

Lecture 10: Interrupts on the MSPM0

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers



Lesson Objectives

- Understand basic concepts of interrupts
- Be able to design a program that uses:
 - Polling
 - Interrupts
- Be able to configure interrupts, write ISR's, and set interrupt priorities
- Become familiar with GPIO edge-triggered interrupts



Interrupts: Background

- Whole purpose of MCU devices is to respond to external stimuli by controlling mechanical device
- Question arises: How do we program device to monitor and/or respond to external stimulus?
- Examples:
 - Run a motor every time someone holds a button down
 - Each time a sensor signal goes high, sound an alarm
 - Turn a stepper motor until a limit switch activates
 - Each time a GPS signal is received, log data to flash



Interrupts: Background

- One mechanism to do this is called polling or busy-wait synchronization
- Idea: Continuously check a variable in a loop to see if it has changed
 - Combine with if statement to take action if it has
- Example:
 - Loop until user presses button
 - If so, do something
 - Then wait until user releases button
 - [see next slide for code]



Polling Example

```
while(1){  
  
    // Read button  
    usiButton1 = DL_GPIO_readPins(GPIOA, DL_GPIO_PIN_0) ;  
  
    // If button is pressed...  
    if ( usiButton1 == 0 ) {  
  
        DL_GPIO_setPins(GPIOA, DL_GPIO_PIN_1);  
  
        // Wait until button is released  
        while(DL_GPIO_readPins(GPIOA, DL_GPIO_PIN_0) == 0){}  
  
        DL_GPIO_clearPins(GPIOA, DL_GPIO_PIN_1);  
  
    }  
}
```

Interrupts: Background

- Polling is effective way to program if the I/O task you are performing is simple and predictable
- What are the drawbacks of polling?



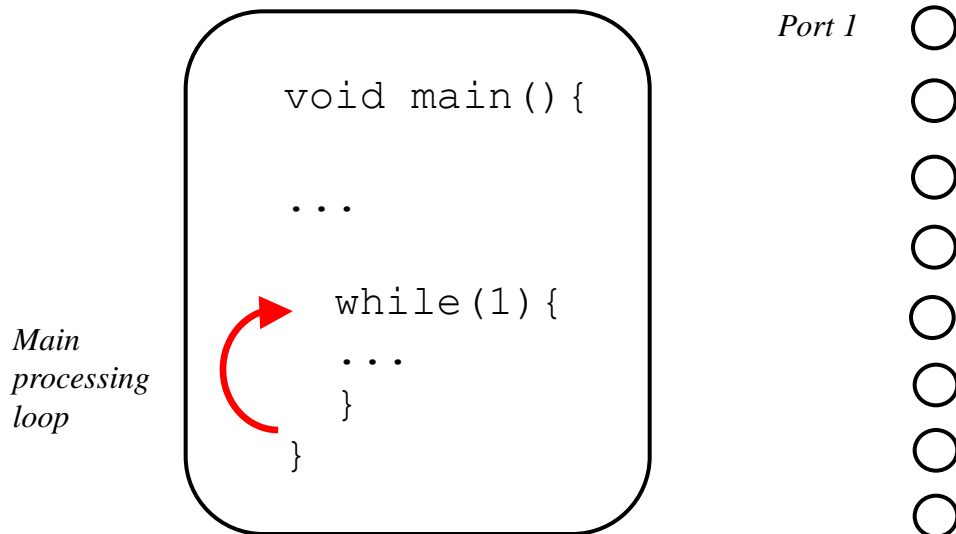
Interrupts: Background

- Polling is effective way to program if the I/O task you are performing is simple and predictable
- What are the drawbacks of polling?
 - Software is tied up checking a certain address and cannot do other tasks (inefficient computation)
 - If we are waiting on multiple possible events, cumbersome to establish priority
 - If software is waiting for long periods, extremely inefficient use of power (inefficient power consumption)
 - *Ideally, would like to be able to put MCU in low power mode while waiting*



