

# Laboratory Exercise 3

## Memory-Mapped I/O, Polling and Timers

The purpose of this exercise is to explore the use of devices that provide input and output capabilities for a processor. You will implement your solutions for this exercise on the DE1-SoC board that is available as part of your assigned workstation, in your practical session on campus. Before working on this exercise, make sure to perform the steps given in the document called *Preface to Laboratory Exercise 3*, which is posted on Quercus along with this exercise. The purpose of the *Preface* is to introduce the *Intel FPGA Monitor Program* application software, which you have to use when implementing ARM programs in a DE1-SoC board.

There are two basic techniques for synchronizing with I/O devices: program-controlled *polling* and *interrupt-driven* approaches. We will use the polling approach in this exercise, writing programs in the ARM\* assembly language. Your programs will be executed on the ARM processor in the DE1-SoC Computer system—you will be required to demonstrate to a TA your working code by executing it on a DE1-SoC board using the Monitor Program, but you may also wish to develop/debug your code at home by using the CPUlator tool. Parallel port interfaces, as well as a *timer* hardware module, will be used as examples of I/O hardware.

In general, a parallel port provides for data transfer in either the input or output direction. The transfer of data is done in parallel and may involve from 1 to 32 bits. The number of bits,  $n$ , and the type of transfer depend on the specific parallel port being used. The parallel port interface can contain the four registers shown in Figure 1.

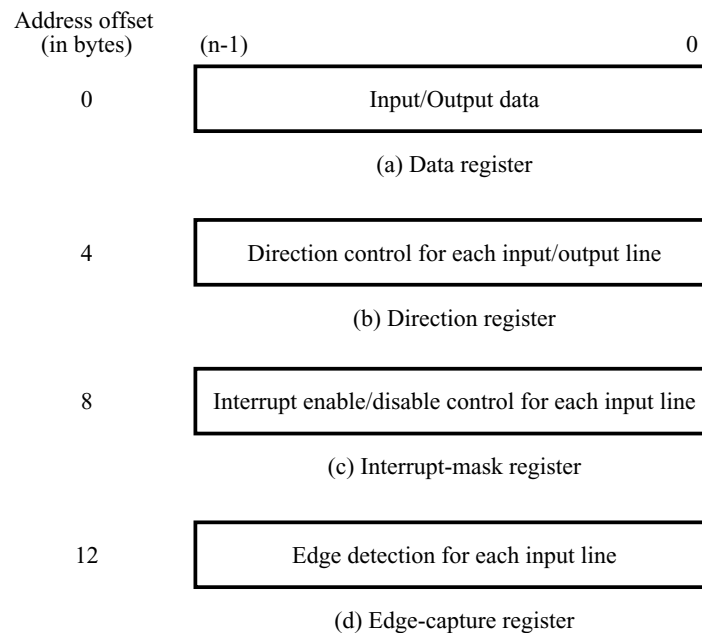


Figure 1: Registers in the parallel port interface.

Each register is  $n$  bits long. The registers have the following purpose:

- *Data* register: holds the  $n$  bits of data that are transferred between the parallel port and the ARM processor. It can be implemented as an input, output, or a bidirectional register.
- *Direction* register: defines the direction of transfer for each of the  $n$  data bits when a bidirectional interface is generated.

- *Interrupt-mask* register: used to enable interrupts from the input lines connected to the parallel port.
- *Edge-capture* register: indicates when a change of logic value is detected in the signals on the input lines connected to the parallel port. Once a bit in the edge capture register becomes asserted, it will remain asserted. An edge-capture bit can be de-asserted by writing to it using the ARM processor.

Not all of these registers are present in some parallel ports. For example, the *Direction* register is included only when a bidirectional interface is permitted by the parallel port. The *Interrupt-mask* and *Edge-capture* registers must be included if interrupt-driven input/output is used.

The parallel port registers are memory mapped, starting at a specific *base* address. The base address becomes the address of the *Data* register in the parallel port. The addresses of the other three registers have offsets of 4, 8, or 12 bytes (1, 2, or 3 words) from this base address. In the DE1-SoC Computer parallel ports are used to connect to SW slide switches, KEY pushbuttons, LEDs, and seven-segment displays.

