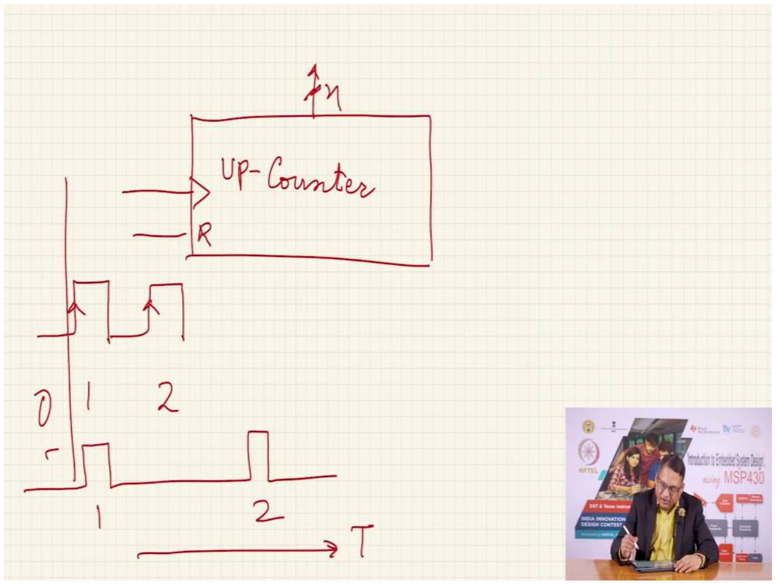
Introduction to Embedded System Design
Professor Dhananjay V. Gadre
Netaji Subhas University of Technology, New Delhi
Lecture 29
MSP430 Timer module: Introduction and Timer Capture

Hello and welcome to a new session for this online course on Introduction to Embedded system design. I am your instructor, Dhananjay Gadre and in this session I am going to cover a very very important aspects of embedded systems which is the ability to count time, measure time, count events so on and so forth.

And the basic building block peripheral, critical peripheral in any micro controller ecosystem is the ability to count time without the intervention of the micro, the CPU and such a peripheral is called Timer and often times it is called Counter/Timer which means it can count as well as count events as well as count time. The basic building block in counter timer is actually a counter. Now, what is a counter?

(Refer Slide Time: 01:20)

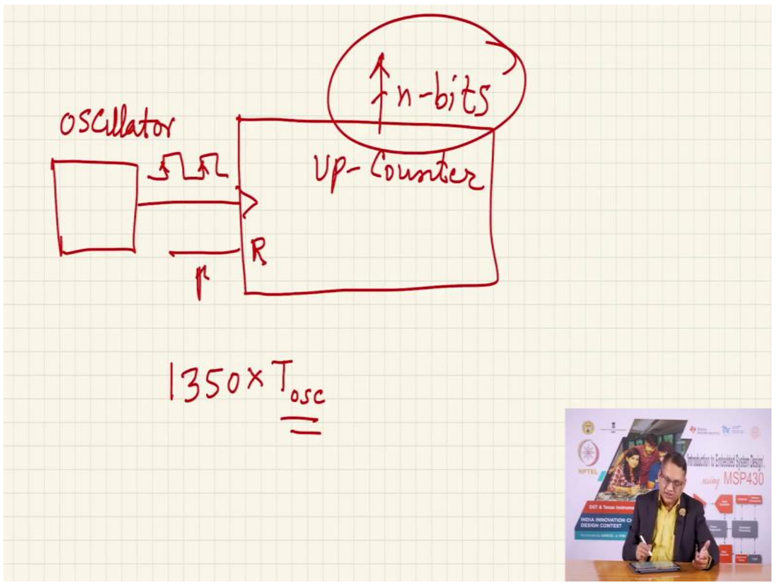


If I show you n-bit counter and typically what you will have n-bits going out. And you would have a clock input, may be you will have a reset pin so when will assert the reset signal, all the n outputs of this counter. So, this is a counter or reset to 0. Thereafter for the given configuration of the clock signal, every time there is a rising edge, the count will increment by 1.

If this was a down counter, then it will decrement by 1. So let us say, this is an up counter and so on. So, if from here if the initial value before this was 0 then and this was an up counter, the count will be 1 here and it will be 2 here. Now, whether the signals which are applied to the clock are like this or like this.

Here it will be one and it will be 2 at this point of time. these are the time and these are the pulses you are getting. If the frequency of the signal which is connected to the clock pin of the counter. We do not know the frequency then we cannot say that how much time is elapsed. All we can say how many events have come, how many clock signals have come and in such a case you are operating this building block as a counter.

(Refer Slide Time: 02:50)

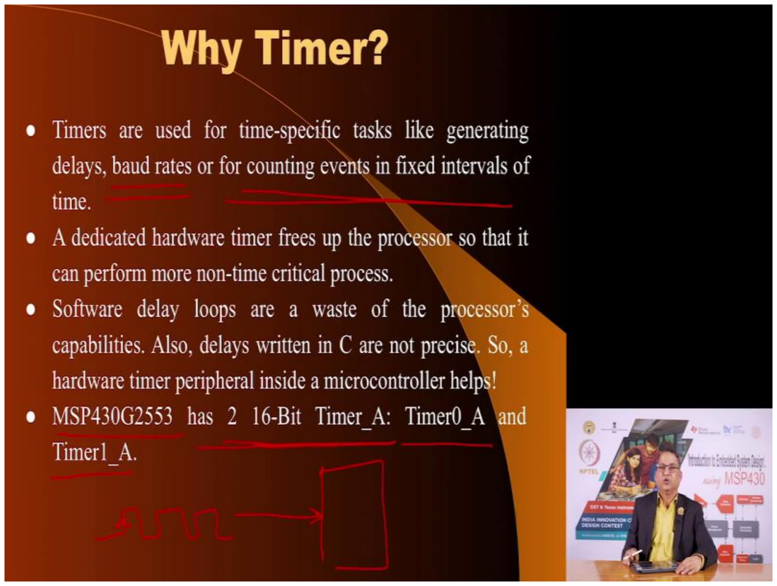


On the other hand, if I use the same hardware. I still have a n-bit counter, n-bits and let us say it is a up-counter. You have a clock signal, you have reset signal but now instead of an external signal which is generating some pulses, if this connected to a oscillator, rectangular output oscillator something like this whose frequency is fixed and after reset assert reset on this pin. It will count these pulses.

And if I query, if I am able to read this n-bits and I find let us say the count is 1030 1350. If I find that the count is 1350, then I can also tell how much time is elapsed since the time I reset this counter. How much time is elapsed? It is 1350 into the T oscillator that is the period of the clock signal. Once I know this by performing this small calculation, I would be able to estimate the time it has elapsed since the time I reset this counter.

So, the basic building block in a micro controller is the time is the counter. If you connect the clock input of that counter to that external pin and expect external signals to be used for clocking the counter, you are maintaining a counter. On the other hand, if the clock input of this counter is derived from an internally generated clock then and if you know the frequency of that clock then you would have converted this counter into a timer. And MSP430 is immensely capable in providing various functions of this timer counter. Let us investigate what all we have.

(Refer Slide Time: 05:12)



Why Timer?

- Timers are used for time-specific tasks like generating delays, baud rates or for counting events in fixed intervals of time.
- A dedicated hardware timer frees up the processor so that it can perform more non-time critical process.
- Software delay loops are a waste of the processor's capabilities. Also, delays written in C are not precise. So, a hardware timer peripheral inside a microcontroller helps!
- MSP430G2553 has 2 16-Bit Timer_A: Timer0_A and Timer1_A.

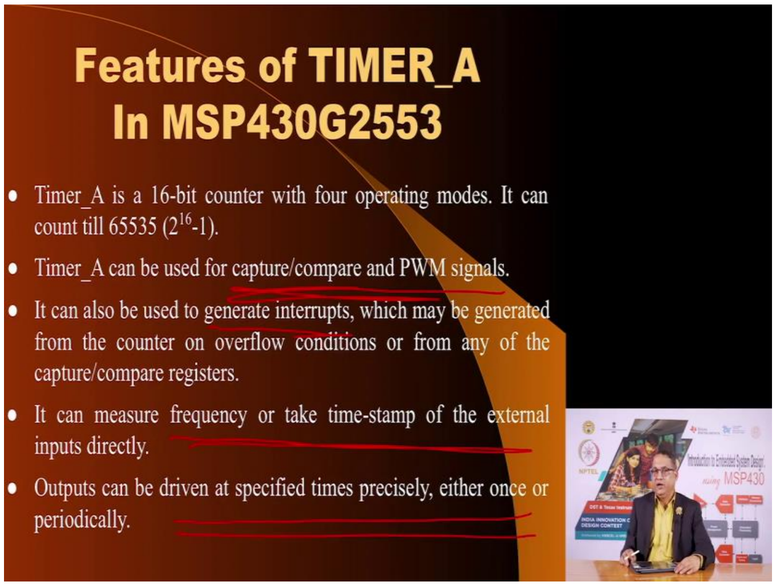
Now, why do we need timers? We can also use the program to count time. The reason why we want to use timers is that we can use the peripheral function to generate timing events. We can use them for generating delays. We can use them for creating what we called as baud rates which we are going to cover in a serial communication lecture subsequently. Or you can use these timers as mentioned for counting the external events and so on.

Having a dedicated hardware for performing this function frees the CPU from performing the task. You can also use the CPU, for example one of the pod pins could be connected to a external signal where you are getting pulses and you can write a program saying, if this pulses high wait for it to be low. Once it goes low, one pulse has been received you can count it and so on. But if you did that your entire CPU life is going to be spent only counting pulses, you would not be able to do any other activity at the same time.

