Lab 3: Basic I/O, Timers and Interrupts

ECSE 324 - Computer Organization

Fall 2017

Introduction

This lab introduces the basic I/O capabilities of the DE1-SoC computer - the slider switches, push-buttons, LEDs and 7-Segment displays. After writing assembly drivers that interface with the I/O components, timers and interrupts are used to demonstrate polling and interrupt based applications written in C.

1 Creating the Project in the Altera Monitor Program

IMPORTANT: The project is structured as outlined below to introduce concepts that are used in writing well organized code. Furthermore, drivers for configuring the Generic Interrupt Controller (GIC) will be provided in the latter part of this lab, and the driver code relies on this project structure. The code will not compile if the project is not organized as described!

First, create a new folder named *GXX_Lab3*, where GXX is the corresponding group number. Within this folder, create a new folder named *drivers*. Finally, within the *drivers* folder, create three folders: *asm*, *src* and *inc*. The final folder structure is shown in Figure 1.

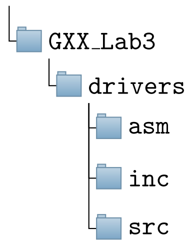


Figure 1: The project folder structure

Create a new file **main.c** and save it in the *GXX_Lab3* folder.

Next, open the Altera Monitor Program and create a new project. Select the created folder *GXX_Lab3* as the project directory, name the project *GXX_Lab3*, set the architecture to *ARM Cortex-A9* and click 'Next'.

When asked to select a system, select *De1-SoC Computer* from the drop-down menu and click 'Next'.

Set the program type as *C Program* and click 'Next'.

In the next menu, add **main.c** to the source files.

In the System Parameters menu, ensure that the board is detected in the 'Host connection' dialogue box and click 'Next'.

Finally, in the memory settings menu, change the Linker Section Presents from 'Basic' to 'Exceptions' and click 'Finish'.

2 Basic I/O

For this part, it is necessary to refer to sections 2.5.6 - 2.5.10 (pp. 8 - 10) and 3.4 (pp. 20 - 21) in the De1-SoC Computer Manual.

Brief overview

The hardware setup of the I/O components is fairly simple to understand. The ARM cores have designated addresses in memory that are connected to hardware circuits on the FPGA, and these hardware circuits in turn interface with the physical I/O components.

In the case of most of the basic I/O, the FPGA hardware can be as simple as a direct mapping from the I/O terminals to the memory address designated to it. For instance, the state of the slider switches is available to the FPGA on bus of 10 wires which carry either a logical '0' or '1'. This bus can be directly passed as 'write-data' to the memory address reserved for the slider switches (0xFF200040 in this case).

It is useful to have slightly more sophisticated FPGA hardware. For instance, in the case of the push-buttons, in addition to knowing the state of the button it is also helpful to know whether a falling edge is detected, signalling a keypress. This can be achieved by a simple edge detection circuit in the FPGA.

The FPGA hardware to interface with the I/O is part of the De1-SoC computer, and is loaded when the .sof file is flashed onto the board. This section will deal with writing assembly code to control the I/O interact by reading from and writing to memory.

Getting started: Drivers for slider switches and LEDs

- **Slider switches:**

  Create a new assembly file called **slider_switches.s** in the *GXX_Lab3/drivers/asm* directory. Create a new subroutine labelled *read_slider_switches_ASM*, which will read the value at the memory location designated for the slider switches data into the **R0** register, and then branch to the link register. Make the subroutine visible to other files in the project by using the **.global** assembler directive. *Remember to use the ARM function calling convention, and save the context if needed!*.

  Next, create a new header file called **slider_switches.h** in the *GXX_Lab3/drivers/inc* directory. The header file will provide the C function declaration for the slider switches assembly driver. Declare the function as *extern int read_slider_switches_ASM()*, and make use of preprocessor directives to avoid recursive inclusion of the header file.

  To help get started, code for the slider switches driver has been provided in Figure 2. Use this as a template for writing future driver code.

- **LEDs:**

  Create a new assembly file called **LEDs.s** in the *GXX_Lab3/drivers/asm* directory. Create two subroutines - *read_LEDs_ASM* and *write_LEDs_ASM*. Again, export both subroutines using the **.global** assembler directive

  Similar to the slider switches driver, the *read_LEDs_ASM* subroutine will load the value at the LEDs memory location into **R0** and then branch to **LR**. The *write_LEDs_ASM* subroutine will store the value in **R0** at the LEDs memory location, and then branch to **LR**.

