- **Setting and clearing bits in C**
- **Using pointers in C**
  - Program to count the number of negative numbers in an area of memory

- **Introduction to the MC9S12 Hardware Subsystems**
  - The MC9S12 timer subsystem

Operators in C

Operator	Action	example
|	Bitwise OR	%00001010 | %01011111 = % 01011111
&	Bitwise AND	%00001010 & %01011111 = % 00001010
^	Bitwise XOR	%00001010 ^ %01011111 = % 01010101
~	Bitwise COMP	~%00000101 = %11111010
%	Modulo	10 % 8 = 2
||	Logical OR	%00000000 || %00100000 = 1
&&	Logical AND	%11000000 && %00000011 = 1
		%11000000 && %00000000 = 0

Setting and Clearing Bits in C

Assembly	C	action
bset DDRB,$0F	DDRB = DDRB | 0x0f;	Set 4 LSB of DDRB
bclr DDRB,$F0	DDRB = DDRB & ~0xf0;	Clear 4 MSB of DDRB
l1: brset PTB,$01,l1	while ((PTB & 0x01) == 0x01)	Wait until bit clear
l2: brclr PTB,$02,l2	while ((PTB & 0x02) == 0x00)	Wait until bit set

Pointers in C

To read a byte from memory location 0xE000:

```
var = *(char *) 0xE000;
```

To write a 16-bit word to memory location 0xE002:

```
*(int *) 0xE002 = var;
```

Program to count the number of negative numbers in an array in memory

```
/* Program to count the number of negative numbers in memory *
   Start at 0xE000, go through 0xEFFF
   Treat the numbers as 8-bit
*/
#include <hidef.h>          /* common defines and macros */
#include "derivative.h"     /* derivative-specific definitions */

unsigned short num_neg;     /* Make num_neg global so we can */
                            /* find it in memory use type int so */
                            /* can hold value larger than 256 */
                            /* Unsigned because number cannot */
                            /* be negative */
main()
{
        char *ptr,*start,*end;

        start = *(char *) 0xE000;     /* Address of first element */
        end = *(char *) 0xEFFF;       /* Address of last element */

        num_neg = 0;

        for (ptr = start; ptr <= end; ptr = ptr+1)
        {
                if (*ptr < 0) num_neg = num_neg + 1;
        }
        _ _asm(swi);                  /* Exit to DBug-12 */
}
```

Hello, World!

• Here is the standard ”hello, world” program:

```
#include <stdio.h>
main()
{
        printf("hello, worl!d\r\n");
}
```

• To write the ”hello, world” program, you need to use the **printf()** function.

• The printf() function is normally a library function

• In CodeWarrior, you can access printf() by doing the following:

1. In your C program, add the following lines:

```
#include <stdio.h>
#include <termio.h>
```

2. In CodeWarrior, select Project, Add Files, and select the file **termio.c**. This is in the CodeWarrior library, which is in the following location on my computer:

c:\Program Files\Freescale\CWS12v5.1\lib\hc12c\src

Your C program will look like this:

```
#include <stdio.h>
#include <termio.h>
main()
{
        printf("hello, world!\r\n");
        _ _asm(swi);
}
```

• The above program is about 1,500 bytes long.

• To load the program into your Dragon12 board, you will have to edit the .s19 file and remove the first line which starts with a S0. The last line of the .s19 file looks this:

```
S105FFFE2029B4
```

This tells the loader to put the 16-bit number 2029 into address FFFE. The address FFFE is in flash EEPROM, and the loader will not be able to write there. You can leave the line in, in which case DBug12 will give you the warning "Can't Write Target Memory", or you can remove this line from the .s19 file to avoid getting the warning. (If you could put 2029 into address FFFE, your MC9S12 would start executing the code at address 0x2029 when you reset the board.)

• You can print out variables as well. Here is an example:

```
#include <stdio.h>
#include <termio.h>
main()
{
        int i;
        for (i=0;i<100;i++) printf("i = %d\r\n",i);
        _ _asm(swi);
}
```

MC9S12 Built-In Hardware

• The MC9S12 has a number of useful pieces of hardware built into the chip.

