

## Lab 3: Debugging and More on Interrupts

EE-500 Embedded Systems and Applications

Electrical Engineering Department, University at Buffalo

Last update: Dr. Praveen Meduri, October 2013

### 1. Objective

The objective of this lab is to learn about the different features of the debugger of uVision. We'll do this by experimenting with several examples. We'll also re-emphasize some aspects about interrupts (UART and Timer) via these examples.

### 2. uVision Debugger

The  $\mu$ Vision Debugger is completely integrated into the  $\mu$ Vision IDE. It provides many features, including the following [1]:

- Disassembly of the code on C/C++ source- or assembly-level with program execution in various stepping modes and various view modes, like assembler, text, or mixed mode
- Multiple breakpoint options including access and complex breakpoints
- Review and modify memory, variable, and register values
- List the program call tree including stack variables
- Review the status of on-chip microcontroller peripherals
- Debugging commands or C-like scripting functions
- Code Coverage statistics for safety-critical application testing
- Various analyzing tools to view statistics, record values of variables and peripheral I/O signals, and to display them on a time axis
- Instruction Trace capabilities to view the history of executed instructions

The  $\mu$ Vision Debugger offers **two operating modes**:

**1) Simulator Mode** - configures the  $\mu$ Vision Debugger as a software-only product that accurately simulates target systems including instructions and most on-chip peripherals (serial port, external I/O, timers, and interrupts; peripheral simulation capabilities vary depending on the device you have selected.). In this mode, you can test your application code before any hardware is available. It gives you serious benefits for rapid development of reliable embedded software.

**2) Target Mode** - connects the  $\mu$ Vision Debugger to real hardware. Several target drivers are available that interface to a:

- ULINK JTAG/OCDS Adapter that connects to on-chip debugging systems
- Monitor that may be integrated with user hardware or that is available on many evaluation boards
- Emulator that connects to the microcontroller pins of the target hardware
- In-System Debugger that is part of the user application program and provides basic test functions
- ULINKPro Adapter a high-speed debug and trace unit connecting to on-chip debugging systems via JTAG/SWD/SWV, and offering Cortex-M3ETM Instruction Trace capabilities

### Debug Menu

The Debug Menu of uVision IDE includes commands that start and stop a debug session, reset the CPU, run and halt the program, and single-step in high-level and assembly code. In addition, commands are 2

available to manage breakpoints, view RTOS Kernel information, and invoke execution profiling. You can modify the memory map and manage debugger functions and settings.

### Debug Toolbar

Take a moment and read pages 67-68 of the uV IDE Getting Started Guide [1]. The discussion on these pages present all the icons of the IDE related to the debugger.

### Pre-lab preparation

Please take some time now and read fully Chapters 7,8,9 from the uV IDE Getting Started Guide [1].

## 3. Example 1 – Blinky 1 Revisited

The files necessary for this example are the same as the first example of lab 1. As mentioned earlier, the  $\mu$ Vision Debugger can be configured as a Simulator or as a Target Debugger. In this example, we'll use the Simulator. Go to the Debug tab of the Options for Target dialog to switch between the two debug modes and to configure each mode. Configure to **Use Simulator**. Before running the simulation, replace the following two lines inside blinky1.c:

```
delay( 1 << 24 );
```

with:

```
delay( 1 << 14 );
```

This is to make the blinking of P2.1 faster inside the simulator; otherwise, we'd need to wait too long to actually see the corresponding bit turning 1 or 0.

### Simulation Debug

Open the Blinky1 project that you created in lab#1 or create a new project; use files from **example1** folder.

1. Click Debug menu option and select Start/Stop Debug Session. A warning about the fact that this is an evaluation version shows up; click OK.
2. Then, a new window appears where we can see the simulation of the program.
3. This window has several different supportive panels/sub-windows where we can monitor changes during the simulation. The left hand side panel, Registers, provides information regarding the Registers of LPC17xx with which we are working.
4. Again, click on the Debug menu option and select Run. The code starts simulating.
5. It is good practice that before going ahead with the actual hardware implementation to perform a debug/simulation session to make sure that our program behaves according to the design requirements.
6. In our example, we use PORT2.
7. Go to Peripherals menu option then select GPIO Fast Interface followed by Port 2.
8. You should get the window that shows P2.1 blinking.
9. Stop the simulation: Debug->Stop or hit the Stop icon from the Toolbar.