  Create a new header file called **LEDs.h** in the *GXX_Lab3/drivers/inc* directory. Provide function declarations for both the subroutines. **The function declaration will not be the exact same as in the slider switches; one of these functions will have to accept an argument!**

```
  .text
  .equ  SW_BASE, 0xFF200040
  .global read_slider_switches_ASM

read_slider_switches_ASM:
  LDR R1, =SW_BASE
  LDR R0, [R1]
  BX LR

  .end
```

(a) Assembly file

```
#ifndef __SLIDER_SWITCHES
#define __SLIDER_SWITCHES

  extern int read_slider_switches_ASM();

#endif
```

(b) Header file

Figure 2: Code for the slider switches driver

- **Putting it together:**

  Fill in the *main.c* file in the *GXX_Lab3* directory. The main function will include the header files for both the drivers, and will send the switches state to the LEDs in an infinite while loop. The code for this file is shown in Figure 3.

```
#include <stdio.h>

#include "./drivers/inc/LEDs.h"
#include "./drivers/inc/slider_switches.h"

int main() {

  while(1) {
    write_LEDs_ASM(read_slider_switches_ASM());
  }

  return 0;
}
```

Figure 3: Code for the main.c file

  Next, open the project settings and add all the driver files to the project. Compile and load the project onto the De1-SoC computer, and run the code. The LED lights should now turn on and off when the corresponding slider switch is toggled.

Slightly more advanced: Drivers for HEX displays and push-buttons

Now that the basic structure of the drivers has been introduced, custom data types in C will be used to write drivers that are more readable and easier to implement. In particular, the following two drivers will focus on using **enumerations** in C.

- **HEX displays:**

  As in the previous parts, create two files **HEX_displays.s** and **HEX_displays.h** and place them in the correct folders.

  The code for the header file is provided in Figure 4. Notice the new datatype *HEX_t* defined in the form of an enumeration, where each display is given a unique value based on a one-hot encoding scheme. This will be useful when writing to multiple displays in the same function call.

  Write the assembly code to implement the three functions listed in the header file. The *HEX_t* argument can be treated as an integer in the assembly code. **The subroutine should check the argument for all the displays HEX0-HEX5, and write to whichever ones have been asserted.** A loop may be useful here!

  HEX_clear_ASM will turn off all the segments of **all** the HEX displays passed in the argument. Similarly, *HEX_flood_ASM* will turn on all the segments. The final function *HEX_write_ASM*

```c
#ifndef __HEX_DISPLAYS
#define __HEX_DISPLAYS

  typedef enum {
    HEX0 = 0x00000001,
    HEX1 = 0x00000002,
    HEX2 = 0x00000004,
    HEX3 = 0x00000008,
    HEX4 = 0x00000010,
    HEX5 = 0x00000020
  } HEX_t;

  extern void HEX_clear_ASM(HEX_t hex);
  extern void HEX_flood_ASM(HEX_t hex);
  extern void HEX_write_ASM(HEX_t hex, char val);

#endif
```

Figure 4: Code for the HEX_displays.h file

takes a second argument **val**, which is a number between 0-15. Based on this number, the subroutine will display the corresponding hexadecimal digit (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) on the display(s).

A sample program is shown in Figure 5 to demonstrate how multiple displays can be controlled in the same function call. Since the value for each display is based on a one-hot encoding scheme, the logical OR of all the displays will assert the bits for all the displays.

```c
#include <stdio.h>

#include "./drivers/inc/HEX_displays.h"

int main() {
    HEX_flood_ASM(HEX0 | HEX1 | HEX2 | HEX3 | HEX4 | HEX5);

    return 0;
}
```

