CE2049: Lecture 9

Today

- KW4: Review
- Lab 1: Drawing Strings
- Interrupts
- Timers: Part 1

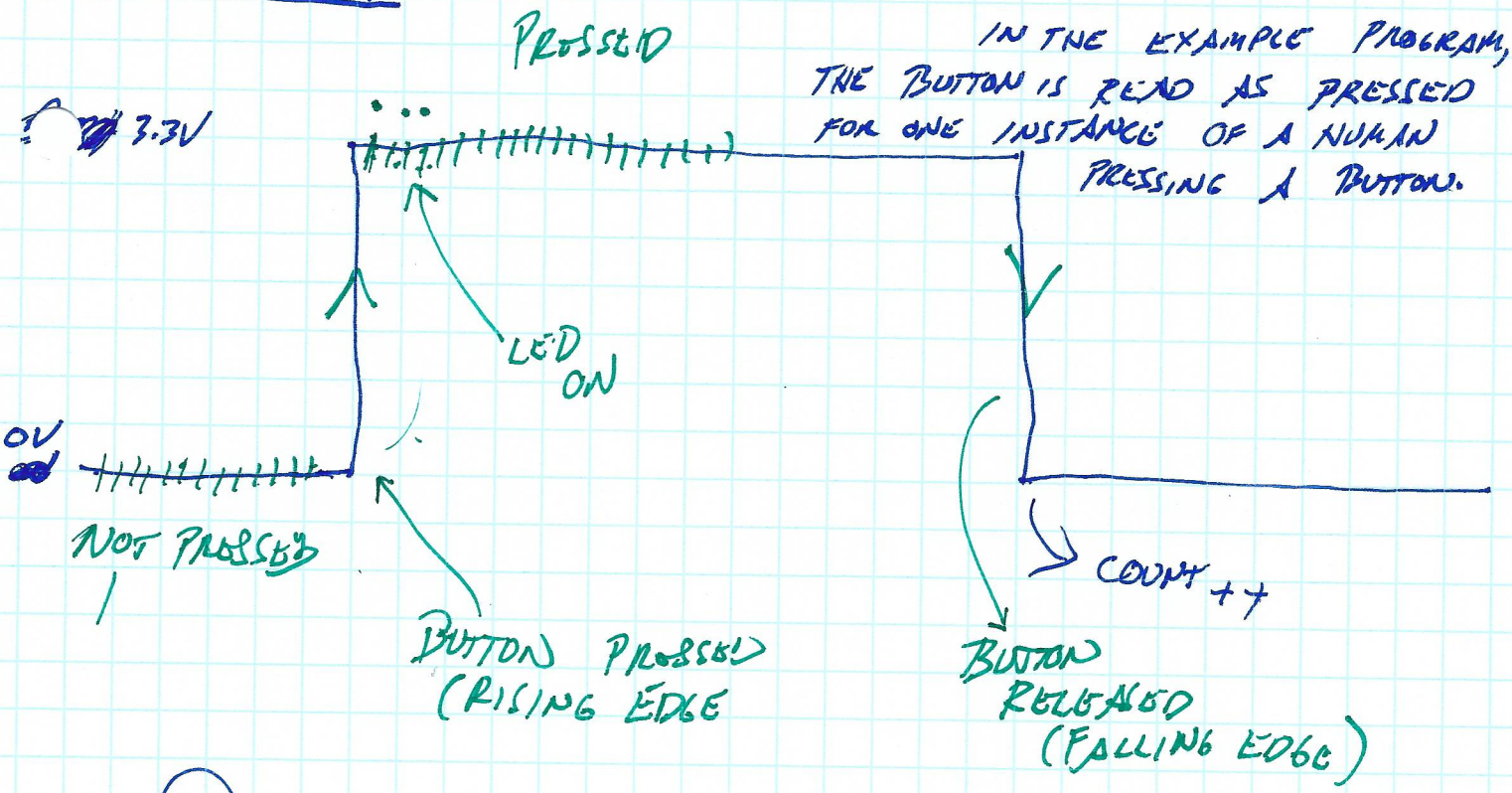
Office Hours

- Today: 5-7PM EDT (Nick)
- Wed
  - 2-4PM EDT (Jonathan)
  - 5-7PM EDT (Nick)
  - Probably other times too (check Discord)
- Thurs: 5-7PM EDT (Nick)

