

**Queen's University**  
**Department of Electrical and Computer Engineering**  
**ELEC 371 Microprocessor Interfacing and Embedded Systems**  
**Fall 2018**

**Lab 3:**  
**Interrupt and Input/Output Programming in C**

Copyright © 2018 by Dr. Naraig Manjikian, P.Eng.  
All rights reserved.

*Any direct or derivative use of this material  
beyond the course and term stated above  
requires explicit written consent from the author,  
with the exception of future private study and review  
by students registered in the course and term stated above.*

**Objectives**

This laboratory activity for *ELEC371* provides the opportunity for students to:

- pursue programming in C to increase the level of abstraction above assembly language,
- access input/output interfaces using constant-valued #define pointers,
- use vendor-provided and instructor-enhanced code that is invoked by interrupt events,
- handle initially one and then multiple timer interrupt sources,
- and employ character-output utility subroutines.


Preparation is specified in this document. The intent is for each individual student to make an honest effort to pursue the specified preparation without assistance from others to the extent that is possible. In this manner, a reasonable basis exists for useful discussion with others, and learning can thereby be made more effective.

## Preparation **BEFORE** Your Scheduled Laboratory Session

- Laboratory programming in the second half of the course will be pursued in the C language. For this purpose, an initial collection of header and source files – including files adapted by the instructor from vendor-provided material – is provided in a package *files\_for\_lab3.zip* for which a link is available on the course Webpage. The package contents are summarized below.
  - *nios2\_control.h*: macros for accessing special registers in the processor
  - *timer.h*: macros for accessing memory-mapped registers in the timer interface
  - *leds.h*: macro for accessing memory-mapped register for the LEDs interface
  - *exception\_handler.c*: instructor-modified vendor-supplied code for exception/interrupt service; saves/restores register values; calls a normal C function defined in the file *isr.c*
  - *isr.c*: despite the name of the file, contains a normal C function that is called from predefined code that is invoked when an exception or interrupt occurs.
  - *main.c*: performs initialization and then provides the basis for a main program loop
- Review lecture material in Chapter 4 on C programming with #define pointer definitions.
- Review the additional lecture material on the vendor-provided exception-handling function.
- Review the code and comments in the files contained in the ZIP package above.
- Consult sample code on pages 24 and 28 of the DE0 Computer reference documentation to obtain some programming guidance that may be relevant, but give higher priority to guidance provided by the instructor about programming in general and specific programming issues in C.
- In the files listed above, fill in the necessary code for initialization and response to timer interrupts.
- To test the program with the on-line simulator, you will need to use a computer with the full vendor software installation, e.g., in Beamish-Munro Hall. Although the on-line simulator directly accepts assembly-language source input, it does not directly accept C source code. A program in C must be compiled fully to produce an executable file in ELF format. The on-line simulator then reads the final executable code in the ELF file.
- Create a *LAB3* folder, and in it, place all six of the files described above with your added code.
- Start the Monitor Program and create a *lab3* project in the *LAB3* folder.
- Set up the project with the two QUEENS computer files as done in previous laboratory activity. In this manner, you will ultimately be able to use the same project to test the code on the hardware.
- **NEW**: For program type, select “C program” instead of assembly language.
- Add the *main.c* file first by itself so that the name of the final ELF file will be *main.elf*.
- Then, add the two remaining source files *exception\_handler.c* and *isr.c*. Based on past experience, a common mistake is that students inadvertently omit one or both of these two files. Always ensure that all of the *.c* files for a program are added to the list.
- *Do not add the .h files*. These files are not compiled; they are #included in the *.c* files.
- **NEW**: For the memory settings, you must use an offset of 200 (in hex) for both *.text* and *.data* sections. These setting are necessary because the *.reset* and *.exceptions* sections must be positioned at the beginning of the memory instead of the *.text* section.
- Because you will not be using the hardware board for your preparation, the “host connection” that would normally show *USB-blaster* will be blank. Simply continue past that point.
- When prompted about downloading the system to the board, answer “no” because you are not using the hardware for your preparation.
- After you have completed the setup of the project, just *compile* the code (not compile-and-load because you have no board in which to load the code).
- If there are any syntax errors, use the error messages to identify the offending files and line numbers, correct the errors, and recompile (without loading). Repeat until there are no errors.

