Lecture 25

Solutions to Midterm #2

Clock System & Timers

What's Next

We have made it to the end of the semester !! – almost

Three more lectures (including today)

- Clock system and timers
- Timer interrupts and Blinky v.3 (Prep for final exam coding part, Friday)
- Looking back at the semester

A final exam in 2 Parts

- Coding Part: Holiday Blinky with timer interrupts
- Theory Part

When?

- Finals week
- Exact dates will be announced on Carmen

Midterm #2

Part 1: Coding task – 1-2-3 Count with me
Your program will start with both LEDs off and wait for a push button to be pressed.

As you press S1, both LEDs on your board will blink an increasing number of times. After your first press the LEDs will blink once; after your second press the LEDs will blink twice; after your third press the LEDs will blink three times, and so on. The state will be reset when you press S2. Please watch the short demo video posted to Carmen exhibiting the desired behavior.

Use a call to following subroutine between blinks

```
delay:
            push    R10
            mov.w   #0xFFFF, R10
countdown:
            dec.w   R10
            nop
            jnz     countdown

            pop     R10
            ret
```

What could/did go wrong?

An interrupt handler is not allowed to modify core registers!

You have to work with *global variables* defined in RAM

Solution to Midterm #2

Start by defining these variable(s):

```
;-------------------------------------------------------------------------------
            .data                                   ; Assemble into RAM
            .retain                                 ; Override ELF conditional linking
            .retainrefs                             ; And retain any sections that have

count:      .word   0
;-------------------------------------------------------------------------------
            .text                                   ; Assemble into program memory.
            .retain                                 ; Override ELF conditional linking
            .retainrefs                             ; And retain any sections that have
;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP                 ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL          ; Stop watchdog timer
```

Remember the assembler directives

`.data` indicates placement into RAM (runtime variables)

`.text` indicates placement into FRAM (code)

`.word` initializes a 16-bit space with given label and value

Solution to Midterm #2

Configuration is same old same old …

```
            ; Configure red LED for output, Red LED is connected to P1.0
            bic.b   #BIT0, &P1OUT
            bis.b   #BIT0, &P1DIR

            ; Configure green LED for output, green LED is connected to P9.7
            bic.b   #BIT7, &P9OUT
            bis.b   #BIT7, &P9DIR

            ; Configure push buttons S1 and S2 for input
            ; S1 is connected to P1.1; S2 is connected to P1.2
            bis.b   #BIT1|BIT2, &P1REN          ; Resistor enabled
            bis.b   #BIT1|BIT2, &P1OUT          ; Pullup resistor
            bic.b   #BIT1|BIT2, &P1IES          ; Interrupt on raising-edge
            bis.b   #BIT1|BIT2, &P1IE           ; Enable port interrupts

            bic.w   #LOCKLPM5, &PM5CTL0         ; Disable power lock

            ; Clear all IFGs in P1 in case they are set during config
            clr.b   &P1IFG

            nop
            eint                                ; Enable general interrupts
            nop

main:       jmp     main
```

Solution to Midterm #2

The only action is in the ISR:

```
P1_ISR:
            ; Check source of interrupt: is it P1.1?
            bit.b   #BIT1, &P1IFG
            jnc     check_S2
service_S1:
            inc.w   count               ; Count the button press

            push    R9
            mov.w   count, R9           ; R9 serves as loop counter
            add.w   R9, R9              ; double the number since
blink:                                  ; two toggles make one blink
            xor.b   #BIT0, &P1OUT       ; toggle red LED
            xor.b   #BIT7, &P9OUT       ; toggle green LED
            call    #delay              ; wait
            dec.w   R9                  ; account for one toggle
            jnz     blink               ; repeat until loop counter == 0

            pop     R9
            bic.b   #BIT1, &P1IFG       ; clear IFG

check_S2:   ; Check source of interrupt: is it P1.2?
            bit.b   #BIT2, &P1IFG
            jnc     fin

service_S2:
            clr     count               ; S2 resets the count
            bic.b   #BIT2, &P1IFG       ; clear IFG

fin:        reti                        ; return from interrupt
```