By off loading this counting business to a counter, your CPU can perform other functions and keep on finding out from the counter. How much, how many events have happened. This is what the infrastructure related to a counter and a timer micro controller usually allows you to do. And MSP430 is no exception. If you use a program to use generate leads as we have seen how we can write a delay function, again the delay function depends on the accuracy of the CPU clock. And if it is not very accurate the delay period will not be very accurate.

On the other hand, if you use the timer, you can use a external crystal oscillator like we have on the MSP430 to provide clock to the timer and then it can generate very accurate delay periods. MSP430G2553 has 2 16-bit Timer A and they are label timer as 0A and Timer1A. and this is over and above the watchdog timer that it has which can also function as a Timer.

(Refer Slide Time: 07:18)



Features of TIMER_A In MSP430G2553

- Timer_A is a 16-bit counter with four operating modes. It can count till 65535 (2^{16}-1).
- Timer_A can be used for capture/compare and PWM signals.
- It can also be used to generate interrupts, which may be generated from the counter on overflow conditions or from any of the capture/compare registers.
- It can measure frequency or take time-stamp of the external inputs directly.
- Outputs can be driven at specified times precisely, either once or periodically.

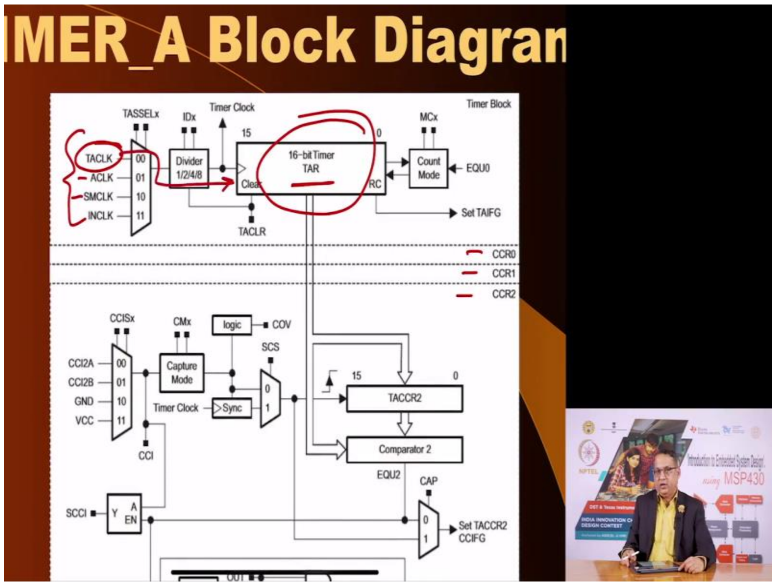
The timers on the MSP430 are 16-bit counter which means they can count from 0 to 65535 which means they can count a very large number. The timers can be used for capture and compare functions. I will come to this feature shortly. And it can also used to generate Pulse Width Modulation signal which we have briefly mentioned in a previous lecture.

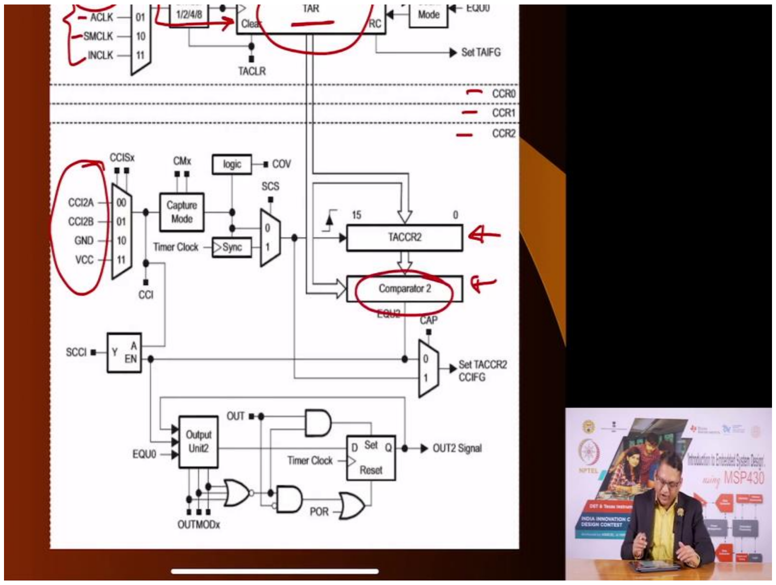
The timers can be used to generate interrupts on a timely fashion. You want to generate interrupt every one second or every 100 mille seconds, you can employ a timer by appropriately programming it, it can generate very accurate interrupt events which you can used to do some housekeeping jobs.

You remember in a previous lecture we said, we want to connect 4 7-segment displays and we want to multiplex them and multiplexing requires that you refresh the display which means one digit which was on you have to turn it off. And after a pre-determined period of time you have to turn second digit on.

Now, you can do the job with the help of the interrupted generated by the timer. You can also use the timer infrastructure to measure frequency of the external signals. You can use it to measure period of external pulses. And you can use it to output information on pins on pre-determined intervals. So, these are the features of the timer.

(Refer Slide Time: 08:46)





Here is a block diagram and I am going to broadly tell you that any timer has actually two parts the timer part that is the counterpart and the rest of the parts which allow you to compare the value of the timer to pre-determined value. So, that you know you have started the timer from 0, I want the notification when the count reaches 5000. So, you are going to compare the value of the timer with the value in a register which you write 5000 whenever that match happens, it will inform you.

So, here is the 16-bit timer. This is called TAR which means Timer register and then you have 3 compare capture channels, they are called CCR0 Capture Compare Register 0. Capture Compare Register 1 and Capture Compare Register 2. So, with one timer, you have 3 channels of capture and compare features. Of course, you can either capture or compare and by choosing a appropriate function whether we want to capture or compare.

What is the meaning of capture? You have to capture the value in the timer. What is the meaning of the compare? You have to compare the value of the timer register with the pre-determined value stored in that compare register. And using these two compare and capture registers; you can do amazing things with the timer.

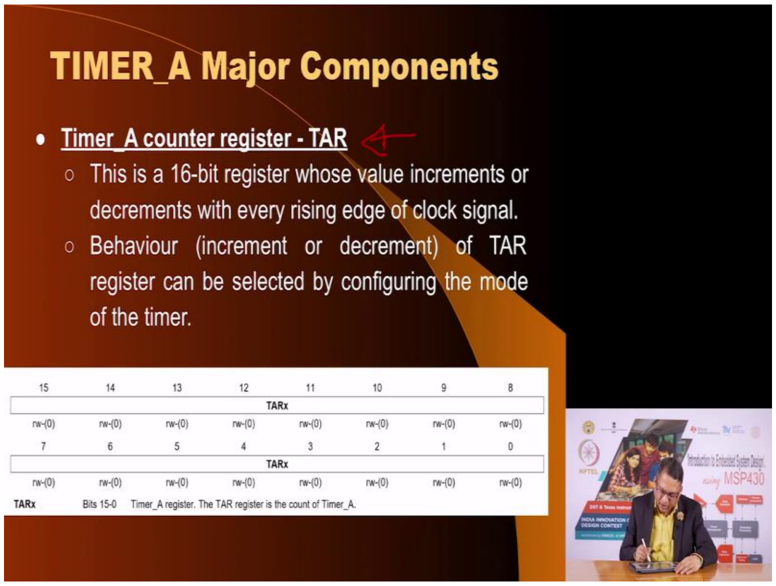
The for this counter, although it says the timer as I mentioned. It can basic building block is the counter. The clock of the timer can be derived from these sources as you see here, you can have the A- clock, oscillatory clock or you can have SM clock and TA clock in this case is actually a pin which means the signal of the pin is used to clock the counter and these feature can be used when you want to count the external pulses, when you want to count external events.

You may remember in one of the previous lectures, suppose in this room we have to count how many people have entered, you can easily install the censor which detects human presence and when it detects the human presence it will generate a pulse. This pulse can be connected to the TA clock pin and it will increment the timer. And you can query the timer at any given point of time after resetting at onset that what is the value at this point of time and you know how many people have entered the room.

So, this is then A clock and SM clock allows you to operate the timer in the timer mode. Why? Because A clock and SM clock is locally generated clock signals and you know the frequency of these clock generator. So, the count in the timer will tell you how much time is elapsed, since time you have started the timer.

Now, when do you compare and when do you capture, you can do that based on signals on micro-controller as well as events within the micro-controller. You can use it to capture the value of the timer. You can use it to compare the value of the timer, using the value written in the comparator. And are going to use this feature to see various activities. So, this is the basic block diagram. You have another timer so this is TimerA0 and TimerA1. And both of them have these three channels which allow you to do multiple things.

(Refer Slide Time: 12:35)



TIMER_A Major Components

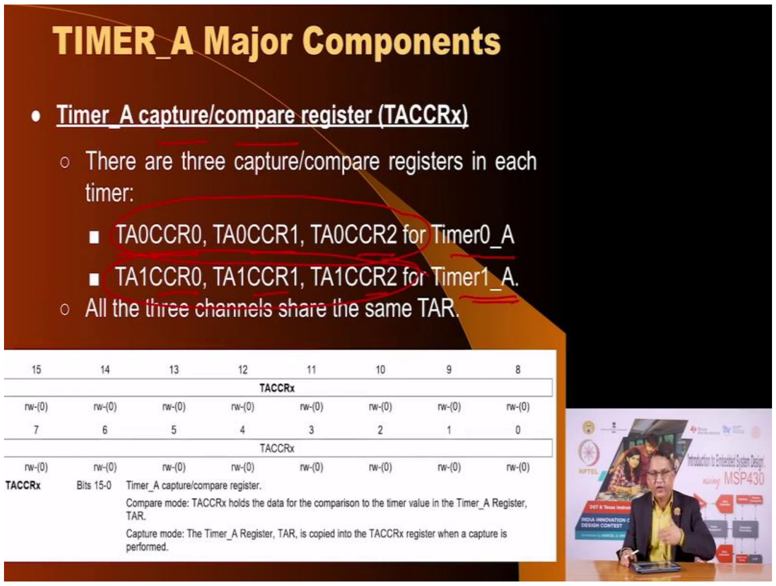
- **Timer_A counter register - TAR**
  - This is a 16-bit register whose value increments or decrements with every rising edge of clock signal.
  - Behaviour (increment or decrement) of TAR register can be selected by configuring the mode of the timer.

