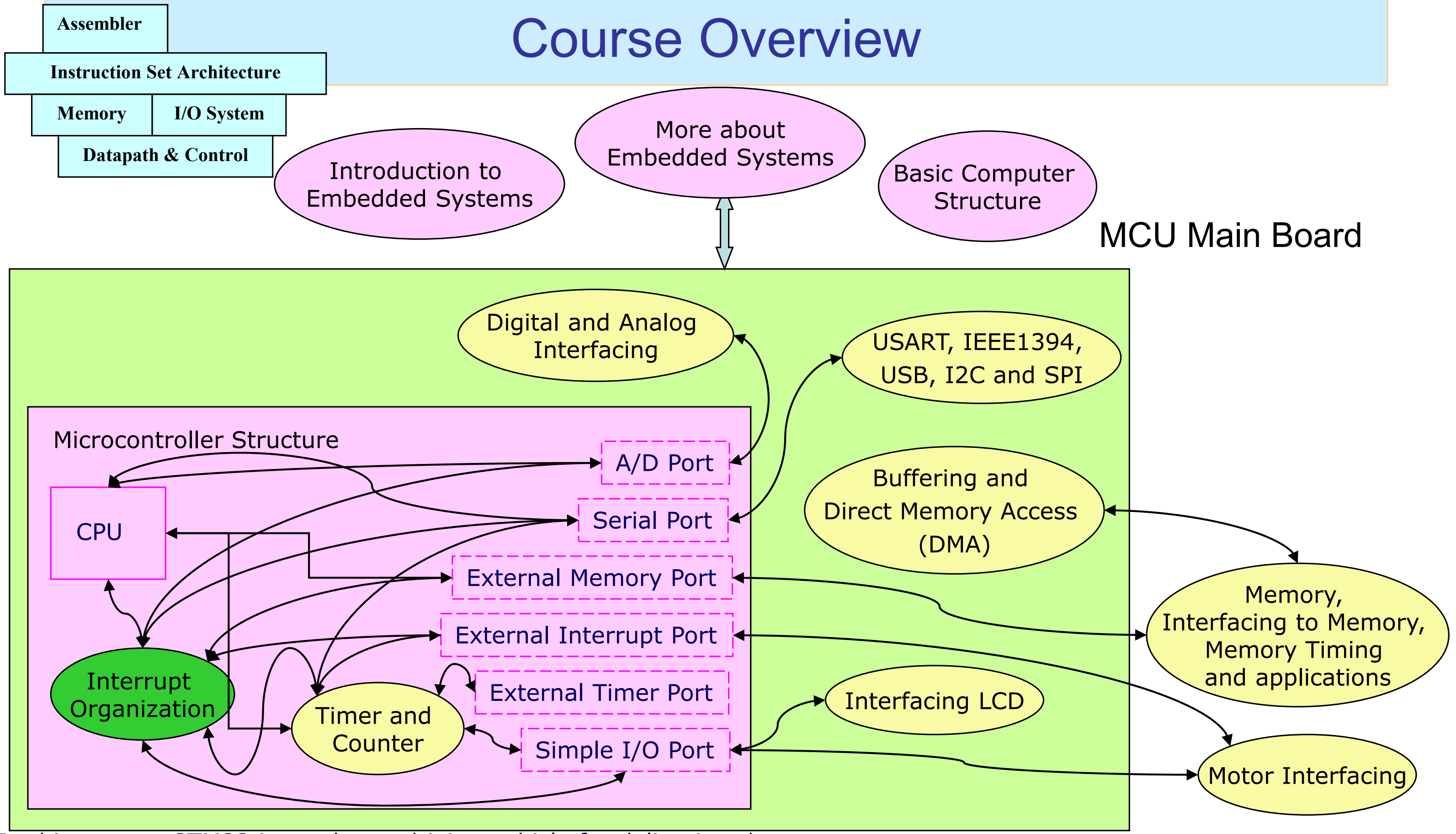
ELEC 3300
Introduction to Embedded Systems

Topic 6
Interrupt Organization

Prof. Vinod Prasad

Interruption of processor for other programs

Course Overview



Assembler
Instruction Set Architecture
Memory
I/O System
Datapath & Control
Introduction to Embedded Systems
More about Embedded Systems
Basic Computer Structure
MCU Main Board
Digital and Analog Interfacing
USART, IEEE1394, USB, I2C and SPI
Buffering and Direct Memory Access (DMA)
Memory, Interfacing to Memory, Memory Timing and applications
Interfacing LCD
Motor Interfacing
Microcontroller Structure
CPU
A/D Port
Serial Port
External Memory Port
External Interrupt Port
External Timer Port
Simple I/O Port
Interrupt Organization
Timer and Counter

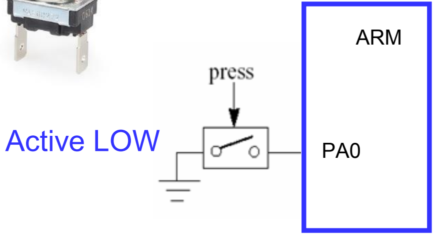
In this course, STM32 is used as a driving vehicle for delivering the concepts.

To be covered	In progress	Done

Expected Outcomes

- On successful completion of this topic, you will be able to:
  - Understand the TWO mechanisms (Polling and Interrupt) which control I/O activity whereby a device that needs CPU's attention to serve its request.
  - Compare polling I/O with interrupt-driven I/O.
  - Understand the timing diagram of interrupt service routine.
  - Evaluate several issues related to:
    - Identification of which device caused the interrupt.
    - Nested and simultaneous interrupts.
  - Understand the interrupt organization in ARM micro-controller.
  - Analyze the factors that need to be considered while deciding on polling or interrupt during a practical implementation.

