Timer Module – Timer A

ReadMeFirst

Lab Folder Content

1) ReadMeFirst
2) TimerModule – Lecture material
3) PinOutSummary
4) InterruptsVectorTable
5) Source code for screencast Interrupt Review

Overview

A Timer Module is one of the most important modules on a microcontroller. With a timer you can generate regular event interrupts with your CPU, awake or asleep, generate custom frequencies and waveforms, produce PWM (Pulse Width Modulation), record the duration of events or time stamp an event, measure frequencies, measure time constants for capacitive or resistive measurements and much more. In this lab we are going to introduce to you the basic idea of the Timer Module and some simple uses of through a few examples.

The microcontroller we have on our Launchpad, MSP430G2553, contains a versatile, and general purpose timer called Timer_A. An MSP430 microcontroller can have multiple timer modules. In G2553, we have two Timer_A modules named Timer_A0 and Timer_A1.

Each Timer module consists of two types of blocks: A Timer Block and (typically three) Capture and Compare Blocks.
1. Timer Block:

This core block is based on a 16 bit TAR (Timer A Register). TAR is used to count clock cycles. The availability of particular clock sources as timer inputs is dependent on the MSP430 microcontroller you are using and which Timer_A module within a microcontroller you are using. For example, in the G2553, external input clock TACLK is available for Timer_A0 but not for Timer_A1. How TAR counts can be selected via setting up the count mode. You can have TAR counts up to $2^{16}-1$ and overflow (continuous mode), have TAR count up to the value stored inside a register named TACCR0 and drop back to zero (up mode) or let TAR counts up to a TACCR0 and counts down back to 0 (up/down mode). TACCR0 will be explained later.

Timer Block also support interrupt capability through the setting of TAIFG (Timer A Interrupt flag)
2. Capture Compare Registers (CCRs)

Capture and Compare Registers (CCRs) are able to either capture the timer TAR value when an input event happens, or compare the timer TAR value with a pre-stored value in TACCRx (Timer A Capture Compare Register x, where x = 0, 1, or 2 is the index of the CCR you are using). You can then setup an output signal (OUTx) or generate a Capture Compare Interrupt.

In the G2553 we have 3 capture compare registers available for each Timer_A module. Therefore in total we have 6 capture compare registers. CCRs are almost all the same, however Capture Compare Register 0 is slightly different because CCR0 is also used to set the upper limit of TAR when TAR is set to “up” or “up/down” mode. Each timer block has its own dedicated TAR block.

Please watch the screencast Timer A Module (i)

Here we provide a basic summary for setting up the clock module.

1) Pay attention that timer A registers are 16 bit long (one word) instead of 8 bit long (one byte) as clock control registers.

2) If you have multiple timing tasks that need to be accomplished, choose which timer module to use for which task.
   a. Notice that timer modules can be slightly different. Therefore understanding these differences is very important.

3) Set up a clock source for your selected timer
   a. You are able to select from SMCLK, ACLK and external clock TACLK. Some timer module also allows input from a pin oscillator
   b. Requirements of the application will determine what clock frequency you should use. Timing specifications would sometimes pose constrains on your clock system settings.

4) Select counting mode for TAR
   a. Remember to leave “stop” mode when you want TAR to run.
   b. If your TAR is not increasing, either your clock source has failed or your counting is in “stop” mode

5) If using capture mode:
   a. Select capture source
b. Setup capture mode: on rising/falling edge

c. Always select sync with timer clock

6) If using compare mode

a. Store value into selected CCR register

7) Setup outmode and interrupt if needed

a. Notice that CCR1, CCR2, and Timer Interrupt is multi-sourced. Therefore you need to manually clear corresponding flag inside your ISR.

b. CCR0 however has its own dedicated interrupt vector.

8) More about setting up capture compare registers

a. The counting ceiling in up and up/down mode only depends on CCR0, not CCR1 or CCR2. The timer register (TAR) does not overflow in up and up/down mode.

b. Device packaging and the selection of timer will determine which output pins are available to you.

- Please make sure you understand the contents of the document TimerAModule before you proceed with the lab experiments.

**Handling Low Power Mode**

Here’s an example of how you should set up G2553 for low power mode. You may skip reading this part if you feel comfortable with low power mode.

