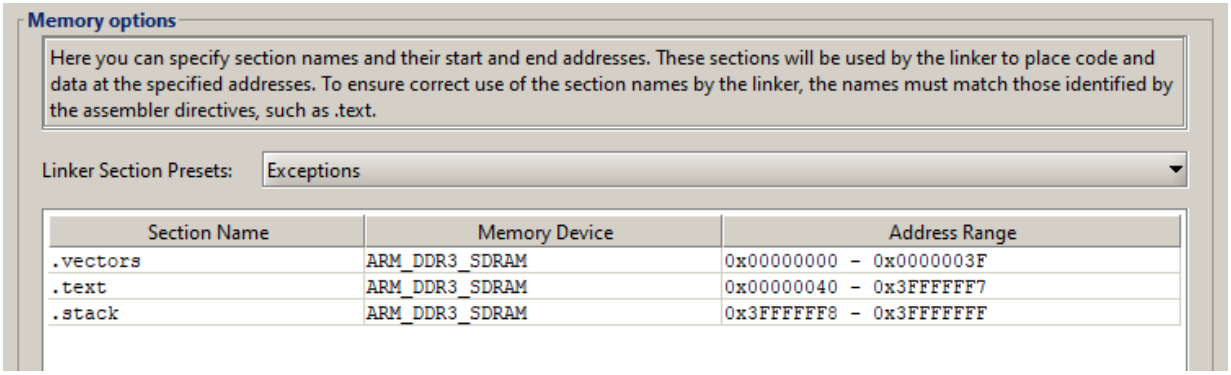
Laboratory 5
Working with Interrupts

Introduction

The objective of this laboratory is to provide you with practical experience working with interrupts/exceptions. Interrupt handling is an extremely important and fundamental aspect of how a microprocessor interacts with the peripheral components to which it is attached. It provides an efficient way for a peripheral to signal to the processor that it requires the processor's attention. The processor can then pause what it is currently doing and quickly respond to the peripheral's request for service.

In this laboratory you will again begin with a supplied example project and modify its operation. Create a new ARM assembly language project and select the *Interrupt Example* sample program. Note that when you get to the last screen of the project setup, the *Memory options* selection will be set to *Exceptions*. (This setting is pre-selected and can't be changed.) You can see from this setup screen that the memory locations used by the ARM processor for its interrupt vector table is defined under the *.vectors* label.



When you build, download, and run the *Interrupt Example* program you will see two things happening. The first is that the red LEDs on the board will strobe left or right. The direction the LEDs move may be toggled by pressing either KEY0 or KEY1. The second activity is that the green LED near the bottom right corner of the board will blink at a frequency of 2 Hz. This example program is actually using three different interrupt events to accomplish this seemingly simple behaviour.

The simplest interrupt event is a reaction to a KEY button being pressed. In the past you actively polled the state of the buttons by reading the data register for the GPIO port to which the KEY buttons are connected. In this program, the GPIO port is setup to generate an interrupt when a KEY is pressed and released. It is actually the release of the button which triggers the interrupt request, and the button state is captured and saved at that time. In response to the KEY interrupt, the processor changes the direction in which the LEDs are moving.

The other two interrupts in this example are generated by two separate timers. The timers are simply counters which count down from a starting value (setup at the beginning of the program) to zero. When they reach zero, an interrupt is generated, and the counter is automatically reset to the same starting value and begins counting again. One of these timers, called the *Interval Timer*, is setup to produce an interrupt every 50ms. That interrupt signals to the processor that it should shift the pattern on the red LEDs over by one position. Similarly, the second timer, called the *HPS Timer*, is setup to produce an interrupt every second. That interrupt indicates to the processor that it should toggle the state of the green LED.

To keep the different aspects of this program organized, the project is broken up into eight separate program files. A brief description of each is provided in the following table.

File Name	Description
`interrupt_example.s`	Main program file that performs setup and contains the main program loop.
`exceptions.s`	Defines the service routines for the different types of exceptions that might occur. The three interrupts used in this program are all of the IRQ type.
`hps_timer_isr.s`	The interrupt service routine for the HPS Timer (which controls the green LED).
`key_isr.s`	The interrupt service routine for the KEY buttons.
`interval_timer_isr.s`	The interrupt service routine for the Interval Timer (which controls the red LEDs).
`defines.s`	General .equ definitions used in this program.
`interrupt_IDs.s`	The list of identifiers used to distinguish between the different sources of the IRQ interrupt.
`address_map_arm.s`	List of addresses where different hardware components are memory mapped. (This same file has appeared in all the example projects.)

Procedure

Create, build, and run the *Interrupt Example* program. Observe its operation, and how the system responds to the KEY press. Notice that the red LEDs change direction when the KEY is released, not when it is pressed. Also notice that the flashing of the green LED is not affected by pressing or holding the KEY buttons, but operates completely independently.

Both the timer and the key interrupts are all in the IRQ interrupt category. When any of those interrupts occur, the processor will follow the IRQ vector in the interrupt vector table (at memory location `0x00000018`). That vector is a branch to the `SERVICE_IRQ` routine located in the `exception.s` file. That routine first saves part of the processor state by pushing registers `{R0–R7,LR}` onto the stack. It then checks for the source of the interrupt by reading the interrupt ID number from the interrupt controller. This ID number tells the processor which peripheral triggered the IRQ interrupt. It then branches to a different service routine for each of the three interrupt sources: `KEY_ISR`, `TIMER_ISR` (for the interval timer), or `HPS_TIMER_ISR`. When the ISR ends, it branches back to `SERVICE_IRQ` which clears the interrupt, and restores the original processor state by popping `{R0–R7,LR}` back off the stack before returning from the interrupt.

Examine each of the three ISRs to familiarize yourself with their general operation.

Modifying the Program

1. The example program is setup to respond to KEY0 or KEY1, but ignores KEY2 and KEY3. This is controlled by the setting of an interrupt mask register for that specific GPIO port. Currently the mask value is set to `0x03` in the CONFIG_KEYS section of the `interrupt_example.s` file. Changing this value to `0x0F` will enable interrupts for any of the KEY buttons.
   Modify the `KEY_ISR` and `TIMER_ISR` routines as needed so that KEY0 sets the LED direction to left-to-right (instead of toggling it), while KEY1 sets the direction to right-to-left.

2. Modify the program further, so that KEY3 stops the red LEDs from cycling, while KEY2 starts the red LED cycling again. Note, do not change the timer settings to achieve the stated behaviour, but rather modify `TIMER_ISR` (and any other parts of the program you deem necessary) so it does not update the LEDs when in the stopped state.

3. For the green LED, the `HPS_TIMER_ISR` code currently increments a location in memory, labelled `TICK`, each time the *HPS Timer* generates an interrupt. The mainline program (in `interrupt_example.s`) continuously checks the `TICK` memory location to see if it is zero or non-zero. If zero, then it does nothing but check again. When the `TICK` location becomes non-zero, it is the mainline that toggles the state of the green LED. It also sets the `TICK` value back to zero again. Modify the mainline and the `HPS_TIMER_ISR` code so that the state of green LED is toggled directly by the `HPS_TIMER_ISR`, not in the mainline.