INTERFACING

INTERRUPT

AMIT

Interrupt

Definition

- An interrupt is a signal sent to the processor that interrupts the current process. It may be generated by a hardware device or a software program. A hardware interrupt is often created by an input device such as a mouse or keyboard. ... An interrupt is sent to the processor as an interrupt request, or IRQ

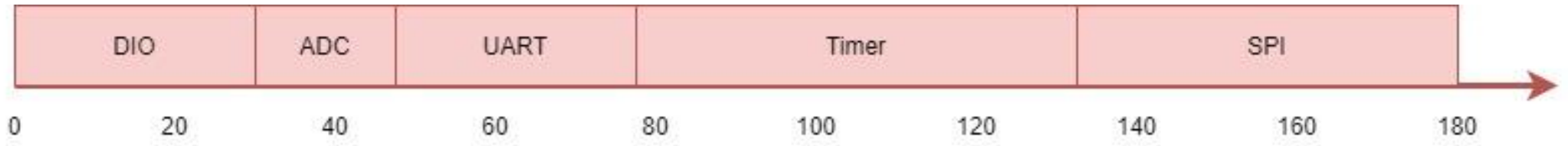
- Interrupt is an event the disturbs the normal execution sequence of the code and makes the processor to jump to another piece of code to execute called Interrupt Service Routine (ISR). After the processor finish executing of the ISR, it returns to the interrupted code to continue from the interrupted instruction.

AMIT

Important Definitions

Determinism

- It is to know what happens at every point in timeline, like the following figure, which every task period is known exactly, as example DIO will take 27.5 ms, ADC will take 20 ms, …. And so on.



AMIT

Important Definitions

Responsiveness:

- It is how fast the processor will response to the event, like the following example code:
  - Assume that the Timer task Event is triggered while the process executes into the DIO task, Do the processor will respond to the timer event?
    - Unfortunately, the processor will not respond to this event until it finishes the DIO, ADC and UART_Rx tasks.

```
int main()
{
    while (1)
    {
        DIO();
        ADC();
        UART_Rx();
        Timer();
        UART_Tx();
    }
}
```

AMIT

Types of Systems:

Super-Loop System:

- It is the system which all periodic tasks will be written in the super loop “while(True)”.
- It is a deterministic system because:
  - After running the DIO task, ADC task will run.
  - After running the ADC task, UART_Rx will run.
  - After running the UART_Rx task, Timer will run.
  - After running the Timer task, UART_Tx will run.
  - After running the UART_Tx task, DIO will run and so on.
- It is a non-responsiveness system because:
  - Assume that the Timer task Event is triggered while the process executes into the DIO task, Do the processor will respond to the timer event?
    - Unfortunately, the processor will not respond to this event until it finishes the DIO, ADC and UART_Rx tasks.

```
int main()
{
    while (1)
    {
        DIO();
        ADC();
        UART_Rx();
        Timer();
        UART_Tx();
    }
}
```

AMIT

Types of Systems:

Super-Loop System:

- The advantages of Super-Loop system is:
  - Easy Software, all Software will be written inside the while(1) “one place”.
  - Needs minimal Hardware resources.
  - Higher determinism.

- The disadvantages of Super-Loop system is:
  - Lower responsiveness.
  - Consumes more power.