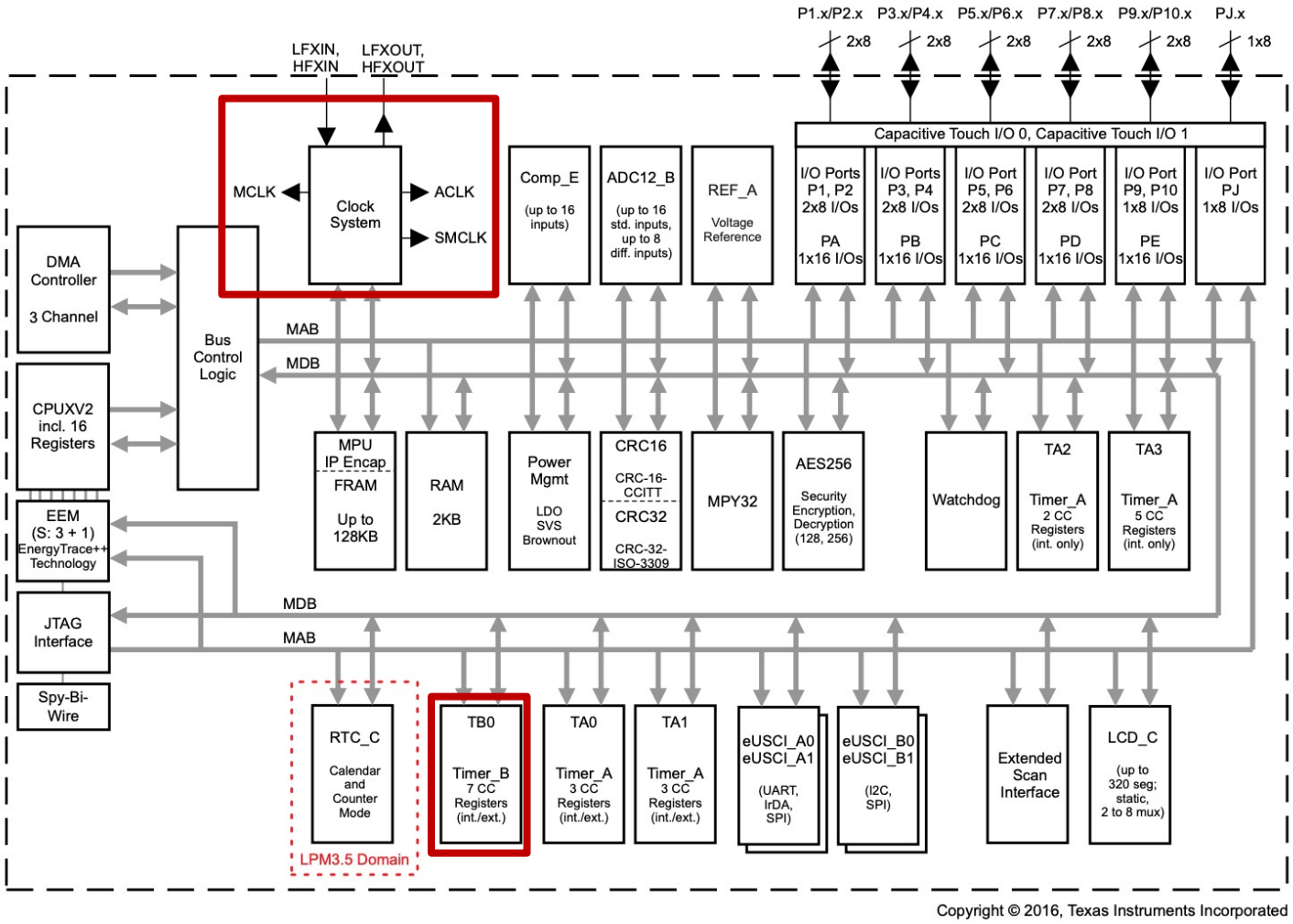
Solution to Midterm #2

My favorite solution from student submissions:

```
P1_ISR:
            bit.b   #BIT2, &P1IFG          ;test if S2 interrupted
            jc      Count_reset            ;reset count if S2 interrupted

            bit.b   #BIT1, &P1IFG          ;test if S1 interrupted
            jnc     end_ISR                ;jump to end if S1 not interrupted
            incd.w  count                  ;fall through for S1 was pushed
                                           ;increment count by 2
            push    count                  ;save count to restore after toggle
Blink_LEDs:
            xor.b   #BIT0, &P1OUT          ;toggle RED LED
            xor.b   #BIT7, &P9OUT          ;toggle Green LED
            call    #delay                 ;delay next instruction
            dec.w   count                  ;decrease count by 1
            jnz     Blink_LEDs             ;if count isnt 0 toggle LEDs again

            pop     count                  ;restore count value
            bic.b   #BIT1, &P1IFG          ;clear S1 button interrupt flag
            jmp     end_ISR                ;jump to end of ISR

Count_reset:
            mov.w   #0, count              ;count gets reset to zero
            bic.b   #BIT2, &P1IFG          ;clear S2 button interrupt flag

end_ISR:
            reti
```

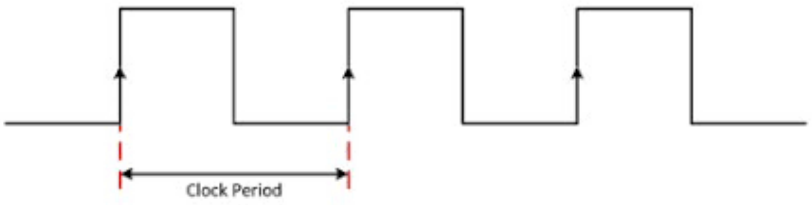
Back to MSP430 Architecture

Many more peripherals on our MCU



The Clock System

A **clock** is essential for every synchronous digital system – including MCUs



The clock signal is a **square wave** whose edges

- trigger hardware throughout the device and
- synchronize different components of the MCU

MCU clocks started simple: a high frequency **crystal oscillator** to drive the CPU and the main bus (typically after the frequency is divided down)

Crystal oscillators are

- accurate
- stable
- take a long time to start and stabilize
- relatively large current draw $\Rightarrow$ relatively high-power consumption

The Clock System

Modern MCU have different needs:

They spend most of their time in **low-power mode** waiting for some event

Then they need to wake up **quickly** and handle the event

Often, they need to keep track of **real time** as well

$\Rightarrow$ There is a need for multiple clock generators

At least two clocks are needed

- **A fast clock to drive the CPU**
  - which can be started and stopped rapidly to conserve energy
  - does not need to be particularly accurate
- A **slow clock** that runs continuously **to monitor real time**
  - uses little power
  - is accurate

Clock System of MSP430FR6989

Our MCU has several clock generators (oscillators) and clock signals

To learn more about them and to be able to configure and use them you have to read the user manual: slau367p.pdf

TEXAS INSTRUMENTS

Chapter 3

SLAU367P–October 2012–Revised April 2020

Clock System (CS) Module

This chapter describes the operation of the clock system, which is implemented in all devices.

