call Standard (ATPCS) corruptible registers) are pushed into the stack.
2. On exit, these registers are popped.
3. The SUBS PC,R14,#4 instruction is used to return from the IRQ function. This instruction restores the PC and the CPSR.

Note: The SPSR_irq is not saved when the Compiler provided IRQ keyword is used. This is the one of the reasons why this approach cannot be used for nesting of interrupts.

2.5.1 Problems with nesting of interrupts using compiler keywords

Two main reasons for the problems to occur are related to LR_irq and SPSR_irq.

If an interrupt handler re-enables interrupts, then calls a subroutine and another interrupt occurs, the return address of the subroutine (stored in the LR_irq) is corrupted when the second IRQ is taken. Lets illustrate this problem with an example:

```
void IRQ_Handler __irq{

// reenable interrupts

foo();
}
```

When the PC jumps to IRQ_Handler (after executing the instruction at 0x18), the return address from this ISR is already stored in LR_irq. Say this is pushed into the stack (SP_irq). Now when foo() is called, the LR_irq get overwritten with the address of the instruction following foo(). Further assume that while code is executing is foo(), a second higher priority interrupt takes place. In this case, LR_irq is overwritten with the return address to foo() thereby corrupting the return address to IRQ_Handler().

Also SPSR_irq is not saved by the Compiler supported Keywords

3. Nesting of interrupts

A reentrant interrupt handler must save the IRQ state, switch processor modes and save the state of the new processor mode before branching to a nested subroutine of C function. ARM recommends switching to System mode while programming reentrant interrupt handlers.

The reason for switching modes lies in the fact that once in system mode, the active link register is LR_sys. This would be stored in the stack before the subroutine is called. When the subroutine is called, return address from the subroutine is saved in LR_sys. If a higher priority interrupt does interrupt this subroutine then the LR_irq is affected but the return address of the subroutine is still intact in the LR_sys.

3.1 Steps for Nesting of Interrupts

The steps for writing the top level ISR in Assembly would be as follows:

1. Save the registers that would be used in the ISR along with SPSR_irq.
2. Clear the source of the interrupt.
3. Switch to system mode and re-enable interrupts.
4. Save the User mode link register (LR_sys) and non-callee saved registers.
5. Call the C interrupt handler function.
6. When the C interrupt handler function returns, restore User Mode registers and disable interrupts.
7. Restore work registers and SPSR_irq.
8. Return from IRQ.

4. Code examples

This example uses Timer1 and External interrupt1 as IRQ interrupt sources. Timer1 has higher priority than External Interrupt1. The application was built for the Keil MCB2100/2130/2140 board. On the board, there are eight LEDs. There is also a push button which is been interfaced to the External Interrupt1 pin. The LEDs are separated into two groups. One set is dedicated to Timer1 and the other set is dedicated to External Interrupt1.

The External Interrupt are configured as a level-triggered interrupt. Timer Match is configured to reset the timer counter and interrupt the ARM7 core on a regular basis. In the External Interrupt1 ISR, LED1,2,3 and 4 are blinked. In the Timer1 ISR, LEDs 5,6,7 and 8 are blinked. After Timer1 ISR is executed, LEDs 1,2,3 and 4 will start blinking again (assumption- the push button is still pressed).

To see the code in action, keep the button interfaced to the External Interrupt 1 pin pressed. LEDs 1,2,3 and 4 will keep on blinking. When the Timer match takes place, it will interrupt the External Interrupt ISR and blink LEDs 5,6,7 and 8.

Two set of examples are provided:

1. Nested Interrupt Example (Top level ISR handler in Assembly).
2. Keil's approach to Nested Interrupts. (Using Inline assembly within C code).

Both approaches are the same and they are provided here for completeness. The second approach is more efficient since the user does have not have to deal with assembly files.

4.1 Nested interrupt example (top level handler in assembly)

4.1.1 Assembly code

This project has the following assembly files:

1. Startup assembly file (called Startup.s in Keil environment)- This file would have the interrupt vector table and other basic startup code. This code is not present in this application note. An example of this file can be found in the simple interrupt handling application note available online or this file is present with the example projects accompanied with Keil MicroVision3.
2. Top level ISR handler- The application can have separate files for each interrupt source used or a single file can be used. If a single file is used then while clearing the interrupt (Step 2 above), the application would have to check the VIC to get the source of the interrupt. In this example, separate files were used. The sample code provided below only shows the External Interrupt1 top level assembly file.

