# Laboratory Exercise 7

Using Interrupts with C code

The purpose of this exercise is to investigate the use of interrupts for the ARM A9 processor, using C code. To do this exercise you need to be familiar with the exceptions processing mechanisms of the ARM processor, and with the operation of the ARM Generic Interrupt Controller (GIC). These concepts are discussed in the tutorials *Introduction to the ARM Processor*, and *Using the ARM Generic Interrupt Controller*. You should also read the parts of the DE1-SoC Computer documentation that pertain to the use of exceptions and interrupts with C code.

This exercise involves the same tasks as those given in Exercise 5, except that this exercise uses C code rather than assembly-language code.

### Part I

Consider the main program shown in Figure 1. The program first initializes the ARM A9 stack pointer for IRQ (interrupt) mode by calling a subroutine named *set_A9_IRQ_stack()*. This step is necessary because, although the C compiler automatically generates code that initializes the SVC-mode (supervisor mode) stack pointer, the C compiler does not generate code to initialize the IRQ-mode stack pointer. The main program then calls subroutines *config_GIC()* to initialize the generic interrupt controller (GIC), and *config_KEYs()* to initialize the pushbutton KEYs port so that it will generate interrupts. Finally, a subroutine *enable_A9_interrupts()* is called to unmask IRQ interrupts in the ARM processor. You are to fill in the missing code for the subroutines in Figure 1. After completing the initialization steps described above, the main program just "idles" in an endless loop.

The purpose of your program is to show the numbers 0 to 3 on the *HEX*0 to *HEX*3 displays, respectively, when a corresponding pushbutton *KEY* is pressed. Since the main program only idles in a loop, you have to control the displays by using an interrupt service routine for the pushbutton KEYs port. Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file, such as *part1.c*, for your main program, and create any other source-code files that you may wish to use. Write the code for the subroutines that are called by the main program. For the *config_GIC()* subroutine set up the GIC to send interrupts to the ARM processor from the pushbutton KEYs port.
2. Figure 2 gives the C code required for the interrupt handler. It is declared with the **__attribute__** specification *interrupt*, and has the special name *__cs3_isr_irq*. Using this declaration allows the C compiler to recognize the code as being the IRQ interrupt handler. The compiler generates an entry corresponding to this code in the ARM exception-vector table.

   You have to write the code for the *pushbutton_isr()* interrupt service routine. Your code should show the digit 0 on the *HEX*0 display when $KEY_0$ is pressed, and then if $KEY_0$ is pressed again the display should be "blank". You should toggle the *HEX*0 display between 0 and "blank" in this manner each

time $KEY_0$ is pressed. Similarly, toggle between "blank" and 1, 2, or 3 on the *HEX*1 to *HEX*3 displays each time $KEY_1$, $KEY_2$, or $KEY_3$ is pressed, respectively.

The bottom part of Figure 2 provides code, using simple loops, which can be used for the other ARM exception handlers. Including these handlers in your code is optional, because the C compiler will generate these handlers automatically if they are not explicitly provided.

3. Make a new Monitor Program project in the folder where you stored your source-code files. In the Monitor Program screen illustrated in Figure 3, make sure to choose Exceptions in the *Linker Section Presets* drop-down menu. Compile, download, and test your program.

```
int main(void)
{
    set_A9_IRQ_stack ();          // initialize the stack pointer for IRQ mode
    config_GIC ();                // configure the general interrupt controller
    config_KEYs ();               // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts ();      // enable interrupts in the A9 processor

    while (1)                     // wait for an interrupt
        ;
}

/* Initialize the banked stack pointer register for IRQ mode */
void set_A9_IRQ_stack(void)
{
    ··· code not shown
}
/* Configure the Generic Interrupt Controller (GIC) */
void config_GIC(void)
{
    ··· code not shown
}
/* Set up the pushbutton KEYs port in the FPGA */
void config_KEYs(void)
{
    ··· code not shown
}
/* Turn on interrupts in the ARM processor */
void enable_A9_interrupts(void)
{
    ··· code not shown
}
```