*(In Main Code)*

Sleep:

```
bis.w #GIE|LPM3,SR       ; Enter LPM3 (Low Power Mode 3)

. . . . .
```

```
jmp       Sleep            ; go to sleep again
```
ISR:

```
bic.w   #LPM3, 0(SP) ; CPU awake on return from ISR
```

Clear interrupt flag for multi-sourced interrupts

```
reti
```

**Handling Interrupts**

Here is a summary of how to setup an interrupt. You may skip reading this part if you are comfortable with interrupts.

You need to consider the following settings to successfully setup interrupt:

1. **Interrupt enable** – for each interrupt, the interrupt enable (IE) is typically a bit in a register. Setting this bit indicates you would like to have that particular interrupt.
   
e.g. `bis.b  #BIT0, &P1IE` ; enable interrupt on P1.0

2. **Interrupt flag** – when the interrupt is enabled and the flag bit is raised/set, an interrupt is thrown
   
e.g. P1.0’s interrupt flag is bit0 in register P1IFG

3. **Interrupt vector** – when an interrupt is thrown, the microcontroller looks into the interrupt vector to extract the address of interrupt service routine. To find the peripherals and their corresponding interrupt vector, please look at the document “InterruptsVectorTable” in your lab folder.

   e.g. To find port1 interrupt vector, look for “I/O Port P1” in the InterruptVectorTable document. The document shows that the location of the vector for port 1 reacts to flags bit0-7 of P1IFG register; the vector is memory section “int02” and it has priority level 18.
Therefore at the end of the program you should use:

```
.sect "int02"

.short YourInterruptServiceRoutineLabel
```

4. **Interrupt service routine** – where interrupt event is handled. Before entering interrupt service routine, the PC and Status register are pushed onto the stack. The routine needs to be terminated with the statement “reti” (return from interrupt). When “reti” is called the PC and SR are pulled back from the stack and the program goes back to the state at the time when the interrupt happened.

   e.g.

   `YourInterruptServiceRoutineLabel:

   Take care of low power mode if any, for example

   bic.w #LPM3, 0(SP)

   Your code

   Clear interrupt flag if interrupt is multi-sourced

   reti`

5. **General interrupt enable** – No maskable interrupt will be able to be thrown unless the general interrupt enable (GIE) bit in the status register is set. Make sure to enable after all system settings are completed (typically right before the ending infinite loop).

   e.g. `bis.b #GIE,SR`
**Experiment 1 – Getting familiar with Timer Block**

In this experiment we are going to guide you through setting up Timer A0’s timer block: Connect SMCLK to timer block, set Timer A0 in continuous counting mode, and generate an interrupt when TAR overflows. We will toggle pin 1.0 inside the interrupt service routine in order to observe the result. Remember that Timer A registers are word (16 bit) long!

1) What is the expected frequency at pin 1.0 under this setup?
   
   o Provide calculation in your lab report (0.5pt)

   Note that in toggling (in an ISR) an output pin to generate a frequency, the actual time period will be double the time period between interrupts.

2) Set up Timer A

   *Set Pin 1.0 as GPIO output.*

   *Set Timer A0 Source Select (TASSEL, bit 9-8, in register TA0CTL) to SMCLK (TASSELx = 10)_2*

   *Use Input Divider (ID, bit 7-6, in register TA0CTL) equal to divide by 2 (IDx = 01)_2.*

   *Set Mode Control (MC, bit 5-4, in register TA0CTL) equal to Continuous Mode (MCx = 10)_2*

   *Enable Timer A Overflow Interrupt (TAIE, bit 1, in register TA0CTL)*

   *Enables global interrupt GIE in the Status Register.*

   *Put MCU into LPM0 (Low Power Mode 0) => LPM0 kills CPU but keeps DCO, SMCLK and ACLK.*

   *Setup interrupt vector: TAIFG Add instructions (Timer A interrupt flag generated by Timer A0 interrupt) is handled by section INT08*

   *Toggle Bit0 of P1OUT inside your Interrupt Service Routine.*
Don’t forget to handle the low power mode in the ISR

3) Measure the waveform at pin1.0.
   o Include a screenshot in your lab report. Does it matches your estimation? (0.5pt)

Experiment 2 – Compare Mode: Frequency Generation

Frequency generation can be used when interacting with peripherals outside the MCU. For example, a motor control driver that accepts a steady frequency as input signal to control motor speed.

