ECE 2049 LECTURE 11

TODAY

- TIMERS PART 2

OFFICE HOURS

- TODAY: 4-6PM (NICK)
- TOMORROW: 3-5PM (RUSNI)
- FRIDAY: EMAIL ME!

ADMINISTRIVIA

- HW5: OUT AFTER CLASS, DUE THURS
  - CONTAINS SHORT SURVEY - PLEASE DO ASAP — YOUR FEEDBACK WILL HELP ME SHAPE THE REST OF THE COURSE!
- LAB 2: DEADLINE POSTPONED — DETAILS TOMORROW

CONTINUING FROM LAST TIME...

Using this information, we can write the register configuration:

Parameters we know	Relevant register field
CLOCK SOURCE: ACLK	TASSEL = 1
COUNTING MODE: UP	MC = 1
DIVIDER = DIVIDE BY 1	ID = 0
MAX-CNT = 327	TA2CLR 0

We can use this to write:

```
TA2CTL = TASSEL_1 | MC_1 | ID_0;
TA2CCR0 = 327;
TA2CCTL0 = CCIE;
```

THIS CONFIGURES THE TIMER TO TRIGGER INTERRUPTS AT $f_{INT} = 0.01$

17.3.1 TAxCTL Register

Timer_Ax Control Register

Figure 17-16. TAxCTL Register

15	14	13	12	11	10	9	8
Reserved						TASSEL	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
ID		MC		Reserved	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Table 17-4. TAxCTL Register Description

Bit	Field	Type	Reset	Description
15-10	Reserved	RW	0h	Reserved
9-8	TASSEL	RW	0h	Timer_A clock source select 00b = TAxCLK 01b = ACLK 10b = SMCLK 11b = INCLK
7-6	ID	RW	0h	Input divider. These bits along with the TAIDEX bits select the divider for the input clock. 00b = /1 01b = /2 10b = /4 11b = /8
5-4	MC	RW	0h	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00b = Stop mode: Timer is halted 01b = Up mode: Timer counts up to TAxCCR0 10b = Continuous mode: Timer counts up to 0FFFFh 11b = Up/down mode: Timer counts up to TAxCCR0 then down to 0000h
3	Reserved	RW	0h	Reserved
2	TACLR	RW	0h	Timer_A clear. Setting this bit clears TAR, the clock divider logic (the divider setting remains unchanged), and the count direction. The TACLR bit is automatically reset and is always read as zero.
1	TAIE	RW	0h	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0b = Interrupt disabled 1b = Interrupt enabled
0	TAIFG	RW	0h	Timer_A interrupt flag 0b = No interrupt pending 1b = Interrupt pending

17.3.3 TAxCCTLn Register

Timer_Ax Capture/Compare Control n Register

Figure 17-18. TAxCCTLn Register

15	14	13	12	11	10	9	8
CM		CCIS		SCS	SCCI	Reserved	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)
7	6	5	4	3	2	1	0
OUTMOD			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

Table 17-6. TAxCCTLn Register Description

Bit	Field	Type	Reset	Description
15-14	CM	RW	0h	Capture mode 00b = No capture 01b = Capture on rising edge 10b = Capture on falling edge 11b = Capture on both rising and falling edges
13-12	CCIS	RW	0h	Capture/compare input select. These bits select the TAxCCR0 input signal. See the device-specific data sheet for specific signal connections. 00b = CCIxA 01b = CCIxB 10b = GND 11b = VCC
11	SCS	RW	0h	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0b = Asynchronous capture 1b = Synchronous capture
10	SCCI	RW	0h	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit.
9	Reserved	R	0h	Reserved. Reads as 0.
8	CAP	RW	0h	Capture mode 0b = Compare mode 1b = Capture mode
7-5	OUTMOD	RW	0h	Output mode. Modes 2, 3, 6, and 7 are not useful for TAxCCR0 because EQUx = EQU0. 000b = OUT bit value 001b = Set 010b = Toggle/reset 011b = Set/reset 100b = Toggle 101b = Reset 110b = Toggle/set 111b = Reset/set
4	CCIE	RW	0h	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag. 0b = Interrupt disabled 1b = Interrupt enabled
3	CCI	R	0h	Capture/compare input. The selected input signal can be read by this bit.
2	OUT	RW	0h	Output. For output mode 0, this bit directly controls the state of the output. 0b = Output low 1b = Output high

17.3.4 TAxCCRn Register

Timer_A Capture/Compare n Register

Figure 17-19. TAxCCRn Register

15	14	13	12	11	10	9	8
TAxCCRn							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
TAxCCRn							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Table 17-7. TAxCCRn Register Description

Bit	Field	Type	Reset	Description
15-0	TAxCCR0	RW	0h	Compare mode: TAxCCRn holds the data for the comparison to the timer value in the Timer_A Register, TAR. Capture mode: The Timer_A Register, TAR, is copied into the TAxCCRn register when a capture is performed.