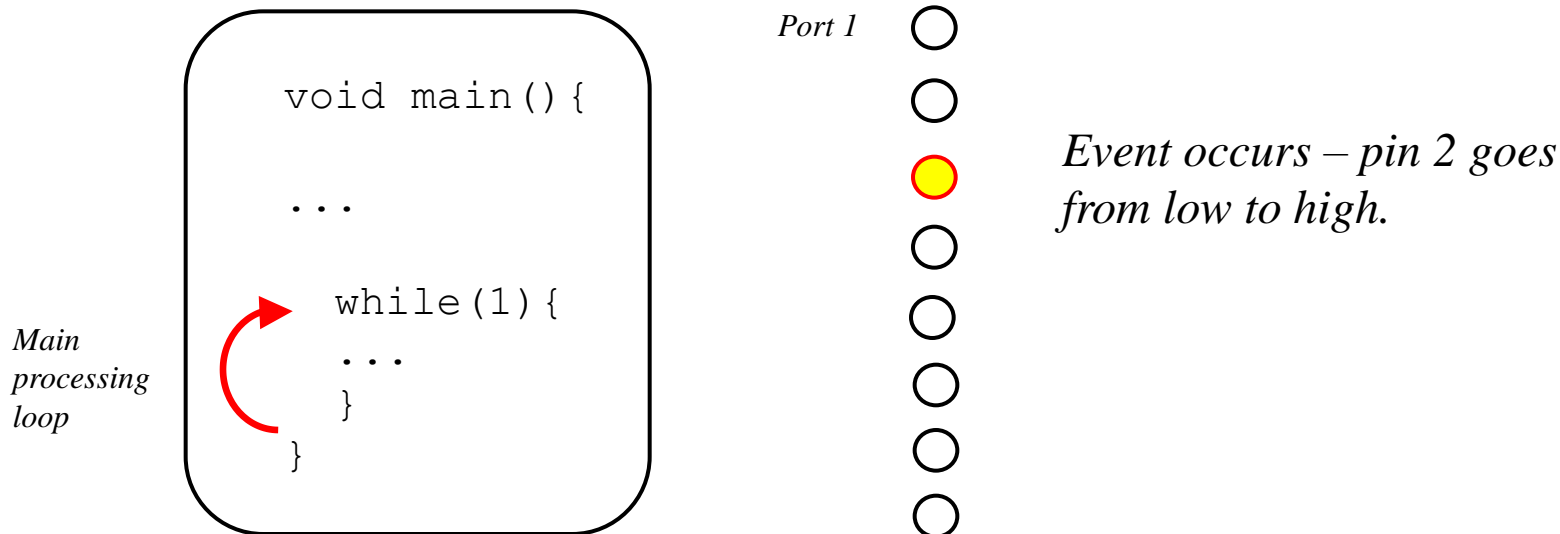
Interrupts

- An interrupt is an *asynchronous* switch in processor execution
 - Asynchronous means that it occurs on demand, not with some specific timing (like polling does)
- Interrupt execution:



