

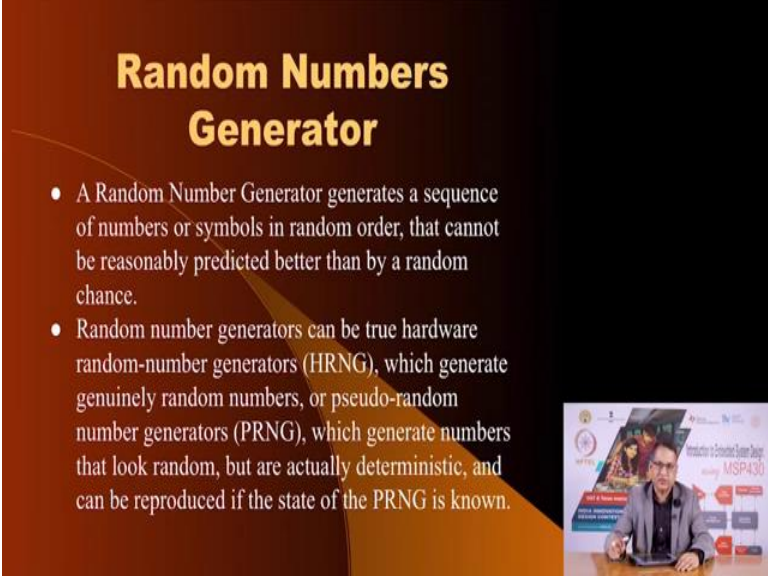
**Introduction to Embedded System Design**  
**Professor Dhananjay V. Gadre**  
**Electronics and Communication Engineering**  
**Netaji Subhas University of Technology**  
**Badri Subudhi**  
**Electrical Engineering Department**  
**Indian Institute of Technology, Jammu**  
**Lecture 32**

**ADC and DAC using R2R Ladder and Random Number Generation using LFSR1**

Hello and welcome to new session for this online course on Introduction to Embedded System Design I am your instructor Dhananjay Gadre. In this session, we are going to consider remaining topics related to analog and digital converter and the ability of the microcontroller to read and generate analog voltages.

We are also going to consider how the microcontroller can generate random numbers and a little bit part about one of the experiments that you previously saw that we did which was to power MSP-430 microcontroller using a lemon battery invoking the low power modes I am going to show you little more details about it.

(Refer Slide Time: 01:18)



**Random Numbers Generator**

- A Random Number Generator generates a sequence of numbers or symbols in random order, that cannot be reasonably predicted better than by a random chance.
- Random number generators can be true hardware random-number generators (HRNG), which generate genuinely random numbers, or pseudo-random number generators (PRNG), which generate numbers that look random, but are actually deterministic, and can be reproduced if the state of the PRNG is known.

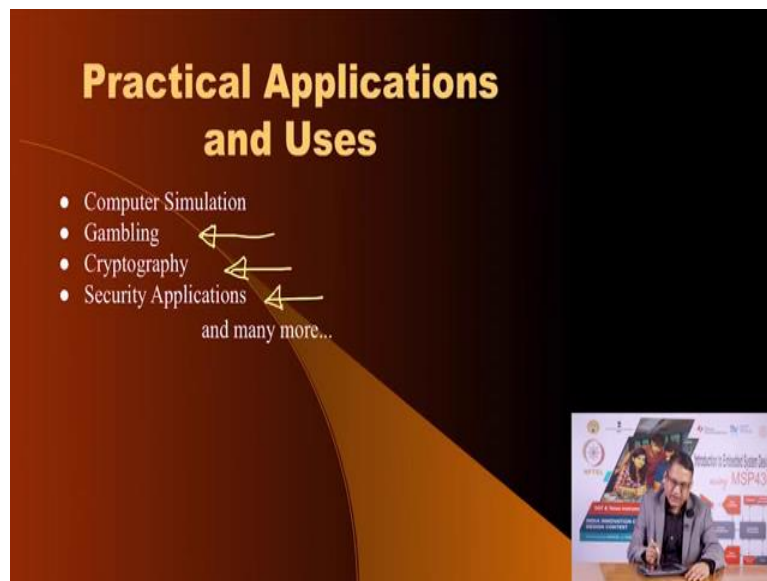
The slide features a dark background with a light-colored abstract shape. In the bottom right corner, there is a small inset image showing the instructor, Professor Dhananjay V. Gadre, sitting at a desk with a laptop and a presentation board. The board displays the title 'Introduction to Embedded System Design' and 'Random Number Generation using LFSR1'.

So in this lecture we are going to talk about first of all the random numbers generators. Now random numbers are very important for embedded applications. A random number generator produces a random number which cannot be reasonably predicted part from being by random

chance. These can be true hardware based random number generators although there are no true random number generators all mechanisms are some sort of pseudo-random number generators.

You can do that using hardware approach or you can utilize some software approach. We are going to discuss both these methods here.

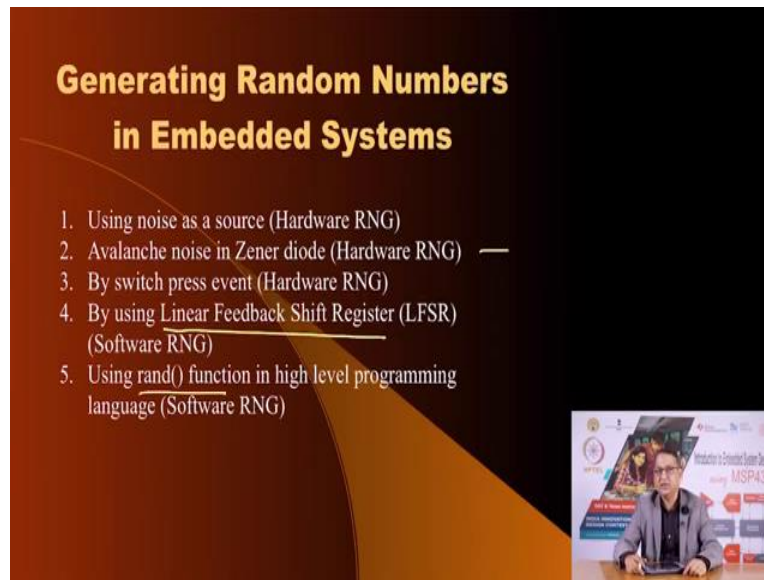
(Refer Slide Time: 01:55)



Now what are the applications of such random number generators? You would use it in computer stimulations, you can also use it for gambling, you may have seen that one of the projects that we demonstrated in the second lecture was dice. And for that we use some sort of random number generator. You could use it in cryptography and many security applications and there are many such applications.

