Lab 3: 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.

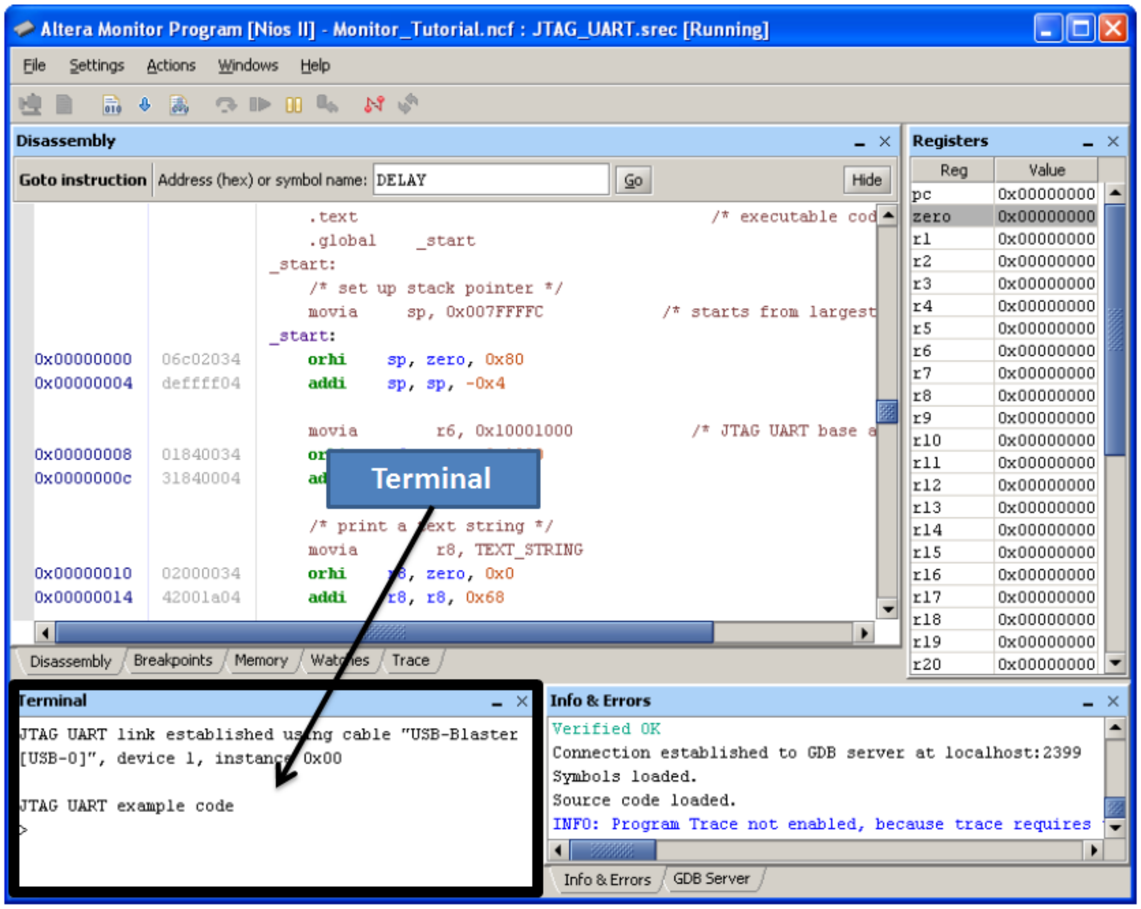
In this lab, you will extend your solution to Lab 2 to communicate with two new devices, the JTAG UART and timer, using interrupts, *while* controlling the Car World game. Specifically, you are to display the current speed of the car or the current state of the sensors in the Monitor Program terminal, 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). To send characters, make sure the terminal has focus by clicking on top of it.