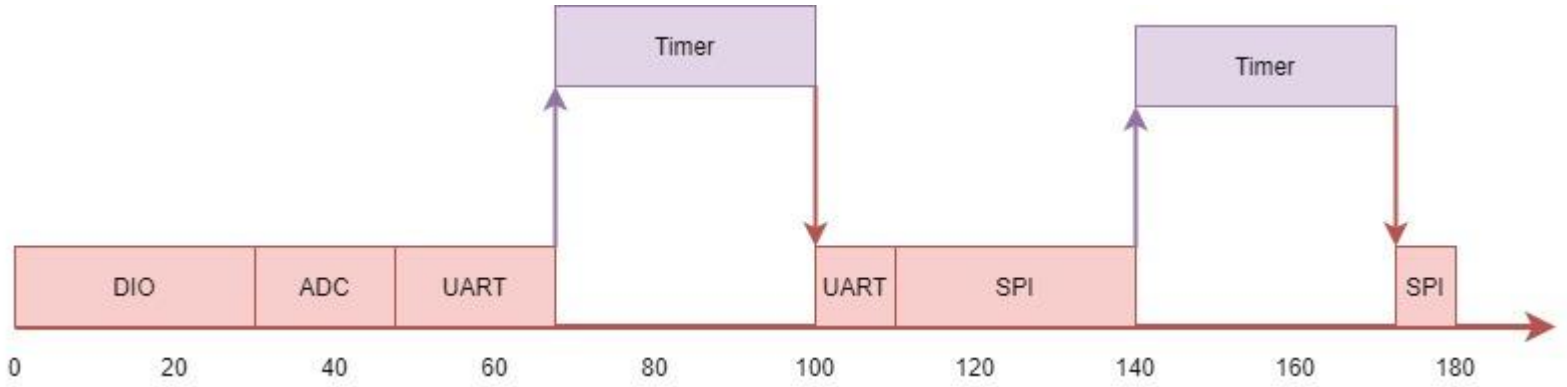
```
int main()
{
    while (1)
    {
        DIO();
        ADC();
        UART_Rx();
        Timer();
        UART_Tx();
    }
}
```

AMIT

Types of Systems:

Foreground-Background System:

- It is a system which is based on interrupt event triggered, this event will interrupt the normal code “code inside the super-loop” to execute most important task exists inside Interrupt Service Routine Code “ISR”.
- This system depends on interrupting the Super-Loop code.



Types of Systems:

Foreground-Background System:

- Now, in this system, if the timer event is triggered, the processor will immediately execute the task which is based on Timer event triggered.
- In this system, the Timer event is not exactly determined when it will be triggered.
- In this system, we need extra hardware to immediately capture the event when it is triggered.

```
int main()
{
    while (1)
    {
        DIO();
        ADC();
        UART_Rx();
        SPI_Tx();
    }
}
ISR(TIMER)
{
    //higher priority task;
}
```

AMIT

Types of Systems:

Foreground-Background System:

- The advantages of Super-Loop system is:
  - Higher responsiveness.
  - Low power consumption.

- The disadvantages of Super-Loop system is:
  - Lower determinism.
  - Complex in Software.
  - Extra Hardware resources.

```
int main()
{
    while (1)
    {
        DIO();
        ADC();
        UART_Rx();
        SPI_Tx();
    }
}
ISR(TIMER)
{
    //higher priority task;
}
```

AMIT

Types of Interrupts:

Classification According to Processor:

- **External Interrupt**:
  - It is any interrupt which occurs outside the processor, exactly inside any peripheral -outside the processor- supports firing interrupt, Like as Example:

    - External Interrupt Peripheral (**EXTI**).
    - Timer Peripheral(**Timer**).
    - Analog to Digital Converter Peripheral(**ADC**).
    - Serial Peripheral Interface(**SPI**).
    - Universal Asynchronous Receiver Transmitter Peripheral(**UART**).

AMIT

Types of Interrupts:

Classification According to Processor:

- **Internal Interrupt**:
  - It is any interrupt which occurs inside the processor itself, interrupt can be internally fired, Like as Example:
    - Hard Fault Interrupt:
      - It causes as example if the processor executes a wrong instruction which does not exist into the instruction set.
    - Division by zero:
      - It causes due to division by zero, it may be configured to fire interrupt or gives the result by Zero directly.

AMIT

Types of Interrupts:

Classification According to Processor:

- **Internal Interrupt**:
  - It is any interrupt which occurs inside the processor itself, interrupt can be internally fired, Like as Example:
    - SWI “software interrupt”:
      - It is an assembly instruction that causes a software interrupt will any hardware event.
    - Core Peripherals:
      - It causes due to peripherals exists inside the processor, and this peripheral supports interrupt firing.

