Semiconductors

# AN10414

## Handling of spurious interrupts in the LPC2000

Rev. 01 — 4 January 2006 Application note

### Document information

| Info | Content |
|---|---|
| **Keywords** | Spurious interrupts, LPC2000, IRQ, FIQ, Interrupt |
| **Abstract** | Describes spurious interrupts and how they can be effectively handled in the ARM7 LPC2000 family |

PHILIPS

### Revision history

| Rev | Date | Description |
|---|---|---|
| 01 | 20060104 | 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**



© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

# 1. Introduction

Spurious interrupts can occur in the LPC2000 just like in any ARM7TDMI-S based microcontroller using the Vectored Interrupt Controller (VIC) and if handled correctly, spurious interrupts can be serviced just like any other interrupt request. As shown in the test cases below, they cause the interrupt to be serviced in a different handler as compared to the original interrupt handler itself. If not handled correctly, the application might experience a reset-like behavior.

In the LPC2000, spurious interrupts have occurred while using the watchdog and the UART peripherals. Since the root cause of spurious interrupts lies in the interaction of the VIC and the ARM7 core, it is recommended to always program a small handler to service these interrupts.

For more information regarding general interrupt and nested interrupt handling please refer to application notes AN10254 (Handling interrupts using IRQ and FIQ for LPC2000) and AN10381 (Nesting of interrupts) respectively.

# 2. Spurious interrupts explained

## 2.1 How can spurious interrupts occur?

Let's consider a real-life application:

1. Vectored Interrupt Controller (VIC) detects an IRQ interrupt request and sends the IRQ signal to the core.
2. Core latches the IRQ state.
3. Processing continues for a few cycles due to pipelining.
4. IRQ Handler loads Interrupt Service Routine (ISR) address from VIC.

Spurious interrupts can occur if, in step 3 the VIC state changes.This could happen under the following conditions.

1. The instruction being executed in step 3 disables interrupts.
2. The interrupt which caused the IRQ signal in the first place got cleared.

The first condition may take place while using a watchdog in the application. The second condition may take place when the RDA/CTI interrupt is enabled in the UART. Please note that using these peripherals does not necessarily mean that spurious interrupts will always take place. The timing of the interrupt coupled with the instructions in the pipeline would lead to the occurrence of spurious interrupt. It is recommended to program a spurious interrupt handler if the above peripherals are used.

## 2.2 Response to spurious interrupts

If the state of the VIC changes before the core can load the IRQ address from the VIC Vector Address Register (VICVectAddr at address 0xFFFF030) then the PC will be loaded with the value of the VIC Default Vector Address Register (VICDefVectAddr at address 0xFFFF034). On reset, this register is initialized to zero. So if this register is left at its reset state, the PC would be loaded with 0 and code will start executing from this location thereby producing a reset-like behavior (if the code is run from the on-chip flash).


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

### 2.3 Handling of spurious interrupts

As mentioned above, the VIC would provide the value of the VIC Default Vector Address register to the PC when a spurious interrupt occurs. During VIC initialization, along with providing an ISR address to the VIC Vector Address Register, please program an appropriate handler into the VIC Default Vector Address register for servicing a spurious interrupt. In this handler, the following should be done:

1. Find the source of the interrupt (if there are multiple interrupt sources).
2. Clear the interrupt source (optional as shown in the UART case).
3. Update the VIC by writing to the VIC Vector Address Register.

Consider this routine as a regular IRQ ISR routine because the core will still be in IRQ mode when the spurious interrupt is serviced. So please use the appropriate compiler directives to classify this function as a IRQ interrupt handler.

## 3. Specific instances of spurious interrupts

### 3.1 Watchdog handling

The feed sequence of the watchdog cannot be interrupted by any interrupt source. Hence before the feed sequence is applied interrupts have to be disabled in the VIC and then re-enabled after the feed sequence is completed. So a typical feed sequence would take the following form:

```
VICIntEnClr=0x... // Disabling interrupts in the VIC
WDFEED=0xAA;   //Feeding the Watchdog
WDFEED=0x55;
VICIntEnable=0x.. // Enabling interrupts in the VIC
```

The disabling of interrupts before the feed sequence may lead to the occurrence of spurious interrupts. Considering step3 (which is mentioned in section 2.1), if the instruction being executed would be the instruction that disables interrupts (before the feed sequence) then it would cause a spurious interrupt.

