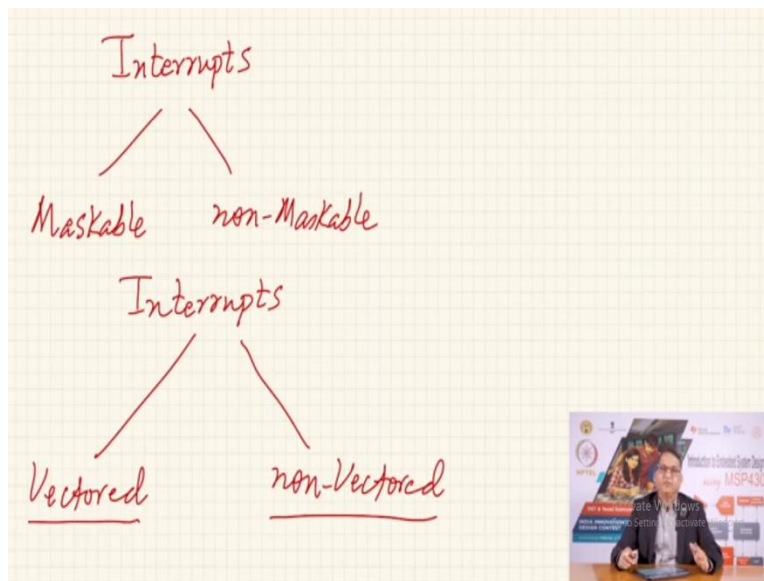


**Introduction to Embedded System Design**  
**Dhananjay V. Gadre (NSUT, New Delhi)**  
&  
**Badri Subudhi (IIT, Jammu)**  
**Indian Institute of Technology, Delhi**  
**Module 8**  
**Lecture 26**  
**Interrupts in MSP430 - Continued**

(Refer Slide Time: 00:38)



Let us resume our discussion on the interrupts that we have been having, in this session today. Now, let me also mention a little bit more about these interrupts. As we have seen, interrupts can be classified like - maskable interrupts or non-maskable. We have seen that on our MSP430, we have both varieties.

There is one more way to classify interrupts and that deals with vectored and non-vectored. What is the meaning? When I use the term vectored interrupts, that means whenever a interrupt happens, I know exactly where to go looking for the subroutine to deal with that interrupt. I can also have non-vectored interrupts which means well, an interrupt has happened but I exactly do not know where to go and therefore the system must provide me some information to help me locate the subroutine to handle this non-vectored interrupt.

Well, MSP430, most of the interrupts are vectored interrupts, all the interrupts are vectored interrupts. The source of the interrupt also supplies the address. In fact, this address is available

in the memory map of MSP430. The top, that is the highest memory locations are reserved for storing vectors for each of the interrupt sources. Also, now imagine that at a given time, when the microcontroller is ready to, you know, branch off for executing a interrupt subroutine, what if two interrupts happened at the same time? Of course, microcontroller cannot go to satisfy the needs of both the interrupt sources at the same time, so it must priorities.

(Refer Slide Time: 02:44)

**Vectored Interrupts**

- MSP430 uses vectored interrupt.
- In a vectored interrupt, the source that interrupts supplies the branch information to the computer.
- Vector address is stored in the vector table.
- Each interrupt vector has a distinct priority, which is used to select which vector is taken if more than 1 interrupt is active.
- Priority is fixed: cannot be changed by user. A higher address means higher priority.
- Reset vector has highest address 0xFFFE.

FFFFE & FFFF  
-----  
FFFC & FFFD

The slide also features a small inset image of a presenter in a video lecture format.

And this is called, what is the interrupt priority - which means the one with, if two events happen, the one with higher priority will be serviced first. After the completion of that interrupt, you can go and execute a subroutine for the second interrupt.

The reset vector, and we see how this is prioritized in MSP430. The higher vector address means a higher priority. And so reset vector has higher priority, why? Because it is located at this address and just to repeat, the address is a two addresses that is your instruction is at FFFE and FFFF, alright? So you are going, so the vector at this address will have the highest priority. The one after that will be FFFC and FFFD would be the lower priority interrupt compared to this and so on.

(Refer Slide Time: 03:45)

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Timer Flash key violation PC out-of-range <sup>(1)</sup>	PORIFG RSTIFG WDTIFG KEYV <sup>(2)</sup>	Reset	OFFFh	31, highest
NMI Oscillator fault Flash memory access violation	NMIFG OFFG ACCVFG <sup>(2)(3)</sup>	(non)-maskable (non)-maskable (non)-maskable	OFFCh	30
Timer1_A3	TA1CCR2 CCFG <sup>(4)</sup>	maskable	OFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCFG, TAIFG <sup>(4)(5)</sup>	maskable	OFF8h	28
Comparator_A+	CAIFG <sup>(6)</sup>	maskable	OFF6h	27
Watchdog Timer	WDTIFG	maskable	OFF4h	26
Timer0_A3	TA0CCR0 CCFG <sup>(4)</sup>	maskable	OFF2h	25
Timer0_A3	TA0CCR2 TA0CCR1 CCFG, TAIFG <sup>(4)(5)</sup>	maskable	OFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 DC status	UCARXIFG, UCBORXIFG <sup>(7)(8)</sup>	maskable	OFFEh	23
USCI_A0/USCI_B0 transmit USCI_B0 DC receive/transmit	UCA0TXIFG, UCB0TXIFG <sup>(7)(8)</sup>	maskable	OFFCh	22
ADC10 (MSP430G2x53 only)	ADC10IFG <sup>(9)</sup>	maskable	OFFAh	21
			OFF8h	20
IO Port P2 (up to eight flags)	P2IFG.0 to P2IFG.7 <sup>(10)</sup>	maskable	OFF6h	19
IO Port P1 (up to eight flags)	P1IFG.0 to P1IFG.7 <sup>(10)</sup>	maskable	OFF4h	18
			OFF2h	17
			OFF0h	16
See <sup>(1)</sup>			OFFDEh	15
See <sup>(2)</sup>			OFFDEh to OFFC0h	14 to 0, lowest

Here is the clipping from the MSP430 data sheet which lists all the interrupts - what are their names, the source and what is the flag referring to the interrupt flag that we mentioned and what is the address of the vector for that interrupt and what is the priority. Of course, as I mentioned '31, highest' priority is for reset and as you go down, the priority decreases which means if two interrupts happened, the one with the highest priority will be serviced first.

And as you see, we, the sources of interrupt are - Power up condition, external reset - that is a user pressing the reset button or non-maskable interrupt or a timer here - comparators, watchdog timers. Another timer - the serial communication protocols, the ADC and last but not the least, the two ports that we have on MSP430 G 2553.

In fact, you would remember that MSP430 G2553 is actually available in several physical footprints. The one which we have on our lunch box is in a dip format; it has two ports. But if the MSP430 is chosen to be of the SMD version, it has three ports. Port 1, Port 2 and Port 3. But, Port 3 does not have any interrupt capability. The only interrupt capability you have for this microcontroller number is for port 1 and port 2. And as you see, port 1 has the lowest priority in this and also here you see these are all maskable interrupts and only the NMI interrupt is non-maskable. So let us proceed further.

Now, as I mentioned, an interrupt flag is an important element in the whole process of an interrupt being generated, being recognized and being serviced. And the interrupt flag, once it is

generated, it will automatically vector to a address location and usually it will have a single source. But there are peripherals where a single vector is going to help you generate sources from several pins.