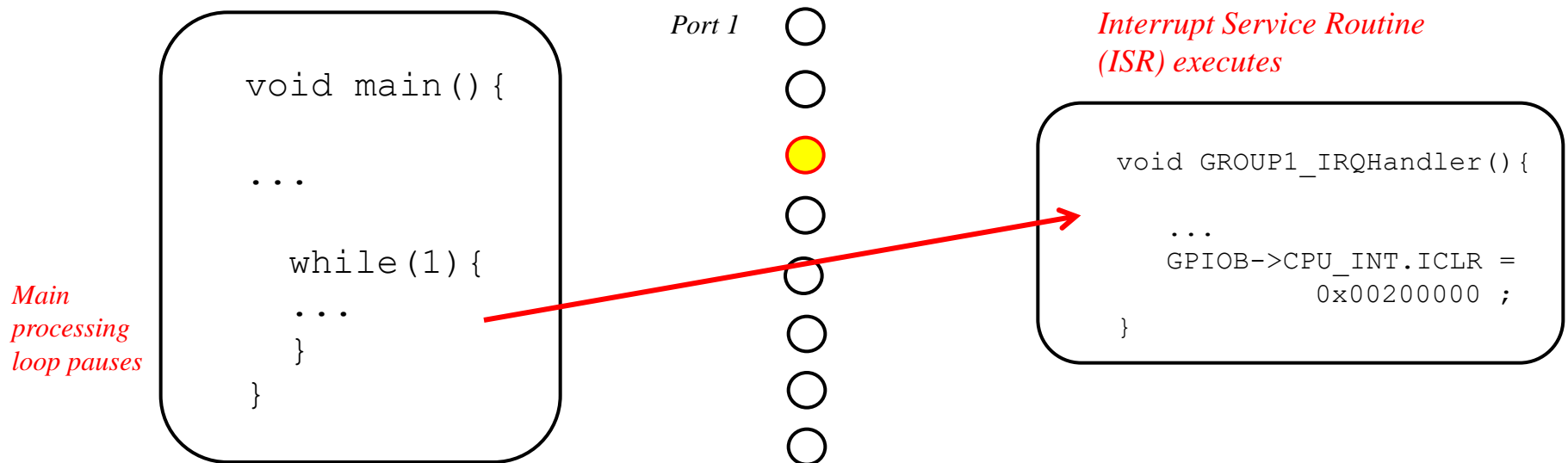
Interrupts

- An interrupt is an *asynchronous* switch in processor execution
 - Asynchronous means that it occurs on demand, not with some specific timing (like polling)
- Interrupt execution:



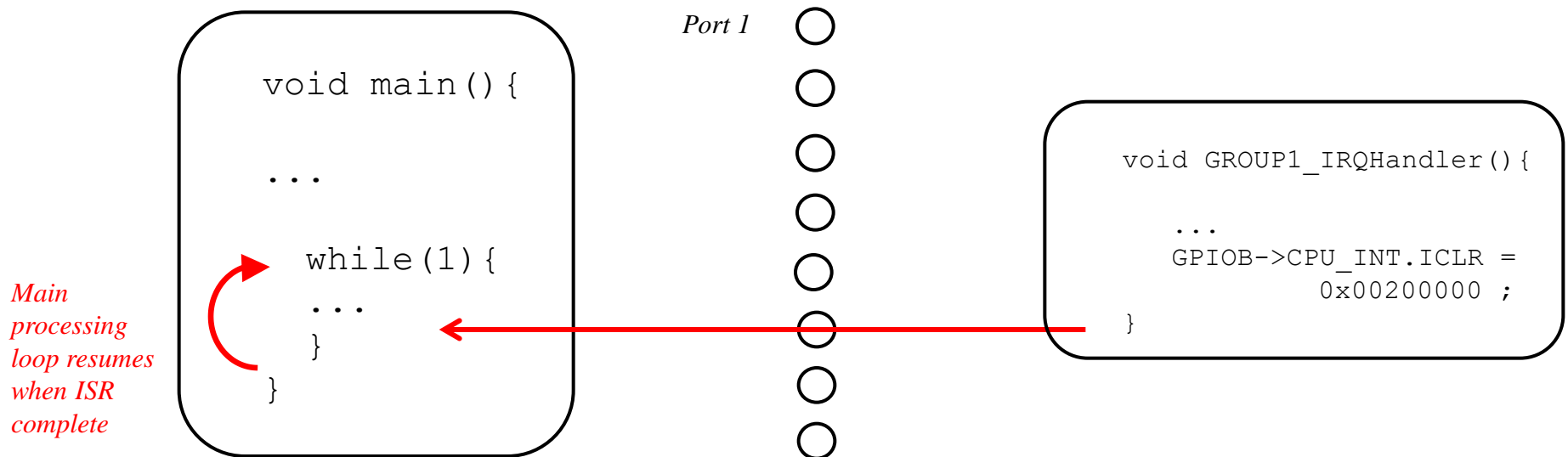
Interrupts

- An interrupt is an *asynchronous* switch in processor execution
 - Asynchronous means that it occurs on demand, not with some specific timing (like polling)
- Interrupt execution:



Interrupts

- An interrupt is an *asynchronous* switch in processor execution
 - Asynchronous means that it occurs on demand, not with some specific timing (like polling)
- Interrupt execution:



Interrupts

- Most hardware features on the MCU can generate interrupts and can have an associated ISR
 - Analog-to-digital converter
 - Flash memory controller
 - Timers
 - UARTs
 - GPIO ports
 - I2C
 - Serial Peripheral Interface
 - etc...



