# Today's plan:

- Announcements
- PWM
- Python and graphical user interface (GUI).
- Timers on the MSP430
- Sensors

# Announcement:

Projects: you should start planning. You will need to provide a short written description (about 1/2 page) of what you plan to build. In it, describe (possibly in points):

1) the function of the project, 2) an outline of 'how it will work,' both in terms of user interface and in terms of what hardware does what. 3) A list of what kinds of parts you will require. 4) How you plan to acquire parts. Submit on Canvas before your lab 6 section.

Project Scope: Your project **must** use the MSP430 as a central component. Your project should incorporate *at least one* non-trivial external hardware component (sensor, motor, display etc.), which was not used in the manual parts 1-9. Your project may (but is not required) to communicate with a host computer or an iPhone for display or user interaction.

Due before lab 6 (week of February 23, 2025)

# Pulse Width Modulation (PWM)

PWM outputs on MSP430F5529.

P1.2

P1.3

P1.4

P1.5

P2.4

P2.5

Notice: they are all set up by the control registers unique to each PWM channel (registers starting with TA). Some have to share period.

# Pulse Width Modulation (PWM)

```
#include <msp430.h>
int i = 0;
void main(void){ WDTCTL = WDTPW|WDTHOLD; // Stop WDT
 P1DIR |= BIT2;       // Output on Pin 1.2
 P1SEL |= BIT2;       // Pin 1.2 selected as PWM
 TA0CCR0 = 500;     // PWM period 500/1.048 microseconds
 TA0CCR1 = 50;       // PWM duty cycle
 TA0CCTL1 = OUTMOD_7; // TA0CCR1 reset/set-high voltage
                                    // below count, low voltage when past
 TA0CTL = TASSEL_2 + MC_1 + ID_0 + TAIE;
 // Timer A control set to SMCLK, count up mode MC_1, no division
  //- keep 1.048 MHz, enable interrupt
 _bis_SR_register(LPM0_bits); // Enter Low power mode 0
}
```

One needs to read a description in

Microprocessor family

# Program issues - flickering

The result of this program is sometimes with unstable frequency or duty cycle. To fix it we add the following code:

```
// Loop until XT1, XT2 & DCO fault flags are cleared
do {
    UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
 // Clear XT2, XT1, DCO fault flags
    SFRIFG1 &= ~OFIFG;                          // Clear global oscillator fault flag
} while (SFRIFG1 & OFIFG);                      // Test if any fault remains
```

Sometimes it works with just these 2 commands without the do-while loop

# Names of Registers and Bits

They are all in include file "msp430.h"