• Different versions of the 9S12 have slightly different pieces of hardware. Information about the hardware modules is found in data sheet for the modules.

• We are using the MC9S12DP256 chip.

• Here is some of the hardware available on the MC9S12DP256:

– **General Purpose Input/Output (GPIO) Pins**: These pins can be used to read the logic level on a MC9S12 pin (input) or write a logic level to an MC9S12 pin (output). We have already seen examples of this – **PORTA** and **PORTB**. Each GPIO pin has an associated bit in a data direction register which you use to tell the MC9S12 if you want to use the GPIO pin as input or output. (For example, **DDRA** is the data direction register for **PORTA**.)

- **Timer-Counter Pins**: The MC9S12 is often used to time or count events. For example, to use the MC9S12 in a speedometer circuit you need to determine the time it takes for a wheel to make one revolution.

  - To keep track of the number of people passing through a turnstile you need to count the number of times the turnstile is used.
  - To control the ignition system of an automobile you need to make a particular spark plug fire at a particular time. The MC9S12 has hardware built in to do these tasks.

* For information, see the **ECT_16B8C Block User Guide**.

– **Pulse Width Modulation (PWM) Pins**: To make a motor turn at a particular speed you need to send it a pulse width modulated signal. This is a signal at a particular frequency (which differs for different motors), which is high for part of the period and low for the rest of the period. To have the motor turn slowly, the signal might be high for 10% of the time and low for 90% of the time. To have the motor turn fast, the signal might be high for 90% of the time and low for 10% of the time.

* For information, see the **PWM_8B8C Block User Guide**.

– **Serial Interfaces**: It is often convenient to talk to other digital devices (such as another computer) over a serial interface. When you connect your MC9S12 to the PC in the lab, the MC9S12 talks to the PC over a serial interface. The MC9S12 has two serial interfaces: an asynchronous serial interface (called the Serial Communications Interface, or SCI) and a synchronous serial interface (called the Serial Peripheral Interface, or SPI).

* For information on the SCI, see the **9S12 Serial Communications Interface (SCI) Block User Guide**.

* For information on the SPI, see the **SPI Block User Guide**.

– **Analog-to-Digital Converter (ADC)**: Sometimes it is useful to convert a voltage to a digital number for use by the MC9S12. For example, a temperature sensor may put out a voltage proportional to the temperature. By converting the voltage to a digital number, you can use the MC9S12 to determine the temperature.

* For information, see the **ATD_10B8C Block User Guide**.

• Most of the MC9S12 pins serve dual purposes. For example, **PORTT** is used for the timer/counter functions. If you do not need to use PORTT for timer/counter functions, you can use the pins of PORTT for GPIO. There are registers which allow you to set up the PORTT pins to use as GPIO, or to use as timer/counter functions. (These are called the **Timer Control Registers**).

Introduction to the MC9S12 Timer Subsystem

• The MC9S12 has a **16-bit counter** that normally runs with a **24 MHz clock**.

• Complete information on the MC9S12 timer subsystem can be found in the ECT_16B8C Block User Guide. **ECT** stands for **Enhanced Capture Timer**.

• When you reset the MC9S12, the clock to the timer subsystem is initially turned off to save power.

– To turn on the clock you need to write a 1 to Bit 7 of register **TSCR1** (Timer System Control Register 1) at address **0x0046**.

• The clock starts at **0x0000**, counts up (0x0001, 0x0002, etc.) until it gets to **0xFFFF**. It rolls over from 0xFFFF to 0x0000, and continues counting forever (until you turn the counter off or reset the MC9S12).

• It takes **2.7307 ms** (65,536 counts/24,000,000 counts/sec) for the counter to count from 0x0000 to 0xFFFF and roll over to 0x0000.

• To determine the time an event happens, you can read the value of the clock (by reading the 16-bit TCNT (Timer Count Register) at address **0x0044 (page 28 of MC9S12DP256B)**.