(Refer Slide Time: 06:19)



For example port 1 or port 2 - each of these ports as you know has 8 pins and each of the 8 pins can generate an interrupt. But if the pin refers to port 1, all the 8 pins will have a single vector. So it will take you to the same location and it will have to execute the same subroutine at that location. In that subroutine you will have to identify which was, of all the 8 pins, which caused the interrupt, right? So we will deal with this.


So when you are when you are ready incorporate interrupts in your system, in your programming, in your embedded system, there are a few things that you have to keep in mind. You must keep the interrupt subroutine very short. And one of the ways to ensure that is to not use any delay subroutine in your interrupt function, in your interrupt subroutine - why?

When you take too long in that interrupt subroutine, what are you doing is - you are constraining the rest of the execution of the main program or you are starving other sources of interrupts. You want to give all the sources of interrupts a fair chance, a fair participation, therefore you must keep the interrupt service routine very short. Now, what if you want to actually execute a bigger program? The way to deal with that is to interact, deal with the requirement in the main program, not in the interrupt subroutine.

When you are executing a interrupt subroutine, by default all the other maskable interrupts are disabled- that is the meaning. Therefore, you must keep the execution of the current interrupt subroutine to a shortest possible program length so that you can quickly finish it. When you go back, it will enable the interrupts, so that other interrupt sources could get a chance.

Now, for interrupt to be accepted, there are several conditions that have to be satisfied. The first is that because interrupts are often generated from external sources, which means they are asynchronous to the execution of the current, the program on the microcontroller, it does not mean the moment and interrupt is generated, the microcontroller would suspend whatever it is doing and will rush off to execute the subroutine for that interrupt.

(Refer Slide Time: 08:53)



**Interrupt Acceptance**

1. Any currently executing instruction is completed.
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR (Status Register) is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
5. The interrupt flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6. The SR is cleared (for use within the ISR). This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7. The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.

The slide features a dark background with orange and white text. A white bracket on the right side groups items 1 through 5. In the bottom right corner, there is a small inset image of a man in a suit sitting at a desk with a laptop, with a presentation screen behind him showing various logos and text.

The very, at the very least, it will complete the current program, current instruction, if it has started fetching that instruction it will complete it. And as you know, the length of these instructions take one clock cycle or it could take 6 clock cycles. So if in the first clock cycle, an interrupt event it happens, it will have to wait in the 6 clock cycles associated with the current instruction are completed.

Also, once the interrupt is recognized, if there was no interrupt, the microcontroller would have gone to the next location to fetch the next instruction and execute it. But now because interrupt has happened, it must store the value of the program counter somewhere and this somewhere is,

location is a memory location in the RAM and we call this memory location, special region in that RAM, we call it 'stack'.

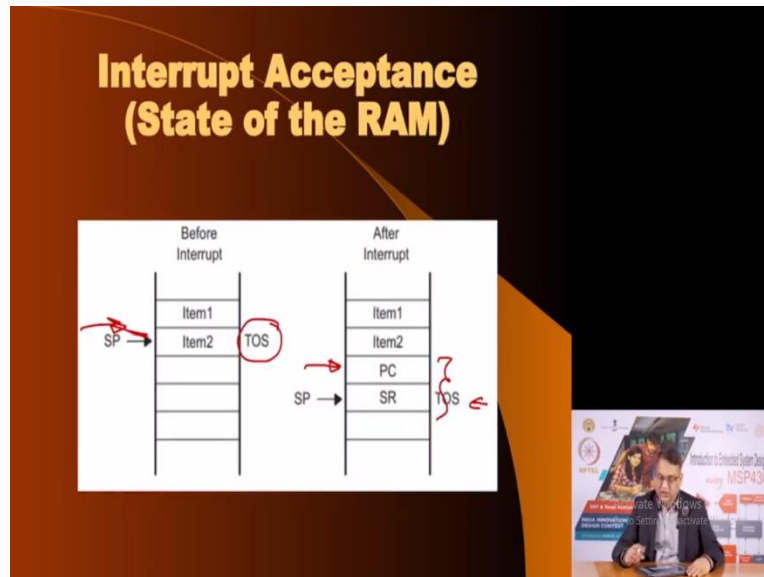
And so the microcontroller saves the value of the program counter on the stack. Then apart from the program counter, the state of the microcontroller is reflected in the status register. Even this is saved on the stack. So that when you resume your interrupted program, you can resume with the state which you had before the interrupt happened.

Now, if multiple interrupts have happened, it would priorities and it would give the highest priority to the highest priority, higher priority interrupt and execute an instruction - subroutines related to that interrupt. The interrupt flag resets automatically on single source flags, meaning if there a source of interrupt which is not shared with other resources, the moment the interrupt is recognized, it will reset this flag, so that next time another event happens on that resource, it can be captured.

It also clears the status register within the interrupt subroutine. So when you enter the interrupt subroutine, the status register reflected, reflecting from the main program is saved on the stack and within the interrupt subroutine, you get a clean status register all the bits are reset to 0 and this will allow you to actually exit low power modes. The GI, that is the Global Interrupt bit which is like the global switch is also cleared and further interrupts are disabled - which means when you are in a interrupt subroutine, if any other interrupt happens, it is going to have to wait.

Once these conditions are all met, then before you go into the subroutine the interrupt vector is loaded on the program counter and from there, the microcontroller start fetching and executing the instructions of that subroutine.

(Refer Slide Time: 11:53)



To help come back to the main program, so this is your, this is a reflection of what will be the state of the RAM. So let us say, before the interrupt happened, the micro, the stack pointer was pointing to this address and whenever the interrupt happens, it has to store to contents, two words one is the program counter and the other is status register. So now TOS stands for top of the stack. So what was top of the stack here is now at this location four memory locations are used to store the, these to register; the program counter and the status register.

(Refer Slide Time: 12:42)

## Returning from an Interrupt

1. The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.
2. The PC pops from the stack and begins execution at the point where it was interrupted.

The diagram shows two stack states: 'Before' and 'After'. In the 'Before' state, the stack contains Item1, Item2, PC, and SR. The stack pointer (SP) points to SR, and the top of stack (TOS) is at the top. In the 'After' state, the stack contains Item1, Item2, PC, and SR. The stack pointer (SP) points to Item2, and the TOS is at the top. The SR and PC are popped from the stack.

Now, once you are inside the interrupt subroutine and you have executed the program, you would do the same things that you did while getting into the interrupt subroutine in the reverse order. The status register of the previous setting will be popped from the stack which means when you had, when you had interrupted, when you had been interrupted in the main program, the status register was saved on the stack it will be recovered.

And the program counter is also recovered from the stack, that value, put in the program register so that you can resume executing the program which got interrupted, which got suspended - that we call as the main program, so this is shown here. Now, for handling interrupts, for the purpose of illustration here, we are going to use the input output pins. And so we are going to look at all the registers which help manipulate, which help control or which help, you know, handle all the interrupts associated with the GPIO; general purpose input output pins. And we have three register which help us manage the interrupts associated with the input output pins.



(Refer Slide Time: 14:06)

## Interrupt Registers for GPIO

- All these registers are 8-bit registers. ←
- PxIE - Interrupt enable register ← P1IE
- PxIES - Interrupt edge selection register ←
- PxIFG - Interrupt flag register ←

The diagram illustrates the interrupt mechanism. It shows several interrupt sources: GPIO, TIMER\_A, and NMI. Each source has an IFG bit (Interrupt Flag Generated) and an IE bit (Interrupt Enable). The IFG bit is set when an interrupt occurs. The IE bit is used to enable or disable the interrupt. A red circle highlights the IE bit, and a blue circle highlights the IFG bit. A master switch labeled 'Global Interrupt Enable (GIE)' is also shown, which enables all maskable interrupts. A note states: 'NMI MSP430 supports a Non-Maskable Interrupt (NMI). This interrupt is not disabled by GIE bit. It's usually used for critical external system events.'