AMIT

How the Interrupt occurs:

Interrupt Theory:

- **Main event to capture the interrupt event**:
  - First, the peripheral must support interrupt firing, and it must have a configurable interrupt mask(PIE).
  - Then, the peripheral must have a mark to determine it finished its functionality or not.(PIF)
  - Controlling on the whole interrupt events to manage masking on them in one step(GIE).

AMIT

How the Interrupt occurs:

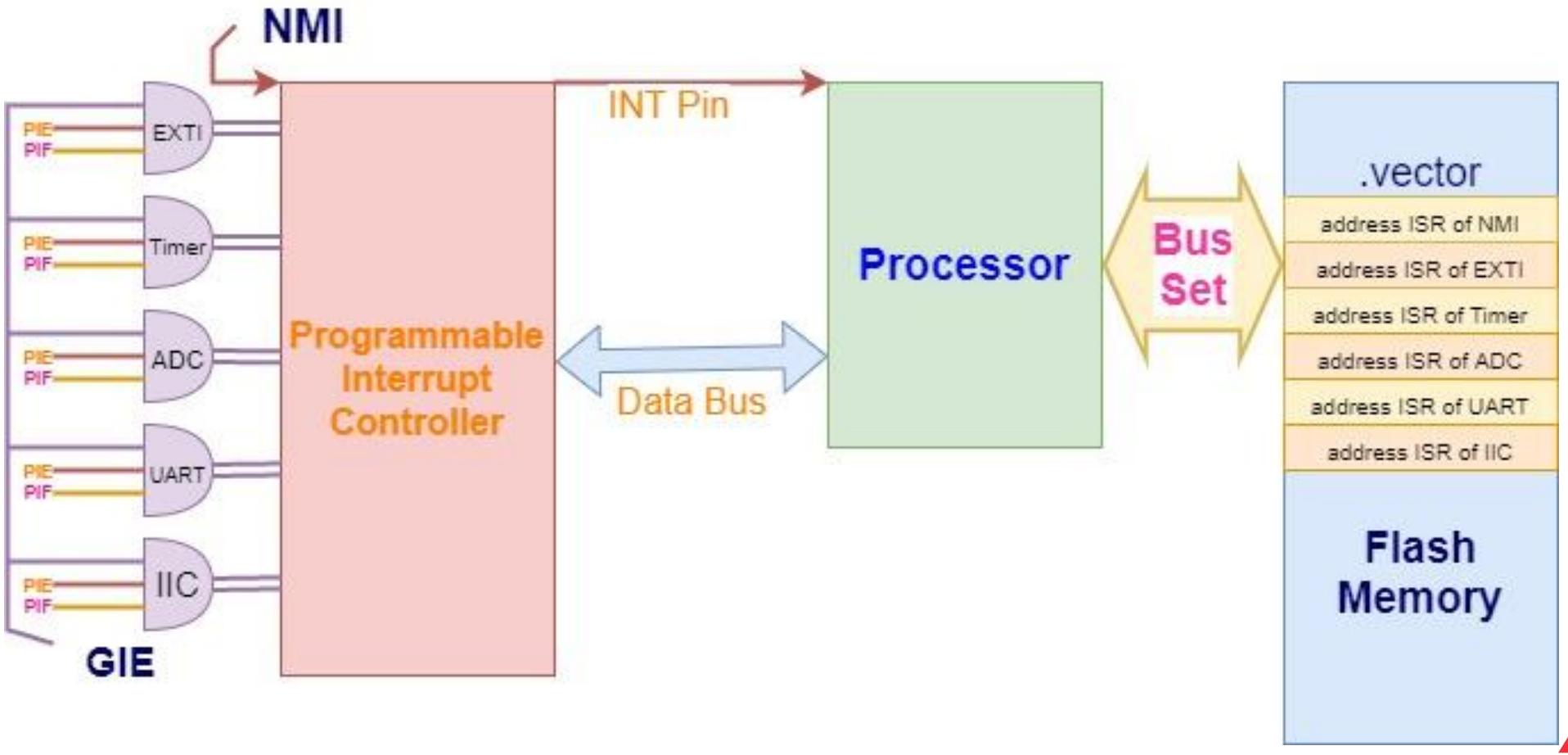
Interrupt Controller Hardware:

- **Programable Interrupt Controller(PIC)**:
  - PIC is a hardware element that handles the interrupt serving and priorities. All interrupts are connected to the PIC, when any interrupt happens, the PIC receives the interrupt request and generate a signal to the processor on its INT pin. Then the PIC tells the processor the ID of the interrupt happened through a special data bus.
  - Inside the processor there is a piece of memory called vector table. Every location of this memory (called vector) corresponding to a certain interrupt. When the process receives an interrupt request from the PIC and gets its ID, it jumps to the corresponding location in the vector table to find the address of its ISR.
  - The PIC also handles the interrupt priorities, if two interrupts happened at the same time, the upper interrupt would have higher priority. The PIC tells the processor about the upper interrupt first, then tells the processor about the lower one.

AMIT

How the Interrupt occurs:

Interrupt Hardware Circuit:



AMIT

How the Interrupt occurs:

Interrupt Handling Scenario:

- As we mentioned before, every peripheral supports interrupt, it must have an interrupt mask bit, Called **P**eripheral **I**nterrupt **E**nable "**PIE**", this bit must be set if you want to fire interrupt.
- Every peripheral supports interrupt, it must have an interrupt flag bit, Called **P**eripheral **I**nterrupt **F**lag "**PIF**", this bit is set if the peripheral finishes its functionality.

- There is a bit which its functionality to mask all the maskable-interrupt sources at one step, it is called "**G**lobal **I**nterrupt **E**nable "**GIE**".

- So, if all previous bits are set, the output of AND gate will be one, so PIC will receive an interrupt signal on its input pins, Now what will exactly happen?

AMIT

How the Interrupt occurs:

Interrupt Handling Scenario:

- PIC will send a signal to INT Pin to indicate the processor that an interrupt event has been occurred.
- Then, PIC will send the ID of the triggered interrupt on the Data Bus.
- Processor will leave the execution of normal code and it will jump to the location related to the ID into the vector table in the memory.
- Into this location, there is the address of ISR function related to the ID of triggered interrupt, so the processor will jump to this address and execute it.