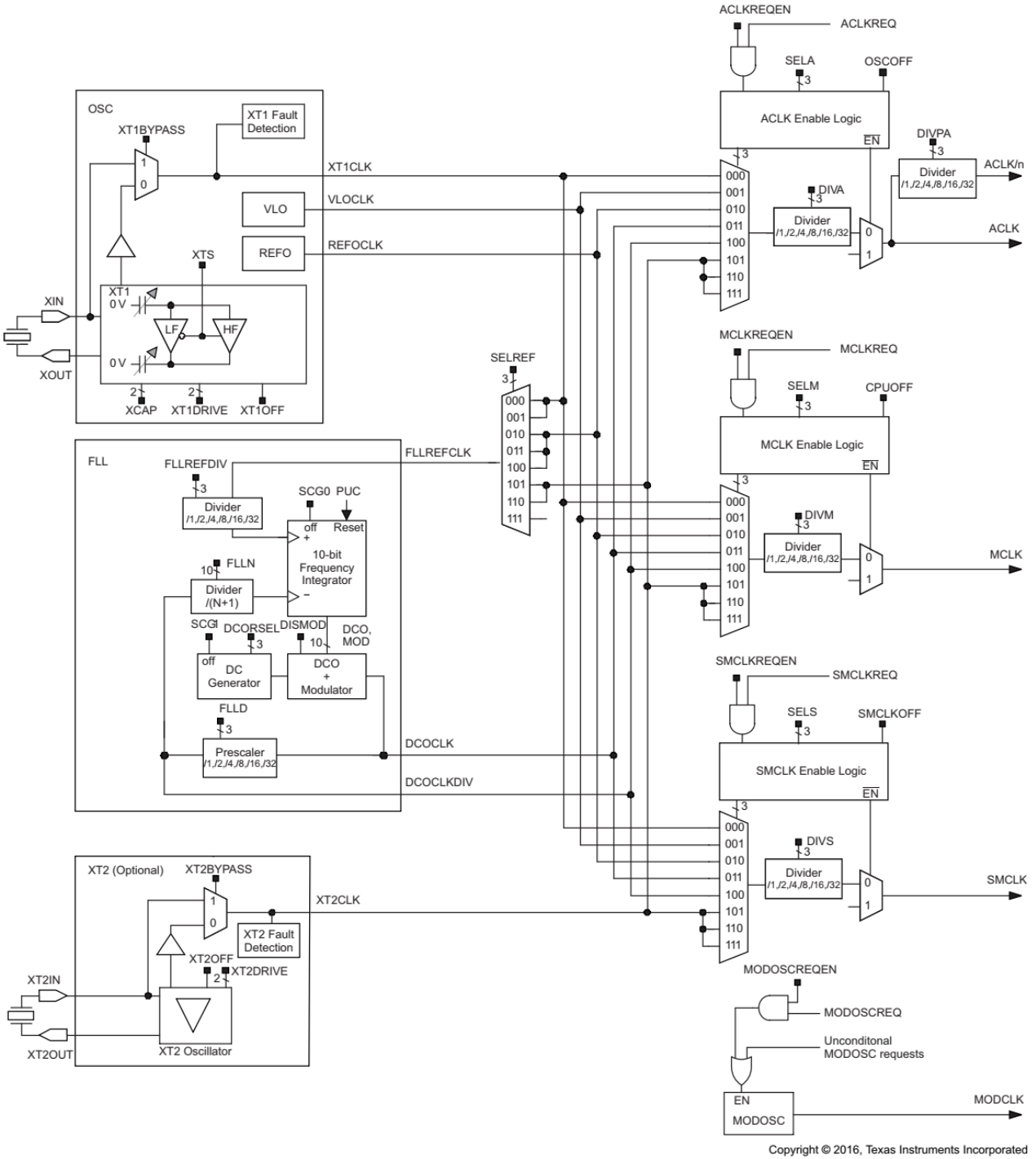



**Figure 5-1. UCS Block Diagram**

### 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<br>00b = TAxCLK<br>01b = ACLK<br>10b = SMCLK<br>11b = INCLK |
| 7-6 | ID | RW | 0h | Input divider. These bits along with the TAIDEX bits select the divider for the input clock.<br>00b = /1<br>01b = /2<br>10b = /4<br>11b = /8 |
| 5-4 | MC | RW | 0h | Mode control. Setting MC = 00h when Timer_A is not in use conserves power.<br>00b = Stop mode: Timer is halted<br>01b = Up mode: Timer counts up to TAxCCR0<br>10b = Continuous mode: Timer counts up to 0FFFFh<br>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.<br>0b = Interrupt disabled<br>1b = Interrupt enabled |
| 0 | TAIFG | RW | 0h | Timer_A interrupt flag<br>0b = No 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<br>00b = No capture<br>01b = Capture on rising edge<br>10b = Capture on falling edge<br>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.<br>00b = CCIxA<br>01b = CCIxB<br>10b = GND<br>11b = VCC |
| 11 | SCS | RW | 0h | Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.<br>0b = Asynchronous capture<br>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 from this bit. |
| 9 | Reserved | R | 0h | Reserved. Reads as 0. |
| 8 | CAP | RW | 0h | Capture mode<br>0b = Compare mode<br>1b = Capture mode |
| 7-5 | OUTMOD | RW | 0h | Output mode. Modes 2, 3, 6, and 7 are not useful for TAxCCR0 because EQUx = EQU0.<br>000b = OUT bit value<br>001b = Set<br>010b = Toggle/reset<br>011b = Set/reset<br>100b = Toggle<br>101b = Reset<br>110b = Toggle/set<br>111b = Reset/set |
| 4 | CCIE | RW | 0h | Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.<br>0b = Interrupt disabled<br>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.<br>0b = Output low<br>1b = Output high |

