Lab 10

Advanced Timer Features

In this lab, we will learn using multiple channels of the timer module that enable timing concurrent durations. We will also learn timer-based output and use it to implement a Pulse Width Modulation (PWM) signal that controls the brightness of the LED. Finally, we will learn using timer-based input to create timestamps on events.

10.1 Timer's Multiple Channels

The Timer_A module has multiple channels that allow timing recurring intervals simultaneously. For example, a Timer_A module with three channels, referred to as Timer_A3, is capable of timing three recurring durations simultaneously based on the one counting register (TAR). A Timer_A module with five channels is referred to as Timer_A5. MSP430 MCUs often have multiple instances of Timer_A with multiple channels each. For example, an MCU may have two modules with three channels each and are referred to as Timer0_A3 and Timer1_A3, one being module #0 and the other module #1. The features of our chip are summarized in the data sheet `(slas789c)` on p. 7. This page shows that our MCU has four Timer_A modules; one module has two channels, two modules have three channels each and one module has five channels.

A duration is timed by using Timer_A's compare events. As an example, let's use Channel 0 to get an interrupt 4000 cycles from now. Channel 0 has a 16-bit register called TACCR0. We write TACCR0=4000-1 and we start TAR at 0. When TAR reaches the value in TACCR0, the compare event of Channel 0 triggers an interrupt. Similarly, we can time simultaneous intervals on Channels 1 and 2 by using their respective registers TACCR1 and TACCR2.

To time simultaneous intervals with multiple channels, the timer module is operated in the continuous mode, in which TAR counts from 0 up to 65,535 (64K). The channels schedule their interrupts by looking ahead from the current value of TAR, as illustrated in Figure 10.1.

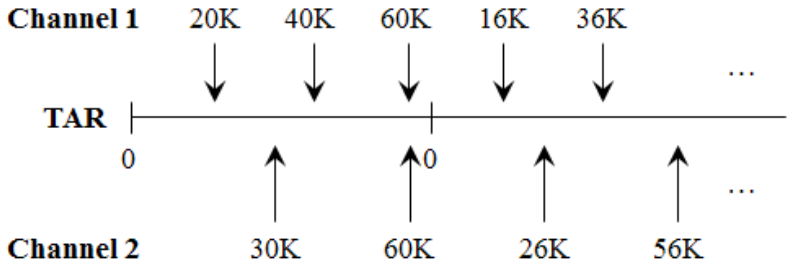

Figure 10.1: Using two channels of Timer_A

In the figure, Channel 1 is scheduling periodic interrupts every 20K cycles (K=1024). Each time an interrupt occurs, the register of Channel 1 is advanced by 20K cycles in order to schedule the next interrupt. What happens after 60K knowing that TAR and the channels' register are 16-bit? When the code adds 20K to Channel 1's register, the following math occurs:

```
60K + 20K = 80K (on 16-bit; leftmost bit worth 64K is dropped)
= 80K - 64K = 16K
```

The answer is 16K since the hardware is doing a 16-bit addition. It turns out that this answer will result in the correct duration for the next interval. To go from 60K to 16K, TAR counts up to 64K (that's 4K cycles), then rolls back to zero and counts up to 16K for a total of 20K. Therefore, this overflow operation is harmless and we can simply keep adding 20K to Channel 1's register.

Let's validate the operation of Channel 2 in the figure, which is setting up periodic interrupts every 30K cycles. After 60K, the next interrupt is schedule at: 60K + 30K (on 16-bit) = 90K - 64K = 26K. From 60K, TAR counts up to 64K (that's 4K cycles), then rolls back to zero and counts up to 26K, for a total of 30K cycles.

Based on this approach, the timer can support multiple channels that work simultaneously. Note that Channel 0 is a special channel in the up mode since it sets the upperbound of TAR. However, Channel 0 doesn't have any special responsibility in the continuous mode and can be used interchangeably with the other channels.

Interrupt Events of Timer_A

Timer_A has multiple interrupt events that are summarized in Table 10.1. The first column describes the event and the second column shows the trigger. The third column shows the enable and flag bits and the register in which these bits are located. Finally, the rightmost column shows the vector associated with each event. The rollback-to-zero event and Channels 1 and 2 share the A1 vector. Therefore, the interrupt flags of these events are cleared by the software. On the other hand, Channel 0 exclusively uses the A0 vector. Therefore, Channel 0's interrupt flag is cleared by the hardware.