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
```

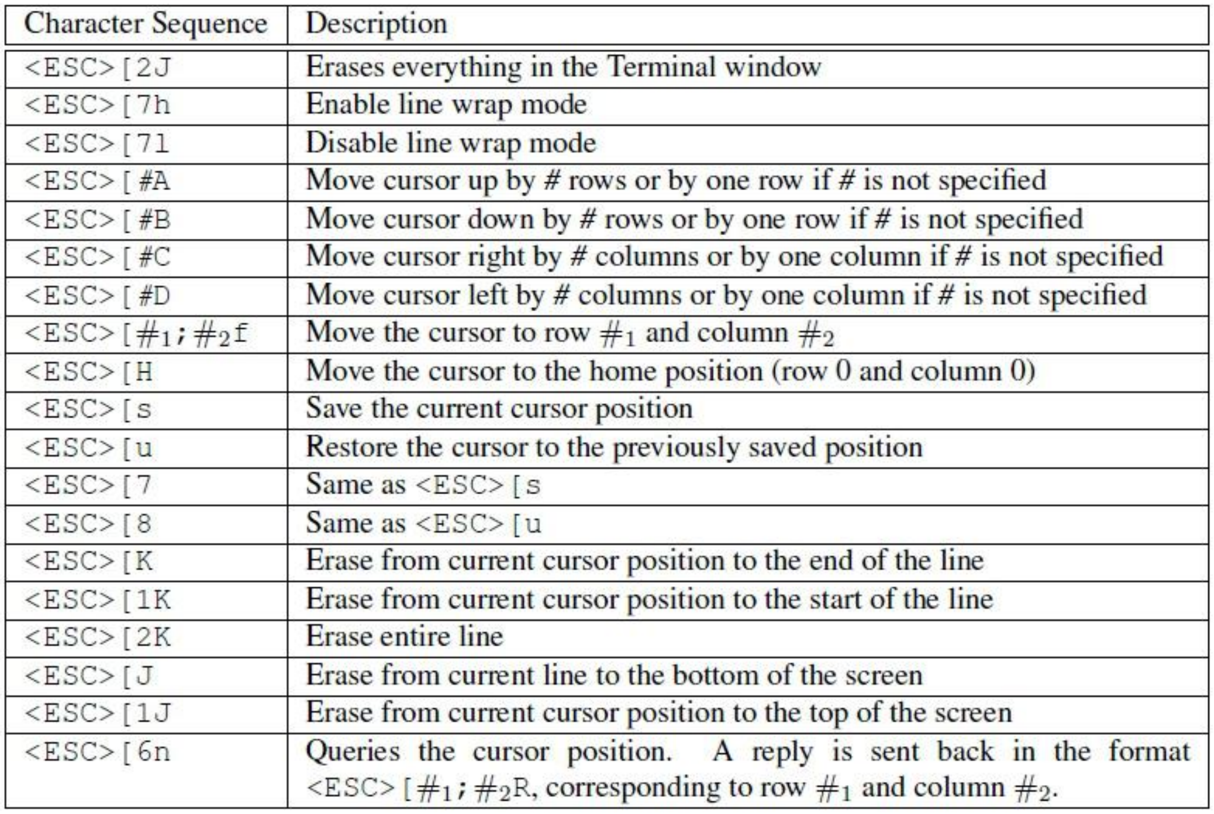
Reg	Value
pc	0x00000000
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000000
r18	0x00000000
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 23 of the Monitor Program Tutorial. The table is reproduced below. **`<ESC>`** represents a character with value 27 (0x1b). For example, sending the four-byte 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

In order to simplify the task of setting up interrupts, you will be given example code that initializes the Generic Interrupt Controller (GIC) for you, so that the interval timer and JTAG devices will be able to send their interrupts to the ARM core.

The parts you have to fill in are:

```
    /* ECE352 - CHECK AND HANDLE INTERRUPTS HERE */          (line 92)
```

Here, the interrupt ID has been placed in **R5** for you. You have to find out what device caused the interrupt, and handle it appropriately. The return code is also written for you.