All these registers are 8 bit width. We have an interrupt enable register call PXIE; X here are referred to which port we are talking of. For example, if you are talking of port 1, that register will be P1IE. We have a interrupt edge selection register, why? Because you could generate and interrupt on the rising edge of a signal on a port pin, you could also generate and interrupt on the falling edge.

So you have to tell the microcontroller, do you want to be interrupted when the signal goes like this or do you want to be interrupted when the signal goes like this. So that is indicated to the microcontroller by writing appropriate values into this register called interrupt edge selection register. And the third, the most important register is the interrupt flag register and for reference, I have a replicated that previous block diagram of interrupts, so that we can assign or we can locate these registers in this block diagram.

So program interrupt enable register is this part, here, here. So if you want that a particular interrupt should be enabled, you must enable it in this register. So it affects this red part. The edge selection and the flag register together are here and so you must get a 1 in this blue bit, if that is 1, and it this switch which is represented by that red circle, if that switch is closed, then only it would generate an interrupt, how provided this master switch, let me color it in a third color master switch here, is also turned on.

And that is by a flag called global interrupt enable flag and it can be enabled in various ways, it can also be disabled and we will see two methods of controlling this flag in a program subsequently. So to reiterate we have three registers that we must manipulate so as to handle interrupts on the input output pins of MSP430 microcontroller. Let us proceed forward.

(Refer Slide Time: 16:42)

**PxIE**

- PxIE - Interrupt enable register
- 8 Bit Register corresponding to 8 pins on each port.
- Setting any bit as '0' in this register disables the interrupt for the corresponding pin.
- All the bits in this register are 0 by default.
- Setting any bit as '1' in the register enables the interrupt for the corresponding pin.

Hand-drawn diagram of the PxIE register:

7							0
PxIE	0	0	0	0	0	0	1

The diagram shows an 8-bit register labeled 'PxIE' with bits numbered 7 to 0. The bits are 0, 0, 0, 0, 0, 0, 0, 1. A handwritten label 'P1.0' is placed above the bit 0 position.

Now, P1IE. that is, port 1, if I want to talk about port 1, interrupt enable. This is a 8 bit register corresponding to the 8 pins that you have for port 1. If you want to enable interrupt on any of the pins, you must write 0, you must write 1 in that bit. So let me say, draw the 8 bits here. So this would be your P1IE, this is bit 0, this is bit 7. If I write a 1 here, that means I want interrupt associated with P1.0 port 1 bit 0, to be enabled.

Of course, a signal has to be applied to port 1 bit 0 for interrupt to actually happen, but this you are enabling it. And if you want to disable it, you write a 0. So if I write 0 in the rest, that means I do not want to be interrupted by any signal on any of the pins of port 1 except bit 0. This is the interpretation of port 1 interrupt enable register.

(Refer Slide Time: 18:00)

**PxIES**

- PxIES - Interrupt edge selection register
- 8 bit register corresponding to 8 pins on each port.
- The bits of this register select the interrupt edge transition type on the corresponding pins.
- Setting a bit as '0' in the register enables interrupt flag when transition is from low to high.
- Setting the bit as '1' in the register enables interrupt flag when transition is from high to low.

PIIES [ ] [ ] [ ] [ ] [ ] [ ] [ 0 ] [ 1 ]

The diagram shows a horizontal register with 8 bits. The last two bits are labeled '0' and '1'. A vertical arrow points down to the '0' bit, and another points up to the '1' bit. The text 'PIIES' is written to the left of the register.

Second in the line is the interrupt edge selection register. Like I mentioned, you can generate an interrupt on the rising edge and you can generate an interrupt on the falling edge. You have to tell the micro controller which edge would you like to generate an interrupt and again here, there are 8 bits. And here I am going to write the name of the register this one is P1 IES, interrupt edge selection.

If I write a 1 here, that means it will generate an interrupt when there is a transition from high to low; so 1 means high to low. So a falling edge will generate an interrupt if you have written a 1 in that corresponding event. If you write 0, that means you want an interrupt in the rising edge. So depending upon your requirement, depending upon the state of the logic on that particular pin, you can write a 1 or a 0. Corresponding bit in the previous register interrupt enable register has to be 1, otherwise writing a 1 or 0 here makes no sense whatsoever.

(Refer Slide Time: 19:20)

**PxIFG**

- PxIFG - Interrupt flag register
- 8 bit register corresponding to 8 pins on each port.
- A bit of the interrupt flag register is set when the selected interrupt edge occurs on the corresponding pin.
- Using software PxIFG bits can be set or reset.
- PxIFG bits must be reset after the interrupt via software. Setting the PxIFG via software can generate software initiated interrupts.

P1.0

P1IFG [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ 1 ]

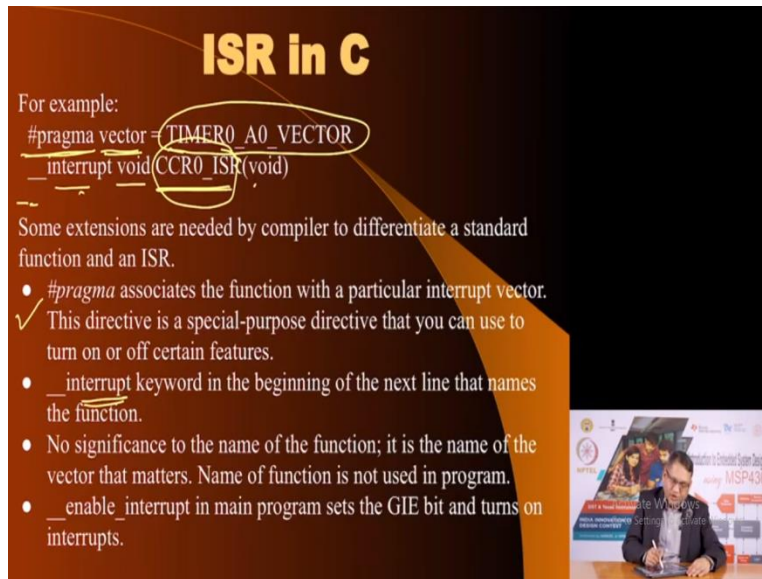
The slide features a dark background with a light-colored curved shape on the right side. A small inset image shows a person presenting in front of a screen displaying technical content related to the MSP430 microcontroller.

And the third is interrupt flag register. This flag register when, if it becomes 1, that it will become 1 upon which condition? That you have enabled the register and an external signal which is qualified by the edge selection register has been received, then and then only, a particular bit in this register will be set to 1. And if that bit is set to 1, subject to other conditions being met, it will lead to an interrupt.

So let me again show it here. So it which is P1IFG, if this bit is 1, that means and interrupt on port 1 bit 0 has been recognized and if the global interrupt enable bit is enabled, this will fly; it will interrupt the CPU and the CPU will go to the interrupt vector associated with port 1 and from there it will execute the subroutine and you can do whatever you want within that subroutine. That is the meaning of this. Of course these bits can also be set or reset in, by software, by writing a 1 or 0 and this will generate a software controlled interrupt that is the meaning. Normally, you do not want to do that, but that is also possible.