### 12.2.5.2 Output Example — Timer in Up Mode

The OUTx signal is changed when the timer *counts* up to the TA0CCRx value, and rolls from TA0CCR0 to zero, depending on the output mode. An example is shown in Figure 12-12 using TA0CCR0 and TA0CCR1

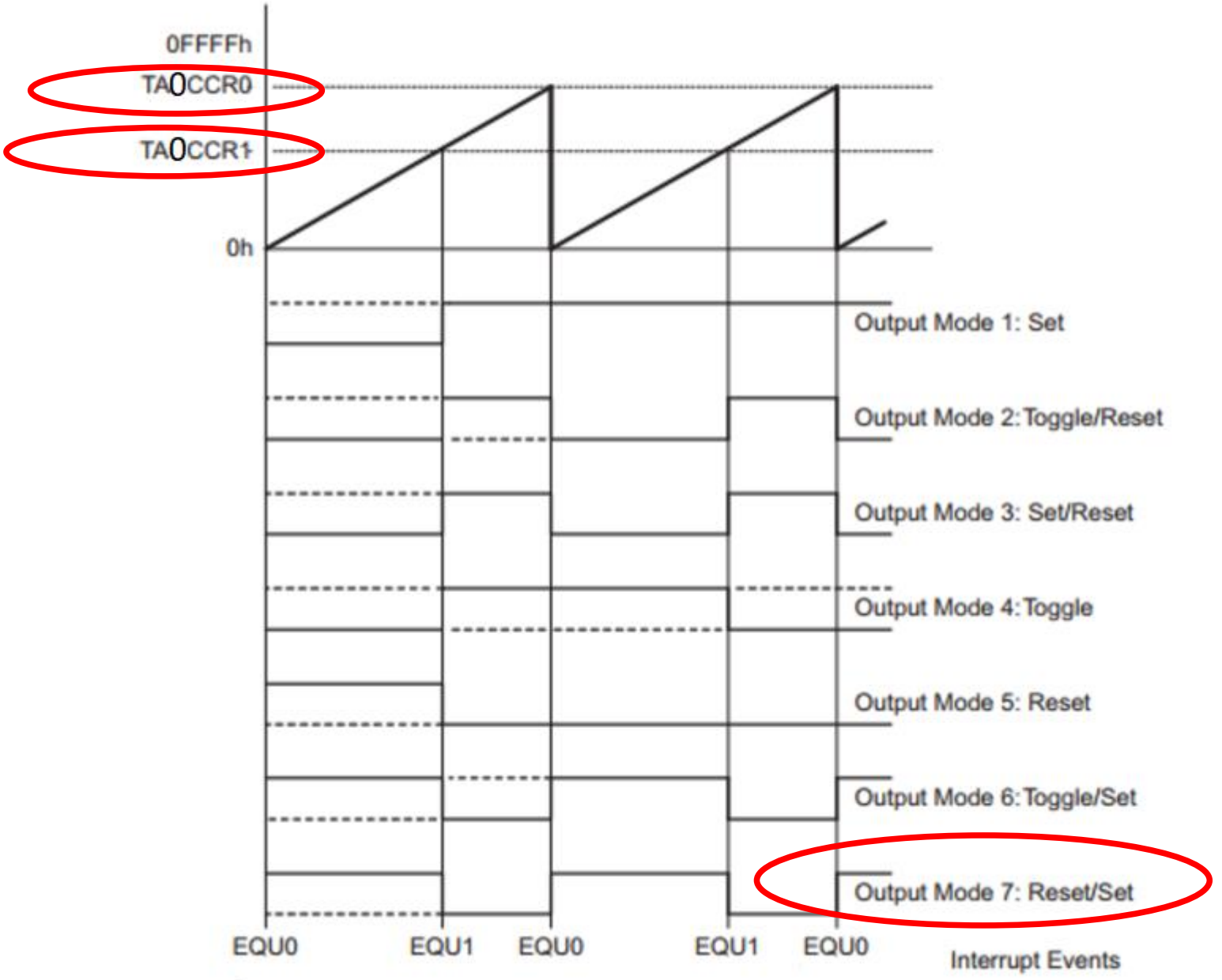



**Figure 12-12. Output Example—Timer in Up Mode**

# Ultrasonic distance sensor

Idea comes from bats and dolphins

- Send a pulse (chirp) of high frequency sound
- Wait for the echo
- Estimate or calculate distance from a time between the send time and return time

Radars work on the same principle but with electromagnetic waves.