Below we will build an application block by block and see the effect of spurious interrupts. A simple timer interrupt-based application is considered in this case. Timer1 is configured to interrupt the core on a Match value. On match, the timer is reset and the ARM7 core is interrupted.

#### 3.1.1 Step 1: Simple timer interrupt application

Some notes on the source code provided below:

1. The spurious interrupt handler is provided but is not used until step 3. The watchdog code is commented and will be used in Step 2. The initialization of the VIC Default Vector Address with the spurious interrupt handler address is commented and will be used in Step 3.
2. The startup assembly code is not provided. Most of our tool vendors provide this assembly file as part of the tool evaluation package. Several LPC2000 application notes have sample code of the assembly startup file.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

3. The application runs at 60MHz and was run on the LPC2148. The same code can be applied to any other ARM7 LPC device.
4. The code is run from the on-chip Flash and the MAM is enabled with MAMTIM value 0x4 (this is done in the assembly startup file). The PLL is also setup in the assembly code itself. The crystal on-board is 12MHz.
5. Pins are toggled in key sections of the application so that the output is very clear on the oscilloscope. Due to the sensitivity of the timing of spurious interrupts and for the sake of clarity, the outputs have been solely captured on the oscilloscope.

The C code is provided below along with the output from the oscilloscope.

```
#include <LPC21xx.H>

void Initialise(void);
void timer_ISR() __irq;
void spurious_handler() __irq;

//*********************** MAIN ******************************
int main (void)
{
       int i;

       // Routine that initialises the LPC2000 device
       Initialise();

        // This for loop helps in identifying the start of the application
        // Port Indicator: P0.9
        for(i=0;i<100;i++)
               {
                      IOCLR0=0x200;
                      IOSET0=0x200;
               }

         // Start Timer1
        T1TCR=0x1;

         // Loop forever
         while (1)
               {
                     // Port Indicator: P0.10
                     IOCLR0=0x400;

                     //****************** Step2 ********************
                     // Uncomment the below block for Step2
                    /*VICIntEnClr=0x20;
                     WDFEED=0xAA;
                     WDFEED=0x55;
                     VICIntEnable=0x20;*/

                     IOSET0=0x400;
```