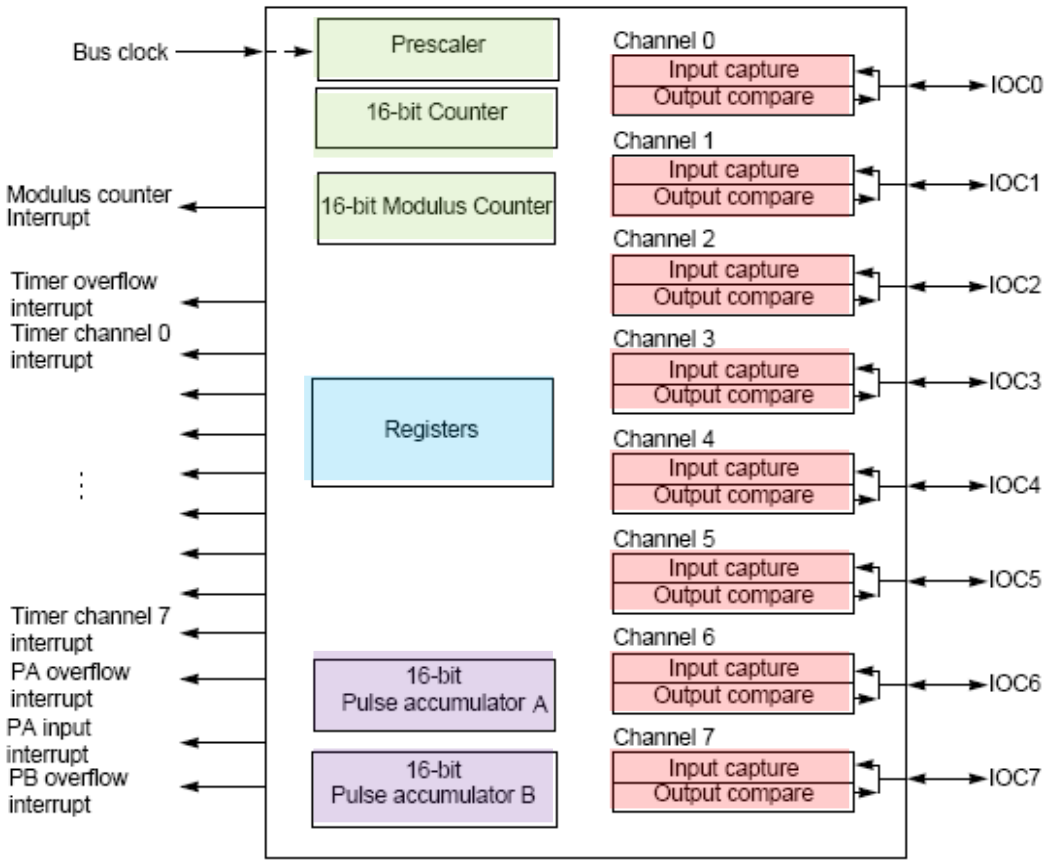


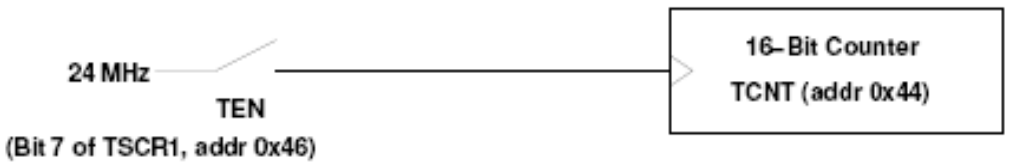
Figure 1-1 Timer Block Diagram

Timer inside the MC9S12:

When you enable the timer (by writing a 1 to bit 7 of TSCR1), you connect a 24−MHz oscillator to a 16−bit counter.

You can read the counter at address **TCNT**.

The counter will start at 0, will count to 0xFFFF, then will roll over to 0x0000. It will take 2.7307 ms for this to happen.



To enable timer on MC9S12, set Bit 7 of register TCSR1:

```
bset    TSCR1,#$80          TSCR1 = TSCR1 | 0x80;
```

Why using the OR operator and not TSCR1=0x80?

3.3.6 TSCR1 — Timer System Control Register 1

Register offset: $_06

	BIT7	6	5	4	3	2	1	BIT0
R	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-6 Timer System Control Register 1 (TSCR1)

Read or write anytime.

TEN — Timer Enable
0 = Disables the main timer, including the counter. Can be used for reducing power consumption.
1 = Allows the timer to function normally.