Table 10.1: Multiple Interrupt Events of Timer_A

Event	Trigger	Bits	Vector
Rollback-to-zero	TAR = 0	TAIE / TAIFG in TACTL	A1
Channel 0 Compare Event	TAR = TACCR0	CCIE / CCIFG in TACCTL0	A0
Channel 1 Compare Event	TAR = TACCR1	CCIE / CCIFG in TACCTL1	A1
Channel 2 Compare Event	TAR = TACCR2	CCIE / CCIFG in TACCTL2	

Flashing Two LEDs using Two Channels

Write a code that sets up two periodic interrupts and toggles two LEDs when the interrupts occur. Use ACLK based on the 32 KHz crystal and divide it by four. Set up Channel 0 so that it raises periodic interrupts every 0.1 seconds and toggles the red LED and set up Channel 1 so that it raises periodic interrupts every 0.5 seconds and toggles the green LED. Engage a suitable low-power mode. Complete the missing parts of the code.

```
#include <msp430fr6989.h>
#define redLED BIT0          // Red at P1.0
#define greenLED BIT7        // Green at P9.7

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;        // Stop WDT
    PM5CTL0 &= ~LOCKLPM5;            // Enable GPIO pins

    P1DIR |= redLED;
    P9DIR |= greenLED;
    P1OUT &= ~redLED;
    P9OUT &= ~greenLED;

    // Configure Channel 0
    TA0CCR0 = ...                    // Find # cycles
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;
```

```
    // Configure Channel 1
    ...

    // Start the timer (divide ACLK by 4)
    ...

    // Engage a low-power mode
    ...

    return;
}

// ISR of Channel 0 (A0 vector)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    P1OUT ^= redLED;                // Toggle the red LED
    TA0CCR0 += 3277;                // Schedule the next interrupt
    // Hardware clears Channel 0 flag (CCIFG in TA0CCTL0)
}

// ISR of Channel 1 (A1 vector)
#pragma vector = ...                // fill the vector name
__interrupt void T0A1_ISR() {
    ...
}
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Do a visual comparison to your phone's stopwatch and check that the durations appear to be correct.
- Show how the number of cycles for each channel is derived.
- Submit the code in your report.

10.2 Using Three Channels

Modify the code of the previous part so that it implements the pattern shown in Figure 10.2. Channels 0 and 1 toggle the red LED and green LED every 0.1 seconds and 0.5 seconds, respectively, just like in the previous part. In addition, Channel 2 sets up periodic interrupts every 4 seconds. These interrupts halt and resume the flashing as shown in the figure. For four seconds, the LEDs are flashing, each at its respective rate, then for the next four seconds, the LEDs are off.

In the code, the ISR of the A1 vector services the interrupt events of Channels 1 and 2. Therefore, the ISR needs to detect which of the two interrupts has occurred, as shown in the code excerpt below.

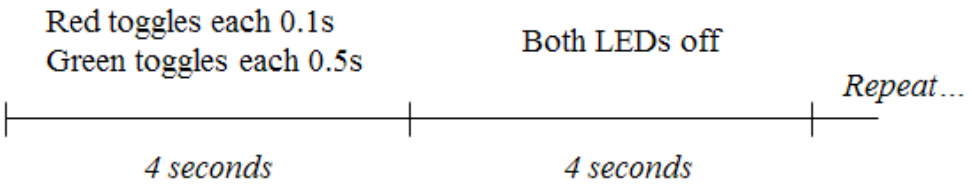

Figure 10.2: Using three channels of Timer_A

```
// ISR of A1 vector
#pragma vector = ...
__interrupt void T0A1_ISR() {
    // Detect Channel 1 interrupt (check the flag)
    if((TA0CCTL1 & CCIFG)!=0) {
      ...
    }

