

Queen's University
Department of Electrical and Computer Engineering
ELEC 371 Microprocessor Interfacing and Embedded Systems
Fall 2019

Lab 3:
Interrupt and Input/Output Programming in C

Copyright © 2019 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 registers in 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 multiple 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. The working program must cause an LED to blink on/off at the specified rate.*
- 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 with the complete code-generation tool chain to produce an executable file in ELF format. The on-line simulator can 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.
- Select the DE0 **Media** Computer to have hardware support for multiply/divide instructions.
- ****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 be using simulation 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, select the 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 write C code with special macros and built-in functions 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 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 in the hardware.
- Verify proper functionality of printed output in the terminal window and LED blinking.

In-Lab Part 2: Handling Multiple Interrupt Sources in C

- Make backup copies of *main.c* and *isr.c* as snapshots of your working program from Part 1 above.
- Building on the experience gained in Lab 2, the aim for this part is to extend the single-interrupt program from the preparation and Part 1 to a multiple-interrupt program (but still relatively simple).
- Create a **new** header file *button.h* and introduce in this file three *#define* statements for the data register, the mask register, and the edge register of the pushbutton interface of the DE0 computer system. Consult the vendor documentation using the link provided on the course Webpage.
- Introduce an *#include* statement for *button.h* in *main.c* and also in *isr.c*.
- Modify the initialization function in *main.c* to prepare the pushbutton interface for interrupts using Button 1, as done in Lab 2, but this time in C code, rather than in assembly language. Do not forget to set the *ienable* bits properly.
- Again, similar to Lab 2, modify *isr.c* to respond to pushbutton interrupts (from Button 1). Do not forget to clear the interrupt. For the action to perform on each Button 1 interrupt, simply toggle the second-lowest bit of the LEDs using an XOR operation. (The lowest bit of the LEDs is controlled by the existing code that responds to timer interrupt. The occurrence of pushbutton interrupts should not affect the response for timer interrupts. XOR operations permit the LEDs to be toggled independently.)
- 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, correct them 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 each time Button 1 is pressed and released.

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.