University of Victoria
Department of Electrical & Computer Engineering
CENG 255 - Introduction to Computer Architecture
Laboratory Manual
Laboratory Experiment #4

By

Farshad Khunjush, Nikitas J. Dimopoulos, and Kin F. Li

Laboratory 4: Polling & External Interrupts

4.1 Goal

In this lab, you will learn the different kinds of exceptions, and the behavior of exception in general.

4.2 Objectives

Upon completion of this lab, you will be able to:

- Recognize different kinds of exceptions
- Understand the behavior of interrupts.
- Understand the advantages and disadvantages of interrupts relative to polling.
- What is an Interrupt Service Routine (ISR) and how to write it.

4.3 **Prelab (You are required to submit your written preparation for this part, which will be graded by the lab instructor during the lab, and you have to include the graded Prelab in your final report.)**

- **Explain different kinds of exception.**
- **What address does a ColdFire processor branch to when receiving an external interrupt?**
- **What is the purpose of the VBR register?**
- **What is the purpose of the SR[s] bit?**
- **What is the purpose of the SR[i] bit?**
- **What is an exception vector table?**
- **Explain PORT TC Control registers and their functions.**

4.4 Discussion

In this section, we discuss trap and interrupt concepts that could change the execution flow of the program when special conditions occur. They are designed to be special

events whose occurrence cannot be predicted precisely. It is worth mentioning that traps occur as a result of an event during a program's execution. However, an interrupt is an event in hardware that is triggered by an external source.

4.4.1 Traps

A ***trap*** is a kind of automatic procedure call initiated by some events during the execution of a program. For example, if the divisor of a division instruction is equal to zero, a ***divided-by-zero*** trap will occur. As a result of this, the execution of the current program is transferred to some ***fixed*** memory location instead of continuing in sequence in the executing program. At the fixed location, there exists a branch instruction to a routine called the ***trap handler***, which performs some appropriate actions. The essential point about a trap is that it is ***synchronous*** with the program execution, meaning that it happens as a result of executing some instructions. Some common traps are ***undefined op-codes***, ***privileged instructions***, ***virtual memory page faults***, and ***system calls*** (i.e., operating system services).

4.4.2 Interrupts

An interrupt is an event in hardware that triggers the processor to jump from its current program counter to a specific point. Interrupts are designed to handle special events which occurrence cannot be predicted precisely. The main difference between interrupts and traps is that interrupts are initiated by external devices and are asynchronous with the program's execution. Similar to traps, an interrupt stops the running program and transfers control to a fixed address that calls the appropriate ***interrupt handler*** (a.k.a. Interrupt Service Routine (ISR)), which performs some appropriate actions. The control will be returned to the interrupted program, when the handler finishes. The critical part of returning from the interrupt handler is to restart the interrupted process in the same state it was when the interrupt occurred.

The external interrupt's address where programs are transferred to depends on the hardware. For instance, the external interrupt handler's address in ColdFire depends on an 8-bit vector number, which is received from the interrupt controller module.

4.4.2.1 ColdFire Interrupt Structure

In ColdFire, the interrupt controller generates the interrupts.

The ColdFire architecture provides support to respond to interrupts. After an interrupt occurs hardware takes the following steps:

- Stops executing the current program.
- Saves the state of the machine:
    - Makes an internal copy of SR (status Register) and enters supervisor mode by setting the SR[s] bit, and the interrupt priority mask SR[i] is set to the level of the current IRQ. This inhibits all interrupts of a lower level.
- Determines the 8-bit interrupt source that comes from the interrupt controller module.
- Creates an exception stack frame on top of the stack. This frame includes two 32-bit longwords. The first is the PC of the instruction that was executing when the interrupt occurred. The second word consists of several fields. ColdFire Programmer's Manual discusses this part in more details in Chapter 11.
- Determines the starting address of the interrupt handler. This address is obtained as follows:
    - Vector number = 64 + interrupt source
    - VBR is a register that stores the base address of the exception vector table
    - Interrupt handler's staring address = VBR + 4 * Vector number
- Fetches the instruction at the address and leaves exception processing mode.