Figure 5: Sample program that uses the HEX displays driver

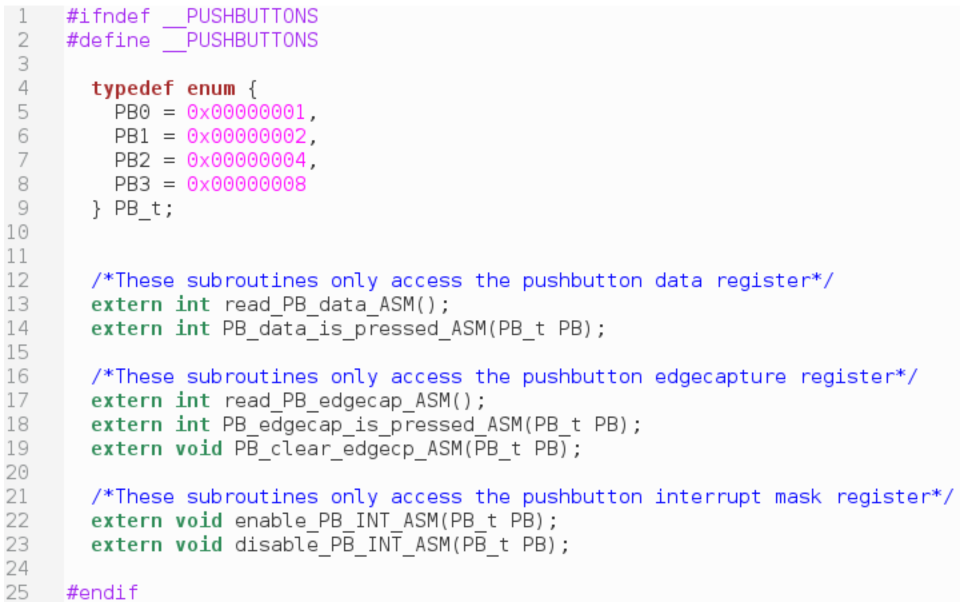
- **Pushbuttons:**

  Create two files **pushbuttons.s** and **pushbuttons.h** and place them in the correct folders.

  Write the assembly code to implement the functionality described in the header file, as shown in Figure 6

- **Putting it together:**

  Modify the *main.c* file to create an application that uses all of the drivers created so far. As before, the state of the slider switches will be mapped directly to the LEDs. Additionally, the state of the last four slider switches SW3-SW0 will be used to set the value of a number from 0-15. This number will be displayed on a HEX display when the corresponding pushbutton is pressed. For example. pressing KEY0 will result in the number being displayed on HEX0. Since there are no pushbuttons to correspond to HEX4 and HEX5, switch on all the segments of these two displays. Finally, asserting slider switch SW9 should clear all the HEX displays.



```c
#ifndef __PUSHBUTTONS
#define __PUSHBUTTONS

  typedef enum {
    PB0 = 0x00000001,
    PB1 = 0x00000002,
    PB2 = 0x00000004,
    PB3 = 0x00000008
  } PB_t;


  /*These subroutines only access the pushbutton data register*/
  extern int read_PB_data_ASM();
  extern int PB_data_is_pressed_ASM(PB_t PB);

  /*These subroutines only access the pushbutton edgecapture register*/
  extern int read_PB_edgecap_ASM();
  extern int PB_edgecap_is_pressed_ASM(PB_t PB);
  extern void PB_clear_edgecp_ASM(PB_t PB);

  /*These subroutines only access the pushbutton interrupt mask register*/
  extern void enable_PB_INT_ASM(PB_t PB);
  extern void disable_PB_INT_ASM(PB_t PB);

#endif
```

Figure 6: Code for the pushbuttons.h file

3 Timers

For this part, it is necessary to refer to sections 2.4.2 (p. 4) and 3.2 (p. 20) in the De1-SoC Computer Manual.

Brief introduction

Timers are simply hardware counters that are used to measure time and/or synchronize events. They run on a known clock frequency that is programmable in some cases (by using a phase-locked loop). Timers are usually (but not always) down counters, and by programming the start value, the time-out event (when the counter reaches zero) occurs at fixed time intervals.

HPS timer drivers

Create two files **HPS_TIM.s** and **HPS_TIM.h** and place them in the correct folders. The code for the header file is shown in Figure 7.

```c
#ifndef __HPS_TIM
#define __HPS_TIM

  typedef enum {
    TIM0 = 0x00000001,
    TIM1 = 0x00000002,
    TIM2 = 0x00000004,
    TIM3 = 0x00000008
  } HPS_TIM_t;

  typedef struct {
    HPS_TIM_t tim;
    int timeout; // in usec
    int LD_en;
    int INT_en;
    int enable;
  } HPS_TIM_config_t;

  /*  Note: The argument is a struct pointer!         */
  extern void HPS_TIM_config_ASM(HPS_TIM_config_t *param);

  /*  Reads the value of the S-bit (offset = 0x10)    */
  /*  The nature of the return value will depend on   */
  /*  whether this function is able to read the       */
  /*  S-bit value of multiple timers in the same call */
  extern int HPS_TIM_read_INT_ASM(HPS_TIM_t tim);

  /*  Resets the S-bit and the F-bit                  */
  /*  This function should support multiple timers in */
  /*  the argument                                    */
  extern void HPS_TIM_clear_INT_ASM(HPS_TIM_t tim);

#endif
```

