MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 5

Lab Guide

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

In the lectures, we learn about **interrupts**, a flexible method that microcontrollers can use to handle important **asynchronous** events that may occur in a system. We will also use the interrupt for communication and rudimentary multi-tasking in embedded systems.

Basic Process

Recall from the lecture that when an interrupt occurs, the microcontroller automatically calls a specialized subroutine called an **interrupt service routine (ISR)** or **interrupt handler**. This routine must follow certain general rules and structures, as we saw in lecture. In this lab, we will investigate setting up interrupts for the 68HC12, as well as data sharing and safe interrupt handling. To begin using interrupts, we must first tell the microcontroller where in memory our ISR is located. Recall that we do so through the use of a **vector table**. This table may be user adjustable, or it may not. The 68HC12 implements an interrupt vector table which is extremely flexible. We can automatically trigger interrupts based on the occurrence of a wide variety of internal and external events:

	Vector Address	Interrupt Source	CCR Mask	Priority
Non-Maskable	0xFFFE – 0xFFFF	Reset	None	Highest
	0xFFFC – 0xFFFD	COP Clock Monitor Fail Reset	None	
	0xFFFA – 0xFFFB	COP Fail Reset	None	
	0xFFF8 – 0xFFF9	Unimplemented Op-code	None	
	0xFFF6 – 0xFFF7	Software Interrupt (SWI)	None	
	0xFFF4 – 0xFFF5	External Interrupt (XIRQ)	'X' bit	
Maskable	0xFFF2 – 0xFFF3	IRQ or Key Wake Up D	'I' bit	
	0xFFF0 – 0xFFF1	Real Time Interrupt	'I' bit	
	0xFFEE – 0xFFEF	Timer Channel 0	'I' bit	
	0xFFEC – 0xFFED	Timer Channel 1	'I' bit	
	0xFFEA – 0xFFEB	Timer Channel 2	'I' bit	
	0xFFE8 – 0xFFE9	Timer Channel 3	'I' bit	
	0xFFE6 – 0xFFE7	Timer Channel 4	'I' bit	
	0xFFE4 – 0xFFE5	Timer Channel 5	'I' bit	
	0xFFE2 – 0xFFE3	Timer Channel 6	'I' bit	
	0xFFE0 – 0xFFE1	Timer Channel 7	'I' bit	
	0xFFDE – 0xFFDF	Timer Overflow	'I' bit	
	0xFFDC – 0xFFDD	Pulse Accumulator Overflow	'I' bit	
	0xFFDA – 0xFFDB	Pulse Accumulator Input Edge	'I' bit	
	0xFFD8 – 0xFFD9	SPI Serial Transfer Complete	'I' bit	
	0xFFD6 – 0xFFD7	SCI0	'I' bit	
	0xFFD4 – 0xFFD5	SCI1	'I' bit	
	0xFFD2 – 0xFFD3	A-to-D Converter	'I' bit	
	0xFFD0 – 0xFFD1	Key Wakeup J (stop wakeup)	'I' bit	
	0xFFCE – 0xFFCF	Key Wakeup H (stop wakeup)	'I' bit	
	0xFF80 – 0xFFCD	Reserved		Lowest

When an interrupt occurs, the CPU responds as soon as the current atomic instruction is completed. Note that there are three 'long' instructions (REV, REVW, and WAV) which **can** be interrupted and are **not** atomic. However, if interrupted they will actually **resume** execution after the interrupt completed (RTI) – they are a special case, and are not common in most other microcontrollers.

Recall that on receiving an interrupt request, the microcontroller first checks if that particular interrupt source is masked. The 68HC12 implements a coarse masking scheme, in that there is effectively only one masking bit (bit 'I' in the CCR) which enables or disables all of the maskable interrupts at the same time (with one exception). If the interrupt source is not masked, then the vector table entry for that source is read and (eventually) loaded into PC, initiating a call to that particular ISR. The following is a brief summary of the potential interrupt sources in the vector table above:

1. **Reset** – Automatically generated when the microcontroller is reset. Can be used in final embedded applications to allow start-up or initialization code to run before main( ) starts. Typically, we will not touch this ISR in the labs – it is used by the boot-loader program that allows us to download programs to the 68HC12 EVB.

