Introduction to Embedded System Design

Professor Dhananjay V. Gadre
Netaji Subhas University of Technology, New Delhi

Lecture - 25

Interrupts in MSP430

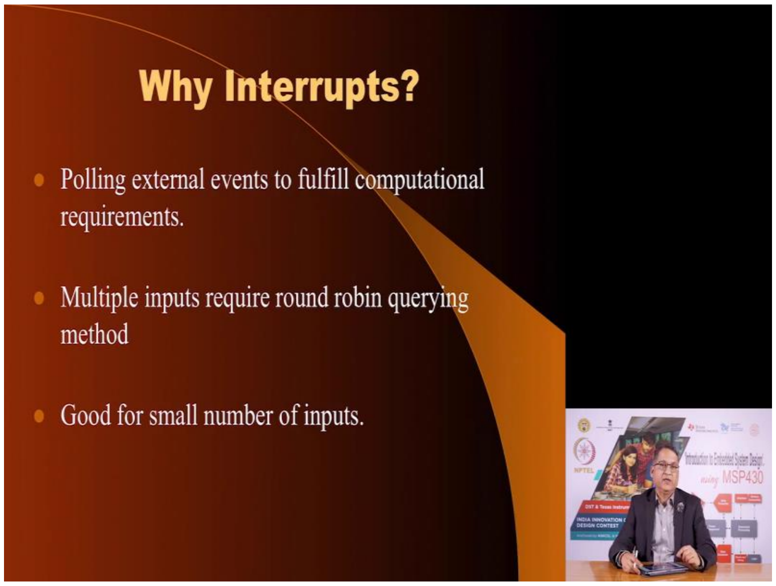
Hello and welcome to a new session for this online course on Introduction to Embedded System Design. As usual, I am your instructor Dhananjay Gadre. In this lecture, we are going to talk about Interrupts. Interrupts are very important paradigm, is an important paradigm, is a mechanism which gives a feeling of multitasking or being able to do multiple things at the same time. Of course, this is not true, because at the end of the day your microcontroller, whether it is MSP430 or any other small microcontroller has only one CPU and a single CPU can only perform one operation at a time.

But because it is able to perform those operations at a very fast rate compared to the human interaction levels or speeds of human interaction, it is possible to give a perception that the microcontroller or your microcontroller in the embedded system is actually doing multiple things at the same time. And so we are going to introduce you to this concept of interrupts.

Now, why do we need interrupts? It is an important idea because a microcontroller or an embedded system would like to perform several functions at the same time. For example, let us say that in a system you have many, many switches or you have few switches, few selector knobs, and an external user such as a human being may press either the switch or a knob or some other switch and you would want the microcontroller to respond to this external event.

Now if a microcontroller is trying to read one switch, then it will have to wait for that switch to be pressed, if it is waiting for that switch to be pressed, what if the user presses another switch? How does one, how does one deal with such a situation? So (micro) the way to resolve this is to bring in this concept of interrupts.

(Refer Slide Time: 02:29)



Why Interrupts?

- Polling external events to fulfill computational requirements.
- Multiple inputs require round robin querying method
- Good for small number of inputs.

So, interrupts allows you to perform multiple tasks instead of polling. Now what is polling? For example, I give this idea that there are several switches, maybe there are several knobs, and one way to deal with an external user pressing any of the switches or turning any of the knobs would be to go to a switch, wait for it to be pressed for some time, then go to another switch wait for it to be pressed, if it is not pressed go to the third switch so on and so forth, and it can keep on polling. Polling means waiting for a action, waiting for an event to happen and if there is no event it can go and look at another event, possibility of another event.

Now, this polling is fine but when the number of such external inputs increases or let me continue with this example, let us say there are several switches and the user presses one switch, but the moment a user presses a switch, the microcontroller will detect that that switch has been pressed. What if during that time another switch is pressed?

But because the microcontroller is waiting for this switch to be pressed and it has now detected that this switch has been pressed, it will wait for it to be released, during that time if somebody else or the human being presses another switch and releases it, it is possible that the microcontroller will miss it, and therefore the way to handle such external events would be to give time slots and perform an action. If there is no action go to another task or another input, wait for it to be performed, if not you can keep on going from one to the second to the third to the fourth and so on.

So, polling external events using a method which is often called round-robin is a good idea and I can give you an analogy. Let us say I am giving a class, I am delivering a lecture in a classroom, a physical classroom where there are few students and these students want to ask me questions based on what I have taught. Now one way to deal with that is, set a rule that I will after I teach a particular module, let us say I teach a concept and at the end of the concept I will, starting from let us say left of the classroom, I will ask every student, have you understood it? Do you have any doubt? Have you understood it? Do you have any doubt? Have you understood it? Do you have any doubt? I can go from left to right and cover the entire class.

Now, if I do this method, if I use this method of trying to find if somebody has a question and would they like resolution of that question, this method of providing a resolution is called polling and this works well if the number of people in this exercise are limited and they are small. Now imagine the number of participants, number of students in the class increases. If that is the case, let us say I start polling this class from the left and there is a student on the right, extreme right and he has a doubt, he has a question, but till I come to that student, he will have to wait.

Whereas after I ask him what if the just the previous student had another question and he would have to wait till I come back after completing all the students again to that second last student, and so polling is problematic when the number of inputs, number of participants increases and this is where the idea of interrupt comes in. How does it work?

