Lab 6: Interrupt-Driven I/O

Description

The purpose of this lab is to familiarize yourself with the use of *interrupts*. Interrupts are a very powerful mechanism that can be used to support multitasking, handling of exceptional conditions (i.e., unexpected results during calculation such as a division by zero), emulation of unimplemented (in hardware) instructions, single-step execution for debugging, and operating system calls/protection among others.

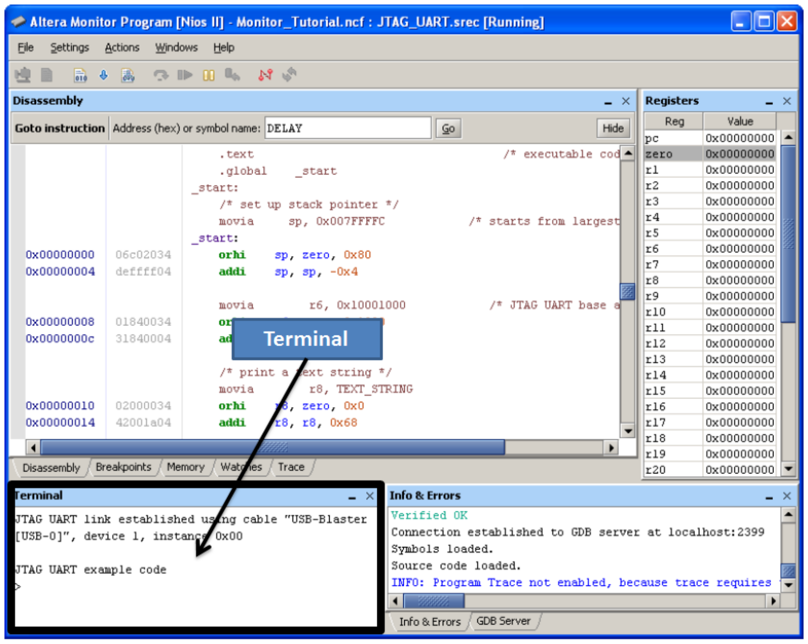
In this lab, you will extend your solution to Lab 5 to communicate with two new devices, the JTAG UART and timer, using interrupts, *while* controlling the Car World game. Specifically, you are to display the current speed of the car or the current state of the sensors in the Monitor Program terminal, using key presses to switch between these two values:

- Once every second, triggered by a timer interrupt, print the current car speed *or* the current sensor state to the Monitor Program terminal, using the JTAG UART device (This is a second JTAG UART, not the same one used in Lab 5). Print both values in hexadecimal. When printing the value, the new displayed value should replace the old value in the terminal, rather than continually printing new values one after the other.
- When the user presses a key, detected using a JTAG UART read interrupt, change the current display mode: 's' should cause the car **s**peed to be displayed each second, while 'r' should cause the car senso**r**s to be displayed each second. Pressing other keys should be ignored. Start your program in "sensors" mode.
- You must use interrupts to read key presses from the JTAG UART and to be notified of timer timeouts. You should not use interrupts for writing to the terminal JTAG UART. You should not use interrupts to control the Car World game.
- To make things easier, do not communicate with the Car World UART inside the interrupt handler. Rather, save the speed and the sensors state to a dedicated memory location ("global variable") when processing Car World UART packets in your main program loop. In the timer interrupt handler, display the saved slightly-stale value. (Why: What happens if an interrupt occurs in the middle of sending a packet, or while waiting for a response?)

Background

How to print to the terminal

The characters sent to the terminal JTAG UART will be displayed on the terminal window. For example, sending byte `0x34` to the terminal JTAG UART will display character '4'. In the other direction, typing in the terminal window will send data to the terminal JTAG UART (also using ASCII code).


Altera Monitor Program [Nios II] - Monitor_Tutorial.ncf : JTAG_UART.srec [Running]
File Settings Actions Windows Help
Disassembly
Goto instruction
Address (hex) or symbol name: DELAY
Go
Hide
Terminal
Disassembly Breakpoints Memory Watches Trace
Registers
Info & Errors
Info & Errors GDB Server

```
                                .text                                /* executable cod
                                .global    _start
                            _start:
                                /* set up stack pointer */
                                movia     sp, 0x007FFFFC           /* starts from largest
                            _start:
0x00000000    06c02034         orhi     sp, zero, 0x80
0x00000004    deffff04         addi     sp, sp, -0x4

                                movia       r6, 0x10001000          /* JTAG UART base a
0x00000008    01840034         or
0x0000000c    31840004         ad

                                /* print a text string */
                                movia       r8, TEXT_STRING
0x00000010    02000034         orhi     r8, zero, 0x0
0x00000014    42001a04         addi     r8, r8, 0x68
```