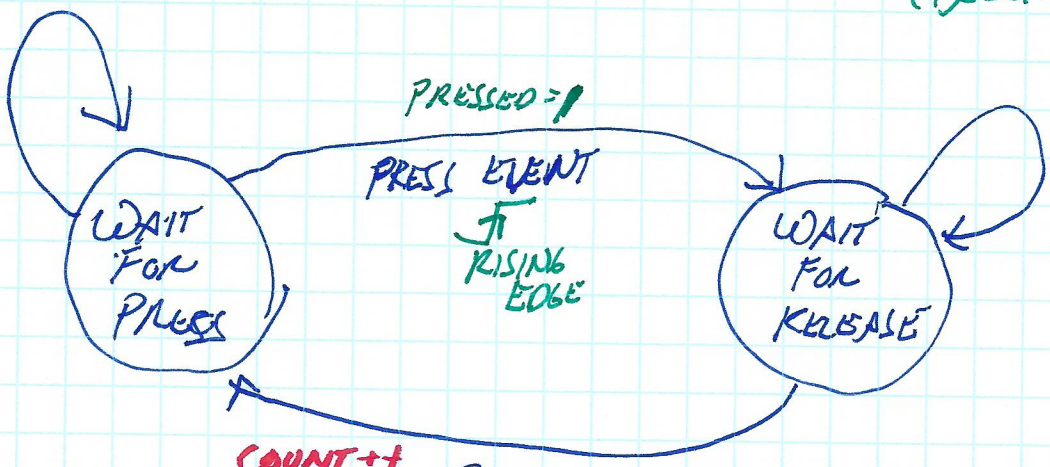
Administrivia

- Lab 1
  - Signoff due Wed ~~6/22~~ (6/23) by 7PM EDT
  - Report due Thurs (6/24) by 11:59~~9~~PM EDT
  - Helpful Resources
    - Lab 1 Demo Video
    - Lab 1 Concepts Video
  - If you are having a tough time with the lab, please reach out to the course staff!
    - ... even if you don't know what to ask
  - If you ~~will~~ need help/signoffs and you ~~could~~ can't come to office hrs, let us know! We can find a time for you.
- KW5: Online after class
- ~~Grades~~ Grades for Lab 0, Exam: Out soon

NCW4: BUTTONS

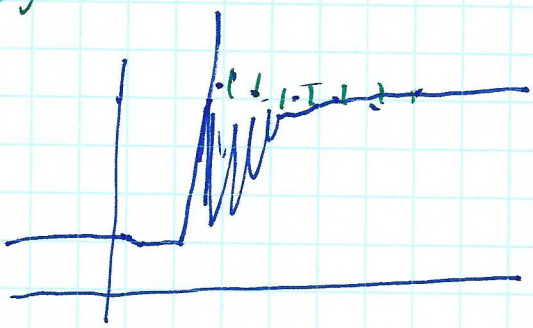


IN THE EXAMPLE PROGRAM, THE BUTTON IS READ AS PRESSED FOR ONE INSTANCE OF A HUMAN PRESSING A BUTTON.



KIND OF THIS WORKS, BUT MIGHT STILL BE DUE TO SWITCH BOUNCE.

INACCURATE CONTACT



ALTERNATIVES

- THRESHOLDS "PRESSED FOR N READS"
- ~~[illegible]~~ TIMING "PRESSED FOR ≥ 10ms".

