EEL4742C: Embedded Systems

Experiment 6:

Advanced Timer Methods

Ryan Koons

11/16/2020

Introduction: This lab utilizes advanced timing methods for the MSP430 such as multiple channels, timers, and generating PWM signals. For using periodic interrupts, up mode is the best choice since the overflow is automatically handled and interrupts remain scheduled on time. To do so each channel is defined a different TACCRx based on its individual period and number of cycles; then, this value needs to be incremented within the interrupt-again the ISR handles the overflow. Make sure to clear the flag whenever the hardware doesn't do so automatically. A big part of this lab is concerned with scheduling and enabling/disabling interrupts. The next part of the lab involves using multiple timers since one timer will be used for PWM, and the other for a one second toggle. The last part of the lab involves driving a PWM signal to an LED; this well be covered in more detail in latter sections.

6.1 Timing with Two Channels: The first section of Lab 6 required utilizing multiple channels to toggle two different LEDs at different rates. The basics of timer were covered in previous labs and will not be explained in great detail for this particular lab. Channel 0 was utilized for the red LED timing. TA0CCR0 was assigned a value of 3276. At 32kHz, this yields a value of 0.1 seconds, since 32768*0.1=3277. The external 32768 Hz crystal was utilized for interrupt timing in this lab. Then the CCIE and CCIFG were set and cleared respectively within the TA0CCTL0 register; these are used to enable the particular interrupt and clear the corresponding flag. Next, channel 1 was configured for the green LED timing. TA0CCR1 was assigned a value of 16384-1, to yield a 0.5 second delay at 32kHz; since, 32768*0.5=16384. Similarly, TA0CCTL1 CCIE and CCIFG bits were configured. The last step in the main was to configure TA0CTL (continuous mode, divide by 1, and ACLK select), and activate low power mode 3.

The rest of the code's functionality was handled within the ISR for timer A0 and A1. Within the A0 Vector, every time this is ISR triggered, the red LED was toggled and TA0CCR0 was incremented a value of 3277 to schedule the next interrupt. A similar event takes place for the green LED. Except, it now increments TA0CCR1 by a value of 16384, since it is triggering interrupts at a different rate from the other channel. It is also important to note that flag must be cleared for this ISR since it is shared by other vectors.

Analysis:

6.1 CCS Code:

```c
// Using Timer_A with 2 channels
// Using ACLK @ 32 KHz (undivided)
// Channel 0 toggles the red LED every 0.1 seconds
// Channel 1 toggles the green LED every 0.5 seconds
#include <msp430fr6989.h>
#define redLED BIT0 // Red at P1.0
#define greenLED BIT7 // Green at P9.7

void convert_32khz();

void main(void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
PM5CTL0 &= ~LOCKLPM5; // Enable GPIO pins
P1DIR |= redLED;

P9DIR |= greenLED;
P1OUT &= ~redLED;
P9OUT &= ~greenLED;
convert_32khz();

// Configure Channel 0
TA0CCR0 = 3277-1; // @ 32 KHz --> 0.1 seconds
TA0CCTL0 |= CCIE;
TA0CCTL0 &= ~CCIFG;
// Configure Channel 1 (write 3 lines similar to above)
TA0CCR1 =16384-1; //@ 32,768 Hz--> 0.5s
TA0CCTL1 |= CCIE;
TA0CCTL1 &= ~CCIFG;

// Configure timer (ACLK) (divide by 1) (continuous mode)
TA0CTL = TASSEL_1 | ID_0 | MC_2;

// Engage a low-power mode, automatically enables GIE
_low_power_mode_3();
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) ... fill the vector name below
#pragma vector = TIMER0_A1_VECTOR
__interrupt void T0A1_ISR() {
 // Toggle the green LED
    P9OUT ^= greenLED;
 // Schedule the next interrupt
    TA0CCR1 += 16384;
 TA0CCTL1 &= ~CCIFG;
}
```

```
//Function for Converting ACLK to 32Khz
void convert_32khz()
{
//By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz
//Reroute pins to LFXIN/LFXOUT functionality
PJSEL1 &= ~BIT4;
PJSEL0 |= BIT4;
//wait until oscillator fault flags are cleared
CSCTL0 = CSKEY; //unlock CS registers
do {
CSCTL5 &= ~LFXTOFFG; //local fault flag
SFRIFG1 &= ~OFIFG; //global fault flag
} while((CSCTL5 & LFXTOFFG) != 0 );
CSCTL0_H = 0; //Lock CS registers
return;
}
```