2. **COP** – Stands for 'Computer Operating Properly.' This is a form of watchdog timer that can be implemented to prevent deadlocks or infinite waiting in programs. Basically, when activated the user program becomes responsible for resetting the COP timer within a finite time interval. If it does not, then this interrupt is automatically generated, indicating the user program has become unresponsive, and allowing corrective action.

3. **Software Interrupt (SWI)** – An interrupt can be manually generated through an instruction in code. Although conceptually similar to simply calling a pre-defined subroutine, this allows the user program (the one generating the SWI) and the interrupt handler to be maintained **separately**. Since the software interrupt uses the vector table, the user program does not need to know or care about its location or implementation details, or even include its code during compilation. This allows one to implement a simple '**operating system**' of pre-defined routines available to user programs that are **pre-loaded** and **completely separate** from these user programs. This same concept (on a much more complex scale) allows desktop operating systems to function.

4. **External Interrupts (XIRQ and IRQ)** – The 68HC12 makes pins available for **external interrupt handling**. In other words, there are physical pins on the 68HC12 which, when brought electrically high or low (depending on user-specified settings), automatically trigger an interrupt. We can configure a variety of behaviors for XIRQ in particular. Note that XIRQ is the only **partially maskable** interrupt for the 68HC12 – its masking behavior is controlled by the 'X' bit in the CCR, not the 'I' bit as for other sources. Upon reset of the microcontroller, the X bit is set, masking external interrupts. User programs can clear this mask (by 'ANDing' the CCR with 0xBF, for example), enabling external interrupts. Once

unmasked, user software cannot mask external sources (i.e., set the X bit) again, making this source non-maskable. Only a further reset can mask this source.

5. **Unimplemented Op-Code** – Automatically generated if the user attempts to execute an op-code for an instruction that does not exist, or if the formatting is otherwise unexpected.

6. **Real-time Interrupt and Timer Channel 0, 1, 2, …, 7** – We will examine timers in detail in a later lab on analog processing and control loops, but we will make simple use of them here. A **timer** is typically a free-running counter (i.e., a special function register that simply counts upwards at a specific counting rate). When the counter overflows (i.e., the value wraps back around to zero), we can configure the microcontroller to generate an **interrupt request**. The 68HC12 implements multiple configurable timers, allowing the generation of complex sequences of timing.

7. **Pulse Accumulator** – The reverse of a timer is also implemented; the pulse accumulator acts as an automatic counter for incoming electrical pulses. It can be configured to count to a certain level, to automatically trigger an interrupt after overflow of the counting register occurs, or on an incoming pulse 'edge.'

8. **SPI Transfer Complete** – One interrupt is for serial communication, which we will look at in the lab on communication protocols. It is generated automatically when an SPI transfer is complete, allowing the user program to avoid busy waiting while data is being sent between devices using the SPI protocol.

9. **A to D Converter Sample** – As with the example in the lecture slides, the 68HC12 can be configured to automatically generate an interrupt when an analog to digital conversion is complete. This is very useful for implementing control loops and similar structures, as you do not need to manually wait for samples to be ready.

10. **Reserved Space** – The 68HC12 has many variants, each of which may implement different added features. For instance, some implement additional communication protocols (such as I2C, USB, etc.). Others may support specific input or output devices (motor drivers, sensors, etc.) directly. These variants have the ability to implement their own unique interrupt sources using this reserved space.

When an interrupt is being handled by the 68HC12, it **automatically** disables interrupts **and** protects registers and the CCR, as per what we saw in lecture. The stack contents immediately at the start of an ISR will thus be as follows:

Free Stack Space		
[CCR]	←	SP
[A]	←	SP+1
[B]	←	SP+2
[X]	←	SP+3
[Y]	←	SP+5
Return Address	←	SP+7

So, if you were writing an assembly-language ISR, much of the work is done for you. The downside of this implementation is that the 68HC12 introduces significant overhead for any ISR, even if the user code itself is relatively short. If your ISR does not use registers B, X, and Y, for example, the time used to protect these registers is wasted. Thus, your goal when using the 68HC12 in particular should be to **balance** ISR use – interrupts are more efficient for implementing **urgent but relatively infrequent** tasks. **Regular** but urgent tasks *might* be better implemented using a normal loop.

