بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

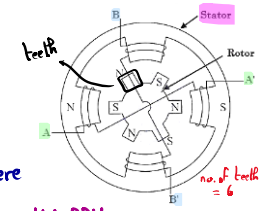
CSE211s:

Introduction to Embedded Systems

Stepper Motor & Interrupts

﴿يَرْفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُوا۟ مِنكُمْ وَٱلَّذِينَ أُوتُوا۟ ٱلْعِلْمَ دَرَجَٰتٍ﴾

* Stepper Motor :



→ Stepper Motors are used in Applications where precise positioning is more important than → high RPM, high torque, high efficiency

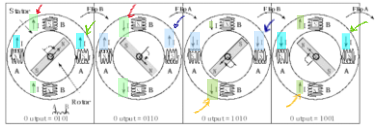
Use DC motor for applications requiring high torque or high speed, & use stepper motor for applications requiring accurate positioning at low speed

→ A bipolar stepper Motor has two Coils on the Stator, labeled A & B. Unipolar stepper divides those two Coils into four parts A, A' & B, B'

→ Unipolar Stepper has less torque than an equivalent-sized bipolar stepper. Also, unipolar steppers are easier to interface

→ there is always current Flowing through both Coils. When Current Flows through both Coils, the motor doesn't spin.

- to Move a bipolar Stepper we reverse the direction of Current through one of the Coils (not both)
- to Move it again we reverse the Current in the other Coil



- let the direction of the current be indicated by up & down

→ to make the current go up, the microcontroller outputs a binary 01

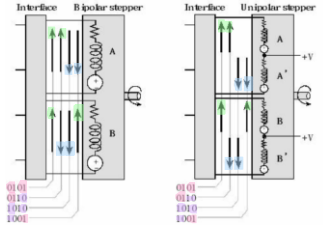
→ to make the current go down, it outputs a binary 10

* Since there are 2 coils, Four Outputs will be required (0101 ≡ up/up)

* to Spin the motor, we output the sequence ⇝ 0101_2 , 0110_2 , 1010_2 , 1001_2 ⋯ → ① over & over. Each Output causes the motor to rotate a fixed angle.

→ Spinning in Clockwise direction

to rotate the other direction, we reverse the sequence ⇝ 0101_2, 1001_2, 1010_2, 0110_2 ...



Because moving the motor involves accelerating a mass against a load friction, after we output a value, we must wait an amount of time before we can output again. if we output too fast, the motor doesn't have time to respond → ②

* the amount of rotation Caused by each Current reversal is a fixed angle depending on the number of teeth on the magnets.

for illustration

$$\text{Step angle} = \frac{360}{2 * \text{no. of teeth}}$$

A.K.A Step Size

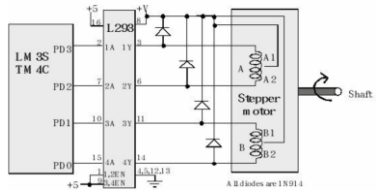
for problems

$$\text{Number of Steps per revolution} = \frac{360}{\text{Step Angle}}$$

$$\text{RPM} = 1000 * 60 * \frac{1}{\text{Number of Steps per revolution}} * \frac{1}{\text{Delay between states}}$$

(RPM: Rotation per Minute; 1000: milisecond → sec.; 60: sec → minute; Delay between states: in milisecond)

* Interfacing Stepper Motor to a Microcontroller :



[1] Initialize the interfacing pins to be digital Output

```
void stepper_motor_init()
{
  SYSCTL_RCGCGPIO_R |= 0x08;          //enable port D clock
  while ( (SYSCTL_PRGPIO_R&0x08) == 0) {}   //check for clock status
  GPIO_PORTD_AFSEL_R &= ~0x0F;        //no alternative function
  GPIO_PORTD_PCTL_R &= ~0xFFFF;
  GPIO_PORTD_DIR_R |=  0x0F;          //set PD3-0 as output
  GPIO_PORTD_AMSEL_R &= ~0x0F;        //disable analong
  GPIO_PORTD_DEN_R |= 0x0F;           //enable digital I/O
}
```

[2] Determine the delay between states

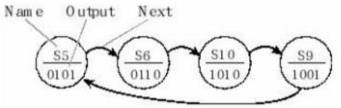
$$\text{delay (in milisec)} = \frac{60 * 1000}{\text{States per revolution} * \text{RPM}}$$