Interrupts

- By default, if you do not explicitly enable an interrupt, code executes serially and no interrupts will occur
- Setting up an interrupt and defining appropriate ISR to run when interrupt occurs is your responsibility
 - There are no “default” ISR’s programmed into MCU
- Deciding when to use interrupts, and when to use polling, is your choice as software developer
 - Use polling when I/O structure is simple and fixed
 - Use interrupts when I/O timing is variable and/or structure is complex

Interrupt Priority

- Potentially, your code may have several interrupts enabled at one time
 - i.e., you have one interrupt for ADC and one interrupt for edge-triggered GPIO
 - What happens if both events happen at same time? Which ISR runs?
 - *Or, if GPIO event happens when ADC ISR is executing?*
- This is why you must define interrupt priority when enabling an interrupt



Interrupt Priority

- You can set interrupt priority by assigning a value between 0 and 4
 - 0 is highest priority, 4 is lowest
- Priority rules:
 - If an ISR of higher priority is running and lower priority interrupt is triggered, ISR of lower priority will wait until higher one finishes and run right afterward
 - If an ISR of lower priority is running and higher priority interrupt is triggered, ISR of higher priority takes over, runs to completion, and returns execution to lower priority ISR



Interrupt Setup

- To setup an interrupt, two levels of interrupt setup must occur

*Interrupt
controller
configuration*

Nested Vector Interrupt Controller (NVIC)

- Stores interrupt priorities
- Stores which peripherals have interrupt enabled
- Actually responsible for interrupting processor

GPIO Interrupt Config

- GPIOA->POLARITY31_16 register
- GPIOA->CPU_INT.IMASK register
- etc.

ADC Interrupt Config

- ADC0->ULLMEM.CPU_INT.IMASK register
- ADC->ULLMEM.CPU_INT.ICLR register
- etc.

UART Interrupt Config

- UART0_INST->CPU_INT.IMASK register
- UART0_INST->CPU_INT.ICLR register
- etc.

*Peripheral-
specific
configurations*
























Interrupt Controller

- Nested Vector Interrupt Controller (NVIC) is responsible for:
 - Notifying processor when interrupt execution has occurred
 - Storing interrupt priorities and deconflicting simultaneous interrupts
 - Storing memory locations of interrupt service routines
- Two primary NVIC registers you need to know about:
 - NVIC->IP[x]
 - NVIC->ISER[0] and NVIC->ICER[0]



NVIC Priority Register (NVIC->IP)

- Interrupt priority is stored in the NVIC->IP[x] registers

ISR name	NVIC priority	Priority bits	NVIC enable	Enable bit
PendSV_Handler	SCB->SHP[1]	23  22	--	--
SysTick_Handler	SCB->SHP[1]	31  30	--	--
GROUP0_IRQHandler	NVIC->IP[0]	7  6	NVIC->ISER[0]	0
GROUP1_IRQHandler	NVIC->IP[0]	15  14	NVIC->ISER[0]	1
TIMG8_IRQHandler	NVIC->IP[0]	23  22	NVIC->ISER[0]	2
UART3_IRQHandler	NVIC->IP[0]	31  30	NVIC->ISER[0]	3
ADC0_IRQHandler	NVIC->IP[1]	7  6	NVIC->ISER[0]	4
ADC1_IRQHandler	NVIC->IP[1]	15  14	NVIC->ISER[0]	5
CANFD0_IRQHandler	NVIC->IP[1]	23  22	NVIC->ISER[0]	6
DAC0_IRQHandler	NVIC->IP[1]	31  30	NVIC->ISER[0]	7
SPI0_IRQHandler	NVIC->IP[2]	15  14	NVIC->ISER[0]	9
SPI1_IRQHandler	NVIC->IP[2]	23  22	NVIC->ISER[0]	10
UART1_IRQHandler	NVIC->IP[3]	15  14	NVIC->ISER[0]	13
UART2_IRQHandler	NVIC->IP[3]	23  22	NVIC->ISER[0]	14
UART0_IRQHandler	NVIC->IP[3]	31  30	NVIC->ISER[0]	15
TIMG0_IRQHandler	NVIC->IP[4]	7  6	NVIC->ISER[0]	16
TIMG6_IRQHandler	NVIC->IP[4]	15  14	NVIC->ISER[0]	17
TIMA0_IRQHandler	NVIC->IP[4]	23  22	NVIC->ISER[0]	18
TIMA1_IRQHandler	NVIC->IP[4]	31  30	NVIC->ISER[0]	19
TIMG7_IRQHandler	NVIC->IP[5]	7  6	NVIC->ISER[0]	20
TIMG12_IRQHandler	NVIC->IP[5]	15  14	NVIC->ISER[0]	21

Each interrupt has its own 2-bit slot that can store a number between 0 and 4

Table 5.3.2. The MSPM0 NVIC registers. Each register is 32 bits wide. Bits not shown are zero.