### Conforming to the ARM Procedure Call Standard

The ARM *Procedure Call Standard* (PCS) provides guidelines (“rules”) for how registers should be used in a program, especially when subroutines are involved. The PCS states that a subroutine is allowed to modify only ARM registers R0 to R3, but it is not permitted to have changed the contents of ARM registers R4 to R11 after execution of the subroutine. If you need to use registers R4 to R11 inside a subroutine, then you must save their current values by pushing them onto the stack at the beginning of the subroutine, and then restore these saved values by popping them off the stack before returning from the subroutine. (Note: if you are using CPUlator, then you will see an error stating that you have “clobbered” one or more of registers R4 to R11 if you over-write these registers in a subroutine without saving/restoring their values as required by the PCS.)

The proper procedures for passing parameters into a subroutine, and returning results, are also specified by the PCS. It states that parameters have to be passed to a subroutine using registers R0 to R3. If you need to transmit more than four parameters into a subroutine, then the “excess” ones should be passed via the stack. Similarly, a single result has to be returned from a subroutine using register R0. If more results need to be returned, then registers R1 to R3 can be used. If there is a need to return even more than four results, then these can be passed back to the caller via the stack.

**You are required to follow the PCS** for all ARM code that you write in this exercise, in all subsequent lab exercises, and on all test/exam questions in this course.

## Part I

You are to write an ARM assembly language program that displays a decimal digit on the seven-segment display *HEX0* as described below. The other seven-segment displays *HEX5* – 1 should be set to be blank.

The DE1-SoC Computer contains a parallel port connected to the seven-segment displays *HEX3* – 0. The port is memory mapped at the base address 0xFF200020. A second parallel port is connected to *HEX5* – 4, at the base address 0xFF200030. Figure 2 shows how the display segments are connected to the parallel ports.

The following functionality should be included:

- If *KEY<sub>0</sub>* is pressed on the board, you should set the number displayed on *HEX0* to 0.
- If *KEY<sub>1</sub>* is pressed then you should increment the displayed number, but don’t let the number go above 9 (i.e. pressing the key won’t change the value if it is already at 9).
- If *KEY<sub>2</sub>* is pressed then decrement the number, but don’t let the number go below 0 (i.e. pressing the key won’t change the value if it is already 0).
- Pressing *KEY<sub>3</sub>* should blank the display, and pressing any other KEY after that should return the display to 0.

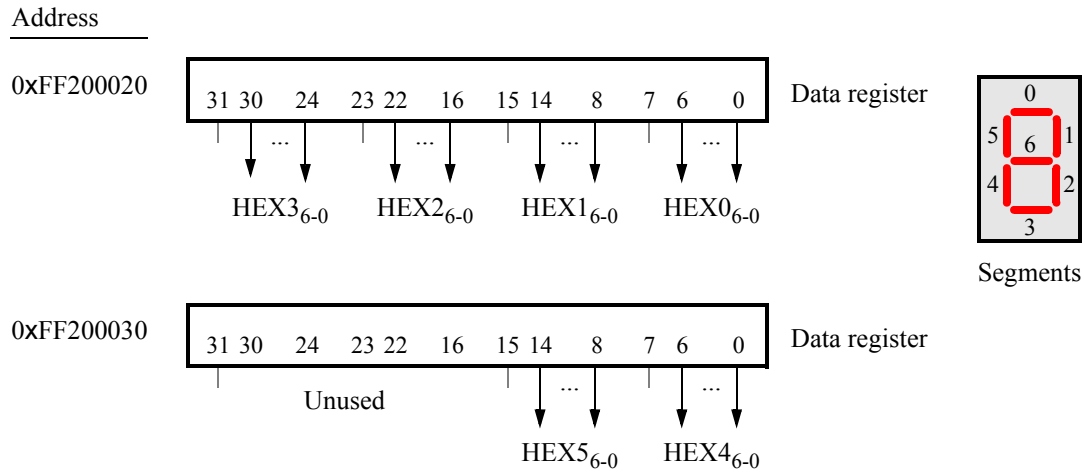


Figure 2: The parallel ports connected to the seven-segment displays  $HEX5 - 0$ .

Address	31	30	...	4	3	2	1	0	
0xFF200050	Unused				KEY <sub>3-0</sub>				Data register
Unused	Unused								
0xFF200058	Unused				Mask bits				Interruptmask register
0xFF20005C	Unused				Edge bits				Edgecapture register

Figure 3: The parallel port connected to the pushbutton  $KEYs$ .