6.2 Timing with Three Channels: This section of the lab required re-using the 6.11 code but also utilizing another channel with a 4-second delay. This third channel would then alternate between allowing the LEDs to blink for 4 seconds, and then not. In order to achieve a 4 second delay, clock division needed to be utilized; A clock divider of 4 was selected to yield a sufficient number of cycles (based on 16 bit integer size) for a 4 second delay. However, this meant that the values for TA0CCR0 and TA0CCR1 need to be recalculated. For channel 0: 8192*0.1= 820. So, a value of 820-1 was set. For channel 1: 8192*0.5= 4096. And for channel 3 8192*4=32768. The interrupt enable and interrupt flag bits also had to be configured within TA0CCR2.

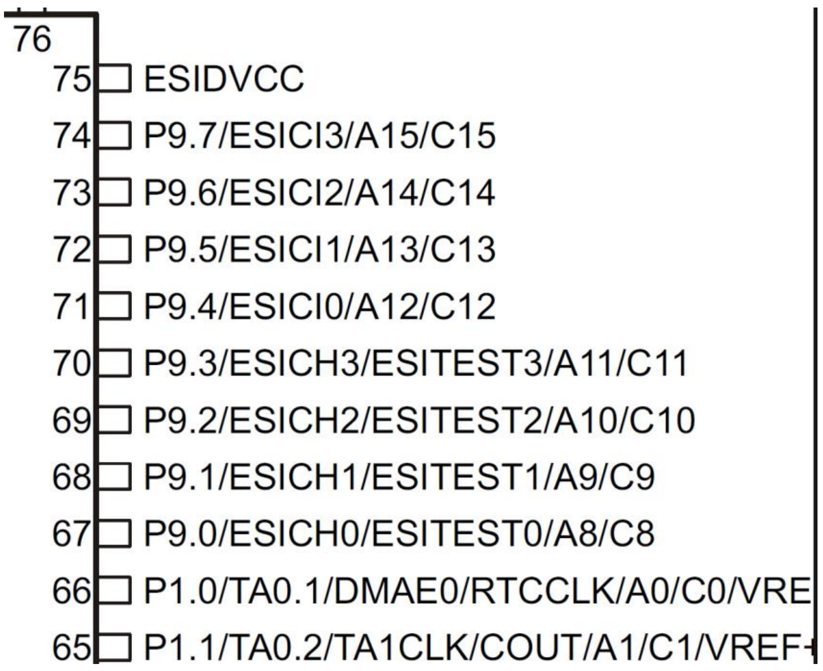
The next difference in the code lies within the ISR's. A global variable "state" was defined so that the ISR can determine whether or not to blink the LEDs. The A1 Vector is identical to the previous section. For the green LED toggle, an if statement had to be added to make sure that the flag that triggered the ISR call was in TA0CCTL2 and not TA0CCTL3. The actual code for the green LED toggle is again the exact same. Then the code, checks to see if the TA0CCTL3 was raised, it then checks the value of state. If state is 1 (default) then the LEDs stop blinking, interrupt are disabled for both channel 1 and 0, and the state variable is toggled to 0. Then an else if statement executes. When this is selected, TA0CCR0 and TA0CCR1 are given a value of TA0R + their original TA0CCRx. Then, the interrupt flags are cleared, and the interrupt enable bits are set, and state is toggled to 1. The last step, is to increment TA0CCR2 for its next scheduling and clear the flag.