    // Detect Channel 2 interrupt
    if(...) {
      ...
    }
}
```

In this code, it is helpful to declare a status variable that keeps track whether the LEDs are flashing or are currently off. The status variable is used at the 4-second transition point.

One requirement in this code is that the interrupts of Channels 1 and 2 must be turned off during the off period. This means the xIE bits of Channels 1 and 2 are changed to zero during the off period and are re-enabled at the end of the four-second interval. One thing we would need to consider here is when is the best time to clear the flag: when an interrupt is disabled or when an interrupt is re-enabled? Since we want to ignore events (flags being raised) during the off period, the flags should be cleared when the interrupts are re-enabled.

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Do a visual comparison to your phone's stopwatch and check that the durations appear to be correct.
- Submit the code in your report.

10.3 Driving a PWM Signal on the Pin

The Timer_A module supports output patterns that can be driven on the pins and these patterns can be used to implement a PWM signal. Such output signals are driven directly by the timer without CPU intervention, therefore, the CPU can remain in low-power mode while the PWM is ongoing.

A PWM signal is shown in Figure 10.3. The signal's period is fixed and is marked by the vertical dashed lines. If the period is 0.001 second, then the PWM frequency is 1000 Hz. The relative duration of the high level within a period is described as the duty cycle. In the figure, the first pulse corresponds to a duty cycle of 25% and the second pulse corresponds to a duty cycle of 50%.

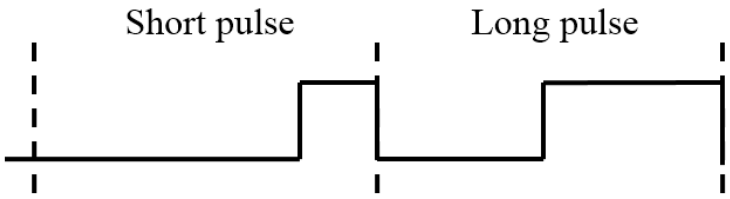

Figure 10.3: Pulse Width Modulation (PWM) Signal

A PWM signal can be used to drive a motor, in which case a higher duty cycle corresponds to a higher motor speed. In this lab, we'll drive a PWM signal on the pin to which the LED is connected. By modifying the duty cycle, multiple brightness levels appear on the LED. It's essential that the PWM frequency is high enough so that the user doesn't see the LED blinking. We'll set up a PWM frequency of 1000 Hz.

Configuring the Pin

Any pin on the MCU that is described as having the function TAx.y on the pinout diagram can be used to drive timer-based output. On our LaunchPad board, the red and green LEDs are connected to P1.0 and P9.7, respectively. Let's find out if these pins support timer-based output. Looking at the pinout diagram in the MCU's Data Sheet `(slas789c)` on p. 9, we find the pin of P1.0 described as the following.

```
P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VREF-/VeREF-
```

The TAx.y format we're looking for is shown above as TA0.1 This means that the pin can be used for output driven by Timer_A module #0 Channel 1. Since the red LED is connected to this pin, a PWM driven on this pin implements brightness levels on the red LED. Find out if P9.7 supports timer-based output.

The default functionality of all pins in MSP430 is GPIO. Therefore, the pin above is configured to P1.0 at reset. The pin can be reconfigured by looking at the data sheet tables `(slas789c)` on p. 96. This page shows the information below to configure the pin as TA0.1:

```
P1DIR bit = 1      P1SEL1 bit = 0      P1SEL0 bit = 1
```

The values shown above should be set in bit 0 since the overlapping GPIO pin is P1.0.

Output Patterns

The output patterns that are supported by the timer are described in the FR6xx User's Guide `(slau367o)` Chapter 25 on p. 649. We will use Reset/Set mode, which is mode #7. Its operation is illustrated in Fig-

ure 10.4.

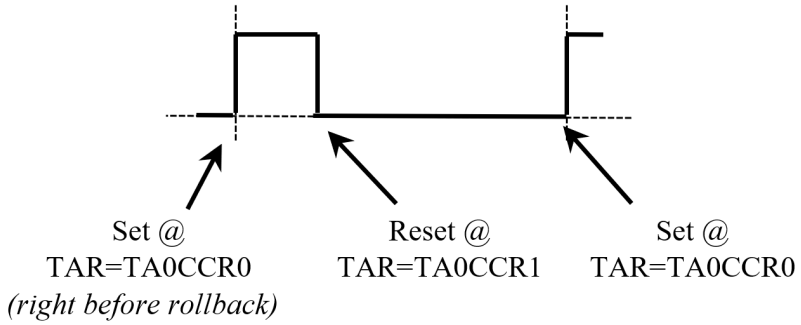

Figure 10.4: Reset/Set Output Pattern

We're aiming at setting up a PWM signal with a frequency of 1000 Hz. The period is 0.001 seconds and corresponds to 33 cycles based on a 32 KHz clock signal. Therefore, we'll run the timer in the up mode with a period of 33 cycles.

The output mode we're using has two actions (action1/action2), i.e., Reset/Set and is set up at Channel 1. The first action is triggered by Channel 1's compare event (i.e. when TAR=TACCR1) and the second action is triggered by Channel 0's compare event (i.e. when TAR=TACCR0=33)[1]. As shown in Figure 10.4, the vertical dashed lines mark the timer's period in which TAR counts from 0 to TACCR0=33. The figure's example corresponds to TACCR1=8. The Reset event is triggered by Channel 1's compare event when TAR=8 and the Set event is triggered by Channel 0's compare event when TAR=33, which happens right before TAR rolls back to zero. This corresponds to a duty cycle of 8/33 that's close to 25% as shown in the figure. The duty cycle can be increased by changing TACCR1 to a larger value, up to 33.

Activating an output mode (e.g. Reset/Set) at a channel is done by modifying the OUTMOD field in the channel's register (TACCTLx). This register's format can be found in the Family User's Guide (slau367o) in the Timer_A chapter (at the end of the chapter).

The code is shown below. Read the code to understand how it works and complete the two missing lines.