If for any reason the timer is not active, there is no ÷64 clock for the pulse accumulator since the ÷64 is generated by the timer prescaler.

TSWAI — Timer Module Stops While in Wait
0 = Allows the timer module to continue running during wait.
1 = Disables the timer module when the MCU is in the wait mode. Timer interrupts cannot be used to get the MCU out of wait.

TSWAI also affects pulse accumulators and modulus down counters.

TSFRZ — Timer and Modulus Counter Stop While in Freeze Mode
0 = Allows the timer and modulus counter to continue running while in freeze mode.
1 = Disables the timer and modulus counter whenever the MCU is in freeze mode. This is useful for emulation.

TSFRZ does not stop the pulse accumulator.

TFFCA — Timer Fast Flag Clear All
0 = Allows the timer flag clearing to function normally.
1 = For TFLG1($0E), a read from an input capture or a write to the output compare channel ($10–$1F) causes the corresponding channel flag, CnF, to be cleared. For TFLG2 ($0F), any access to the TCNT register ($04, $05) clears the TOF flag. Any access to the PACN3 and PACN2 registers ($22, $23) clears the PAOVF and PAIF flags in the PAFLG register ($21). Any access to the PACN1 and PACN0 registers ($24, $25) clears the PBOVF flag in the PBFLG register ($31). This has the advantage of eliminating software overhead in a separate clear sequence. Extra care is required to avoid accidental flag clearing due to unintended accesses.

• To put in a delay of 2.7307 ms, you could wait from one reading of 0x0000 to the next reading of 0x0000.

• **Problem:** You cannot read the TCNT register quickly enough to make sure you will see the 0x0000.

To put in a delay for 2.7307 ms, could watch timer until TCNT == 0x0000:

```
      bset  TSCR1,#$80          TSCR1 = TSCR1 | 0x80;
l1:   ldd   TCNT        [3]     while (TCNT != 0x0000) ;
      bne l1            [3/1]
```

Problem: You might see 0xFFFF and 0x0001, and miss 0x0000

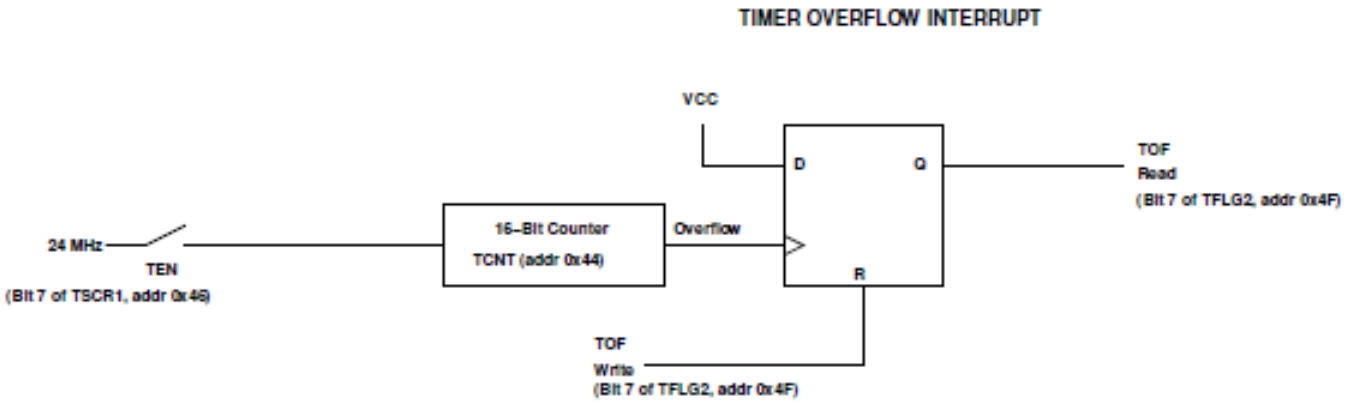
• **Solution:** The MC9S12 has built-in hardware which will set a flip-flop every time the counter rolls over from 0xFFFF to 0x0000.

• To wait for 2.7307 ms, just wait until the flip-flop is set, then clear the flip-flop, and wait until the next time the flip-flop is set.

