Lab 3

Timer Module

In this lab, we will program the microcontroller's timer module (Timer_A) to time events. We will use the timer in two modes known as the continuous mode and the up mode. Our codes will use the polling technique, which is a simple approach based on reading a flag continuously.

3.1 The Continuous Mode

The MSP430FR6989 chip contains a timer module known as Timer_A. The timer module is versatile and supports many features. In this lab, we will use the module to time durations.

The timer uses a clock signal of a known frequency as input and counts cycles to time durations. Timer_A has a 16-bit register called TAR (Timer_A Register). This register counts up by one at each cycle of the clock signal. Since TAR is 16-bit, it can count in the range [0 - 65,535].

In the continuous mode, TAR counts from 0 to the largest value of 65,535, then rolls back to zero and continues counting, as shown below.

```
TAR:    0, 1, 2, ..., 65535, 0, 1, 2, ..., 65535, 0, 1, ...
                             ↑                    ↑
                         TAIFG set            TAIFG set
```

Each time TAR rolls back to zero **while counting**, the 1-bit flag TAIFG (Timer_A Interrupt Flag) is set (changed to 1) automatically by the hardware, as shown above. If TAR is set to zero through a line of code (i.e. TAR did not roll back to zero while counting), TAIFG is not set. The software acts based on the flag which can be done in two ways. The software can simply poll the flag (read it continuously), which is the approach that we will do in this lab. Alternatively, the software can set up an interrupt mechanism which relieves the burden of having to poll the flag (we'll do this in a future lab). Either way, when the flag is raised, the software takes action (e.g. toggle an LED), and clears the flag so it can be raised again. If the flag is not cleared, it remains high and wouldn't indicate anymore when the timer period has elapsed.

Timer_A Configuration

The configuration registers of any module are found in the respective chapter in the Family User's Guide (`slau367o`) at the end of the chapter. That is, the Family User's Guide has a chapter on Timer_A and the end of the chapter shows all the registers that configure the timer and the possible values they can have. Below, we show the main configuration register of the timer.

Timer_A is started and configured using the TACTL (Timer_A Control) register, which format is shown in Table 3.1. The TASSEL field is used to select the clock signal used by the timer. ACLK (Auxiliary Clock) is typically configured to a 32 KHz crystal that is soldered on the board while SMCLK (Sub Master Clock) is generated from an RC oscillator that's inside the chip and is set to 1.000000 MHz by default. Its frequency can be modified via software. Once a clock signal is selected, it can be divided within the timer module by either 1, 2, 4 or 8, to slow down its frequency. The frequency division is done using the ID field.

Table 3.1: Timer_A Control Register (TACTL)

TASSEL	2-bit	Timer_A Source Select (selects the clock signal) (1: ACLK) (2: SMCLK)
ID	2-bit	Input Divider (divides the input clock frequency inside the timer) (0: Div by 1) (1: Div by 2) (2: Div by 4) (3: Div by 8)
MC	2-bit	Mode (0: Stop) (1: Up Mode) (2: Continuous Mode)
TACLR	1-bit	Timer_A Clear (sets TAR to 0 when asserted)
TAIFG	1-bit	Timer_A Interrupt Flag Raised when TAR rolls back to zero while counting

The MC (Mode) field is used to select the mode. By default, it's 0 and the timer is stopped so it doesn't draw current. We'll change it to 2 in this section to run the timer in the continuous mode. If the timer is running and we wish to stop it, we can change MC back to 0.

The bit TACLR (Clear) is used to force TAR to go to zero. If TAR is counting and TACLR is asserted, TAR goes to zero immediately and resumes counting. We usually assert this bit at the start to ensure TAR starts at zero.

Finally, the bit TAIFG is set to 1 by the hardware when TAR rolls back to zero while counting. The software will monitor this bit to know when the timer period has elapsed.

Masking Operations

To simplify accessing the bit fields inside TACTL, the header file defines masks that make our job easier. The mask TASSEL_1 has a value of 1 at the position of TASSEL. Similarly, the mask TASSEL_2 has a value of 2 at the position of TASSEL. The masks ID_0, ID_1, ID_2, ID_3 have the values 0, 1, 2, 3, respectively, at the position of ID. Similarly, there are two masks MC_1 and MC_2. Finally, there is mask TACLR that has 1 at the position of TACLR and a mask TAIFG that has 1 at the position of TAIFG.

When the mask is 1-bit, there's no underscore in the name of the mask. For example, a 1-bit field CAP will have a mask called CAP. But a 2-bit field RES will have masks called RES_0, RES_1, ... RES_3. These masks are defined in the .h file that we included in the code. One great benefit these masks provide is that we don't have to be aware of the bit position of the fields within the register.