Topic **Page**

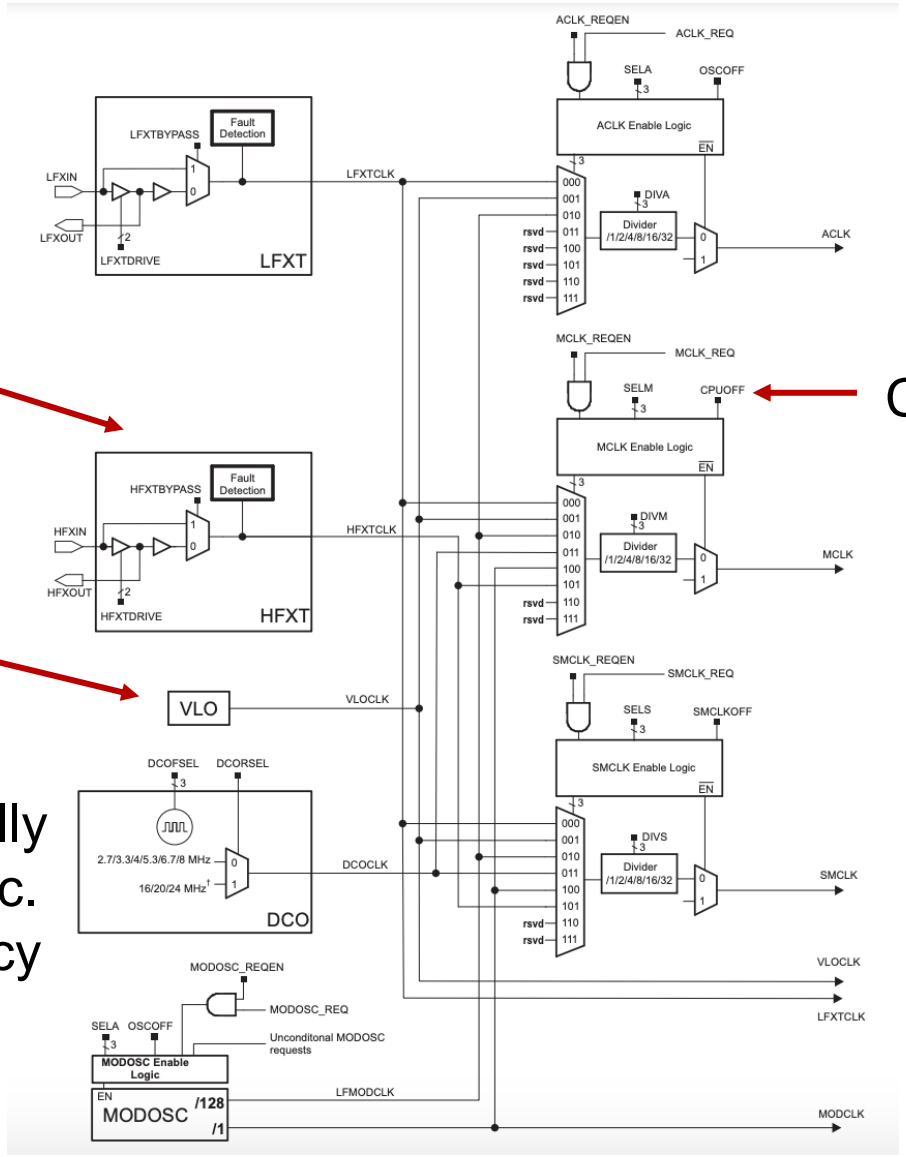
Clock System of MSP430FR6989

Low Freq. Crystal Osc.

High Freq. Crystal Osc.

VLO – Very Low Osc.