```
    /* ECE352 - CONFIGURE INTERVAL TIMER HERE */             (line 107)

    /* ECE352 - CONFIGURE JTAG DEVICE HERE */                (line 114)
```

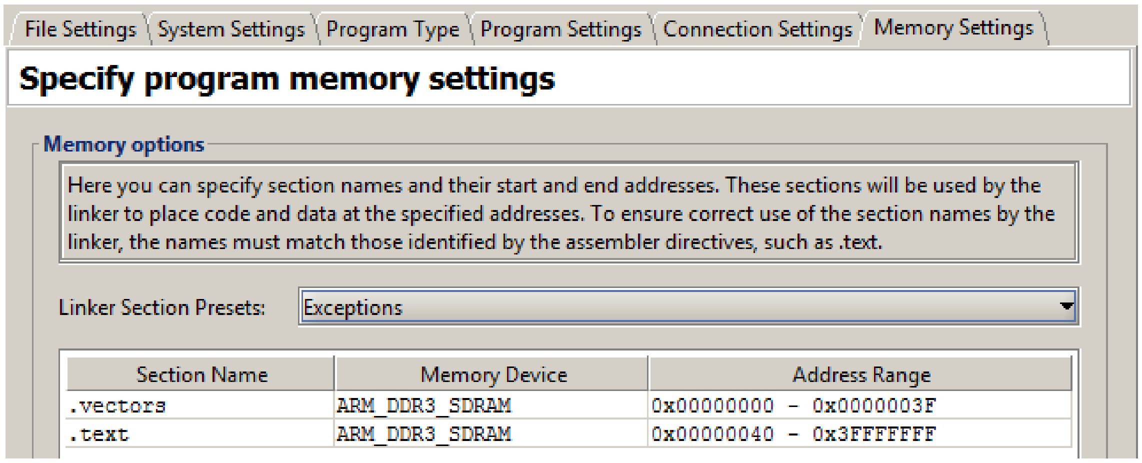
Here, you must configure the peripherals to enable their interrupts (and in the case of the timer, configure its period).

```
    /* ECE352 - MAIN PROGRAM */                              (line 140)
```

Your Car World driver program will go here. As discussed in the lab description, it will need to additionally store the sensor and speed data somewhere the interrupt handlers can access without communicating with the Car World UART.

Project setup and modifications

You will use a setup similar to Lab 2's. However, you need to make sure the ARM interrupt vector is placed at memory address **0**. To do so, in your project memory settings you need to choose the **Exceptions** preset:



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
.vectors	ARM_DDR3_SDRAM	0x00000000 - 0x0000003F
.text	ARM_DDR3_SDRAM	0x00000040 - 0x3FFFFFFF

JTAG UART documentation

A UART (Universally Asynchronous Receiver-Transmitter) core, to allow for communication between a terminal and the DE1-SoC Board.

Device	JTAG UART		
Input/Output	Both		
Address Base	0xFF201000		
Address Map	**Address**	**R/W**	**Description**
	base	R/W	Data Register 31:16 - Number of characters available to read (before this read - see Notes) 15 - read data is valid 7:0 - the data itself
	base+4	R/W	Control Register 31:16 - Spaces available for writing 10 - AC (has value of 1 if JTAG UART has been accessed by host, cleared by writing 1 to it) 9 - Write interrupt is pending 8 - Read interrupt is pending 1 - Enable write interrupts 0 - Enable read interrupts
Initialization	None		
Interrupts	**Triggered**	On data-received or able to send (see below)	
	IRQ Line	80	
	Enable	Set bit 1 and/or 0 in the Control Register for write and read interrupts respectively.	
	Acknowledge	Read (write) interrupts acknowledged by reading (writing) to the Data Register	
Software Setup	The Monitor Program creates a terminal window automatically, that can be used to send characters to the UART and receive the output from the JTAG UART.		
Reference	Altera JTAG Core Datasheet		

Notes

The Data Register is used for both sending and receiving data — the type of instruction executed (str or ldr) determines whether you send or receive. Every time you read from this register an 8-bit data byte is ejected from the receive FIFO, and every time you write to it you insert a byte to the send FIFO. Note that reading the data register word will obtain the

"Number of Bytes available" in bits [31:16], data valid [15], and will also read the data in bits [7:0] and eject another byte (if any) from the queue. The "Number of Bytes available" returns the utilization of the queue before the current read operation.

Use bit 15 to determine if a read is valid. The "Number of bytes available" must not be used for polling. It returns the state of the queue one cycle earlier than when the bytes are actually available. For example, reading the data register in the same cycle as a byte is received will result in "1 byte available", but the byte will not be valid, and no bytes will be dequeued from the FIFO.

If you try to send a character when the write FIFO is full, the character will not be sent, and will be lost.

Assembly Example: Sending the character 'A' through the JTAG