The exception vector table contains 256 longwords, which specify the addresses of interrupt handler routines. For example, vector number 0 contains the initial value of stack pointer, and vector number 1 has the initial value of PC. The processor uses these values, when it comes out of reset. Vector numbers 64 through 255 are for user-defined interrupts. We are mainly concerned about these vectors and IRQ1 as well as IRQ7. ColdFire Programmer's Manual discusses this part in more details in Chapter 11.

4.4.3 Polling

Polling a device usually means reading its status register every so often until the device's status changes to indicate that it has a request or has accomplished a job. The disadvantage in this method is that the CPU needs to keep asking the same question over and over when it could be doing useful work. In this situation, the performance of the application is much lower than the situation that interrupts are exploited. However, if the processor is for a single task, we may use this approach due to its simplicity.

4.5 Lab Work

In this section, first, we discuss the loading process of ISR and provide the information that we need to configure and use PORT TC in ColdFire. Then, we present some templates that could be used for polling and interrupt service routines. Finally, you are to change these templates to fulfill the lab's goals.

4.5.1 Setting the ISR's address

As stated, the entries in the vector table determine the addresses of interrupt service routines. For example, the entry for IRQ1 would be at the VBR + (64 + 1) * 4 (i.e., VBR + 0x104) address. The VBR is also initialized to 0x2000_0000 by the startup code. This means that the memory address for IRQ1 interrupt service routine is equal to 0x2000_0104. Therefore, we need to set entries in the vector table to be able to handle the received interrupts. In the CodeWarrior development tools, this can be accomplished by calling the following function:

mcf5xxx_set_handler(int vector_number, void (*handler)()).

For example, to set the interrupt handler of interrupt number one to the following handler we have to call the function as illustrated below:

```
__interrupt__ void sw2_handler () {
        .......
}
```

In the main function we have to use the following code.

mcf5xxx_set_handler(64+1,(uint32) &sw2_handler); // SW2 for IRQ1

It should be noted that, this setting by itself is not sufficient to set up our interrupt handler; we still need to program the interrupt controller module (INTC module). In the following we will elaborate this issue.

4.5.2 Interrupt Controller Registers

In ColdFire, interrupts are controlled through some control registers. These registers are memory-mapped that could be accessed through regular load and store instructions. These registers' addresses are relative to the address of a base register (IPSBAR), which is set to 0x4000_0000 by the startup code. In this part the registers that must be programmed are **interrupt mask** and **interrupt control** registers.

The **interrupt mask register (IMR)** consists of two 32-bit registers: IMRH and IMRL. Bit 1 in IMRL corresponds to interrupt source 1, bit 2 to interrupt source 2 and so on. Upon resetting the processor, the IMR is set to all '1'. If a bit is set ('1') in the interrupt mask register, it means the corresponding interrupt is masked. To enable an interrupt, we have to write a 0 into the correct bit corresponding to the interrupt source. For example, we can use the following code to enable IRQ1.

MCF_INTC0_IMRL &= ~(MCF_INTC_IMRL_MASK1 |MCF_INTC_IMRL_MASKALL);

There is also an **interrupt control register (ICR)** for each interrupt source, ICR1 (MCF_INTC_ICR1) through (MCF_INTC_ICR63). Using this register, we can set the priority and the level of each interrupt source. We do not need to set this register because IRQ1 through IRQ7 have a fixed level and priority. For more information, you might consult MCF52235 ColdFire® Integrated Micro-controller Reference Manual, which is available on the lab's web page.

4.5.3 ColdFire PORT TC information

In ColdFire, the digital I/O pins are grouped into 8-bit ports. Some ports do not use all 8 bits. Each port has registers that configure, monitor, and control the port pins. Figure 1 is a block diagram of the MCF52233 ports.