Now what are the various methods by which you can generate such random numbers? One is using a noise because noise is random, so if you sample that noise you would get a value which is random and therefore this could be a source of your random number.

(Refer Slide Time: 2:51)




You could use for that noise source you could use a high value resistor, you could sample external sound and so on, another source of such hardware noise is through a Zener diode, you could use a switch to stop high frequency counter, the value in that could be random if the switch press is a synchronous event that is a human has pressed it you can use a interesting topology called a linear feedback shift register.

We are going to dedicate few slides on describing what a linear feedback shift register is, we are going to show couple of experiments how we can use LFSR to generate pseudo-random numbers and at the end you can also use built-in random function that many embedded C compilers support.

(Refer Slide Time: 03:33)

## Linear Feedback Shift Register

- A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state.
- Most commonly used linear function of single bit is Exclusive-or (XOR).
- The initial value is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state.
- So, if one seed is used for two LFSRs, they produce the same sequence of numbers




Now a linear feedback shift register is a shift register basically it is a whole few D-type flip-flops which the output of one feeds the input of the second and so on. And then the last one is fed back into the input. The of course, you just do not connect it like that, you operate on the outputs in some way and the linear function could be through a EX-OR or EX-NOR gate. Now because these are flip-flops you need to initialize them with some value when you turn the power on and that value is called the seed value.

(Refer Slide Time: 04:13)

## Linear Feedback Shift Register

- Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle.
- However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.
- The bits in the LFSR state that influence the input are called taps.

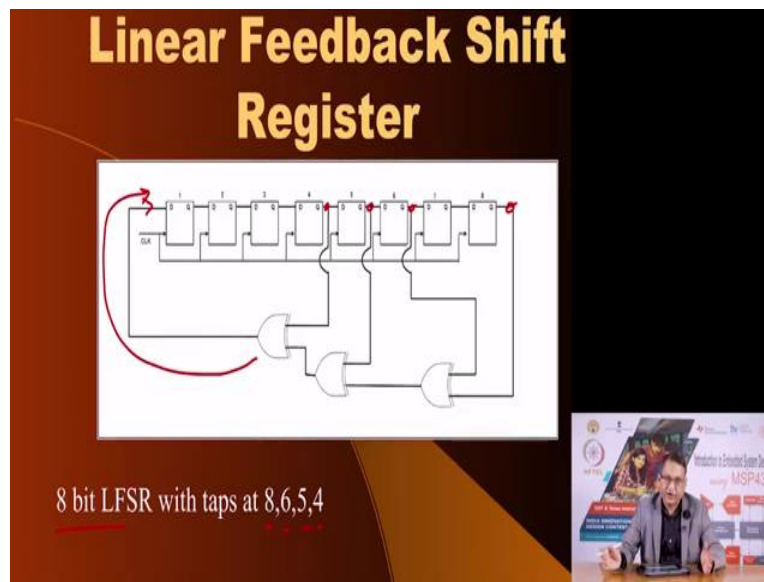
$n$ -bit  $\underline{\underline{2^n - 1}}$



The linear feedback shift registers work on the principle that if you have  $n$ -bits of flip-flop then if you chose these feedback functions appropriately you could have in principle  $2^n - 1$  combinations, so this is just like a counter when you have  $n$  bits of D-type flip-flops you can generate  $2^n$  combinations except in a counter these are always you know either it is increasing value or decreasing value, up counter or down counter.

So you have  $2^n$ , in the case of LFSR what is called as a maximal length LFSR you will not get  $2^n$  combinations but one less and that is why we say it is  $2^n - 1$ . We will see why it is so.

(Refer Slide Time: 05:08)



Now basically LFSR consists of D type flip-flops as you see here, here is the input of the first one that feeds the second one and so on. Now, from some of these flip-flops you take the output and pass it through certain EX-OR gates and that feeds the input. Now, when you load it with a initial number and then you apply the clock for every clock signal the values that the flip-flops will produce they will appear random and that is why this is called a pseudo-random number generator.

Now, the combination in this case we have shown a 8-bit LFSR now wherever you take the value out here, here, here and here these are called taps. So we have taps at 8, 6, 5 and 4 for a 8-bit shift register. Now how do we find out where these taps should be applied when you have a different length of LFSR?

(Refer Slide Time: 06:04)

The slide features a table with columns for 'n', 'XNOR from', 'n', 'XNOR from', 'n', 'XNOR from', and 'n', 'XNOR from'. The rows correspond to n values from 3 to 35. Handwritten red annotations on the left side of the table include '2^30' next to n=30, '161' next to n=160, and '2^30 - 1' next to n=160. A video inset in the bottom right corner shows a man speaking at a desk with a presentation slide titled 'Introduction to Embedded System Design using MSP430'.

n	XNOR from	n	XNOR from	n	XNOR from	n	XNOR from
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	49	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,138,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,03	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122
18	18,11	60	60,59	102	102,101,36,35	144	144,143,75,74
19	19,6,2,1	61	61,60,46,45	103	103,94	145	145,93
20	20,17	62	62,61,6,5	104	104,103,94,93	146	146,145,87,86
21	21,19	63	63,62	105	105,99	147	147,146,110,109
22	22,21	64	64,63,61,60	106	106,91	148	148,121
23	23,18	65	65,47	107	107,105,44,42	149	149,148,40,39
24	24,23,22,17	66	66,65,57,56	108	108,77	150	150,97
25	25,22	67	67,66,58,57	109	109,108,103,102	151	151,148
26	26,6,2,1	68	68,59	110	110,109,98,97	152	152,151,87,86
27	27,5,2,1	69	69,67,42,40	111	111,101	153	153,152
28	28,25	70	70,69,55,54	112	112,110,69,67	154	154,152,27,26
29	29,27	71	71,65	113	113,104	155	155,154,124,123
30	30,8,1	72	72,66,25,19	114	114,113,33,32	156	156,155,47,46
31	31,29	73	73,48	115	115,114,101,100	157	157,156,131,130
32	32,22,2,1	74	74,73,59,58	116	116,115,46,45	158	158,157,132,131
33	33,20	75	75,74,65,64	117	117,115,99,97	159	159,128
34	34,27,2,1	76	76,75,41,40	118	118,85	160	160,159,142,141
35	35,33			119	119,83	161	161,143

Well, you do not have to sweat too much if you search the internet you will find many sources, for example we have taken this source from a Xilinx website and they have linear feedback shift register which instead of using EX-OR gate it uses EX-NOR and it has a value of n that is the number of flip-flops from 3 to 160 bits and you can imagine the total number of random sequences that will produce is 2 raise to power 161 minus 1 and that is a very-very large number.