Note that once the ISR is complete, a return from interrupt (RTI) instruction must be executed. This instruction will (if no other interrupts are pending) automatically POP all protected registers and return to the program that was executing. If additional interrupts were generated during the ISR, or if multiple interrupts simultaneously arrived earlier, then another ISR will be automatically called, starting the interrupt handling process again. When multiple sources of interrupts are pending, they are handled in the **fixed** priority order indicated in the earlier table. Notice that, as discussed in lecture, if interrupt requests are consistently raised faster than they are handled, under this implementation it is possible for the microcontroller to **never** return to executing the original program.

Basic Timer Operation

We will be writing a simple program in this lab using the **timer** and the LCD control routines that we created last lab. The timer will be used to sequentially count upward, and the result will be displayed on two lines of the LCD screen. We will use this program to investigate **race conditions** and shared variable access.

The basic functionality of a timer is that of a free-running counter; when enabled this counter is **automatically** incremented **without** the active execution of any instructions by the microcontroller. In other words, it does not consume processing cycles to count. Depending on the implementation, the timer may be configured to count in a number of different patterns. It may also be configured to generate an interrupt when certain events occur, such as **overflow** of the timer count. We will use the timer overflow interrupt to automatically increment two shared variables.

Interrupt Masking

Recall that one of the most basic requirements for implementing interrupts correctly in any system is the ability to **mask interrupts**. For the 68HC12, interrupts are automatically masked during an ISR. However, when accessing shared data in other parts of our program (such as the 'main' routine) we must define a **critical section** to protect data access. To do so we require the ability to disable any interrupts that may access shared data.

On some microcontrollers, simple (global) interrupt masking is available through distinct commands such as CLI / SEI (clear / set interrupt mask). As we saw earlier, for the 68HC12 in particular, interrupt masking is coarse and is controlled by just two bits in the CCR. We will make judicious use of masking in our program to ensure that the boot-loader can still

maintain communication with the computer – otherwise, you would have to reset the power to the board every time you made a coding mistake!

Complete Timer Operation

We will be making use of the TOF (Timer OverFlow) interrupt service routine in our program. The 68HC12 implements multiple timers, each of which count independently and have separate interrupt sources (Channel 0 through 7). All timers share a common overflow interrupt, which is the simplest source for us to use here.

Timer operation on the 68HC12 is controlled by a number of SFRs. The most basic is the timer **count** register, TCNT. When the timer is enabled, a 24 MHz clock signal is connected to a 16-bit counter. This counter will increment on each pulse from the clock signal, starting from 0x0000 and counting to 0xFFFF before overflowing and rolling back to 0x0000. This means that by default, a timer overflow will occur every 2.7307 ms:

$$T_{overflow} = \frac{65536\ counts}{24{,}000{,}000\ counts/s} = 2.7307\ ms$$

Note that we can read TCNT at any time, including inside an ISR, but we cannot write to this SFR. The behaviour of the counter, including what interrupts are triggered by it, is governed by additional SFRs. The first is called 'Timer System Control Register 1' or TSCR1:

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
Read:	TEN	TSWAI	TSFRZ	TFFCA	PRNT	0	0	0
Write:						Do not write / reserved		
On Reset:	0	0	0	0	0	0	0	0

The function of this register is as follows:

TEN – Enables the timer system when set to '1'. Can be turned off to save power; default state is off (see the 'on reset' row).
TSWAI – Determines if timer stops while in 'wait mode' – a '1' disables the timer when the 68HC12 is in wait mode.
TSFRZ – Determines if the timer stops while in 'freeze mode' – a '1' disables the timer in this mode.
TFFCA – Functionality related to the timer fast flag clear bit; leave it as '0' for this lab.

Given the above, we can now enable the timer system by setting TEN (Bit 7) of TSCR1 to '1.' However, we will also need additional control – the second control register, TSCR2, implements additional functionality.