- The parallel port connected to the pushbutton  $KEYs$  has the base address 0xFF200050, as illustrated in Figure 3. In your program, use the *polling* I/O method to read the *Data* register to see when a button is being pressed.
- When you are not pressing any  $KEY$  the *Data* register provides 0, and when you press  $KEY_i$  the *Data* register provides the value 1 in bit position  $i$ . Once a button-press is detected, be sure that your program waits until the button is released. You *must not* use the *Interruptmask* or *Edgecapture* registers for this part of the exercise.

Create a new folder to hold your solution for this part and put your code into a file called *part1.s*. You will need to make a Monitor Program project for running your code on a DE1-SoC board; as mentioned earlier, you may want to and test and debug your program at home using CPULator. You can refer to previous lab exercises for examples of assembly-language code for displaying numbers on the seven-segment displays.

## Part II

Write an ARM assembly language program that displays a two-digit decimal *counter* on the seven-segment displays  $HEX1 - 0$ . The counter should be incremented approximately every 0.25 seconds. When the counter reaches the value 99, it should start again at 0. The counter should stop/start when any pushbutton  $KEY$  is pressed.

To achieve a delay of approximately 0.25 seconds, use a delay-loop in your assembly language code. A suitable example of such a loop is shown below, which gives a good value for the delay when using the actual ARM processor on the DE1-SoC board (for the CPULator, a much smaller delay value (e.g., =500000) has to be used,

because the *simulated* ARM processor in the CPULator tool “executes” code *much* more slowly than the real ARM processor on the DE1-SoC board.)

```
DO_DELAY:  LDR    R7, =200000000 // for CPULator use =500000
SUB_LOOP:  SUBS   R7, R7, #1
          BNE     SUB_LOOP
```

To avoid “missing” any button presses while the processor is executing the delay loop, you should use the *Edge-capture* register in the *KEY* port, shown in Figure 3. When a pushbutton is pressed, the corresponding bit in the *Edgecapture* register is set to 1; it remains set until your program resets it back to 0. You reset an *Edgecapture* bit by writing a 1 into the corresponding position of the register.

Put your code into a file called part2.s, and test and debug your program.

### Part III

In Part II you used a delay loop to cause the ARM processor to wait for approximately 0.25 seconds. The processor loaded a large value into a register before the loop, and then decremented that value until it reached 0. In this part you are to modify your code so that a *hardware timer* is used to measure an exact delay of 0.25 seconds. You should use polling I/O to cause the ARM processor to wait for the timer.

The DE1-SoC Computer includes a number of hardware timers. For this exercise use the timer called the ARM A9 *Private Timer*. As shown in Figure 4 this timer has four registers, starting at the base address 0xFFEC600. To use the timer you need to write a suitable value into the *Load* register. Then, you need to set the enable bit *E* in the *Control* register to 1, to start the timer. The timer starts counting from the initial value in the *Load* register and counts down to 0 at a frequency of 200 MHz. The counter will automatically reload the value in the *Load* register and continue counting if the *A* bit in the *Control* register is set to 1. When it reaches 0, the timer sets the *F* bit in the *Interrupt status* register to 1. You should poll this bit in your program to cause the ARM processor to wait for the timer. To reset the *F* bit to 0 you have to write the value 1 into this bit-position.

Address	31	...	16	15	...	8	7	3	2	1	0	Register name
0xFFEC600	Load value											Load
0xFFEC604	Current value											Counter
0xFFEC608	Unused					Prescaler		Unused	I	A	E	Control
0xFFEC60C	Unused										F	Interrupt status

Figure 4: The ARM A9 Private Timer registers.

Put your code into a file called part3.s, and test and debug your program.

## Part IV

In this part you are to write an assembly language program that implements a real-time clock. Display the time on the seven-segment displays *HEX3* – 0 in the format *SS:DD*, where *SS* are seconds and *DD* are hundredths of a second. Measure time intervals of 0.01 seconds in your program by using polled I/O with the ARM A9 Private Timer. You should be able to stop/run the clock by pressing any pushbutton *KEY*. When the clock reaches *59:99*, it should wrap around to *00:00*.

Put your code into a file called *part4.s*, and test and debug your program.