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

TARx	Bits 15-0	Timer_A register. The TAR register is the count of Timer_A.

So, the major components of the timer are Timer Counter register as we have seen TAR in both the timers we had they all 16 bit registers. They all read-write registers.

(Refer Slide Time: 12:48)



TIMER_A Major Components

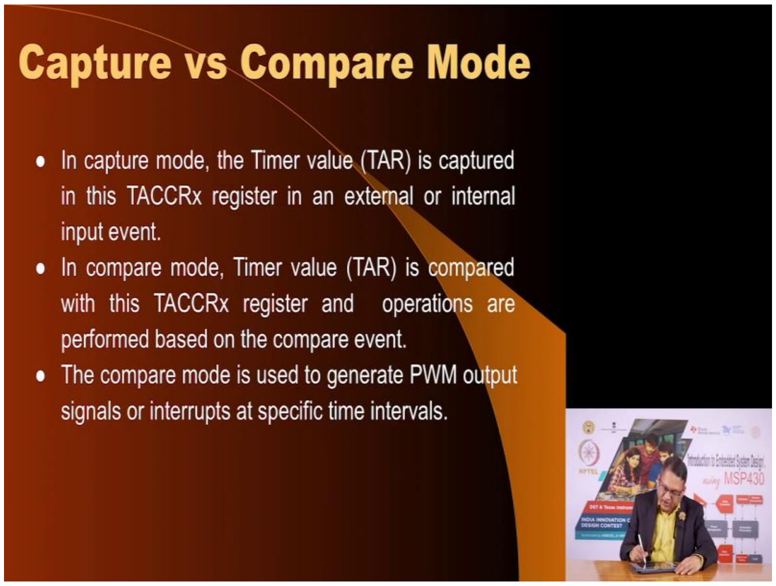
- **Timer_A capture/compare register (TACCRx)**
  - There are three capture/compare registers in each timer:
    - TA0CCR0, TA0CCR1, TA0CCR2 for Timer0_A
    - TA1CCR0, TA1CCR1, TA1CCR2 for Timer1_A.
  - All the three channels share the same TAR.

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

TACCRx	Bits 15-0	Timer_A capture/compare register. Compare mode: TACCRx 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 TACCRx register when a capture is performed.

You also have timer capture and compare register and there are three capture and compare register for each of the timer. Timer A0CCR0 Timer capture compare register 0, Capture compare register 1, register 2 for Timer 0A and capture compare register 0, 1 and 2 for Timer 1A. And both these three channels for say for this will use a given timer. These three channels will use a second timer register and you can write values in this, you can read values from this to know whether you want to capture or you want to compare and so on.

(Refer Slide Time: 13:38)

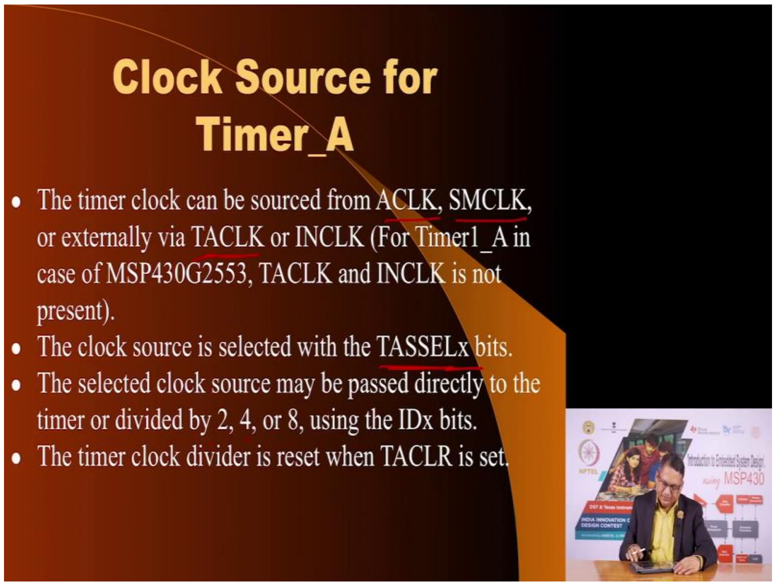


Capture vs Compare Mode

- In capture mode, the Timer value (TAR) is captured in this TACCRx register in an external or internal input event.
- In compare mode, Timer value (TAR) is compared with this TACCRx register and operations are performed based on the compare event.
- The compare mode is used to generate PWM output signals or interrupts at specific time intervals.

In the capture mode, the value of the timer is stored and captured in the capture register and you can read it thereafter. In the compare mode, you load a value in the register and wait for timer value to reach that number. When that number is reached some event will happen. And usually a flag will be set and if you want a interrupt will be generated and these feature is used to generate Pulse width modulated signal.

(Refer Slide Time: 14:08)

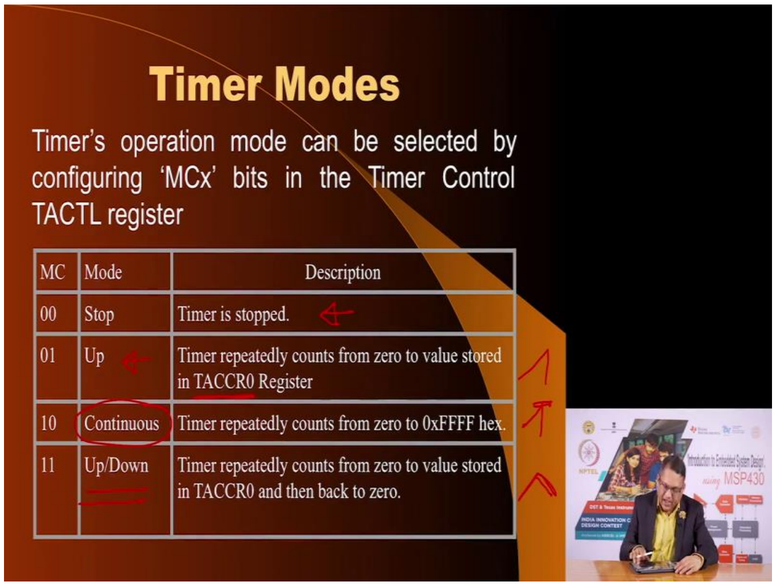


Clock Source for Timer_A

- The timer clock can be sourced from ACLK, SMCLK, or externally via TACLK or INCLK (For Timer1_A in case of MSP430G2553, TACLK and INCLK is not present).
- The clock source is selected with the TASSELx bits.
- The selected clock source may be passed directly to the timer or divided by 2, 4, or 8, using the IDx bits.
- The timer clock divider is reset when TACLR is set.

The source of the timer can be A clock or SM clock and you can source it from an external pin like TA clock. The clock source is selected with the help of the registers in this these register bits and once you have selected the source of clock, you can also further divide it, so you can divide it by 1, 2, 4 or 8 and so on.

(Refer Slide Time: 14:35)



Timer Modes

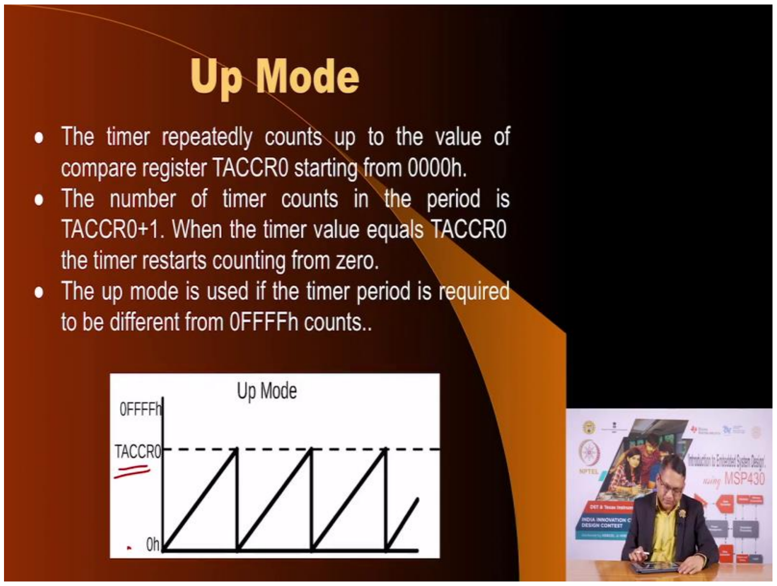
Timer's operation mode can be selected by configuring 'MCx' bits in the Timer Control TACTL register

MC	Mode	Description
00	Stop	Timer is stopped.
01	Up	Timer repeatedly counts from zero to value stored in TACCR0 Register
10	Continuous	Timer repeatedly counts from zero to 0xFFFF hex.
11	Up/Down	Timer repeatedly counts from zero to value stored in TACCR0 and then back to zero.