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
Read: **Write:**	TOI	0	0	0	TCRE	PR2	PR1	PR0
On Reset:	0	0	0	0	0	0	0	0

This second SFR primarily controls **pre-scaling and overflow** functions:

TOI – Timer overflow interrupt enable bit; set to '1' to generate an interrupt when a timer overflow occurs. This will obviously have to be set to one for an interrupt to be generated, along with TEN. However, notice that we can allow the timer to continue to run (TEN = 1) while **not** generating interrupts (TOI = 0). This is useful in that it allows us to circumvent the limitations of interrupt source masking on the 68HC12 – we can disable **just** those interrupts generated by timer overflow, if we so desire.

TCRE – Timer counter reset enable bit; set to '0' to have counter run free (normal mode), or set to '1' to have the counter reset to zero on a successful **output compare**. This mode allows us to have the counter count until its output **matches** a given input; this is extremely useful in that we can use it to construct many types of **analog to digital converters**.

Pre-scaling functionality is controlled by the three bits PR0-PR2. The idea behind pre-scaling is that the timer is limited in count (16-bit width for a count of 0xFFFF) before an overflow is generated. When driven by a $24\,MHz$ clock, the maximum time between overflow events is $\sim 2.7\,ms$. If we want to create a delay longer than this time, we **pre-scale** the clock; in other words, if we pre-scale by a factor of P, then every P cycles of the $24\,MHz$ clock increments the counter by only 1. The result is a longer timer between overflows:

PR2	PR1	PR0	Pre-scale Factor	Input Clock	Effective Timer Clock	Maximum Time Delay
0	0	0	1	24 MHz	24 MHz	2.7 ms
0	0	1	2	24 MHz	12 MHz	5.5 ms
0	1	0	4	24 MHz	6 MHz	10.9 ms
0	1	1	8	24 MHz	3 MHz	21.8 ms
1	0	0	16	24 MHz	1.5 MHz	43.7 ms
1	0	1	32	24 MHz	0.75 MHz	87.4 ms
1	1	0	64	24 MHz	0.375 MHz	174.8 ms
1	1	1	128	24 MHz	0.1875 MHz	349.5 ms

To create delays longer than $\sim 350\,ms$, one would need to **manually** count the number of times the timer overflows. For the remainder of this lab, we will leave the timer at its default pre-scaling factor of one. You should keep this functionality in mind for your projects involving the 68HC12, however.

We are now close to being able to use the basic timer overflow functionality. The last step is to understand the mechanism behind timer overflow interrupt handling. When the timer overflows, a specific bit in a status SFR is set: Bit 7 of the Timer Flag Register 2 (**TFLG2**).

Side Note: TFLG2 is actually only one bit; all others are zero and have no function.

If we were not using interrupts, we could actually manually poll (constantly check) for when this Bit 7 of TFLG2 becomes one – this would indicate that the timer has recently overflowed. When using interrupts, however, an interrupt request is automatically generated upon an overflow. In this case, we still must **manually clear** this bit to allow subsequent overflow interrupt requests to be generated. Thus, the steps to **starting** a basic timer are as follows:

1. Enable the timer system by setting TEN = 1 in TSCR1.
2. Enable the timer overflow interrupt source by setting TOI = 1 in TSCR2.
3. Clear the overflow status flag in TFLG2 by setting BIT7 of TFLG2 equal to one.

To keep the timer running (and generating interrupts) we must do the following **on every overflow**:

1. Clear the overflow status flag in TFLG2 by setting BIT7 of TFLG2 equal to one.

We now have the basic steps required to create a timed event which occurs every 2.7 ms. We will use this knowledge to complete Part 1 of the lab procedure.

Lab – Part 1: Critical Sections and Shared Data

Step 1. We will be using the LCD library from the previous lab to make our results more visible. To make this library more user-friendly, a few useful functions have been added:

void printLCDText(char *str) – Prints a string of text to the current line of the LCD. If there is a '\n' character, the function automatically moves down to the next LCD line. All strings **must** end with a '$' character.
void printLCDNumber(int num) – Prints an integer number at the current LCD position.
void clearLCD() – Clears the LCD screen.

Begin by loading the 'Lab5_1.mcp' project. Download and run the program on the board; you will notice by examining 'main.c' that the program implements a simple loop. There are two global variables ('var1' and 'var2') which will later be **shared** with the ISR. For now, these variables are incremented within the main loop, and the current count is displayed every 500ms or so on the LCD. You should notice that the count **remains the same at all times** for this version of the program.