NVIC Priority Register (NVIC->IP)

- Example: Set interrupt priority for GROUP1 to level 2

```
NVIC->IP[0] = 0x00008000 ; // Write 1 to bit 15
```

- Example: Set interrupt priority for ADC0 to 0

```
NVIC->IP[1] = 0x00000000; // Clear bits 6-7 of register
```



NVIC Enable/Disable Registers

- To enable interrupt, set appropriate bit in NVIC->ISER[0] or NVIC->ISER[1] registers
 - Clearing bits has no effect in this register
- To disable interrupt, set appropriate bit in NVIC->ICER[0] or NVIC->ICER[1] registers
 - Clearing bits has no effect in this register



NVIC Enable/Disable Registers






















ISR name	NVIC priority	Priority bits	NVIC enable	Enable bit
PendSV_Handler	SCB->SHP[1]	23  22	--	--
SysTick_Handler	SCB->SHP[1]	31  30	--	--
GROUP0_IRQHandler	NVIC->IP[0]	7  6	NVIC->ISER[0]	0
GROUP1_IRQHandler	NVIC->IP[0]	15  14	NVIC->ISER[0]	1
TIMG8_IRQHandler	NVIC->IP[0]	23  22	NVIC->ISER[0]	2
UART3_IRQHandler	NVIC->IP[0]	31  30	NVIC->ISER[0]	3
ADC0_IRQHandler	NVIC->IP[1]	7  6	NVIC->ISER[0]	4
ADC1_IRQHandler	NVIC->IP[1]	15  14	NVIC->ISER[0]	5
CANFD0_IRQHandler	NVIC->IP[1]	23  22	NVIC->ISER[0]	6
DAC0_IRQHandler	NVIC->IP[1]	31  30	NVIC->ISER[0]	7
SPI0_IRQHandler	NVIC->IP[2]	15  14	NVIC->ISER[0]	9
SPI1_IRQHandler	NVIC->IP[2]	23  22	NVIC->ISER[0]	10
UART1_IRQHandler	NVIC->IP[3]	15  14	NVIC->ISER[0]	13
UART2_IRQHandler	NVIC->IP[3]	23  22	NVIC->ISER[0]	14
UART0_IRQHandler	NVIC->IP[3]	31  30	NVIC->ISER[0]	15
TIMG0_IRQHandler	NVIC->IP[4]	7  6	NVIC->ISER[0]	16
TIMG6_IRQHandler	NVIC->IP[4]	15  14	NVIC->ISER[0]	17
TIMA0_IRQHandler	NVIC->IP[4]	23  22	NVIC->ISER[0]	18
TIMA1_IRQHandler	NVIC->IP[4]	31  30	NVIC->ISER[0]	19
TIMG7_IRQHandler	NVIC->IP[5]	7  6	NVIC->ISER[0]	20
TIMG12_IRQHandler	NVIC->IP[5]	15  14	NVIC->ISER[0]	21

Table 5.3.2. The MSPM0 NVIC registers. Each register is 32 bits wide. Bits not shown are zero.

Example: Enable
UART0 interrupt

```
NVIC->ISER[0] |= 0x00008000 ;           // Set bit 15 of register
```

Note: During startup, bits in ISER registers are all cleared and bits in ICER registers are all set.

NVIC Control: Driverlib

- Easiest way to configure NVIC is using driverlib functions
 - Recall that driverlib functions are just same register bit manipulation, wrapped in more readable function
- Setting interrupt priority:

```
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
```

Note: This sets bits in NVIC->IP[x].

Examples:

```
// Set Timer 0 interrupt to priority 3
NVIC_SetPriority (TIMER_0_INST_INT_IRQN, 3 ) ;

// Set ADC0 interrupt to priority 2
NVIC_SetPriority (ADC12_0_INST_INT_IRQN, 2) ;
```

NVIC Control: Driverlib

- Enabling and disabling interrupts:

```
void NVIC_EnableIRQ ( IRQn_Type IRQn )  
  
void NVIC_DisableIRQ ( IRQn_Type IRQn )
```

Note: This sets bits in NVIC->ISER or NVIC->ICER.