A switch is connected to MCU board

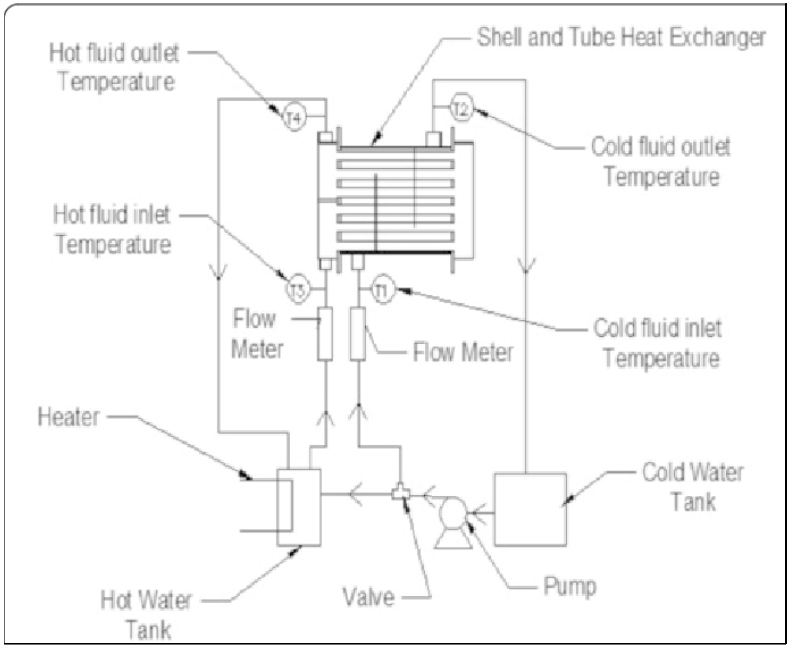


Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

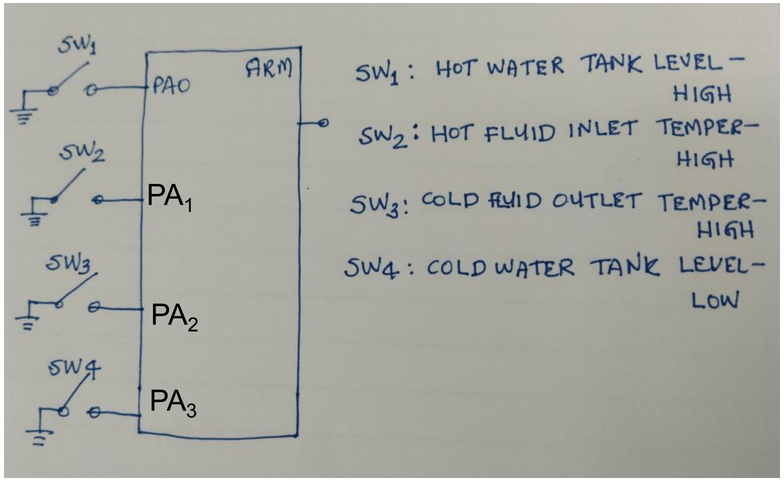
Physical Devices	Pin Assignment	Signal Type	Initialization (Configuration)	Signals at Physical connection
Micro Switch	General Purpose Input & Output	Input / ~~Output~~	General Purpose IO setting	On/Off

A Practical Industry Scenario

Heat Exchanger


Hot fluid outlet Temperature
Shell and Tube Heat Exchanger
T4
T2
Cold fluid outlet Temperature
Hot fluid inlet Temperature
T3
T1
Flow Meter
Flow Meter
Cold fluid inlet Temperature
Heater
Cold Water Tank
Hot Water Tank
Valve
Pump

Multiple devices (SW_1 to SW_4) connected to CPU for servicing.



SW_1 : HOT WATER TANK LEVEL - HIGH

SW_2 : HOT FLUID INLET TEMPER- HIGH

SW_3 : COLD FLUID OUTLET TEMPER- HIGH

SW_4 : COLD WATER TANK LEVEL- LOW

How do we address the needs of multiple devices?
Which device should be serviced first? What are the schemes?

Polling and

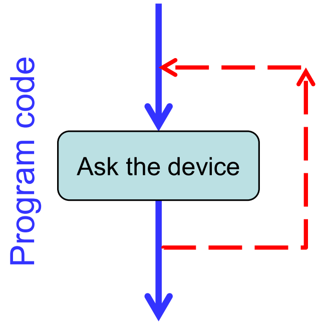
Interrupts

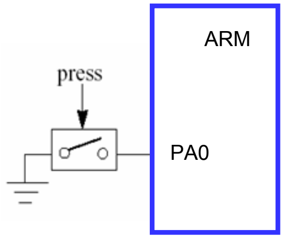
$\frac{300}{15}$

Polling

- Polled I/O requires the CPU to ask a device if the device requires servicing
  - For example, if the devices have changed status
  - Software polls the devices to know when a device will be serviced

- Polling requires code to loop until the device is ready
  - Consumes a lot of CPU cycles
  - Have a guaranteed delay but some applications do not have a predictable delay (e.g. In loading a printer buffer with text, the buffer can run out before the printer is polled again and thus printing stops)

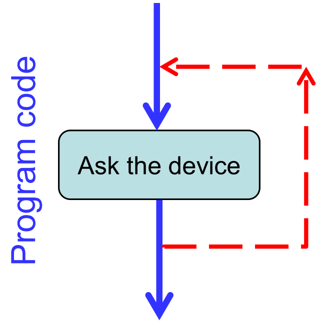




Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

Writing a polling algorithm (Single device)

- Initialization
  - Configure a GPIO as an appropriate function (say, input or output port)
- Implementation
  - Write a while-loop / if-then-else loop for checking the status of the GPIO



Initialization

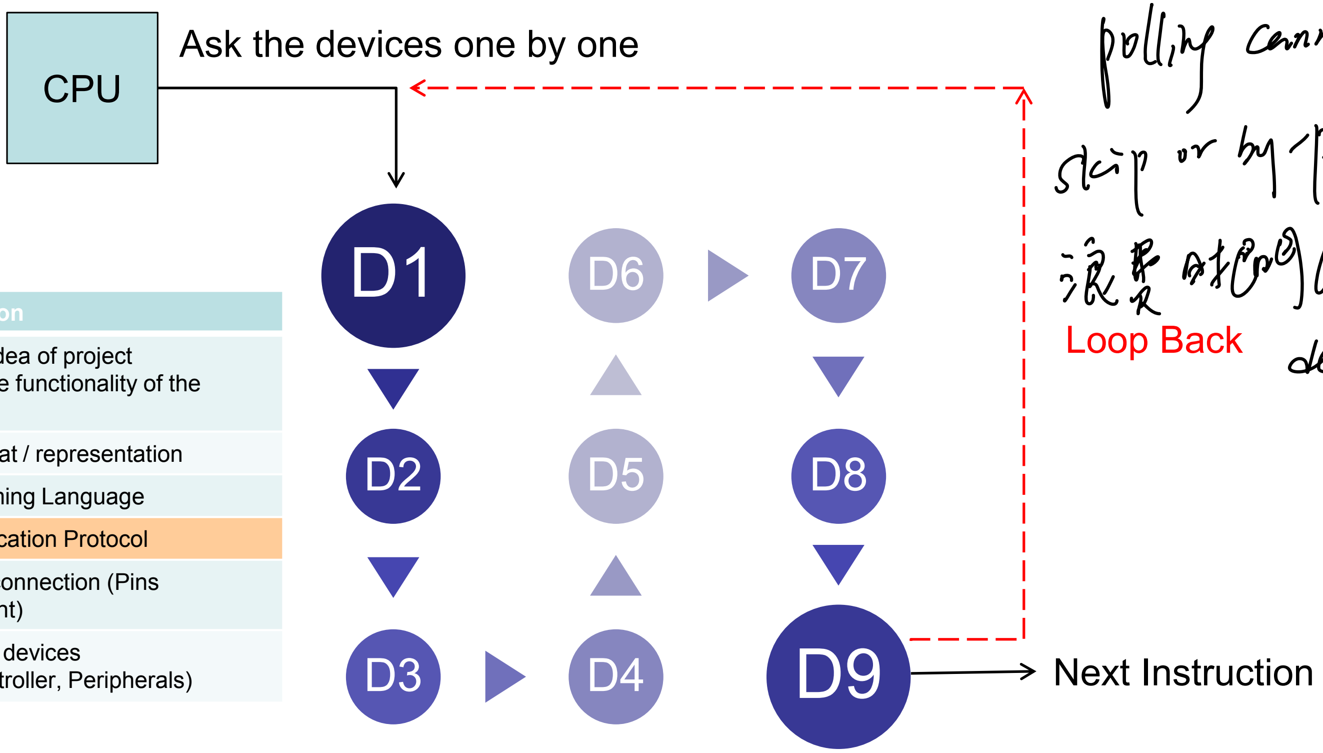
```
main()
{
        Configure PA0 as input port
        while()
        {
                Check if PA0 is pressed,
                        if yes, …. ; break
                        if not, ….  ; back to while loop
        }
}
```