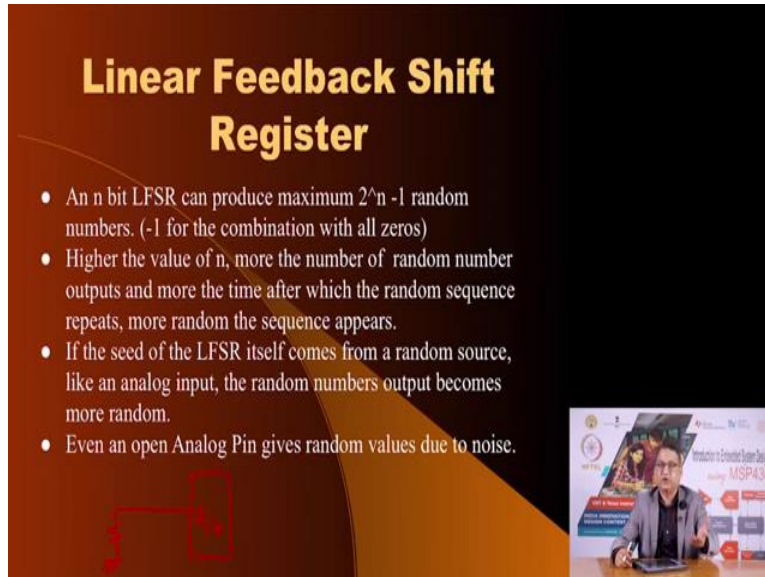
Remember that 2 raise to power 20 is appropriately a million, 2 raise to power 30 is a billion and so you can imagine what 2 raise to power 60 would be, a very-very large number. So the sequence is that such a shift register will produce compare to the last value you cannot predict the next one and that is why this comes in a category of software based random number generators.

Now why do we have minus 1 here? Why do we always have 1 less is if the shift register had a initial value of 0, then no matter what operation that you perform on it, it will always remain 0. So we have to exclude that from the number of combinations that are available and that is why it is always 2 raise to power n minus 1.

So this is what you get, now the question is how do you get the seed value? if you write a program in which you define the shift register and the taps you are problem will be how do you initialize the shift register? If you always initialize it with a known value because in the program you say seed is equal to so and so forth it will always you will always predict the, can always

predict the sequence. So getting first seed value is a tricky proposition but there are many options.

(Refer Slide Time: 07:57)



## Linear Feedback Shift Register

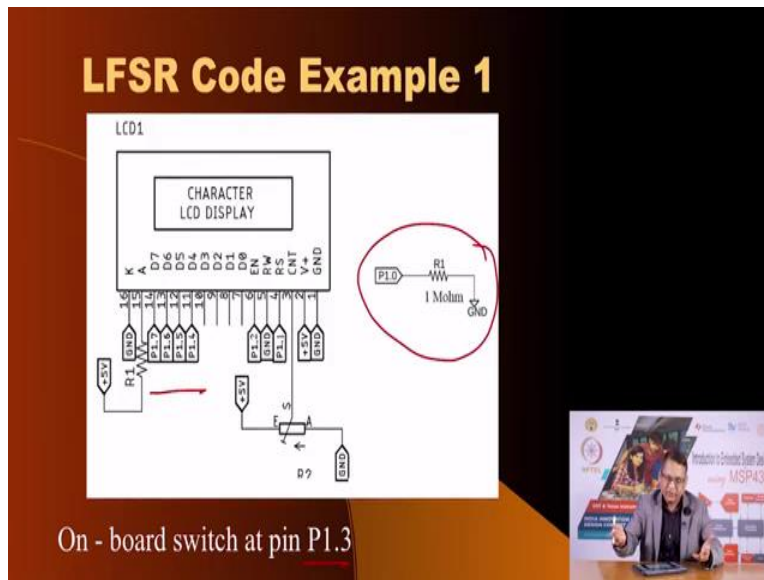
- An n bit LFSR can produce maximum  $2^n - 1$  random numbers. (-1 for the combination with all zeros)
- Higher the value of n, more the number of random number outputs and more the time after which the random sequence repeats, more random the sequence appears.
- If the seed of the LFSR itself comes from a random source, like an analog input, the random numbers output becomes more random.
- Even an open Analog Pin gives random values due to noise.

The slide features a dark background with a light-colored diagonal line. A small inset video in the bottom right corner shows a person speaking at a desk with a laptop, with a presentation slide visible behind them that includes the text 'Production of Pseudo-Random Numbers using MSP430'.

If you have a microcontroller such as MSP-430 and it has analog input if you put a large enough resistor or even if you keep the analog input open, and if you sample this signal you will find that it reads the random number or it reads the value which you cannot predict and this could act as a seed. So what we have done is we have used this mechanism to illustrate some a couple of code example.



(Refer Slide Time: 08:25)




In the first code example we are going to have connector switch on our MSP-430 lunch box on p 1.3 we have also connected displays as we have seen in our previous examples in the 4 bit mode and we have connected a very large value resistors 1 mega ohm resistor to ground. What we do, that when you run this program every time when you initialize it, when the program starts running for the first time it will sample the analog input.

We use this value as a seed initialize the LFSR with the seed and then every time you press the switch it will generate the next number and you can find out what does numbers are.