- Open a Web browser, navigate to the on-line simulator of the original DE0 computer system, and choose *File* → *ELF Executable Load*. In the dialog box for file selection, navigate to your *LAB3* folder and select *main.elf* (assuming that you added *main.c* as the first file in setup of the project).
  - Because the compiler-generated code uses normal load/store instructions for accessing memory-mapped input/output interfaces, there is one more step that is necessary in the simulator. In the “Settings” window at the bottom-left corner, scroll down to find “Memory: Suspicious use of cache bypass.” Uncheck that option. Doing so will tell the simulator to permit normal load/store instruction to access input/output interfaces. Otherwise, the simulator will stop execution. (There is a way to force the compiler to generate ldwio/stwio instructions, but we will avoid unnecessary complications by just not having a data cache that would require such instructions for access to input/output interfaces.)
  - Finally, click on the “Continue” button to initiate execution, and verify that the LED blinks on and off as expected. If the program behavior is incorrect, review your code, identify the problem(s), revise the code as necessary, recompile (without loading), load the updated ELF file into the simulator, and test again. Repeat until correct behavior is obtained.
- 
- Once the basic program above is functioning properly, introduce C code for *PrintChar(ch)* and *PrintString(s)* in a new file *chario.c* with its associated *chario.h*, as developed in Tutorial 5.
  - In the Monitor Program, under program settings, add *chario.c* to the list of files (but not *chario.h*).
  - Introduce an *#include* statement for *chario.h* in *main.c*. In the *main()* function, after initialization, use *PrintString()* with the character string “ELEC371 Lab 3\n” as its input argument. Double quotes are necessary in C to define the content of the string. The compiler automatically adds a zero byte at the end to terminate the string. The newline \n character included within the double quotes causes output to move to the next line for any subsequent characters that are printed.
  - Recompile the revised program (without loading) in the Monitor Program, load the ELF file into the simulator, and test its behavior to confirm that output characters are printed and that the LED blinking with interrupts continues as expected.
- 
- As a final extension of the program to prepare before your lab session, introduce a software flag for interaction between the interrupt code and the main program.
  - Define a global variable `int flag;` in *main.c* (after *#include* and *#define* statements, but before the functions).
  - To make the code in *isr.c* aware of the existence of this global variable, introduce the declaration `extern int flag;` before function code in *isr.c*.
  - Set the flag variable to 1 in *interrupt\_handler()* when a timer interrupt occurs (as in Tutorial 6).
  - Use an *if* statement in the body of the infinite loop in *main()* to check if the flag is *non-zero*. If so, clear the flag to zero and use *PrintChar()* to display the asterisk character ‘\*’ as output.
  - Again, recompile the revised program (without loading) in the Monitor Program, load the ELF file into the simulator, and test its behavior to confirm that the initial output string appears, the LED blinks as expected, and characters are printed one at a time on each timer interrupt.

## In-Lab Part 1: Testing of Prepared Program on Hardware

- Plug the DE0-CV board into the computer using the USB cable, and turn the power on.
- Assuming that you have properly prepared a working program (as tested in simulation) in a *LAB3* folder on your network storage, start the Monitor Program and open your existing *lab3* project.
- In the Monitor Program, select *Actions* → *Download System* if necessary to configure the FPGA.
- The program should have already been compiled for testing in simulation, hence it should only be necessary to select *Actions* → *Load*. Of course, it is possible to select *Actions* → *Compile & Load*.
- Press  or select *Actions* → *Continue* to initiate execution of the program.
- Verify proper functionality of printed output in the terminal window and LED blinking.