implementation

Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

Polling in multiple devices

- Polled I/O requires the CPU to ask devices if the devices require servicing

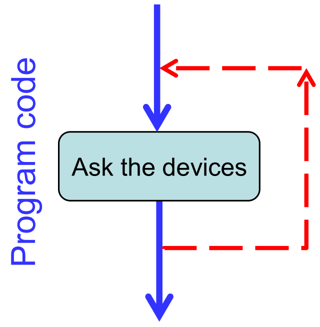


polling cannot skip or by-pass 浪费时间 long delay

Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

Writing a polling algorithm (Multiple devices)

- Initialization
  - Configure a GPIO as an appropriate function (say, input or output port)
- Implementation
  - Write a while-loop / if-then-else loop for checking the status of the GPIO



Initialization

implementation

```
main()
{
        Configure PA0, PA1, …., as input ports *
        while()
        {
                Check if PA0 is pressed,
                        if yes, …. ; break
                Else Check if PA1 is pressed,
                        if yes, …..; break
                Otherwise, …… ; back to while loop
        }
}
```

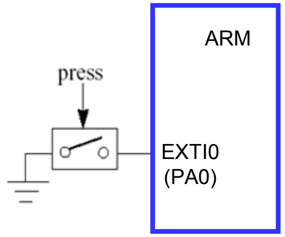
* remark

Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

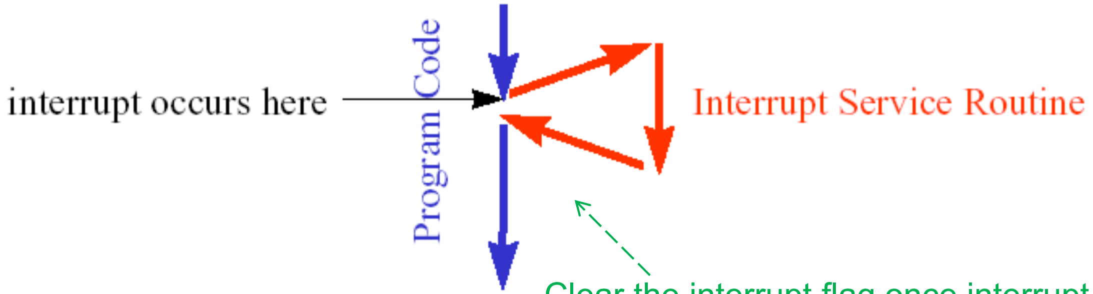
Tradeoff: Programming is easy, Latency is large.

Interrupts

- Efficient alternative to Polling.
- Interrupt I/O allows the device to interrupt the CPU announcing that the device requires attention
  - This allows CPU to ignore devices unless they request servicing via interrupt
- Interrupts do not require code to loop until the device is ready
  - Device interrupts CPU when it needs attention
  - Code can go off to do other things



EXTI0 - External Interrupt

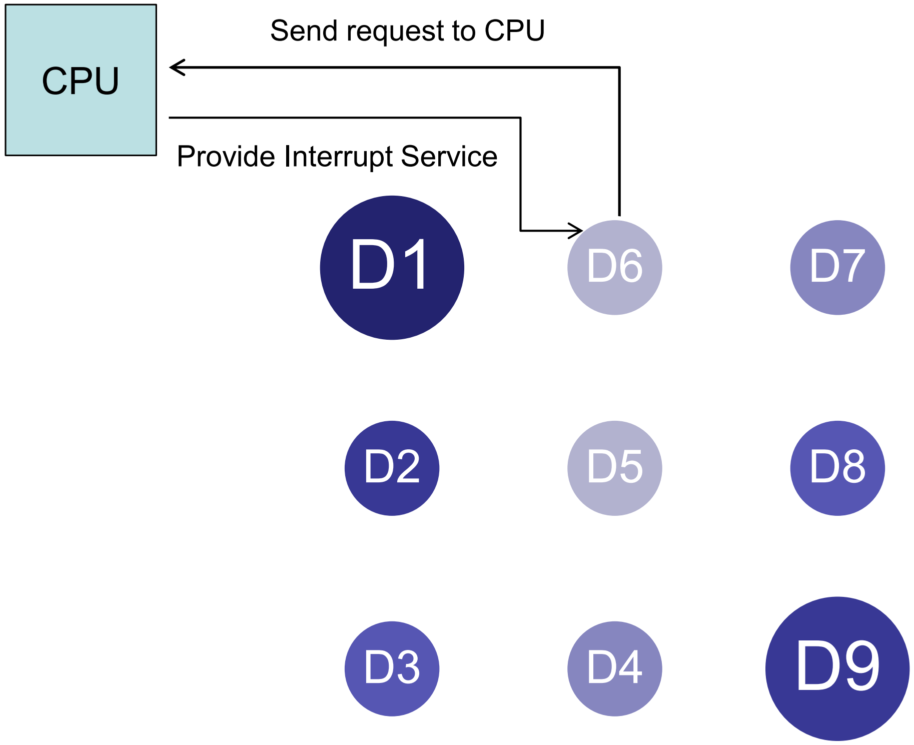


Clear the interrupt flag once interrupt service routine is over.

Disadvantage: Software does not know when an interrupt will occur, and this makes it more difficult to write code

Interrupts

Interrupt I/O allows the device to interrupt the CPU announcing that the device requires attention



Description
Abstract idea of project (Define the functionality of the system)
Data format / representation
Programming Language
Communication Protocol
Physical connection (Pins assignment)
Hardware devices (Microcontroller, Peripherals)

Interrupt Service Routine (ISR)

For every interrupt, there is an interrupt service routine (ISR), or interrupt handler, which is the block of codes that executes the servicing.

When an interrupt occurs, the microcontroller runs the ISR.

For every interrupt, there is a fixed location in memory that holds the address of its ISR.

The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.

ISR is also called device driver in case of the hardware interrupts and called exception handler in the case of software interrupts.

Hardware and Software Interrupts

Hardware Interrupt

An electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral.

Example: When we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

Software Interrupt

Caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor.

Example: If the processor's ALU runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message.

What happens when Interrupt occur?