So let us see what is the flow of the program when we are dealing with interrupts, this is very important. Now in C, how do you tell the program, when you are writing a program in C language, how do you to tell the compiler that this subroutine that you are writing is to, for the purpose of handling a particular interrupt.

(Refer Slide Time: 21:12)



## ISR in C

For example:

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void(CCR0_ISR(void))
```

Some extensions are needed by compiler to differentiate a standard function and an ISR.

- `#pragma` associates the function with a particular interrupt vector.
- ✓ This directive is a special-purpose directive that you can use to turn on or off certain features.
- `__interrupt` keyword in the beginning of the next line that names the function.
- No significance to the name of the function; it is the name of the vector that matters. Name of function is not used in program.
- `__enable_interrupt` in main program sets the GIE bit and turns on interrupts.

© 2012 Texas Instruments Incorporated. All rights reserved. MSP430

And you do that by using a # declaration called #pragma. This is a keyword vector. You are telling that you are going to deal with timer 0, a 0 vector and what program to be executed in response to this vector? This is the name of the program. So you, this is by the way, double this, interrupt void these are all necessary, you cannot change these. This, is your choice, this name is your choice; you can put your own name.

So you have to have a #pragma, name given. Then this is a keyword here and the name itself the name of the subroutine is your choice, you can put whatever you want, of course it should not be a reserved keyword or anything else this is required. Once you to write this in your C program, you must therefore, there after write this subroutine associated with this label. Now void here means you are not passing any parameter to the subroutine nor is it causing any result back to you. Then how does a interrupt program help you? Through way off shared memory locations, alright? Variables. So we will see how that is handled.

Now I have put together this list from the MSP430 header file so that you know what are the names for each of the vectors; like for timer 0, what is the vector and so on.

(Refer Slide Time: 22:55)

```
929/.....
930 * Interrupt Vectors (offset from 0xFFE0)
931 ...../
932
933 #define VECTOR_NAME(name)      name##_ptr
934 #define EMIT_PRAGMA(x)         _Pragma(#x)
935 #define CREATE_VECTOR(name)    void (* const VECTOR_NAME(name))(void) = &name
936 #define PLACE_VECTOR(vector,section) EMIT_PRAGMA(DATA_SECTION(vector,section))
937 #define ISR_VECTOR(func,offset) CREATE_VECTOR(func); \
938                               PLACE_VECTOR(VECTOR_NAME(func), offset)
939
940 #ifndef __ASM_HEADER__ /* Begin #defines for assembler */
941 #define TRAPINT_VECTOR        ".int00" /* 0xFFE0 TRAPINT */
942 #else
943 #define TRAPINT_VECTOR        (0 * 1u) /* 0xFFE0 TRAPINT */
944 #endif
945 #ifndef __ASM_HEADER__ /* Begin #defines for assembler */
946 #define PORT1_VECTOR          ".int02" /* 0xFFE4 Port 1 */
947 #else
948 #define PORT1_VECTOR          (2 * 1u) /* 0xFFE4 Port 1 */
949 #endif
950 #ifndef __ASM_HEADER__ /* Begin #defines for assembler */
951 #define PORT2_VECTOR          ".int03" /* 0xFFE6 Port 2 */
952 #else
953 #define PORT2_VECTOR          (3 * 1u) /* 0xFFE6 Port 2 */
954 #endif
955 #ifndef __ASM_HEADER__ /* Begin #defines for assembler */
956 #define ADC10_VECTOR          ".int05" /* 0xFFEA ADC10 */
957 #else
958 #define ADC10_VECTOR          (5 * 1u) /* 0xFFEA ADC10 */
959 #endif
```

So here is the, as you can see, this is the vector for trap, that is the non-maskable interrupt. No, this is not the non-maskable interrupt, it should come here somewhere. So this is a trap vector. Then you have port 1 vector, that is the vector you would go to you when there is an interrupt on port 1. Here it is for port 2 and so on. So in your C program, you must use these exact names in capitals for, to enable you to link your interrupt subroutine for that particular peripheral. We will illustrate it with an example.

(Refer Slide Time: 23:41)

**What should be done in an ISR**

- Save system context (done automatically)
- Re-enable interrupts if required. ←
- The interrupt code ←
- If multi-source interrupt, then read associated register to determine source and clear the flag.
- Restore system context (done automatically)
- Return

SR + PC

The slide features a dark background with a curved orange and black graphic on the right side. A small video inset in the bottom right corner shows a man in a suit speaking at a podium with various logos, including 'MSP430'.

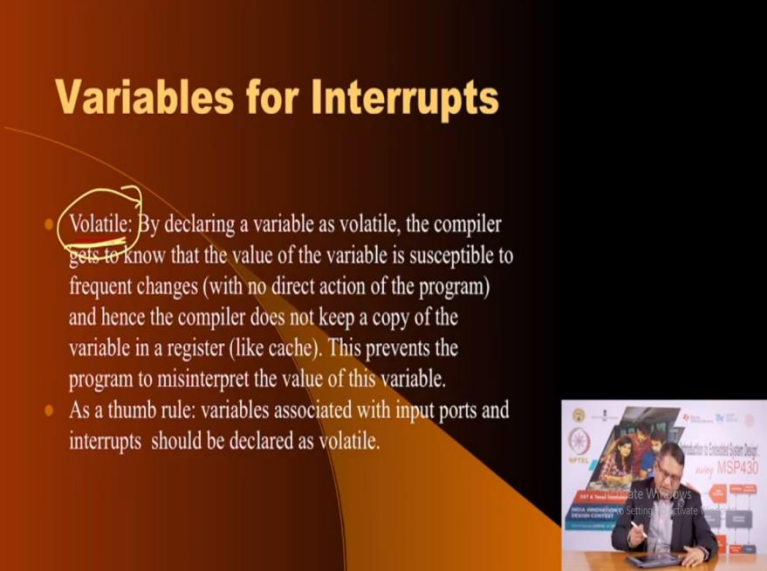
Now what do you do, what happens in an interrupt subroutine? Well the moment the main program is interrupted, you must save the context, and what do I mean by the context? Context is reflected by two things the status register plus the program counter. So both these registers have to be saved so that you can resume using the values here whatever program was interrupted.

So this must be saved and this is done automatically by the microcontroller, how? It saves these two registers on the stack. Now once you enter a interrupt subroutine, the global interrupt enable bit it is turned off, which means no interrupts can happen during the execution of this program. But you as a programmer and a designer are free to enable the interrupts which is what we are saying here. It is possible to enable interrupts within the interrupt subroutine and I would request you, I would advise you to exercise great caution in trying to re-enable interrupts within the interrupt subroutine.

Of course, what after that would come is the actual program that you would execute in the subroutine. Now, if the source of an interrupt at multiple sources, it is your responsibility that it is a responsibility of the program to reset the flag, so that subsequent setting of the flag will generate another interrupt and once that is done you can restore the system context and return meaning when you return, automatically the system will recover the stacks register, the status register and the program counter from the stack and put it into the respective registers, that is, the status register and the program counter which means you will go back to the main program

which had been interrupted and you will resume your execution from that point onwards. So this is the way things happen in an interrupt subroutine.

(Refer Slide Time: 25:40)



## Variables for Interrupts

- **Volatile:** By declaring a variable as volatile, the compiler gets to know that the value of the variable is susceptible to frequent changes (with no direct action of the program) and hence the compiler does not keep a copy of the variable in a register (like cache). This prevents the program to misinterpret the value of this variable.
- As a thumb rule: variables associated with input ports and interrupts should be declared as volatile.

The slide features a dark orange background with a white circle around the word 'Volatile' in the first bullet point. In the bottom right corner, there is a small inset photograph of a man in a suit sitting at a table, likely during a presentation or lecture.

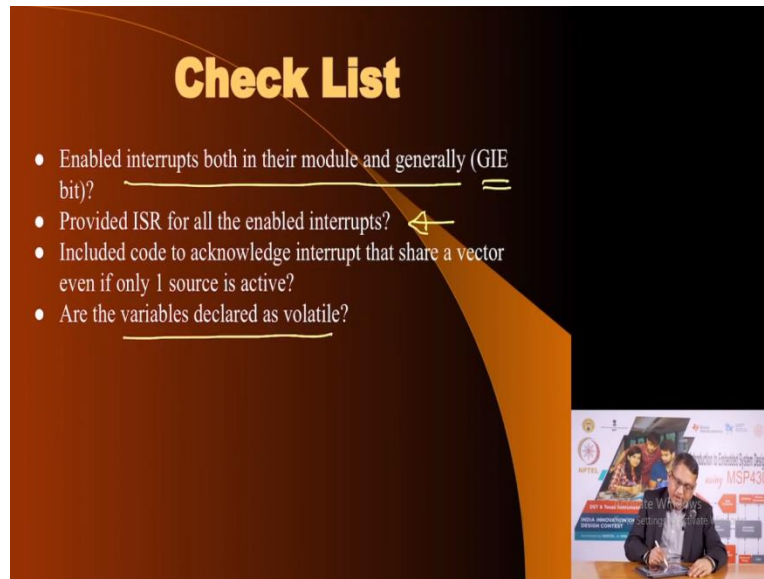
Now, as I mentioned how does an interrupt handle variables? How does it communicate with the main program? It is not that an interrupt happens, you go into interrupt subroutine, you do something there and you come back, no. Often times, you would want to convey to the main program that an interrupt happened and this is the input provided. And that is done by using variables, but these variables are not normal variables, they have to be declared as volatile, and volatile is a keyword, so you have to append this name 'volatile' in front of any variable.

What it does is often times the C compiler will keep a copy of variables in registers because reading anything from a register happens quickly but if the main variable is in the memory, you may not get its exact value. But by using this keyword, you are telling the compiler please do not use a copy of that value, go into the memory which is being reserved for that variable, fetch the value from there and then pass it.

And as a rule of thumb, all the variables associated with ports and interrupts should be declared as volatile. And we will see that example either in this or subsequent lectures.



(Refer Slide Time: 27:03)



So here is a checklist of things that you must perform in the main program so that interrupts can happen, if they happen, and if they happen they can be serviced. You must have enabled interrupts both in the module and the global interrupt enable bit. You must have interrupt subroutine for each of the enabled interrupts, you must acknowledge that interrupt meaning you must disable the flag within the interrupt subroutine.

And if you are using any variables, you have to ensure that they are declared as volatile. Now what we are going to do is, we are going to illustrate how you can experiment with interrupts on your MSP430 lunch box. So in the first exercise, we are going to connect the lunch box to our laptop and we will discuss this code, we will download this code on the lunchbox and using the peripheral on the lunch box itself, we will see how interrupts can be generated and how they can be serviced, that is how a interrupt subroutine can be executed in response to an interrupt be generated on the lunch box.

So take out your lunch box and connect it to your laptop and let us go to through the code, so that you understand how the code works and how you can modify it. So in the first exercise we are going to use the switch on the lunchbox as well as the LED on the lunch box to illustrate this interrupt being generated by the switch.

(Refer Slide Time: 28:45)

### HelloInterrupt