17.3.5 TAxIV Register

Timer_Ax Interrupt Vector Register

Figure 17-20. TAxIV Register

15	14	13	12	11	10	9	8
TAIV							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
TAIV							
r0	r0	r0	r0	r-(0)	r-(0)	r-(0)	r0

Table 17-8. TAxIV Register Description

Bit	Field	Type	Reset	Description
15-0	TAIV	R	0h	Timer_A interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Capture/compare 1; Interrupt Flag: TAxCCR1 CCIFG; Interrupt Priority: Highest 04h = Interrupt Source: Capture/compare 2; Interrupt Flag: TAxCCR2 CCIFG 06h = Interrupt Source: Capture/compare 3; Interrupt Flag: TAxCCR3 CCIFG 08h = Interrupt Source: Capture/compare 4; Interrupt Flag: TAxCCR4 CCIFG 0Ah = Interrupt Source: Capture/compare 5; Interrupt Flag: TAxCCR5 CCIFG 0Ch = Interrupt Source: Capture/compare 6; Interrupt Flag: TAxCCR6 CCIFG 0Eh = Interrupt Source: Timer overflow; Interrupt Flag: TAxCTL TAIFG; Interrupt Priority: Lowest

How do the definitions for register fields work?

A constant is provided for each possible value for most config register fields, with values placed in the appropriate fields for each register. So convenient!

Ex.

	(Field name)				(Value (decimal))	
TASSEL_1	0000	0001	0000	0000
MC_1	0000	0000	0001	0000
TASSEL_1 | MC_1	0000	0001	0001	0000

-OR- TASSEL_1 + MC_1

⇒ You can combine these constants to make almost all possible register combinations!

Since none of the fields of a single register overlap, we can combine them using either addition or bitwise OR (|) operations.

Ex.

(OR) 0 1 / 1 1 0 / 1 1

(ADD) 0 1 / + 1 0 / 1 1

Note: For a single-bit field, the constant is just the name (Ex. CCIE, TAIE), not CCIE_1, etc.

Step 4: Write Interrupt Service Routine (ISR) and enable interrupts

… how do we write interrupts in our code, anyway?

An ISR for Timer A2 looks like this:

```
// Example syntax for TimerA2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{
   // Do something
   // ...
}
```

In addition, in your `main()` you must **enable interrupts** to tell the CPU to handle them:

```
// Using pre-defined macros in msp430.h

_BIS_SR(GIE);          // Global interrupt enable
// ... OR ...
__enable_interrupt();
```

(The above macros are equivalent. You will see both of them in example code and notes in this class.)

Back to the example: what does it mean when we get an interrupt from Timer A2?
What should the ISR do?

FOR TIMER, IT MEANS THAT Δt_{INT} HAS ELAPSED

Each interrupt means that the timer has reached MAX_CNT, meaning that 328 ticks of ACLK ~= 0.01s have elapsed.

Thus, the ISR should count how many interrupts have occurred… and do nothing else:

```
// Global count of clock ticks
unsigned long int timer = 0;

#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{

}
```

COUNTER: NUMBER OF INTERVALS OF Δt_{INT} THAT HAVE ELAPSED

TIMER++;

⇒ Ex. TIMER = 1234;

$\Delta t_{INT} = 0.01s$

$(1234 \text{ TICKS})(0.01s/\text{TICK}) = 12.34 \text{ SEC ELAPSED}$

Important note: ALWAYS keep your ISRs short. Why?
What happens if your ISR hasn't completed before the next ISR arrives?

In general, NEVER do any of the following in an ISR:

- Write to the display
- Flush the display
- Do floating point math
- Call expensive functions like `sin()` or `sprintf()`

THINGS YOU CAN DO

- COUNTERS
- IF STATEMENTS
- DIGITAL I/O PINS
- SET/CLEAR FLAGS.

DEADLINE MISSED!

- NEXT ISR WILL BE LATE!
- TAKES TIME AWAY FROM MAIN()

THINGS TO DO W/ TIMER VARIABLE

1. KEEP TRACK OF ELAPSED TIME

```
IF( EVENT) {
    START_TIME = TIMER;
}

. . .

IF((CURRENT_TIME - START_TIME) > ___)
{ DO_THING();
}
```

2. SCHEDULE EVENTS "EVERY N TICKS"

```
IF((TIMER % N) == 0){
    DO_THING();
}
```

THIS IS BEST FOR SCHEDULING EVENTS INSIDE AN ISR — MORE ON THIS LATER.