LADAR (LIDAR)

# Connecting the ultrasonic distance sensor to test it:

5V

Pulse from Generator

**Oscilloscope**

Pulse amplitude should be 2V, offset 0 (no negative part), short with frequency allowing echo to return from a long distance (say 15 m) before the next pulse comes in.

You should see the width of the echo pulse on the oscilloscope changing when you move an object (for example box) farther or closer to the sensor.

**Distance measurement with real-time display on the computer.**
**Program should:**

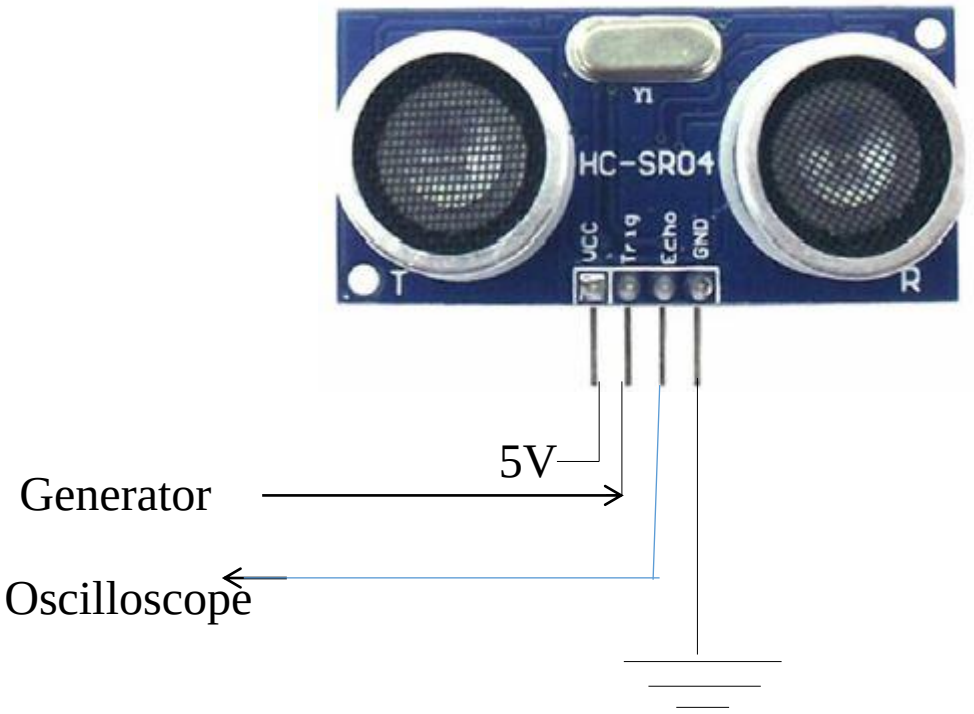
1. Trigger the ultrasonic distance measurement sensor to begin a measurement.