CPU saves current context (Program Counter, Registers, etc)

⇩

CPU grabs address of the interrupt service handler (routine) from a vector table (each interrupt has an index into the table)

⇩

CPU executes the interrupt service handler

⇩

Interrupt service handler does operations, clears flags etc

⇩

Interrupt service handler exits

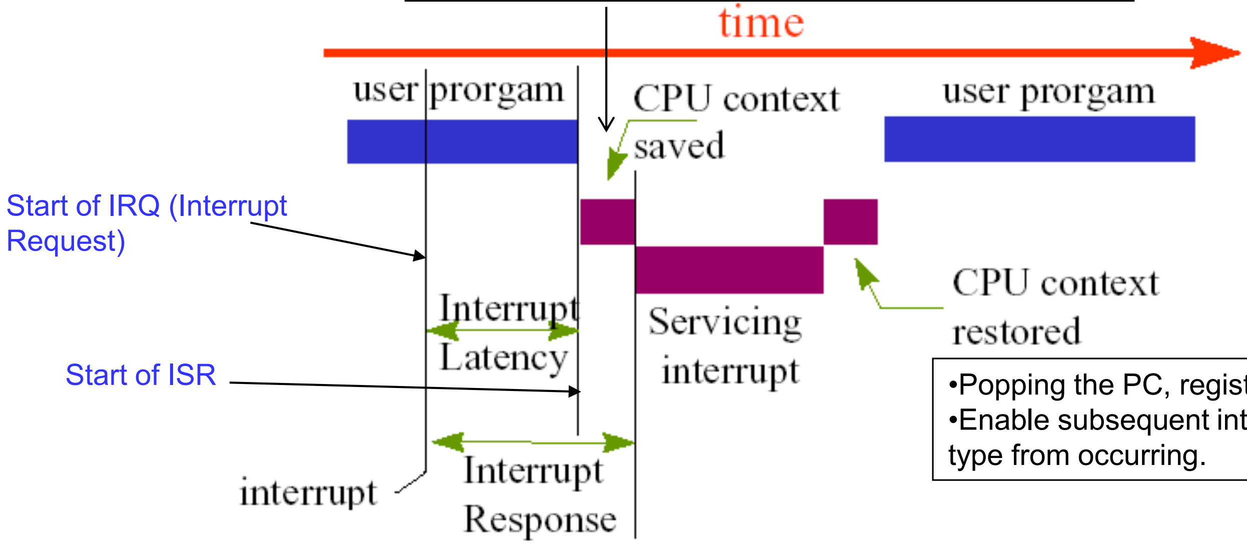
⇩

CPU restores context (Program Counter, Registers, etc) and returns to main flow

Timing diagram of Interrupts

- Before an Interrupt Service Routine (ISR) is executed, it must save away the current program’s registers (if it touches those registers)

•Pushing the PC, registers etc. onto the stack register
•Disable subsequent interrupts of the same type from occurring. However, interrupts of a higher priority can still occur.



•Popping the PC, registers etc onto the stack
•Enable subsequent interrupts of the same type from occurring.

What causes Interrupt Latency?

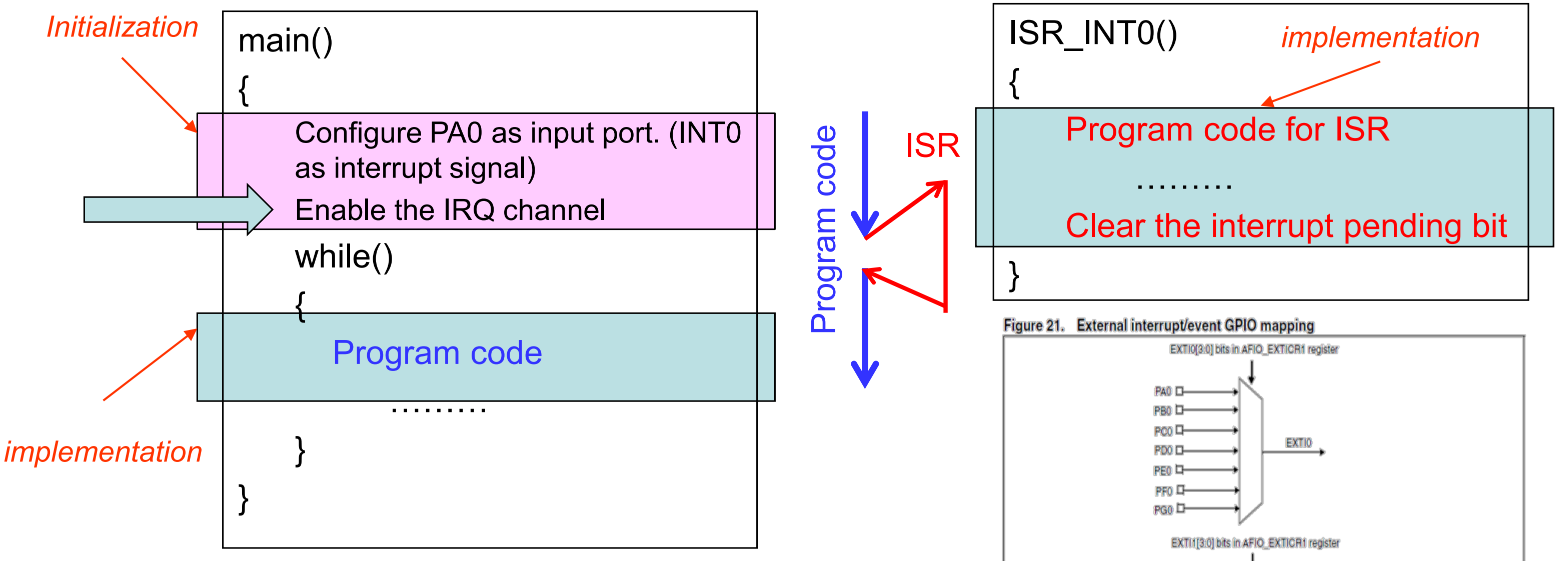
Interrupt latency: Delay between the start of an Interrupt Request (IRQ) and the start of the respective Interrupt Service Routine (ISR).

Reasons for interrupt latency:

- Processor has to complete the current instruction.
- Interrupts are disabled when the processor accesses critical OS data structures.
- If an ISR is already executing and cannot be interrupted, then this also increases the interrupt latency.

Writing an interrupt algorithm

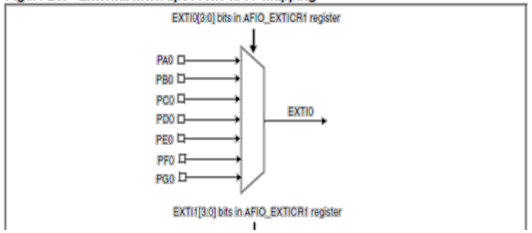
- Initialization
    - Select an appropriate pin (PA0) for handling interrupt signal
    - Write an ISR for handling this interrupt signal
    - Activate ISR by setting the corresponding Interrupt Request (IRQ) channel



```
main()
{
        Configure PA0 as input port. (INT0
        as interrupt signal)
        Enable the IRQ channel
        while()
        {
            Program code
            .........
        }
}
```

```
ISR_INT0()
{
        Program code for ISR
        .........
        Clear the interrupt pending bit
}
```

Figure 21. External interrupt/event GPIO mapping



EXTI0[3:0] bits in AFIO_EXTICR1 register
PA0
PB0
PC0
PD0
PE0
PF0
PG0
EXTI0
EXTI1[3:0] bits in AFIO_EXTICR1 register

Polling v/s Interrupt

You are going to bed at night and want to get up at 6 am the next day.