```
1#include <msp430.h>
2
3#define SW BIT3 // Switch -> P1.3 ← 08h
4#define RED BIT7 // Green LED -> P1.7 ← 80h
5
6/*@brief entry point for the code*/
7void main(void) {
8    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer ←
9
10    P1DIR |= RED; // Set LED pin -> Output
11    P1DIR &= ~SW; // Set SW pin -> Input
12    P1REN |= SW; // Enable Resistor for SW pin
13    P1OUT |= SW; // Select Pull Up for SW pin
14
15    P1IES &= ~SW; // Select Interrupt on Rising Edge
16    P1IE |= SW; // Enable Interrupt on SW pin
17
18    _bis_SR_register(GIE); // Enable CPU Interrupt
19
20    while(1);
21}
22
23/*@brief entry point for switch interrupt*/
```

Handwritten annotations on the slide include:

- A circuit diagram showing a pull-up resistor connected between VCC and pin P1.3.
- A bit mask diagram: `00001000` with bit 3 highlighted.
- Hex values `08h` and `80h` with arrows pointing to the corresponding bit definitions in the code.

So as you know, on the lunchbox, there is a switch connected to bit 3, that is P 1.3 you know that that is the a switch connected to this. You also know that this switch is connected in the pull up mode, meaning there is a resistor and a switch like this the switch is grounded on the other side. This side of the pull up resistor is connected to VCC and this junction here is connected to P 1.3 that is the meaning of pull up register.

And you also know that we have a LED connected on P 1.7. So we are going to use the available resources on the lunch box itself to illustrate this idea of an interrupt and how an event on the interrupt or interrupt been generated by this switch can be handled with a interrupt subroutine.

So we have to include, as you know, this header file in our C program. We have already defined that we have a switch which is on bit 3. Now, you see bit 3 is a mask, which means, what it actually is, is that is why I have 8 bits like this this is bit 0, 1 to 7, so this is 0, 1, 2, 3, so it means 1 0 0 0 0 0 0 0 that is the meaning of bit 3.

And we will use this to mask this particular bit or to do bit operations on this bit as and when required. Similarly red is a defined as a masked bit 7, that means, so bit 3 actually becomes 0 8 this value is 0 8 in X and bit 7 being 1 means it is 8 0 X, that is the meaning of bit 7. That is bit 7 is 1, rest bits are 0 - that is the meaning of these masks.

Now in our main program, as we always do, we are we have disabled the watchdog timer. And we have to now tell the microcontroller that we want to use red, we want to use LED on pin 1.7, therefore that pin must be declared as output. And we want to use is a switch on pin P 1.3 and that must be declared as input. And so this instruction, you are deciding the direction of the P 1.7 bit as output.

The P 1.3 pin is being declared as input. You are also enabling an internal pull-up resistor by using this REN register. And then you are writing a 1 into P1 out corresponding bit so that you are enabling the internal pull up resistor also. But beware, that there is also an external pull up resistor so, this instruction is superfluous but it does not hurt.

And then, you are saying that you want to you have to decide where do you, how do you want to generate the interrupt- do you want to generate the interrupt on falling edge for rising edge. Since I am in the interrupt enable register, I am, edge selection register, I am writing 0 that means I have, I have chosen to generate an interrupt on the rising edge. And then, on the interrupt enable register I make that corresponding bit that is bit 3 equal to 1, that means I have enabled the interrupt to happen on this pin. And then this instruction is used to turn the global interrupt enable bit to 1, now I am ready.

(Refer Slide Time: 32:56)

## Interrupt Registers for GPIO

- All these registers are 8-bit registers. ←
- P<sub>x</sub>IE - Interrupt enable register ← P<sub>1</sub>IE
- P<sub>x</sub>IES - Interrupt edge selection register ←
- P<sub>x</sub>IFG - Interrupt flag register ←

**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: \_int\_SR\_register(GIE);  
Disable: \_int\_SR\_register(GIE);

If I go back to that block diagram, basically what I have done is, here using that instruction I have turned this on and the earlier instruction I have enabled the particular bit. Now we are ready to do something in the main program.

(Refer Slide Time: 33:15)