Figure 7: Code for the HPS_TIM.h file

This driver uses a new concept in C - **structures**. A structure is a composite datatype that allows the grouping of several variables to be accessed by a single pointer. They are similar to arrays, except that the individual elements of a structure can be of different datatypes! Writing this driver will help demonstrate how structures can be useful by modifying multiple parameters easily.

Notice how the first subroutine *HPS_TIM_config_ASM* takes a struct pointer as an argument. The reason for this is that if a struct is passed directly to a function, the compiler unpacks the struct elements at compile time and passes them as individual arguments to the function. Since in most cases the number of arguments will be greater than the number of argument registers, the compiler will place the extra arguments on the stack. This is perfectly fine if all the code is handled by the compiler, but since this lab requires handwritten assembly drivers, it causes the programmer a lot of extra overhead when retrieving the arguments in the assembly subroutine. By passing a struct pointer, the individual elements can be easily accessed at the corresponding offset from the base address passed in the pointer. For instance, the *timeout* element can be accessed in the assembly subroutine via a load instruction from the address in **R0** offset by 0x4.

Implement assembly subroutines for the three functions shown in the header file. The second subroutine *HPR_TIM_read_INT_ASM* need not support multiple timer instances passed in the argument, but if it does then the return value should be set appropriately to reflect the S-bit value of all the timers present in the argument. The other two subroutines should be able to handle multiple timers.

Notice how the *timeout* struct element is given in microseconds. This hides the hardware specific details of the timer from the C programmer. Since all of the HPS timers do not run on the same clock frequency, the subroutine must calculate the correct load value for the corresponding timer in order to achieve the desired timeout value.

IMPORTANT: In the *HPS_TIM_config_ASM* subroutine, each timer should first be disabled before writing any of the other configuration parameters. The value in the enable parameter can then be written last.

IMPORTANT: The De1-SoC computer manual has omitted to mention that the S-bit in the interrupt status register will be asserted only if the I-bit in the control register is set to 0 (in order to unmask the timeout event)!

A sample program that uses the HPS timer driver is shown in Figure 8. Notice how all four HPS timers are configured to have a 1 second timeout in the same function call. If the assembly driver functions correctly, the program will count from 0-15 on all four HEX displays at the same rate of 1 second. It is important to remember that the configuration values in the struct are implemented at a level of abstraction above the hardware, with the aim of providing a better hardware interface to the C programmer. How these values are then used in the assembly driver should be governed by the hardware documentation (De1-SoC computer manual).

Creating an application: Stopwatch!

Create a simple stopwatch using the HPS timers, pushbuttons, and HEX displays. The stopwatch should be able to count in increments of 10 milliseconds. Use a single HPS timer to count time. Display milliseconds on HEX1-0, seconds on HEX3-2, and minutes on HEX5-4.

PB0, PB1, and PB2 will be used to start, stop and reset the stopwatch respectively. Use another HPS timer set at a faster timeout value (5 milliseconds or less) to poll the pushbutton edgecapture register.

```c
#include <stdio.h>
#include "./drivers/inc/HEX_displays.h"
#include "./drivers/inc/HPS_TIM.h"

int main() {
  int count0 = 0, count1 = 0, count2 = 0, count3 = 0;

  HPS_TIM_config_t hps_tim;

  hps_tim.tim = TIM0|TIM1|TIM2|TIM3;
  hps_tim.timeout = 1000000;
  hps_tim.LD_en = 1;
  hps_tim.INT_en = 1;
  hps_tim.enable = 1;

  HPS_TIM_config_ASM(&hps_tim);

  while(1) {
    if(HPS_TIM_read_INT_ASM(TIM0)) {
      HPS_TIM_clear_INT_ASM(TIM0);
      if(++count0 == 16)
        count0 = 0;
      HEX_write_ASM(HEX0, count0);
    }

    if(HPS_TIM_read_INT_ASM(TIM1)) {
      HPS_TIM_clear_INT_ASM(TIM1);
      if(++count1 == 16)
        count1 = 0;
      HEX_write_ASM(HEX1, count1);
    }

    if(HPS_TIM_read_INT_ASM(TIM2)) {
      HPS_TIM_clear_INT_ASM(TIM2);
      if(++count2 == 16)
        count2 = 0;
      HEX_write_ASM(HEX2, count2);
    }

    if(HPS_TIM_read_INT_ASM(TIM3)) {
      HPS_TIM_clear_INT_ASM(TIM3);
      if(++count3 == 16)
        count3 = 0;
      HEX_write_ASM(HEX3, count3);
    }

  }
  return 0;
}
```