(Refer Slide Time: 09:06)


```
1 #include <msp430.h>
2 #include <inttypes.h>
3
4 #define SW BIT3 // Switch -> P1.3 (On-board Switch, Pull-up configuration)
5
6 #define CPO 0
7 #define DATA 1
8
9 #define AN IN BIT0 // Channel A0
10
11 #define LCD_OUT P1OUT
12 #define LCD_DIR P1DIR
13 #define DA BIT4
14 #define D5 BIT5
15 #define D6 BIT6
16 #define D7 BIT7
17 #define R5 BIT1
18 #define EN BIT2
19
20 /**
21  *Brief Delay function for producing delay in 0.1 ms increments
22  *Param t milliseconds to be delayed
23  *return void
24  */
25 void delay(uint16_t t)
26 {
27     uint16_t i;
28     for(i=t; i > 0; i--)
29         __delay_cycles(100);
30 }
31
32 /**
33  *Brief function to pulse EN pin after data is written
34  *return void
35  */
36 void pulseEN(void)
37 {
38     LCD_OUT |= EN;
39     delay(1);
40     LCD_OUT &= ~EN;
41     delay(1);
42 }
```

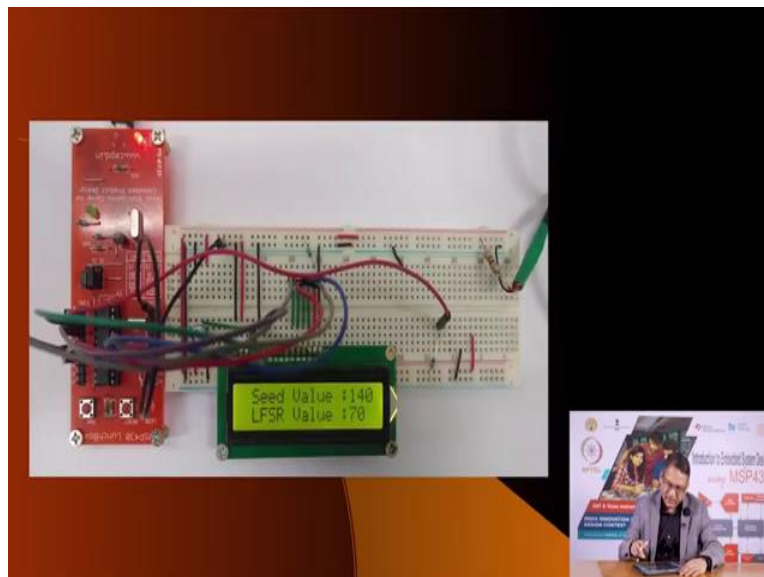
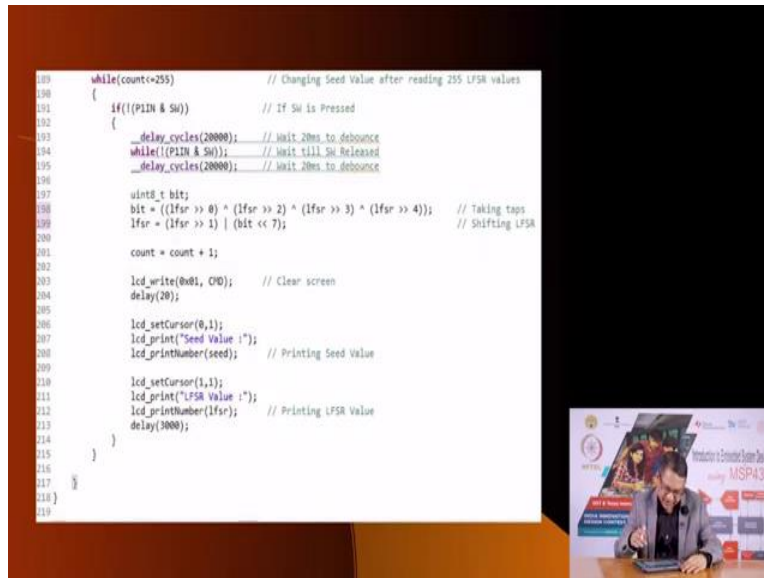


This is the code example I am not going to go through it, it is by this time you would be able to very easily understand how it works.

(Refer Slide Time: 09:18)

```
17 /**
18  *Brief
19  * These settings are for enabling ADC10 on Launchpad
20  */
21 void register_settings_for_ADC10()
22 {
23     ADC10A0R |= ADSC; // P1.8 ADC option select
24     ADC10CTL1 = INCH_0; // ADC Channel -> 1 (P1.8)
25     ADC10CTL0 = SREF_0 + ADC10REF_3 + ADC10RNG; // Ref -> Vcc, 0x 0x 500, ADC - On
26 }
27
28 /*Brief entry point for the code*/
29 void main(void)
30 {
31     WDTCTL = WDTPW + WDTHOLD; // Stop watchdog (not recommended for code in production and devices working in field)
32     P1DIR &= ~SW; // Set SW pin -> Input
33
34     lcd_init(); // Initializing LCD
35     register_settings_for_ADC10();
36
37     while(1)
38     {
39         ADC10CTL0 |= ENC + ADC10SC; // Sampling and conversion start
40
41         while(ADC10CTL1 & ADC10BUSY); // Wait for conversion to end
42
43         uint_t seed = (ADC10MEM & 0xFF); // Lower 8-bits of ADC conversion result as seed
44         uint_t i; for (i = seed; i <= 0; i++); // i = seed; i <= 0; i++
45         uint_t count = 0;
46
47         lcd_write(0x01, 0x01); // Clear screen
48         delay(20);
49
50         lcd_setCursor(0,1);
51         lcd_print("Seed value: ");
52         lcd_printNumber(seed); // Printing Seed value
53
54         lcd_setCursor(1,2);
55         lcd_print("Press Switch");
56     }
57 }
```






Here is the seed part initialization and now you are trying to count numbers and print them. Now what you will see is when you press the switch first time it will when you turn it on you will get a seed value and now when you press the switch it will feed the seed value into the LFSR which is a 8-bit LFSR and will produce the next number.

So every time you press a switch, you will get a new number and try to play with it to see whether you can predict the next number.