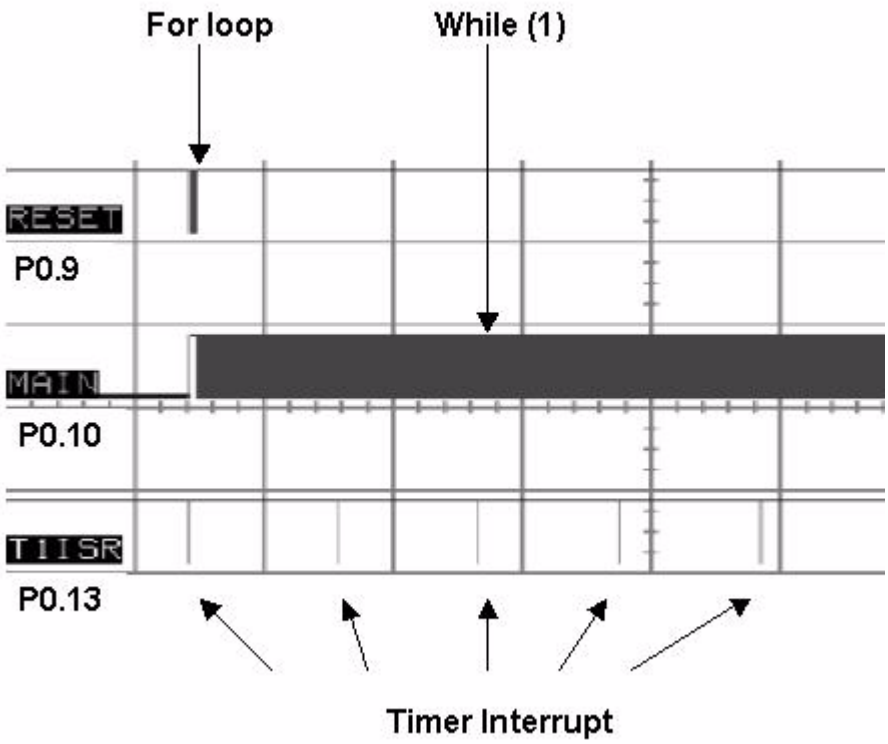
© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

```
                }
 }


//*********************** INITIALISE ******************************
void Initialise()
{
        // GPIO
         IODIR0 = 0xFFFF;
         IOSET0=  0xFFFF;

         // Setting pclk same as cclk
         VPBDIV=0x1;

         // Timer 1 Configuration
         T1TCR=0x0;
         T1TC=0x0;
         T1PR=0x0;
         T1PC=0x0;

         // Setting the Match value
         T1MR0=0xffff;

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

         // VIC Configuration
         VICIntSelect=0x0; /* Timer 1 selected*/
         VICIntEnable= 0x20; /* Timer 1 interrupt*/
         VICVectCntl0= 0x25; /* Address of the ISR */
        VICVectAddr0=(unsigned long)timer_ISR;

         //****************** Step3 *******************
          // Uncomment below statement for Step3
         //VICDefVectAddr=(unsigned long)spurious_handler;

         //****************** Step2 *******************
         // Uncomment the below block for Step2
         /* Configure the Watchdog
         WDTC=0xFFFFFFFF;
         WDMOD=0x1;
          WDFEED=0xAA;
          WDFEED=0x55;     */
}

//*********************** TIMER ISR ******************************
void timer_ISR() __irq
{
       // Port Indicator: P0.13
       IOCLR0=0x2000;
```



© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

```
        // Clearing the interrupt and updating the VIC
        T1IR=0x1;
        VICVectAddr=0xFF;

        IOSET0=0x2000;
}

//******************** SPURIOUS INTERRUPT HANDLER*********************
void spurious_handler() __irq
{
       // Port Indicator: P0.15
       IOCLR0=0x8000;

       // Clearing the interrupt and updating the VIC
       T1IR=0x1;
       VICVectAddr=0xFF;

       IOSET0=0x8000;
}
```



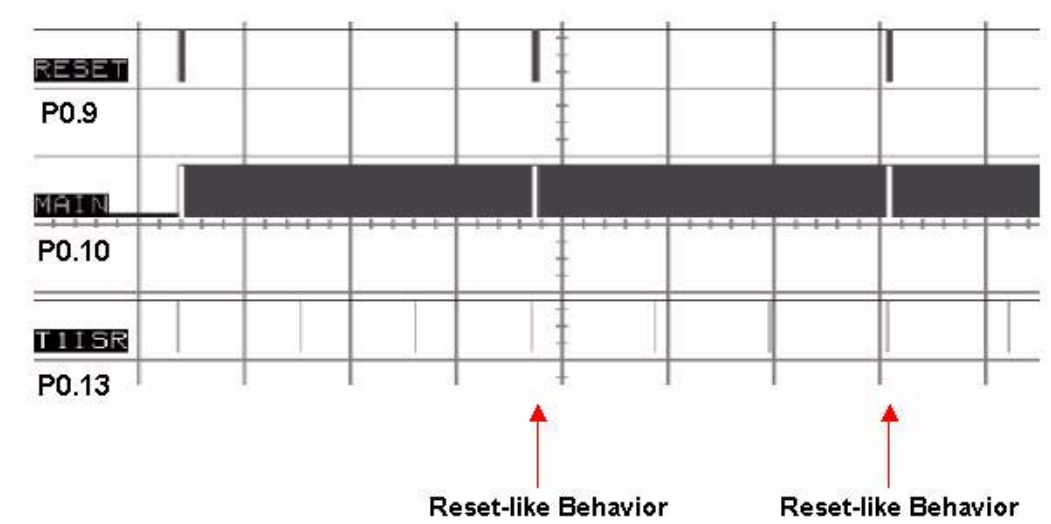

The above snapshot shows the output of Step 1 stage of the application. There are no spurious interrupts. The watchdog code is commented out and spurious interrupts are not handled. Some notes covering the above output:

1. P0.9 (For loop) gives a very good indication of the start of application.
2. The core executing While(1) loop instructions is shown by P0.10.
3. Timer1 match interrupts are occurring nearly every 1ms.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

### 3.1.2 Step 2: Adding the watchdog code (spurious interrupts occur)

Lets uncomment the watchdog code which would include the configuration of the watchdog in the Initialize() routine and the feed sequence in the while(1) loop. Once this code is added to the existing application, please build the code in your tool environment and run the application from flash. Spurious interrupts would occur and a reset-like behavior (shown by P0.9) can be noticed in the output.



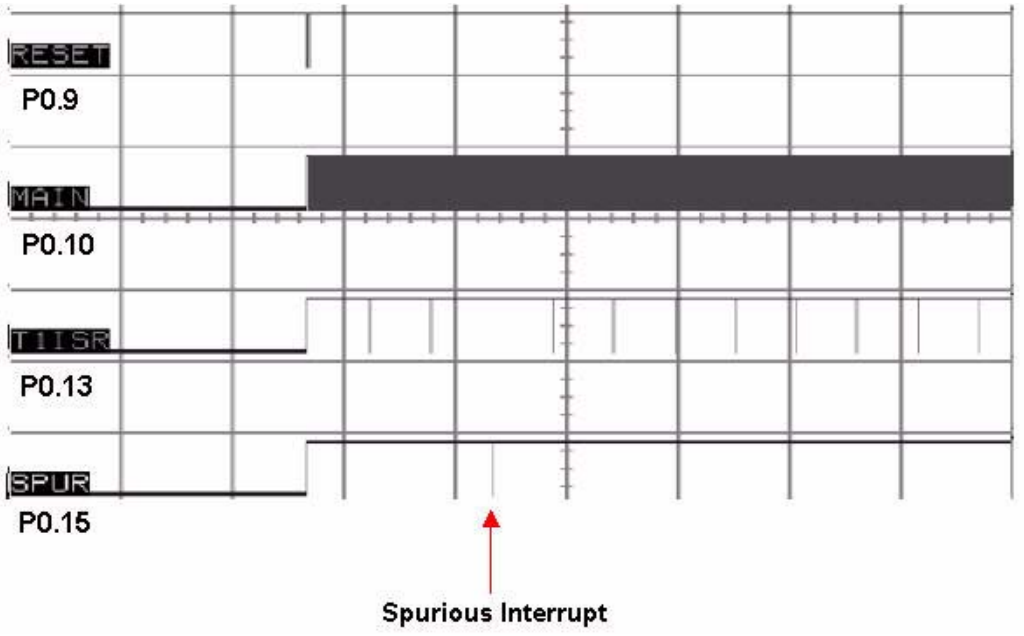

### 3.1.3 Step 3: Adding the Spurious Interrupt Handler

Now, lets uncomment the following statement in the C code.

```
// Spurious Interrupt Handler
VICDefVectAddr=(unsigned long)spurious_handler;
```

The above statement would program the VIC Default Vector Address register with the address of the spurious interrupt handler.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.




As shown above, even if a spurious interrupt took place it is handled correctly but in a different handler.

## 3.2 UART RDA/CTI interrupt

Spurious interrupts have seen to be occurring while using the Receive Data Available (RDA)/ Character Receive Time-Out (CTI) Interrupt in UART0/1. The RDA interrupt is activated when the UART Rx FIFO reaches the defined trigger level. The CTI interrupt is shared with the RDA interrupt and is activated when the UART Rx FIFO has received at least one character and no UART Rx FIFO activity has occurred in 3.5 to 4.5 character times. For more details on RDA/CTI interrupt please refer the UART chapter in the respective User Manual.