Imagine I change the rule of participation in the class, that instead of me after having taught a concept that I will individually ask each and every student if they have a doubt. Let us say I change the rule that whenever I teach a concept after that anybody who has a doubt can raise their hand. So, this act of raising hand is equivalent to generating an interrupt, of course when I am teaching and if somebody raises a hand, I will have the option of ignoring it because I might be facing the blackboard and after I have taught that concept I will turn around and I will see how many hands are raised.

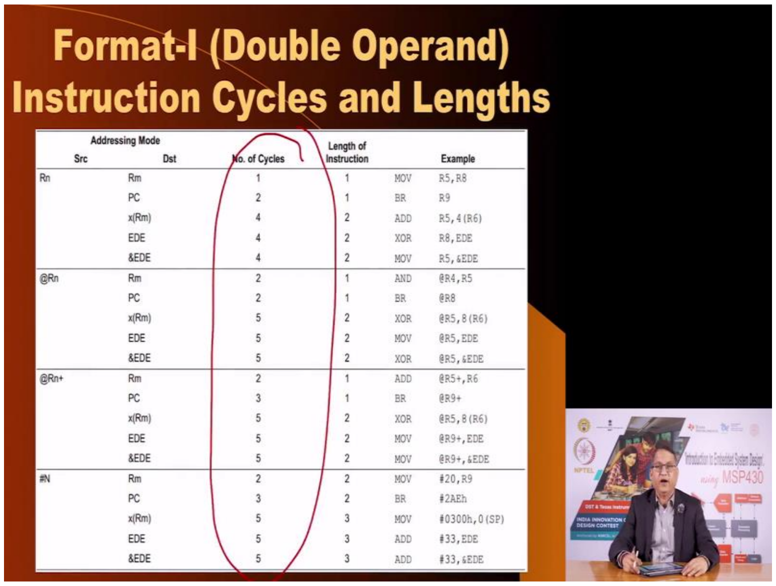
If there are no hands raised, what does it mean? That they, students have received my, that particular topic well and I can continue to the next topic. But if some students have a doubt they would raise hand and again, I would scan from left to the right and the one raised hand that I find in this scanning process from left to right, the first one I find, first hand that I find raised I will ask them what is the question and I will answer that and then I will go to the next raised hand and so on so forth.

Now imagine in this situation the first person raises a hand on the left, I ask what that question is, I answer that question, in the meantime another person towards the right had also raised the hand, but because I can only answer one question at a time. I choose to prioritize the person on the left and by the time I finish that question and I started scanning towards the right, I found that the second person has put his hand or her hand down.

For me, my job is done. If I find no other hand raised till the end of the room, I would imagine that the second person probably waited long enough did not get an opportunity or it is also possible that the first question that was raised was also the same question that the second person had to ask and therefore he or she found it unnecessary to continue to raise the hand.

So, this idea of raising a hand is very, very similar to this concept of interrupt. So, an interrupt is a mechanism by which a microcontroller system or a digital system can look at external events whenever they happen, they can respond to those events. Now before we come to that aspect, I want to show you how much time does MSP430 take to execute any given instruction.

(Refer Slide Time: 08:59)

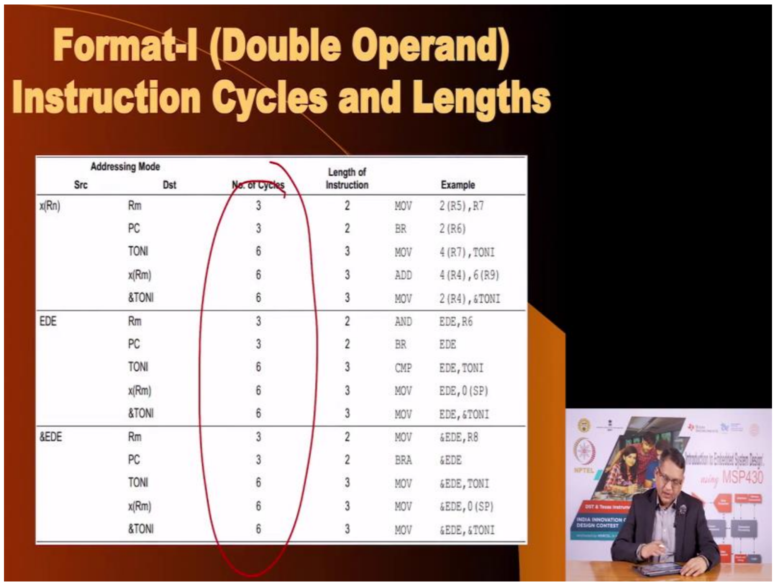


Format-I (Double Operand) Instruction Cycles and Lengths

Addressing Mode		No. of Cycles	Length of Instruction	Example	
Src	Dst				
Rn	Rm	1	1	MOV	R5,R8
	PC	2	1	BR	R9
	x(Rm)	4	2	ADD	R5,4(R6)
	EDE	4	2	XOR	R8,EDE
	&EDE	4	2	MOV	R5,&EDE
@Rn	Rm	2	1	AND	@R4,R5
	PC	2	1	BR	@R8
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R5,EDE
	&EDE	5	2	XOR	@R5,&EDE
@Rn+	Rm	2	1	ADD	@R5+,R6
	PC	3	1	BR	@R9+
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R9+,EDE
	&EDE	5	2	MOV	@R9+,&EDE
#N	Rm	2	2	MOV	#20,R9
	PC	3	2	BR	#2AEh
	x(Rm)	5	3	MOV	#0300h,0(SP)
	EDE	5	3	ADD	#33,EDE
	&EDE	5	3	ADD	#33,&EDE

So, I have listed here the various types of instructions and in this especially, look at this third column where it says, second column where it says how many cycles are required to execute a given instruction. And there are several such sheets.

