Programming for Embedded Systems
Lecture 3: Timer A

Bernhard Firner

February 4, 2015

Timing

- We've been using the WDT as our clock
- This has a few problems
  - Very coarse, and only a few settings
  - Can only time one interval at a time
- The MSP430 has specialized timer modules than can do more

Timer Modules

- The MSP430 family has two high-precision timer modules
  - Timer A and Timer B
  - The G2553 has two Timer As (A0 and A1)
- These timers have multiple modes
- Also have multiple interrupts
  - So you can count multiple time intervals simultaneously
- Can also set an interrupt to stop the counter
  - Used to time event durations
- See section 12.2 in the MSP430x2xx family user guide

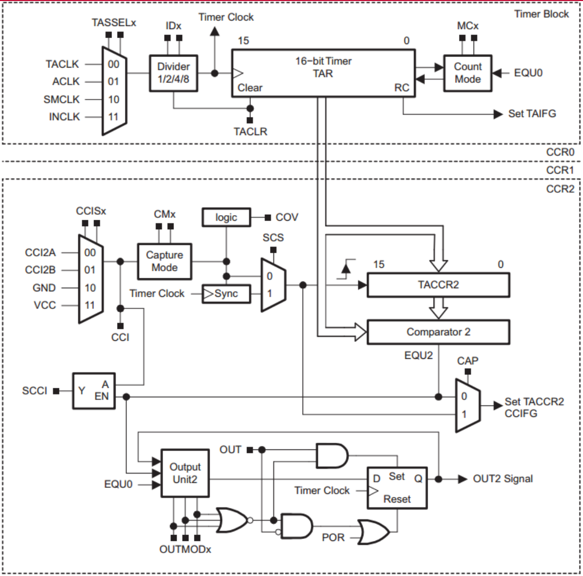
Timer A Overview

- There is one counter associated with the timer
- Timer A has three *capture compare registers*
    - TACCR0-2
- The TACCRx registers can each trigger an interrupt

Timer A Also Control Output Pins

- The OUTMOD registers directly connect timer output to output pins
- This allows the hardware to directly generate time-based output
    - We'll use one of these output modes soon

Timer A Block Diagram



- The three TACCRs are nearly identical
- CCIFG is the "capture compare interrupt flag"

The Control and Counter Registers

- TACTL
  - Timer A Control
  - This register controls how timer A runs, its clock source, etc
  - This register is 2 bytes wide
- TAR
  - Timer A Counter
  - Timer A uses this register to keep track of the clock ticks that have passed
  - Interrupts can be triggered when this resets to 0
- Please note, these are also called TA0CTL and TA0R since TA1 is also present

Family User's Guide, pg 370

12.3.1 TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						**TASSELx**	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		**MCx**		**Unused**	**TACLR**	**TAIE**	**TAIFG**
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted. 01 Up mode: the timer counts up to TACCR0. 10 Continuous mode: the timer counts up to 0FFFFh. 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.

Clock Source

- As with the WDT, you can select a clock source
  - Controlled by bits 9-8 in TACTL
- 00: TACLK
- 01: ACLK
- 10: SMCLK (this is what we've been using with the WDT)
- 11: TACLK
- SMCLK is the "sub-main" clock, sourced from DCO by default

Clock Divider

- We can divide the clock signal if it is too fast
    - Bits 7-6 in TACTL
- 00: divide by 1
- 01: divide by 2
- 10: divide by 4
- 11: divide by 8

Timer Mode Control

- Unlike the WDT, you can run Timer A in 4 modes
  - controlled by bits 5-4 in the TACTL register
- 00: Stop mode, timer does not run
- 01: Count up to the value in TACCR0
- 10: Continuous counting up to 0xFFFF
- 11: Count up to TACCR0 then down to 0

Timer A TAR Interrupts

- There is an interrupt enable, a flag, and an interrupt vector
  - Bit 1 in TACTL is the interrupt enable (TAIE)
  - Bit 0 in TACTL is the interrupt flag (TAIFG)
- In Code Composer the correct pragma for the interrupt is

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TimerA(void) { ... }
```

- Don't forget to also call __enable_interrupt();

Before Using the Timer

- You can reset the current state of the timer by writing to TACLR
  - Bit 2 in the TACTL register
- Hardware will set this back to 0
- Writing a 1 will clear the count in TAR
  - Also resets the count direction for mode 3 (up/down) and the clock divider

Capture/Compare Registers

- TACCR0-2
  - Timer A Capture Compare Registers
  - TAR is compared to TACCR0 to see if counting is complete
  - Interrupts can also be triggered when TAR == TACCR0-2
  - In capture mode (duration measurement) the time before an event occurs is written into this register

TACCR0 Interrupts

- There is another interrupt vector for when TAR == TACCR0