DCO – Digitally controlled Osc. High frequency



~ 32 kHz

ACLK – Auxiliary clock

CPUOFF in SR

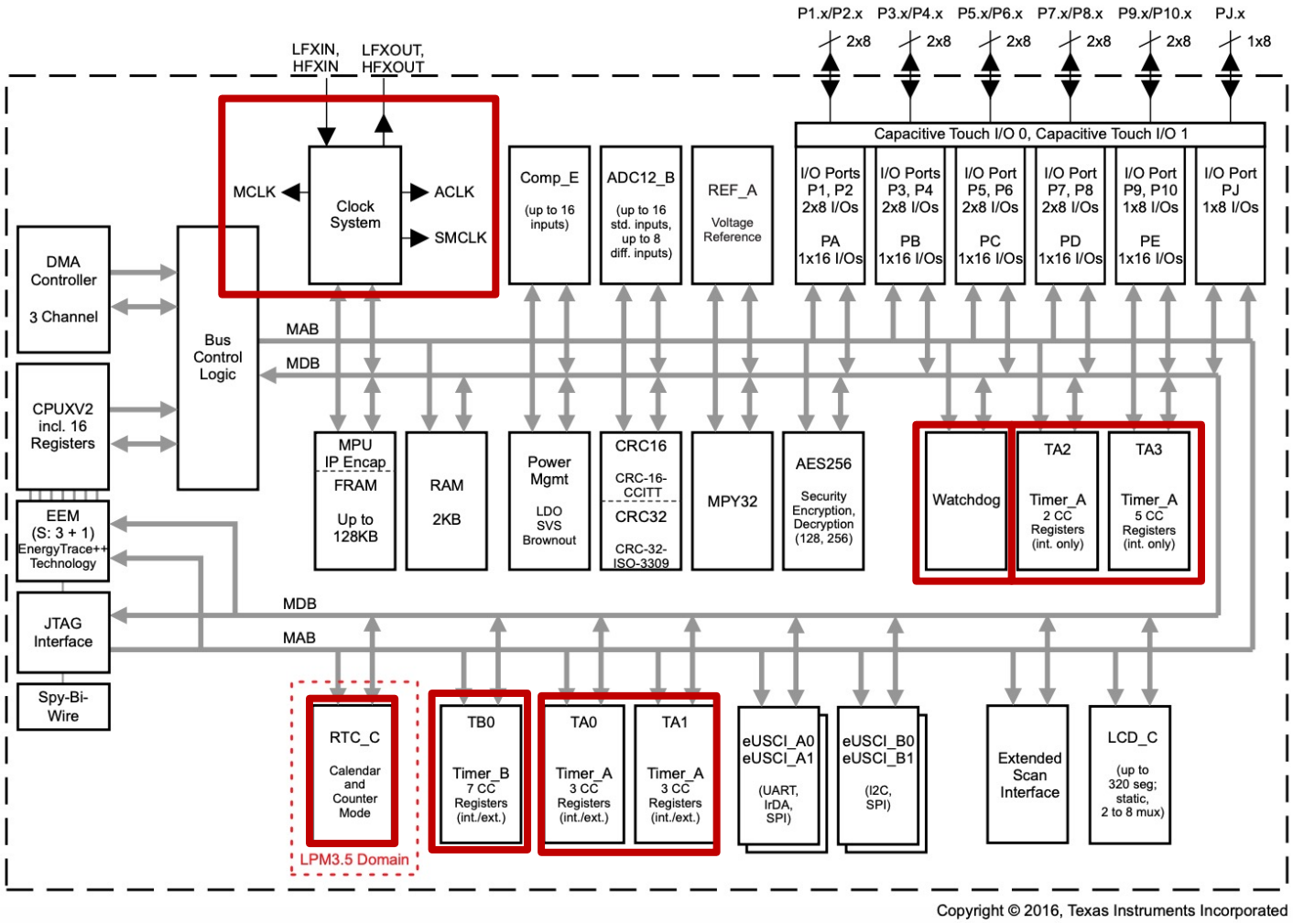
MCLK – Master clock
Drives the CPU

~ 1 MHz

SMCLK – Subsystem Master clock

Timers in MSP430FR6989

The clock system feeds into multiple timers



Timers in Software

We have implemented *timers* in software before

e.g., for Midterm 2 we had a countdown timer to create a fixed time of delay

```
delay:
            push    R10
            mov.w   #0xFFFF, R10
countdown:
            dec.w   R10
            tst.w   R10
            jnz     countdown

            pop     R10
            ret
```

`tst.w` here is not needed !!!
I have used it here to increase the period of the delay
Could have used `nop` too

Alternatively, we could have *counted up*

```
delay_2:
            clr.w   R10
countdown:
            inc.w   R10
            jnz     countdown

            ret
```

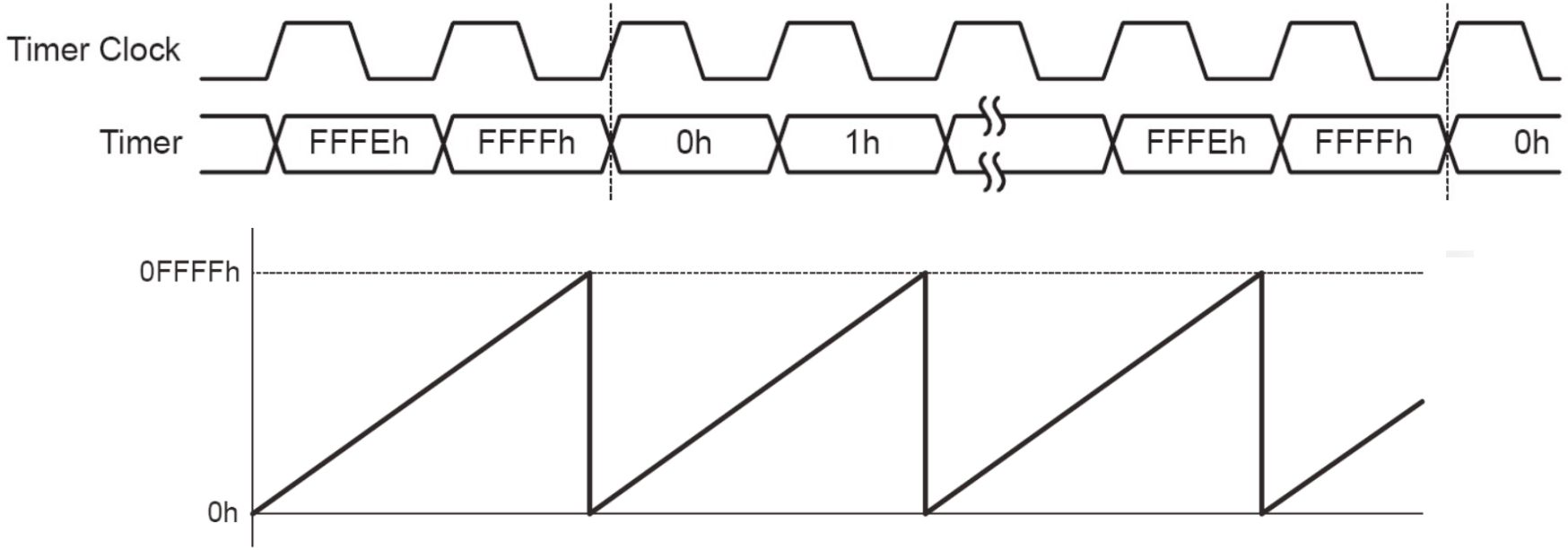
Hardware timers work according to the same principle – but independent of the CPU

Timers in Hardware

A *software* timer *works* BUT

- is not precise enough: CPU is driven by MCLK which is not very precise
- is very waste-full: wastes CPU cycles and power

A hardware timer counts the cycles of a clock signal and resets when the count reaches a certain number (e.g., exceeds the range of a 16-bit register)



Watchdog Timer

32-bit timer that can be used as a **watchdog timer** or as an interval timer

A **watchdog timer (WDT)** performs a system restart after a software problem occurs – e.g., an infinite loop

We have always stopped the WDT

```
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
```

For real life applications, the WDT can/should be left active

Software needs to periodically *pet the dog* before each WDT interval expires

```
; Periodically clear an active watchdog
mov.w   #WDTPW+WDTIS2+WDTIS1+WDTCNTCL, &WDTCTL
```

If the WDT interval is allowed to expire

- Interrupt flag `WDTIFG` will be set in `SFRIFG1.0` i.e., `BIT0` of `SFRIFG1` register
- A system reset will occur