You can generate a frequency in the up mode, up-down mode with CCR0 or in continuous mode. In the up mode, you can generate a frequency very easily by using CCR0, but you cannot use CCR1 and CCR2 for this purpose. In continuous mode, you can use any of the three CCRs and use them simultaneously to give you multiple frequencies, but the setup is slightly more complicated. In this experiment we are going to explore the differences.

In this experiment we will explore using both the up mode and the continuous mode to generate frequencies.

In this document, CCRx refers to the Capture Compare Module. CCRx register refers to the actual capture compare register.

Part I

Generate a frequency of 100Hz in up mode using Capture Compare Module 0 (CCR0) in Timer A0.

1) Use 1MHz SMCLK for Timer A0 and set Timer A0 to count up mode

   Set Timer A0 Source Select (TASSEL, bit 9-8, in register TA0CTL) to SMCLK (TASSELx = 10)_2

   Set Mode Control (MC, bits 5-4, in register TA0CTL) to Up Mode (01)_2
Theoretically what value should be stored in TA0CCR0 register? Show your reasoning in the lab report (1pt)

2) Setup Capture Compare Module 0 in Timer A0

(By default your capture compare module is in compare mode)

Store your calculated value in TA0CCR0 register.

Setup CCR0 interrupt by setting CCIE bit in TA0CCTL0

Enable Global Interrupt Enable (GIE is Status Register SR)

Put MCU to sleep in LPM0

Setup interrupt vector: CCR0 interrupt of Timer A0 is handled by section INT09

3) Setup pin 1.1 for frequency generation

Setup pin 1.1 to be an output pin

Inside your interrupt routine, toggle pin 1.1 to generate the desired frequency.

– Include a screenshot of the oscilloscope in the report. (1pt)

Part II

In part II, we will modify and build on top of Part I to generate a separate frequency of 200Hz using CCR1 in Timer A0.

1) Convert Part I frequency generation from using counter up mode to using counter continuous mode

Set Mode Control (MC, bit 5-4 in register TA0CTL) equal to Continuous Mode \((10)_2\)

Inside the CCR0 interrupt handler, add to TA0CCR0 register a certain value (what value should you use?) and toggle pin 1.1
2) Add CCR1 to the problem. Theoretically what value should be stored in TA0CCR1 register to generate 200Hz frequency?

Setup pin 1.2 to be an output pin

Setup CCR1 interrupt by setting CCIE bit in TA0CCTL1

Setup interrupt vector: CCR1 interrupt of Timer A0 is handled by section INT08 (Notice that INT08 is multi-sourced. What flag should you clear?)

Inside your interrupt routine, add to TA0CCR1 register a certain value (what value should you use?) and toggle pin 1.2 to generate the desired frequency.

- Include a screenshots oscilloscope with both frequencies showing simultaneously (1pt)
- Answer the question: What would happen if you left counting mode as up mode? (1pt)

*Please read this section carefully before proceeding to Experiment 3!*

**Output Modes of Timer A**

In the previous experiment we toggled a pin in timer interrupt routines to generate various frequencies. In this lab we are going to generate a frequency using the output module of Timer A without having to use interrupts.
You can find the schematic of the output module in Timer A handout as well as in the family guide. We include part of the schematic below:

As you can see the output mode is handled by hardware. This means that you can generate signals at the output without the need to use ISRs to toggle pins and spend CPU computing cycles when you use this mode.

Output pins of Timer A output module are called TAx.y. “x” is the timer number. In G2553 we have two timers TimerA_0 and TimerA_1 so x is 0 or 1. “y” is the capture compare register. y is either 0, 1 or 2. A list of available output modes and their description are summarized below. Notice that there are two types of output modes, one only depends on the capture compare register selected (CCR0, CCR1, CCR2), like modes 0, 1, 4 and 5. When the timer counts up to the selected CCR register value, the corresponding output pin is set, reset or toggled. Name of these modes simply follows the operation: “Output”, “Set”, “Toggle” and “Reset”.

The other type of output mode is related to the selected capture compare register (CCR1 or CCR2) and the special CCR0 register. The output changes when the timer reaches the selected CCR register and change again when it reaches the CCR0 value. Examples are modes 2, 3, 6 and 7. In the table you can also see the names of these modes as: “Toggle/Reset”, “Set/Reset”, “Toggle/Set” and “Reset/Set”. In these modes we use the output modules of CCR1 and CCR2, it is not useful to use the output mode of CCR0.