Examples:

```
// Enable GPIOA interrupt  
NVIC_EnableIRQ( GPIOA_INT_IRQn ) ;  
  
// Enable UART 0 interrupt  
NVIC_EnableIRQ( UART0_INT_IRQn ) ;
```

- See driverlib documentation for detailed explanation of these and similar functions



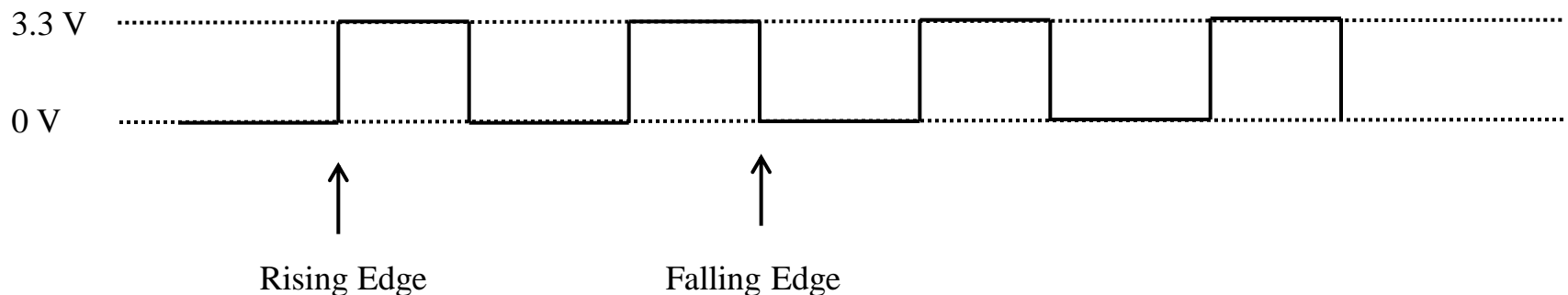
GPIO Interrupt Setup

- We now know how to set up global interrupt structure
- How do we set up interrupts for individual peripheral devices?
- Will discuss GPIO today, other peripherals in later classes
 - Each peripheral interrupt is similar but may have unique features



GPIO Interrupt Setup

- GPIO interrupts occur either on high-to-low transition on input pin (falling edge) or low-to-high transition on input pin (rising edge)



- Usually edge triggering is used when signal transitions quickly, but gradually-transitioning signals also work



GPIO Interrupt Setup

- To setup GPIO interrupts, configure GPIOA->CPU_INT.IMASK and GPIOA->POLARITY31_16, GPIOA->POLARITY15_0 registers
- When edge trigger occurs, interrupt flag bit will be enabled in GPIOA->CPU_INT.RIS register

1020h	IIDX	Interrupt index	CPU_INT	Go
1028h	IMASK	Interrupt mask	CPU_INT	Go
1030h	RIS	Raw interrupt status	CPU_INT	Go
1038h	MIS	Masked interrupt status	CPU_INT	Go
1040h	SET	Interrupt set	CPU_INT	Go
1048h	ICLR	Interrupt clear	CPU_INT	Go



GPIO Interrupt Setup

- GPIOA->POLARITY31_16, GPIOA->POLARITY15_0 register specifies 2-bit code for each GPIO pin for when interrupt should be triggered (rising edge, falling edge, rising and falling edge)

Figure 9-57. POLARITY15_0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DIO15		DIO14		DIO13		DIO12		DIO11		DIO10		DIO9		DIO8	
R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO7		DIO6		DIO5		DIO4		DIO3		DIO2		DIO1		DIO0	
R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h		R/W-0h	

Table 9-57. POLARITY15_0 Field Descriptions

Bit	Field	Type	Reset	Description
31-30	DIO15	R/W	0h	Enables and configures edge detection polarity for DIO15. 0h = Edge detection disabled 1h = Detects rising edge of input event 2h = Detects falling edge of input event 3h = Detects both rising and falling edge of input event



GPIO Interrupt Setup

- GPIOA->CPU_INT.IMASK register enables (“unmasks”) interrupt on pin
 - Set to 1 to enable interrupt on that pin
 - Set to 0 to disable interrupt on that pin
- Example: Enable falling-edge interrupt on pin PA7

```
// Configure interrupt on PA7 for falling edge events (3.3V -> 0 V transition)
GPIOA->POLARITY15_0 = 0x00008000 ;

// Arm interrupt on PA7
GPIOA->CPU_INT.IMASK |= 0x00000080 ;
```

GPIO Interrupt Setup

- Interrupt Flag register (GPIOA->CPU_INT.RIS) is where trigger flags are stored
 - RIS (raw interrupt status) bit is set by hardware when edge trigger is detected
 - It stays set until we reset it to zero (done in ISR)

Raw interrupt status. Reflects all pending interrupts, regardless of masking. The RIS register allows the user to implement a poll scheme. A flag set in this register can be cleared by writing 1 to the ICLR register bit even if the corresponding IMASK bit is not enabled.