6.2 CCS Code

```c
// Using Timer_A with 2 channels
// Using ACLK @ 32 KHz (undivided)
// Channel 0 toggles the red LED every 0.1 seconds
// Channel 1 toggles the green LED every 0.5 seconds
#include <msp430fr6989.h>
#define redLED BIT0 // Red at P1.0
#define greenLED BIT7 // Green at P9.7

void convert_32khz();

static int state=1; //state variable for 4 second toggle

void main(void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
PM5CTL0 &= ~LOCKLPM5; // Enable GPIO pins
P1DIR |= redLED;

P9DIR |= greenLED;
P1OUT &= ~redLED;
P9OUT &= ~greenLED;
convert_32khz();

// Configure Channel 0
TA0CCR0 = 820-1; // @ 8,192 KHz --> 0.1 seconds
TA0CCTL0 |= CCIE;
TA0CCTL0 &= ~CCIFG;
// Configure Channel 1 (write 3 lines similar to above)
TA0CCR1 =4096-1; //@ 8,192 Hz--> 0.5s
TA0CCTL1 |= CCIE;
TA0CCTL1 &= ~CCIFG;

// Configure Channel 2 (write 3 lines similar to above)
TA0CCR2 =32768-1; //@ 8,192 Hz--> 4.0s
TA0CCTL2 |= CCIE;
TA0CCTL2 &= ~CCIFG;


// Configure timer (ACLK) (divide by 4) (continuous mode)
TA0CTL = TASSEL_1 | ID_2 | MC_2| TACLR;

// Engage a low-power mode, automatically enables GIE
_low_power_mode_3();
return;
}

// ISR of Channel 0 (A0 vector)
#pragma vector = TIMER0_A0_VECTOR
```

```
__interrupt void T0A0_ISR() {
P1OUT ^= redLED; // Toggle the red LED
TA0CCR0 += 820; // Schedule the next interrupt
// Hardware clears Channel 0 flag (CCIFG in TA0CCTL0)
}

// ISR of Channel 1 (A1 vector) ... fill the vector name below
#pragma vector = TIMER0_A1_VECTOR
__interrupt void T0A1_ISR() {
    if((TA0CCTL1 & CCIFG) != 0)
    {
 // Toggle the green LED
    P9OUT ^= greenLED;
 // Schedule the next interrupt
    TA0CCR1 += 4096;
 TA0CCTL1 &= ~CCIFG;
    }
    if((TA0CCTL2 & CCIFG) != 0)
    {
       if(state==1)
       {
           //turn off LEDs
           P1OUT &= ~redLED;
           P9OUT &= ~greenLED;
           //disable interrupts
           TA0CCTL0 &= ~CCIE;
           TA0CCTL1 &= ~CCIE;
           state=0;

       }

       else if(state==0)
       {
            //restore interrupts and clear flags, restore channels
           TA0CCR0 = TA0R + 820;
           TA0CCR1 = TA0R + 4096;
           TA0CCTL0 &= ~CCIFG;
           TA0CCTL1 &= ~CCIFG;
           TA0CCTL0 |= CCIE;
           TA0CCTL1 |= CCIE;
           state=1;
       }

       TA0CCR2 += 32768;
       TA0CCTL2 &= ~CCIFG;

    }
}

//Function for Converting ACLK to 32Khz
void convert_32khz()
{
//By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz
//Reroute pins to LFXIN/LFXOUT functionality
PJSEL1 &= ~BIT4;
PJSEL0 |= BIT4;
//wait until oscillator fault flags are cleared
CSCTL0 = CSKEY; //unlock CS registers
do {
CSCTL5 &= ~LFXTOFFG; //local fault flag
SFRIFG1 &= ~OFIFG; //global fault flag
} while((CSCTL5 & LFXTOFFG) != 0 );
CSCTL0_H = 0; //Lock CS registers
return;
}
```