Figure 8: Sample program that uses the HPS timer driver

4 Interrupts

For this part, it is necessary to refer to section 3 (pp. 19-32) in the De1-SoC Computer Manual. Additional information about the interrupt drivers that are provided can be found in 'Using the ARM Generic Interrupt Controller' which is available on *myCourses*.

Brief introduction

Interrupts are hardware or software signals that are sent to the processor to indicate that an event has occurred that needs immediate attention. When the processor receives an interrupt, it pauses the current code execution, handles the interrupt by executing code defined in an Interrupt Service Routine (ISR), and then resumes normal execution.

Apart from ensuring that high priority events are given immediate attention, interrupts also help the processor to utilize resources more efficiently. Consider the polling application from the previous section, where the processor periodically checked the pushbuttons for a keypress event. Asynchronous events such as this, if assigned an interrupt, can free the processors time and use it only when required.

Using the interrupt drivers

Download the following files from *myCourses*:

- **int_setup.c**
- **int_setup.h**
- **ISRs.s**
- **ISRs.h**
- **address_map_arm.h**

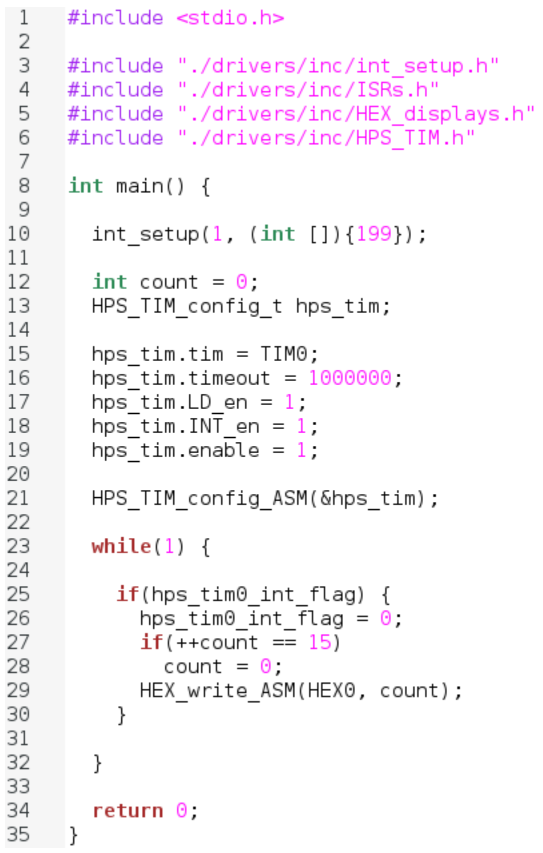
Within the *GXX_Lab3/drivers/* directory, place C files in the *src*, header files in the *inc*, and assembly files in the *asm* directories. Only the **ISRs.s** and **ISRs.h** files will need to be modified in applications. **Do not modify the other files**

Before attempting this section, get familiarized with the relevant documentation sections provided in the introduction. To demonstrate how to use the drivers, a simple interrupt based application using HPS TIM0 is shown.

Note: Ensure that in the memory settings menu in the project settings, the Linker Section Presets has been changed from 'Basic' to 'Exceptions'!

To begin, the code for the **main.c** file is shown in Figure 9. The *int_setup()* function is the only thing needed to configure the interrupt controller and enable the desired interrupt IDs. It takes two arguments: an integer whose value denotes the number of interrupt IDs to enable, and an integer array containing these IDs. In this example, the only interrupt ID enabled is 199, corresponding to HPS TIM0.