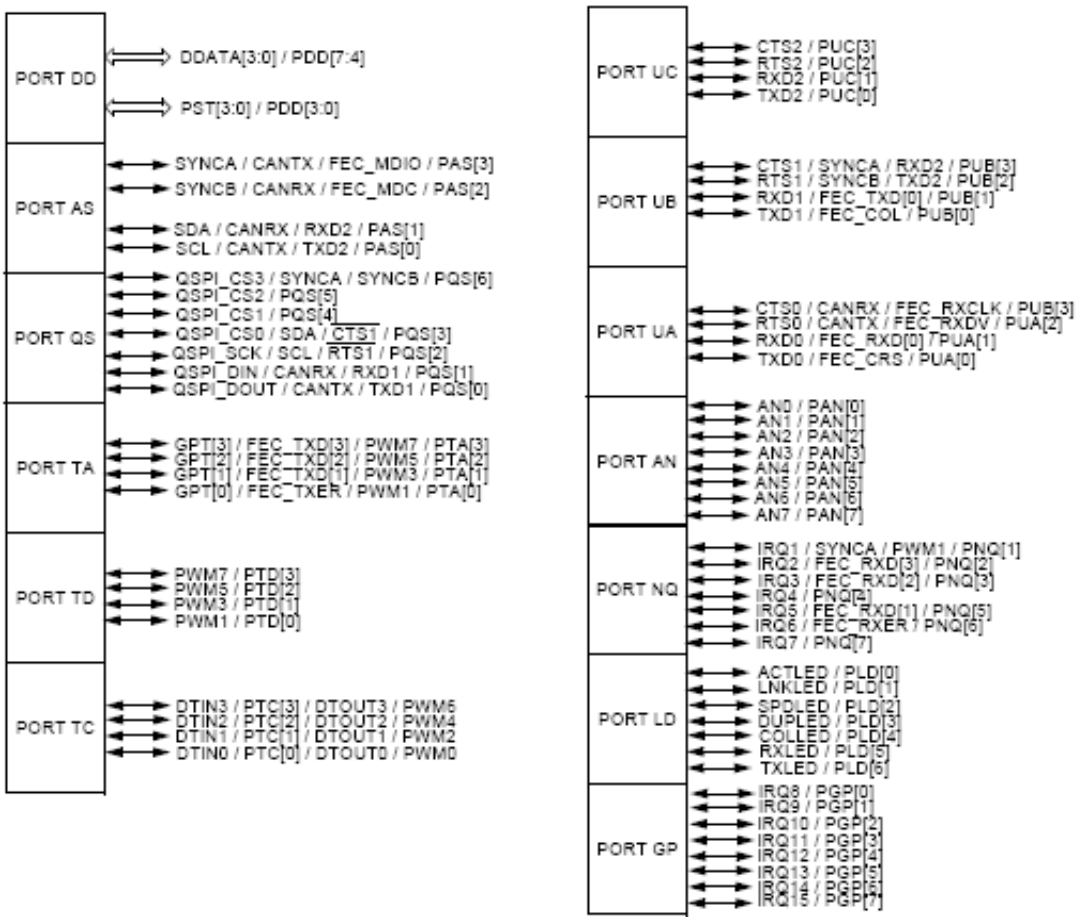

Figure 1- General Purpose I/O Module block diagram

Each port has registers that configure, monitor, and control the port pins. These registers support the following operations:

- Storing output pin data
- Controlling pin data direction
- Reading current pin state
- Setting and clearing output pin data registers

You might consult MCF52235 ColdFire® Integrated Micro-controller Reference Manual, which is available on the lab's web page. In the following, we will describe the registers that are necessary to program general purpose I/O for PORT TC.

4.5.3.1 Port Output Data Registers (PORT*n*)

The PORT*n* registers store the data to be driven on the corresponding port *n* pins when the pins are configured for digital output. The PORT*n* registers with a full 8-bit implementation are shown in Figure 2. The remaining PORT*n* registers use fewer than 8 bits. The PORT*n* registers are read/write. At reset, all bits in the PORT*n* registers are set. Reading a PORT*n* register returns the current values in the register, not the port *n* pin values. PORT*n* bits can be set by setting the PORT*n* register, or by setting the corresponding bits in the PORT*n*P/SET*n* register. They can be cleared by clearing the PORT*n* register, or by clearing the corresponding bits in the CLR*n* register. Port TC data output register is located at IPSBAR+0x10_000F (MCF_GPIO_PORTTC).