Watchdog Timer

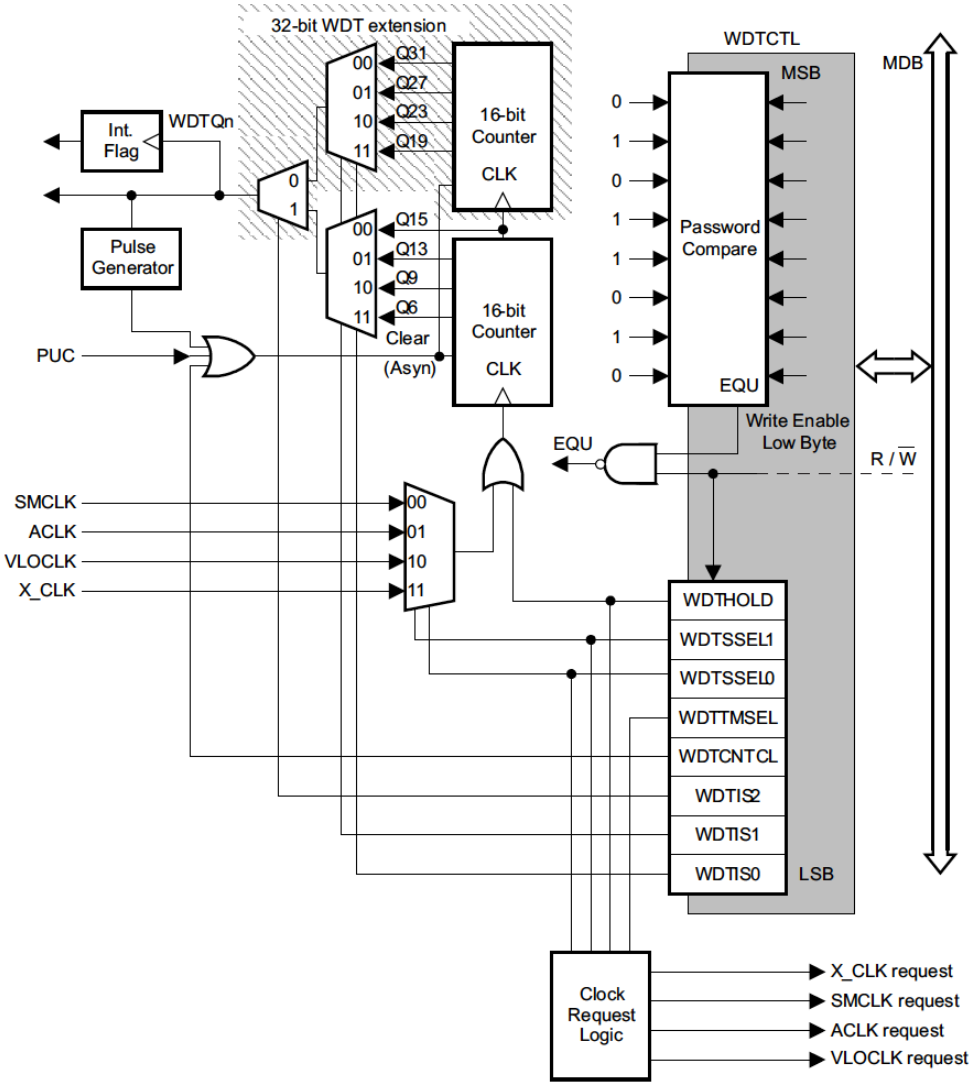


Figure 24-1. Watchdog Timer Block Diagram

The WDTCTL register is password protected!

```
mov.w   #WDTPW|WDTHOLD,&WDTCTL
```

The password is 0x5A

An attempt to write to WDTCTL without the correct password results in a PUC (Power Up Clear)

Timer B

16-bit timers that can be used for different purposes

- **Interval timer** (programmable to be 8, 10, 12, or 16 bits)
- Supports capture/compare
- Supports Pulse Width Modulation (PWM)

Clock source can be ACLK, SMCLK or external; selected with `TBSSEL` bits

Four mode of operations selected by the `MC` bits

Table 26-1. Timer Modes

MC	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of compare register TBxCL0.
10	Continuous	The timer repeatedly counts from zero to the value selected by the CNTL bits.
11	Up/down	The timer repeatedly counts from zero up to the value of TBxCL0 and then back down to zero.

For full configuration information refer to the user manual **slau367p.pdf**

Timer B Modes of Operation

Continuous mode:

timer repeatedly counts up to $TBxR_{(max)}$ and restarts from zero

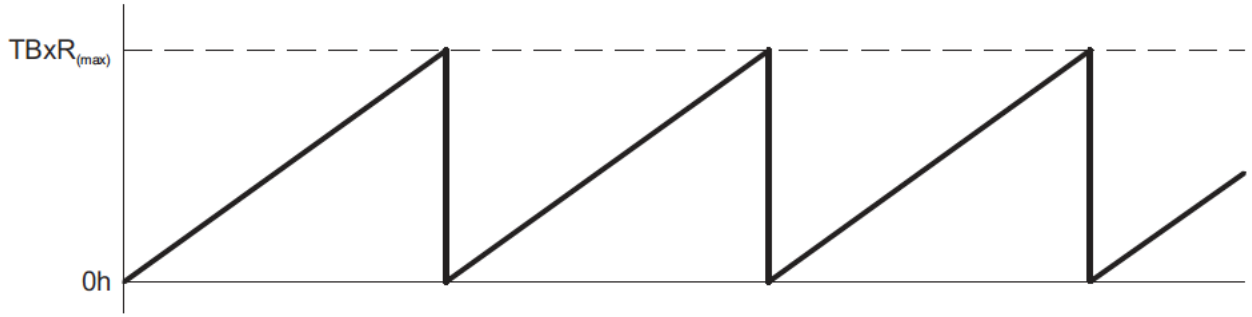


Figure 26-4. Continuous Mode

The `TBIFG` interrupt flag is set when the timer counts from $TBxR_{(max)}$ to zero