[3] Write the Sequence on the data register

```
while(1)
{
  GPIO_PORTD_DATA_R = 5;
  delay(50);
  GPIO_PORTD_DATA_R = 6;
  delay(50);
  GPIO_PORTD_DATA_R = 10;
  delay(50);
  GPIO_PORTD_DATA_R = 9;
  delay(50);
}
```

→ ① the Spinning Sequence

→ ② Assume the delay is 50ms



* Interrupts :

→ interrupts are the events that temporarily suspend the main program,

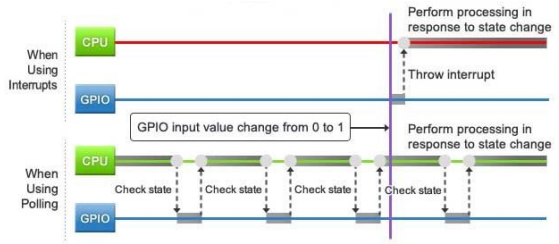
① pass the Control to the external Sources & execute their task.

② It then passes the Control to the main program where it had left off

* Polling vs Interrupts :

→ while polling is a simple way to check for state changes, there's a cost. if the checking interval is too long, there can be a long lag between occurrence & detection. & you may miss the change Completely, if the state changes back before check. A shorter interval will get Faster & more reliable detection, but also Consumes much more processing time & power, Since many more checks will come back negative (A.K.A Busy-waiting)

→ An Alternative approach is to use interrupts. with this method, the state change generates an interrupt Signal that Causes the CPU to ① suspend its Current operation & ② Save its Current state, then ③ execute the processing associated with the interrupt, & then ④ restore its pervious state & ⑤ resume where it left off.



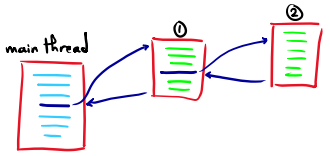
→ An interrupt is the automatic transfer of software execution in response to a hardware event (trigger) that is asynchronous with the Current software execution

* Thread : it's defined as the path of action of software as it executes (أثناء تنفيذه)

- the execution of the ISR is called a background thread

this thread is created by the hardware interrupt request & is killed when ISR returns from interrupt

A new thread is created for each interrupt request



* Process : it's defined as the action of software as it executes.

* ISR : Stands for Interrupt Service Routine, it's a software process (يُعنى) invoked by an interrupt request from a hardware device.

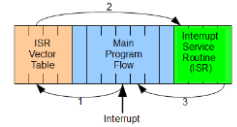
It handles the request & sends it to the CPU, interrupting the active process. when the ISR is Complete, the process is resumed.

* IRQ : Stands for Interrupt Request. PCs use interrupt requests to manage various hardware operations.

It's important to assign different IRQ addresses to different hardware devices.

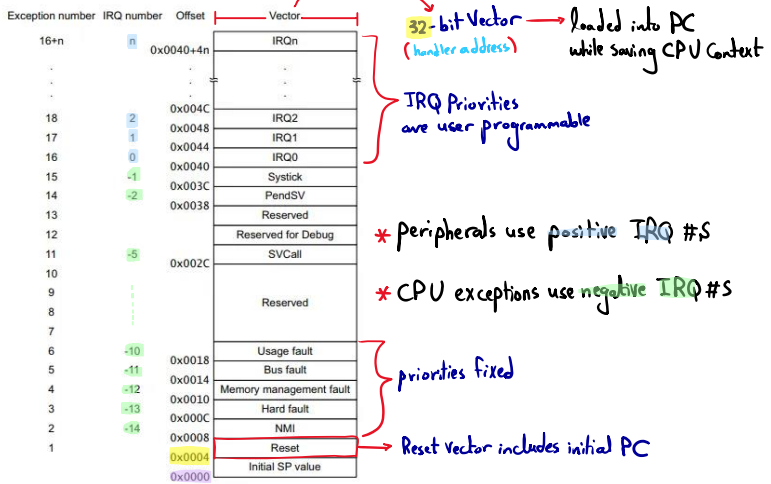
* Interrupt Vector Table : it's a table of memory addresses of interrupt/exception handler routines

→ it defines where the Code of a particular interrupt/exception service routine is located in microcontroller memory.



Vector : means memory address

there are 139 interrupt Vector



* Vector table is stored in ROM starting from address 0x0004
* Start up Code initializes the whole vector table by placing dummy ISRs
* ROM location 0x0000 has the initial stack pointer, & location 0x0004 Contains the initial program Counter, which is Called the reset Vector.

it points to the reset handler which's the first thing executed following reset

Similar to Newton's law of inertia (an object in motion tends to stay in motion), computer processes continue to run unless interrupted. Without an interrupt request, a computer will remain in its current state. Each input signal causes an interrupt, forcing the CPU to process the corresponding event. ☺

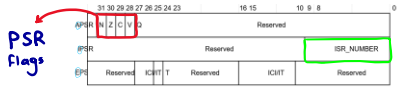
* Exceptions : it's a Synchronous event that occurs during the execution of a thread that interrupts the normal flow of instructions.