```
JTAG_WRITE_POLL:
  ldr r3, [r1, #4]     /* Load from the JTAG */
  lsrs r3, r3, #16     /* Check only the write available bits */
  beq JTAG_WRITE_POLL /* If this is 0, data cannot be sent yet */
  mov r2, #'A'         /* ASCII code for A */
  str r2, [r1]         /* Echo the data back to the JTAG */
```

Assembly Example: Polling the JTAG until valid data has been received

```
  ldr r1, =0xFF201000  /* r1 now contains the base address */
JTAG_READ_POLL:
  ldr r3, [r1]         /* Load from the JTAG */
  ands r2, r3, #0x8000 /* Mask other bits */
  beq JTAG_READ_POLL   /* If this is 0, data is not valid */
  and r2, r3, #0x00FF  /* Data read is now in r2 */
```

C Example: Sending "Hello World" through the UART

```
#define JTAG_UART_DATA ((volatile int*) 0xFF201000)
#define JTAG_UART_CONTROL ((volatile int*) (0xFF201000+4))

int main()
{
    unsigned char hwld[] = {'H','e','l','l','o','
','W','o','r','l','d','\0'};
    unsigned char *pOutput;

    pOutput = hwld;
    while(*pOutput) //strings in C are zero terminated
    {
         //if room in output buffer
         if((*JTAG_UART_CONTROL)&0xffff0000  )
         {
            //then write the next character
            *JTAG_UART_DATA = (*pOutput++);
         }
     }
}
```

Timer documentation

The timer is a peripheral that allows the user to measure real-time as a number of clock cycles. A user loads the timer with the number of clock cycles they'd like to wait, and then polls the "Timeout" bit or optionally enables an interrupt to indicate to the processor that the period has elapsed.

Device	Timer
Configuration	6 32-bit mapped registers (only lower 16 bits used)
Input/Output	Both
Address Base	Tiner 1: 0xFF202000 and Timer2: 0xxFF202020

Address Map	**Address**	**R/W**	**Description**
	base	R/W	Status Register bits: 1 - Run (1 if timer is running) 0 - Timeout (1 if timer has timed out - write 0 to this address to clear)
	base+4	R/W	Control Register bits: 3 - stop (write 1 to stop timer) 2 - start (write 1 to start timer) 1 - cont (if this bit is 1, timer will restart and continue when it times out, otherwise it will just reload the timeout period, but not start) 0 - interrupt enable for timeouts
	base+8	R/W	Periodl - lower 16 bits of Timeout period
	base+12	R/W	Periodh - upper 16 bits of Timeout period
	base+16	R/W	Counter Snapshot (lower 16 bits)
	base+20	R/W	Counter Snapshot (upper 16 bits)

Initialization	None (though a period must be written before running the timer, or it will immediately timeout)

Interrupts		
	Triggered	On timeout
	IRQ Line	Timer1 use IRQ line 72 ; Timer2 use IRQ Line 74
	Enable	Bit 0 of Control Register
	Acknowledge	Write 0 to status register to clear timeout bit (see address map above)

Hardware Setup	None
Reference	Full documentation from Altera

Notes

The timer counts downwards on a 100MHz clock, and runs in terms of clock cycles (a period of 100000000 will cause the timer to timeout in 1 second).

The full 32-bit period of the timer is given by the combination of "Periodh" and "Periodl". Both those registers are only 16-bits, even though they take up 32-bits of the address space.

When a timeout occurs, the Timeout bit in the Control Register will stay as 1 until the user writes 0 to the status register.

When a timeout occurs, the timer's counter is reset to the period, regardless of the "continue" bit. The continue bit determines whether the timer will then wait until 1 is written to the start bit again, or continue running immediately.

Reading the time remaining in the timer can not be done by reading the Periodh/l registers. Instead, the Counter Snapshot is used to copy the current time remaining, which can then be safely read by the user. By writing to either one of the snapshot registers (the written value is ignored), the current value of Periodh and Periodl, will be copied into the corresponding snapshot registers, which can then be read as in example 2 below.

WARNING: Do not try to read the Periodh/l registers directly.

Assembly Example: Setting the timer to run for 1000 clock cycles, and stop when it times out