### Breakpoints

1. Let's set two breakpoints on lines:

```
LPC_GPIO2->FIOPIN |= 1 << 1; // make p2.1 high
```

```
LPC_GPIO2->FIOPIN &= ~( 1 << 1 ); // make p2.1 low
```

inside **blinky1.c**. To set a breakpoint, right-click on each of these lines, on the left margin of the panel that displays this file and then select Insert/Remove Breakpoint. 3

2. Go to Peripherals menu option then select GPIO Fast Interface followed by Port 2 to show the GPIO2 Fast Interface.
3. Debug->Run. Notice that the simulation starts and runs till the first breakpoint where it stops. Notice that P2.1 is 0. To continue the simulation click the icon “Step (F11)” once. What happens? P2.1 is turned 1 and we stepped with the simulation to the next instruction inside our program.
4. Step (F11) again more times. Observe what happens each time. While stepping inside the delay() function, observe the value of local variable “i” inside the panel labeled Call Stack + Locals on the bottom right side of the uVision IDE. Notice how “i” is incremented. To get out from within the delay() function click the icon “Step out (Ctrl-F11)”.
5. Once you get a hang of it, stop the simulation.

### Logic Analyzer

The debugger includes a logic analyzer that will allow you to study the relative timing of various signals and variable changes. It's activated using the button in the debugger. Note that you can only use the logic analyzer when you're running in the simulator, not on the board.

The logic analyzer is mostly self-explanatory. Use the Setup button to add channels. You can use symbolic names such as FIO2PIN to determine what registers or variables to watch, and you can specify a mask value (which is ANDed with the contents of the register or variable) and a right shift value (which is applied after the AND operation). You can view the result as a numeric value (“analog”) or as a bit value. At any point you can stop the updating of the screen (and/or stop the simulation itself), and change the resolution to zoom in or out. You can also scroll forward and backward in time using the scrollbar below the logic analyzer window. There are also “prev” and “next” buttons to move quickly from one transition to another.

1. Open the Logic Analyzer by clicking the icon Analysis Windows->Logic Analyzer
2. Click Setup... in the new window of the logic analyzer. Then, click New (Insert) icon and type FIO2PIN. Type in 0x00000011 as “And Mask”.
3. Go to Peripherals menu option then select GPIO Fast Interface followed by Port 2 to show the GPIO2 Fast Interface.
4. Run simulation and observe how the signal changes inside the Logic Analyzer window.

### 4. Example 2 – UART1 sends “Hello World! “ Once Only

The files necessary for this example are located in **example2** folder as part of the downloadable archive for this lab. This example is a modified (simplified) version of the example from lab#2. The simplified version uses only UART0 to simply send to the PC the string of characters “Hello World! “.

Please take a moment and read the new files **uart.h**, **uart.c**, and **uarttest.c**. Observe the differences compared to the original example from lab#2. Discuss with your team member the functionality of this new example.

To work with this example, go to your own folder where you have saved **UART\_Example\_PowerAVR /** (this is the code-bundle of lab#2) and copy the whole directory **UART\_Example\_PowerAVR /** to **UART\_Example\_PowerAVR \_modified/**. Then, replace the files **uart.h**, **uart.c**, and **uarttest.c** from **UART\_Example\_PowerAVR \_modified/** with the ones provided in folder **example2** of the downloadable archive of this lab.

Launch uVision and open the project from **UART\_Example\_PowerAVR \_modified/**. Build the project and download to the board as you did in lab#2. Use a Putty terminal (or a HyperTerminal if you use Windows XP) to see that indeed “Hello World! “ is printed out.

## 5. Example 3 – Blink LED using Timer 0 Interrupt

The file necessary for this example is located in **example3** folder as part of the downloadable archive for this lab.

First, create a new uVision project and use the provided source file, **blink1\_lec09.c**. Build and download. Observe operation and comment. Also, take some time and read the file to remember anything that it does.

### Target Debug

1. Configure the debugger to **ULINK2/ME Cortex Debugger**.
2. Click Debug menu option and select Start/Stop Debug Session.
3. Open the Logic Analyzer by clicking the icon Analysis Windows->Logic Analyzer
4. Click Setup... in the new window of the logic analyzer. Then, click New (Insert) icon and type FIO2PIN. Type in 0x00000011 as “And Mask”.
5. Go to Peripherals menu option then select GPIO Fast Interface followed by Port 1 to show the GPIO2 Fast Interface.
6. Run simulation and step through the execution of the program. Observe how the signal changes inside the Logic Analyzer window as well as inside the peripheral monitoring window. Comment.

**Due to software issues, you may not be able to see the waveform in the logical analyzer.**

## 6. Example 4 – Drawing circles on the 320x240 pixels LCD display of the PowerAVR board

You are given two versions of this example: Version 1 files are located in **lab3\_circles1** and Version 2 files are located in **lab3\_circles2**. Both versions do the same thing: plot randomly sized circles at random locations and of random colors on the 320x240 LCD display of the board. Create two different uVision projects for each version of this example. Create these projects inside the **UART\_Example\_PowerAVR /** directory with all the examples of the code-bundle used in lab#2 so that you will not need to copy standard header files from **common/**. Build and download each of the projects. Observe their operation. Read and compare the source code from **main\_circles1.c** and **main\_circles2.c**. Which version do you think is better and why? Is there anything that you would change to make this example more efficient?