Polling: You <u>see the clock every few hours</u> to check whether it is time to wake up!

Interrupt: Sleep peacefully and wake up <u>when alarm rings</u>.

The most important reason why the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service.

In Interrupt method, microcontroller can serve many devices based on the priority assigned to it.

The polling method cannot assign priority because it checks all devices in a round-robin fashion.

Polling v/s Interrupt

	INTERRUPT	POLLING
1	The device notices the CPU that it requires its attention.	CPU steadily checks whether the device needs attention.
2	Not a protocol, it is a hardware mechanism.	It is a protocol, not a hardware mechanism.
3	The device is serviced by interrupt handler (ISR).	The device is serviced by CPU.
4	Interrupt can take place at any time.	CPU steadily ballots the device at regular or proper interval.
5	Interrupt request line is used as indication for indicating that device requires servicing.	Command ready bit is used as indication for indicating that device requires servicing.
6	Processor is evoked only when any device interrupts it.	Processor waste many processor cycles by repeatedly checking the command-ready little bit of each device.

Choosing between Polling and Interrupts

Polling must be done sufficiently fast to ensure that data is not lost.

I/O devices such as printers, keyboards, etc. require that the CPU execute some code to "service" the device. For example, incoming characters have to be read from a data register on the peripheral interface and stored in a buffer.

If a serial interface can receive up to 1000 characters/second and can only store the last character received, it must be checked at least once per millisecond to avoid losing data → Speed is a concern in polling.

Polling introduces a fixed overhead for each installed device.

Because we need to periodically check each device, irrespective of whether it requires service or not.

No such fixed overhead in Interrupts because ISR is executed ONLY when device needs attention.

Polling routines must be integrated into each and every program that will use that peripheral. Programs must be written to periodically poll and service all the peripherals they use.

Choosing between Polling and Interrupts

Is Interrupt a perfect method?

Interrupt Latency: Responding to an interrupt typically requires executing additional clock cycles to save the processor state, fetch the interrupt number and the corresponding interrupt vector, branch to the ISR and later restore the processor state.

Some factors to consider when deciding whether to use polling or interrupts include:

- Can the device generate interrupts?

  If the peripheral is very simple then it may not have been designed to generate interrupts.

- How complex is the application software?

  If the application is a complex program that would be difficult to modify in order to add periodic polls of the hardware then you may have to use interrupts.

  On the other hand, if the application is a controller that simply monitors some sensors and controls some actuators then polling may be the best approach.

Choosing between Polling and Interrupts

What is the maximum time allowed between polls?

If the device needs to be serviced with very little delay then it may not be practical to use polling.

What fraction of polls are expected to result in data transfer?

If the rate at which the device is polled is much higher than the average transfer rate, then a large fraction of polls will be “wasted”.

Using interrupts will reduce this polling overhead.

General Guideline (Take home Message): Use interrupts when the overhead due to polling would consume a large percentage of the processor time or would complicate the design of the software.

Choosing between Polling and Interrupts

Practical Case Study-1:

You are designing a steam boiler control system in which the steam pressure in the boiler is monitored by a pressure sensor. If the pressure suddenly rises to a dangerously high value, the processor must take rapid action to reduce the pressure. Assume that there are no other external devices interfaced with the processor that needs processor's attention/action. Would you use Polling or Interrupt?

Considerations:

Are there many devices connected to the processor?

If yes, polling many devices will consume time, may cause delay in attending the emergency.

If no, polling would be a suitable choice because immediate attention of the processor is possible.

What if we use Interrupt in this case?

Interrupt latency is a concern.

Choosing between Polling and Interrupts

Practical Case Study-2:

In a general purpose microcomputer, a range of devices are connected which include switches, keyboards, printers, modems, etc. Some of the devices operate at low speed (such as a manually operated switch), while others operate at higher speeds. What would be your choice – Polling or Interrupt?

Considerations:

Are there many devices connected to the processor?

What is the action response time (speed) expected by the devices – Some demands immediate response whereas others can wait?

Do we need to define priority for different devices?

Interrupt would be a better choice.

What if we use Polling in this case?

System will fail to satisfy the demand of critical devices (that need fast response).

No provision for setting up priority.

Interrupts: Issues to be dealt with

- Enabling/disabling interrupt (all or some)
  - Can be enabled / disabled by hardware or software
  - Those interrupts that can be enabled / disabled by programmer are called Maskable Interrupts.
  - Non-Maskable Interrupts cannot be disabled by programmer.
  - Examples of non-maskable interrupt: Internal system chipset errors, memory corruption problems, parity errors and high-level errors which need immediate attention.
  - Non-maskable interrupt is a way to prioritize certain signals within the operating system.

- Identification of which device caused the interrupt

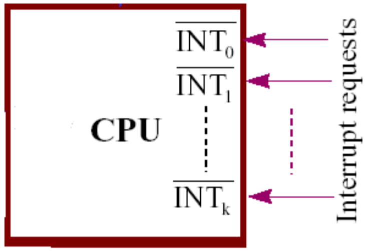
- Handling simultaneous interrupts

- Handling nested interrupts: A new interrupt to be processed, even before it finishes handling the current interrupt. This enables you to prioritize interrupts and make improvements to the latency of high priority events.

Identification of Interrupting Device

- The **problem** is: CPU has received an interrupt request, but which device caused the interrupt?

- **Solution**: One approach is simple polling of the status flags in each device's interface. In programming terms, there is a 'skip chain'.



Save program registers

Skip chain

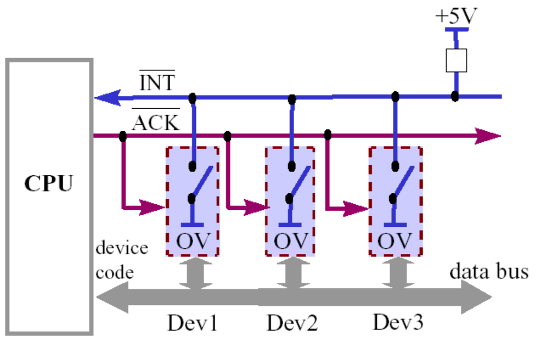
```
if (dev1 interrupted)
        go to dev1 handler
elseif (dev 2 interrupted)
        go to dev 2 handler
elseif (dev 3 interrupted)
        go to dev 3 handler
else    unidentified interrupt.
```

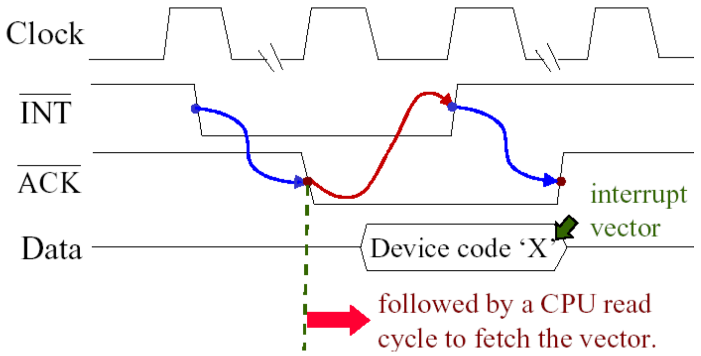
The CPU determines address of interrupt handler based on the type of interrupt. This is what 8051 μC does.