(Refer Slide Time: 09:11)

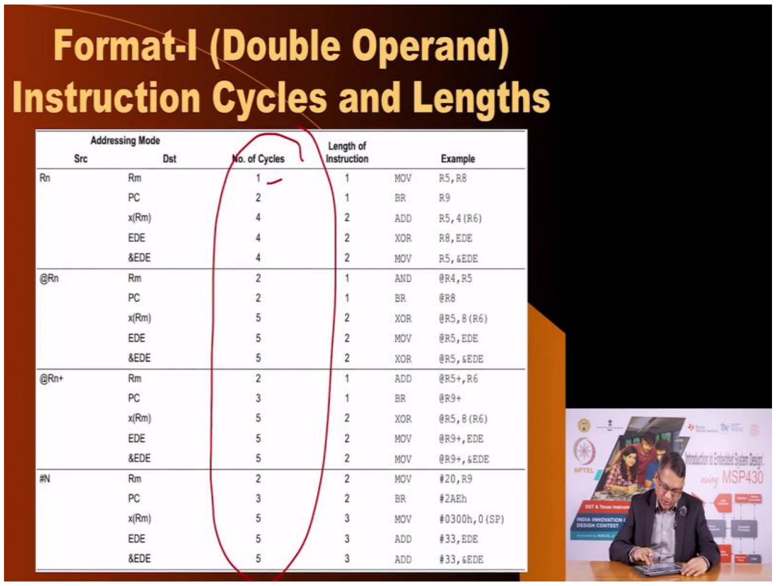


Format-I (Double Operand) Instruction Cycles and Lengths

Addressing Mode		No. of Cycles	Length of Instruction	Example	
Src	Dst				
x(Rn)	Rm	3	2	MOV	2(R5),R7
	PC	3	2	BR	2(R6)
	TONI	6	3	MOV	4(R7),TONI
	x(Rm)	6	3	ADD	4(R4),6(R9)
	&TONI	6	3	MOV	2(R4),&TONI
EDE	Rm	3	2	AND	EDE,R6
	PC	3	2	BR	EDE
	TONI	6	3	CMP	EDE,TONI
	x(Rm)	6	3	MOV	EDE,0(SP)
	&TONI	6	3	MOV	EDE,&TONI
&EDE	Rm	3	2	MOV	&EDE,R8
	PC	3	2	BRA	&EDE
	TONI	6	3	MOV	&EDE,TONI
	x(Rm)	6	3	MOV	&EDE,0(SP)
	&TONI	6	3	MOV	&EDE,&TONI

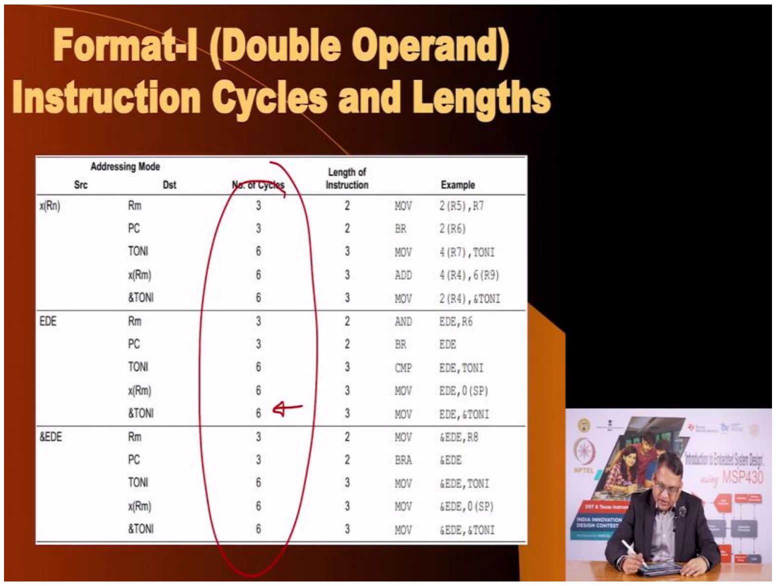
Here now you see, here, they are taking three cycles and the cycles here refer to the clock cycles.

(Refer Slide Time: 09:21)



Format-I (Double Operand) Instruction Cycles and Lengths

Addressing Mode		No. of Cycles	Length of Instruction	Example	
Src	Dst				
Rn	Rm	1	1	MOV	R5,R8
	PC	2	1	BR	R9
	x(Rm)	4	2	ADD	R5,4(R6)
	EDE	4	2	XOR	R8,EDE
	&EDE	4	2	MOV	R5,&EDE
@Rn	Rm	2	1	AND	@R4,R5
	PC	2	1	BR	@R8
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R5,EDE
	&EDE	5	2	XOR	@R5,&EDE
@Rn+	Rm	2	1	ADD	@R5+,R6
	PC	3	1	BR	@R9+
	x(Rm)	5	2	XOR	@R5,8(R6)
	EDE	5	2	MOV	@R9+,EDE
	&EDE	5	2	MOV	@R9+,&EDE
#N	Rm	2	2	MOV	#20,R9
	PC	3	2	BR	#2AEh
	x(Rm)	5	3	MOV	#0300h,0(SP)
	EDE	5	3	ADD	#33,EDE
	&EDE	5	3	ADD	#33,&EDE