As an example, let's configure Timer_A to use SMCLK, divided by 4, continuous mode, and TAR starting at zero. This is done with the following line of code:

```
TACTL = TASSEL_2 | ID_2 | MC_2 | TACLR;
```

The line of code above will set TAIFG to zero since it was not mentioned. Since flags are critical to correct operation, sometimes we elect to clear the flag again out of precaution, which is done as the following:

```
TACTL &= ~TAIFG;    // AND with inverse of mask to clear the bit
```

We note that when we do `TACTL=...` as in the first line, this intends to modify all the fields in TACTL. Any field not mentioned in the line of code will be set to zero. On the other hand, when we do `TACTL |=...` or `TACTL &=...`, only the fields mentioned will be affected and the other fields remain unchanged.

In the two lines of code above, we used the variable name TACTL. The microconroller chip contains multiple independent Timer_A modules, called Timer0_A, Timer1_A, etc. In this experiment, we'll program the Timer0_A, therefore, the variable name in our code will be TA0CTL.

Configuring ACLK to the 32 KHz Crystal

In this experiment, we want to use the ACLK clock based on the 32 KHz crystal. By default, ACLK is configured to a built-in RC oscillator running at a frequency of 4.8 MHz / 128 = 37.5 KHz. We will reroute ACLK to the 32 KHz crystal that's soldered on the LaunchPad board. This is done by calling the function below. Here is how it works. Looking at the LaunchPad user's guide (page 29), the schematic shows that the 32 KHz crystal is attached to pins that have dual functionality: LFXIN/PJ.4 and LFXOUT/PJ.5. The functionality we're looking for is LFXIN (Low-frequency crystal in) and LFXOUT (Low-frequency crystal out). Looking at the chip's data sheet (page 123), we can find the settings to configure the pins to the LFXIN/LFXOUT functionality. All that is required is setting PJSEL1 (Bit 4) to 0 and PJSEL0 (Bit 4) to 1.

Furthermore, when the crystal is started, we need to wait for it to settle. We'll do this using the local and global oscillator fault flags. When these flags are cleared and remain cleared, it means the crystal clock is ready for use. The whole configuration is shown in the function `config_ACLK_to_32KHz_crystal()` below. The function first configures the pins to the crystal functionality then waits on the fault flags to remain cleared.

```
//**********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
    // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

    // Reroute pins to LFXIN/LFXOUT functionality
    PJSEL1 &= ~BIT4;
    PJSEL0 |= BIT4;

    // Wait until the oscillator fault flags remain 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;
}
```

We note that if we don't call this function in the code, the default ACLK will be 37.5 KHz which can be easily confused with the crystal at 32 KHz.

Flashing the LED

Let's write a code that flashes the red LED based on a delay that's set by the timer. Use the ACLK clock signal and configure it to the 32 KHz crystal, do not apply a divider on the clock, run the timer in the continuous mode and clear TAR at the start. Below is the code template. Write the code.

```
// Flashing the LED with Timer_A, continuous mode, via polling
#include <msp430fr6989.h>
#define redLED BIT0              // Red LED at P1.0
#define greenLED BIT7            // Green LED at P9.7

void main(void) {
  // Stop the Watchdog timer
  ...

  // Unlock the GPIO pins
  ...

  // Configure the LEDs as output
  ...

  // Configure ACLK to the 32 KHz crystal (function call)
  ...

  // Configure Timer_A
  // Use ACLK, divide by 1, continuous mode, clear TAR
  TA0CTL = ...;

  // Ensure flag is cleared at the start
  TA0CTL &= ~TAIFG;

  // Infinite loop
  for(;;) {
      // Wait in this empty loop for the flag to raise
      while( ... ) {}

      // Do the action here
      ...
  }

}
```

Perform the following and include the answers in your report.

- Complete the code and demo it to the TA.
- Write an analysis showing what delay you expect to observe and show how you computed the delay.
- Measure the observed delay with your phone's stopwatch for at least 20 seconds. The timing should match closely since the crystal is accurate. Report if it matches.
- Write an analysis showing what delays we expect to observe if the clock is divided by 2, 4 or 8. Test these cases and report if they match what you expected.

3.2 The Up Mode

In the up mode, the upperbound of TAR is set to any value that we choose. TAR counts from zero up to the value of register TACCR0. The timer has multiple channels and each channel has its own register and comparator. The registers of Channels 0, 1, 2 are TACCR0, TACCR1, TACCR2, respectively. When the timer runs in the up mode, the register of Channel 0, TACCR0, becomes the upperbound of TAR. Therefore, the up mode is inherently linked to Channel 0. The timeline below shows how TAR counts in the up mode. When TAR reaches TACCR0, it rolls back to zero and continues counting. Similar to the continuous mode, when TAR rolls back to zero (while counting), the TAIFG flag is set by the hardware.

```
TAR:    0, 1, 2, ..., TACCR0, 0, 1, 2, ..., TACCR0, 0, 1, ...
                              ↑                     ↑
                           TAIFG set             TAIFG set
```

The register TACCR0 is 16-bit. To have a period of 100 cycles, we set TACCR0=100-1=99. TAR then counts between 0 and 99, spending one cycle at each count.