```
LDR         R0, =0xFF202000 /* base address for the timer */
LDR         R1, =1000       /* clock cycles */
STR         R1, [R0, #0x8]  /* store low part (high will be ignored) */
LSR         R1, R1, #16     /* prepare high part */
STR         R1, [R0, #0xC]  /* set high part (in this case it'll be 0) */
MOV         R1, #0x4        /* start, no continuation & no interrupts */
STR         R1, [R0, #0x4]  /* start the timer */
```

C Example: Reading the Timer value

```
#define Timer 0xFF202000
#define TimerStatus ((volatile short*) (Timer))
#define TimerControl ((volatile short*) (Timer+4))
#define TimerTimeoutL ((volatile short*) (Timer+8))
#define TimerTimeoutH ((volatile short*) (Timer+12))
#define TimerSnapshotL ((volatile short*) (Timer+16))
#define TimerSnapshotH ((volatile short*) (Timer+20))

#define ADDR_LEDR ((volatile long *) 0xFF200000)

long numclks,numchigh,numclow;
int main()
{
   int i;

   // Configure the timeout period to maximum
   *(TimerTimeoutL)=0xffff;
   *(TimerTimeoutH)=0xffff;
   // Configure timer to start counting and to always continue
   *(TimerControl)=6;

   while (1)
   {
      *(TimerSnapshotL)=0; //write to timer to get snapshot
      numclow = *(TimerSnapshotL); //get low part
      numchigh = *(TimerSnapshotH); //get high part
      numclks = numclow | (numchigh << 16); //assemble full number
      *ADDR_LEDR = numclks;
   }
}
```

Preparation (3 marks)

1. How do you tell the JTAG UART to send read interrupts?

2. How do you tell the Timer to send timeout interrupts?

3. In the IRQ servicing routine of the example code you can find this instruction:

   ```
   SUBS    PC, LR, #4
   ```

   What does this line of code do, and why is it done?

4. Which registers must you backup before overwriting inside an interrupt handler?

5. If you want to call a function implemented in C from inside your interrupt handler, which registers must you back up?

6. Why are there two instructions of the form

   ```
   LDR     SP, =0x????????
   ```

   in the main program of the sample code?

7. Write a simple test program that uses the Timer to print a '#' character via the UART every second. You can use the same template given to you, with a main program that does nothing:

   ```
   LOOP:   B LOOP
   ```

   You do not need to initialize the UART (leave the JTAG initialization empty) or handle interrupts for devices that are not the Timer.

8. Write another test program to echo characters back from the UART using read interrupts. This is similar to the previous step, except you will initialize the UART for interrupts as opposed to the Timer, then in its handler read the character and immediately write it back.

9. Enhance your solution to Lab 2 to display speed and sensor state over the JTAG UART, as stated in the description section above. Use interrupts both to read from the terminal JTAG UART and to check the Timer for a timeout.

Submit your code and your answers (one submission per group) following the Submission Instructions Document by 6pm on Monday Oct 2.

In Lab (7 Marks)

Demonstrate your program. If your In Lab program doesn't work, you can demonstrate your simple test programs from the Preparation for **1.5** marks each.