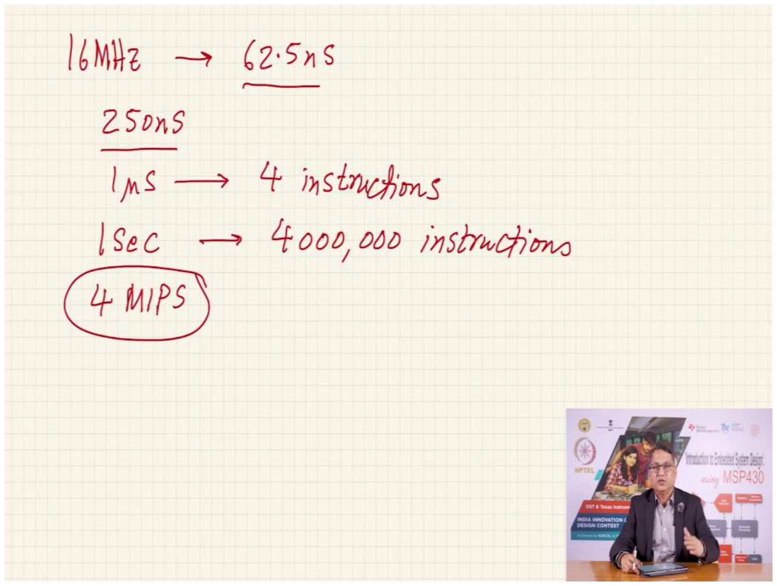


Format-I (Double Operand) Instruction Cycles and Lengths

Addressing Mode		No. of Cycles	Length of Instruction	Example	
Src	Dst				
x(Rn)	Rm	3	2	MOV	2(R5),R7
	PC	3	2	BR	2(R6)
	TONI	6	3	MOV	4(R7),TONI
	x(Rm)	6	3	ADD	4(R4),6(R9)
	&TONI	6	3	MOV	2(R4),&TONI
EDE	Rm	3	2	AND	EDE,R6
	PC	3	2	BR	EDE
	TONI	6	3	CMP	EDE,TONI
	x(Rm)	6	3	MOV	EDE,0(SP)
	&TONI	6	3	MOV	EDE,&TONI
&EDE	Rm	3	2	MOV	&EDE,R8
	PC	3	2	BRA	&EDE
	TONI	6	3	MOV	&EDE,TONI
	x(Rm)	6	3	MOV	&EDE,0(SP)
	&TONI	6	3	MOV	&EDE,&TONI

Similarly, so you have one, you have a range from 1 clock cycle for executing an instruction, going up to 6 clock cycles. So, this is the longest instruction. Now, let us also consider how much time, what is the frequency, what is the time period of this clock cycle?

(Refer Slide Time: 09:43)



So, if we, we know that MSP430 works at a maximum clock frequency of 16 megahertz. So, 16 megahertz is your clock frequency, this leads to a clock period of 62.5 nanoseconds, which means the fastest instruction will execute in 62.5 nanoseconds and the longest will be 6 and so it will be about 400, 380 nanosecond or so. Now, of course, we need not argue here exactly which instruction is taking how much time, let us take an average.

Let us say that all the instructions in my program are on an average taking 4 clock cycles, which means roughly we are talking of 62 into 4, roughly 260, let us you know round it off to 250 nanoseconds. 250 nanosecond is the average time any given instruction takes in a given program, which means if one instruction takes 250 nanoseconds that means in 1 microsecond I am going to execute 4 instructions. In 1 second, I will execute 4 million instructions.

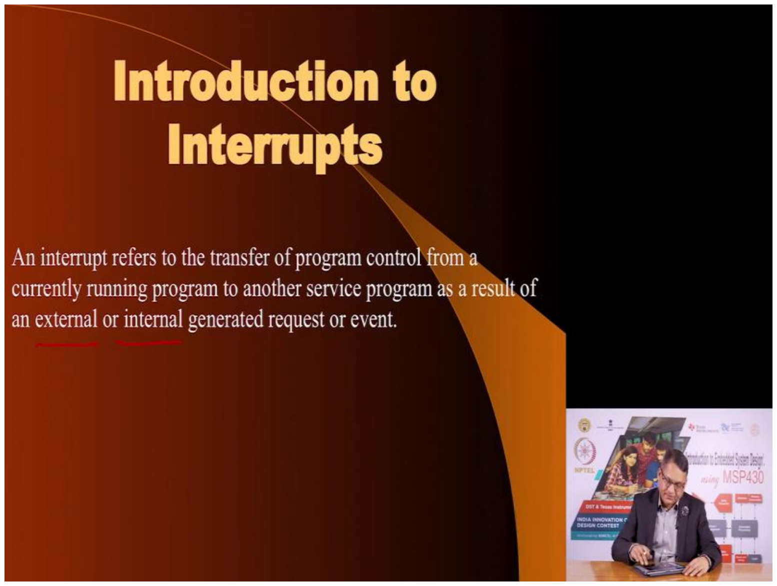
And this, in fact, refers to the capability or the speed of the processor, it is often expressed like this that for a sample program, for a reference program where we are assume, where we have assumed that each instruction will take 4 clock cycles, this MSP430 microcontroller can perform 4 MIPS, at the rate of 4 MIPS that is it can execute a program at the rate of 4 million instructions per second. Now that is a large number.