Step 2. Close the previous project and open 'Lab5_2.mcp'. Open 'main.c' and observe the changes. First, you may notice that there are three lines initializing the timer as an interrupt source and enabling it. You may also notice that the shared variables 'var1' and 'var2' are now incremented in a new subroutine called TOF_ISR – this is **our** timer overflow ISR. Notice the 'interrupt' type and 'VectorNumber_Vtimovf' label included in the function declaration, which together indicate to the compiler that this is our implementation of the TOF ISR. Lastly, you may notice that there is no function prototype before main( ) for TOF_ISR; all ISRs have fixed prototypes to allow the compiler to directly find them if they are located in your code.

Step 3. Download and run the 'Lab5_2' project. Answer **Question 1** and **Question 2** in the deliverable. To clarify what is happening, you can try adding a shortWait(500) to the middle of the LCD print functions (see the code below). This last step is not necessary to complete the questions, however.

```
printLCDText("Var 1: $");
printLCDNumber(var1);
printLCDText("\n$");
shortWait(500);
printLCDText("Var 2: $");
printLCDNumber(var2);
shortWait(500);
```

Step 4. Modify the code from Step 3 to solve the problem you identified in Question 2. Remember that interrupts are already automatically disabled inside the ISR. However, even in main( ) you should be careful – **do not** disable/enable all interrupts; you should work only with the TOF interrupt. Also, keep in mind that any **critical sections** should be as short as possible. Lastly, recall that the 68HC12 only has CLI/SEI instructions available for interrupt masking, which enable/disable **all** interrupts. You may want to think of an alternate solution which affects **only** TOF for best results. Write your modified code for **Question 3**.

Lab – Part 2: External Interrupts and Masking

For the second half of the lab, we will be testing the use of external interrupts. An external interrupt can be triggered in a variety of ways; we will re-visit this topic later for input and output. We will be using Port P as an external input; the 68HC12 allows interrupts to externally trigger from multiple pins on Port P. For these pins, we have a choice of two triggering modes, **edge and level triggering**.

Edge-Triggered Interrupt – In edge-triggered mode, the interrupt pins are nominally held electrically low by a set of **pull-down** resistors. An IRQ is then generated whenever there is a transition from **low to high**, but is generated **once and only once** for every high→low transition.

Level-Triggered Interrupt – In level-triggered mode, the external interrupt pins are nominally at a low ('1') level. If an external source pulls them high for a period of time, an interrupt request is generated.

Please note that most of what follows can be duplicated for other ports (such as Port J, Port H, etc.). This means that external interrupts can actually be triggered from changes on many pins due to the wiring of the 68HC12 (internal) and the evaluation board (external). This can be useful, given that having only one external interrupt pin could otherwise be very limiting. In the case of multiple ports being used, you would need to add code to the ISR to determine the actual external source, however.

Step 5. We will begin by testing edge-triggered functionality. Close the previous project and open 'Lab5_3.mcp.' Open 'main.c' and look at the initialization code that has been added.

Note that this particular evaluation board has a minimum number of peripherals connected to Port P; it is connected only to the 7-segment LED displays and the motor drivers, which we can ignore for the following steps.

1. [DDRP = 0b1111100;] – First we set the **data direction register** for Port P, setting Bit0 and Bit1 to zero, making PP0 and PP1 inputs.

2. [PERP = 0b00000011;] – This step sets the **pull-down register**, indicating (through Bit0 = Bit1 = '1') that the pull-down resistors for PP0 and PP1 should be enabled. This means that PP0 and PP1 are electrically connected through a small resistance to ground, making the nominal input (i.e., with nothing else connected) zero.

3. [PPSP_PPSP0 = 1; PPSP_PPSP1 = 1;] – This step tells the 68HC12 to generate an external interrupt request on any **rising edge** (an electrical low to high transition) on Port P pins PP0 and PP1.