The CTI interrupt will be cleared by any Rx FIFO activity. The RDA interrupt is cleared when the Rx FIFO falls below the defined trigger level. Spurious interrupts can occur during either CTI or RDA interrupts.

Lets consider the following scenario (CTI interrupt):

1. The trigger level for the Rx FIFO is set to four and characters are transmitted over a terminal program like HyperTerminal or TeraTerm Pro.
2. Once the first character is received, the core will start getting CTI interrupts. In the interrupt service routine, a check is done if the interrupt is an CTI or RDA.
3. In this case if a CTI interrupt took place it is ignored and if an RDA interrupt occurred then the buffer is read to clear the interrupt.
4. Since the CTI interrupts would only be cleared by a FIFO activity they would keep on interrupting the core till the buffer is read or till a character is received in the FIFO.
5. Now lets consider that a CTI interrupt took place and the VIC reported this event to the core. The core latched the IRQ state. So now the core would be in step3 (Section 2.1). Lets assume that at this very moment a character is received by the UART FIFO. This would clear the CTI interrupt thereby causing a spurious interrupt.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

A similar situation might occur while dealing with RDA interrupts. Lets consider the following scenario (RDA interrupt)

1. The trigger level for the Rx FIFO is set to four and characters are transmitted over a terminal program like HyperTerminal or TeraTerm Pro.
2. Say the fourth character was received which triggered an RDA interrupt. The VIC would report this event to the core. The core latches the IRQ state.
3. Now lets assume a CTI interrupt took place earlier (to report the arrival of the previous characters) and this CTI interrupt is cleared by reading the FIFO too. Lets further assume the timing of the FIFO read happens after the core latches the RDA interrupt.
4. Reading of the FIFO would clear the CTI and RDA interrupts and this would lead to a spurious interrupt.

In the example code below, the CTI interrupts scenario is shown. Since the interrupt is cleared, when entering the handler there is no way of knowing whether a CTI or RDA interrupt caused the spurious interrupt.

So it is recommended that in the handler the buffer should be read all the time. This will avoid any characters been lost. The VIC state needs to be updated. There is no need to clear the interrupt as in the case of the watchdog since the interrupt would be already cleared.

The test setup is same as for the watchdog application. In step1, spurious interrupts would take place due to CTI interrupts (as shown with oscilloscope snapshots). In step2, the spurious interrupt handler is added. The output of this application is heavily dependent on the stream of characters coming into the devices. The snapshots shown are sample outputs taken over a particular stream of characters sent over the TeraTerm Pro.

### 3.2.1 Simple UART application

In this application, UART0 is configured to receive characters and once the buffer size reaches 4, an RDA interrupt should be triggered. C Code is as follows:

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

void uart_isr(void) __irq;
void default_ISR(void) __irq;

//*********************** MAIN ******************************

int main (void)
     {
          int i;

          // Enable GPIO for port indicators
          IODIR0 = 0xFFFF;
          IOSET0 = 0xFFFF;

          // Enable RxD0 and TxD0
          PINSEL0 = 0x00000005;

           // Initialize UART0
```


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

```
            U0LCR = 0x83;
            U0DLL = 97;
            U0LCR = 0x03;
            U0FCR=  0x47;
             U0IER=  0x1;

            // Initialize VIC
            VICIntSelect=0x0;
            VICIntEnable= 0x40;
            VICVectCntl0= 0x26;
            VICVectAddr0=(unsigned long)uart_isr;

            //******************** Step2 *********************************
            // Remove the below comment while building for step2
            // VICDefVectAddr=(unsigned long)default_ISR;

            // Delay loop so that P0.10 (set high above) remains high for sometime
            // Helps in the clarity of the scope
            for(i=0;i<2000000;i++){}

            // P0.10 would remain low for the duration of the application
            // It should only go high the code starts executing from 0x0
             IOCLR0=0x400;

             // Loop Forever
             while (1)
                 {               }

     }


//************************ UART IRQ ISR ******************************

void uart_isr() __irq
    {
        int itemp,j;
         int buffer;