Human interaction, on the other hand, the act of pressing a switch and releasing it is a much slower process, which means if a human is going to press few switches, at this rate the microcontroller will be able to scan and poll many, many, many switches in 1 second. Even so, you would be deprived of executing the actual program while the microcontroller is waiting for a switch to be pressed, and therefore this idea of interrupt is brought into picture.

Now interrupt involves, the microcontroller is executing a program and let us say an external event wants attention, an external event here, for example, could be a user presses a switch. And so instead of the microcontroller continuing to execute the program that it was executing, it suspends it for some time, goes off to execute a subroutine which deals with this external event, in this case, a user has press a switch and then it returns back after it has found out which switch was pressed, it can come back to the main program and continue the execution of that program.

So, an interrupt refers to the transfer of control of the microcontroller from the existing program to execute a special subroutine, perform it and then come back quickly to resume the ongoing program that it was executing.

(Refer Slide Time: 13:14)

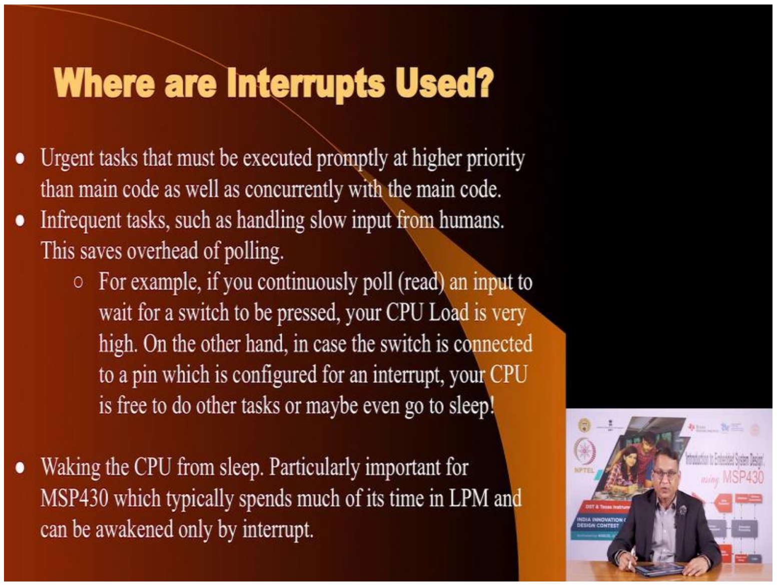


Introduction to Interrupts

An interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request or event.

Now, this event could be because of an external event, meaning somebody has pressed a switch or because of an internally generated request or event, for example, maybe the microcontroller has a ADC, Analog to Digital Converter. Analog to Digital Converter takes a long time to complete a conversion and after the conversion is over the ADC peripheral could generate an interrupt telling the microcontroller that whatever conversion was to be performed, it is over, kindly look at the result and do whatever you have to do. So, interrupts could be generated from external sources as well as internal peripherals.

(Refer Slide Time: 13:53)



Where are Interrupts Used?

- Urgent tasks that must be executed promptly at higher priority than main code as well as concurrently with the main code.
- Infrequent tasks, such as handling slow input from humans. This saves overhead of polling.
    - For example, if you continuously poll (read) an input to wait for a switch to be pressed, your CPU Load is very high. On the other hand, in case the switch is connected to a pin which is configured for an interrupt, your CPU is free to do other tasks or maybe even go to sleep!
- Waking the CPU from sleep. Particularly important for MSP430 which typically spends much of its time in LPM and can be awakened only by interrupt.

Now, where are interrupts used? Well, typically when you have an urgent task, you are performing a main program and some urgent task comes, you would like to deal with it by suspending your existing program dealing with this urgent tasks and coming back to continue to execute the ongoing program.

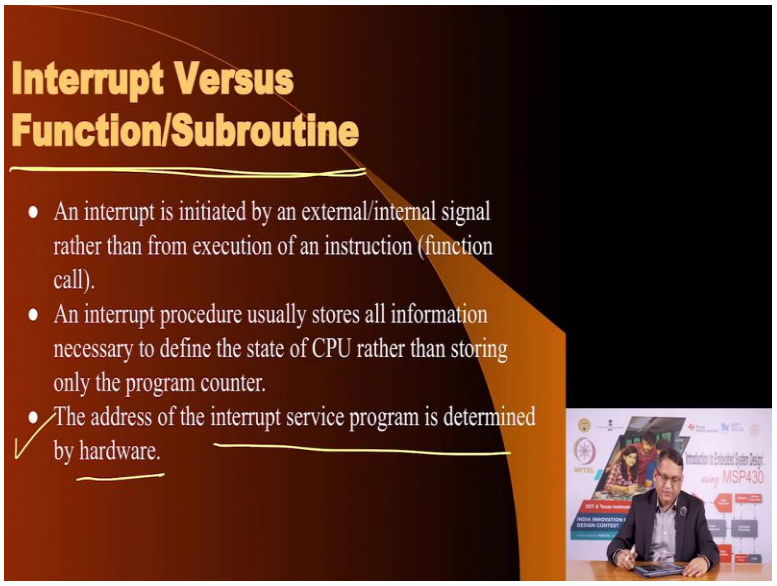
Infrequent tasks, which happen once in a while, you would like to deal with them using an interrupt and as I mentioned, human inputs are very slow, they may be infrequent compared to the rate at which a microcontroller can execute a program, human interaction is much, much infrequent and so you would, you could handle the external events generated by inputs by humans through interrupts.