Figure 1: Main program for Part I.

```
/* Define the IRQ exception handler */
void __attribute__ ((interrupt)) __cs3_isr_irq (void)
{
    /* Read the ICCIAR from the CPU interface in the GIC */
    int address = MPCORE_GIC_CPUIF + ICCIAR;
    int int_ID = *((int *) address);

    if (int_ID == KEYS_IRQ)             // check if interrupt is from the KEYs
        pushbutton_ISR ( );
    else
        while (1);                      // if unexpected, then stay here

    /* Write to the End of Interrupt Register (ICCEOIR) */
    address = MPCORE_GIC_CPUIF + ICCEOIR;
    *((int *) address) = int_ID;

    return;
}

/* Define the remaining exception handlers */
void __attribute__ ((interrupt)) __cs3_reset (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_undef (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_swi (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_pabort (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_dabort (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_fiq (void)
{
    while (1);
}
```

Figure 2: Exception handlers.

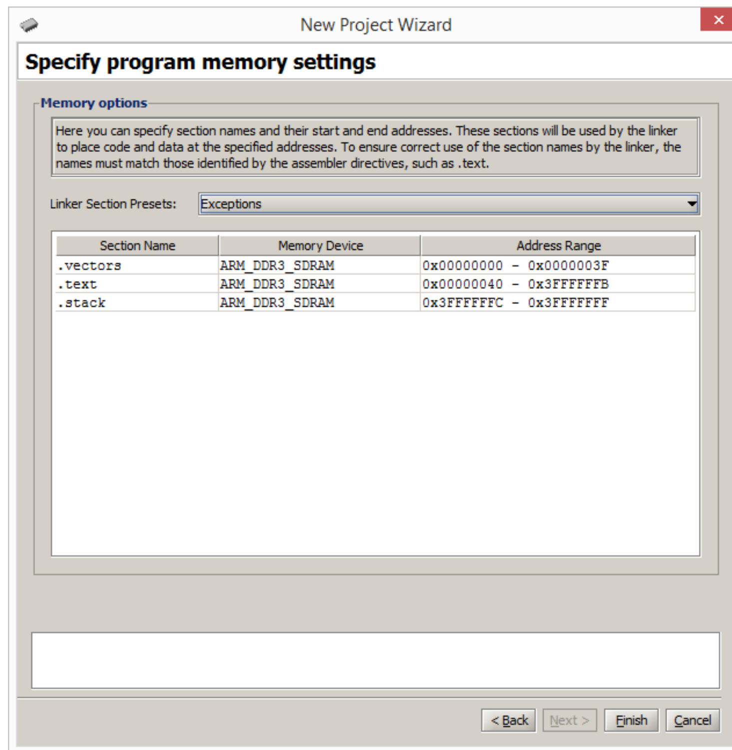


Figure 3: Selecting the Exceptions linker section.

**Part II**

Consider the main program shown in Figure 4. The code is required to set up the ARM stack pointer for interrupt mode, initialize some devices, and then enable interrupts. The subroutine *config_GIC( )* configures the GIC to send interrupts to the ARM processor from two sources: HPS Timer 0, and the pushbutton KEYs port. The main program calls the subroutines *config_HPS_timer( )* and *config_KEYS( )* to set up the two ports. You are to write each of these subroutines. Set up HPS Timer 0 to generate one interrupt every 0.25 seconds.

In Figure 4 the main program executes an endless loop writing the value of the global variable *count* to the red lights LEDR. In the interrupt service routine for HPS Timer 0 you are to increment the variable *count* by the value of the *run* global variable, which should be either 1 or 0. You are to toggle the value of the *run* global variable in the interrupt service routine for the pushbutton KEYs, each time a KEY is pressed. When *run* = 0, the main program will display a static count on the red lights, and when *run* = 1, the count shown on the red lights will increment every 0.25 seconds.