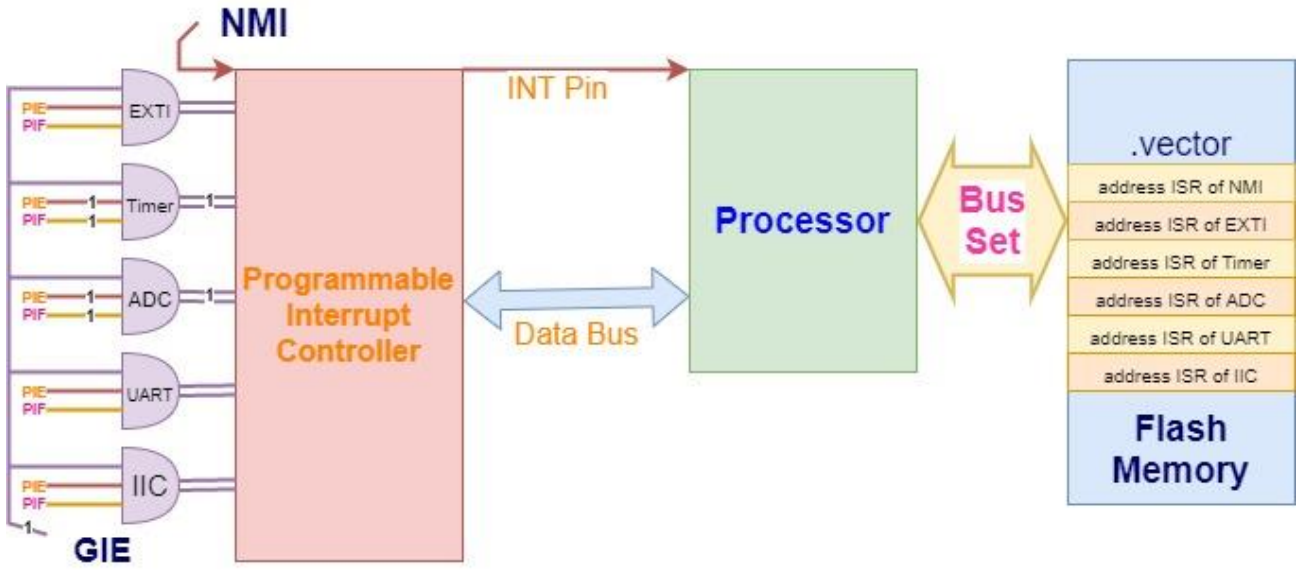
AMIT

How the Interrupt occurs:

PIC Functionality:

- As we mentioned before, PIC send a signal on INT Pin, and the ID of triggered interrupt on Data Bus to the process.

- PIC manages the interrupts if more than one interrupt has been triggered at the same time, Like the following figure, ADC and Timer fire interrupt at the same time, what will happened?



AMIT

How the Interrupt occurs:

PIC Functionality:

- In this Case, PIC will send one triggered signal per time, so it will choose one of them depending on the priority.

- Priority here is a fixed priority depending on vector table, I mean the priority of interrupt is not configurable, depending on the hardware priority, I mean which is connected first has the highest priority and vice versa.

- So, PIC will Timer signal at first to the processor, and while starting the execution, the PIF bit of Timer will be cleared by hardware.

- So, the expectation that after starting the ISR of Timer, PIC will send the signal of ADC, and it will interrupt the execution of Timer ISR, but the ADC has a priority lower than Timer, so it can not be even called a nesting.

AMIT

How the Interrupt occurs:

PIC Functionality:

- Nesting:
  - It is the ability of interrupt the interrupts, I mean that any higher priority interrupt can interrupt the execution of lower priority interrupt.

- So, according to the nesting definition, it is a mandatory to clear the GIE bit before starting execution of the ISR of any interrupt.

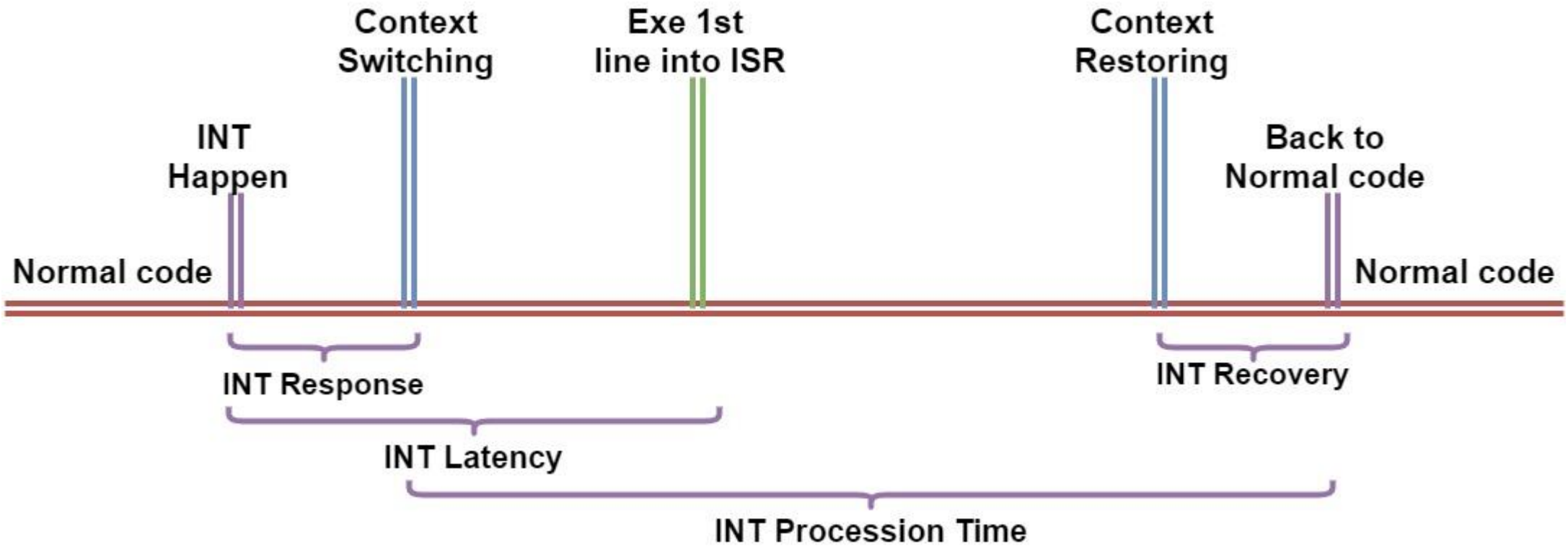
- So, our micro controller does not support the nesting-interrupt concept.