  ```
  #pragma vector=TIMER0_A0_VECTOR
  __interrupt void Timer0A(void) { ... }
  ```

- The other interrupts are in TIMER0_A1_VECTOR
  - This may be a bit confusing, so double-check when using

The TAR Register

- Do not read this register when the timer is running
- The CPU clock and the timer are not synchronized
  - This register might change in the middle of a read
- If you absolutely must read this register
  - Read it multiple times
  - Take a majority vote for each bit

Starting Timer A

- Set mode to anything but 0 and have a valid clock input
- In modes 1 and 3 (up and up/down) write a non-zero value to TACCR0
  - Once TACCR0 is non-zero TAR will start counting to it

Mode 1: Up

- The timer will count to the value in TACCR0 and then reset to 0
- The timer starts from 0
  - There will be TACCR0 clock ticks before TACCR0 CCIFG is set
  - There will be TACCR0+1 clock ticks before TAIFG is set

Mode 2: Continuous

- The timer will count to 0xFFFF and then reset to 0
- The timer starts from 0
  - There will be 0x10000 clock ticks before TAIFG is set
  - TACCR0 CCIFG will still be set when TAR == TACCR0

Mode 3: Up/Down

- The timer will count to the value in TACCR0 and then count back down to 0
- The timer starts from 0
    - There will be TACCR0 clock ticks before TACCR0 CCIFG is set
    - There will be another TACCR0 clock ticks before TAR reaches 0 and TAIFG is set

Handling Capture/Compare Interrupts

- All interrupts are stored in the TAIV register
- Every read of the TAIV register resets the bit for the highest priority interrupt
  - Means that you don't need to clear the interrupt flags
- However, the Timer A interrupt handles multiple interrupt sources!

Interrupt Sources

- The TAIV register must be checked to see what caused an interrupt

```
switch (TAIV) {
    case TA0IV_NONE:
        //No interrupt
        break;
    case TA0IV_TAIFG:
        //TAR overflow to 0
        break;
    case TA0IV_TACCR1:
        //TAR reaches TACCR1
        break;
    case TA0IV_TACCR2:
        //TAR reaches TACCR2
        break;
}
```

- Interrupts from TACCR0 occur in a different vector
- Notice that we need to explicitly say TA0 instead of TA

Timer A Example

- The bit patterns to set TACTL are already defined in CCS

```
//Mode two (continuous), divide by 8,
//SMCLK clock source, interrupt enabled.
TACTL = MC_2 | ID_3 | TASSEL_2 | TAIE;
__enable_interrupt();
```

- TimerA interrupt will now start firing

```
#pragma vector=TIMER0_A1_VECTOR
```

Basic Timer A Example

- Basic use of Timer A - how fast is this?

```
/*
 * main.c
 */
int main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer

    //Set the clock to a slow 1MHz
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;

    //Set up pin 1.0 as an output pin
    P1DIR |= BIT0;

    //Mode two, divide by 8, SMCLK clock source, interrupt enabled.
    TACTL = MC_2 | ID_3 |  TASSEL_2 | TAIE;
    __enable_interrupt();

    while (1);
    return 0;
}
```

Basic Timer A Example Continued

```
#include <msp430.h>

#pragma vector=TIMER0_A1_VECTOR
__interrupt void TA1() {
    //Reading from TAIV clears it
    switch (TAIV) {
        case TA0IV_TAIFG:
            //Just toggle when TA0 interrupt is triggered
            P1OUT ^= BIT0;
            break;
        default:
            //Don't care about other interrupts
            break;
    }
    //Don't need to reset any timer registers
    //The counter will keep counting
}
```

Using TACCR0

- This is the capture compare register
- Stores a 16 bit value (up to 65535)
- Enable TACCR0 interrupts

```
TACCTL0 = CCIE;
```

- Interrupts for TACCR0 (and only 0) are handled in a different vector

```
#pragma vector=TIMER0_A0_VECTOR
```

Blinking Two LEDs

- We can blink the green LED on P1.6
- Let's have the TACCR0 interrupt control that one

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TA0() {
    //The TACCR0 interrupt fired
    P1OUT ^= BIT6;
}
```

- In main set P1.6 as output, have TACCR0 count to half of overflow, and enable the TACCR0 interrupt

```
P1DIR |= BIT0 | BIT6;
TACCR0 = 32768;
//Enable the interrupt for TACCR0
TACCTL0 = CCIE;
```

More Timer Features

- Timer A can also directly drive output from the pins
- This allows the hardware to support several special features
- We will be using it to support *digital to analog* conversion
- More on this and Timer A next time