Also, in the case of MSP430, and this is very important and specific to MSP430 because it has a lot of power-saving modes. Once a microcontroller enters a power-saving mode, it is going to suspend operation of a program and if you want to resume program execution, it is very important that you exit these low power modes. As we have discussed, there are several low-power modes, it has four or five low-power modes and the way to exit the low-power mode and continue, start, begin executing the program requires an interrupt.

And so in MSP430, when you are, when you have entered a low-power mode, the only way to get out of it is by way of an interrupt, whatever may be the source of that interrupt. We will discuss this when we discuss low-power modes and how interrupts are used to exit these low-power modes.

Now, how do you handle interrupts? As I mentioned, your main program that you were running would be suspended and you would execute a subroutine, and therefore it may appear that an interrupt is performing a response to an interrupt is like executing a subroutine, indeed it is.

(Refer Slide Time: 16:00)



Interrupt Versus Function/Subroutine

- An interrupt is initiated by an external/internal signal rather than from execution of an instruction (function call).
- An interrupt procedure usually stores all information necessary to define the state of CPU rather than storing only the program counter.
- The address of the interrupt service program is determined by hardware.

But it is different from a conventional function or subroutine, it is definitely a little different for various reasons. In a normal function or subroutine, you call that function. But in an interrupt, you do not call that interrupt subroutine, the interrupt subroutine gets called. Why? Because it is often the result of a asynchronous operation like a user pressing a switch.

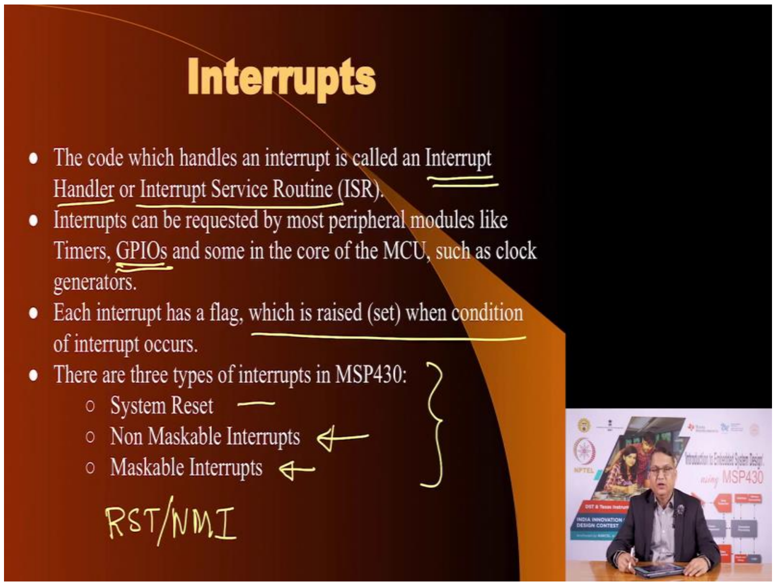
In a normal subroutine, you, of course, suspend your current execution and you go off to execute that subroutine. But after the subroutine is finished, you want to come back to where you were earlier, and therefore the way to do that would be to save the program counter value onto the stack.

Now in the case of interrupt subroutine, you do not know when you get called and therefore you need to not only store the address of the instruction which you are executing or would have executed on to the stack, but also the state of the microcontroller and the state of the microcontroller here refers to various flags like zero, carry and so on. So even those need to be saved on the stack.

Also, you need to know that for a given interrupt, where is the subroutine. Because there may be several sources of interrupts and you cannot go to a single address to find out who created, who generated that interrupt, and therefore let me execute this particular program in response to that input. The way it is handled is that for all the various inputs and all the various sources of interrupts, you associate a unique address, where you locate your, where you put your subroutine in which would be executed when such a interrupt happens.

So, that is this aspect the address of the interrupt subroutine is determined by the hardware. The address of a normal subroutine is determined by the software. Depending upon how big your code is, the assembler or compiler will put that subroutine at someplace, but and that place can change. But the subroutine for interrupt is decided by the microcontroller, meaning it is decided by the hardware and when we see the details, you will realize what I mean.

(Refer Slide Time: 18:26)



Interrupts

- The code which handles an interrupt is called an Interrupt Handler or Interrupt Service Routine (ISR).
- Interrupts can be requested by most peripheral modules like Timers, GPIOs and some in the core of the MCU, such as clock generators.
- Each interrupt has a flag, which is raised (set) when condition of interrupt occurs.
- There are three types of interrupts in MSP430:
  - System Reset
  - Non Maskable Interrupts
  - Maskable Interrupts

RST/NMI

So, interrupts require that a subroutine be executed to in response to this event, and that subroutine is often called, very popularly called Interrupt Subroutine or Interrupt Handler. Interrupts can be generated by most peripherals in MSP430, they can be generated by general-purpose input-output pins here. As we know that on the MSP430 that we are using on our lunchbox, there are two ports P1 and P2. Any input pin on P1 and P2 can generate an interrupt. Each interrupt has a flag and when that flag is raised, the condition for interrupt is satisfied and the microcontroller will do something in response to that condition.

Also broadly on the MSP430, there are three types of interrupts. One is the system reset that is, we have discussed the reset part. Each time you press, generate a reset for whatever, whatever may be the source of that reset, it is treated like an interrupt.