Module 8. Timers: Theory and Practice

Topics

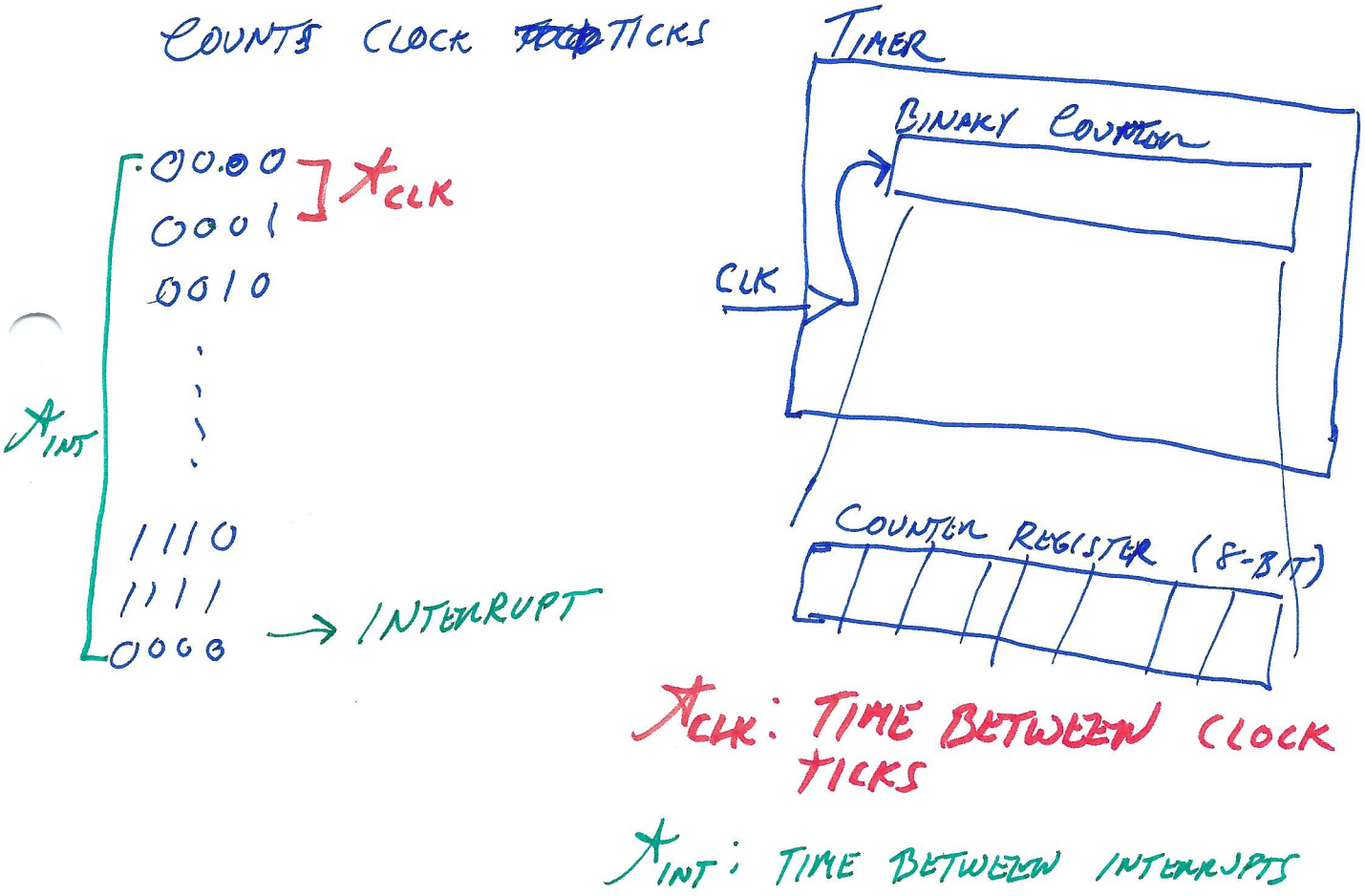
- Intro to Interrupts
- Intro to Timers

WHY TIMERS?

- CAN'T RELY ON EXECUTION OF CODE FOR PRECISE TIMING

But first… what is a timer?

TIMER: CIRCUIT THAT COUNTS CLOCK TICKS



t_{CLK}: TIME BETWEEN CLOCK TICKS

t_{INT}: TIME BETWEEN INTERRUPTS

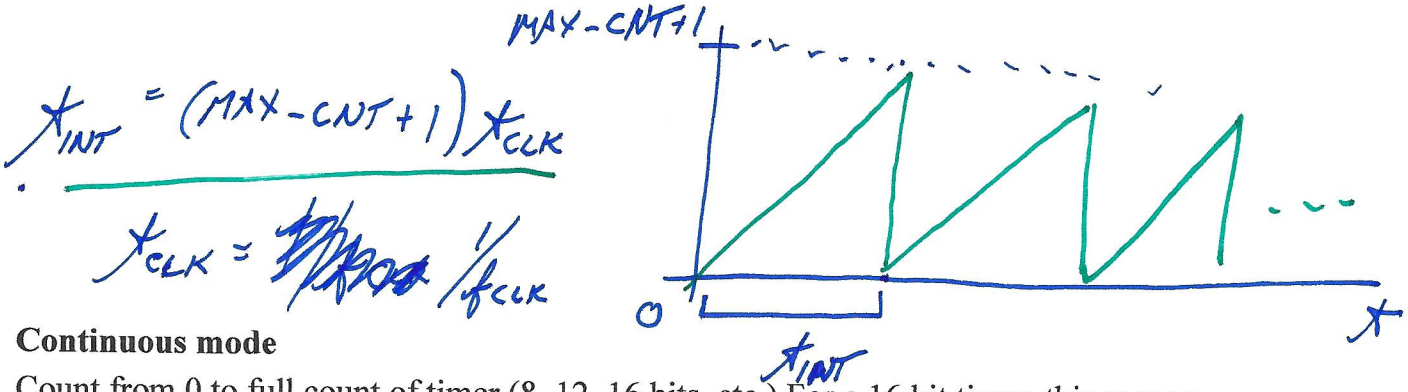
Most microcontrollers have timers in some form. Timers can be used to generate **interrupts** at particular intervals, generate PWM signals, measure frequency of input signals, and more! In this course, we will focus on the generation of timer interrupts, which tell the CPU that a certain amount of time has passed.

