EE128: Data Acquisition, Instrumentation and Process Control
University of California, Riverside

Lab 3: Interrupt Handling

Spring 2019

Objective

To verify concept and to become familiar with interrupt-handling techniques for the 9S12DG256 microcontroller, and to improve skills on designing and debugging microcontroller based systems.

References

- Dragon12-JR Trainer User's Manual
- Freescale MC9S12DG256 Device User Guide
- Freescale HCS12 Core User Guide
- Freescale HCS12 Interrupt Module Manual
- Reference Guide for D–Bug12 Version 4.x.x

Equipment

- PC running MS Windows
- Digital Multi-Meter (DMM)
- Dragon12-JR 9S12DG256 EVB
- Function generator
- **Breadboard** (you need to bring one)

Parts

- 2 each 7-segment display, common cathode
- 1 each 4-bit DIP switch (piano type)
- 2 each 390Ω x 8, independent resistor network (DIP)

Software

- Freescale CodeWarrior for HC12 v5.1 (C cross compile & programming environment)
- RealTerm (terminal emulation program)
    - If you don't like RealTerm, try TeraTerm, which is another serial terminal with a file transmission capability.

Background

For this lab, you will use the $\overline{\text{IRQ}}$ pin to generate interrupts. The $\overline{\text{IRQ}}$ pin can be set to interrupt based on a low level signal (default) or on a falling edge. In this lab, we want to trigger our interrupts on a **falling edge**. To enable the $\overline{\text{IRQ}}$ pin to trigger on a falling edge, there is an interrupt control register called **INTCR** (located at memory location 0x001E). At the beginning of your program, you need to set bit 7 to 1. Bit 6

should also be set to 1 to enable IRQ interrupts. You can accomplish this by simply using the following statement:

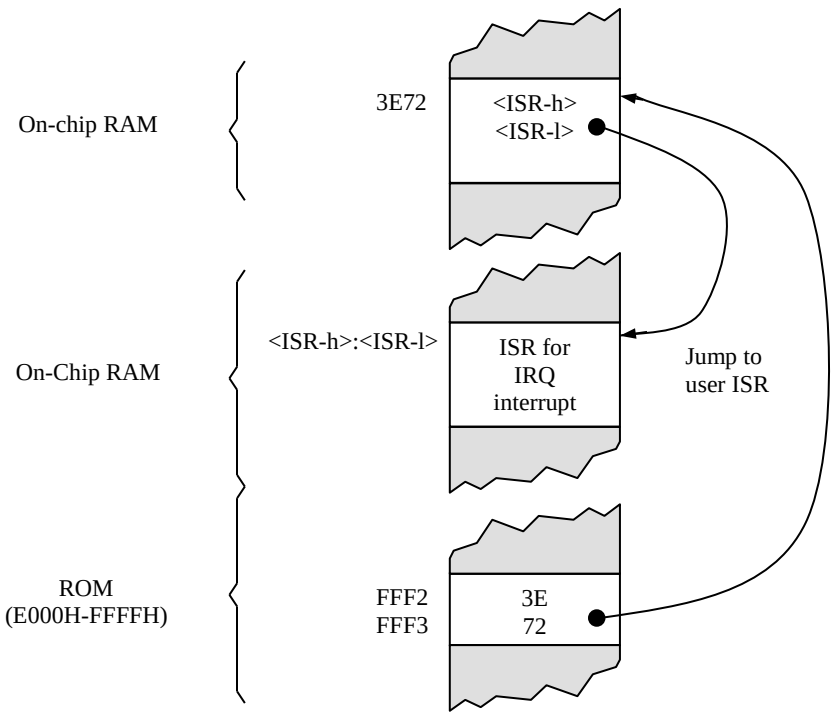
```
INTCR = 0xc0;
```

or by using pre-defined bit assignments:

```
INTCR_IRQE = 1;       /* IRQ to be falling edge triggered */
INTCR_IRQEN = 1;      /* Enable IRQ Interrupt */
```

Once this is set up, a falling edge on the $\overline{\text{IRQ}}$ pin causes a hardware interrupt to 9S12DG256. A user *interrupt service routine* (ISR), specifically written for the *IRQ interrupt*, will be invoked. The address of this routine (for IRQ interrupt) must be stored in the *IRQ interrupt vector* -- a Motorola specified location, specifically, FFF2-FFF3 in the 9S12DG256 addressing space.