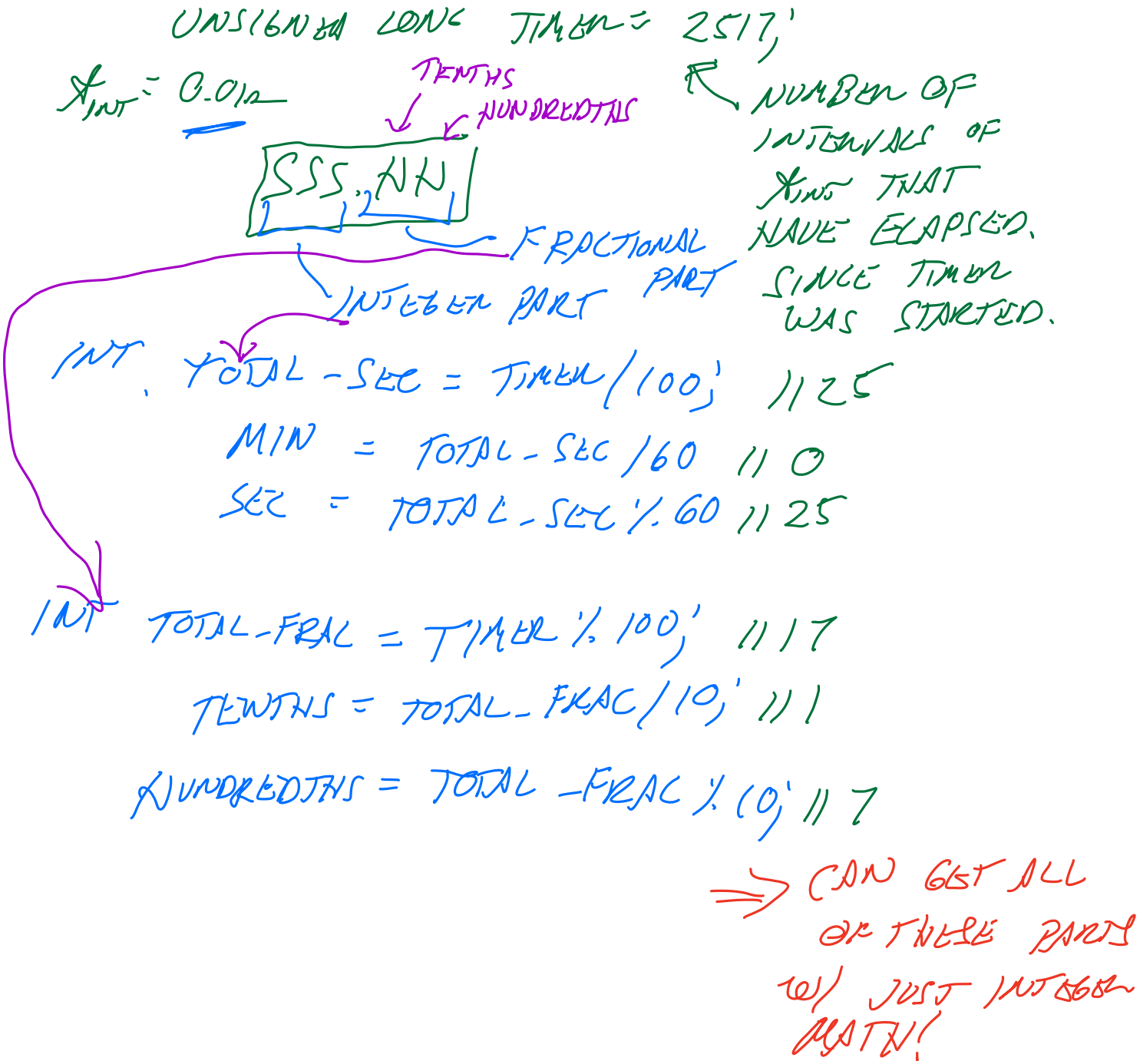
Examples: Using the timer variable

Stopwatch

Now that our ISR is properly configured, what does the variable timer represent? How do you use it to actually display the time?

The timer represents the number of 0.01 second intervals that have elapsed since Timer A2 was started. To use it, we need to convert this to minutes and seconds in order to display it.

How do we do this? Note that we want to do it using *integer math*, since floating point is slow and we eventually want to put this information on the display.



Timer accuracy

How accurate will our stopwatch be? Is that accuracy acceptable?
The duration of one ACLK tick = 1/32768 Hz = 3.05e-5 seconds

$T_{INT} = 0.01_s = \frac{MAX\_CNT + 1}{f_{CLK}}$

$\frac{327 + 1}{32768} = 0.010009\overline{7}_s$

WHEN WE SAY 0.01s HAS ELAPSED, ACTUAL TIME IS HIGHER.