6.3 Basic PWM with LED: This section of the code required utilizing the timer and driving a PWM signal to the red LED on the MSP430 Launchpad. The first step is the necessary pin diversion to designate the PWM signal for the red LED. To do so the corresponding bit for P1.0 was set in the P1DIR register, cleared in the P1SEL1 register, and set in the P1SEL0 register. These GPIO pin diversions were all found in the chip's datasheet. It is worth noting that prior to this, it had to be determined if the red LED could be used for PWM; so, the pinout diagram was consulted. Red had TA0.1 next to it which meant it can be used, whereas, the green LED did not and can therefore not be used. The external 32 kHz crystal was once again used for this section of the lab.TA0CCR0 was assigned a value of 33-1. At 32Khhz, this yields a 1000Hz toggle which is fast enough for the user not to detect any change in brightness. TA0CTL was then configured in up mode, and TA0CCTL1 was assigned OUTMOD_7 which selected the reset set mode which would be used to drive the preset PWM signal with concurrent events. Then, TA0CCR1 was assigned a value of 2 to make the change in brightness from a standard led illumination. This value can be set anywhere from 0-32, and the larger value corresponds with a brighter led.


76
75 ESIDVCC
74 P9.7/ESICI3/A15/C15
73 P9.6/ESICI2/A14/C14
72 P9.5/ESICI1/A13/C13
71 P9.4/ESICI0/A12/C12
70 P9.3/ESICH3/ESITEST3/A11/C11
69 P9.2/ESICH2/ESITEST2/A10/C10
68 P9.1/ESICH1/ESITEST1/A9/C9
67 P9.0/ESICH0/ESITEST0/A8/C8
66 P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VRE
65 P1.1/TA0.2/TA1CLK/COUT/A1/C1/VREF

6.3 CCS Code

```
// Using Timer_A with 2 channels
// Using ACLK @ 32 KHz (undivided)
// Channel 0 toggles the red LED every 0.1 seconds
// Channel 1 toggles the green LED every 0.5 seconds
#include <msp430fr6989.h>
#define redLED BIT0 // Red at P1.0
#define greenLED BIT7 // Green at P9.7

void convert_32khz();

void main(void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
PM5CTL0 &= ~LOCKLPM5; // Enable GPIO pins
P1DIR |= redLED;

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

//Pin diversion for Red LED--PWM
P1DIR |= BIT0;
P1SEL1 &= ~BIT0;
P1SEL0 |= BIT0;
convert_32khz();

//Start Timer in up mode
TA0CCR0= 33-1; // @32Khz, 0.001 seconds (1000 Hz)
//ACLK, Divide by 1,
TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

//Configure Channel 1 for PWM
TA0CCTL1 |= OUTMOD_7; //Reset Set Output Pattern
TA0CCR1 = 2;  //change brightness of LED, 0-32
while(1){}
return;
}


//Function for Converting ACLK to 32Khz
void convert_32khz()
{
//By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz
//Reroute pins to LFXIN/LFXOUT functionality
PJSEL1 &= ~BIT4;
PJSEL0 |= BIT4;
//wait until oscillator fault flags are cleared
CSCTL0 = CSKEY; //unlock CS registers
do {
CSCTL5 &= ~LFXTOFFG; //local fault flag
SFRIFG1 &= ~OFIFG; //global fault flag
} while((CSCTL5 & LFXTOFFG) != 0 );
CSCTL0_H = 0; //Lock CS registers
return;
}
```

6.4 PWM with an Additional Timer: This code is similar to 6.3, but utilizes another timer and channel to toggle the LED brightness. TA1CCR0 is assigned a value f 32768-1, which at 32KHz yields a 1.0 second delay. This delay will be used to iterate through the different LED brightness levels. The interrupt enable and flag bits are also configured. Then TA1CTL is configured for ACLK and up mode. The rest of the code is an ISR that toggles the green LED (to indicate brightness level change) and as long as TA0CCR1 is less than 30, it increments it by five. Otherwise, it assigns a value of 0. This cause TA0CCR1 to cycle from 0 to 30 at five segment increments. The two timer modules are both being driven by the same eternal crystal clock; however, they have completely different channels and different TAxCCRx values that can capture and compare different values. Each timer can be configured in different modes, with different dividers, clocks, etc. However, in this example this was not used.