After enabling interrupts for the desired IDs, the hardware devices themselves have to be programmed to generate interrupts. This is done in the code above via the HPS timer driver. Instructions for enabling interrupts from the different hardware devices can be found in the documentation.



```c
#include <stdio.h>

#include "./drivers/inc/int_setup.h"
#include "./drivers/inc/ISRs.h"
#include "./drivers/inc/HEX_displays.h"
#include "./drivers/inc/HPS_TIM.h"

int main() {

  int_setup(1, (int [])(199});

  int count = 0;
  HPS_TIM_config_t hps_tim;

  hps_tim.tim = TIM0;
  hps_tim.timeout = 1000000;
  hps_tim.LD_en = 1;
  hps_tim.INT_en = 1;
  hps_tim.enable = 1;

  HPS_TIM_config_ASM(&hps_tim);

  while(1) {

    if(hps_tim0_int_flag) {
      hps_tim0_int_flag = 0;
      if(++count == 15)
        count = 0;
      HEX_write_ASM(HEX0, count);
    }

  }

  return 0;
}
```

Figure 9: Interrupts example: The main.c file

Now that HPS TIM0 is able to send interrupts, ISR code is needed to handle the interrupt events. Notice how in the while loop of the main program, the value of *hps_tim0_int_flag* is checked to see if an interrupt has occurred. The ISR code is responsible for writing to this flag, and also for clearing the interrupt status in HPS TIM0.

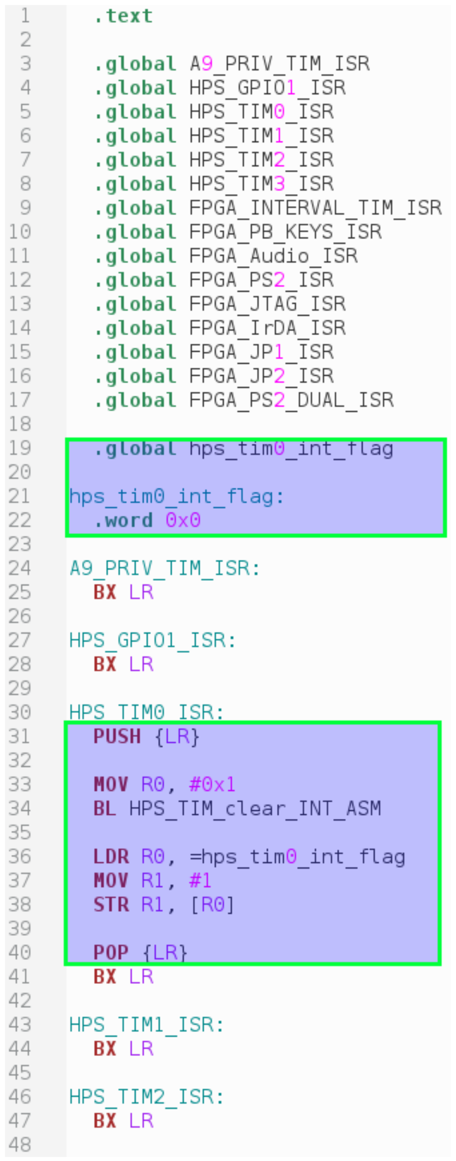
When interrupts from a device are enabled and an interrupt is received, the processor halts code execution and branches to the appropriate subroutine in the **ISRs.s** file. This is where the ISR code should be written. Figure 10 shows the ISR code for HPS TIM0. In the ISR, the interrupt status of the timer is cleared, and the interrupt flag is asserted.

Finally, in order for the main program to use the interrupt flag, it is declared in the **ISRs.h** file as shown in Figure 11.

IMPORTANT: When ISR code is being executed, the processor has halted normal execution. Lengthy ISR code will cause the application to freeze. ISR code should be as lightweight as possible!

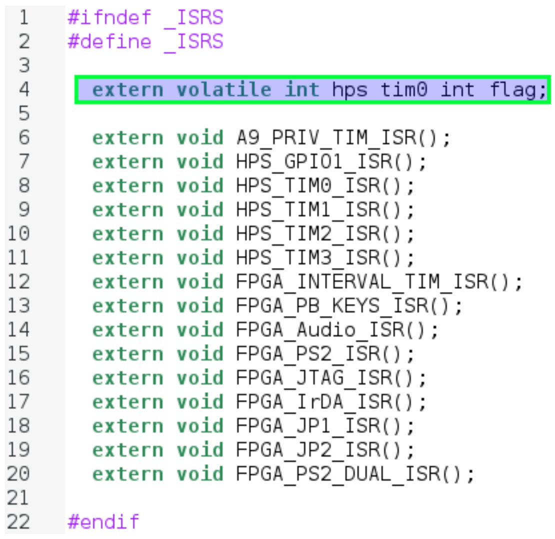
Interrupt based stopwatch!