Reg	Value
pc	0x00000000
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000000
r18	0x00000000
r19	0x00000000
r20	0x00000000

```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00

JTAG UART example code
>
```

```
Verified OK
Connection established to GDB server at localhost:2399
Symbols loaded.
Source code loaded.
INFO: Program Trace not enabled, because trace requires
```

The Monitor Program's terminal interprets certain character sequences ("escape codes") as commands that control the terminal's output, such as clearing the window or moving the cursor. These escape codes can be found on page 24 of the Monitor Program Tutorial. The table is reproduced below. **`<ESC>`** represents a character with value 27 (0x1b). For example, sending the four-character sequence **`<ESC> [2K`** (`1b 5b 32 4b`) will erase an entire line.

Character Sequence	Description
`<ESC>[2J`	Erases everything in the Terminal window
`<ESC>[7h`	Enable line wrap mode
`<ESC>[7l`	Disable line wrap mode
`<ESC>[#A`	Move cursor up by # rows or by one row if # is not specified
`<ESC>[#B`	Move cursor down by # rows or by one row if # is not specified
`<ESC>[#C`	Move cursor right by # columns or by one column if # is not specified
`<ESC>[#D`	Move cursor left by # columns or by one column if # is not specified
`<ESC>[`$\#_1$`;`$\#_2$`f`	Move the cursor to row $\#_1$ and column $\#_2$
`<ESC>[H`	Move the cursor to the home position (row 0 and column 0)
`<ESC>[s`	Save the current cursor position
`<ESC>[u`	Restore the cursor to the previously saved position
`<ESC>[7`	Same as `<ESC>[s`
`<ESC>[8`	Same as `<ESC>[u`
`<ESC>[K`	Erase from current cursor position to the end of the line
`<ESC>[1K`	Erase from current cursor position to the start of the line
`<ESC>[2K`	Erase entire line
`<ESC>[J`	Erase from current line to the bottom of the screen
`<ESC>[1J`	Erase from current cursor position to the top of the screen
`<ESC>[6n`	Queries the cursor position. A reply is sent back in the format `<ESC>[`$\#_1$`;`$\#_2$R, corresponding to row $\#_1$ and column $\#_2$.

How to Use Interrupts

Interrupts must be enabled in three places:

1. **Enable interrupt on the peripheral.**
   For example, to configure the Timer to interrupt the processor when it times out, you need to set the 0th bit of the Timer's Control Register to 1.
2. **Unmask the IRQ line corresponding to the peripheral on the processor.**
   This is done using the processor's `ienable` control register (`ctl3`). For example, the Timer is associated with IRQ line 0, so you unmask this interrupt line by setting bit 0 of `ienable` to high.
3. **Enable interrupts globally in the processor.**
   This is done by setting bit 0 high in the processor's `status` (`ctl0`) register.

Determining the Source of an Interrupt

1. **Find out what device caused the interrupt.**
   The `ipending` (`ctl4`) control register tells you which IRQ line(s) are requesting an interrupt. For example if bit 0 of `ipending` is high you know that the Timer is requesting an interrupt. Multiple interrupts may happen simultaneously, therefore multiple bits of the register `ipending` can be 1 at the same time.
2. **Find out what event caused the interrupt.**
   Some devices, such as the Timer, have only one event that causes an interrupt, so the previous step is sufficient to determine the source of the interrupt. Some devices can have multiple events that cause interrupts, such as the JTAG UART with both read and write interrupts. In this case you may need to check the device's control register(s) to see what event(s) actually caused the interrupt.

Returning from an Interrupt

When a device interrupt occurs, the processor jumps to the ISR. However, we need to be able to return to the code that was executing before interrupt happened. The processor saves the address of the instruction *after* the instruction that was aborted in the **`ea`** (exception address) register. Therefore you must adjust the saved address in `ea` so that the processor re-executes the aborted instruction, by subtracting 4 from `ea` (each instruction is 4 bytes), then executing the **`eret`** (exception return) instruction to return to normal program execution.

How to debug interrupts

The debugger shows the state of the `status`, `ienable`, and `ipending` registers.