⇒ Examples : including some interrupts , resets , fault handlers, ... etc

- Exceptions are managed by NVIC

⇒ PSR : The Program Status Register Combines

- APSR — Application
- IPSR — Interrupt
- EPSR — Execution



⇒ IPSR : Contains the ISR number of the current exception activation

Can include up to 255 ISR

* Context Switch :

- Actions taking to switch from one thread to another
- it's hardware dependent

↳ in TM4C123 : ① Current instruction is finished

② Eight Registers is pushed on the stack (on top) → R0, R1, R2, R3, R12, LR, PC, PSR

③ LR is set to 0xFFFF FFF9

④ IPSR is set to the interrupt number

⑤ PC is loaded with interrupt vector

* Nested Vectored Interrupt Controller (NVIC):

→ the major role of NVIC is to handle interrupt as The NVIC :

① initiates a call to that vector table

② locate the interrupt number that has occurred

③ picks the address of the service routine from the vector table

→ it provides the users with a feature to prioritize the interrupts according to their use & need

⚠ the first 16 exceptions cannot be prioritized as they are system exceptions & the system will not let the user change the priority of its interrupts

→ Allows a programmable priority level of 0-7 for each interrupt. A higher level to a lower priority, so level 0 is the highest interrupt priority

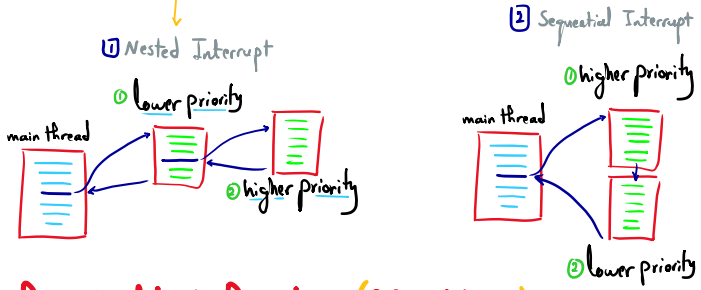
→ the role of NVIC is manage all low & high priority interrupts in such a way that the higher priority interrupts always gets to execute before a lower priority interrupts occurs earlier.

↝ if a low priority interrupt has occurred & being executed. while a low priority interrupt is still executing, a higher priority interrupt occurs, ARM CPU will pause the low priority interrupt & stants to execute high priority interrupt

* But why "Nested" ?

→ Nesting of interrupts is somewhat similar to nested for-loops

i.e. processing an interrupt with higher priority with in another interrupt with lower priority



• Priority Mask Register (PRIMASK):

→ the primask register prevents activation of all exceptions with programmable priority

0 — No effect

1 — prevents the activation

to enable global interrupts → PRIMASK must equal zero

• Base Priority Mask Register (BASEPRI):

→ the BASEPRI register defines the minimum priority for exception processing

0x00 — no effect

Nonzero — the processor doesn't process any exception with a priority value greater than or equal to BASEPRI

* NVIC Register Descriptions:

- Interrupt Set enable (NVIC_ENn_R) : (on read)

→ The ENn registers enable interrupts & show which interrupts are enabled

→ NVIC provides 5 interrupt Set enable registers (n = 0 → 4)

* On writing
  - = 0 → No effect
  - = 1 → enables the interrupt

you can't disable the interrupt by clearing a bit, as there is a register specified for disabling → DISn

Interrupt 0-31 Set Enable (EN0)

Base 0xE000.E000
Offset 0x100
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

→ Each bit should be set to enable the interrupt for a specific peripheral

interrupts from IRQ = 0 → 31 are controlled by bit 0 → 31 in EN0

interrupts from IRQ = 32 → 63 are controlled by bit 0 → 31 in EN1

⋮

Thus, the register n & bit b are determined by IRQ number as follows

→ n = INT(IRQ/32) → b = (IRQ % 32) OR (IRQ − 32 * n)

using calculator

* for example :

IRQ	ISR name in Startup.s
30	GPIOPortF_Handler

∴ n = INT(30/32) = 0 → EN0

∴ b = (30 % 32) = 30 OR (30 − 32*0) = 30

∴ For PortF interrupt → mask bit 30 in EN0

- Interrupt Priority level (NVIC_PRIn_R) :