However, this area is pre-occupied by the DBug-12 monitor program in flash memory. For our RAM application, we want to keep DBug-12 intact and not modify the interrupt vectors in flash memory (however, for more permanent projects, we may want to place our own program in flash memory and program the specific vectors). To get around this, the DBug-12 monitor uses a *pseudo interrupt vector* for each type of interrupt. This pseudo interrupt vector table is in RAM (starting at 0x3E00) and is initialized to all zeros. If you want to use interrupts in the RAM application mode, then you simply replace the ISR address into the corresponding pseudo interrupt vector table. For example, the starting address of the IRQ_ISR should be placed at location (0x3E72). For more details on how this works, please refer to the DBug-12 manual.



ISR Installation

There are two common practices to install a user-defined ISR. One is to pre-calculate the address of the ISR and load it into either the real vector table (when running a flash mode program) or into the pseudo vector table (when running a RAM application). This way, you tell the compiler that you want to place the ISR always at the same location in memory. This can be tricky, especially if you don't know the sizes of the interrupt service routines.

Alternatively, you can have the linker determine the memory location of the ISR, and place this address into the specific vector entry. This structure looks like this:

```
typedef void (*near tIsrFunc)(void); /* Type of interrupt vector entry */

__interrupt void IRQ_ISR(void); /* Declaration of ISR for IRQ */

const tIsrFunc _vect @0x3E72 = IRQ_ISR; /* 0x3E72: entry for IRQ */
```

Your then have to define your ISR and main code as follows:

```
__interrupt void IRQ_ISR(void) {
      < interrupt code >
}

void main(void) {
      PORTE = 0x2;      /* IRQ PIN PE1 PULL HIGH */
      INTCR = 0xC0;     /* IRQ to be falling edge triggered */
                        /* and enable IRQ Interrupt */

      EnableInterrupts;

      < main code (loop) >
}
```

Lab Exercise

In this lab, we will design and implement a digital counter driven by interrupts on the 9S12DG256 board:

1) A "clock tick" circuit, which is conceptually a flip-flop Q driven by the hardware signal CLK. This clock tick will be displayed using a single *decimal point* on a 7-segment display. For every clock tick received, the decimal point should pulse on and off (Use set up in Figure 1).

2) A two-decade, bi-directional counter, (whose value ranges from 0 through 99 decimal), driven by the same CLK signal will be created. The operation of the counter is dictated by the following two digital signals:

   CNT_HLD: 0 for normal operation, and 1 for hold; and
   CNT_DIR: 0 for upward counting, and 1 for downward.

   The state of this counter is to be displayed using two 7-segment LED displays. See Figure 1.

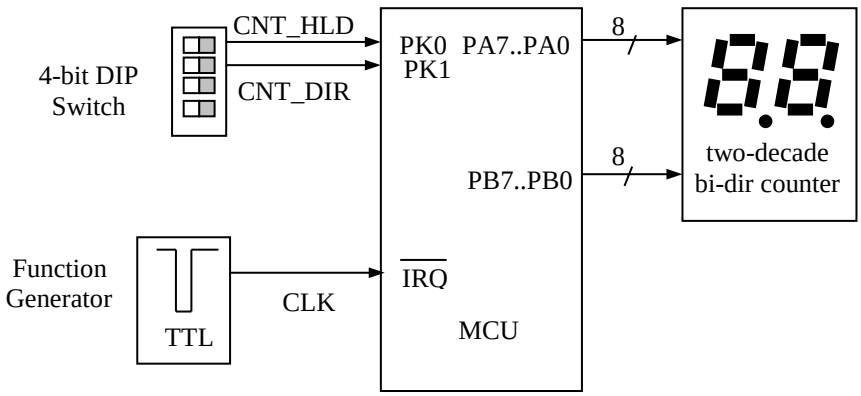


Figure 1

Design Considerations

Resource Allocation

The seven segment displays can be driven through Ports A and B. Make sure you use current limiting resistors.