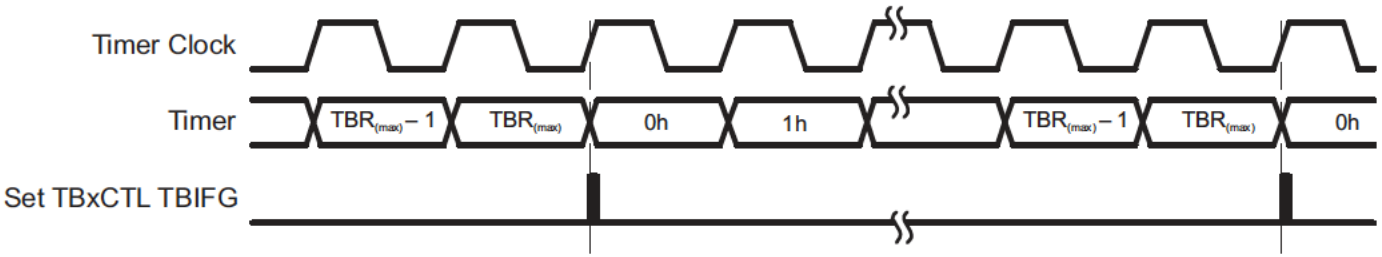
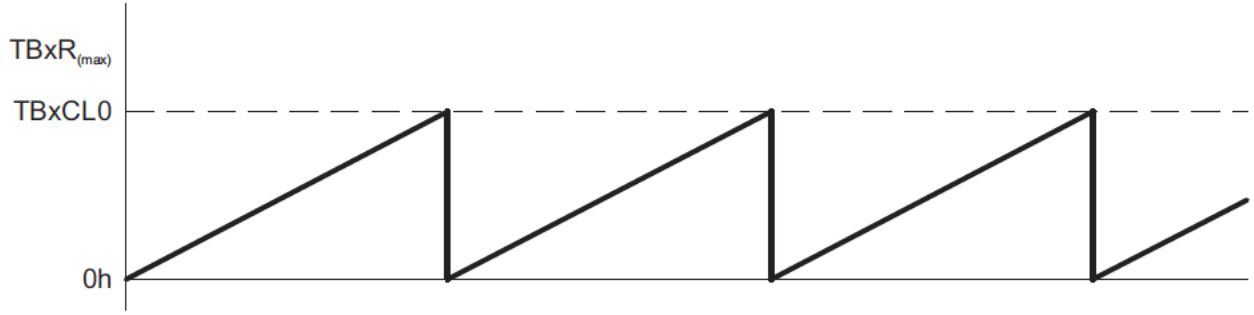


Figure 26-5. Continuous Mode Flag Setting

Timer B Modes of Operation

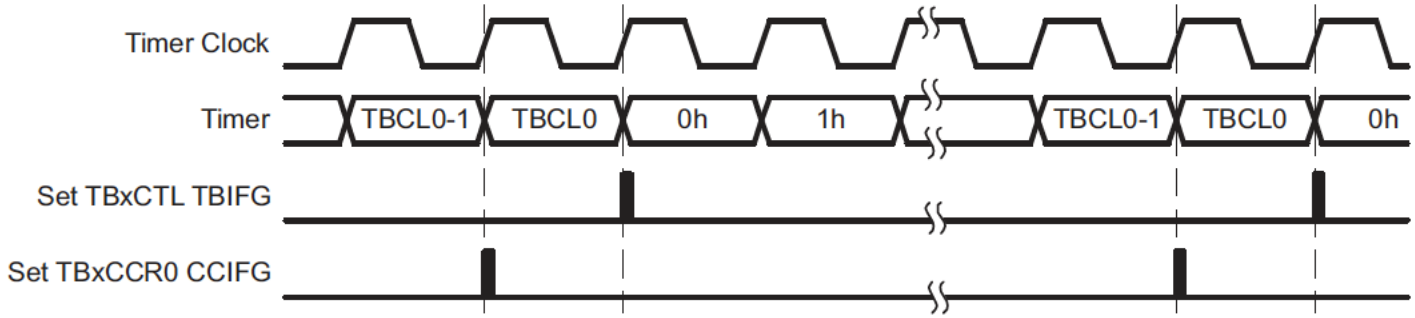
Up mode is used if the timer period must be different from `TBxR(max)` counts

The timer repeatedly counts up to the value of compare latch `TBxCL0`



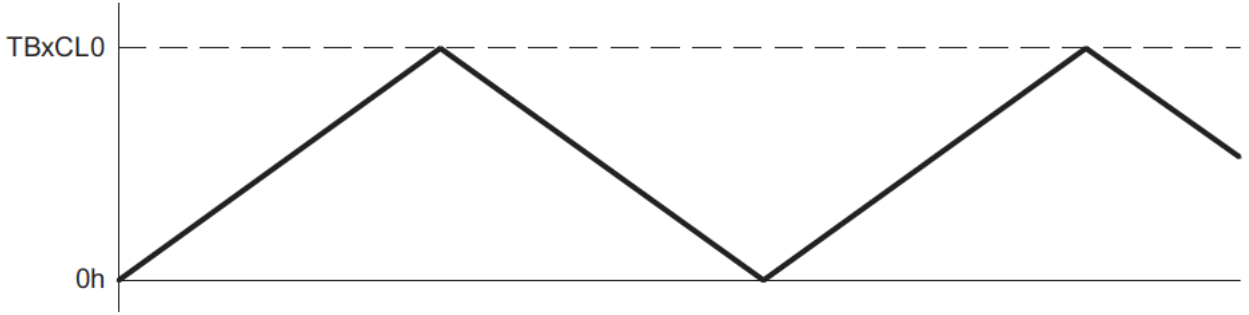
With this mode two interrupt flags are set:

- `TBxCCR0 CCIFG` flag when the timer counts to the `TBxCL0` value
- `TBIFG` flag when the timer counts from `TBxCL0` to zero