• You can find the state of the flip-flop by looking at bit 7 (the **Timer Overflow Flag** (**TOF**) bit) of the Timer Flag Register 2 (**TFLG2**) register at address **0x004F**.

• You can clear the flip-flop by writing a 1 to the TOF bit of TFLG2.

Solution: When timer overflows, it latches a 1 into a flip−flop. Now when timer overflows (goes from 0xFFFF to 0x0000), **Bit 7 of TFLG2 register is set to one**. **Can clear register by writing a 1** to Bit 7 of TFLG register.




```
    bset TSCR1,#$80      ; Enable timer
l1: brclr TFLG2,#$80,l1 ; Wait for Bit 7 of TFLG2 is set
    ldaa #$80

    …
    program …
    …

    staa TFGL2           ; Clear TOF flag
```

```
TSCR1 = TSCR1 | 0x80;          //Enable timer
while ((TFLG2 & 0x80) == 0) ; // Wait for TOF

…
program …
…

TFLG2 = 0x80;                  // Clear TOF
```

3.3.13 TFLG2 — Main Timer Interrupt Flag 2

Register offset: $_0F

	BIT7	6	5	4	3	2	1	BIT0
R	TOF	0	0	0	0	0	0	0
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-13 Main Timer Interrupt Flag 2 (TFLG2)

TFLG2 indicates when interrupt conditions have occurred. To clear a bit in the flag register, write the bit to one.

Read anytime. Write used in clearing mechanism (set bits cause corresponding bits to be cleared).

Any access to TCNT will clear TFLG2 register if the TFFCA bit in TSCR register is set.

TOF — Timer Overflow Flag

Set when 16-bit free-running timer overflows from $FFFF to $0000. This bit is cleared automatically by a write to the TFLG2 register with bit 7 set. (See also TCRE control bit explanation.)

• **Another problem:** Sometimes you may want to delay longer than 2.7307 ms, or time an event which takes longer than 2.7307 ms. This is hard to do if the counter rolls over every 2.7307 ms.

• **Solution:** The MC9S12 allows you to slow down the clock which drives the counter.

• You can **slow down the clock by dividing the 24 MHz clock** by 2, 4, 8, 16, 32, 64 or 128.

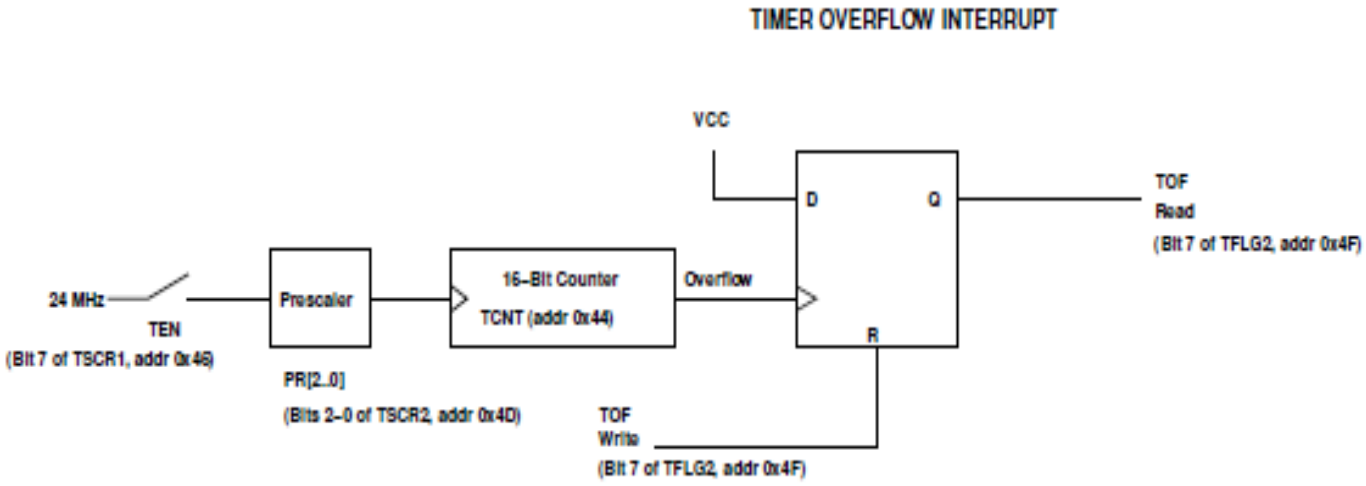
• You do this by writing to the prescaler bits (**PR2:0**) of the **Timer System Control Register 2** (**TSCR2**) Register at address **0x004D**.