These are the modes of the timer, the basic timer that is the TAR works in 4 modes. You can stop the timer which means once you stop it, it can hold the value. Or you can use it in the up mode means it can count from 0 and it will keep on counting till a highest value but the highest value in the capture compare register are 0. The other second mode of operation is what is called as the continues mode in which it will go from 0 count upto the maximum count which is FFF and again reset to 0 and again keep on counting.

And then you have an up down mode where it goes from 0, counts upto maximum value and decrements from maximum value to 0. And so on and so forth. So, you will see in up count it will goes like a ram and in the continuous mode it also goes to ram but the value the maximum value is FFF. In the up down mode it goes like a triangle way form. Will see this in diagrams here.

(Refer Slide Time: 15:40)

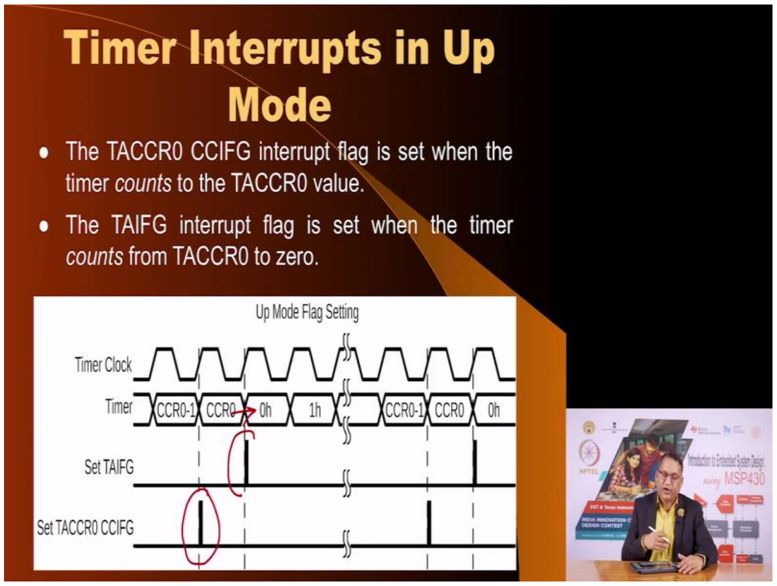


Up Mode

- The timer repeatedly counts up to the value of compare register TACCR0 starting from 0000h.
- The number of timer counts in the period is TACCR0+1. When the timer value equals TACCR0 the timer restarts counting from zero.
- The up mode is used if the timer period is required to be different from 0FFFFh counts..

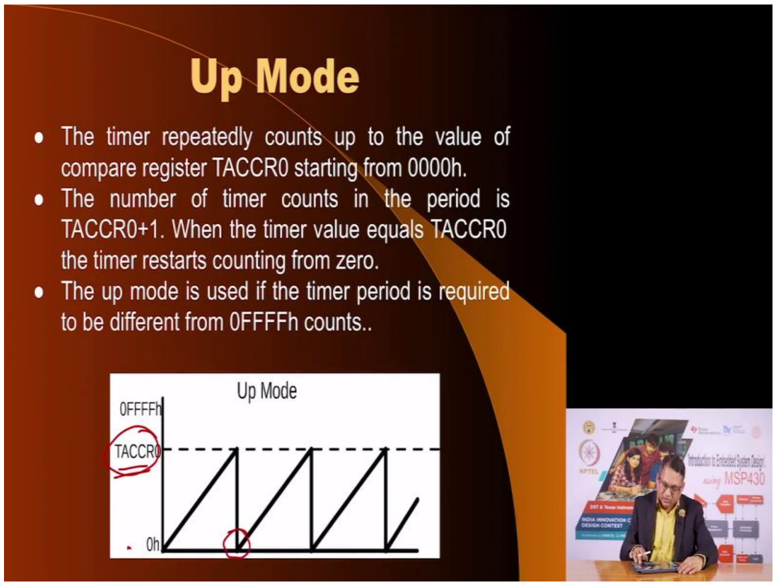
So, this is the up mode, it goes from 0 but the maximum value you set in the Timer A capture compare register 0. So, the point to note is the value in capture compare register is very important. That is the register which decides upto what value the timer will count.

(Refer Slide Time: 16:02)



Timer Interrupts in Up Mode

- The TACCR0 CCIFG interrupt flag is set when the timer *counts* to the TACCR0 value.
- The TAIFG interrupt flag is set when the timer *counts* from TACCR0 to zero.

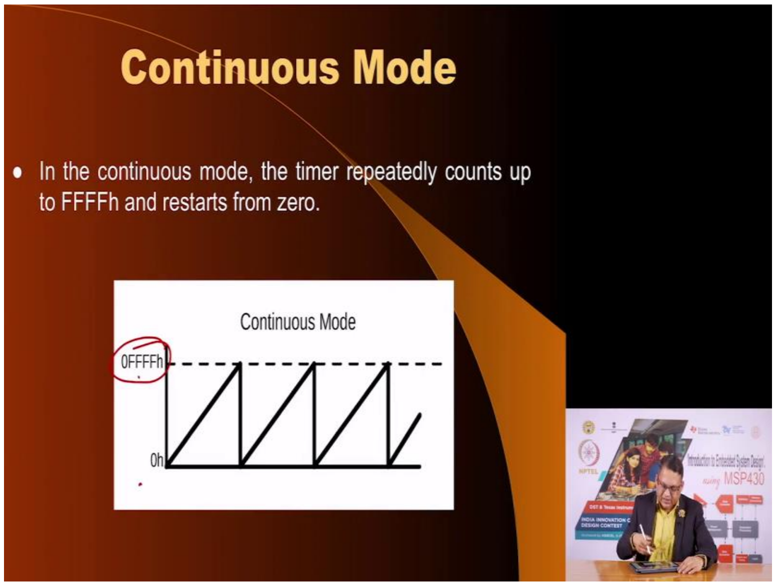


Up Mode

- The timer repeatedly counts up to the value of compare register TACCR0 starting from 0000h.
- The number of timer counts in the period is TACCR0+1. When the timer value equals TACCR0 the timer restarts counting from zero.
- The up mode is used if the timer period is required to be different from 0FFFFh counts..

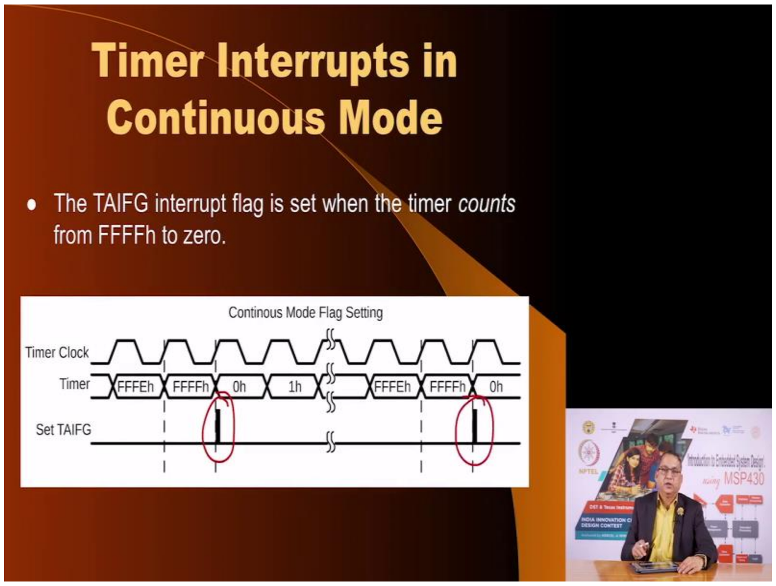
In this mode, you can use it to generate interrupt at two times. One once when it reaches this value and once when it resets to 0, when it become 0. As you see here, you have generate a interrupt here when it goes from here to here and when it is here. So, there are two times this counting process when interrupts can be generated.

(Refer Slide Time: 16:30)



Continuous Mode

- In the continuous mode, the timer repeatedly counts up to FFFFh and restarts from zero.



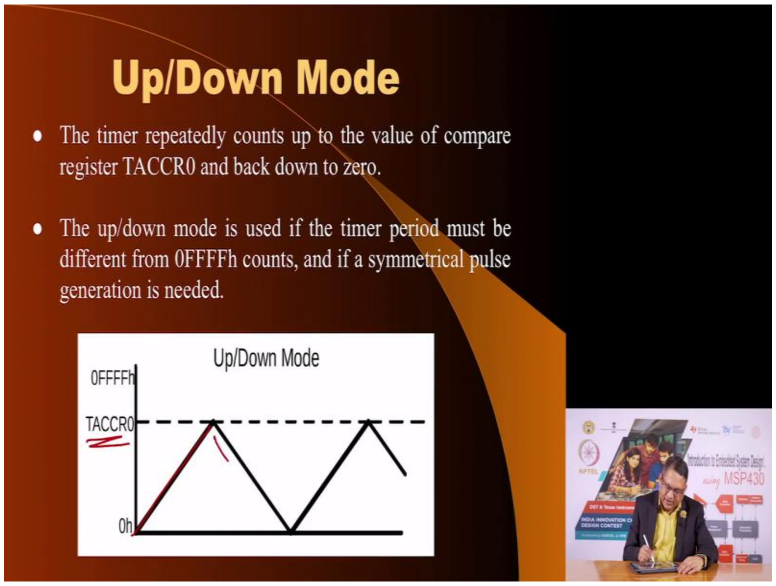
Timer Interrupts in Continuous Mode

- The TAIFG interrupt flag is set when the timer *counts* from FFFFh to zero.