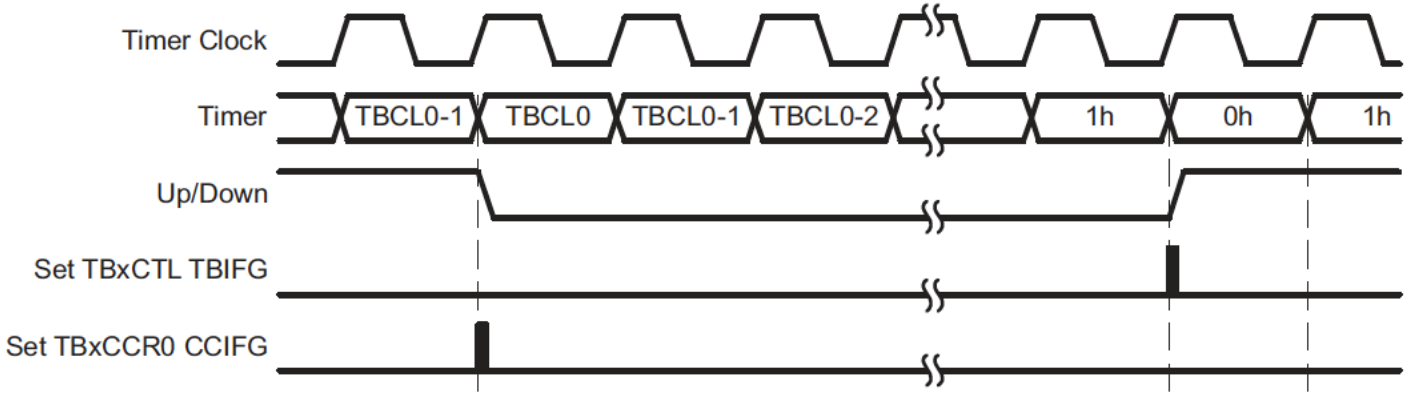
Timer B Modes of Operation

In **up/down mode** the timer counts up to the value of compare latch `TBxCL0` and back to zero resulting in a symmetric pulse

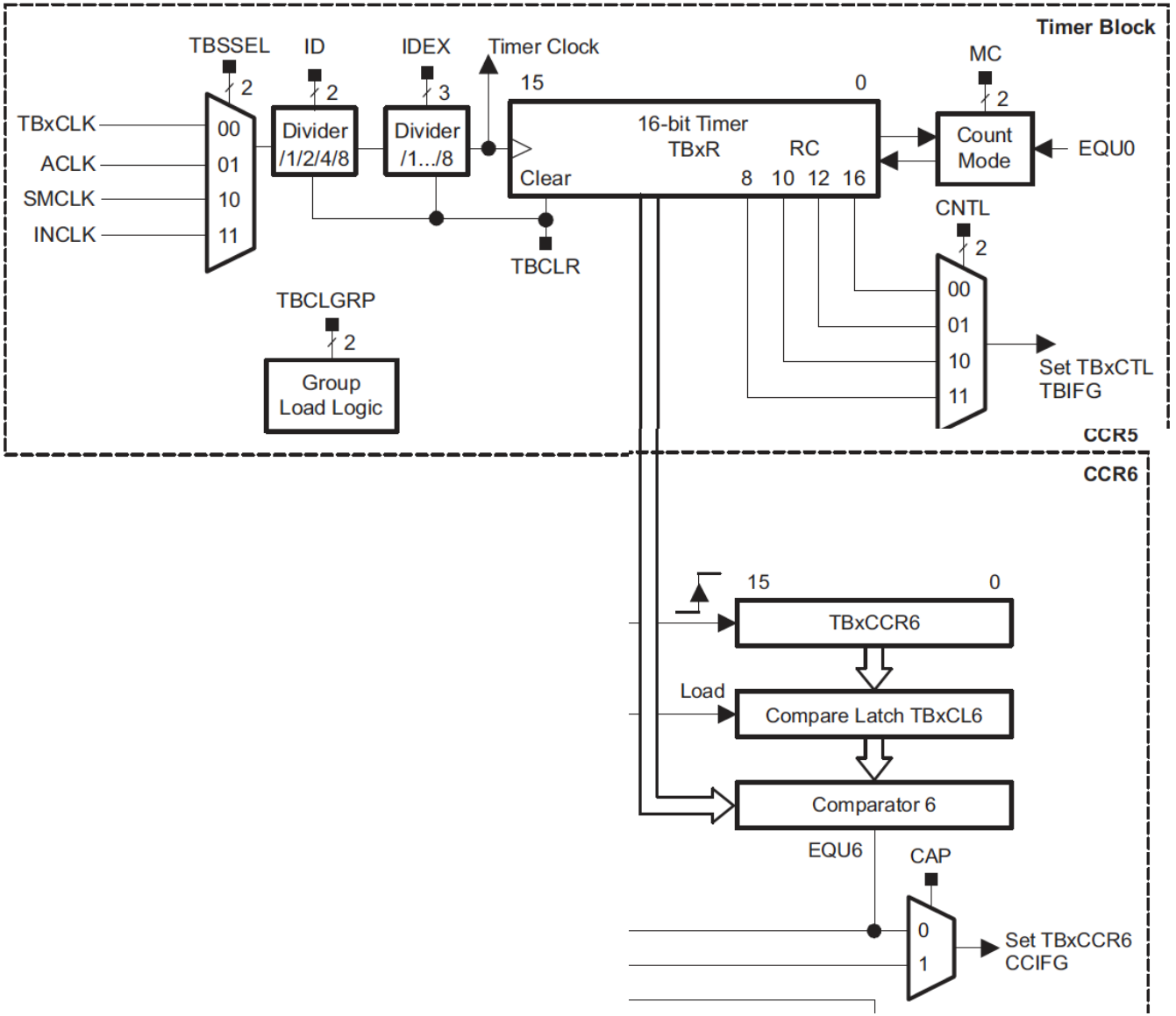


With this mode two interrupt flags are set:

- `TBxCCR0 CCIFG` flag when the timer counts to the `TBxCL0` value
- `TBIFG` flag when the timer counts from one to zero



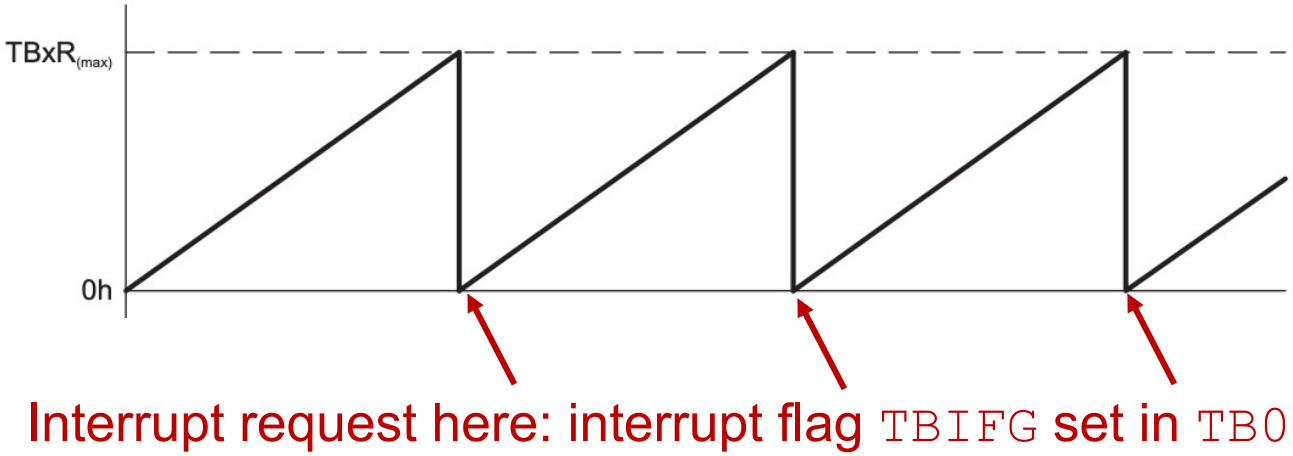
Timer B Block Diagram



Working with Timer B0 Interrupts

We will use Timer B0 in continuous mode to trigger periodic interrupts
And toggle the red LED when an interrupt is generated ⇒ **Blinky version 3**

We configure Timer B0 to throw periodic interrupts



We write an interrupt service routine (ISR) to serve Timer B interrupts
Populate the interrupt vector table (IVT)

Timer B Interrupts

I have configured Timer B for you – if you want to use a different configuration (e.g., with a higher frequency), you have to read the user's manual (, figure out what the registers do, and set the values

```
; Configure Timer B0 to raise interrupts
            bis.w   #TBCLR, &TB0CTL
            bis.w   #TBSSEL__ACLK, &TB0CTL      ; connected to ACLK
            bis.w   #MC__CONTINUOUS, &TB0CTL    ; continuous mode
            bis.w   #TBIE, &TB0CTL              ; enable interrupts
```

You know how to write an ISR

How do we populate the IVT?

Check out **Table 6-4. Interrupt Sources, Flags, and Vectors** in the data sheet for MSP430FR6869

https://www.ti.com/lit/ds/symlink/msp430fr6989.pdf

Also available on Carmen: **SLAS789D.pdf**

Timer B Interrupts

Table 6-4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power up, Brownout, Supply Supervisor External Reset $\overline{RST}$ Watchdog time-out (watchdog mode) WDT, FRCTL MPU, CS, PMM password violation FRAM uncorrectable bit error detection MPU segment violation FRAM access time error Software POR, BOR	SVSHIFG PMMRSTIFG WDTIFG WDTPW, FRCTLPW, MPUPW, CSPW, PMMPW UBDIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG, MPUSEG3IFG ACCTEIFG PMMPORIFG, PMMBORIFG (SYSRSTIV) (1) (2)	Reset	0FFFEh	Highest
System NMI Vacant memory access JTAG mailbox FRAM bit error detection MPU segment violation	VMAIFG JMBINIFG, JMBOUTIFG CBDIFG, UBDIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG, MPUSEG3IFG (SYSSNIV) (1) (3)	(Non)maskable	0FFFCh	
User NMI External NMI Oscillator fault	NMIIFG, OFIFG (SYSUNIV) (1) (3)	(Non)maskable	0FFFAh	
Comparator_E	Comparator_E interrupt flags (CEIV) (1)	Maskable	0FFF8h	
Timer_B TB0	TB0CCR0.CCIFG	Maskable	0FFF6h	
Timer_B TB0	TB0CCR1.CCIFG to TB0CCR6.CCIFG, TB0CTL.TBIFG (TB0IV)(1)	Maskable	0FFF4h	
Watchdog timer (interval timer mode)	WDTIFG	Maskable	0FFF2h	

address for interrupt vector is **0xFFF4**

Recall: Populating the IVT

This is how we did it for Port 1, we'll do the same for Timer B

We start by finding the **word address for interrupt vector: 0xFFDA**

We locate the word address in the linker file "lnk_msp430fr6989.cmd"

lnk_msp430fr6989.cmd | main.asm | main.asm | *main.asm | m

```
103     INT34                   : origin = 0xFFD4, length = 0x0002
104     INT35                   : origin = 0xFFD6, length = 0x0002
105     INT36                   : origin = 0xFFD8, length = 0x0002
106     INT37                   : origin = 0xFFDA, length = 0x0002
107     INT38                   : origin = 0xFFDC, length = 0x0002
108     INT39                   : origin = 0xFFDE, length = 0x0002
```

We add the **label of the ISR** to the Interrupt Vectors (at the end of *.asm)

```
;-----------------------------------
; Interrupt Vectors
;-----------------------------------
            .sect   ".int37"
            .short  P1_ISR

            .sect   ".reset"
            .short  RESET
```

Identifies address 00xFFDA

Label of ISR