Fundamental timer counting modes

Most timers have a number of *counting modes*:

Unidirectional mode (called "Up mode" on the MSP430)

Count from 0 to a *programmer set* maximum count value (which we call MAX_CNT).



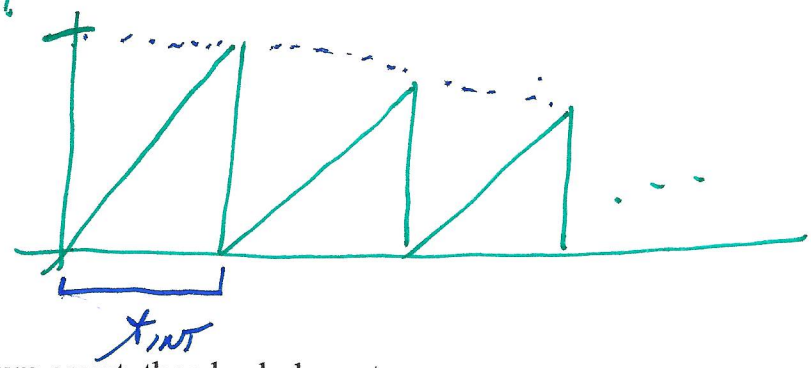
$T_{INT} = (MAX\text{-}CNT + 1) T_{CLK}$

$T_{CLK} = 1/f_{CLK}$

Continuous mode

Count from 0 to full count of timer (8, 12, 16 bits, etc.) For a 16 bit timer, this means:

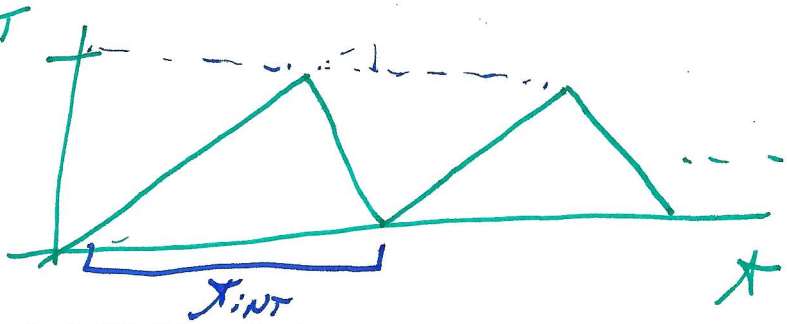
FOR A 16 BIT TIMER OR TIMER B: 0 - (2^16 - 1). = 0 - 65535



$T_{INT} = 2^{16} T_{CLK}$

Up/Down Mode

Counts from 0 to *programmer set* maximum count, then back down to zero



$T_{INT} = (2 * MAX\text{-}CNT) T_{CLK}$

In each mode, most timers (like those on the MSP430) will trigger an interrupt when the count transitions back to 0.

Most timer peripherals have two "operating modes", which control how they use the counter:

- **Capture mode**: Records the counter value when a certain input changes
  - ↳ FREQUENCY COUNTING.
- **Compare mode**: Performs an operation when the counter value reaches a certain value
  - ↳ PERIODIC INTERRUPTS ← WE WILL DO THIS IN LAB
  - ↳ GENERATE PWM SIGNALS

Interrupts

Interrupt: A signal sent to the CPU from a peripheral or external source

- Typically, an interrupt is either a request for the CPU to do something or a notification that the peripheral has something (ie, data) available for the CPU to use.
- The CPU can choose to accept (or to "service") the interrupt, or ignore it. Certain interrupts, called "non-maskable interrupts" (NMIs) cannot be ignored.

Interrupts on the CPU are handled by a special function called an **Interrupt Service Routine (ISR)**.

Note: Interrupts are not just for timers!
Many peripherals on the MSP430 can generate interrupts for different reasons:

- Timers (periodic events)
- ADC: Data is ready
- I/O: (only certain pins)
- Serial ports/other interfaces "Data is ready"
- More!

What does the CPU do when it receives interrupts?
Arrival of interrupts is *asynchronous* to the program's execution.

How do interrupts work internally?