2. Do nothing till the “echo” pin goes high

3. Time the interval while the echo pin is high

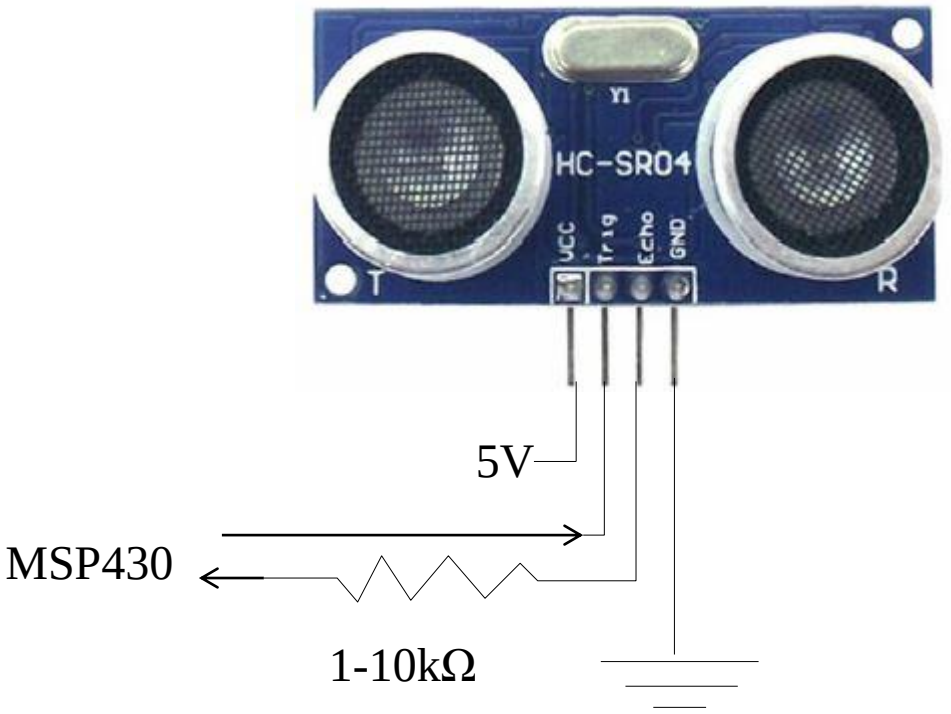
4. Transmit the time interval through the serial port to the host computer where a python program will receive it and print or plot it.

Connecting the ultrasonic distance sensor to Hantek for testing:




The SR04 is a 5V device! To protect the MSP430 input from it, you should put a resistor in between the 'Echo' line and the MSP430. 3.3 V pulse is high enough to trigger it.

Connecting the ultrasonic distance sensor:




The SR04 is a 5V device! To protect the MSP430 input from it, you should put a resistor in between the 'Echo' line and the MSP430. 3.3 V pulse is high enough to trigger it.

# Python and GUI's

For part 8 you will need Python installed and example temperature programs tested.

We used Python in Phys 219 so you are familiar with it and have it installed on your computers.

Python programs are interpreted, not compiled which make them faster to change but the execution is slower and it takes more memory.

They have the same format on windows, mac and Linux

Remember that in python white spaces and indenting matters (unlike C where it's just for readability)

.

C vs Python:

Python programs generally consume much more memory and execute much more slowly than an equivalent program in C.

For our applications Python programs are fast enough.

Why are we using python?

- It is much easier to make a non-trivial program (with GUI) work cross-platform in python than C.

- Interacting with system hardware (e.g. serial port) is more straightforward in Python.

To make your python programs run tolerably fast:

If you are manipulating arrays of numbers,
1) always use numpy arrays
2) never iterate over the array if you can avoid it (and you can almost always avoid it!)

e.g.:

```
import numpy as np   # is similar to an include file, but way more powerful.
a = np.arange(0,50,1) # a is an array of 50 elements: 0, 1, 2 ... 49
b=a * 0.2                                   # b is an array 0, 0.2, 0.4, ... 0.98
```

As compared to:

```
a =  range(0,50,1)
b=[]
for i in range(50):
        b[i] = a[i]*0.2
```

Remember that in python # is a symbol for comment like // in C

```
import numpy as np
...
with serial.Serial(port,9600,timeout = 0.050) as ser
#sometimes serial has trouble unless you change the bound rate
# with timeout=0, read returns immediately, even if no data
# with timeout=.05, ser.read will wait for up to 50 ms for a byte to appear
# from the serial port, if there isn't one waiting.
...

#port = "/dev/ttyACM0"  #for Linux
#port = "COM3" #For Windows Com5 or Com7 Find out which is connected to Launchpad
#port = "/dev/tty.uart-XXXX" #For Mac

...
```

Python program List_All_coms.py is now included with temperature programs.
It works on all 3 systems.

# Python resources:

There are tons of python resources on the web.