2.7307 ms will be too short if you want to see lights flash. You can slow down clock by dividing it before you send it to the 16−bit counter. By setting prescaler bits **PR2,PR1,PR0** of TSCR2 you can slow down the clock:

PR	Divide	Freq	Overflow Rate
000	1	24 MHz	2.7307 ms
001	2	12 MHz	5.4613 ms
010	4	6 MHz	10.9227 ms
011	8	3 MHz	21.8453 ms
100	16	1.5 MHz	43.6907 ms
101	32	0.75 MHz	87.3813 ms
110	64	0.375 MHz	174.7627 ms
111	128	0.1875 MHz	349.5253 ms

To set up timer so it will overflow every 87.3813 ms:

```
bset TSCR1,#$80          TSCR1 = TSCR1 | 0x80;
ldaa #$05                TSCR2 = 0x05;
staa TSCR2
```



3.3.10 TIE — Timer Interrupt Enable Register

Register offset: $_0C

	BIT7	6	5	4	3	2	1	BIT0
R W	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
RESET:	0	0	0	0	0	0	0	0

Figure 3-10 Timer Interrupt Enable Register (TIE)

Read or write anytime.

The bits in TIE correspond bit-for-bit with the bits in the TFLG1 status register. If cleared, the corresponding flag is disabled from causing a hardware interrupt. If set, the corresponding flag is enabled to cause a interrupt.

C7I–C0I — Input Capture/Output Compare "x" Interrupt Enable

3.3.11 TSCR2 — Timer System Control Register 2

Register offset: $_0D

	BIT7	6	5	4	3	2	1	BIT0
R	TOI	0	0	0	TCRE	PR2	PR1	PR0
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-11 Timer System Control Register 2 (TSCR2)

Read or write anytime.

TOI — Timer Overflow Interrupt Enable
0 = Interrupt inhibited
1 = Hardware interrupt requested when TOF flag set

TCRE — Timer Counter Reset Enable

This bit allows the timer counter to be reset by a successful output compare 7 event. This mode of operation is similar to an up-counting modulus counter.
0 = Counter reset inhibited and counter free runs
1 = Counter reset by a successful output compare 7

If TC7 = $0000 and TCRE = 1, TCNT will stay at $0000 continuously. If TC7 = $FFFF and TCRE = 1, TOF will never be set when TCNT is reset from $FFFF to $0000.

PR2, PR1, PR0 — Timer Prescaler Select

ECT_16B8C Block User Guide V01.03

These three bits specify the number of ÷2 stages that are to be inserted between the bus clock and the main timer counter.

Table 3-4 Prescaler Selection

PR2	PR1	PR0	Prescale Factor
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

The newly selected prescale factor will not take effect until the next synchronized edge where all prescale counter stages equal zero.

3.3.12 TFLG1 — Main Timer Interrupt Flag 1

Register offset: $_0E

	BIT7	6	5	4	3	2	1	BIT0
R W	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
RESET:	0	0	0	0	0	0	0	0

Figure 3-12 Main Timer Interrupt Flag 1 (TFLG1)

TFLG1 indicates when interrupt conditions have occurred. To clear a bit in the flag register, write a one to the bit.

TFLG1 indicates when interrupt conditions have occurred. To clear a bit in the flag register, write a one to the bit.

Use of the TFMOD bit in the ICSYS register ($2B) in conjunction with the use of the ICOVW register ($2A) allows a timer interrupt to be generated after capturing two values in the capture and holding registers instead of generating an interrupt for every capture.

Read anytime. Write used in the clearing mechanism (set bits cause corresponding bits to be cleared). Writing a zero will not affect current status of the bit.