Top level ISR Handler:

```
// *********************************************************************
VECTADDR EQU    0xFFFFF030
EXTINT EQU    0xE01FC140

// The following four lines of code are specific to the Keil Assembler
 AREA  ISRCODE,CODE
PUBLIC isr_ext?A
 EXTERN CODE32 (ext?A)
isr_ext?A PROC CODE32

// Registers to be used in the ISR should be stacked along with SPSR_irq and LR_irq
 STMFD SP!,{R0,R1,LR}
 MRS R0,SPSR
 STMFD SP!,{R0}

// Clear the source of the interrupt
 MOV R1,#0x2
 LDR R0,=EXTINT
 STR R1,[R0]

// Move to System mode and re-enable interrupts
 MSR cpsr_c,#0x1f

// Stack lr_sys and other register
 STMFD SP!,{R0-R3,R12,LR}

// Branch to C function
 BL ext?A

// Pop lr_sys and ATPCS registers
 LDMFD SP!,{R0-R3,R12,LR}

// Move to IRQ and disable interrupts. For considering the scenario that an interrupt
// occurs while IRQ is disabled please refer the ARM FAQ online.
 MSR cpsr_c,#0x92

// Update VIC
 MOV R1,#0xff
 LDR R0,=VECTADDR
```

```
STR R1,[R0]

// Pop registers,SPSR_irq,lr_irq
LDMFD SP!,{R0}
MSR SPSR_cf,R0
LDMFD SP!,{R0,R1,LR}

// Return
SUBS PC,R14,#0x04
ENDP
END

// *********************************************************************
```

4.1.2 C code

```
#include <LPC21xx.H>                        /* LPC21xx definitions  */

void Initialise(void);

//Functions defined as seperate assembly files
extern void isr_timer(void) __irq;
extern void isr_ext(void) __irq;

// C Functions called from the Top level assembly handler
void timer_ISR() __arm;
void ext()__arm;

int main (void)
{
int i=0,j;
Initialise();

   /* Start timer */
   T1TCR=0x1;

while (1)
{
}
}

// Basic Initialzation routine
void Initialise()
{
// LED's are connected to P1.16..23
IODIR1 = 0xFF0000;                     /* P1.16..23 defined as Outputs  */
IOCLR1 = 0xFF0000;

/* Initialize Timer 1 */
T1TCR=0x0;
```

```
T1TC=0x0;
T1PR=0x4;
T1PC=0x0;

/* End user has to fill in the match value*/
T1MR0=0x...;

/* Reset and interrupt on match */
T1MCR=0x3;

/* External interrupts setup as level-sensitive triggered. Some of the steps mentioned
here are taking into considertaion the EXTINT.2 Errata   */
EXTINT=0x2;
VPBDIV=0x0;
EXTMODE=0x0;
VPBDIV=0x0;
EXTPOLAR=0x0;
VPBDIV=0x0;

/* Initialize VIC */
VICIntSelect=0x0; /* Timer 1 selected*/
VICIntEnable= 0xf020; /* Timer 1 interrupt*/
VICVectCntl1=0x2f;
//isr_ext is defined in the assembly file. Shown in the assembly code above
VICVectAddr1=(unsigned long )isr_ext;
VICVectCntl0= 0x25;
//isr_timer is defined in the assembly file. Not shown in the assembly code above
VICVectAddr0=(unsigned long)isr_timer;
}


void timer_ISR() __arm
{
     int i,j;

// Blink LEDs 5,6,7,8 five times
 for ( j=0;j<5;j++)
 {
 for(i=0;i<2000000;i++){}
     IOSET1 = 0x00F00000;
 for(i=0;i<2000000;i++){}
     IOCLR1 = 0x00F00000;
}
}

void ext()__arm
{
  int i,j;
 EXTINT=0x2;

for (j=0;j<3;j++){
```

```
for(i=0;i<2000000;i++){}
 IOSET1 = 0x000F0000;
for(i=0;i<2000000;i++){}
 IOCLR1 = 0x000F0000;
}
}
```

4.2 Keil's approach of nesting interrupts

Keil's approach implements two macros (with inline assembly code) that can be used in the beginning and end of an ISR respectively as shown in the C code below. Changes in the code below from the previous example are shown in bold.