Problem with this approach is time wasted in 'skip-chain' processing, especially when there are many possible interrupting devices.

Identification of Interrupting Device

- Instead of programmed device identification, the device supplies the CPU with interrupt vector using an “Interrupt Acknowledge” bus cycle.
- Interrupt vectors are memory addresses which inform the CPU as to where to find the ISR (interrupt handler).





The interrupting device sends a self-identifying code back to the CPU during the “Interrupt ACK” cycle

Nested and Simultaneous Interrupts

- When there is the possibility of several devices interrupting
  - How to handle if a second interrupt request occurs while a previous interrupt is being serviced?
  - How to handle if two devices make interrupt requests during the same machine cycle?
- It is often the case that one device have priority over another (e.g. real-time clock v/s printer)
- ***Device A*** should be allowed to interrupt the serving of ***Device B*** if only if **A's** priority is higher than ***B's.***
- Similarly, if **A** and **B** interrupt simultaneously, **A** should be serviced first.

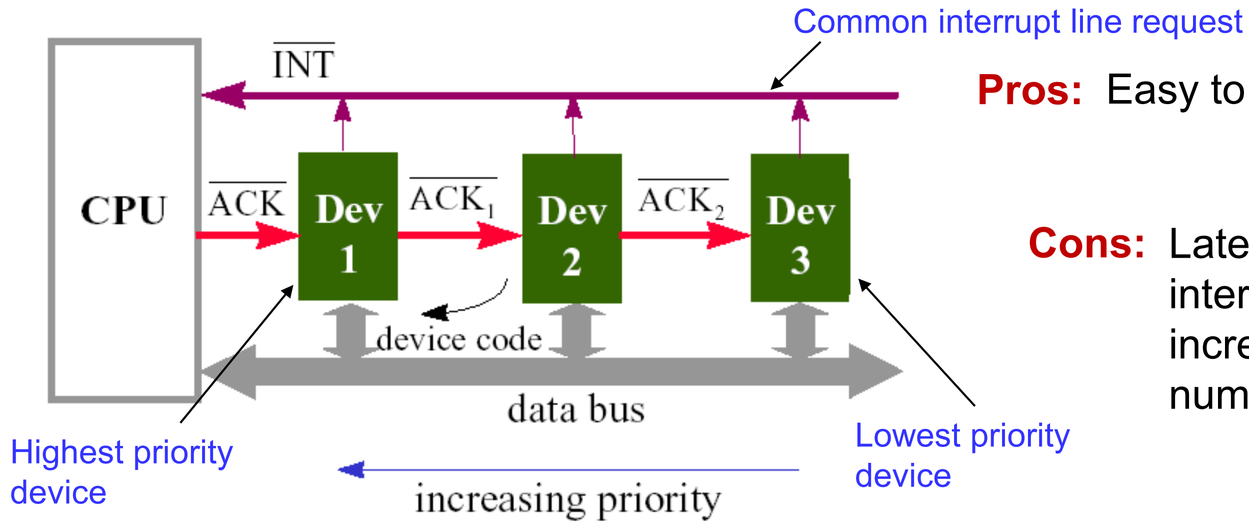
- On deciding the interrupt priority, there are two main approaches:
  - Daisy-chaining of the acknowledge line (simplest)
  - Use of a specialized Priority Interrupt Controller

Daisy-Chaining

Serial connection of all the devices which generate an interrupt signal. The interrupt line request is common to all devices.

The device with the highest priority is placed at the first position followed by lower priority devices (in decreasing order of priority).

The CPU responds to the interrupt by enabling the interrupt acknowledge line. This signal is received by the Device-1. The acknowledge signal passes to Device-2 only if Device-1 is not requesting an interrupt.

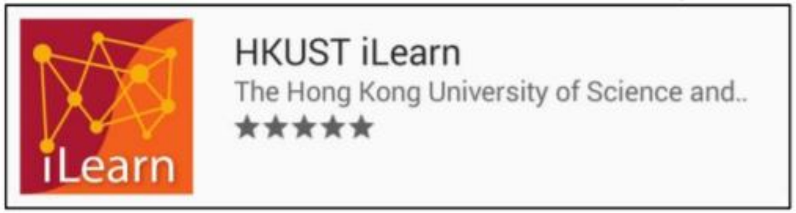


Pros: Easy to program

Cons: Latency to address interrupt request increases with the number of devices.

In-class activity

For Android devices, search **HKUST iLearn** at Play Store.



HKUST iLearn

The Hong Kong University of Science and..

For iOS devices, search **HKUST iLearn** at App Store.

HKUST iLearn

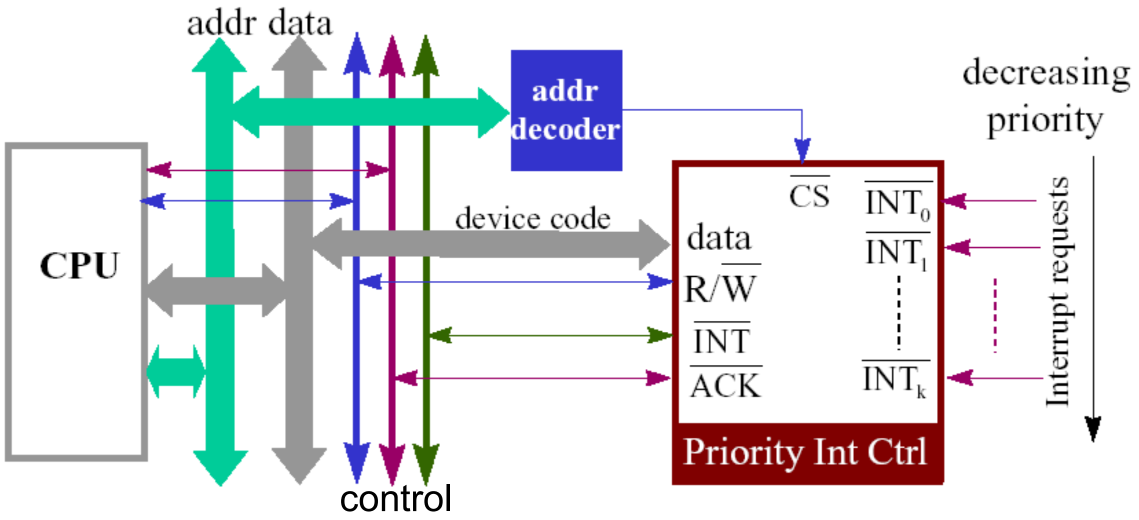
The Hong Kong University of Sc...