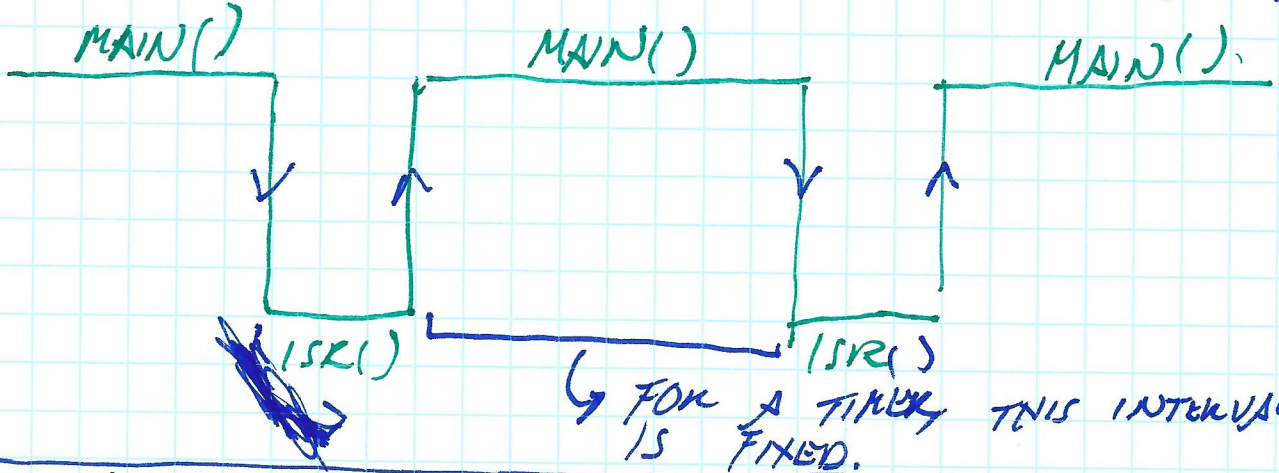
Peripherals that can trigger interrupts so do by issuing a request to the CPU's *interrupt controller*, often over a dedicated wire called an interrupt request line. The interrupt controller decides how the CPU processes each interrupt:

WHEN AN INTERRUPT OCCURS:

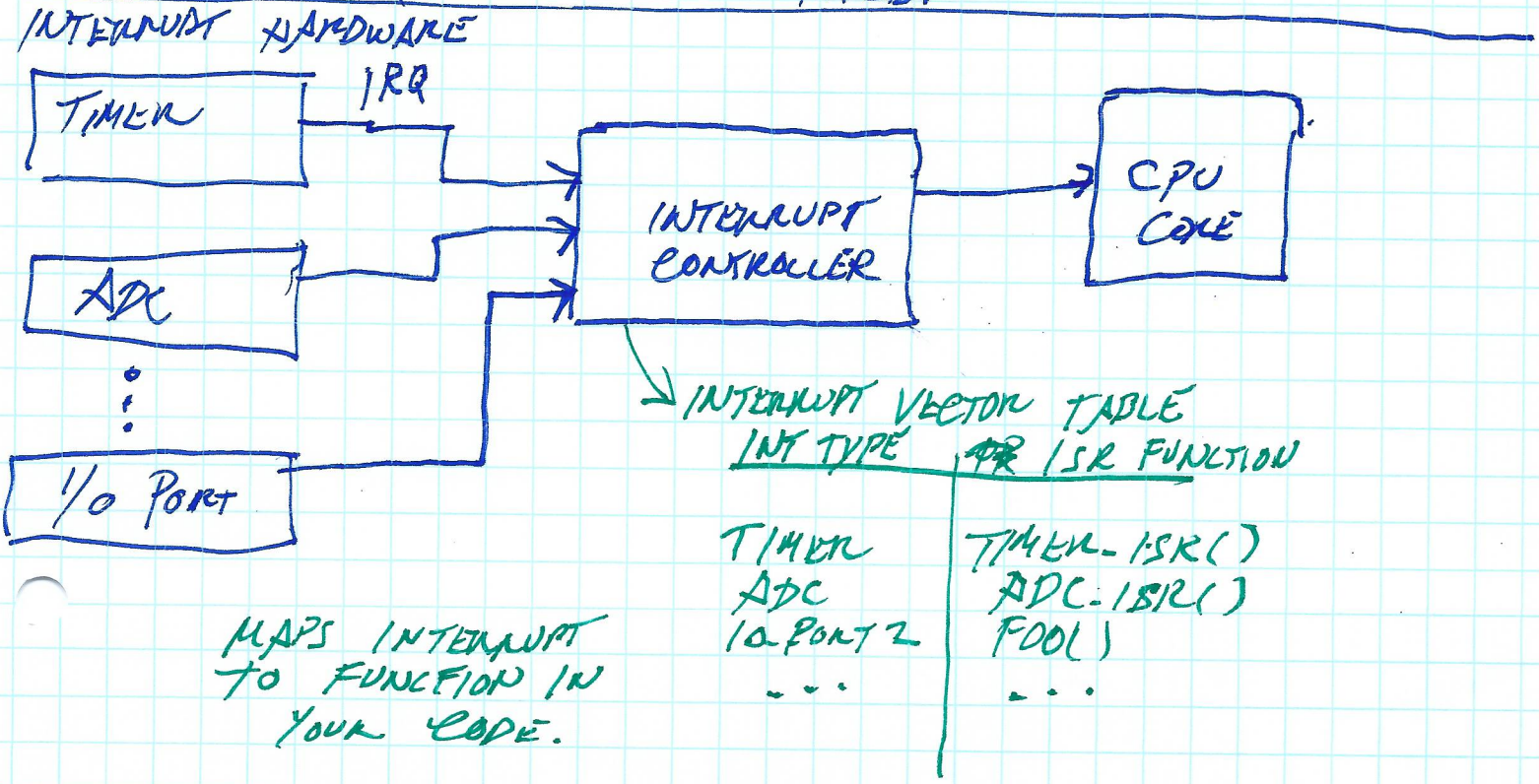
1. STOPS EXECUTING CURRENT INSTRUCTION, SAVES "CONTEXT"
   → CURRENT POSITION IN PROGRAM (PROGRAM COUNTER) + OTHER INFO