Continous Mode Flag Setting
Timer Clock
Timer
FFFEh
FFFFh
0h
1h
FFFEh
FFFFh
0h
Set TAIFG
Introduction to Embedded System Design using MSP430
NPTEL

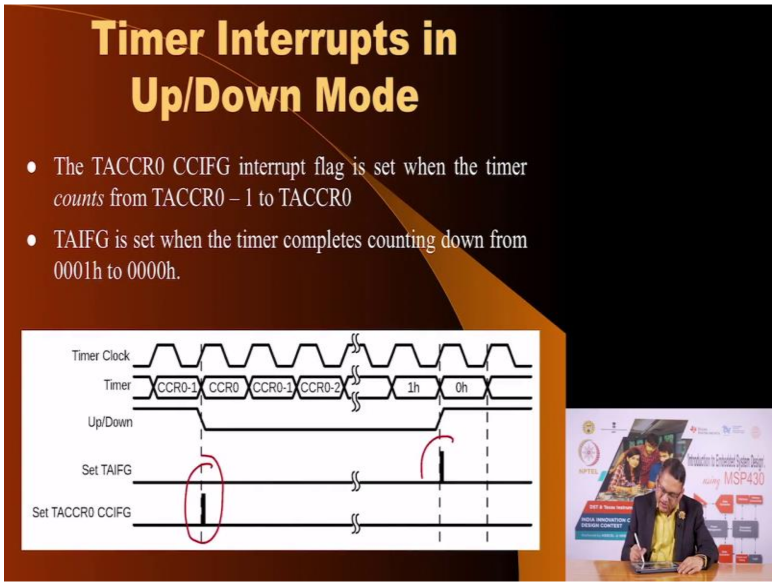
Then you have the continuous mode, very similar to the continuous mode except the value the maximum value will be this. It will start from 0 and it will go to maximum capability of the timer and it will reset to 0. And when it resets to 0, it will generate an interrupt as you see here. When it resets to 0, it is generating an interrupt.

(Refer Slide Time: 16:53)



Up/Down Mode

- The timer repeatedly counts up to the value of compare register TACCR0 and back down to zero.
- The up/down mode is used if the timer period must be different from 0FFFFh counts, and if a symmetrical pulse generation is needed.

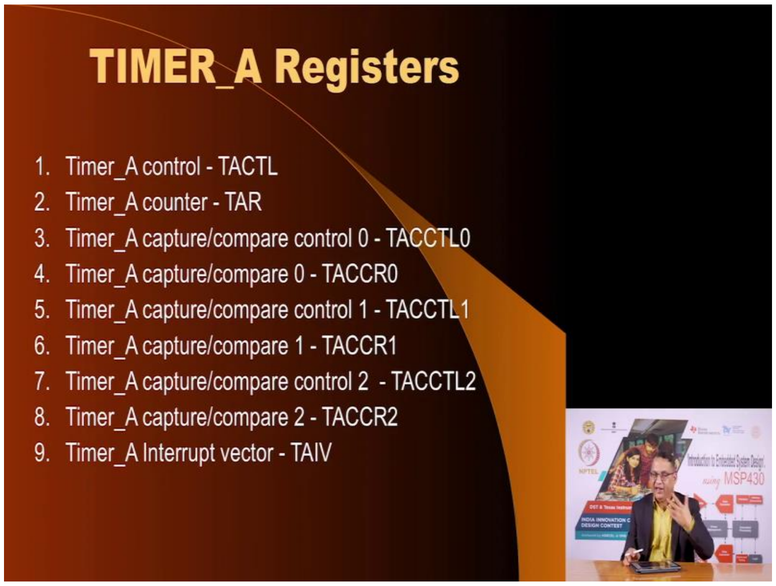


Timer Interrupts in Up/Down Mode

- The TACCR0 CCIFG interrupt flag is set when the timer *counts* from TACCR0 – 1 to TACCR0
- TAIFG is set when the timer completes counting down from 0001h to 0000h.

In the up down mode, it goes from 0 to the maximum value set in this register. And once it reaches there it decrements it till it reached 0. In this mode, it is going to generate when it is just reaching that and when it is reaching 0.

(Refer Slide Time: 17:13)

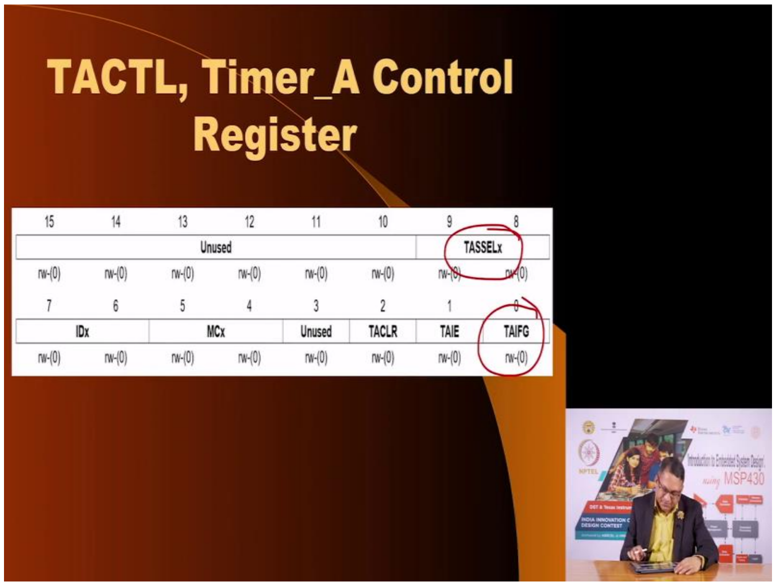


TIMER_A Registers

1. Timer_A control - TACTL
2. Timer_A counter - TAR
3. Timer_A capture/compare control 0 - TACCTL0
4. Timer_A capture/compare 0 - TACCR0
5. Timer_A capture/compare control 1 - TACCTL1
6. Timer_A capture/compare 1 - TACCR1
7. Timer_A capture/compare control 2 - TACCTL2
8. Timer_A capture/compare 2 - TACCR2
9. Timer_A Interrupt vector - TAIV

These are the registers associated with the timer. And I strongly recommend you to go through the data sheet to understand each of these. The best understanding will come when you write a program and play with the various features.

(Refer Slide Time: 17:26)

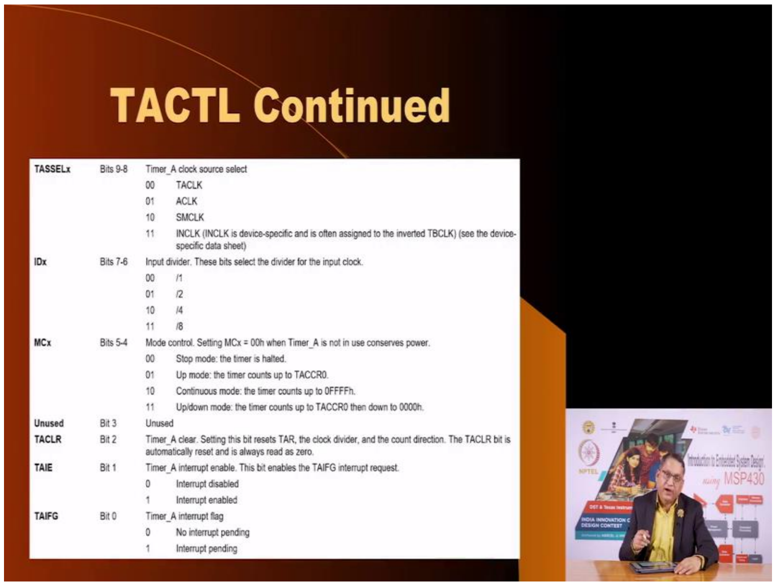


TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

This is the timer control register. And as you see these are the bits which allow you to turn the timer on and off. These are the bits which allow you to unable the interrupts.

(Refer Slide Time: 17:41)

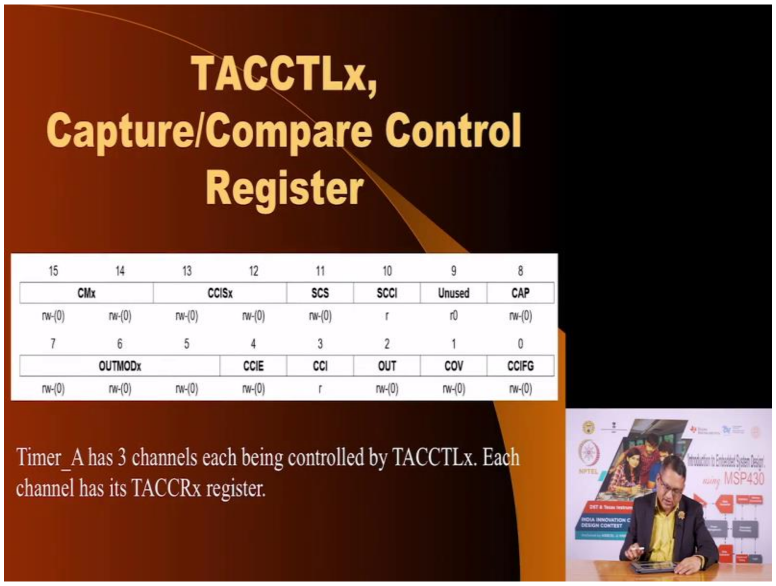


TACTL Continued

TASSELx	Bits 9-8	Timer_A clock source select	
		00	TACLK
		01	ACLK
		10	SMCLK
		11	INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.	
		00	/1
		01	/2
		10	/4
		11	/8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.	
		00	Stop mode: the timer is halted.
		01	Up mode: the timer counts up to TACCR0.
		10	Continuous mode: the timer counts up to 0FFFFh.
		11	Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused	
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.	
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.	
		0	Interrupt disabled
		1	Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag	
		0	No interrupt pending
		1	Interrupt pending

I recommend you again to go through these registers.