4.2.1 C code

```
#include <LPC213x.H>                        /* LPC21xx definitions  */

// Macro for enabling interrupts, moving to System mode and relevant stack operations

#define IENABLE                          /* Nested Interrupts Entry */   \
  __asm { MRS     LR, SPSR      }    /* Copy SPSR_irq to LR     */   \
  __asm { STMFD   SP!, {LR}     }    /* Save SPSR_irq           */   \
  __asm { MSR     CPSR_c, #0x1F }    /* Enable IRQ (Sys Mode)   */   \
  __asm { STMFD   SP!, {LR}     }    /* Save LR                 */   \

// Macro for disabling interrupts, switching back to IRQ and relevant stack operations

#define IDISABLE                          /* Nested Interrupts Exit */    \
  __asm { LDMFD   SP!, {LR}     }     /* Restore LR              */   \
  __asm { MSR     CPSR_c, #0x92 }     /* Disable IRQ (IRQ Mode)  */   \
  __asm { LDMFD   SP!, {LR}     }     /* Restore SPSR_irq to LR  */   \
  __asm { MSR     SPSR_cxsf, LR }     /* Copy LR to SPSR_irq     */   \

void Initialise(void);
// Timer and External Interrupt ISR
void timer_ISR() __irq;
void ext()__irq;

int main (void)
{
int i=0;
Initialise();

/* Start timer */
T1TCR=0x1;

while (1)
{ }
}

// Basic Initialzation routine
void Initialise()
```

```
{
// LEDs are connected to P1.16..23
IODIR1 = 0xFF0000;
IOCLR1 = 0xFF0000;

/* Initialize Timer 1 */
T1TCR=0x0;
T1TC=0x0;
T1PR=0x4;
T1PC=0x0;

/* End user has to fill in the match value*/
T1MR0=0x...;

/* Reset and interrupt on match */
T1MCR=0x3;

/* External interrupts setup as level-sensitive triggered. Some of the steps mentioned
here are taking into considertaion the EXTINT.2 Errata   */
EXTINT=0x2;
VPBDIV=0x0;
EXTMODE=0x0;
VPBDIV=0x0;
EXTPOLAR=0x0;
VPBDIV=0x0;

/* Initialize VIC */
VICIntSelect=0x0;
VICIntEnable= 0xf020;
// Setting up VIC to handle the external interrupt. External Interrupt1 has priorty 1
VICVectCntl1=0x2f;
VICVectAddr1=(unsigned long )ext;

// Setting up Timer0 to handle the external interrupt. Timer0 has priorty 0
VICVectCntl0= 0x25; /* Address of the ISR */
VICVectAddr0=(unsigned long)timer_ISR;
}

// Timer1 ISR
void timer_ISR() __irq
{
     int i,j;

// Clear the Timer interrupt
 T1IR=0x1;

 IENABLE;

// Blink LEDs 5,6,7,8 five times
for ( j=0;j<5;j++)
 {
```

```
 for(i=0;i<2000000;i++){}
     IOSET1 = 0x00F00000;
 for(i=0;i<2000000;i++){}
     IOCLR1 = 0x00F00000;
}
IDISABLE;

// Update the VIC
VICVectAddr =0xff;
}

//External Interrupt 1 ISR
void ext()__irq
{
int i,j;

// Clear External Interrupt1
EXTINT=0x2;

IENABLE;

// Blink LEDs 1,2,3 and 4
for ( j=0;j<3;j++)
{
for(i=0;i<2000000;i++){}
IOSET1 = 0x000F0000;
for(i=0;i<2000000;i++){}
IOCLR1 = 0x000F0000;
}

IDISABLE;

// Update VIC
VICVectAddr =0xff;
}
```

5. References

1. ARM7TDMI-S Technical Reference Manual
2. ARM Developer Suite (v1.2) Developer Guide

6. Disclaimers

Life support — These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. When the product is in full production (status 'Production'), relevant changes will be communicated via a Customer Product/Process Change Notification (CPCN). Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

7. Trademarks

Notice — All referenced brands, product names, service names and trademarks are the property of their respective owners.

8. Contents

Date of release: 6 June 2005

Published in The Netherlands