## 7. Lab Assignment

Write a program that uses the LCD screen of the **PowerAVR board** to display a smiley face ☺ in the center of the screen. In your program, you should use the Timer 0 Interrupt to trigger the change of color for the smiley face every other second. The smiley face's color should alternate between yellow and red. The size of the face should be approximately the size of a quarter. The background can be any other color different from yellow and red.

**Hint:** Start with modifying any of the projects from Example 4 above. This example has already functions for drawing empty circles and lines. I have included already place holders for functions that you would need to describe/write (inside **CRIS\_UTILS.c** and **CRIS\_UTILS.h**). Then, you also need only to implement the logic of the main program by changing the main() function. The timer 0 interrupt is already set up in the Example 4 for you.

## 8. Credits and references

- [1] Keil ARM, Getting Started, Creating Applications with µVision;  
<http://www.keil.com/product/brochures/uv4.pdf> (included in lab#1 files too)
- [2] uVision IDE and Debugger; <http://www.keil.com/uvision/debug.asp>
- [3] Lab Manual for ECE455 [https://ece.uwaterloo.ca/~ece455/lab\\_manual.pdf](https://ece.uwaterloo.ca/~ece455/lab_manual.pdf)

## APPENDIX A: More on Interrupts (based in part on [3])

The LPC1768 microprocessor can have many sources of interrupts. All the internal peripherals are capable of generating interrupts. The specific conditions that produce interrupts can be set individually for each peripheral. The individual interrupts can be enabled or disabled using a set of registers (think of memory locations in the “memory space”).

Selected GPIO pins can also be set to generate interrupts. The push button INT0 is connected to pin P2.10 of the LPC1768 microprocessor. This pin can be a source of external interrupts to the MCU. The table below shows different functionalities that can be assigned to P2.10 pin.

If you plan to use P2.10 as GPIO, then you should also enable this source of interrupt as described in section 9.5.6 of the LPC17xx user manual. Note that you can set the P2.10 pin to be sensitive to either the rising edge or the falling edge. More information on clearing the interrupt pending bit can be found in table 123 in section 9.5.6.1, page 139 of the user manual.

To write an **interrupt handler** in C we need to describe/create a function with an appropriate name and it will automatically be used (it will be called automatically via the pointers stored inside the **vector table**). The name of this function consists of the prefix from the table below plus the keyword “**Handler**” appended (e.g., **TIMER0\_IRQHandler**).

---

Int#	Prefix	Description
------	--------	-------------

---

0	WDT_IRQ	Watchdog timer
1	TIMER0_IRQ	Timer 0
2	TIMER1_IRQ	Timer 1
3	TIMER2_IRQ	Timer 2
4	TIMER3_IRQ	Timer 3
5	UART0_IRQ	UART 0
6	UART1_IRQ	UART 1
7	UART2_IRQ	UART 2
8	UART3_IRQ	UART 3
9	PWM1_IRQ	PWM 1 (not used on MCB1700)
10	I2C0_IRQ	I2C 0 (not used on MCB1700)
11	I2C1_IRQ	I2C 1 (not used on MCB1700)
12	I2C2_IRQ	I2C 2 (not used on MCB1700)
13	SPI_IRQ	SPI (used for communicating with LCD display)
14	SSP0_IRQ	SSP 0 (not used on MCB1700)
15	SSP1_IRQ	SSP 1 (not used on MCB1700)
16	PLL0_IRQ	PLL 0 (interrupts not used by our labs)
17	RTC_IRQ	Real-time clock
18	EINT0_IRQ	External interrupt 0
19	EINT1_IRQ	External interrupt 1 (not used on MCB1700)
20	EINT2_IRQ	External interrupt 2 (not used on MCB1700)
21	EINT3_IRQ	External interrupt 3 (not used on MCB1700) & GPIO interrupt
22	ADC_IRQ	ADC end of conversion
23	BOD_IRQ	Brown-out detected (not used)
24	USB_IRQ	USB
25	CAN_IRQ	CAN
26	DMA_IRQ	DMA
27	I2S_IRQ	I2S (not used on MCB1700)
28	ENET_IRQ	Ethernet
29	RIT_IRQ	Repetitive-interrupt timer
30	MCPWM_IRQ	Motor control PWM
31	QEI_IRQ	Quadrature encoder
32	PLL1_IRQ	USB phase-locked loop
33	USBActivity_IRQ	USB activity
34	CANActivity_IRQ	CAN activity 8

---

A particular peripheral can generate its interrupts for a variety of reasons, which are configured within that peripheral. For example, timers can be configured to generate interrupts either on match or on capture. The priorities of the interrupts can be set individually. See sections 6.5.11 to 6.5.19 of the user manual for details.