	7	6	5	4	3	2	1	0
R W	PORT*n*7	PORT*n*6	PORT*n*5	PORT*n*4	PORT*n*3	PORT*n*2	PORT*n*1	PORT*n*0
Reset:	1	1	1	1	1	1	1	1

Figure 2- Port Output Data Register

4.5.3.2 Port Data Direction Registers (DDR*n)*

The DDR*n* registers control the direction of the port *n* pin drivers when the pins are configured for digital I/O. The DDR*n* registers are read/write. At reset, all bits in the DDR*n* registers are cleared to 0s. Setting any bit in a DDR*n* register configures the corresponding port *n* pin as an output. Clearing any bit in a DDR*n* register configures the corresponding pin as an input. Figure 3 shows the format of these registers.

	7	6	5	4	3	2	1	0
R W	DDR*n*7	DDR*n*6	DDR*n*5	DDR*n*4	DDR*n*3	DDR*n*2	DDR*n*1	DDR*n*0
Reset:	0	0	0	0	0	0	0	0

Figure 3- Port Data Direction Register

Port TC DDR register is located at IPSBAR+0x10_0023 (MCF_GPIO_DDRTC).

4.5.3.3 Port Pin Data/Set Data Registers (PORT*n*P/SET*n*)

The PORT*n*P/SET*n* registers reflect the current pin states and control the setting of output pins when the pin is configured for digital I/O. The PORT*n*P/SET*n* registers are read/write. At reset, the bits in the PORT*n*P/SET*n* registers are set to the current pin states. Reading a PORT*n*P/SET*n* register returns the current state of the port *n* pins. Writing 1s to a PORT*n*P/SET*n* register sets the corresponding bits in the PORT*n* register. Writing 0s has no effect. Figure 4 illustrates this register. This register is located at IPSBAR+0x10_0037 (MCF_GPIO_SETTC).

	7	6	5	4	3	2	1	0
R W	PORTnP7	PORTnP6	PORTnP5	PORTnP4	PORTnP3	PORTnP2	PORTnP1	PORTnP0
Reset:	1	1	1	1	1	1	1	1

Figure 4- PORT*n*P/SET*n* — Port Pin Data/Set Data Registers [7:0]

4.5.3.4 Port Clear Output Data Registers (CLR*n*)

Writing 0s to a CLR*n* register clears the corresponding bits in the PORT*n* register. Writing 1s has no effect. Reading the CLR*n* register returns 0s. The CLR*n* registers are read/write. PORT TC clear output data register is located at IPSBAR+0x10_004B (MCF_GPIO_CLTRC). Figure 5 shows the format of this register.

	7	6	5	4	3	2	1	0
R W	CLRn7	CLRn6	CLRn5	CLRn4	CLRn3	CLRn2	CLRn1	CLRn0
Reset:	0	0	0	0	0	0	0	0

Figure 5- Port Clear Output Data Registers (CLR*n*)

4.5.3.5 Pin Assignment Registers

All pin assignment registers are read/write. If multiple pins are configured for a single function, then the result is undefined.

The **dual function pin assignment registers** allow each pin controlled by each register bit to be configured between the GPIO function and the primary function. Figure 6 shows this register's format.

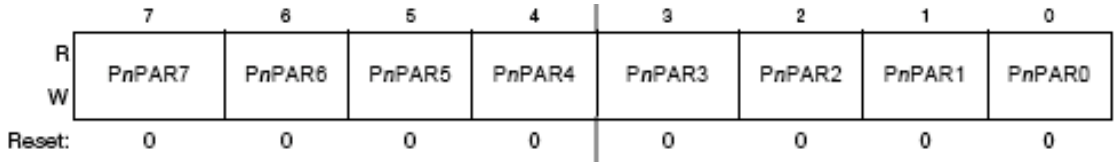

Figure 6- Dual Function Pin Assignment Register

The **quad function pin assignment registers** allow each pin controlled by each register bit to be configured between the GPIO function, primary function, alternate 1 function, and the alternate 2 function. PORT TC pin assignment register is located at IPSBAR+0x10_0057 (MCF_GPIO_PTCPAR). Figure 7 shows this register.