## In-Lab Part 2: Using an Additional Timer Interface in the QUEENS Computer

- Make backup copies of *main.c* and *isr.c* as snapshots of your working program from Part 1 above.
- From the course Webpage, open the document that describes the QUEENS Computer and find the address map of the system. In the address map, identify the original timer interface that was inherited from the vendor DE0 Computer with its starting address and bit position in the processor *ienable/ipending* registers. Then, immediately below the original timer, identify *timer\_0*, the first of the new timer interfaces introduced by the instructor for the QUEENS Computer with its starting address and its different bit position in the processor *ienable/ipending* registers.
- Open the file *timer.h* and introduce additional macros for registers in new interface *timer\_0*. It is suggested that you use macro names such as *T0\_STATUS* and *T0\_CONTROL* that are similar yet sufficiently distinct from the names used for the original timer interface. *Make certain to use the correct addresses for the memory-mapped registers of the new timer interface.*
- Open the file *isr.c* and introduce an additional *if* statement separate from and directly below the existing *if* statement for the original timer. In the condition for the new *if* statement, check the appropriate bit of the value read from the processor *ipending* register to determine if an interrupt request has been asserted from the new timer interface.
- A useful technique in C for generating desired patterns for bit-masking operations is to use left-shift operations. Note that bit position  $k$  (starting from 0 for the least-significant bit) corresponds to the position for the numerical value of  $2^k$ . Shifting to the left corresponds to multiplication, and shifting by  $k$  bit positions to the left corresponds to multiplication by  $2^k$ . Therefore, to place a single bit 1 in bit position  $k$ , the equivalent mathematical expression is  $2^0 \cdot 2^k$ , i.e., take a 1 in bit position 0 and shift it to the left by  $k$  bits. In C syntax, this would be expressed as  $1 \ll k$ , but in this case, the value of  $k$  for the bit position in the *ipending* register is a constant for the *timer\_0* interface, as given in the document for the QUEENS Computer. Because both the value being shifted and the amount of the shift are constants, providing this expression in C source code will allow the compiler to pre-compute the final shifted value, which will be a constant in the compiled output.
- Complete the body the *if* statement for interrupts from new interface *timer\_0* to simply toggle bit 1 of the LEDs using an XOR operation (the original timer should continue to toggle bit 0).
- *Do not forget to clear the interrupt for the new timer interface.*
- Finally, edit the file *main.c* and modify the initialization performed in the *main()* function to have the new interface *timer\_0* generate interrupts every 0.25 second (twice as often as the original timer).
- *Do not forget to set the appropriate bit for the new timer interface in the processor ienable register.* Note that the technique described above with the left-shift operation is applicable in this case. The additional factor is that now two bits must be set in *ienable*. A straightforward way to do so is to use the logical-OR operator in C (*|*) to combine together two bit patterns for use in a single write.

- Make certain that you have saved any files that were modified in the preceding steps.
- In the Monitor Program, compile and load the program. (If there are syntax errors, certainly correct those until the program compiles successfully.)
- Execute the modified program and verify that all of the previous aspects function correctly (LED 0 toggles every 0.5 s, initial string printed followed by a character for every original timer interrupt), but now also that LED 1 toggles every 0.25 s to reflect the inclusion of interrupt-based code that uses new timer interface *timer\_0*.

**DEMONSTRATE THE WORKING PART 2 PROGRAM TO OBTAIN CREDIT.**

### **In-Lab Part 3: Extension of Multiple-Interrupt Program**

- Make additional backup copies of *main.c* and *isr.c* as snapshots of your working program from Part 2 above.
- Use the specifications given to you in your lab session to apply the required extension of your working program from Part 2 above.
- Compile and load the program, and execute it to verify that it functions as intended.

**DEMONSTRATE THE WORKING PART 3 PROGRAM TO OBTAIN CREDIT.**