(Refer Slide Time: 17:46)



TACCTLx, Capture/Compare Control Register

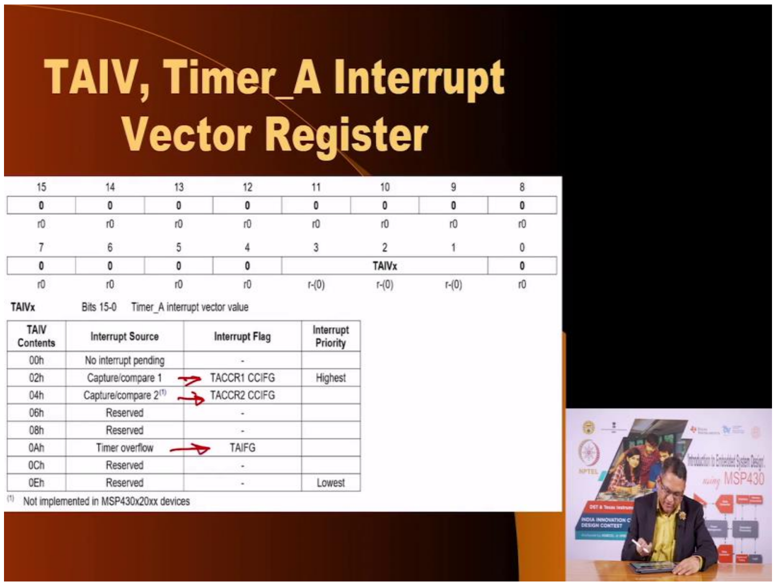
15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

Timer_A has 3 channels each being controlled by TACCTLx. Each channel has its TACCRx register.

This is the capture control register. This are the values, bits which you can decide which mode of the timer whether it is going to operate in the capture mode or the compare mode.

(Refer Slide Time: 18:07)



TAIV, Timer_A Interrupt Vector Register

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	0	0		TAIVx		0
r0	r0	r0	r0	r-(0)	r-(0)	r-(0)	r0

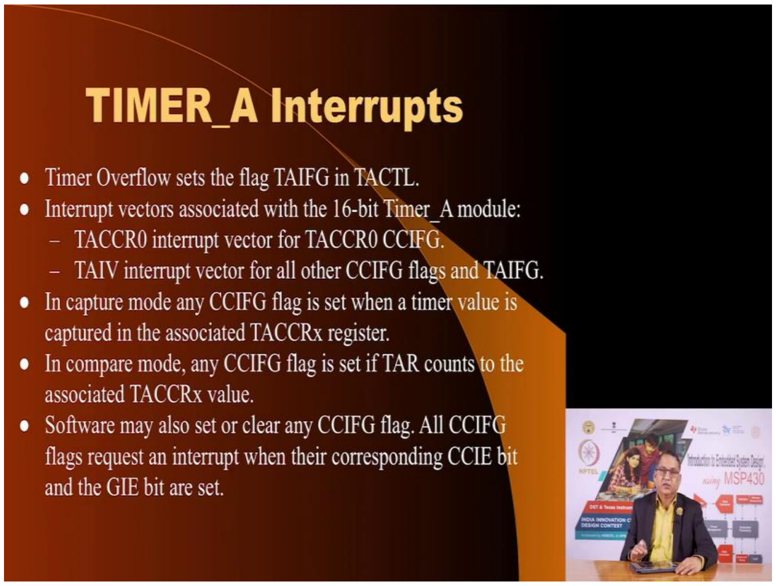
TAIVx Bits 15-0 Timer_A interrupt vector value

TAIV Contents	Interrupt Source	Interrupt Flag	Interrupt Priority
00h	No interrupt pending	-	
02h	Capture/compare 1	TACCR1 CCIFG	Highest
04h	Capture/compare 2[1]	TACCR2 CCIFG	
06h	Reserved	-	
08h	Reserved	-	
0Ah	Timer overflow	TAIFG	
0Ch	Reserved	-	
0Eh	Reserved	-	Lowest

[1] Not implemented in MSP430x20xx devices

And this is the interrupt vector for the interrupts that will be generated because of the Timer A operation and as you see, many of the interrupts are shared. Here is the interrupt of capture compare 1 and this is the interrupt of the capture compare 2. This is the interrupt of the timer overflow for a given timer.

(Refer Slide Time: 18:30)

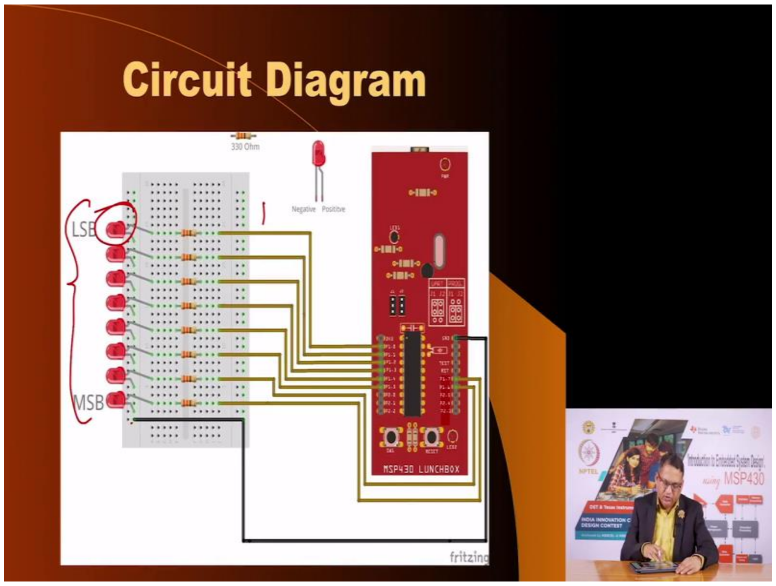


TIMER_A Interrupts

- Timer Overflow sets the flag TAIFG in TACTL.
- Interrupt vectors associated with the 16-bit Timer_A module:
  - TACCR0 interrupt vector for TACCR0 CCIFG.
  - TAIV interrupt vector for all other CCIFG flags and TAIFG.
- In capture mode any CCIFG flag is set when a timer value is captured in the associated TACCRx register.
- In compare mode, any CCIFG flag is set if TAR counts to the associated TACCRx value.
- Software may also set or clear any CCIFG flag. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are set.

And there are many many sources of interrupts for these timers. I again very strongly recommend you that you go through the slides as well as the data sheets.

(Refer Slide Time: 18:42)

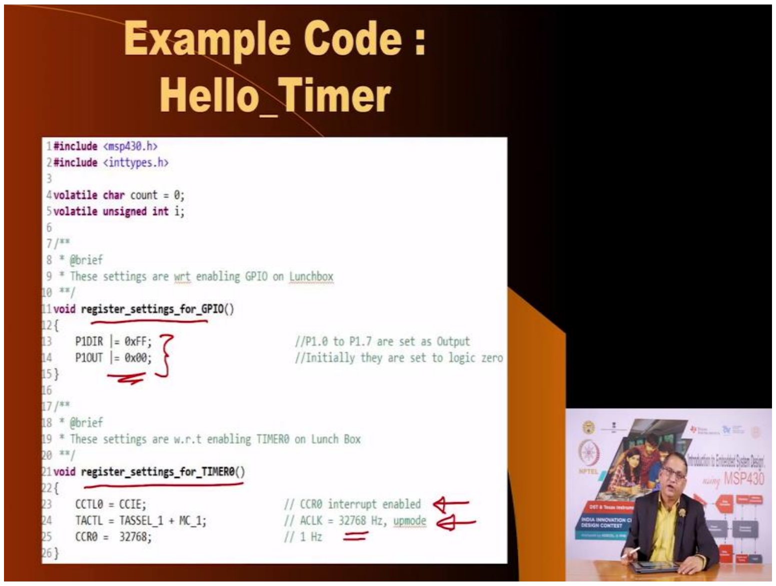

Circuit Diagram

Now, we are going to run a little program to illustrate what can be achieved with the timer. Now, in a previous exercise, we saw that we can use a delay function to toggle LED at a certain rate by changing the delay period, you can slow down the toggling rate or you can make it fast. Here, we can do that except we are not just toggling a single LED, we are actually counting, the count on 8 LEDs which we connected to port 1 bits.

We have connected 8 LEDs, as you see this part of the LED is grounded all these LEDs are grounded which means whenever this bit is one that LED is going to glow. Now, we are going to have a internal counter which will count from 0 to 255. When the value 0 all the LED will be off. When it is 1, one LED will be on, when it is 2, the first LED will turn off, the second will turn on and so on and so forth. So, you will you are going to see the counter value in a binary form.

But who is changing the count, the count is changing at every one second but that one second duration is not being achieved by some delay loop. And instead we have used a timer to generate an interrupt at every one second. And at every one second interval, we are incrementing the count and outputting the count onto the port 1 register. So, that you will see the LEDs blinking or rather counting information at the rate of one second. There is no other interaction; there is no switch anything like that. Once you load the program and download it and power it up, you will see the LEDs blinking.