Also, the reset pin is shared with the non-maskable interrupt, you may recall that that pin is actually called RST/NMI, which means the same pin can be used as a reset input as well as a non-maskable interrupt. NMI stands for non-maskable interrupt. And apart from these two sources, you also have many, many peripherals which provide or generate what are called as maskable interrupts. We are going to discuss, what is the meaning of non-maskable versus maskable very shortly.

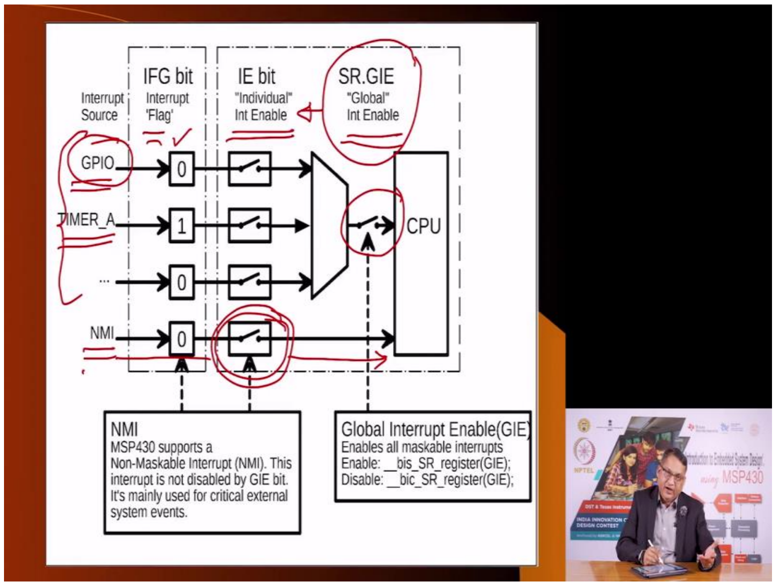
Now, what is the meaning of maskable interrupt and non-maskable interrupt, and let me take you to usual normal life to illustrate that idea? Like, let us go back to that classroom example. Somebody has raised a hand, but because I was doing something important on the blackboard, I can choose to ignore the raised hand, that would mean that I have masked that source of interrupt.

On the other hand, let us say that this classroom has a fire alarm, and anytime the fire alarm rings, would I wait for whatever I am doing on the blackboard to finish it? No, I would run for my life. So the source of this particular interrupt, as illustrated by a fire alarm, is an example of a non-maskable interrupt. I do not have the freedom or the liberty to ignore such an event, such an interrupt and such interrupts are called non-maskable interrupt.

So, broadly the classification of interrupts is maskable interrupts, such interrupts, such events, which can be ignored or suspended for some time, meaning suspended as in you need not respond to them by executing that interrupt subroutine, you can let them wait, and some interrupts which are non-maskable. MSP430 offers both of them.

In the case of MSP430, will see in the next slide that even the non-maskable interrupt can be disabled. In general, in general microcontrollers, non-maskable interrupt cannot be disabled, cannot be suspended at all. But here there is a little flexibility that there is a mechanism to even ignore that or disable them.

(Refer Slide Time: 22:00)



NMI
MSP430 supports a Non-Maskable Interrupt (NMI). This interrupt is not disabled by GIE bit. It's mainly used for critical external system events.

Global Interrupt Enable(GIE)
Enables all maskable interrupts
Enable: __bis_SR_register(GIE);
Disable: __bic_SR_register(GIE);

So, this diagram here is an important illustration of how interrupts are generated, detected, and then responded to. So as I mentioned, the source of the interrupt could be input-output pins here. The source of an interrupt could be the timer counting time. It could be the, a non-maskable interrupts. Now the first important consideration that a interrupt will be serviced would be that it must generate a request, it must generate an event. And when a event is generated, it is captured in a register called interrupt flag, interrupt flag register.

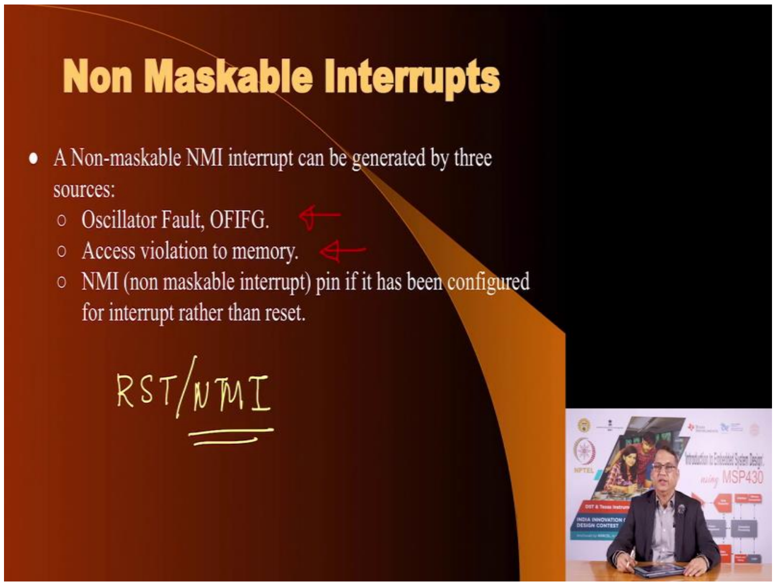
So, if the flag is 1, that means there is an external event. However, it is up to you do you want to respond to that event or do you want to ignore it and the second part of this block diagram shows you that. Now you see this GPIO could generate an interrupt, but you could ignore it if this switch is open. So, you have a register called interrupt enable register if you have disabled those interrupts, this request for an interrupt will not proceed forward to the CPU.