2. EXECUTES ISR

3. "RESTORES CONTEXT" AND GOES BACK TO WHERE IT LEFT OFF IN THE PROGRAM

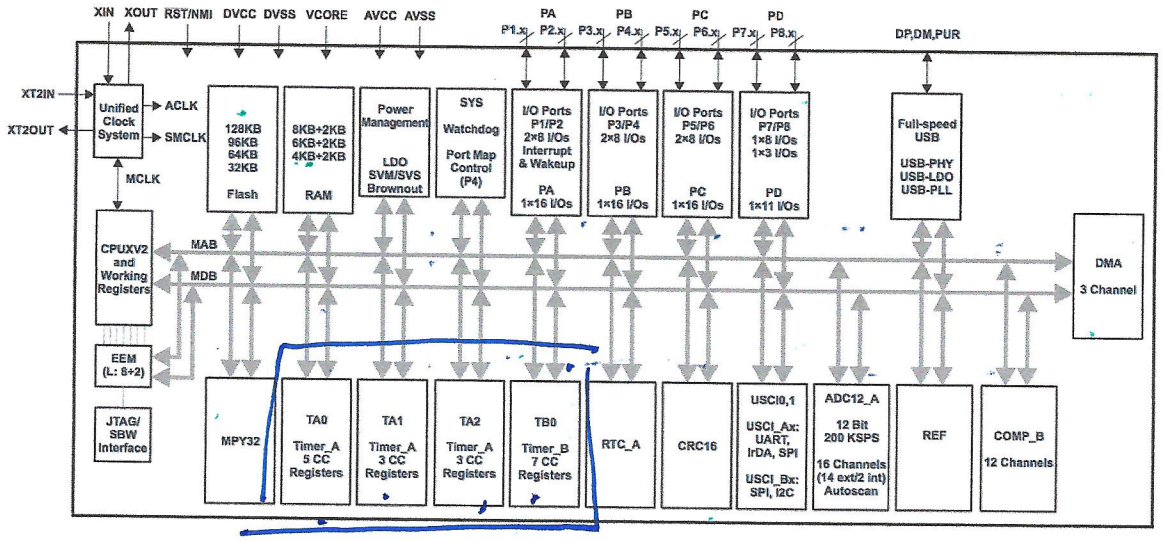


→ FOR A TIMER, THIS INTERVAL IS FIXED.



Timers on the MSP430

The MSP430F5529 has a number of timer peripherals, as shown in the system block diagram:



MSP430s have two main types of timers, Timer A, and Timer B; both types function in very similar ways, but have some subtle differences. Each chip can have multiple Timer A's and B's, as shown in the block diagram.

The MSP430F5529 has the following timers:

Timer B: Has 7 capture/compare units, can generate PWM signals → BUZZER

Timer A0: Multiple capture compare modules, can generate PWM → LCD

Timer A1: Functionally the same as A0 → LCD

Timer A2: 3 capture compare registers → WE WILL USE THIS IN LAB

Additionally, the MSP430F5529 has the following other peripherals that contain timers:

- A Basic Timer, which has some real-time clock features
- The **Watchdog timer** (WDT)
  - When the WDT is on, it must continuously have its count reset within the program
  - If the count reaches zero, it **resets the MSP430!**

Why does the WDT exist? To prevent your program from getting stuck in some kind of unrecoverable state. We don't want to deal with the WDT in your labs, which is why the first line of every program we write is:

```
WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
```

This disables the watchdog timer. If you wait too long to stop the watchdog timer, your program will reset only a few milliseconds after startup!

Configuring timers on the MSP430

Like the UCS module, timers are highly configurable!

We will stick to the basic configurations and **only add complexity when we need it!** This is a good design practice, and also makes our lives easier!

Timer A has the following registers:

17.3 Timer_A Registers

Timer_A registers are listed in Table 17-3 for the largest configuration available. The base address can be found in the device-specific data sheet.