Since our microcontroller chip has multiple Timer_A modules (called Timer0_A, Timer1_A, etc), we'll use the term TA0CCR0 in the code to indicate that we're using Timer0_A Channel 0 register.

Let's write a code that runs the timer in the up mode and toggles the LED every 1 second. Configure the timer so that it uses ACLK based on the 32 KHz crystal, do not apply a divider, select the up mode and a number of cycles that corresponds to 1 second period.

The lines below show the main differences from the code of the earlier part. We need to set the number of cycles in the up mode. It's a good idea to do this before starting the timer. The remaining parts of the code are similar to the code of the previous part. We should wait on the TAIFG flag to raise when TAR rolls back to zero and then take action.

```
// Set timer period
TA0CCR0 = ...

// Timer_A: ACLK, div by 1, up mode, clear TAR
TA0CTL = ...
```

Perform the following and include the answers in your report.

- Write the code and demo it to the TA.
- Show how you computed the value of TA0CCR0.
- Compare the flashing time to your phone's stopwatch for at least 20 seconds and report if the timing matches.
- What value of TA0CCR0 achieves a delay of 0.1 seconds? Round up to the nearest integer and test this value.
- What value of TA0CCR0 achieves a delay of 0.01 seconds? Round up to the nearest integer and test this value. What do you observe?

3.3 Application: Signal Repeater

In this part, we will design a signal repeater that samples a pulse signal and replays it. Such a repeater can be used to combat signal fading when the transmission wire is too long or can be used to translate between voltage domains. We'll implement the approach of sample-then-replay, which means we're either sampling the input or we're replaying the signal, but not both at the same time.

The pulse signal arrives as input (emulated by the push button) and is transmitted as output on the red LED. Our software measures the duration of the pulse and replays the same duration on the red LED. The following are the criteria of the application. The timer should be off initially so it doesn't consume power and should be turned only when it's needed. When a signal arrives (emulated by the button being pushed), we start the timer in the continuous mode, set TAR to zero, and start counting the number of cycles for as long as the pulse lasts (button is being pushed). When the pulse ends, we stop the timer immediately and capture the value of TAR since it represents the duration of the pulse. At this point, we will turn on the red LED and immediately start the timer in the up mode with a period equal to the number of cycles that we measured. We poll the flag to figure out when the duration has elapsed. At that point, we turn off the red LED and stop the timer to save power.

The timer's TAR register is 16-bit, so we can declare a 16-bit variable (of type unsigned int) to copy TAR into. The maximum number of cycles that we can time is 65,536 cycles (or 64K cycles) since TAR is 16-bit. If the signal lasts beyond that, TAR rolls back to zero and we can figure this out by checking the rollback flag. In such case, we will consider the signal to be too long to be replayed and we'll turn on the green LED indicating an error message (Error: signal time too long). The green LED should remain on until button S2 is pushed, which turns off the green LED and the system becomes ready to sample a new signal.

Setup your code using ACLK based on the 32 KHz crystal and apply a divider of 1. Compute the largest delay that can be supported and test it. While developing this application, we can create a breakpoint in the code so that the debugging software shows us the number of cycles that has been measured. But the final code should not rely on such information as it will run in the normal mode. Finally, the timer should be off initially so it doesn't consume power and should remain off when no pulse is being observed.

The video at the link below shows a demo of the pulse repeater:

`https://youtu.be/ZJux_cjxh30`

Perform the following:

- Compute the maximum pulse delay that our code supports for all cases of the divider (1, 2, 4, 8) based on the 32 KHz crystal.
- Explain what the tradeoff is when we change between dividers.
- Demo your programs and submit your C code

Student Q&A

Submit the answers to the questions in your report.

1. So far, we have seen two ways of timing delays: using a delay loop and using Timer_A. Which approach provides more control and accuracy over the delays? Explain.

2. Explain the polling technique and how it's used in this lab.

3. Is the polling technique a suitable choice when we care about saving battery power? Explain.

4. If we write 0 to TAR using a line code, does TAIFG go to 1?

5. From what we have seen in this lab, which mode gives us more control over the timing duration: the up mode or the continuous mode?

6. In this lab, you were given a summary of the timer's main control register, TACTL, and the fields within. To practice reading the documentation, find this information in the Family User's Guide `(slau367o)` document and include a screenshot of TACTL's layout and the table describing the fields within it. This information can be found at the end of the Timer_A chapter in the Family User's Guide.