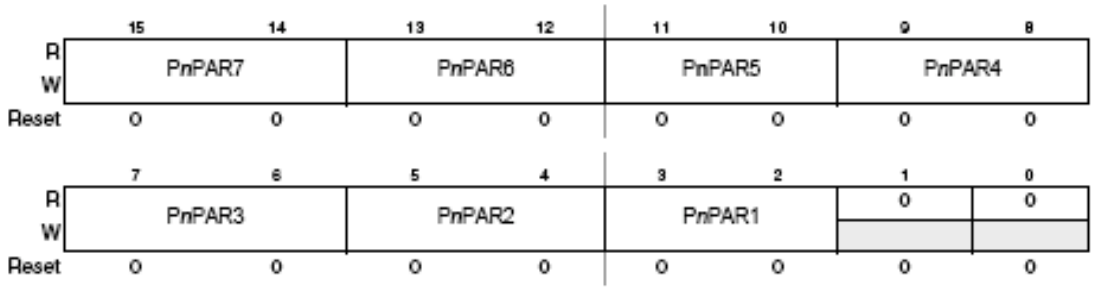

Figure 7- Quad Function Pin Assignment Register

4.6 Polling

This part discusses a method to check the status of a device to perform appropriate actions in a particular situation. The following program tests the status of Port C. This piece of program uses a switch to change the status of port C. In this experiment, initially all LEDs are off. Pressing the switch (SW1) makes one LED toggle.

```
#include <stdio.h>
#include "common.h"

void NewInitIO()
{
```

```
        MCF_GPIO_PTCPAR = MCF_GPIO_PTCPAR_TIN0_GPIO;
        MCF_GPIO_DDRTC = MCF_GPIO_DDRTC_DDRTC0;
        MCF_GPIO_PORTTC = 0x00;

 /* Enable signals as GPIO */
        MCF_GPIO_PTCPAR = 0
            | MCF_GPIO_PTCPAR_TIN3_GPIO
            | MCF_GPIO_PTCPAR_TIN2_GPIO
            | MCF_GPIO_PTCPAR_TIN1_GPIO
            | MCF_GPIO_PTCPAR_TIN0_GPIO;

/* DATA DIRECTION configuration */
/* PORTTC data direction as output PORTTC (3 , 2 , 1 , 0) = 1; */
        MCF_GPIO_DDRTC = 0
            | MCF_GPIO_DDRTC_DDRTC3
            | MCF_GPIO_DDRTC_DDRTC2
            | MCF_GPIO_DDRTC_DDRTC1
            | MCF_GPIO_DDRTC_DDRTC0;

/* IRQ7 data direction configured as inputs */
/* this operation avoids changing the former values of the other pins in PORTNQ */
        MCF_GPIO_DDRNQ = MCF_GPIO_DDRNQ
            & ~MCF_GPIO_DDRNQ_DDRNQ7;

        /* IRQ7 pins (SW1 on 52233) */
        MCF_GPIO_PNQPAR = 0
            | MCF_GPIO_PNQPAR_IRQ7_GPIO;
}
int main()
{
   int Switch1;
        asm
        {
                jsr         NewInitIO
        }
   while (1) {
        Switch1 = MCF_GPIO_SETNQ;
        Switch1 &= MCF_GPIO_PORTNQ_PORTNQ7;
        if (Switch1 == 0x0){
            MCF_GPIO_PORTTC = 0x01;
        }
        else{
             MCF_GPIO_PORTTC = 0x00;
        }
   }
   return 0;
}
```

Figure 8- The code for polling SW1 switch and toggling on M52233

For this part of the lab, you are required to change the program as follows:

a) Revise the program in such a way that another switch (SW2) can do the same operation.

b) Considering having two switches, change the program so that the LEDs toggle

only if you push the switches in the following order: Pushing the switch number one (SW1) twice and the switch number two (SW2) once.