Modify the stopwatch application from the previous section to use interrupts. In particular, enable interrupts for the HPR timer used to count time for the stopwatch. Also enable interrupts for the pushbuttons, and determine which key was pressed when a pushbutton interrupt is received. There is no need for the second HPS timer that was used to poll the pushbuttons in the previous section.



```
  .text

  .global A9_PRIV_TIM_ISR
  .global HPS_GPIO1_ISR
  .global HPS_TIM0_ISR
  .global HPS_TIM1_ISR
  .global HPS_TIM2_ISR
  .global HPS_TIM3_ISR
  .global FPGA_INTERVAL_TIM_ISR
  .global FPGA_PB_KEYS_ISR
  .global FPGA_Audio_ISR
  .global FPGA_PS2_ISR
  .global FPGA_JTAG_ISR
  .global FPGA_IrDA_ISR
  .global FPGA_JP1_ISR
  .global FPGA_JP2_ISR
  .global FPGA_PS2_DUAL_ISR

  .global hps_tim0_int_flag

hps_tim0_int_flag:
  .word 0x0

A9_PRIV_TIM_ISR:
  BX LR

HPS_GPIO1_ISR:
  BX LR

HPS_TIM0_ISR:
  PUSH {LR}

  MOV R0, #0x1
  BL HPS_TIM_clear_INT_ASM

  LDR R0, =hps_tim0_int_flag
  MOV R1, #1
  STR R1, [R0]

  POP {LR}
  BX LR

HPS_TIM1_ISR:
  BX LR

HPS_TIM2_ISR:
  BX LR

```

Figure 10: Interrupts example: The ISR assembly code



```c
#ifndef _ISRS
#define _ISRS

  extern volatile int hps_tim0_int_flag;

  extern void A9_PRIV_TIM_ISR();
  extern void HPS_GPIO1_ISR();
  extern void HPS_TIM0_ISR();
  extern void HPS_TIM1_ISR();
  extern void HPS_TIM2_ISR();
  extern void HPS_TIM3_ISR();
  extern void FPGA_INTERVAL_TIM_ISR();
  extern void FPGA_PB_KEYS_ISR();
  extern void FPGA_Audio_ISR();
  extern void FPGA_PS2_ISR();
  extern void FPGA_JTAG_ISR();
  extern void FPGA_IrDA_ISR();
  extern void FPGA_JP1_ISR();
  extern void FPGA_JP2_ISR();
  extern void FPGA_PS2_DUAL_ISR();

#endif
```

Figure 11: Interrupts example: Flag declaration in ISRs.h

5 Grading

The TA will ask to see the following deliverables during the demo (the corresponding portion of your grade for each is indicated in brackets):

- Slider switches and LEDs program (10%)
- Entire basic I/O program (15%)
- Polling based stopwatch (30%)
- Interrupt based stopwatch (25%)

Full marks are awarded for a deliverable only if the program functions correctly and the TA's questions are answered satisfactorily.

A portion of the grade is reserved for answering questions about the code, which is awarded individually to group members. All members of your group should be able to answer any questions the TA has about any part of the deliverables, whether or not you wrote the particular part of the code the TA asks about. Full marks are awarded for a deliverable only if the program functions correctly and the TA's questions are answered satisfactorily.

Finally, the remaining 20% of the grade for this Lab will go towards a report. Write up a short (3-4) page report that gives a brief description of each part completed, the approach taken, and the challenges faced, if any. Please don't include the entire code in the body of the report. Save the space for elaborating on possible improvements you made or could have made to the program.

Your final submission should be a **single compressed folder** that contains your report and all the code files, correctly organized (.c, .h and .s).

This Lab will run for **three weeks**, from October 16th to November 3rd. There will be no lab sessions in the second week, October 23rd to October 27th because of the midterm. You should demo your code during the two active weeks, **within your assigned lab period.** The report for Lab 3 is due by 11:59 pm, November 10.