(Refer Slide Time: 20:40)



Example Code : Hello_Timer

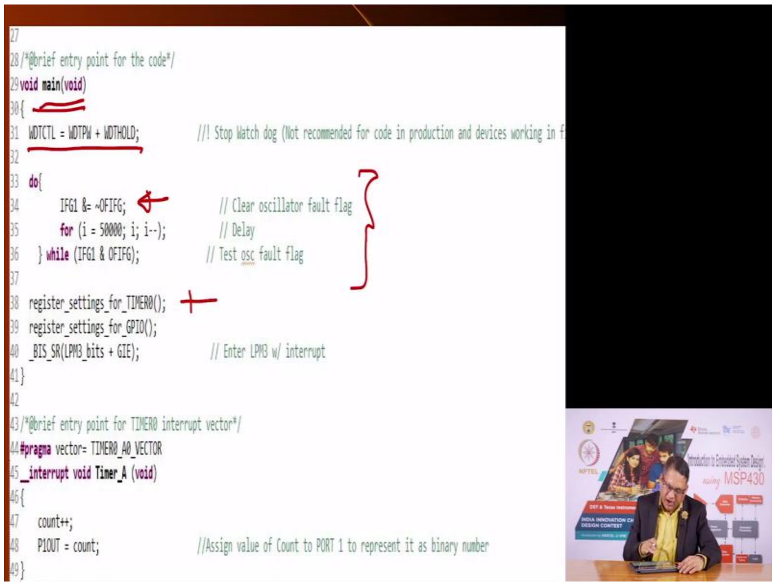
```
#include <msp430.h>
#include <inttypes.h>

volatile char count = 0;
volatile unsigned int i;

/**
 * @brief
 * These settings are wrt enabling GPIO on Lunchbox
 **/
void register_settings_for_GPIO()
{
    P1DIR |= 0xFF;                         //P1.0 to P1.7 are set as Output
    P1OUT |= 0x00;                         //Initially they are set to logic zero
}

/**
 * @brief
 * These settings are w.r.t enabling TIMER0 on Lunch Box
 **/
void register_settings_for_TIMER0()
{
    CCTL0 = CCIE;                        // CCR0 interrupt enabled
    TACTL = TASSEL_1 + MC_1;             // ACLK = 32768 Hz, upmode
    CCR0 =  32768;                       // 1 Hz
}
```

So, here in this program, we have certain registers, we have certain sub-routines. These sub-routines simply determines that all the pins of the port 1 should be set as output and their initial value set to 0. So, that when you turn on the values will be 0. Here, we are choosing the timer various modes of timer 0. What we are doing is, we are saying that we want to generate a interrupt on capture compare register 0.

The clock for the timer is A clock. And A clock is source from the low frequency crystal as you know on our MSP430 lunchbox, there is 32.768 kilohertz crystal. The frequency of that crystal will be use to drive this counter, drive this timer and we want to compare when that count reaches 32768 by loading the capture counter register to 32768.

When it reaches 32768 that means 32,768 pulses from the oscillator have elapsed. And since the frequency is 32.768 kilohertz that means one second has elapsed. Upon that event it will generate an interrupt.

(Refer Slide Time: 22:06)

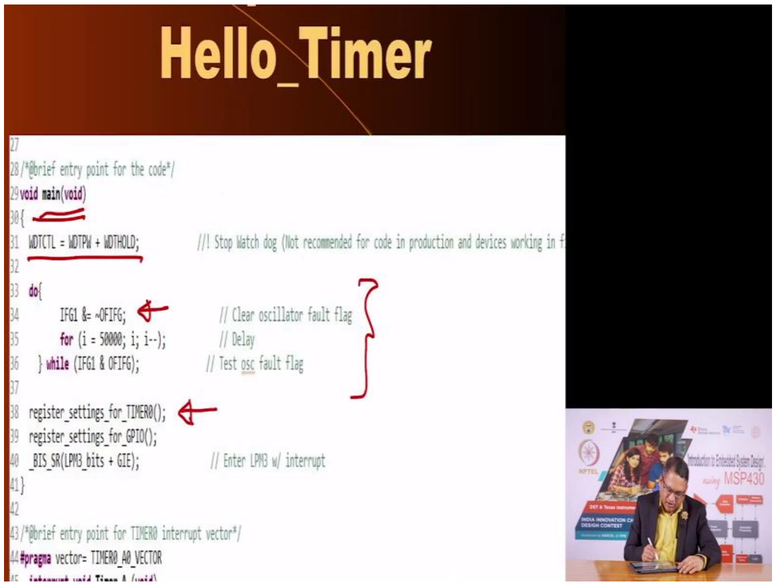


```
/*@brief entry point for the code*/
void main(void)
{
 WDTCTL = WDTPW + WDTHOLD;          //! Stop Watch dog (Not recommended for code in production and devices working in f

 do{
      IFG1 &= ~OFIFG;               // Clear oscillator fault flag
      for (i = 50000; i; i--);      // Delay
   } while (IFG1 & OFIFG);          // Test osc fault flag

 register_settings_for_TIMER0();
 register_settings_for_GPIO();
 _BIS_SR(LPM3_bits + GIE);          // Enter LPM3 w/ interrupt
}

/*@brief entry point for TIMER0 interrupt vector*/
#pragma vector= TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
   count++;
   P1OUT = count;                   //Assign value of Count to PORT 1 to represent it as binary number
}
```

See let us see the program, what we are doing as we always do. This is the main void and we have disabled the watchdog timer. Now, you remember when we were discussing about the clocks, the ACLK clock derived out of low frequency oscillator as well as low frequency low frequency crystal oscillator. Both have to start working.

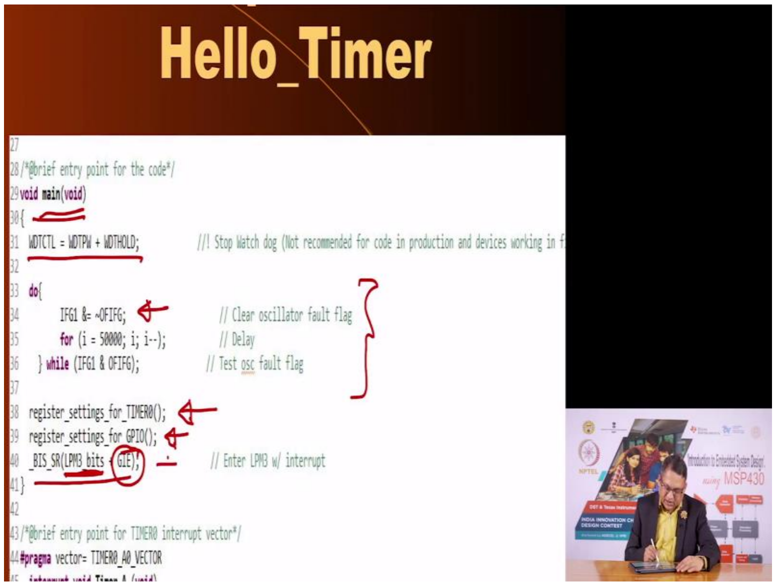
They are not very fast oscillators which means they will not start working right away after reset. It may take some time to oscillators to build and stabilize and to do that you have to check for this flag as we have seen in the previous lecture. We are just repeating that code here. We are waiting in this loop to make sure that our oscillator is stable, was that oscillator stable. We are calling this sub routine which we have seen previously here. Here we are here we are feeding that A clock to timer 0 A0.

(Refer Slide Time: 23:08)



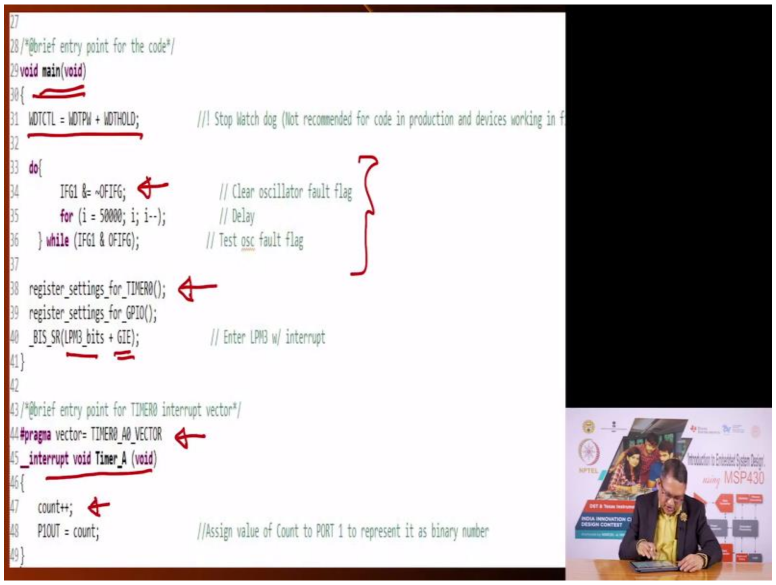
Hello_Timer

```
/*@brief entry point for the code*/
void main(void)
{
WDTCTL = WDTPW + WDTHOLD;          //! Stop Watch dog (Not recommended for code in production and devices working in f

do{
      IFG1 &= ~OFIFG;              // Clear oscillator fault flag
      for (i = 50000; i; i--);     // Delay
  } while (IFG1 & OFIFG);          // Test osc fault flag

register_settings_for_TIMER0();
register_settings_for_GPIO();
_BIS_SR(LPM3_bits + GIE);          // Enter LPM3 w/ interrupt
}

/*@brief entry point for TIMER0 interrupt vector*/
#pragma vector= TIMER0_A0_VECTOR
```



Hello_Timer

```
/*@brief entry point for the code*/
void main(void)
{
WDTCTL = WDTPW + WDTHOLD;          //! Stop Watch dog (Not recommended for code in production and devices working in f

do{
      IFG1 &= ~OFIFG;              // Clear oscillator fault flag
      for (i = 50000; i; i--);     // Delay
  } while (IFG1 & OFIFG);          // Test osc fault flag

register_settings_for_TIMER0();
register_settings_for_GPIO();
_BIS_SR(LPM3_bits + GIE);          // Enter LPM3 w/ interrupt
}

/*@brief entry point for TIMER0 interrupt vector*/
#pragma vector= TIMER0_A0_VECTOR
```