Figure 9-15. RIS

31	30	29	28	27	26	25	24
DIO31	DIO30	DIO29	DIO28	DIO27	DIO26	DIO25	DIO24
R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h
23	22	21	20	19	18	17	16
DIO23	DIO22	DIO21	DIO20	DIO19	DIO18	DIO17	DIO16
R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h
15	14	13	12	11	10	9	8
DIO15	DIO14	DIO13	DIO12	DIO11	DIO10	DIO9	DIO8
R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h
7	6	5	4	3	2	1	0
DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h



GPIO Interrupt Setup

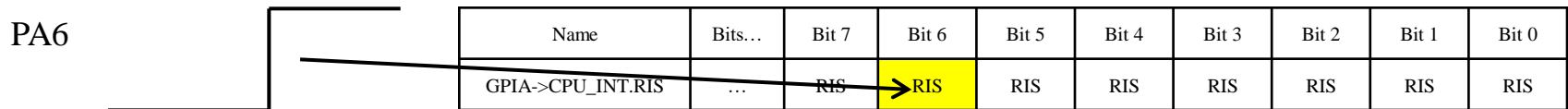
- Whenever NVIC detects that IFG bit is set, it will trigger interrupt (as long as associated GPIOA->CPU_INT.IMASK bit is set, meaning interrupt is unmasked)
 - If we don't clear bit when ISR is done, it just enters ISR again!
- Clearing interrupt flag done by setting appropriate bit in GPIOA->CPU_INT.ICLR register

```
// Clear interrupt flag for PA7 (without clearing any other interrupt flags)
GPIOA->CPU_INT.ICLR |= 0x00000080 ;
```



GPIO Interrupt Setup

- Side note: MSPM0 hardware will always set RIS bit when edge transition specified by POLARITY bit occurs
 - If IMASK bit is enabled, this triggers interrupt
 - If IMASK bit not enabled, you can still check this RIS bit using polling technique described earlier
 - *It is still set by hardware, even if it does not trigger interrupt*



- This is handy feature if you still want to use edge triggering without using interrupt feature



GPIO Interrupt Service Routine

- In Interrupt Service Routine (ISR), we perform any action we like
 - But we must clear RIS bit at the end for proper operation
- Example:

```
// increments every time PA4 sees falling edge
volatile uint32_t FallingEdges = 0;

void GROUP0_IRQHandler(void){

    FallingEdges = FallingEdges + 1; // increment counter

    GPIOA->DOUT31_0 ^= 0x00000001 ; // toggle LED (on PA0)

    GPIOA->CPU_INT.ICLR &= ~0x00000010;    // clear interrupt flag ("acknowledge")
}
```



GPIO Interrupt – Driverlib

- Driverlib provides straightforward calls to manage GPIO interrupts
 - To select rising or falling edge (sets bits in POLARITY31_16 and POLARITY15_0 registers)

```
void DL_GPIO_setLowerPinsPolarity (GPIO_Regs *gpio, uint32_t polarity)
void DL_GPIO_setUpperPinsPolarity (GPIO_Regs *gpio, uint32_t polarity)
```

- To enable interrupt on pin (sets bits in IMASK register)

```
void DL_GPIO_enableInterrupt (GPIO_Regs *gpio, uint32_t pins)
```



GPIO Interrupt – Driverlib

- To clear GPIO interrupt flag
 - Sets bits in ICLR register

```
void DL_GPIO_clearInterruptStatus (GPIO_Regs *gpio, uint32_t pins)
```

- Example (same as previous, except using driverlib)

```
// increments every time PA4 sees falling edge
volatile uint32_t FallingEdges = 0;

void PORT1_IRQHandler(void) {

    FallingEdges = FallingEdges + 1; // increment counter

    DL_GPIO_togglePins(GPIOA, DL_GPIO_PIN_0) ;    // toggle LED (on PA0)

    DL_GPIO_clearInterruptStatus(GPIOA, DL_GPIO_PIN_4) ; // clear interrupt flag
}
```

ISR Routines

- You define your own ISR's in your .c file(s)
 - Can be named whatever you want
 - Should only return void
- Caveat: The function name must be registered with the interrupt handler in `startup_mspgm0g3507_ticlang.c`
 - The interrupt service routine function names for each possible interrupt are stored in this file
 - This is called “registering” your ISR
- Is an ISR ever explicitly called in your code?