        // Read U1IIR and check if it is a CTI or RDA interrupt by checking bits 3:1
      // in the UART0 Interrupt Identification Register (U0IIR)
       itemp=U0IIR;
       itemp=(itemp>>1)& 0x7;

       // If CTI interrupt then toggle port pin 13 and continue with regular
       // operation
       if (itemp==6)
              {
                    IOCLR0=0x2000;
                   IOSET0=0x2000;
                }
```



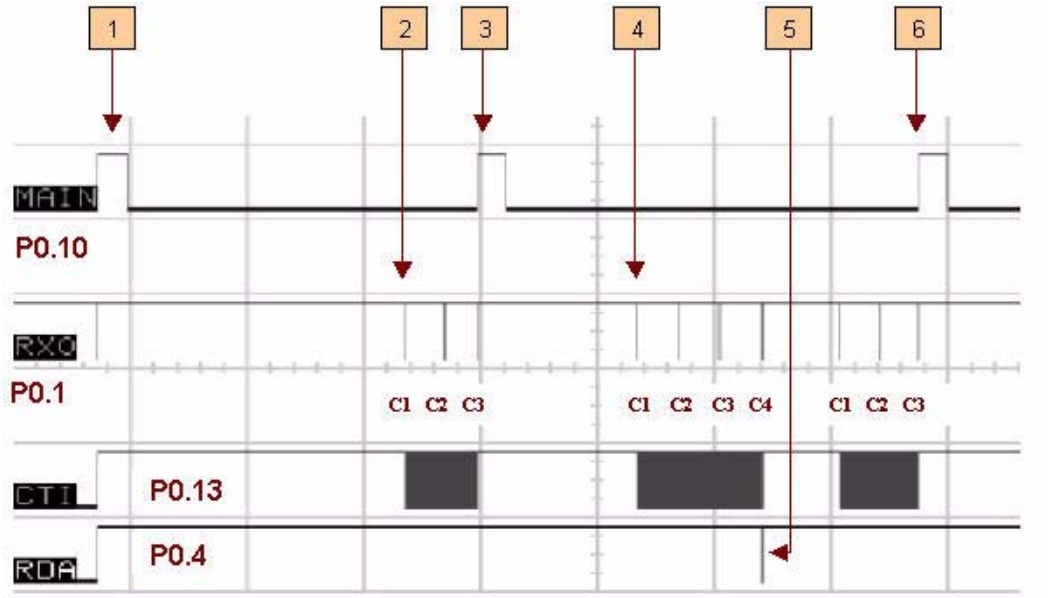
© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

```
        // If RDA interrupt then read the Rx FIFO and toggle port pin 4
        if (itemp==2)
              {
                    //Reading the FIFO
                    while (U0LSR & 0x01)
                           {
                              // The data is not important for this application
                              // so it is simply read into a variable
                               buffer=U0RBR;
                           }
                   IOCLR0=0x10;
                  IOSET0=0x10;
                }

       // Updating the VIC
       VICVectAddr=0xFF;
     }

//******************** SPURIOUS INTERRUPT HANDLER*********************

void default_ISR() __irq
    {
         int buf;

         // Port Indicator: P0.15
         IOCLR0=0x8000;
          IOSET0=0x8000;



         //Reading the FIFO
         while (U0LSR & 0x01)
               {
                      buf=U0RBR;
               }
          VICVectAddr=0xFF;