When TFFCA bit in TSCR register is set, a read from an input capture or a write into an output compare channel ($10–$1F) will cause the corresponding channel flag CnF to be cleared.

C7F–C0F — Input Capture/Output Compare Channel “n” Flag.

C0F can also be set by 16 - bit Pulse Accumulator B (PACB). C3F - C0F can also be set by 8 - bit pulse accumulators PAC3 - PAC0.

Setting and Clearing Bits in C

• To put a specific number into a memory location or register (e.g., to **put 0x55 into PORTA**):

movb #$55,PORTA **PORTA = 0x55;**

• To set a particular bit of a register (e.g., **set Bit 4 of PORTA**) while leaving the other bits unchanged do a bitwise OR of the register and a mask which has a 1 in the bit(s) you want to set, and a 0 in the other bits:

bset PORTA,#$10 **PORTA = PORTA | 0x10;**

• To clear a particular bit of a register (e.g., **clear Bit 5 of PORTA**) while leaving the other bits unchanged do a bitwise AND of the register and a mask which has a 0 in the bit(s) you want to clear, and a 1 in the other bits. You can construct this mask by complementing a mask which has a 1 in the bit(s) you want to set, and a 0 in the other bits:

bclr PORTA,#$20 **PORTA = PORTA & 0xDF;**

PORTA = PORTA & ~0x20;

Using ~0x20 is better than using 0xDF because it is less likely that you will make a mistake when complementing 0x20 in your head.

• To change several bits of a register, AND the register with 1's in the bits you want to leave unchanged, then OR the result with 1's in the bits you want to set, and 0's in the bits you want to clear. For example, **to set bits 2 and 0, and clear bit 1** (write 101 to bits 2-0) **of TSCR2**, do the following:

```
bclr TSCR2,#$02          TSCR2 = TSCR2 & ~0x02;
bset  TSCR2,#05          TSCR2 = TSCR2  | 0x05;
```

or

```
TSCR2 = (TSCR2 & ~0x02) | 0x05;
```

• Write to all bits of a register when you know what all bits should be, such as when you initialize it. Set or clear bits when you want to change only one or a few bits and leave the others unchanged.

C Program to implement a delay

```
#include <hidef.h>
#include "derivative.h"

void delay(void);

main()
{
        TSCR1 = TSCR1 | 0x80;     /* Enable timer subsystem */
        TSCR2 = 0x05;             /* Set overflow time to 87 ms */
        TFLG2 = 0x80;             /* Make sure TOF bit clear */
        while (1) {
                PORTB = PORTB + 1;
                delay();
        }
}

void delay(void)
{
        while ((TFLG2 & 0x80) == 0x00) ;      /* Wait for timer */
                                              /* overflow */
        TFLG2 = 0x80;                         /* Clear TOF bit */
}
```

• **Problem**: Cannot do anything while waiting

• **Solution**: Interrupt – can do other things, and hardware will signal processor when overflow occurs

• Need to understand how processor handles exceptions – resets and interrupts

• Start by looking at what happens when the MC9S12 is reset

What Happens When You Reset the MC9S12?

• What happens to the MC9S12 when you turn on power or push the reset button?

• How does the MC9S12 know which instruction to execute first?

• On reset the MC9S12 loads the PC with the address located at address **0xFFFE** and **0xFFFF**.

• Here is what is in the memory of our MC9S12:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
FFF0	F6	EC	F6	F0	F6	F4	F6	F8	F6	FC	F7	00	F7	04	F0	00

• On reset or power-up, the first instruction your MC9S12 will execute is the one located at address **0xF000**.

Introduction to Interrupts

Can implement a delay by waiting for the TOF flag to become set:

```
void delay(void)
{
        while ((TFLG2 & 0x80) == 0) ;
        TFLG2 = 0x80;
}
```

Problem: Can't do anything else while waiting.

Solution: Use an interrupt to tell you when the timer overflow has occurred.

Interrupt: Allows the HCS12 to do other things while waiting for an event to happen. When the event happens, tell HCS12 to take care of event, then go back to what it was doing.

What happens when HCS12 gets an interrupt: HCS12 automatically jumps to part of the program which tells it what to do when it receives the interrupt (**Interrupt Service Routine**).