- Now, we know that the interrupt force the processor to jump to ISR to fast execution, and after execution, the processor completes the execution of normal code again, but how do it happen?

AMIT

How the Interrupt occurs:

Interrupt Timeline:



How the Interrupt occurs:

Interrupt Timeline:

- The interrupt signal is sent while processor executes the normal code.
- In the INT response, there is a delay until starting the context switching, delay causes because if the processor runs any instruction, the instruction must be finished first before starting the context switching.
- The operation into the context switching is a mandatory to save what the processor does into the normal code.

AMIT

How the Interrupt occurs:

Interrupt Timeline:

- **Context Switching operation:**
  1. Clear PIF bit.
  2. Clear GIE bit.
  3. Save PC register in Stack.
  4. Go to vector table at the int source location.
  5. Create JUMP instruction to the address of ISR.
  6. Save GPRs registers.
  7. Save PSW registers.
  8. Execute the body of ISR.
  9. Execute the RETI “return interrupt instruction”, into this instruction the GIE will be set again.
  10. Retrieve the PSW.
  11. Retrieve the GPRs.
  12. Retrieve the PC.

AMIT

Types of Interrupts:

Classification According to Interrupt Type:

- **Maskable Interrupt**:
  - It is any interrupt which can be masked, I mean any interrupt can be controlled by a **PIE** bit, also can be masked by **GIE** bit.
- **Non-Maskable Interrupt**:
  - It is any interrupt which can not be masked, I mean it has only **PIF** which it is connected directly to **PIC,** so any **NMI** when it occurs, it is immediately served.
  - The only **NMI** exists into our micro is **RESET,** once it occurs, it must be served.
  - **RESET** can be triggered by several events:
    - Power on.
    - Brown-out detector.
    - **RESET** Pin.
    - Watch-dog timer and JTAG**.**

AMIT

EXTERNAL INTERRUPT

AMIT

Handling Techniques:

Polling:

- In polling handling technique, the processor keep asking about the event if it happened or not.
- Polling can be continuous operation like the first example code, in this mechanism, the processor is looked-up until the timer flag rises.
- Polling can be periodic operation like the second example code, it is also called “watch technique”, in this mechanism, the processor will check the flag value every while iteration, if it rises, UART task will be executed, if not the processor will execute the DIO task.
- It is a high deterministic system, but it is a blocking system.
- It can be use with short tasks or critical tasks.

```
int main()
{
    System_init();
    while (1)
    {
        DIO();
        // continuous polling
        while (TimerFlag == 0);
        UART();
    }
}
```

```
int main()
{
    System_init();
    while (1)
    {
        DIO();
        // periodic polling
        // watch technique
        if (TimerFlag == 0)
            { UART(); }
    }
}
```

AMIT

Handling Techniques:

Interrupt:

- In interrupt handling technique, the processor is doing other tasks, but once the event happens, the interrupt controller “**PIC**” notifies the processor that an interrupt is happened.
- It is a high responsiveness system “not blocking system”.
- It is a low deterministic system.
- It can be used with longtime tasks.

```
int main()
{
    System_init();
    while (1)
    {
        DIO();
        ADC();
    }
}
ISR(TIMER)
{
    UART();
}
```