→ to Configure the needed priority level for certain interrupt

→ There are 35 priority level registers groups (n = 0 → 34)

Each group sets the interrupt priority for 4 peripherals where only upper 3 bits from each segment are used

PRIn Register Bit Field	Interrupt
Bits 31:29 (S3)	Interrupt [4n+3]
Bits 23:21 (S2)	Interrupt [4n+2]
Bits 15:13 (S1)	Interrupt [4n+1]
Bits 7:5	Interrupt [4n]

Segment 0

Assign priority level
Value between 0 → 7

Interrupt 0-3 Priority (PRI0)

Base 0xE000.E000
Offset 0x400
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	INTD			reserved					INTC			reserved				
Type	RW	RW	RW	RO	RO	RO	RO	RO	RW	RW	RW	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	INTB			reserved					INTA			reserved				
Type	RW	RW	RW	RO	RO	RO	RO	RO	RW	RW	RW	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Segment

Thus, the priority group number n & the segment s are determined by IRQ number as follows :

→ n = INT(IRQ/4) → s = IRQ % 4 OR (IRQ − 4 * n)

* for example :

IRQ	ISR name in Startup.s
30	GPIOPortF_Handler

∴ n = INT(30/4) = 7 → PRI7

∴ s = (30 % 4) = 2 OR (30 − 4 * 7) = 2

∴ For PortF priority register → PRI7 Segment 2 → bits 21:23

- System Handler priority (NVIC_SYS_PRIn_R) :

→ to handle system exception priority like (usage fault, bus fault, ... / SVCall / Systick)

→ Systick is considered as a system exception not as an interrupt

* to enable Systick interrupt

Address	31-24	23-17	16	15-3	2	1	0	Name
SE000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R

INTEN → set bit 1 in Control Register

* to Configure Systick priority

NVIC_SYS_PRI3_R

Bit/Field	Name	Type	Reset	Description
31:29	TICK	RW	0x0	SysTick Exception Priority This field configures the priority level of the SysTick exception. Configurable priority values are in the range 0-7, with lower values having higher priority.

* NMI : Non maskable Interrupt, this interrupt cannot be disabled

if errors happen in other exception handlers, a NMI will be triggered.

Aside from the Reset exception, it has the highest priority of all exceptions.

* GPIO Interrupts :

→ Configuring the interrupts for individual pins in each port

Address	7	6	5	4	3	2	1	0	Name
$4000.4404	IS	IS	IS	IS	IS	IS	IS	IS	GPIO_PORTA_IS_R
$4000.4408	IBE	IBE	IBE	IBE	IBE	IBE	IBE	IBE	GPIO_PORTA_IBE_R
$4000.440C	IEV	IEV	IEV	IEV	IEV	IEV	IEV	IEV	GPIO_PORTA_IEV_R
$4000.4410	IME	IME	IME	IME	IME	IME	IME	IME	GPIO_PORTA_IM_R
$4000.4414	RIS	RIS	RIS	RIS	RIS	RIS	RIS	RIS	GPIO_PORTA_RIS_R
$4000.4418	MIS	MIS	MIS	MIS	MIS	MIS	MIS	MIS	GPIO_PORTA_MIS_R
$4000.441C	ICR	ICR	ICR	ICR	ICR	ICR	ICR	ICR	GPIO_PORTA_ICR_R

→ GPIO Interrupt Sense (GPIOIS) :

- It determines level or edge triggered
- Clearing a bit (0) ⟶ Detect an edge-sensitive on the pin
- Setting a bit (1) ⟶ Detect an level-sensitive on the pin

→ GPIO Interrupt Event Register (GPIOIEV) :

- to decide low level , high level OR falling-edge , Rising-edge
- Clearing a bit (0) & { GPIOIS=0 ⟶ Falling-edge ; GPIOIS=1 ⟶ low-level }
- Setting a bit (1) & { GPIOIS=0 ⟶ Rising-edge ; GPIOIS=1 ⟶ high-level }

→ GPIO Interrupt Both edges (GPIOIBE) :

- Clearing a bit (0) ⟶ Interrupt is controlled by GPIOIEV
- Setting a bit (1) ⟶ Both edges on the pin trigger an Interrupt

→ GPIO Interrupt Clear Register (GPIOICR) :

- The corresponding bit fields in this register clears the interrupt for the respective bit. To ensure that any pervious interrupt has been cleared writing 1 to the respective bit field will clear the interrupt on that pin (= Acknowledge)