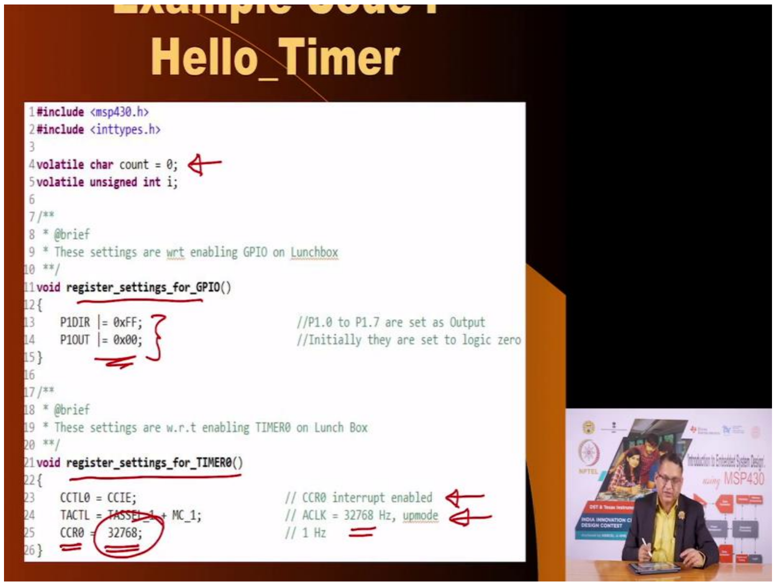


```
/*@brief entry point for the code*/
void main(void)
{
 WDTCTL = WDTPW + WDTHOLD;          //! Stop Watch dog (Not recommended for code in production and devices working in f

 do{
      IFG1 &= ~OFIFG;               // Clear oscillator fault flag
      for (i = 50000; i; i--);      // Delay
   } while (IFG1 & OFIFG);          // Test osc fault flag

 register_settings_for_TIMER0();
 register_settings_for_GPIO();
 _BIS_SR(LPM3_bits + GIE);          // Enter LPM3 w/ interrupt
}

/*@brief entry point for TIMER0 interrupt vector*/
#pragma vector= TIMER0_A0_VECTOR
_interrupt void Timer_A (void)
{
  count++;
  P1OUT = count;                    //Assign value of Count to PORT 1 to represent it as binary number
}
```

And then we are calling this sub routine to make all the bits of port 1 to be turned off. Then what we are doing, then we are entering, we have loaded a value into starters register to unable the global inter flag. This is very important. But we also have chosen to use low power 3. Low power mode 3, this will disable everything except, it will keep on keep the A clock active. And since we have chosen A clock to feed the timer which means timer will keep on working.

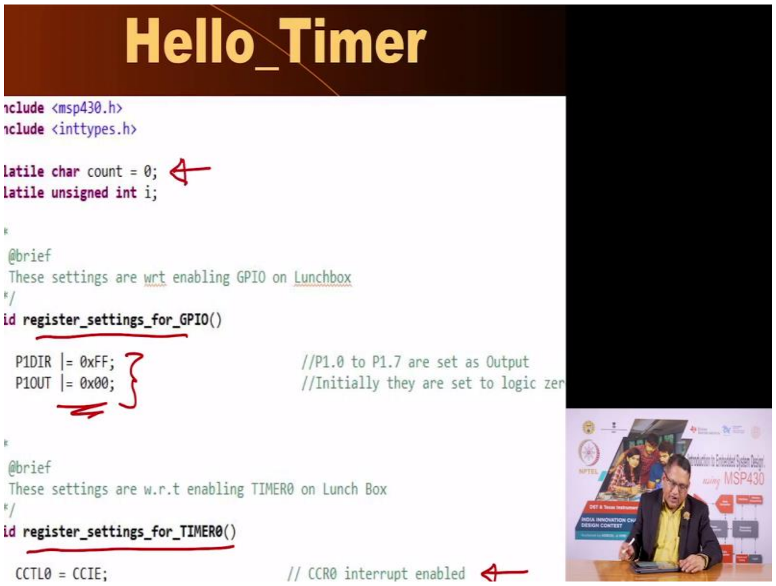
Also, the peripheral such as the ports will keep on working. So, once you execute this instruction. Let me remove the comments here. Once you execute the comments here, you can immediately go in the low power mode 3. And you are enabling the interrupts and before that you have already enable the timer. The timer clock is selected from A clock that means once one second gets over, you are going to go in the interrupt. And this is the interrupt program. We are saying that our vector is timer is 0_A0 vector and this is the code for it. And what is the code? Simply, have a count where you are saying the count is equal to increment the count and what is count.

(Refer Slide Time: 24:41)



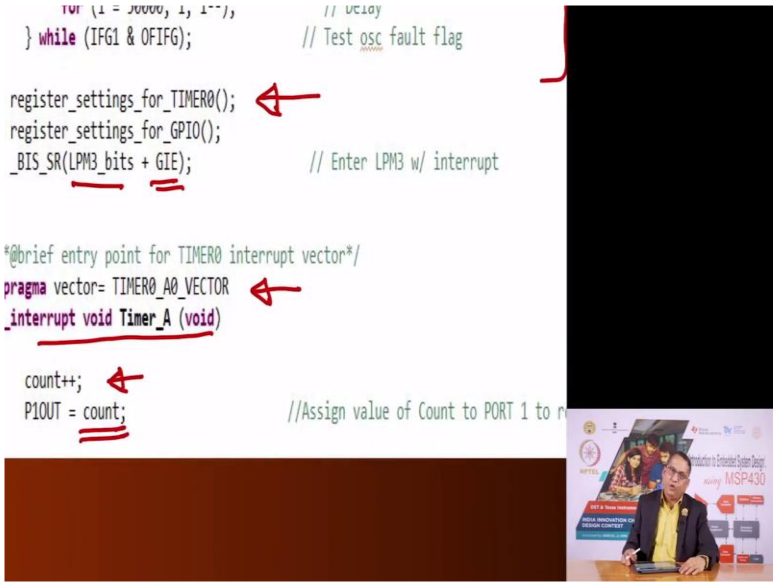
Hello_Timer

```c
#include <msp430.h>
#include <inttypes.h>

volatile char count = 0;
volatile unsigned int i;

/**
 * @brief
 * These settings are wrt enabling GPIO on Lunchbox
 **/
void register_settings_for_GPIO()
{
    P1DIR |= 0xFF;                        //P1.0 to P1.7 are set as Output
    P1OUT |= 0x00;                        //Initially they are set to logic zero
}

/**
 * @brief
 * These settings are w.r.t enabling TIMER0 on Lunch Box
 **/
void register_settings_for_TIMER0()
{
    CCTL0 = CCIE;                        // CCR0 interrupt enabled
    TACTL = TASSEL_1 + MC_1;             // ACLK = 32768 Hz, upmode
    CCR0 = 32768;                        // 1 Hz
}
```



Hello_Timer

```c
nclude <msp430.h>
nclude <inttypes.h>

latile char count = 0;
latile unsigned int i;

*
 @brief
 These settings are wrt enabling GPIO on Lunchbox
*/
id register_settings_for_GPIO()

 P1DIR |= 0xFF;                   //P1.0 to P1.7 are set as Output
 P1OUT |= 0x00;                   //Initially they are set to logic zer

*
 @brief
 These settings are w.r.t enabling TIMER0 on Lunch Box
*/
id register_settings_for_TIMER0()

 CCTL0 = CCIE;                    // CCR0 interrupt enabled
```



```
    for (i = 50000; i; i--);            // Delay
  } while (IFG1 & OFIFG);              // Test osc fault flag

 register_settings_for_TIMER0();
 register_settings_for_GPIO();
 _BIS_SR(LPM3_bits + GIE);              // Enter LPM3 w/ interrupt


*@brief entry point for TIMER0 interrupt vector*/
pragma vector= TIMER0_A0_VECTOR
_interrupt void Timer_A (void)

  count++;
  P1OUT = count;                      //Assign value of Count to PORT 1 to r
```

If you see here the count is see very important. Count is declared as volatile variable of the type CAR which means it is a 8 bit counter and the initial value 0. Now, volatile means whenever you increment it, it will be fetched again and again from the actual location instead of copy being read from any register, internal register. It will be read from memory because this is going to be initialized in memory.

So, we have declared this variable as volatile. And in that variable, we are incrementing and that value incremented value simply outputting on the port 1 bits. So, once you compile, rebuild and download this program. What you should see that every one second, the 8 bits LEDs will start working like a binary counter and it will keep on counting from 0, it will reach all the LEDs are, at 0 all the LEDs will be off.

Then as it gets every one second it will increment and at 255 seconds, you will see all the LEDs on. And then in next second all the LEDs will be off again. And this will keep on happening. And this is basically to show that the delay period of incrementing the LEDs, incrementing the count display on the LEDs is not being done with the help of a CPU program, with the help of the code but it is being performed by the peripheral which is the timer A_0.

And we will see more activities involving the timer such as the capture compare modes. And also how to use the timer in the PWM generation facility. We will see in our subsequent lecture. I suggest you play with this, if you want to change the frequency at the rate the clock the count is being updated, you can go into this part of the code.

And change this number to a lower value and you will see that it blinks the LEDs, it counts the LEDs faster. If you increase this you can have a slower update. So, I suggest that you play with this code, update it, modify it, download it and see how various changes to the code will impact the actual performance. Thank you very much, I will see you very soon, bye bye..