AMIT

External Interrupt Peripheral:

Description:

- The External Interrupts are triggered by the INT0, INT1, and INT2 pins. Observe that, if enabled, the interrupts will trigger even if the INT0..2 pins are configured as outputs.
- This feature provides a way of generating a software interrupt.
- The external interrupts can be triggered by a falling or rising edge or a low level (INT2 is only an edge triggered interrupt).
- This is set up as indicated in the specification for the MCU Control Register – MCUCR – and MCU Control and Status Register – MCUCSR. When the external interrupt is enabled and is configured as level triggered (only INT0/INT1), the interrupt will trigger if the pin is held low.
- Note that recognition of falling or rising edge interrupts on INT0 and INT1 requires the presence of an I/O clock, described in “Clock Systems and their Distribution” on page 22.
- Low level interrupts on INT0/INT1 and the edge interrupt on INT2 are detected asynchronously.
- This implies that these interrupts can be used for waking the part also from sleep modes other than Idle mode.

AMIT

External Interrupt Peripheral:

Brief Description:

- The External Interrupts are triggered by the INT0, INT1, and INT2 pins.
- This interrupt can be triggered by a level or an edge sensitive.
- INT0, INT1 pins can be triggered by:
  - LOW LEVEL.
  - RISING EDGE.
  - FALLING EDGE.
  - ANY LOGICAL CHANGE(rising and falling).
- INT2 pin can be triggered by:
  - RISING EDGE.
  - FALLING EDGE.

AMIT

External Interrupt Peripheral:

Register Description:

- We have access only on the Lower four bits(0→3), the upper is belonged to other peripheral.
- Bit0, Bit1 are used to determine on which level the INT0 will be triggered.
- Bit2, Bit3 are used to determine on which level the INT1 will be triggered.

MCU Control Register – MCUCR

The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

AMIT

External Interrupt Peripheral:

Register Description:

- At the following table, there is the description of these bits to determine on which level the INTx will be triggered.

ISC11 / ISC01	ISC10 / ISC00	Description
0	0	The low level of INTx generates an interrupt request.
0	1	Any logical change on INTx generates an interrupt request.
1	0	The falling edge of INTx generates an interrupt request.
1	1	The rising edge of INTx generates an interrupt request

AMIT

External Interrupt Peripheral:

Register Description:

- We have access only on the sixth-order bit, the remainder bits are belonged to other peripheral.
- Bit6 is used to determine on which level the INT2 will be triggered.

MCU Control and Status Register – MCUCSR

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

External Interrupt Peripheral:

Register Description:

- At the following table, there is the description of these bits to determine on which level the INT2 will be triggered.

ISC2	Description
0	The falling edge of INTx generates an interrupt request.
1	The rising edge of INTx generates an interrupt request

AMIT

External Interrupt Peripheral:

Register Description:

- We have access only on the upper three bits, the remainder bits are belonged to other peripheral.
- Bit5, Bit6, Bit7 are used to Enable The interrupt triggering source for INTx “**PIE**”.
- Writing one means enabling PIE for INTx sources.
- Writing zero means disabling PIE for INTx sources.

General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

External Interrupt Peripheral:

Register Description:

- We have access only on the upper three bits, the remainder bits are reserved.
- Bit5, Bit6, Bit7 are used to determine if the interrupt source is triggered or not “**PIF**”.
- Reading one means the interrupt source is triggered .
- Reading zero means the interrupt source is not triggered .

General Interrupt Flag Register – GIFR

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

AMIT

External Interrupt Peripheral:

Register Description:

- **Clearing Flag Mechanism:**
  - Most of flag are cleared by hardware if its ISR is executed “Interrupt Techniques”.
  - If polling technique is used, there is another clearing mechanism mentioned into the datasheet:
    - The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it This flag is always cleared when INTx is configured as a level interrupt.