TA2CTL

Table 17-3. Timer_A Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	TAxCTL	Timer_Ax Control	Read/write	Word	0000h	Section 17.3.1
02h	TAxCCTL0	Timer_Ax Capture/Compare Control 0	Read/write	Word	0000h	Section 17.3.3
04h	TAxCCTL1	Timer_Ax Capture/Compare Control 1	Read/write	Word	0000h	Section 17.3.3
06h	TAxCCTL2	Timer_Ax Capture/Compare Control 2	Read/write	Word	0000h	Section 17.3.3
08h	TAxCCTL3	Timer_Ax Capture/Compare Control 3	Read/write	Word	0000h	Section 17.3.3
0Ah	TAxCCTL4	Timer_Ax Capture/Compare Control 4	Read/write	Word	0000h	Section 17.3.3
0Ch	TAxCCTL5	Timer_Ax Capture/Compare Control 5	Read/write	Word	0000h	Section 17.3.3
0Eh	TAxCCTL6	Timer_Ax Capture/Compare Control 6	Read/write	Word	0000h	Section 17.3.3
10h	TAxR	Timer_Ax Counter	Read/write	Word	0000h	Section 17.3.2
12h	TAxCCR0	Timer_Ax Capture/Compare 0	Read/write	Word	0000h	Section 17.3.4
14h	TAxCCR1	Timer_Ax Capture/Compare 1	Read/write	Word	0000h	Section 17.3.4
16h	TAxCCR2	Timer_Ax Capture/Compare 2	Read/write	Word	0000h	Section 17.3.4
18h	TAxCCR3	Timer_Ax Capture/Compare 3	Read/write	Word	0000h	Section 17.3.4
1Ah	TAxCCR4	Timer_Ax Capture/Compare 4	Read/write	Word	0000h	Section 17.3.4
1Ch	TAxCCR5	Timer_Ax Capture/Compare 5	Read/write	Word	0000h	Section 17.3.4
1Eh	TAxCCR6	Timer_Ax Capture/Compare 6	Read/write	Word	0000h	Section 17.3.4
2Eh	TAxIV	Timer_Ax Interrupt Vector	Read only	Word	0000h	Section 17.3.5
20h	TAxEX0	Timer_Ax Expansion 0	Read/write	Word	0000h	Section 17.3.6

We will use a subset of these registers:

- **TA2CTL**: Control register for Timer A2
  - ↳ CONFIGURE TIMER BLOCK
- **TA2CCTLx**: Control register for a capture/compare block
  - ↳ CONTROL ONE CAPTURE/COMPARE EVENT
- **TA2CCRx**: Capture/compare register (data register)
  - ↳ MAX_CNT GOES HERE

CONTROL REGISTERS

DATA REGISTERS

We will discuss how to use these registers in detail using an example.

Timer configuration example: A stopwatch

Example: Implement a stopwatch that measures seconds and hundredths of seconds on our development board.

SSS.HH

First, how do we measure the passage of 0.01 seconds? By counting clock ticks. We will do this by configuring a timer for the job. But how do we start?

RESOLUTION OF OUR TIMER

In our labs, we can break any problem involving timers into a set of steps:

1. Select a timer to use: How about Timer A2?

WE ONLY HAVE ONE CHOICE TIMER_A2!

2. Map desired behavior to an operating mode (Up, Continuous, Up/Down)

USUALLY, JUST PICK UP MODE

COUNT FROM 0 → MAX-CNT

COULD USE UP/DOWN MODE IF YOU WANT A LARGER INTERVAL.

3. Select a clock source and configure registers appropriately

$t_{INT} = (MAX\text{-}CNT + 1)\, t_{CLK}$

$0.01_s = (MAX\text{-}CNT + 1)\left(\frac{1}{f_{CLK}}\right)$

$0 - (2^{16} - 1)$

ACLK, OR SMCLK

NEED TO CHOOSE CLOCK SIGNAL:

ACLK: 32768 Hz; $t_{ACLK} = 3.05 \times 10^{-5}$ s

SMCLK: 1.048576 MHz, $t_{SMCLK} = 9 \times 10^{-7}$ s

SINCE WE ONLY NEED 0.01 s RESOLUTION, ACLK IS FINE.

How do we configure the timer registers?

Need: MAX-CNT !

$$f_{INT} = 0.01\,s = \frac{MAX\text{-}CNT + 1}{} \left(\frac{1}{f_{CLK}} \right)$$

$$0.01\,s = \frac{MAX\text{-}CNT + 1}{32768\ Hz}$$

$32768\ Hz$ — TICKS/SECOND

$$MAX\text{-}CNT + 1 = 327.68 \text{ TICKS}$$

$$MAX\text{-}CNT + 1 = 328 \text{ TICKS}$$

$$MAX\_CNT = 327 \text{ TICKS}$$

$$\frac{327}{32768} = 0.00997\ldots\,s \approx 0.01\,s$$

However, we ~~can~~ can only have an integer number of clock ticks!!