On the other hand, you see this non-maskable interrupt can only be stopped by this switch thereafter there is nothing to stop it. Once if this switch is turned on, it can go all the way through. On the other hand, the rest of these interrupts, these are all maskable interrupts you need a individual interrupt enable bit, but there is also a global sort of a switch, if this switch is turned off, then no matter whether an external event happens, no matter whether it has been individually enabled, the CPU will not respond to such interrupts.

But for non-maskable interrupt, there is nothing, there is no roadblock apart from this switch here, there is no roadblock, if this which has been enabled, the moment an external non-maskable interrupt happens, the CPU will have to respond to it. And so we will see how these switches, how do you control them and how do you manipulate them so that external events could lead to an interrupt in response to which the microcontroller will execute a interrupt subroutine.

So, there are three aspects interrupt flag register, interrupt individual interrupt enable bits and a global interrupt enable. For most, for all the, for all the maskable interrupts, you have to deal with all these three parts. For the non-maskable interrupt, you do not have to deal with this global interrupt enable, you only have to deal with the fact that the non-maskable interrupt should generate an event and then it should be enabled, then it would reach from here to the CPU. So now non-maskable interrupts can be generated from several sources.

(Refer Slide Time: 25:14)



Non Maskable Interrupts

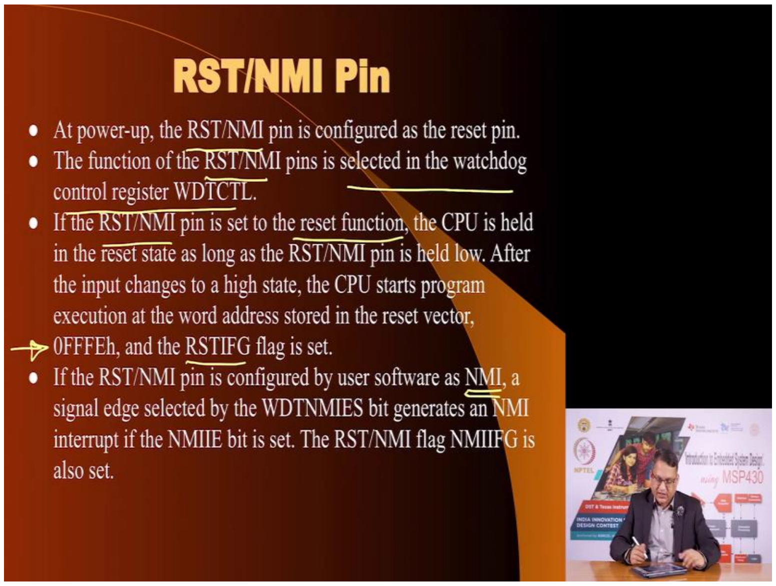
- A Non-maskable NMI interrupt can be generated by three sources:
  - Oscillator Fault, OFIFG.
  - Access violation to memory.
  - NMI (non maskable interrupt) pin if it has been configured for interrupt rather than reset.

RST/NMI

It can be generated because one of the oscillators, and you know that there are three oscillator sources in your MSP430 microcontroller if any of the oscillator is not working the way it should it can generate non-maskable interrupt.

Also, you have program memories; RAM, if you are trying to read or write from locations which are not available on your microcontroller, it would lead to memory access violation that also will generate a non-maskable interrupt. And the third is the non-maskable interrupt pin, which as we have mentioned earlier is actually shared with the RST pin, RST/NMI. If you provide a signal on this, then it could be construed as a non-maskable interrupt. This can also be used to generate a non-maskable interrupt. So these are the three non-maskable interrupt sources.

(Refer Slide Time: 25:59)



RST/NMI Pin

- At power-up, the RST/NMI pin is configured as the reset pin.
- The function of the RST/NMI pins is selected in the watchdog control register WDTCTL.
- If the RST/NMI pin is set to the reset function, the CPU is held in the reset state as long as the RST/NMI pin is held low. After the input changes to a high state, the CPU starts program execution at the word address stored in the reset vector, 0FFFEh, and the RSTIFG flag is set.
- If the RST/NMI pin is configured by user software as NMI, a signal edge selected by the WDTNMIES bit generates an NMI interrupt if the NMIIE bit is set. The RST/NMI flag NMIIFG is also set.

Here is details of, here are the details of the RST/NMI pin. At power-on, when you power up, when you turn the power on, the RST/NMI pin is configured as reset pin, and the function of the RST/NMI pin is selected in the watchdog control register.

Then if the RST/NMI pin is selected as a reset function, the moment you press this switch that is you hold the logic on this pin to 0, it will reset the processor and it will keep it in reset state as long as you hold this pin 0, and when you release it, it will let the microcontroller execute the program for which it will go to the reset vector. As we have discussed, the reset vector is at this address and it will, from this address, it will find the address of the first instruction in the flash program memory and it will execute it. It will also set up this RSTIFG flag so that you know, what is the source of that reset.

If on the other hand, this pin has been configured through software as a NMI, then a single edge selected by this register generates a NMI, non-maskable interrupt, and it will go to the vector for non-maskable interrupt and execute a program in response to it.