The control signals can be read in through Port K. The external CLK is introduced into the system through the $\overline{\text{IRQ}}$ pin.

Programming

Make sure you are able to write the following programs based on your hardware resource allocation:

- Update and display the clock tick;
- Update and display the counter based on CNT_HLD and CNT_DIR (for the counter);

The control program contains three parts:

1. Configuration and initialization

This part of the program sets up the interrupts and initializes the counter (and rotator).

2. Interrupt service routine (ISR)

This portion of the program implements the following three operations: (i) turning the LED on and off, (ii) rotating the one-hot bi-directional counter according to its control signals, and (iii) updating the 2-decade, bi-directional counter.

Invoked upon each IRQ interrupt, the ISR updates the counters according to its control signals, or flips the conceptual flip-flop for the clock-tick. While the display for the latter must be updated every time, the two 7-segment LED displays for the counter are updated only if CNT_HLD is 0.

In summary, your ISR should implement the following pseudo-code:

```
Let counter be a static variable initialized to 0;

__interrupt void IRQ_ISR(void)
{
      Update the clock tick and its display;

      Read Port K;
      if (ROT_HLD is 1) return;     /* rotate hold */
      if (ROT_DIR is 1)             /* rotate left */
            rotate left -- wrap around if end is reached;
      else
            rotate right – wrap around if end is reached;

      Update counter displays;
}
```

3. Main program

The functionality of the main program after initialization is to do nothing. But your configuration/initialization code should resemble the following:

```
void main (void)
{
      Configure Port A and B to be 8-bit output ports;
      Enable IRQ interrupts;

      for (;;){}
}
```

Some **important** tips for this lab:

- Remember to disconnect the wire connected to $\overline{\text{IRQ}}$ whenever you download your code to the EVB.
- Check the *.map file to find the entry address of your code.

Pre-lab

1. Draw a schematic diagram for the experiment.
2. Write the required C program code for the experiment.
3. What is the 9S12 pin number for $\overline{\text{IRQ}}$?

Do you want to complete the experiment during lab hours? Draw a schematic diagram and write program code before attending the lab session.

For the 7-segment display, check the pin assignment first.

Procedures

1. Design the required circuit for the experiment.
2. Build your source program(s) with the compiler to generate the Motorola S-record (.s19) file.
3. Download and run your programs. Continue only if your program works correctly.

Questions

1. Measure the time duration from the falling edge of the CLK signal to the change of the decimal point (from on to off, or vice versa) on a 7-segment display.
2. Based on your experiment, estimate the highest frequency of the CLK signal under which the system can operate correctly.
3. For this lab, we use a falling-edge generated interrupt. If you use level-sensitive interrupts, what extra steps (and/or hardware) would you need? (don't write the entire code or schematic; answer in a brief manner)
4. What are the factors contributing to the interrupt latency? Answer briefly.

Lab Demo (50 points)

Demonstrate your working system to the TA and get a confirmation of completion

Lab Report (50 points)

Make sure you include the following in your report:

1. Abstract (a short paragraph stating the objectives and accomplishments)
2. Experiment system specification (what has been designed and implemented) – **10 points**
   - Flowchart diagram (show how your system works)
   - Photos of your boards and circuits
3. Hardware design – **10 points**
   - Draw schematics of your own; do not copy and paste from the handout
4. Software design – **10 points**
   - High level description of the software
   - Program listing (including comments)
5. Technical problems encountered, and how they are solved
6. Answers to the questions – **10 points**
7. Conclusion
   - A very short concluding remark
   - Summary of the contributions of each member