ISR Routines

- Interrupt vector table in startup_mspmg3507_ticlang.c

```
void (* const interruptVectors[]) (void) = {
    (void (*)(void)) ((uint32_t)&__STACK_END), /* initial SP */
    Reset_Handler, /* The reset handler */
    NMI_Handler, /* The NMI handler */
    HardFault_Handler, /* The hard fault handler */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler, /* SVC call handler */
    0, /* Reserved */
    0, /* Reserved */
    PendSV_Handler, /* The PendSV handler */
    SysTick_Handler, /* SysTick handler */
    GROUP0_IRQHandler, /* GROUP0 interrupt handler */
    GROUP1_IRQHandler, /* GROUP1 interrupt handler */
    TIMG8_IRQHandler, /* TIMG8 interrupt handler */
    UART3_IRQHandler, /* UART3 interrupt handler */
    ADC0_IRQHandler, /* ADC0 interrupt handler */
    ADC1_IRQHandler, /* ADC1 interrupt handler */
    CANFD0_IRQHandler, /* CANFD0 interrupt handler */
    DAC0_IRQHandler, /* DAC0 interrupt handler */
    0, /* Reserved */
    SPI0_IRQHandler, /* SPI0 interrupt handler */
    SPI1_IRQHandler, /* SPI1 interrupt handler */
    0, /* Reserved */
    0, /* Reserved */
    UART1_IRQHandler, /* UART1 interrupt handler */
    UART2_IRQHandler, /* UART2 interrupt handler */
    UART0_IRQHandler, /* UART0 interrupt handler */
    TIMG0_IRQHandler, /* TIMG0 interrupt handler */
    TIMG6_IRQHandler, /* TIMG6 interrupt handler */
    TIMA0_IRQHandler, /* TIMA0 interrupt handler */
    TIMA1_IRQHandler, /* TIMA1 interrupt handler */
    TIMG7_IRQHandler, /* TIMG7 interrupt handler */
    TIMG12_IRQHandler, /* TIMG12 interrupt handler */
}
```

ISR
function
names



- Note: Do not rearrange order of this table.
- Just replace current entry with your function name.
- Function names already there are just defaults – you can use the same or something different. Recommended to just use these names **because this file is shared between all projects in CCS by default.**
- If an interrupt is not enabled, then its associated ISR does not have to be defined anywhere
- i.e., if you are not using ADC interrupt then ADC0_IRQHandler does not need to be defined anywhere

Recap: Interrupt Setup

- There are 6 main steps to setup an interrupt
 1. Disable specific interrupt of interest in NVIC
 - Easiest way is to use driverlib `NVIC_DisableIRQ()` function
 2. In peripheral device registers (e.g., GPIO), set interrupt feature options (if any)
 - For GPIO, this would be rising or falling edge setting in `POLARITY31_16` or `POLARITY15_0` registers (can use `DL_GPIO_setLowerPinsPolarity()` or `DL_GPIO_setUpperPinsPolarity()` functions)



Recap: Interrupt Setup

3. Clear interrupt flag in RIS register

- This is so interrupt doesn't get triggered immediately when you arm it
- This is done by setting bit in ICLR register
- Can be done with `DL_GPIO_clearInterruptStatus()` for GPIO

4. Arm the interrupt in the peripheral device enable register (unmask register)

- For GPIO, this means setting the proper bit in IMASK register
- Can be done with `DL_GPIO_enableInterrupt()` function for GPIO

5. Set interrupt priority in the NVIC (optional)

- Easiest method is to use the driverlib function `NVIC_SetPriority(...)`



Recap: Interrupt Setup

6. Enable specific interrupt in NVIC

- Set bit in ISER register
 - Easiest method is to use driverlib call `NVIC_EnableIRQ()`
- This sounds complicated, but it is really not that bad when you see an example...



volatile Keyword in ISR's

- Final note about ISR's:
 - Any global variable that is modified in an ISR should be declared with the volatile keyword
 - Remember that ISR functions are not called explicitly
 - Thus compiler may perform optimizations with respect to global variable by assuming it only changes in functions explicitly called by program
 - volatile keyword tells compiler that this variable may be changed by hardware (in ISR)
 - *And thus it should not assume that it only changes in functions that are explicitly called*



GPIO Edge Triggered Interrupt

- Example Code