How does HCS12 know where the ISR is located: A set of memory locations called Interrupt Vectors tell the HCS12 the address of the ISR for each type of interrupt.

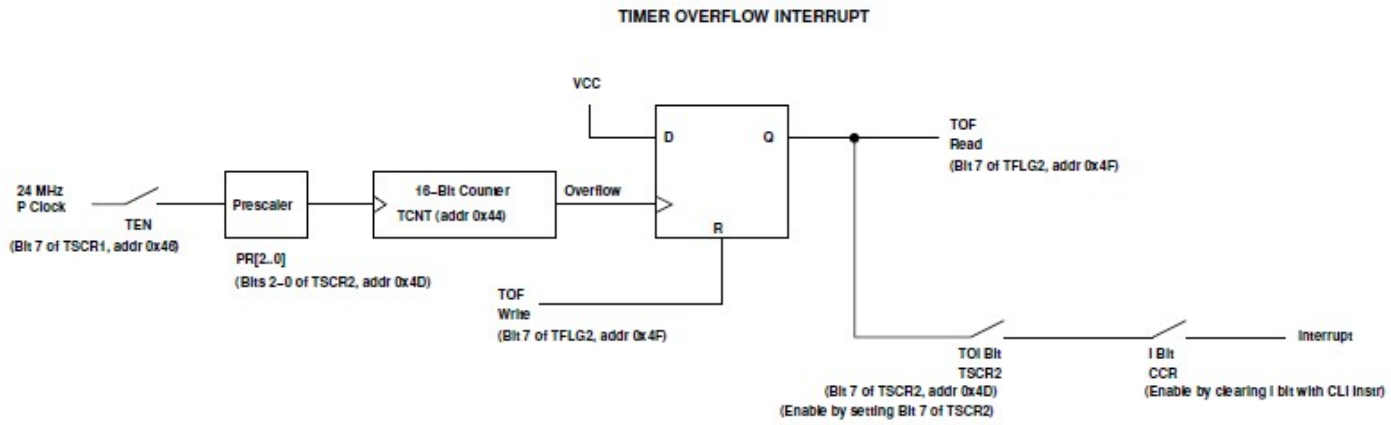
How does HCS12 know where to return to: Return address pushed onto stack before HCS12 jumps to ISR. You use the RTI (Return from Interrupt) instruction to pull the return address off of the stack when you exit the ISR.

What happens if ISR changes registers: All registers are pushed onto stack before jumping to ISR, and pulled off the stack before returning to program. When you execute the RTI instruction at the end of the ISR, the registers are pulled off of the stack.

What happens if you get an interrupt while in an ISR: MC9S12 disables interrupts (sets I bit of CCR) before it starts executing ISR.

To Return from the ISR You must return from the ISR using the RTI instruction. The RTI instruction tells the HCS12 to pull all the registers off of the stack and return to the address where it was processing when the interrupt occurred.

How to generate an interrupt when the timer overflows



To generate a TOF interrupt:

Enable timer (set Bit 7 of TSCR1)
Set prescaler (Bits 2:0 of TSCR2)
Enable TOI interrupt (set Bit 7 of TSCR2)
Enable interrupts (clear I bit of CCR)

Inside TOF ISR:

Take care of event
Clear TOF flag (Write 1 to Bit 7 of TFLG2)
Return with RTI

```
#include <hidef.h>
#include "derivative.h"

interrupt void toi_isr(void);

main()
{
        _ _asm(sei);        /* Disable interrupts */
        DDRA = 0xff;        /* Make Port A output */
        TSCR1 = 0x80;       /* Turn on timer */
        TSCR2 = 0x85;       /* Enable timer overflow interrupt, set */
                            /*  prescaler */
        TFLG2 = 0x80;       /* Clear timer interrupt flag */
        _ _asm(cli);        /* Enable interrupts (clear I bit) */
        while (1)
        {
                /* Put code here to do things */
        }
}


void INTERRUPT toi_isr(void)
{
        PORTB = PORTB + 1;      /* Increment Port A */
        TFLG2 = 0x80;           /* Clear timer interrupt flag */
}
```