Deliverable:

- **Include a well-commented listing of your program.**
- **Explain the changes that you have made in the provided program and the control registers that you needed to change.**

4.7 Interrupts

In this section, we continue with the interrupt concept. The provided program in Figure 9 uses a switch to send an interrupt to the core. The program is running a loop until it receives an interrupt. Having received the interrupt, it completes the execution of the current instruction and jumps to the interrupt handler. We increment the contents of a counter by one inside the interrupt handler and display its contents on the LEDs.

```
#include <stdio.h>
#include "common.h"

unsigned char GlobalPattern = 0x00;

void SetLEDS(unsigned char Pattern)
{
    MCF_GPIO_PORTTC = Pattern;
}

__interrupt__ void sw1_handler () {
    GlobalPattern = (GlobalPattern + 1) % 16;
    MCF_GPIO_PORTTC = GlobalPattern;
    MCF_EPORT_EPFR0 = MCF_EPORT_EPFR_EPF7;
}

void InitIRQ(void)
{
    mcf5xxx_irq_disable();

/* Initialize the demo board leds. They will all be off after this function. */
/* Enable IRQ1 and IRQ7 in interrupt controller */

    MCF_INTC0_IMRL &= ~(MCF_INTC_IMRL_MASK7 | MCF_INTC_IMRL_MASKALL);

/* Program /IRQ1 and /IRQ7 to be falling-edge sensitive */
    MCF_EPORT_EPPAR0 = 0
        | MCF_EPORT_EPPAR_EPPA7_FALLING;

    MCF_EPORT_EPIER0 = 0
        | MCF_EPORT_EPIER_EPIE7;
```

```
    MCF_EPORT_EPDDR0 = 0x00;

/* Put handlers for SW1 and SW2 into exception vector table */
    mcf5xxx_set_handler(64+7,(uint32) &sw1_handler); // SW1 for IRQ7

/* Unmask all interrupts */
    mcf5xxx_irq_enable();
}

void InitIO(void)
{

    MCF_GPIO_PTCPAR = MCF_GPIO_PTCPAR_TIN0_GPIO;
    MCF_GPIO_DDRTC = MCF_GPIO_DDRTC_DDRTC0;
    MCF_GPIO_PORTTC = 0x00;

 /* Enable signals as GPIO */
    MCF_GPIO_PTCPAR = 0
        | MCF_GPIO_PTCPAR_TIN3_GPIO
        | MCF_GPIO_PTCPAR_TIN2_GPIO
        | MCF_GPIO_PTCPAR_TIN1_GPIO
        | MCF_GPIO_PTCPAR_TIN0_GPIO;

  /* Enable signals as digital outputs */
    MCF_GPIO_DDRTC = 0
        | MCF_GPIO_DDRTC_DDRTC3
        | MCF_GPIO_DDRTC_DDRTC2
        | MCF_GPIO_DDRTC_DDRTC1
        | MCF_GPIO_DDRTC_DDRTC0;
}

int main()
{
    int Pattern = 0x00;
    asm
    {
        jsr      InitIO
        move.l   -6(a6),d0
        move.b   d0,(a7)
        jsr      SetLEDS
        jsr      InitIRQ
       bra.s    *+0
    }

    return 0;
}
```

Figure 9- Program to capture the external interrupt from SW1

You are required to change the program to add the following functionality.

a) Revise the program in such a way that another switch (sw2) generates another interrupt and increments the content of the counter.

b) Considering we have two switches, change the program so that the LEDs toggle

only if you receive interrupts in the following order: Pushing switch one (SW1) three times and switch 2 (SW2) twice.

Deliverable:

- **Include a well-commented listing of your program.**
- **Explain the changes that you have made in the provided program and the control registers that you needed to change.**