(Refer Slide Time: 09:52)

## LFSR Code Example 2

```
#include <msp430.h>
#include <ctype.h>
#include <stdio.h>
4
#define RED BIT7 // Switch -> PL3 (On-board Switch, Pull-up configuration)
#define AN BIT0 // Channel AN
5
/**
6 *Brief Delay function for producing delay in 0.1 ms increments
7 *Param t milliseconds to be delayed
8 *Returns void
9 **/
10 void delay(uint16_t t)
11 {
12     uint16_t i;
13     for(i=t; i > 0; i--)
14         __delay_cycles(100);
15 }
16
17 /**
18 * @brief
19 * These settings are set enabling ADC10 on lunchbox
20 **/
21 void register_settings_for_ADC10()
22 {
23     ADC10AER |= AN; // PL3 ADC option select
24     ADC10CTL1 = INCH_0; // ADC Channel -> 1 (PL3)
25     ADC10CTL0 = SREF_0 + ADC10SHT_3 + ADC10ON; // Ref -> YES, 64 CLK SHt, ADC - ON
26 }
```



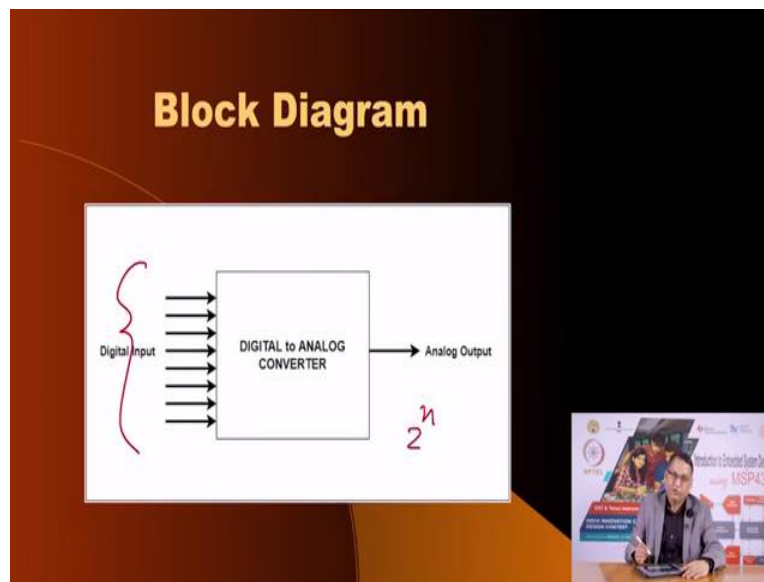
We have new modification, where we modify the shift register LFSR to 32-bit size and instead of using a switch we just constantly update it the value that is shift the numbers and the least significant bit we send it to the onboard LED of the lunch box.

What would you expect? You would expect that the LED will be on and off, on and off but whether it is on or off for how much time it is on or off will be pretty random, so what you would notice is that the LED appears to flicker. So this would be a good way to create a candle like performance on a LED it would appear like a flickering candle.

So if you compile and download this second code example you and you run it, you would see that the LED which is connected to port 1.7 is flickering at a random rate and it would appear like a candle. The second part of this lecture deals with the mechanism or the requirement of converting digital numbers into analog voltages.

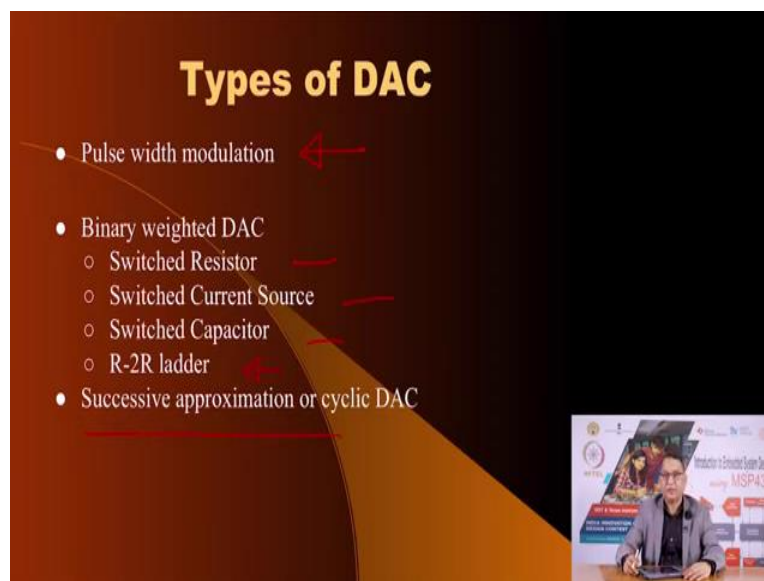
We have already seen this in our previous lecture using the pulse width modulation technique. But the problem with pulse width modulation technique is that it is not able to produce analog voltages at a very fast rate, why? Because of the limitations of the frequency of the PWM signal itself. And so we are going to look at various other options of generating analog voltage using a D to A converter.

(Refer Slide Time: 11:27)



And there are many mechanisms to do that the block diagram is that you have certain number of pins here n-bits so you would be able to generate  $2^n$  voltages in the range of 0 to some maximum voltage.

(Refer Slide Time: 11:39)




There are various methods, the first method we have already seen using a PWM then you can have DAC made out of resistors and current sources and switch capacitor and resistor using R-2R ladder as well as successive approximation type of DAC.

(Refer Slide Time: 11:57)

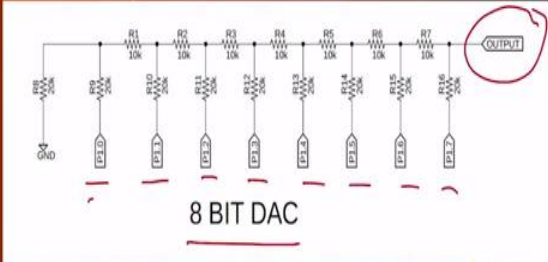

## R-2R Ladder DAC

- An R-2R ladder is a simple and inexpensive way to perform digital-to-analog conversion, using repetitive arrangements of precise resistor networks in a ladder-like configuration.

$R = 10k\Omega$   
 $2R = 20k\Omega$



## 8 bit R-2R Ladder DAC



$$V_{out} = V_{b0} / 256 + V_{b1} / 128 + V_{b2} / 64 + V_{b3} / 32 + V_{b4} / 16 + V_{b5} / 8 + V_{b6} / 4 + V_{b7} / 2$$


In this example here we are going to create a R-2R ladder network it is called R-2R because it only uses two values of resistor, R and a 2R. For example, you could chose R to be 10 kilo ohm and 20 kilo ohm. So you could choose these 2 value of resistors and connect them in this fashion, this is the R-2R ladder network and it is a 8-bit DAC because we have 8 output pins form the microcontroller feeding this DAC and here is the output and this is the equation that governs the bit values and its impact on the actual output.