```

8  WDTCTL = WDTPW | WDTHOLD; // STOP WATCHDOG TIMER
9
10 P1DIR |= RED; // Set LED pin -> Output
11 P1DIR &= ~SW; // Set SW pin -> Input
12 P1REN |= SW; // Enable Resistor for SW pin
13 P1OUT |= SW; // Select Pull Up for SW pin
14
15 P1IES &= ~SW; // Select Interrupt on Rising Edge
16 P1IE |= SW; // Enable Interrupt on SW pin
17
18 _bis_SR_register(GIE); // Enable CPU Interrupt
19
20 while(1);
21
22
23 /*Brief entry point for switch interrupt*/
24 #pragma vector=PORT1_VECTOR
25 __interrupt void Port_1(void)
26 {
27     if(P1IFG & SW) // If SW is Pressed
28     {
29         P1OUT ^= RED; // Toggle RED LED
30         volatile unsigned long i;
31         for(i = 0; i<10000; i++); //delay
32         P1IFG &= ~SW; // Clear SW interrupt flag
33     }
34 }

```

But in the main program in this particular case, there is nothing to do and so we will simply, as you see here, we are going to wait here in a infinite loop while one. So it's, your main program is doing nothing, waiting for something to happen and in this case that something is an external interrupt which will be generated because you press the switch on pin 1.3 port 1.3.

When that happens, before that, see you have declared that you are done #pragma and then vector you have said relates to this vector, port 1 vector and you have named for subroutine port\_1. And this is the interrupt subroutine. So and this is the entire interrupt subroutine for interrupt been generated on port 1.


What happens the moment you press the switch and when you press a switch, because there is a, the switch is already pulled up, when you press a switch, you get a high to low transition. And as you see here, you have selected an interrupt for the rising edge. So what is going to happen is when you press the switch it is going to go low, this is not going to generate an interrupt. But after a while when you release the switch, this act of releasing the switch will generate and interrupt.

(Refer Slide Time: 34:36)

### Example Code 1(Bad ISR): HelloInterrupt



```
#include <msp430.h>
#define SW BIT3 // Switch -> P1.3
#define RED BIT7 // Green LED -> P1.7
// Brief entry point for the code
void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
    P1DIR |= RED; // Set LED pin -> Output
    P1DIR &= ~SW; // Set SW pin -> Input
    P1REN |= SW; // Enable Resistor for SW pin
    P1OUT |= SW; // Select Pull Up for SW pin
    P1IES &= ~SW; // Select Interrupt on Rising Edge
    P1IE |= SW; // Enable Interrupt on SW pin
    _bis_SR_register(GIE); // Enable CPU Interrupt
    while(1); // Infinite loop
}
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void) {
    if(P1IFG & SW) // If SW is Pressed
    {
        P1OUT ^= RED; // Toggle RED LED
        volatile unsigned long i;
        for(i = 0; i < 100000; i++); //delay
        P1IFG &= ~SW; // Clear SW Interrupt flag
    }
}
```

*Handwritten notes:* 08h, 90h, 7, 00001010, 101010



Now, please note here that we are calling this program a bad ISR, meaning this is not a very good style of writing a interrupt subroutine, why? As I mentioned the interrupt subroutine must be kept short, and as you see here, there is we have written a delay program. As you see here, we have written a small delay program. This is not a good way of writing and interrupt but I will explain why this is needed.

(Refer Slide Time: 35:07)



Now when the switch is pressed, it is going to go low but as we know, when a switch is pressed often times it may bounce. So the act of pressing a switch may generate this. So you see multiple rising edge can happen and then it is going to be held low for as long as you keep the switch pressed and when you released again multiple events are likely to happen, multiple rising edge can happen.

And the duration of these bounces is such that the microcontroller will treat each of these bounce as an individual interrupt, even though you press which only once, it may interpret it that you have pressed the switch several times. And therefore to remove that ambiguity, at this point you have waited for these bounces to die of.

But what about the bouncing that may happen when you press the switch? Unfortunately, in our program we have not dealt with it and so it is possible, anyway let us see what the program does, when it this checks if really the interrupt happened because you pressed the switch.

(Refer Slide Time: 36:19)