Interrupts – Topic 6: Questions 1 - 4

Priority Interrupt Controller

A priority interrupt controller (PIC) is used to place interrupt requests into a hierarchy.

Instead of the device communicating directly with the CPU, the PIC will communicate with the CPU after checking the interrupts (PIC is like an outsourcing agent – makes CPU free from interrupt checking procedures).



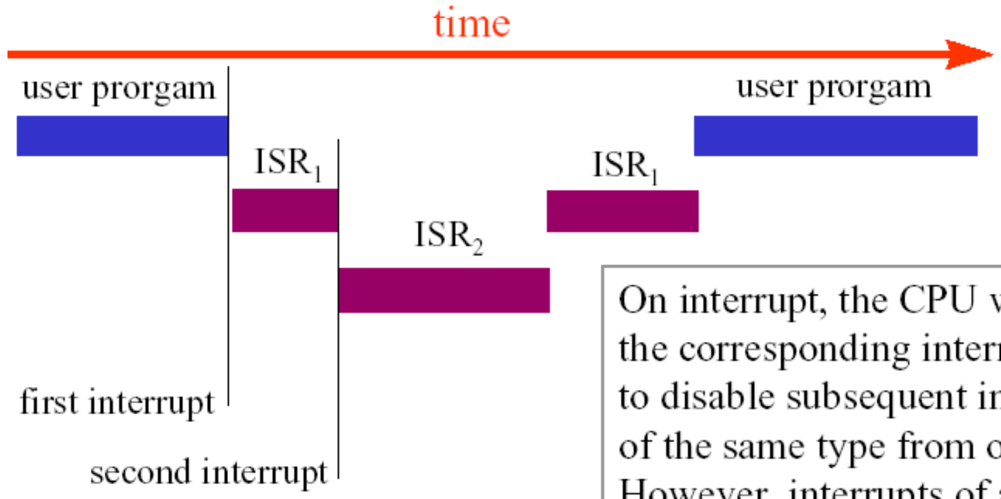
CONS:

- Extra chip.
- Cost.
- Power and Space.

PIC suitable for large number of priority interrupts.

Nested Interrupts

- Interrupts can occur within interrupts Which has higher priority – ISR_1 or ISR_2?



On interrupt, the CPU will clear the corresponding interrupt bit to disable subsequent interrupts of the same type from occuring. However, interrupts of a higher priority can still occur.

What will be the timing diagram if ISR_1 has higher priority over ISR_2?

What will be the timing diagram if ISR_3 that has a higher priority over ISR_2 occurs when ISR_2 is being executed?

STM32 Interrupts

- Nested vectored interrupt controller (NVIC)

Prioritizes interrupts, provides scheme for handling the interrupts that occur when CPU is processing a previously occurred interrupt, ensures that higher priority interrupts are serviced first.

- **Features:**

- 68 interrupt lines

- 16 programmable priority levels

- Low-latency expectation and interrupt handlin

- Power management control: System Control Registers used to control low-power features (e.g., sleep modes).

- **Reference:**

- Table 61 Vector table for connectivity line devices (intended for applications where connectivity and real-time performances are required - industrial control, control panels for security applications, ethernet)

- Table 62 Vector table for XL-density devices

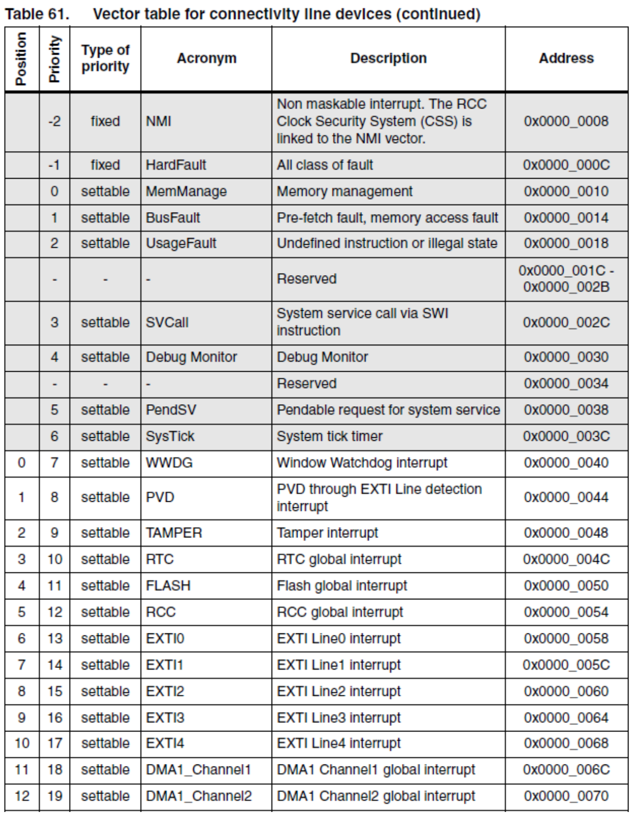


Table 61. Vector table for connectivity line devices (continued)

Position	Priority	Type of priority	Acronym	Description	Address
	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
	-1	fixed	HardFault	All class of fault	0x0000_000C
	0	settable	MemManage	Memory management	0x0000_0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000_0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
	-	-	-	Reserved	0x0000_001C - 0x0000_002B
	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
	-	-	-	Reserved	0x0000_0034
	5	settable	PendSV	Pendable request for system service	0x0000_0038
	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000_0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000_0068
11	18	settable	DMA1_Channel1	DMA1 Channel1 global interrupt	0x0000_006C
12	19	settable	DMA1_Channel2	DMA1 Channel2 global interrupt	0x0000_0070

XL-density devices are STM32F101xx and STM32F103xx microcontrollers where density of Flash memory ranges between 768 Kbytes and 1 Mbyte.

STM32 External interrupt / event controller (EXTI)

Interrupts typically execute a few lines of code whereas **Events** do not necessarily execute code but can signal another peripheral to do something without processor intervention.

Events are normally handled synchronously: the program explicitly waits for an event to be serviced, whereas an interrupt can demand service at any time.

Event example: A timer can generate an event to tell an ADC to sample and then write the measured value to memory using DMA without ever waking up the processor.