}
```

Output of the above application would be as follows:



© Koninklijke Philips Electronics N.V. 2006. All rights reserved.




The above snapshot is explained below by analyzing the different events over time:

1. The beginning of the application is shown by a pulse on pin P0.10. In the very beginning, this pin is configured as output and set high. This level is maintained by adding a huge delay loop just before the while (1) loop.

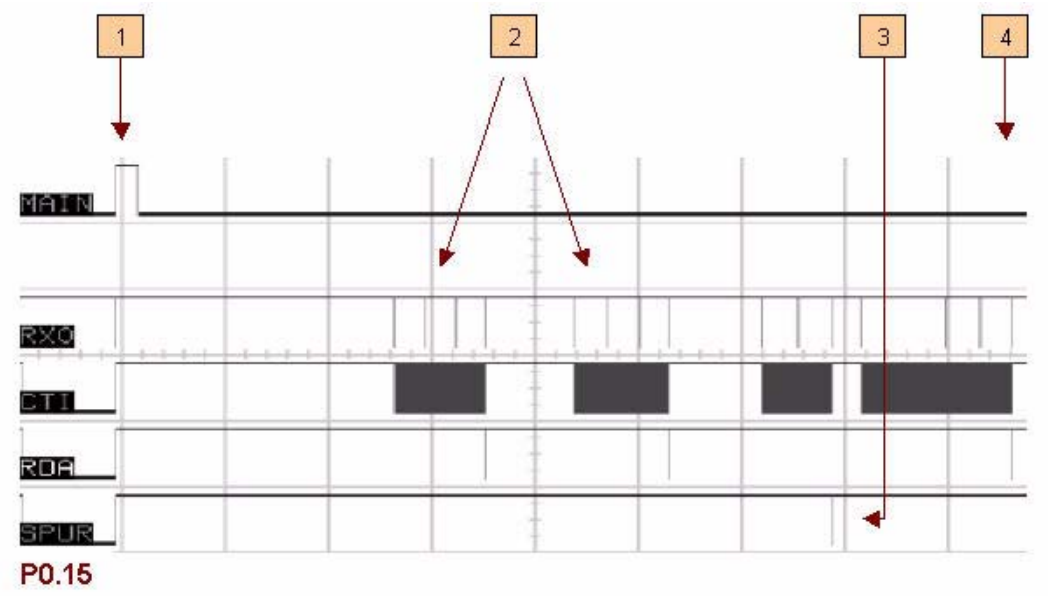
   ```
   for(i=0;i<2000000;i++){}
   ```

   Just before entering the while (1) loop, this pin is set low and should remain low throughout the application.
2. Characters (shown by cx) start streaming into UART0 (which are sent from the PC using Tera Term Pro). After the initialization condition is met, CTI interrupts are triggered. In the ISR, it could be seen that the interrupt is not cleared and a pin is toggled. The interrupt gets cleared only when a character is received by the UART.
3. At this time the third character (c3) was received and this would clear the CTI interrupt. Just before this event, the ARM7 core had latched a CTI interrupt. Going back to section 2.1, this would have been step3. This sequence of events causes a spurious interrupt takes place which is shown by a reset-like behavior in the application. The reset-like behavior is shown by P0.10 which goes high in the very beginning of the application.
4. In this second session, characters are sent again over Tera Term Pro. Spurious interrupts don't place as four characters are received.
5. RDA interrupt takes place after the Rx FIFO reaches the trigger level of four.
6. Characters are sent again and just like before a spurious interrupt takes place.

### 3.2.2 Step2: Adding the spurious interrupt handler

The below snapshot is taken after removing the comment from the statement which loads the spurious interrupt handler in the VIC Default Vector Address register.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.




The above snapshot is explained below by analyzing the different events over time:

1. Same as step 1.
2. Eight characters are received in the RxFIFO and the two RDA interrupts are respectively serviced.
3. A spurious interrupt took place on the reception of the third character. This caused the spurious interrupt handler to be called. In this handler, the buffer was read completely and the VIC is updated by writing to the VIC Vector Address Register
4. After this interrupt, an RDA is triggered after the reception of four more characters (as shown)

## 4. Summary

As the root cause of spurious interrupts lies in the interaction of the VIC and the ARM7 core they may occur while using any peripherals. So it is recommended to always have a handler that would catch a spurious interrupt if it ever takes place.


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

## 5. 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.

## 6. Trademarks

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


© Koninklijke Philips Electronics N.V. 2006. All rights reserved.

## 7. Contents



**© Koninklijke Philips Electronics N.V. 2006**

All rights are reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent- or other industrial or intellectual property rights.

**Date of release: 4 January 2006**
**Document number: AN10414_1**

**Published in The Netherlands**