General Interrupt Flag Register – GIFR

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

External Interrupt Peripheral:

Vector table:

- **Vector Table in ATMEGA32:**
  - This table is written depending on the hardware priority of interrupt source “we mention before”.
  - Vector num into our SDK starts from zero not one as it was written into the datasheet “depend on the start-up code”.
  - So, our interrupt source number becomes the same number into datasheet decreased by one.
  - INT0 number becomes 1 not two, INT1 number becomes 2 not three, …. And so on.

Interrupt Vectors in ATmega32

Table 18. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

AMIT

External Interrupt Peripheral:

Naming of interrupt service routine:

- **ISR function name into out SDK:**
  - void __vector_verctorNum (void);
    - vectorNum is the interrupt source number into the vector table.
    - So, the function of INT0 will be:
      - void __vector_1 (void);
    - because this function has no prototype or software caller “because it is called by hardware event”, the compiler may ignore it.
    - To avoid ignoration, we must write a command to the compiler that this function is a hardware signal not a software caller.
    - So, the next line must be written before the implementation:
      - void __vector_1 (void) __attribute__((signal));

```
void __vector_1 (void) {
    //function body;
}
```

AMIT

External Interrupt Peripheral:

Naming of interrupt service routine:

- **ISR function definition:**
  - Because the function has a long name, we will create a library for MCAL only contains the definition of ISR and vector numbers, Like:

    - #define ISR(vectNum) void vectNum (void) __attribute__((signal));\
      void vectNum (void)
    - #define VECT_INT0 __vector_1
    - #define VECT_INT1 __vector_2
    - And so on, complete all table

  - Now, include this library into program.c file, and the function name will be:
    - ISR(VECT_INT0) {
      //function body
      }

AMIT

External Interrupt Peripheral:

Abstraction Violation:

- The ISR function will be written into the program.c file for MCAL module, but the function which it will be execute into ISR depends on the application.

- The MCAL module is prevented for user's usage, so how to pass the application function to ISR exists into MCAL module?
  - Call back function is the best solution for this case.
  - Call back mechanism depends on passing the address of application function to MCAL module, then MCAL module save this address into its global variable, finally ISR function will dereference this address.
  - By dereferencing this address, the body of application function will be executed into the ISR function.

AMIT

External Interrupt Peripheral:

Pointer to function:

- **Syntax to create pointer to function:**
  - returnType (* pointerName) (argumentsType);
    - EX: point to void add (int x , int y) function:
      - Solution:

        void (* ptr) (int , int) = add;
    - EX: point to int add (void) function:
      - Solution:

        int (* ptr) (void) = add;
    - EX: point to int* add (int x , char y) function:
      - Solution:

        int* (* ptr) (int , char) = add;
    - You notice that the name of function did not need address operator “&”, because the name of function is the address of first instruction into it.

AMIT

External Interrupt Peripheral:

Call Back function implementation:

Application Layer	MCAL Layer

void toggle (void) ;
int main (void)
{
  EXTI_Init();
  EXTI_senceLevel();
  EXTI_CallBack(toggle);
}
 	
void toggle (void)
{
  //DIO_TogglePin();
}
	#include “interrupt.h”
	void (*Gptr)(void) = NULL;
	
	EXTI_CallBack(void(*ptr)(void))
	{
	  if (ptr != NULL)
	  Gptr = ptr;
	}
	ISR(VECT_INT0)
	{
	  if (Gptr != NULL)
	  Gptr();
	}

AMIT

GLOBAL INTERRUPT

AMIT

Global Interrupt Enable :

I-bit:

- I-bit into the SREG register is used to enable global interrupt “written by one” or disable global interrupt “written by zero”.

- I-bit exists into the PSW of atmega32, and it is called by SREG.
  SREG register exists into the register bank, but it has an address because this register is remapped.

- By default, or at the start of program, the GIE is disabled.

Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

AMIT

Global Interrupt Enable :

GIE Module:

- All interrupt sources depend on enabling the I-bit, so GIE must be an independency Module and has the functions to enable or disable it.
- GIE module is very simple till now, it has two functions to enable or disable it.
- Now, write the software Module of GIE:

AMIT

THANK YOU!