Some useful starting points:

Beginners guide to python:
https://wiki.python.org/moin/BeginnersGuide

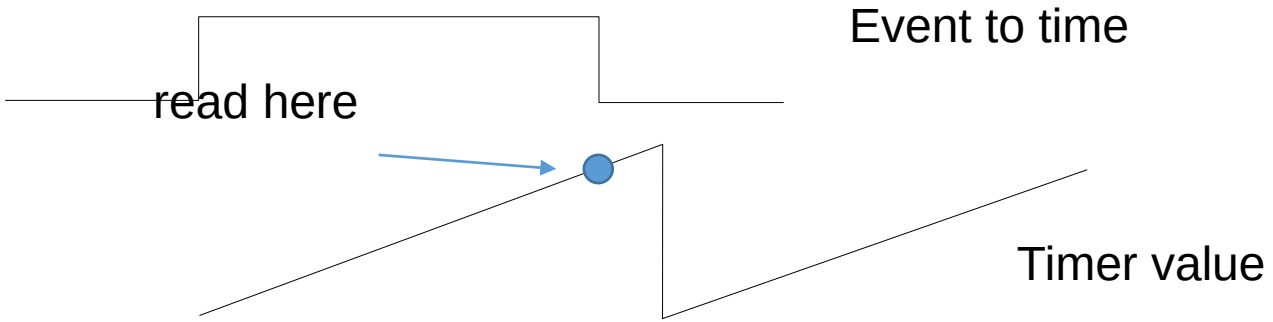
numpy:
http://www.numpy.org/

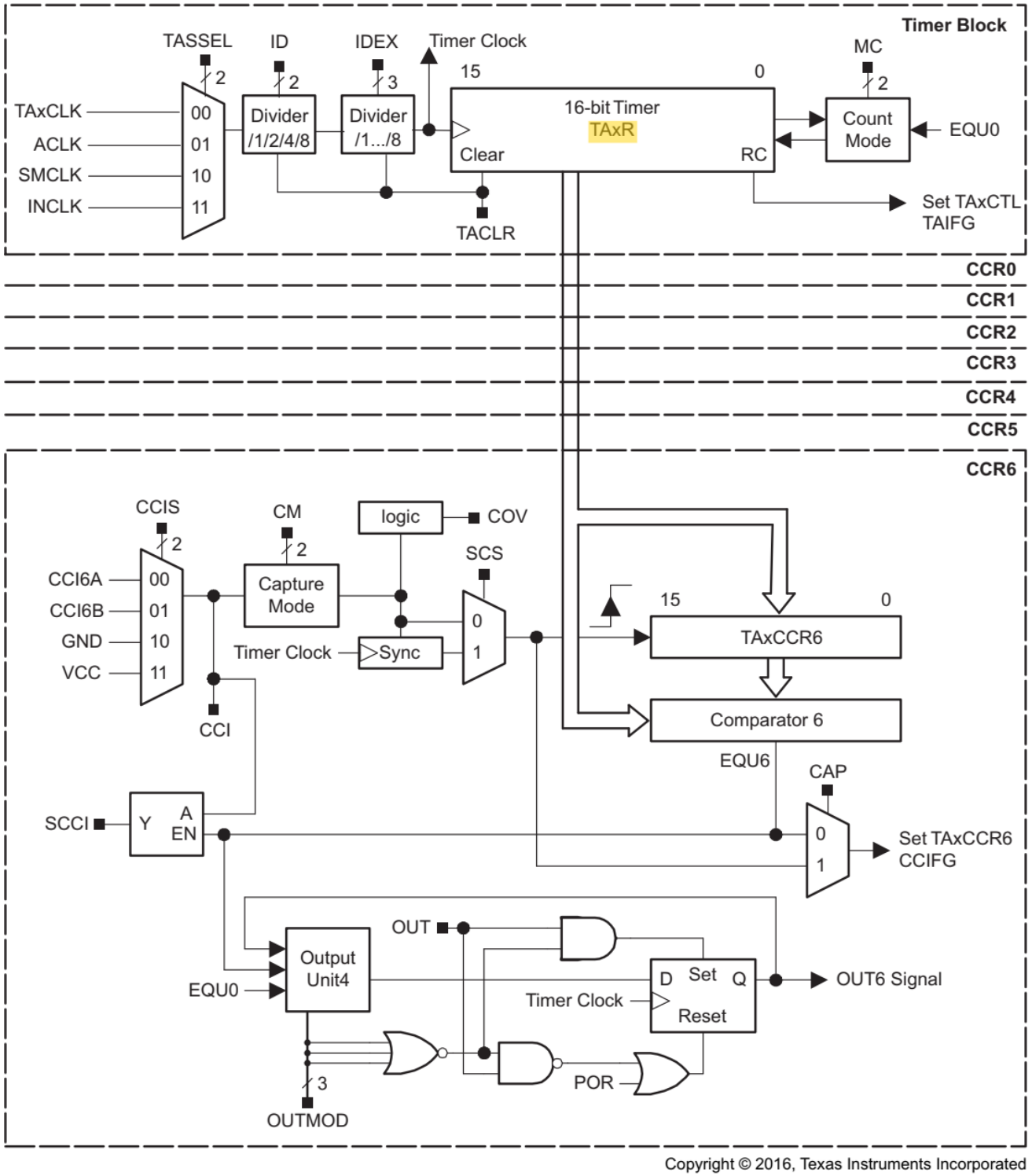
Matplotlib:
https://matplotlib.org/stable/users/index.html