4. [PIFP = 0b00000011;] – This step clears the interrupt **flag** for PP0 and PP1. Like the timer, the bits of PIFP (the **flag register for Port P**) are set to one when a change is detected; if interrupts are enabled this also triggers the interrupt request. We can manually examine this register to detect changes in Port P, as well. Also like the timer, we must clear this flag whenever an event is detected (i.e., inside the interrupt and once at initialization). Note that you can check individual bits of PIFP by using PIFP_PIFP0, PIFP_PIFP1, etc. This allows you to check the **source** of a Port P interrupt inside the ISR.

5. [PIEP = 0b00000011;] – This sets the enable register for Port P interrupts; the ones at Bit0 and Bit1 enable an external interrupt to be generated from changes in PP0 and PP1.

Step 6. Download, compile, and run your code. The LCD will start counting upwards. With the program running, carefully connect a wire from PP0 (upper left connecter near breadboard) to +5V (top left connecter near breadboard). You should notice the count **stop**. What we have done by connecting the wire is electrically connect PP0 to +5 volts, creating a **low-to-high** transition – an **edge** transition. This triggers the ISR, stopping the count.

Remove the wire. You should notice that nothing happens; the interrupt is only triggered on a **low to high** transition, not a **high to low**. Reconnect the wire again – this should **restart** the count.

Try this process **a few times**. You will eventually (or possibly immediately) notice that sometimes the count fails to stop or restart. Think about what is happening – we are **physically** connecting the +5V to the external pin, but on the time scale of the microcontroller we are doing so **very** slowly. Think about what is **mechanically** occurring, and remember this line of thought when later answering Question 5.

Step 7. Modify the code such that PP0 toggles between the count starting and stopping. Have PP1 **manually** increment the count of var1 and var2 by one. Obviously, you will only be able to test this second functionality when the count is stopped. When testing the 'increment by one' function, you may actually notice the count increments by more than one – this issue is related to what you encountered in Step 6. This will depend on how you have written your code, however. Once you have tested your modified code, and record it for **Question 4**. Using the results of your testing and what you observed in Step 6, answer **Question 5**.

Step 8. To solve the problem we encountered earlier, we will use a different external interrupt pin (which can be level triggered) to handle the counting. Close your previous project and open 'Lab5_4.mcp'. This project adds a second interrupt source using the partially-maskable **external IRQ** source we mentioned earlier. This interrupt source is much simpler to enable than the I/O port external interrupts. For initialization, we modify only one SFR – the external interrupt control register, **INTCR**. It has two bits of interest, IREQ (IRQ edge or level select) and IREQN (IRQ enable). Notice that this project has two added lines of initialization.

1. [INTCR_IRQE= 0;] – Sets IRQE bit equal to zero, causing the external IRQ line to be **level-triggered** rather than edge triggered. In this mode, an interrupt will be triggered as soon as the microcontroller 'notices' the level of the IRQ line is low. Note that the microcontroller will **continuously generate interrupt requests** as long as the external line remains low.

2. [INTCR_IRQEN= 0;] – Enables external interrupts.

Test the new functionality by first pausing the count by connecting PP0 to +5V with a wire; this pin's function remains edge-triggered, so you may experience the same problem as before.

With the count **paused**, connect a wire from **PE1** to **GND**. Note that this pin is shared; it is also the external IRQ pin. The PE1 pin is located on the lower-right connector near the breadboard. With the wire connected, you will notice that the displayed count still does not change.

Remove the wire; you will notice the count has changed. Note that the value may seemingly be random if you left the wire connected for a long time; repeat the step a few times with the wire connected for varying amounts of time. Answer **Question 6** and **Question 7**.

This marks the end of Lab 5. If you finish early, you may want to try fixing the above problem completely as a practice exercise. This is not required for hand-in. Note that there are valid ways to solve both the earlier bounce/edge-trigger problem and this new counting/deadlock problem – we will learn about such solutions in our next lecture on input and output.

**** End of Lab 5 ****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 5

Deliverable

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

1. Describe the behavior of the code. In particular, are the numbers equal when displayed?

2. What code issue is causing the behavior you noticed in Question 1?

3. Modified code listing:

4. Modified ISR Code:

5. Speculate on why connecting the wire does not always generate the expected result. (**Hint**: You may want to read about **switch bounce**, a related issue).

6. If you vary how long you leave the wire connected, what happens? Why is this so?

7. Why does the display not update when the wire is connected?