- Features:
  - Up to 20 edge detectors (rising/falling edge in EXTI line) in connectivity line devices
  - Up to 19 edge detectors in other devices for general event / interrupt requests
  - Independent trigger and mask on each interrupt / event line
  - Dedicated status bit for each interrupt line
  - Generation of up to 20 software event / interrupt requests (e.g. Timer)

STM32 Interrupts

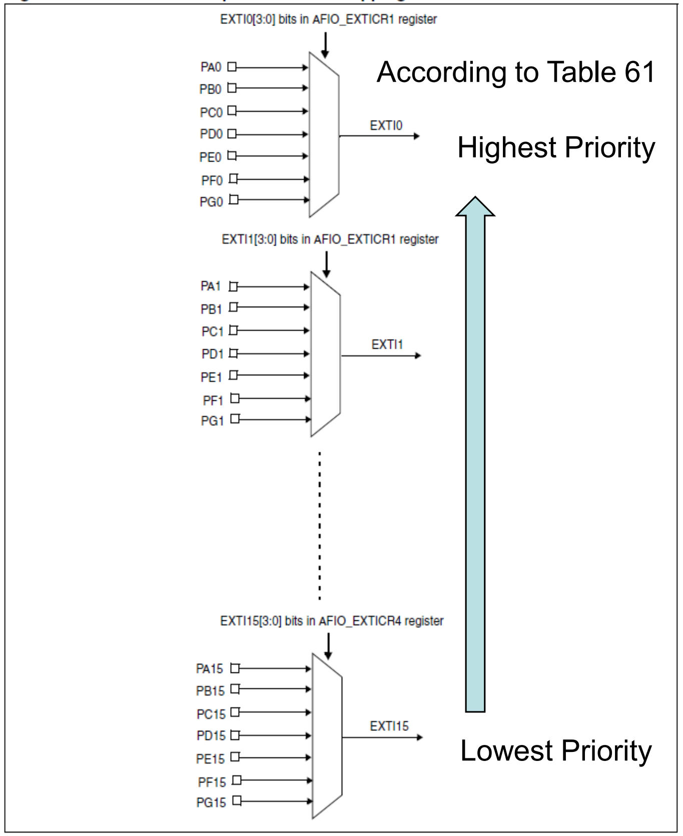
- The priority of external interrupts can be done by the hardware configuration.
- Example 1:
  - Device 1 is connected to PA1
  - Device 2 is connected to PB0

  Which one has a highest priority?
- Example 2:
  - Device 1 is connected to PA1
  - Device 3 is connected to PB1

  Which one has a highest priority?
- Example 3: How can we make Device 1 high priority compared to Device 2?

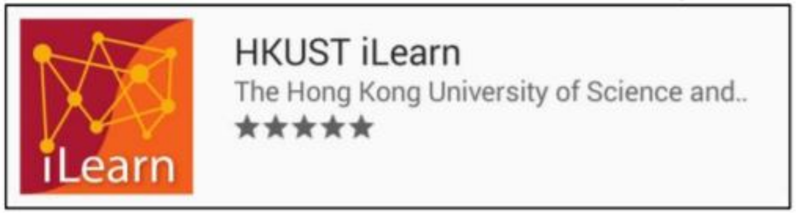
Use polling to check Device 1 first. In principle, hardware does not assign any priority, you have to do it in software.

Figure 21. External interrupt/event GPIO mapping



In-class activity

For Android devices, search **HKUST iLearn** at Play Store.



HKUST iLearn

The Hong Kong University of Science and..

★★★★★

For iOS devices, search **HKUST iLearn** at App Store.

HKUST iLearn

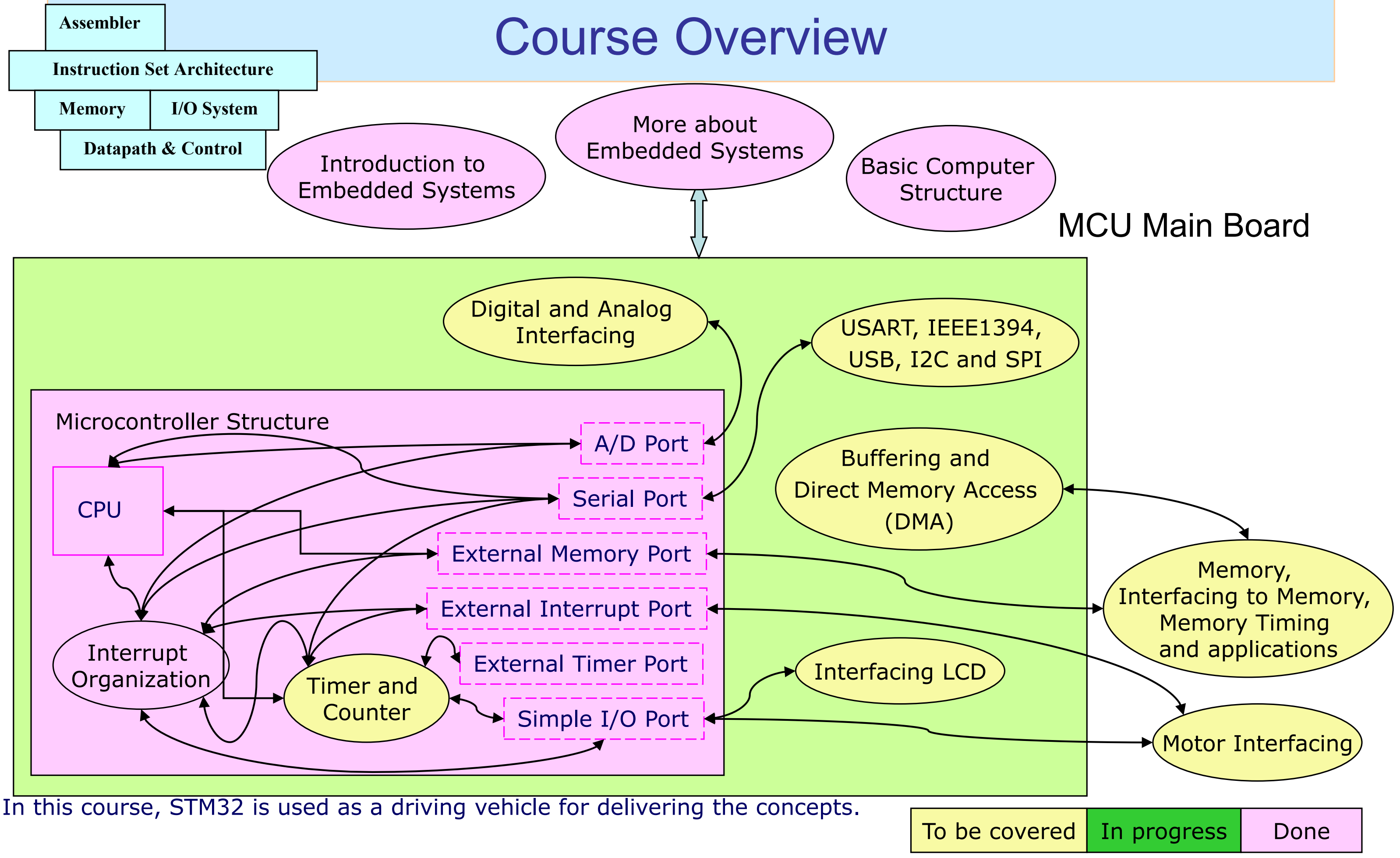
The Hong Kong University of Sc...

Topic 6: Questions 5 - 8

Reflection (Self-evaluation)

- Do you
  - Describe two I/O interfacing techniques: polling I/O and interrupt-driven I/O ?
  - Distinguish the concepts and signaling in polling and interrupt-driven I/O ?
  - List the timing diagram of interrupt service routines?
  - Handle several issues deal with interrupts ?
  - Configure the interrupt settings in ARM micro-controller ?

Course Overview



In this course, STM32 is used as a driving vehicle for delivering the concepts.