```

8  MWCTL = MWPM | MWHLUD; // Stop watchdog timer
9
10 PIDIR |= RED; // Set LED pin -> Output
11 PIDIR &= ~SW; // Set SW pin -> Input
12 PIREN |= SW; // Enable Resistor for SW pin
13 P1OUT |= SW; // Select Pull Up for SW pin
14
15 P1IES &= ~SW; // Select Interrupt on Rising Edge
16 P1IE |= SW; // Enable Interrupt on SW pin
17
18 _bis_SR_register(GIE); // Enable CPU Interrupt
19
20 while(1);
21
22
23 /*@brief entry point for switch interrupt*/
24 #pragma vector=PORT1_VECTOR
25 __interrupt void Port_1(void)
26 {
27     if(P1IFG & SW) // If SW is Pressed
28     {
29         P1OUT ^= RED; // Toggle RED LED
30         volatile unsigned long i;
31         for(i = 0; i<10000; i++); //delay
32         P1IFG &= ~SW; // Clear SW interrupt flag
33     }
34 }

```

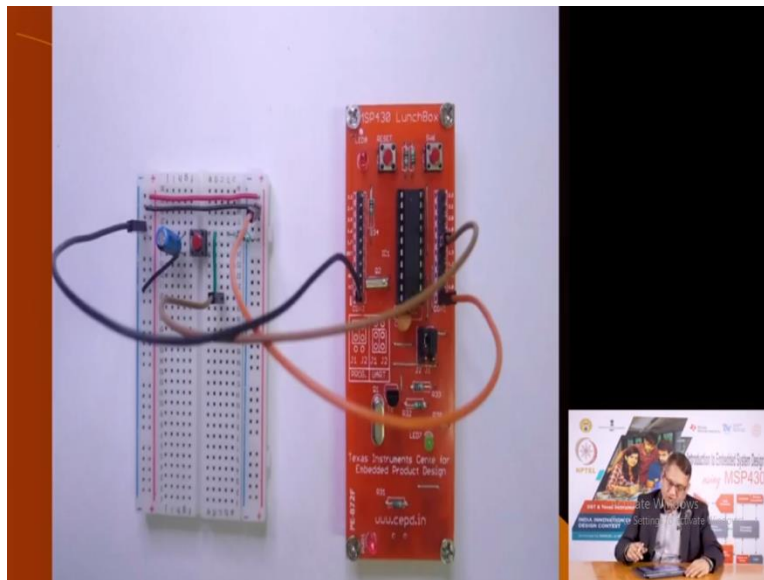
The image shows the code with several handwritten annotations in red. On the left, there are arrows pointing to lines 10-13 and 15-16. A large red bracket on the left side encompasses the entire code block. In the center, a bit diagram shows a byte with bits 0-7, where bit 7 is highlighted with a red circle and an arrow pointing to line 27. On the right, there is a small inset image of a person speaking, likely from a video lecture.

So P1 IFG, is it 1? Isolate that bit using this & operation. This is superfluous because since you have disabled the rest of the interrupts, this interrupt would have happened only because you pressed the switch but it is a good idea. When that happens, and when will this happen? When you press the switch and release it or if while pressing there is a switch bounce and that also generates a rising edge signal you immediately toggle the LED on port 1 bit 7, then you delay

waiting for the bounces to die down and then you reset that flag so that, next time when the user presses a switch it can generate t0, it can generate an interrupt and it can execute this program. And why do we call this example of interrupt subroutine as not a good example? It is because we have used a delay subroutine; we do not recommend using delay subroutine in the interrupt program.

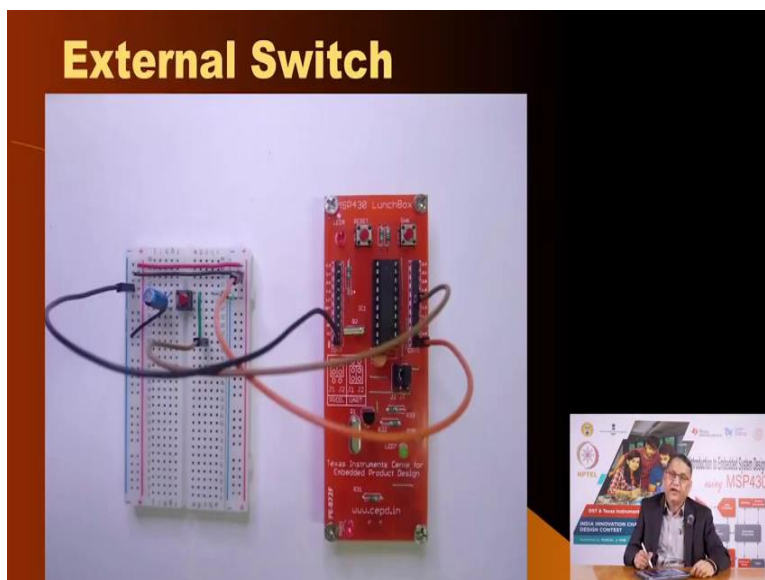
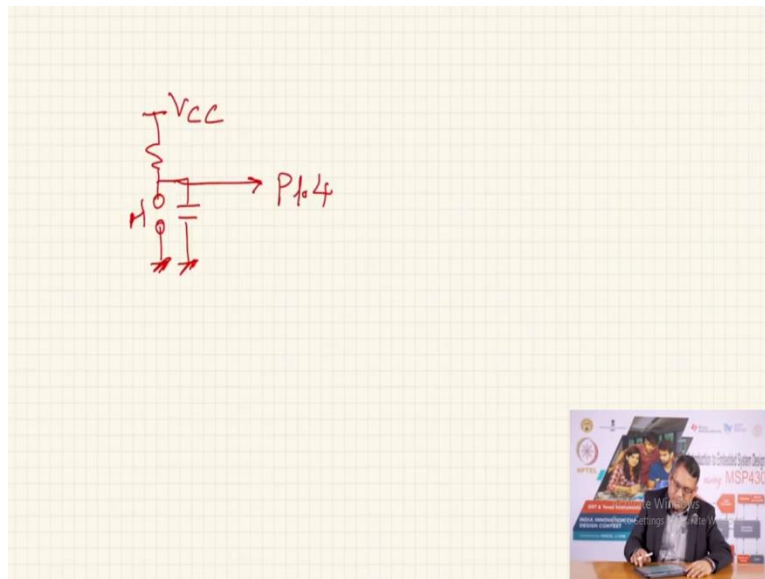
Now, how do we deal with this? Well, make sure that you don't have to de-bounce your switch. Now, if this which is read using polling, this is a good method. But reading a switch or de-bouncing the switch using delay in an interrupt is a complete no-no. So what we do is, we look at another example.

(Refer Slide Time: 37:51)



In this example, we are going to use an external switch and this is the wiring that you should do connect an external switch and de-bounce using hardware. As we have discussed in the past, one way to de-bounce in hardware is by putting a capacitor across the switch. This capacitor acts as a low pass filter and kills any switch bounce.

(Refer Slide Time: 38:17)




So the way this would look like is, here you have a pull-up resistor, here you are you have a switch and across this which we have put a capacitor, and then this signal goes to a pin which I will briefly mention what it is. In this case it is P 1.4. So you need to wire a switch in this configuration and connect it to Port 1 bit 4, here 1 bit 4.



(Refer Slide Time: 38:41)

### Example Code 2(Better ISR): HelloInterrupt\_Rising

```
1#include <msp430.h>
2
3#define SW BIT4 // Switch -> P1.4
4#define RED BIT7 // Green LED -> P1.7
5
6/*Brief entry point for the code*/
7void main(void) {
8    WDCTL = WDTPW | WDTHOLD; // Stop watchdog timer
9
10    P1DIR |= RED; // Set LED pin -> Output
11    P1DIR &= ~SW; // Set SW pin -> Input
12    P1REN |= SW; // Enable Resistor for SW pin
13    P1OUT |= SW; // Select Pull up for SW pin
14
15    P1IES &= ~SW; // Select Interrupt on Rising Edge
16    P1IE |= SW; // Enable Interrupt on SW pin
17
18    __bis_SR_register(GIE); // Enable CPU Interrupt
19
20    while(1);
21}
22
23/*Brief entry point for switch interrupt*/
24#pragma vector=PORT1_VECTOR
25__interrupt void Port_1(void)
26{
27    if(P1IFG & SW) // If SW is Pressed
28    {
29        P1OUT ^= RED; // Toggle RED LED
30        P1IFG &= ~SW; // Clear SW interrupt flag
31    }
32}
```



You are dedicated, you are dedicating for a new switch which you connect on the breadboard. We will continue to use the onboard LED on 1.7. Now you see, we have all the same, we want a rising edge interrupt, which means if I press the switch it is going to go low, that is not going to generate an interrupt. When I release the switch, eventually it will generate an interrupt. And in this particular case, I have ensured by using external hardware mechanism to de-bounce, that I am going to get a clean switch transition from high to low, remaining low for as long as you hold the switch down and when you release it you get something like this.

(Refer Slide Time: 39:35)

### Example Code 1(Bad ISR): HelloInterrupt