# Event Timing

The MSP430F5529 has many timer units that count without CPU intervention. See Chapter 17 in the family reference manual
We used one of these timers to generate PWM signals on an output

One possibility to time an event is to:
1) start the event to time
2) Reset the timer value to 0 for example register TA0R. To start the timer set it up in TAxCTR register. It starts when MC bit is changed from 0 to 2 (continuous)
3) enter a loop that continuously checks to see if the event is finished
4) read the new timer value (overflows?)





Copyright © 2016, Texas Instruments Incorporated

**Figure 17-1. Timer_A Block Diagram**

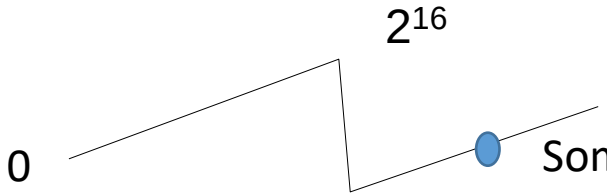
# Example part of the code

```
unsigned int time;
TA0CTL = TASSEL_2 + MC_2 + ID_0;
                    // Timer A control set to SMCLK, MC_2 count up, no division –
                    //keep 1.048MHz,
                      //start the event here
TAOR=0;              //Set timer counter to 0
                      // checking for the event to finish possibly with a while loop

time = TA0R;   //read the counter

// as long as there is no overflow, this gives the right answer
// timer must be configured to count all the way to 0xFFFF and not reset to 0 at TA0CCR0
//like when we used it for PWM)
//
```

# Event Timing - overflows

The timing registers are 16 bits long and run at approximately 1 MHz by default if fed by SM
means that any time longer than $2^{16}$ microseconds will create an overflow – the counter value go
zero and start again.

Event to time




Timer value

# Event Timing – Dealing with overflows

Two strategies:

1) Guarantee that the event to be timed is never longer than one timer period (2^16 counts). How? Hardware prescale so that the counter counts slowly enough (set clock divider with IDx bits in TAxCTL)

PRO: might be easiest solution.
CON: might limit resolution.

2) Keep track of overflows (interrupts, or count overflows)

PRO: get the full resolution possible
CON: more complicated code needed to track overflows

### 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<br>00b = TAxCLK<br>01b = ACLK<br>10b = SMCLK<br>11b = INCLK |
| 7-6 | ID | RW | 0h | Input divider. These bits along with the TAIDEX bits select the divider for the input clock.<br>00b = /1<br>01b = /2<br>10b = /4<br>11b = /8 |
| 5-4 | MC | RW | 0h | Mode control. Setting MC = 00h when Timer_A is not in use conserves power.<br>00b = Stop mode: Timer is halted<br>01b = Up mode: Timer counts up to TAxCCR0<br>10b = Continuous mode: Timer counts up to 0FFFFh<br>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.<br>0b = Interrupt disabled<br>1b = Interrupt enabled |
| 0 | TAIFG | RW | 0h | Timer_A interrupt flag<br>0b = No interrupt pending |

# General Advice:

Get pieces of a complicated system working one by one.

Ensure that each new piece doesn't break any old pieces.

When you combine programs make sure that all the setups are in the beginning. Avoid set up commands in the loops unless needed

Start with the simplest way, then if you have time and/or a good reason, move to a better way.

For example try to get the distance sensor working with 1 byte accuracy and than may be try for 2 bytes

The debugger, voltmeter and oscilloscope can be very helpful!