```
// Setting up a PWM on P1.0 (red LED)
// P1.0 coincides with TA0.1 (Timer0_A Channel 1)
// Configure P1.0 pin to TA0.1 ---> P1DIR=1, P1SEL1=0, P1SEL0=1
// PWM frequency: 1000 Hz -> 0.001 seconds
#include <msp430fr6989.h>
#define PWM_PIN BIT0

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;                // Stop WDT
```

[1]When the continuous mode is used, the second action is triggered by the rollback-to-zero event.

```
    PM5CTL0 &= ~LOCKLPM5;

    // Configure pin to TA0.1 functionality (complete last two lines)
    P1DIR |= PWM_PIN;               // P1DIR bit = 1
    P1SEL1 ...                      // P1SEL1 bit = 0
    P1SEL0 ...                      // P1SEL0 bit = 1

    // Configure ACLK to the 32 KHz crystal
    config_ACLK_to_32KHz_crystal();

    // Configure Channel 1 for Reset/Set mode (Mode #7)
    ...                             // Modify OUTMOD field to 7
    TA0CCR1 = 1;                    // Modify this value between 0 and
                                    // 32 to adjust the brightness level

    // Starting the timer in the up mode; period = 0.001 seconds
    // (ACLK @ 32 KHz) (Divide by 1) (Up mode)
    TA0CCR0 = 33;   // @ 32 KHz --> 0.001 seconds (1000 Hz)
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

    for(;;) {}
    return;
}
```

Once you finish the code, test for multiple values of TA0CCR1 and these should result in different brightness levels. Incorporate the joystick in your code so that moving the joystick left and right decreases and increases the LED's brightness, respectively. Moving the joystick up should set maximum brightness immediately and moving the joystick down should set minimum brightness immediately. A video demo can be found at the link below.

https://youtu.be/cUkwFgXNnlA

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Vary the value of TA0CCR1 between 1 and 32 and ensure this results in various brightness levels.
- Submit the code in your report.

10.4 Timer Input Capture

The timer supports input capture known as the Capture Mode. In this mode, the timer observes when a change occurs at a pin (a rising edge, a falling edge or either) and saves TAR into the register of a timer channel (e.g. TACCR1). The value of TAR serves as a timestamp that indicates when the change was

observed at the pin.

Similar to timer-based output, timer-based input capture is available at pins that are described as having the functionality TAx.y in the pinout diagram. The button S1 on our board is connected to P1.1 which has the following description in the pinout diagram. Based on this description, the pin can be used for input capture with the functionality TA0.2, which corresponds to Timer_A module #0 Channel 2.

```
P1.1/TA0.2/TA1CLK/COUT/A1/C1/VREF+/VeREF+
```

Follow the steps below to write a code that performs input capture when the button is either pushed or released and transmits the timestamps to the PC over UART.

Step 1) Configure the pin to TA0.2 functionality by looking at the table in the chip's data sheet on p. 96. You're looking the set up the functionality TA0.CCI2A (note that it's input A rather than B as we'll need this information later on). Write down the values of P1DIR, P1SEL1 and P1SEL0 and set the corresponding bit values in your code.

Step 2) Configure Channel 2 of Timer #0 for the capture mode. This is done using this channel's register TA0CCTL2. The format of this register is in the Family User's Guide, Timer_A chapter, on p. 657. These are the fields you need to set: CM (capture on both edges), CCIS (capture input A), CAP (set the channel to capture mode), CCIE (to trigger an interrupt when a capture occurs).

Step 3) Start Timer #0 in the continuous mode based on the 32 KHz crystal. Therefore, the timestamps will be in the range 0-65,535.

Step 4) Write the ISR of Timer #0 Channel 2. In the ISR, read the timestamp, which will be in Channel 2's register (TA0CCR2), and transmit it over UART to the PC. You can use a delay loop to wait out the bounces, so they don't transmit a new timestamp. It's also a good idea to clear the flag after the delay loop when all the bounces have elapsed.

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- How long does a button push last? Give the answer as clock cycles and as a duration.
- Submit the code in your report.

Student Q&A

1. Copy the description of P9.7 from the pinout diagram and determine whether this pin supports timer-based output.

2. In the code with three channels, why was it necessary to divide ACLK?

3. In the first part, we configured two periodic interrupts using two channels of the timer. Is this approach scalable? For example, using a Timer_A module with five channels, can we configure five

periodic interrupts? Explain and mention in what mode the timer would run.

4. As an example, Channel 1's interrupt occurs every 40K cycles. The first interrupt is scheduled for when TAR=40K cycles. Explain how the next interrupt is scheduled? Explain the overflow mechanism and show why it results in a correct value.