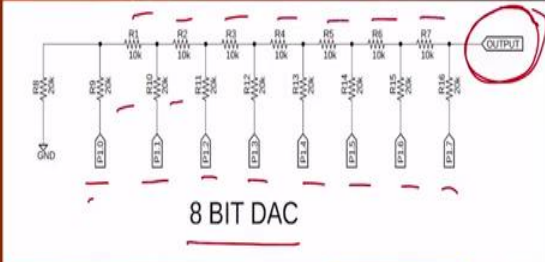
(Refer Slide Time: 12:36)

## Example Code : Hello DAC


```
#include <msp430.h>
#include <types.h>
#define count 8
unsigned int i;
//**
// * @brief
// * These settings are w.r.t enabling GPIO on launch
// **
void register_settings_for_GPIO()
{
    PDIR0 |= BIT0; //P1.0 to P1.7 are set as Output
    P1OUT &= ~(BIT0); //Initially they are set to logic zero
}
//**
// * @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 field)
    register_settings_for_GPIO();
    while(1)
    {
        P1OUT = count; // Assign value of count to PORT 1 to represent it as binary number.
        for(i = 0; i < 5000; i++); // Increment count
    }
}
```



## 8 bit R-2R Ladder DAC

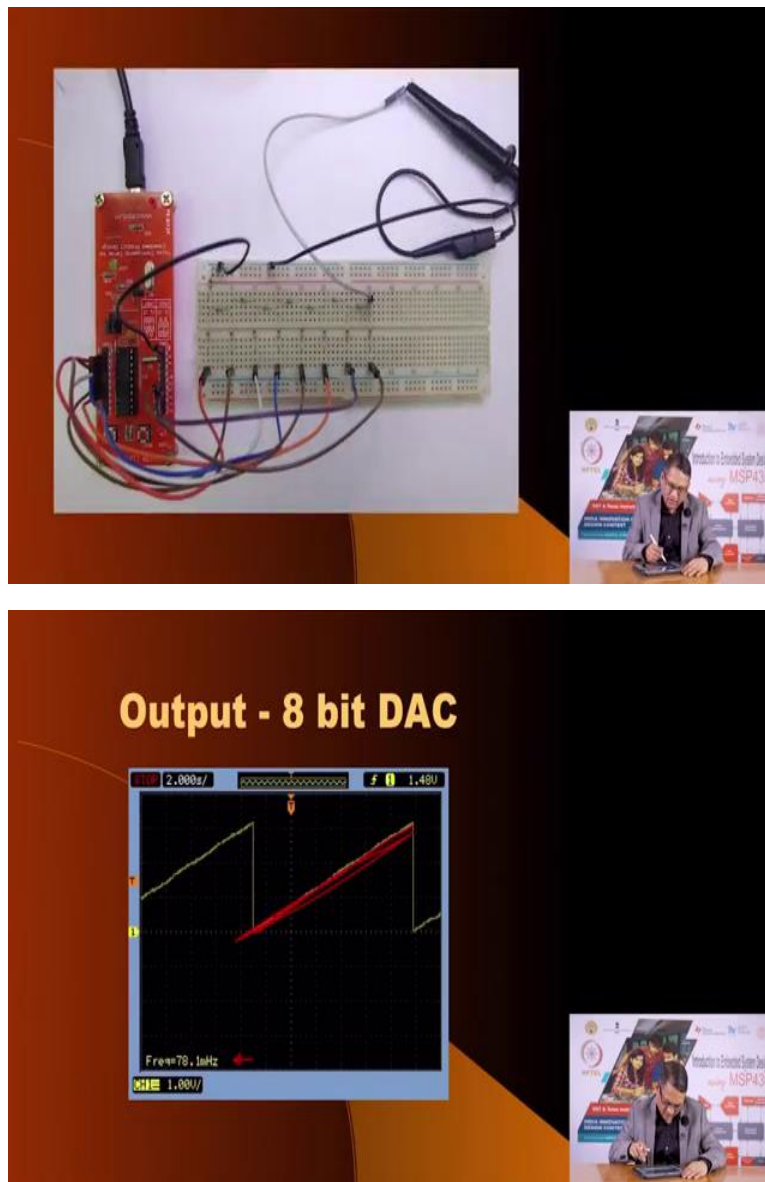


8 BIT DAC

$$V_{out} = V_{b0} / 256 + V_{b1} / 128 + V_{b2} / 64 + V_{b3} / 32 + V_{b4} / 16 + V_{b5} / 8 + V_{b6} / 4 + V_{b7} / 2$$


When you run this code it is going to generate a saw tooth waveform on your output pin which is the output pin here after ( ) (12:48). Now, it is very important that the value of these resistors are all equal, 10 kilo ohms and 20 kilo ohms. But how do you ensure that you get accurate 10 kilo ohm resistor? What you can do is you can have a bunch of resistors and then you can using a multi-meter find out few resistors 10 kilo ohms and few 20 kilo ohms which match each others values as much as possible and then use that to create such a DAC.

(Refer Slide Time: 13:20)



When you run this program here I have shown that you connect port 1 pins to R-2R ladder network and when you connect it to the oscilloscope here is what you would get as a 8-bit DAC and this frequency is very low but that is because the number of bits are large 8-bits and the frequency is not very high. And also you will feel that this saw tooth is not really a straight line and that is because the resistors are not really matched values.


So if you have a better values of resistors which match each other well then you would find that the DAC output in this case is much linear.



(Refer Slide Time: 14:07)

### Example Code : Hello DAC

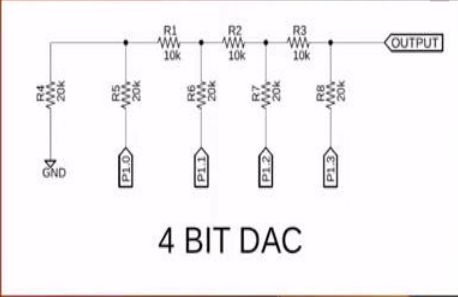
```
#include <msp430.h>
#include <inttypes.h>
int
char count = 0;
unsigned int i;
//**
// Brief
// * These settings are w.r.t enabling GPIO on Launchpad
// **
void register_settings_for_GPIO()
{
    P1DIR |= BIT0; //P1.0 to P1.7 are set as Output
    P1OUT &= ~(BIT0); //Initially they are set to logic zero.
}
//**
// 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 field)
    register_settings_for_GPIO();
    while(1)
    {
        P1OUT = count; // Assign value of count to PORT 1 to represent it as binary number
        for(i = 0; i < 5000; i++); // Increment count
        count++;
    }
}
```




And the reason why the frequency appears to be very low is also because certain amount of delay has been introduced in this code. If you remove this delay you would find that the saw tooth waveform is of a much higher frequency.