Make a new Monitor Program project for this part, and assemble, download, and test your code.

```
int count = 0;                      // global counter for red lights
int run = 1;                        // global, used to increment/not the count variable

int main(void)
{
    volatile int * LEDR_ptr = (int *) 0xFF200000;

    set_A9_IRQ_stack ();            // initialize the stack pointer for IRQ mode
    config_GIC ();                  // configure the general interrupt controller
    config_HPS_timer ();            // configure HPS Timer 0
    config_KEYs ();                 // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts ();        // enable interrupts in the A9 processor

    while (1)                       // wait for an interrupt
        *LEDR_ptr = count;
}

/* Initialize the banked stack pointer register for IRQ mode */
void set_A9_IRQ_stack(void)
{
    · · · code not shown
}
/* Configure the Generic Interrupt Controller (GIC) */
void config_GIC(void)
{
    · · · code not shown
}
/* setup HPS timer */
void config_HPS_timer( )
{
    · · · code not shown
}
/* Set up the pushbutton KEYs port in the FPGA */
void config_KEYs(void)
{
    · · · code not shown
}
/* Turn on interrupts in the ARM processor */
void enable_A9_interrupts(void)
{
    · · · code not shown
}
```

Figure 4: Main program for Part II.

## Part III

Modify your program from Part II so that you can vary the speed at which the counter displayed on the red lights is incremented. All of your changes for this part should be made in the interrupt service routine for the pushbutton KEYs. The main program and the rest of your code should not be changed.

Implement the following behavior. When $KEY_0$ is pressed, the value of the *run* variable should be toggled, as in Part I. Hence, pressing $KEY_0$ stops/runs the incrementing of the *count* variable. When $KEY_1$ is pressed, the rate at which the *count* variable is incremented should be doubled, and when $KEY_2$ is pressed the rate should be halved. You should implement this feature by stopping HPS Timer 0 within the pushbutton KEYs interrupt service routine, modifying the load value used in the timer, and then restarting the timer.

## Part IV

For this part you are to add a third source of interrupts to your program, using the A9 Private Timer. Set up the timer to provide an interrupt every 1/100 of a second. Use this timer to increment a global variable called *time*. You should use the *time* variable as a real-time clock that is shown on the seven-segment displays $HEX5-0$. Use the format MM:SS:DD, where *MM* are minutes, *SS* are seconds and *DD* are hundredths of a second. You should be able to stop/run the clock by pressing pushbutton $KEY_3$. When the clock reaches 59:59:99, it should wrap around to 00:00:00.

Make a new folder to hold your Monitor Program project for this part. Modify the main program from Part III to call a new subroutine, named *config_priv_timer( )*, which sets up the A9 Private Timer to generate the required interrupts. To show the *time* variable in the real-time clock format MM:SS:DD, you can use the same approach that was followed for Part 4 of Lab Exercise 6. In that previous exercise you used polled I/O with the private timer, whereas now you are using interrupts. One possible way to structure your code is illustrated in Figure 5. In this version of the code, the endless loop in the main program writes the values of variables named *HEX_code*3_0 and *HEX_code*5_4 to the 7-segment displays.

Using the scheme in Figure 5, the interrupt service routine for the private timer has to increment the *time* global variable, and also update the *HEX_code*3_0 and *HEX_code*5_4 variables that are being written to the 7-segment displays by the main program.

Make a new Monitor Program project and test your code.

```
int count = 0;                              // global counter for red lights
int run = 1;                                // global, used to increment/not the count variable
int time = 0;                               // global, used for real-time clock
int HEX_code3_0 = 0;                        // global, used for 7-segment displays
int HEX_code5_4 = 0;                        // global, used for 7-segment displays

int main(void)
{
    volatile int * LEDR_ptr = (int *) 0xFF200000;
    volatile int * HEX3_HEX0_ptr = (int *) 0xFF200020
    volatile int * HEX5_HEX4_ptr = (int *) 0xFF200030;

    set_A9_IRQ_stack ();                    // initialize the stack pointer for IRQ mode
    config_GIC ();                          // configure the general interrupt controller
    config_priv_timer ();                   // configure the MPCore private timer
    config_HPS_timer ();                    // configure HPS Timer 0
    config_KEYs ();                         // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts ();                // enable interrupts in the A9 processor

    while (1)                               // wait for an interrupt
        *LEDR_ptr = count;
        *HEX3_HEX0_ptr = HEX_code3_0;       // show the time in the format MM:SS:DD
        *HEX5_HEX4_ptr = HEX_code5_4;
}

··· code not shown for other subroutines

/* Set up MPCore private timer */
void config_priv_timer( )
{
    ··· code not shown
}

··· code not shown for other subroutines
```

Figure 5: Main program for Part IV.

Copyright ©2015 Altera Corporation.