⚠

- It's critical that the interrupt handler clears the interrupt flag before returning from interrupt handler ⟶ otherwise the interrupt appears as if it is still pending and the interrupt handler will be executed again & again forever and the program hanges

- Clearing a bit (0) ⟶ No action
- Setting a bit (1) ⟶ the corresponded edge-triggered interrupt is cleared

→ GPIO Interrupt Mask Register (GPIOIM) :

- Used to enable the interrupt capability of a given peripheral at the module level
- Clearing a bit (0) ⟶ Interrupt is masked (disabled)
- Setting a bit (1) ⟶ Interrupt is unmasked (enabled)

→ GPIO Raw Interrupt Status (GPIORIS) : (Read only)

- If a bit is cleared (0) ⟶ An interrupt Condition has not occurred on the pin
- If a bit is set (1) ⟶ An interrupt Condition has occurred on the pin

- it tells you if there is an interrupt pending regardless of the State of the "interrupt enable" or "interrupt mask" bit.

* For edge-detect interrupts → this bit is cleared by writing a 1 to the corresponding pin in the GPIOICR register
* For level-detect interrupts → this bit is cleared when the level is deassented

→ GPIO Masked Interrupt Status (GPIOMIS) : (Read only)

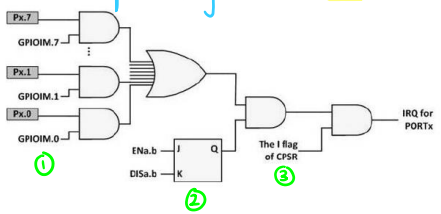
- If a bit is cleared (0) { An interrupt Condition has not occurred on the pin ; OR the pin has been masked }
- If a bit is set (1) ⟶ An interrupt Condition has occurred on the pin

- it tells you there is an interrupt only if "enable" or "mask" bit in the appropriate state

* For edge-detect interrupts → this bit is cleared by writing a 1 to the corresponding pin in the GPIOICR register
* For level-detect interrupts → this bit is cleared when the level is deassented

* Upon Reset, all interrupts are disabled → to enable any interrupt :

"Interrupt Enabling with all 3 levels"



3 steps
1. Enable the interrupt for a specific peripheral module ✓ (go)
2. Enable the interrupts at the NVIC module ✓ (go)
3. Enable the interrupt globally ?

→ by Clearing the I bit in the PRIMASK (press)

in startup files → Assembly functions needed for interrupt enabling & disabling

```
EnableInterrupts
        CPSIE   I
        BX      LR
```

```
DisableInterrupts
        CPSID   I
        BX      LR
```

* CPSIE : Assembly instruction that set I = 0

* CPSID : Assembly instruction that set I = 1

* Using Interrupt Steps :

• Step Zero : → initialize the peripheral as GPIO as we have studied

1 Enable the interrupt for individual pins :

→ Decide the Sense & Configure it then enable the mask bit

```
GPIO_PORTF_IS_R &= ~0x10;          //PF4 is edge-sensitive
GPIO_PORTF_IBE_R &= ~0x10;         //PF4 is not both edges
GPIO_PORTF_IEV_R &= ~0x10;         //PF4 falling edge event
GPIO_PORTF_ICR_R = 0x10;           //clear flag4
GPIO_PORTF_IM_R |= 0x10;           //arm interrupt on PF4
```

Since clearing has no action

= Enable

2 Enable the interrupt for the whole port :

→ Configure the NVIC interrupt Control Registers

Also, Configure the priority if you want

calculations

Another methods

```
NVIC_EN0_R = 0x40000000;        //enable interrupt 30 in NVIC
//NVIC_EN0_R |= (1<<30);

NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // priority 5
//NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|(2<<21);
//NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|(1<<22);
```

3 Enable the global interrupt :

"Method 1" → `EnableInterrupts();` → C statement that's defined in startup file

"Method 2" →

```
__enable_irq();
```

"Method 3" : write assembly instruction in C file

```
__asm
{
  CPSIE I
}
```

4 Write the interrupt handler function

```
void GPIOPortF_Handler()
  {
    GPIO_PORTF_ICR_R = 0x10;
    FallingEdges = FallingEdges + 1;
  }
```

don't forget to clear the interrupt flag since it's a edge-triggered interrupt

﴿وَآخِرُ دَعْوَاهُمْ أَنِ الْحَمْدُ لِلَّهِ رَبِّ الْعَالَمِينَ﴾

* Test yourself → QUIZ