```
1 #include <msp430.h>
2
3 #define SW BIT3 // Switch -> P1.3
4 #define RED LED // Green LED -> P1.7
5
6 /*Brief entry point for the code*/
7 void main(void) {
8     WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
9
10    P1DIR |= RED; // Set LED pin -> Output
11    P1DIR &= ~SW; // Set SW pin -> Input
12    P1REN |= SW; // Enable Resistor for SW pin
13    P1OUT |= SW; // Select Pull up for SW pin
14
15    P1IES &= ~SW; // Select Interrupt on Rising Edge
16    P1IE |= SW; // Enable Interrupt on SW pin
17
18    _bis_SR_register(GI1); // Enable CPU Interrupt
19
20    while(1);
21
22
23 /*Brief entry point for switch interrupt*/
24 #pragma vector=PORT1_VECTOR
25 __interrupt void Port_1(void) {
26
27     if(P1IFG & SW) // If SW is Pressed
28     {
29         P1OUT ^= RED; // Toggle RED LED
30         volatile unsigned long i;
31         for(i = 0; i < 100000; i++) //delay
32             P1IFG &= ~SW; // Clear SW interrupt flag
33     }
34 }
```

The slide includes several annotations: red arrows pointing to specific lines of code, a red box around the `while(1);` line, and a red diagram of a 7-bit register with bits 0-6 set to 0 and bit 7 set to 1. A small inset image shows a person at a desk with a computer monitor displaying a presentation slide about MSP430.

And so when you download this program, what you let me go back to the first program. When you compile, rebuild this program and download, what you will see is when you press the switch if there is no bounce you will not see that toggle off the LED. But when you release it, you should notice that the LED toggles.

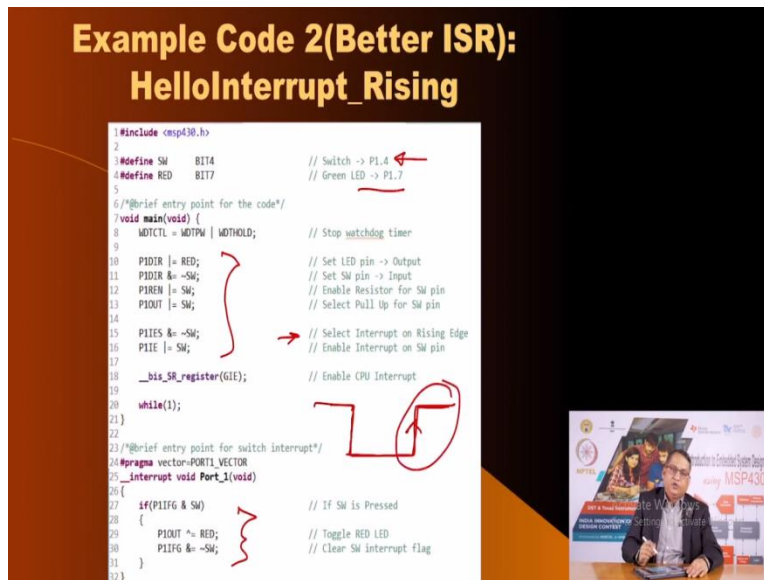
In case when you press the switch the LED toggles, and when you release also it toggles, maybe it's toggle multiple times, you may think that the program is not working the way it should and the explanation of that would be that in your kit it so happens that the switch press and release is generating switch bounce which we are not able to deal with, at least at the time when you press the switch. For the release, we have taken care of by including a delay. And so while releasing there will not be multiple toggling but while pressing it is possible that it toggles when it should not.

On the other hand, when you wire this external switch there is not going to be any bouncing, why? Because we have killed that bouncing with the help of a capacitor.

(Refer Slide Time: 40:35)

### Example Code 2(Better ISR): HelloInterrupt\_Rising

```
1#include msp430.h
2
3#define SW BIT4 // Switch -> P1.4
4#define RED BIT7 // Green LED -> P1.7
5
6/**Brief entry point for the code*/
7void main(void) {
8    WDCTL = WDTPW | WDTHOLD; // Stop watchdog timer
9
10    P1DIR |= RED; // Set LED pin -> Output
11    P1DIR &= ~SW; // Set SW pin -> Input
12    P1REN |= SW; // Enable Resistor for SW pin
13    P1OUT |= SW; // Select Pull Up for SW pin
14
15    P1IES &= ~SW; // Select Interrupt on Rising Edge
16    P1IE |= SW; // Enable Interrupt on SW pin
17
18    __bis_SR_register(GIE); // Enable CPU Interrupt
19
20    while(1);
21}
22
23/**Brief entry point for switch interrupt*/
24#pragma vector=PORT1_VECTOR
25__interrupt void Port_1(void)
26{
27    if(P1IFG & SW) // If SW is Pressed
28    {
29        P1OUT ^= RED; // Toggle RED LED
30        P1IFG &= ~SW; // Clear SW interrupt flag
31    }
32}
```



Here the example is very similar, but there is no delay subroutine here. And rest of the code is very similar you are enabling the interrupts, you are choosing that the interact should happen on the falling edge, on the rising edge, which means when you press the switch and you release it. at the time the LED will toggle.

So please compile this program, download it in the kit, connect an external switch appropriately and what you should observe is when you press the switch nothing happens when you release the switch the LED will toggle, meaning if it was on earlier, after the switch is pressed and released, it will turn off. When you, after it is of you press the switch and release it will turn on that is the meaning of this.

So the interrupt subroutine is a very simple piece of code which just toggles the LED, no delay instructions included in the interrupt subroutine and this instruction will reset that inter flag so that next time the switch is pressed it will be entertained.

(Refer Slide Time: 41:43)

### Example Code 3: HelloInterrupt\_Falling

```
1 #include <msp430.h>
2
3 #define SW BIT4 // Switch -> P1.4
4 #define RED BIT7 // Green LED -> P1.7
5
6 /*Brief entry point for the code*/
7 void main(void) {
8     MDCTL = MDTPW | MOTHOLD; // Stop watchdog timer
9
10    P1DIR |= RED; // Set LED pin -> Output
11    P1DIR &= ~SW; // Set SW pin -> Input
12    P1REN |= SW; // Enable Resistor for SW pin
13    P1OUT |= SW; // Select Pull-up for SW pin
14
15    P1IES |= SW; // Select Interrupt on Falling Edge
16    P1IE |= SW; // Enable Interrupt on SW pin
17
18    __bis_SR_register(GIE); // Enable GIE Interrupt
19
20    while(1);
21
22
23 /*Brief entry point for switch interrupt*/
24 #pragma vector=PORT1_VECTOR
25 __interrupt void Port_1(void)
26 {
27     if(P1IFG & SW) // If SW is Pressed
28     {
29         P1OUT ^= RED; // Toggle RED LED
30         P1IFG &= ~SW; // Clear SW interrupt flag
31     }
32 }
33
```

If you want to convert this program - that you want to toggle the LED when the switch is pressed instead of toggling when you release the switch you just simply modify this part here that you want the interrupt to be generated on the falling edge, which means when you press the switch like this and release, at this point it will toggle the LED.

Rest of the instructions as you see are the same. Here you are enabling the global interrupts and you are waiting in this infinite while loop and the only way to get out of this will be when the interrupt is generated, you will execute this subroutine, you will perform the task of toggling the LED and you will go back to waiting in that while (1) loop indefinitely.


Now we have a small exercise and this would, this exercise will actually help you understand that when the interrupt source is common, how do you handle that?

(Refer Slide Time: 42:43)

## Exercise

Modify the existing code to toggle LED1 when SW1 is pressed and to toggle LED2 when SW2 is pressed.

<u>SW1</u> →	<u>P1.0</u>		<u>LED1</u> →	<u>P1.4</u>
<u>SW2</u> →	<u>P1.1</u>		<u>LED2</u> →	<u>P1.5</u>



So we have an exercise where we would want you to connect two external switches and two external LEDs on the breadboard in the following way. Connect switch 1 to port 1 bit 0, switch 2 to port 1 bit 1. Corresponding to switch 1, we wanted to connect a LED on port 1 bit 4 and another LED for switch 2 for this.

And what do you want you to do? That when switch 1 is pressed and released, it should toggle the LED 1. When switch 2 is pressed, it should toggle LED 2. You can decide whether you want the toggling to happen when the switch is pressed or when it is released. But if switch 1 is pressed, it should not toggle LED 2, it should only toggle LED 1. If switch 2 is pressed, it should not toggle any other LED, but LED 2.

And the way to handle that would be that in the interrupt subroutine, you will have to find out what caused the interrupt. Both these interrupts are going to be routed through the port 1 interrupt. But the source is different. One, in one case, it is switch 1, in the other case it is switch 2. So in your interrupt subroutine, you have to identify which caused the interrupt and based on that, take necessary action. If switch 1 was pressed, you should toggle LED 1. If switch 2 was pressed, you should toggle LED 2. And then correspondingly, clear that flag so that, that switch can again generate an interrupt.

So we would like you to write a program and test it, and satisfy yourself that this is how interrupt subroutines are written and handled. Of course, we will have more examples where we illustrate how interrupts work using other peripherals of MSP430 lunch box. So I will see you very soon in a new video. Thank you.