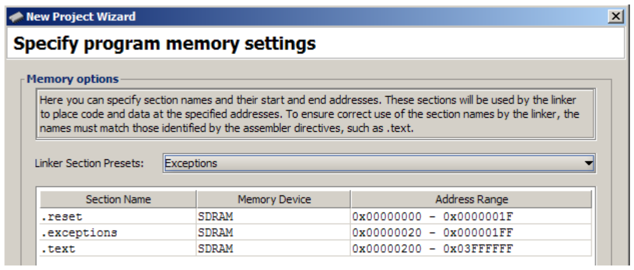
- To see whether interrupts are enabled, observe the `status` and `ienable` registers in the debugger.
- To see whether an interrupt is currently being requested by a device, observe the `ipending` register
- Use a breakpoint at 0x20 to see whether the interrupt handler executes.

Where to Place your Interrupt Service Routine (ISR)

When a processor is interrupted, it causes the processor to jump to location **`0x00000020`**. Your interrupt service routine must be located at this exact location. To do this, place the ISR code in the `.exceptions` section (rather than the usual "`.text`" section). The `.exceptions` section always starts at location 0x20. Use the code below as a template.

```
.section .exceptions, "ax"
myISR:
  [interrupt service routine starts here at 0x20]
```

When compiling your code, you have to ensure that sections `.exceptions` (your ISR) and `.text` (the rest of the code) are placed in non-overlapping memory regions. To do this, change the "Linker Section Preset" to "Exceptions" when creating a new project to change the starting locations of the various sections to suitable values (`.exceptions` to 0x20 and `.text` to 0x200 by default). Overlapping sections result in an error similar to "`section .exceptions loaded at [...] overlaps section .text loaded at [...]`"



Specify program memory settings

Memory options

Here you can specify section names and their start and end addresses. These sections will be used by the linker to place code and data at the specified addresses. To ensure correct use of the section names by the linker, the names must match those identified by the assembler directives, such as .text.

Linker Section Presets: Exceptions

Section Name	Memory Device	Address Range
.reset	SDRAM	0x00000000 - 0x0000001F
.exceptions	SDRAM	0x00000020 - 0x000001FF
.text	SDRAM	0x00000200 - 0x03FFFFFF

After compiling your code, verify that your ISR was actually placed at address `0x00000020`. You can do this by looking at the disassembly in the Monitor program. Typos are a common reason for the ISR to be placed incorrectly.

Links

- How Nios II interrupts work
- Section 8.1: How to use and debug interrupts in the Monitor Program
- JTAG UART (`ff201000`) documentation.
- Timer (`ff202000`) documentation.

Preparation and Quiz (3 marks)

- Read your class notes related to interrupts and this summary of Nios II interrupts.
- Read section 8.1. of the Altera Monitor Program Tutorial on how to use interrupts in the Monitor Program.

1. How do you tell the JTAG UART to send read interrupts?
2. How do you tell the timer to send timeout interrupts?
3. How do you tell the processor to accept interrupts from both Timer1 and the terminal JTAG UART?
4. Write a line of code to adjust ea before returning from the interrupt handler so that the aborted instruction will be re-executed.
5. Which registers must you backup before overwriting inside an interrupt handler?
6. If you want to call a function implemented in C from inside your interrupt handler, which registers must you back up?
7. Write a simple test program to blink an LED with a period of 1 second using timer interrupts.
8. Write a simple test program to echo characters on the terminal JTAG UART using read interrupts.
9. Enhance your solution to Lab 5 to display speed and sensor state over the JTAG UART, as described above. Use interrupts both to read from the terminal JTAG UART and to check the Timer for a timeout.

Be prepared to answer any questions about interrupts on Nios II.

In Lab (2 Marks)

1. Demonstrate your program. If your In Lab program doesn't work, you can demonstrate your simple test programs from the Preparation for 0.5 marks each.
2. Submit your code on blackboard. Put all .s files that you demonstrated to your TA in a zip file called code.zip.

Notes

- Try to get interrupt initialization, triggering, and returning working using a simple main program (infinite loop?) before combining this with your Lab 5 program. Use the debugger to verify that interrupts are triggering and returning.
- You are encouraged to reuse your solution to Lab 5. If done right, this lab will require very little modification to your existing code.
- When trying to type into the Terminal, be sure that the Terminal window is focused (by clicking on it). Otherwise, the key pressed will not be captured by the terminal JTAG UART.