↑ This will be important when we talk about timer accuracy!

Using this information, we can write the register configuration:

Parameters we know	Relevant register field
CLOCK SOURCE : ACLK	TASSEL = 1
COUNTING MODE: UP MODE	MC = 1
DIVIDER : 1/1	ID = 0
MAX-CNT = 327	SET IN TA2CCR0

We can use this to write:

```
TA2CTL = TASSEL_1 + ID_0 + MC_1;

TA2CCR0 = 327;

TA2CCTL0 = CCIE;
```

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

Copyright © 2008–2015, Texas Instruments Incorporated

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!

(field name) ↙ value (decimal)

Ex.

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 <u>combine</u> 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 <u>just the name</u> (Ex. CCIE, TAIE), not CCIE_1, etc.

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

TA2CCTL0 = CCIE;

Copyright © 2008–2015, Texas Instruments Incorporated

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

MAX_CNT GOES HERE!

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

Copyright © 2008–2015, Texas Instruments Incorporated

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
    // ...
}
```

GO ADDS THIS FUNCTION TO INTERRUPT VECTOR TABLE FOR TIMER A2.

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?

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

$T_{INT} = 0.01s$

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)
{
    TIMER++;


}
```

COUNTER OF INTERVALS OF T_{INT} THAT HAVE ELAPSED.

EX. TIMER = 1234

12.34 SECONDS

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

- DEADLINE MISSED! NEXT ISR WILL BE LATE!
- TAKE TIME AWAY FROM MAIN()

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()`

FEW THINGS YOU CAN DO:

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

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.

UNSIGNED LONG TIMER = 2517;

NUMBER OF INTERVALS OF T_INT THAT HAVE ELAPSED SINCE TIMER START

SSS.NN SEC

FRACTIONAL PART

INTEGER PART

T_INT = .01s

INT TOTAL_SEC = TIMER/100; // 25

MIN = TOTAL_SEC/60; // 0

SEC = TOTAL_SEC % 60; // 25

INT TOTAL_FRAC = TIMER % 100; // 17

TENTHS = TOTAL_FRAC/10; // 1

HUNDREDTHS = TOTAL_FRAC % 10; // 7

=7 CAN GET ALL THESE PARTS W/ JUST INTEGER MATH

WHEN DOES THIS ERROR MATTER?

SSS.MH

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

HOW LONG UNTIL ERROR ADDS UP TO 0.01_s?

HOW MANY INTERRUPTS?

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

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

$X = 1023.66 \approx$ 1024 INTERRUPTS

1024 INTERRUPTS $\approx$ 10.24 SECONDS

EVERY 1024 INTERRUPTS, OUR DISPLAY IS OFF BY 0.01

HOW DO WE COMPENSATE FOR THIS?!

$\Rightarrow$ LEAP COUNTING.

STRATEGY: SINCE OUR TIMER IS SLOW BY 1 INTERVAL EVERY 1024 INTERRUPTS, CORRECT BY ADDING AN EXTRA COUNT EVERY 1024 INTERRUPTS.

"LEAP COUNT"

"LEAP SECOND"

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

$t_{INT} = 0.01s$

$t_{INT} = 0.01s = \frac{MAX\_CNT + 1}{f_{CLK}}$

$\frac{327+1}{32768\ Hz} = 0.010005976s$

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

OK FOR LAB, NOT FOR THE 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 => DEVICE IS FAST

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 WOULD TAKE 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

$t_{INT} = \frac{MAX\text{-}CNT + 1}{32768}$

$0.5 = \frac{MAX\text{-}CNT + 1}{32768}$

$MAX\text{-}CNT = 16383$

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

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

REGISTER CONFIG:

Another configuration example

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

$0.0001 = t_{INT} = \frac{MAX\text{-}CNT + 1}{f_{CLK}}$

ACLK = 32768 Hz

SMCLK 1.048576 MHz

TRY ACLK:

$0.0001_s = \frac{MAX\text{-}CNT + 1}{32768}$

$MAX\text{-}CNT = \lfloor 2.27 \rfloor = 2$

2+1 = 3 TICKS OF ACLK

→ ROUND-OFF IS HUGE!!

INSTEAD TRY SMCLK:

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

$MAX\text{-}CNT = 108$

REGISTER CONFIG:

TA2CTL = TASSEL_2 + ID_0 + MC_1

↖ SMCLK ↖ NO DIVIDER ← UP MODE

TA2CCR0 = 108

TA2CCTL0 = CCIE;