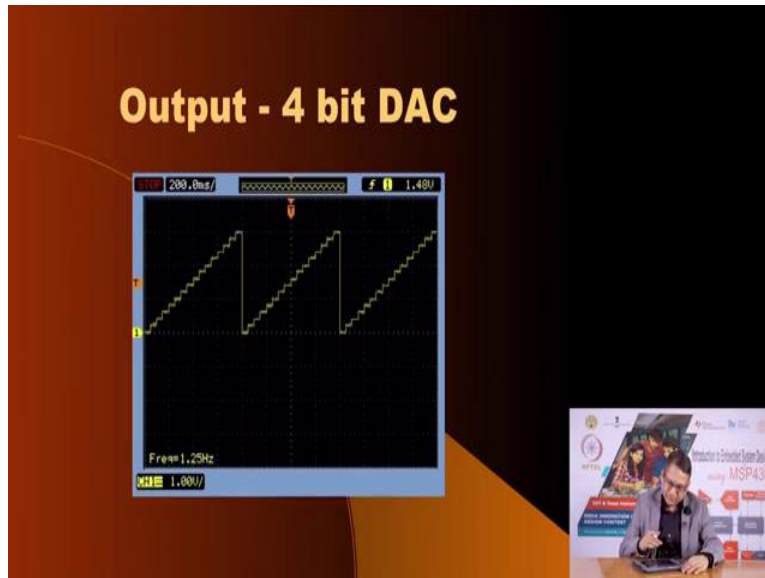
(Refer Slide Time: 14:25)

### 4 bit R-2R Ladder DAC



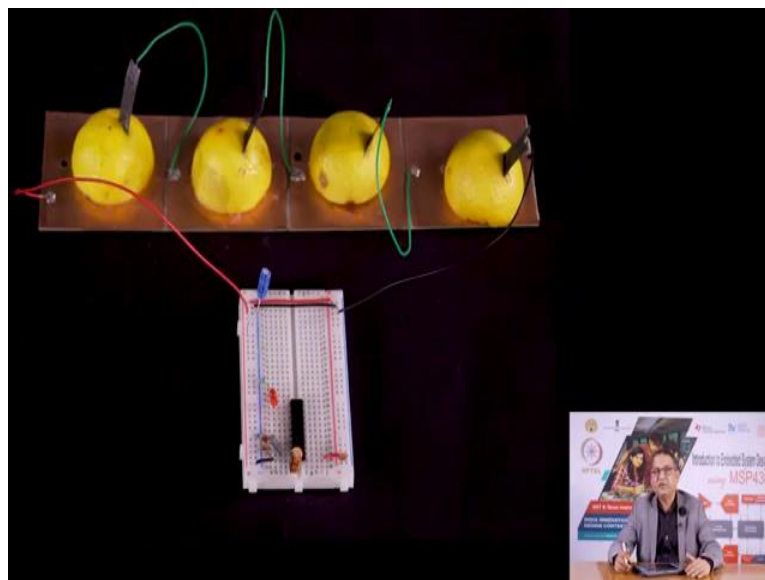
4 BIT DAC

$$V_{out} = V_{b0} / 16 + V_{b1} / 8 + V_{b2} / 4 + V_{b3} / 2$$


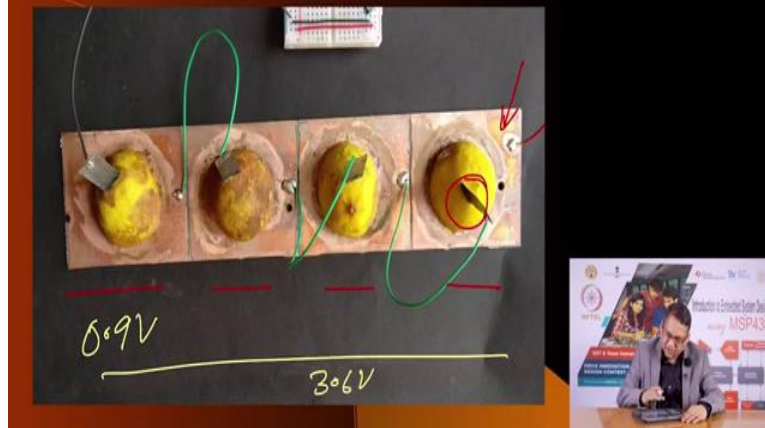


Now instead of 8-bit DAC you can have a 4-bit DAC and now you see the frequency has improved also because the number of steps that you have instead of 256 steps you have only 16 steps.

(Refer Slide Time: 14:40)



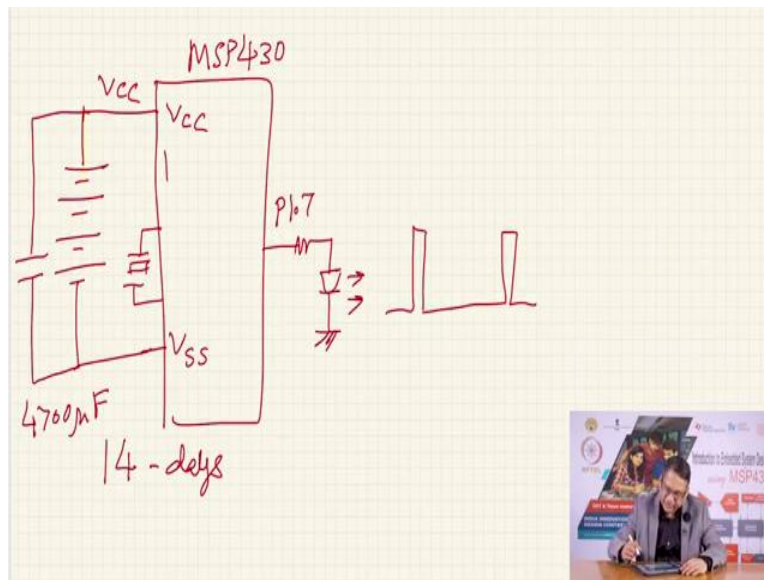
## Lemon Battery Experiment



The third part of this lecture is about revisiting the lemon battery that we created in the past as you remember we I showed you this demonstration that we created a battery out of lemon and two dissimilar metals as electrodes.