All these built-in options provide you with a lot of flexibility.
You can refer to the lab document “PinOutSummary” to look up pin mapping and configuration of the CCR output pins. “TimerA” document also has more details on this output module.

Note that the output module of CCR2 in TimerA0 has no output pin mapped in this 20PDIP packaging of G2553. All three CCRs from Timer A1 do have output module pin mapped.

**Experiment 3 – Use Output Hardware in Capture Compare Module**

In this experiment we are going to modify Experiment 1 to generate a 100Hz frequency without using interrupts by using the output hardware module of Timer A.

1) Remove pin configurations previously on p1.1.

2) Remove the interrupt service routine, it is no longer needed since the toggling is going to be done via the hardware output module.
3) Timer A0 CCR0 output module pin is on pin 1.1. Configure pin p1.1 to be connected to the Timer A0 output module. (Please refer to PinOutSummary document for more details)

4) Add to your main program:

   Configure output mode (OUTMODx, bit 7-5 in register TACCTLx) to toggle mode (100) in TA0CCTL0.

5) Measure and record the frequency generated using this new method.
   - Include a screenshot of the oscilloscope (2pt, must provide code)

Short Answer Questions

1. Explain/Illustrate how Timer A counts differently in continuous mode, up mode and up/down mode. (1pt)
2. What is the advantages and disadvantages of using CCR0 with up mode for frequency generation compared with using the continuous mode? (1pt)

Discussion Questions

1. What Timer A clock frequency and counter mode would you use to generate 10Hz and 5Hz frequencies? What clock source will you use and how? Justify your answer. (3pt)
2. Read the following material on TAIV, Timer A Interrupt Vector Register. Explain in your own words what does TAIV do and how can we use TAIV to figure out the interrupt source in multi-sourced interrupt. (2pt)

Handling of Multi-sourced Interrupt

When the timer A module is running in continuous mode, we can actually use all three CCR channels for frequency generation. However CCR1 and CCR2 share the same
interrupt vector. Therefore you need to use the TAI0V, Timer A Interrupt Vector Register, to figure out which CCR is currently throwing the interrupt.

The way TAI0V works is when interrupts are thrown, the highest priority interrupt generates a number to be stored in the TAI0V register. Any reads from TAI0V register will automatically clear the highest priority interrupt flag and clear the TAI0V value. After that again the highest priority interrupt happens will set the TAI0V values. A table showing the TAI0V contents, interrupt source, corresponding flag and interrupt priority can be found below (therefore you do not need to clear the interrupt flags yourself if you use TAI0V).

<table>
<thead>
<tr>
<th>TAI0V Contents</th>
<th>Interrupt Source</th>
<th>Interrupt Flag</th>
<th>Interrupt Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>No interrupt pending</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>02h</td>
<td>Capture/compare 1</td>
<td>TACCR1 CCIFG</td>
<td>Highest</td>
</tr>
<tr>
<td>04h</td>
<td>Capture/compare 2</td>
<td>TACCR2 CCIFG</td>
<td></td>
</tr>
<tr>
<td>06h</td>
<td>Reserved</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>08h</td>
<td>Reserved</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0Ah</td>
<td>Timer overflow</td>
<td>TAI0G</td>
<td></td>
</tr>
<tr>
<td>0Ch</td>
<td>Reserved</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0Eh</td>
<td>Reserved</td>
<td>-</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

We can quickly evaluate the interrupt source by adding TAI0V to the program counter. An example is as shown below:

```
ADD &TAIV, PC       ; Add offset to Jump table
RETI                ; TAI0V = 0: No interrupt source defined
JMP CCIFG_1_HND     ; TAI0V = 2: TACCR1
JMP CCIFG_2_HND     ; TAI0V = 4: TACCR2
RETI                ; Vector 6: No interrupt source defined
RETI                ; Vector 8: No interrupt source defined
JMP TAI0G_HND       ; Vector 10: TAI0G, Timer A interrupt
RETI                
TAIFG_HND
...                ; Task for Timer A interrupt starts here
RETI                

CCIFG_2_HND
...                ; Vector 4: TACCR2
...                ; TACCR2 interrupt task starts here
RETI                ; Back to main program

CCIFG_1_HND
...                ; Vector 2: TACCR1
...                ; TACCR1 interrupt task starts here
RETI                ; Back to main program
```

Note that to use Timer A1, you should change the TAI0V to TA1IV.