A set of functions is available for enabling and disabling specific interrupts, setting their priority, and controlling their pending status (find them inside **core\_cm3.h** file, which is the so called CMSIS Cortex-M3 Core Peripheral Access Layer Header File):

```
void NVIC_EnableIRQ(IRQn_Type IRQn)
void NVIC_DisableIRQ(IRQn_Type IRQn)
void NVIC_SetPriority(IRQn_Type IRQn, int32_t priority)
uint32_t NVIC_GetPriority(IRQn_Type IRQn)
void NVIC_SetPendingIRQ(IRQn_Type IRQn)
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)
IRQn_Type NVIC_GetPendingIRQ(IRQn_Type IRQn)
```

The IRQn names are just the prefix from the table above with an “n” appended (e.g., TIME0\_IRQn).

We can also enable or disable interrupts altogether using `__disable_irq()` and `__enable_irq()`.

We can also trigger any interrupt in software (inside our C programs) by writing the interrupt number to the **NVIC->STIR** register (values up to 111 are permitted). We must clear interrupt conditions in the interrupt handler. This is done in different ways, depending on what caused the interrupt. For example, if we have **INT0** configured to generate an interrupt, you would clear it by setting the low-order bit of the **LPC\_SC->EXTINT** register.

For detailed descriptions of all interrupts, you should read Chapter 6 of the NXP LPC17xxx User Manual.

## APPENDIX B: Listing of source file blink1\_lec09.c used in Example 3 of this lab

```
//
// this is a simple example, which turns one of the PowerAVR board's LEDs
// on/off; it uses a Timer 0 interrupt; we discussed it in lecture#9 in
// class;
//
#include "LPC17xx.h"
int main (void)
{
    // (1) Timer 0 configuration (see page 490 of user manual)
    LPC_SC->PCONP |= 1 << 1; // Power up Timer 0 (see page 63 of user manual)
    LPC_SC->PCLKSEL0 |= 1 << 2; // Clock for timer = CCLK, i.e., CPU Clock (page
    56 user manual)
    // MR0 is "Match Register 0". MR0 can be enabled through the MCR to reset
    // the Timer/Counter (TC), stop both the TC and PC, and/or generate an
    interrupt
    // every time MR0 matches the TC. (see page 492 and 496 of user manual)
    LPC_TIM0->MR0 = 1 << 23; // Give a value suitable for the LED blinking

    // frequency based on the clock frequency
    // MCR is "Match Control Register". The MCR is used to control if an
    // interrupt is generated and if the TC is reset when a Match occurs.
    // (see page 492 and 496 of user manual)
    LPC_TIM0->MCR |= 1 << 0; // Interrupt on Match 0 compare
    LPC_TIM0->MCR |= 1 << 1; // Reset timer on Match 0
    // TCR is "Timer Control Register". The TCR is used to control the Timer
    // Counter functions. The Timer Counter can be disabled or reset
    // through the TCR. (see page 492 and 494 of user manual)
    LPC_TIM0->TCR |= 1 << 1; // Manually Reset Timer 0 (forced);
```

```

LPC_TIM0->TCR &= ~(1 << 1); // Stop resetting the timer
// (2) Enable timer interrupt;
// TIMER0_IRQn is 1, see lpc17xx.h and page 73 of user manual
NVIC_EnableIRQ(TIMER0_IRQn); // see core_cm3.h header file
// (3) Some more one-time set-up's;
LPC_TIM0->TCR |= 1 << 0; // Start timer (see page 492 and 494 of user manual)
LPC_SC->PCONP |= ( 1 << 15 ); // Power up GPIO (see lab1)
LPC_GPIO2->FIODIR |= 1 << 1; // Put P2.1 into output mode. LED is connected
to P2.1
// (4) infinite loop;
while (1) // Why do we need this?
{
    // do nothing
}
return 0;
}
// Here, we describe what should be done when the interrupt on Timer 0 is
handled;
// We do that by writing this function, whose address is "recorded" in the
vector table
// from file startup_LPC17xx.s under the name TIMER0_IRQHandler;
void TIMER0_IRQHandler(void)
{
    // IR is "Interrupt Register". The IR can be written to clear interrupts. The
    IR
    // can be read to identify which of eight possible interrupt sources are
    // pending. (see page 492 and 493 of user manual)
    if ( (LPC_TIM0->IR & 0x01) == 0x01 ) // if MR0 interrupt (this is a sanity
    check);
    {
        LPC_TIM0->IR |= 1 << 0; // Clear MR0 interrupt flag (see page 492 and 493 of
        user manual)
        LPC_GPIO2->FIOPIN ^= 1 << 1; // Toggle the LED (see lab1)
    }
}

```