One was the copper here, this is copper and what is inserted in the lemon is a different metal, actually extracted out of alkaline battery and using this combination each of these, so this are 4 cells each of these cells produces 0.9 volts, so roughly from this battery you get about 3.6 volts.

(Refer Slide Time: 15:30)

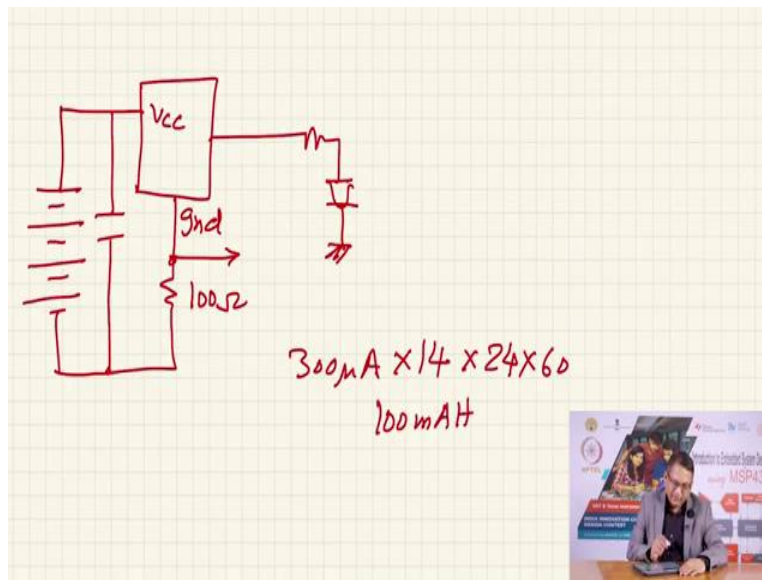


And using setup which was like this that this is the lemon battery 1, 2, 3 and 4 we connected it to a microcontroller MSP-430 remember we programmed it in the lunch box, then we took the IC out put it in the breadboard and connected the crystal to the crystal pins so that is working at 32 kilo Hertz and then on the P1.7 pin we connected the LED like this and the microcontroller this is the VCC and this is the VCC pin, this is the VSS that is the ground pin and ofcourse the crystal was connected to appropriate pins of the microcontroller.

And you saw, notice that this pin, this LED would blink every second in a flash and so I am going to go through the code and also talk about the current that this entire setup consumed. We monitored this battery over a period of 14 days that is this battery continued to operate work non-stop for 14 days. We measured the current that was being consumed by the setup I will tell you how and then we estimated what was the capacity of this battery.

Now when the battery volt this battery typically has a very large output impedance which means if you try to draw large current the voltage will drop. And so to make sure that the microcontroller is not effected when the LED is turned on it would take more current, so we connected large capacitor for measurement about 4700 micro farad. So that it would filter out whenever the LED is being turned on.

(Refer Slide Time: 17:46)



And we noticed the setup let me show you the setup what we did was, we took the battery here and connected it to the microcontroller VCC and had a small resistor connected here this is the ground pin of the MSP-430. And we connected a very large value of capacitor across this and then we use this, this was about 100 ohm resistor and we observed the voltage on this.

And from this we estimated and ofcourse you had the LED connected here and from here we estimated the amount of current being consumed by this. The average current when the LED was on it would consume about 500 micro amperes of current when the LED was not on it would be less but the average current was roughly 300 micro amperes.

So 300 micro amperes into 14 days into 24 hours into 60 minutes will give you roughly 100 mah battery capacity. So this is the kind of performance you can expect out of a simple battery that you can create on your own.

(Refer Slide Time: 19:18)

```
#include <msp430.h>
/**
 * @brief
 * * These settings are set enabling GP10 on launch
 * */
void register_settings_for_GP10()
{
    P1DIR |= BIT7; // P1.7 as Output
    P1OUT &= ~BIT7; // Initialize with zero
}
/**
 * * @brief
 * * These settings are w.r.t enabling TDR0 on launch
 * */
void register_settings_for_TDR0()
{
    CCT0 = CCT0; // CCR0 interrupt enabled
    TACTL = TASSL1_1 + MC_1; // MCLK = 32768 Hz, Up mode
    CCR0 = 32768; // 1 Hz
}
/** @brief entry point for the code */
void main(void)
{
    WDCTL = WDTPW + WDTHOLD; // Stop Watch dog (Not recommended for code in production and devices working in field)
    unsigned int i;
    do{
        IFG1 &= ~OIFG; // Clear oscillator fault flag
        for (i = 50000; i; i--); // Delay
    } while (IFG1 & OIFG); // Test osc fault flag
    register_settings_for_TDR0(); // Enter LPM0 w/ interrupt
    register_settings_for_GP10();
    BIS_SR(LPM0 bits + GIE); // Enter LPM0 w/ interrupt
}
/** @brief entry point for TDR0 interrupt vector */
#pragma vector= TDR0_A0_VECTOR
__interrupt void Timer_A (void)
{
    P1OUT |= BIT7; // LED HIGH
    __delay_cycles(200); // LED ON
    P1OUT &= ~BIT7; // LED LOW
}
```

This is the code, you can compile and download this code. Here is the main program the watchdog timer has been turned off. The code was using the crystal oscillator and so it has to be ensured that the oscillator is working.

So the oscillator bit was monitored if the oscillator flag shows a fault you are going to wait for it and after that you turned the timer on to give you 1 second interrupt, you turned the pins so that you are making the port P1.7 as output and then you turned the microcontroller to operate in the low power mode 3 with the interrupt enabled turned on and so every time the timer expired it would take you to the interrupt vector which is here.

In that the bit was made high for a little while and then made low and then you would go back into low power mode and that was the reason why the microcontroller consumed very little power for its operation, the only time it consume more power was when the LED was on. So I recommend that you build this code and if you want you can create such a battery at you own place and see how MSP-430 microcontroller operates in such a low power situation by turning everything off expect the time when the interrupt is generated and the LED is turned on.

So, this is to highlight the low power operation of MSP-430 and with this we cover how MSP-430 can be used to generate random numbers, how MSP-430 can be used to generate analog voltages and also how MSP-430 has this rich low power modes of operation and you could

optimize your system so that it conserves as much power as possible. I am going to stop this lecture here and I will see you soon with the new topic, thank you.