OK FOR LAB, NOT FOR OLYMPICS.

TERMINOLOGY:

- "REPORTED TIME": TIME REPORTED OR USED BY DEVICE OVER SOME INTERVAL
- "ACTUAL TIME": "REAL" TIME THAT HAS ELAPSED OVER THAT INTERVAL.

IF ACTUAL TIME > REPORTED TIME => DEVICE IS SLOW

IF ACTUAL TIME < REPORTED TIME => DEVICE IS FAST

WHEN DOES THIS ERROR MATTER?

SSS:MM

- FOR STOPWATCH: IF DISPLAY IS OFF BY 0.01s, WE WILL SHOW THE WRONG VALUE!!!

HOW LONG UNTIL ERROR ADDS UP TO 0.01s?
HOW MANY INTERRUPTS?

$$0.01s = |(X \text{ INTERRUPTS})(\text{REPORTED TIME} - \text{ACTUAL TIME})|$$

$$0.01s = |(X \text{ INTS})(0.01s - 0.010009576s)|$$

$X = 1023.661 \approx 1024$ INTERRUPTS

≈ 10.24 SECONDS

EVERY 10.24 SECONDS, DISPLAY IS OFF BY 0.01

HOW DO WE COMPENSATE FOR THIS??

$\Rightarrow$ "LEAP COUNTING"

STRATEGY: SINCE TIMER IS SLOW BY ONE INTERVAL EVERY 1024 INTERRUPTS, CORRECT BY ADDING 1 INTERVAL EVERY 1024 INTERRUPTS.

"LEAP COUNT"

~~"LEAP SECOND"~~

Since our stopwatch will run slow, how long until it is off by 0.01 second?
Here is how we can add a leap count for this example:

LEAP_CNT=0

```
#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{
  IF(LEAP_CNT < 1024){
      TIMER++;
      LEAP_CNT++;
  }ELSE{
      TIMER+=2;
    LEAP_CNT=0;
}
```

When using leap counting, we can use the following general rule:

IF TIMER IS SLOW, ISR SHOULD DOUBLE INCREMENT

IF TIMER IS FAST, ISR SHOULD SKIP A COUNT

In our example, is our leap counting solution perfect? For how long will it be accurate to 0.01 sec?

EX. SAY TIMER = 1022

TIMER++ => 1023

ON NEXT INTERVAL, ADD AN EXTRA COUNT

TIMER = 1025

WITH LEAP COUNTING, IT TAKES 1024 COUNTS TO GET ERROR GREATER THAN 0.01s

SSS.NN

OK FOR STOPWATCH, SINCE MAX ON DISPLAY IS ONLY 999 => DON'T NEED TO CORRECT MORE.

REPORTED TIME 10.23

ACTUAL

(1023)(.010009766s)

= 10.239999...

10.25s

(1024)(.010009766s)

= 10.250003845s

DIFF = .385ms

NO 15

TA2CCTL0 = CCIE;
TA2CCR0 = 16383;
TA2CTL = TASSEL_1 + ID_0 + MC_1;

Configuration example

Configure Timer A2 for 0.5 sec resolution. Do you need to use leap counting?

ASSUME UP MODE.
ACLK

$t_{INT} = 0.5$ s

$t_{INT} = \frac{MAX\_CNT + 1}{32768}$

$0.5 = \frac{MAX\_CNT + 1}{32768}$

$MAX\_CNT = 16383$

REGISTER CONFIG:

CAN PLUG MAX-CNT BACK INTO EQUATION TO FIND ACTUAL TIME:

$t_{INT,ACTUAL} = \frac{16383+1}{32768} = \frac{16384}{32768} = 1/2 = 0.5$ s EXACTLY

IN THIS CASE... MAX-CNT DIVIDES EVENLY, SO NO LEAP COUNTING!

Another configuration example

What if you wanted 0.0001 second resolution? What do we do now?

$0.0001 = t_{INT} = \frac{MAX\_CNT + 1}{f_{CLK}}$

ACLK = 32768 Hz
SMCLK 1.048576 MHz

TRY ACLK:

$0.0001_s = \frac{MAX\_CNT + 1}{32768}$

$MAX\_CNT = \lfloor 2.27 \rfloor = 2$

2+1 = 3 TICKS OF ACLK

↘ ROUND-OFF IS HUGE!!

INSTEAD TRY SMCLK:

$0.0001 = \frac{MAX\_CNT + 1}{1048576 \text{ TICKS/INT}}$

$MAX\_CNT = 108$

REGISTER CONFIG:

TA2CTL = TASSEL_2 + ID_0 + MC_1
(↖ SMCLK, ↖ NO DIVIDER, ← UP MODE)

TA2CCR0 = 108
TA2CCTL0 = CCIE;