6.4 CCS Code

```
// Using Timer_A with 2 channels
// Using ACLK @ 32 KHz (undivided)
// Channel 0 toggles the red LED every 0.1 seconds
```

```c
// Channel 1 toggles the green LED every 0.5 seconds
#include <msp430fr6989.h>
#define redLED BIT0 // Red at P1.0
#define greenLED BIT7 // Green at P9.7

void convert_32khz();

void main(void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
PM5CTL0 &= ~LOCKLPM5; // Enable GPIO pins
P1DIR |= redLED;

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

//Pin diversion for Red LED--PWM
P1DIR |= BIT0;
P1SEL1 &= ~BIT0;
P1SEL0 |= BIT0;
convert_32khz();

//Start Timer in up mode
TA0CCR0= 33-1; // @32Khz, 0.001 seconds (1000 Hz)
//ACLK, Divide by 1,
TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

TA1CCR0=32768-1; //@32kHz, 1.0 second delay
TA1CCTL0 |= CCIE;
TA1CCTL0 &= ~CCIFG;
TA1CTL = TASSEL_1 | ID_0 |MC_1 |TACLR;

//Configure Channel 1 for PWM
TA0CCTL1 |= OUTMOD_7; //Reset Set Output Pattern
TA0CCR1 = 0;  //change brightness of LED, 0-32

// Engage a low-power mode, automatically enables GIE
_low_power_mode_3();
return;
}

// ISR of Timer 1 Channel 0 (A0 vector)
#pragma vector = TIMER1_A0_VECTOR
__interrupt void T1A0_ISR() {
 // Toggle the green LED
    P9OUT ^= greenLED;
if(TA0CCR1< 30 )
    TA0CCR1+=5;
else TA0CCR1=0;
}


//Function for Converting ACLK to 32Khz
void convert_32khz()
```

```
{
//By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz
//Reroute pins to LFXIN/LFXOUT functionality
PJSEL1 &= ~BIT4;
PJSEL0 |= BIT4;
//wait until oscillator fault flags are cleared
CSCTL0 = CSKEY; //unlock CS registers
do {
CSCTL5 &= ~LFXTOFFG; //local fault flag
SFRIFG1 &= ~OFIFG; //global fault flag
} while((CSCTL5 & LFXTOFFG) != 0 );
CSCTL0_H = 0; //Lock CS registers
return;
}
```

Student Q&A:

1) ACLK was divided because otherwise at 32Khz, the program would've required a TACCRx value of 128,000+ which is much larger than the 65535 16-bit integer allowance.
2) Yes, this is scalable, as long as the ISR is properly configured. Each channel has its own interrupt enable and interrupt flag bit. So, each channel can properly trigger an interrupt; however, each interrupt will create some latency so lots of periodic interrupts will be slightly inaccurate, though still operational. Periodic interrupts are best in up mode, since the code handles the overflow and the ISR can be used to individually schedule each interrupt.
3) (rounding to 64,000 max) The next interrupt will occur at 16k cycles. The code handles the overflow, so it takes 24k cycles to reach overflow and from there take 24k more cycles. 24k+16k= 40, and the TAR makes it so that overflow works correctly and interrupts are scheduled on time. Also, 40+40-64=16(k).

Conclusion: This was lab was certainly the best way to become comfortable with the timing mechanisms of the MSP430. After this lab, the difference between all of timers, modules, channels, and registers is now understood. The MSP has different timers that can be configured completely differently; each timer is composed of channels that are capable of capturing/comparing certain values (up mode). These channels of course also have interrupt enable and flag bits. Scheduling was a very big portion of this lab; scheduling is usually done by assigning a value to TACCRx and then incrementing it during polling or during the ISR. The overflow is automatically taken care of during this process. This lab also covered the preset modes for driving PWM signals in the MSP430. These pins need to first be investigated for compatibility using the pinout diagram; then the pin needs to be diverted (GPIO diversion). The last step is to organize the TACCRx value, and configure the register. After this lab, it will certainly be much easier to perform any tasks that involve the timer.