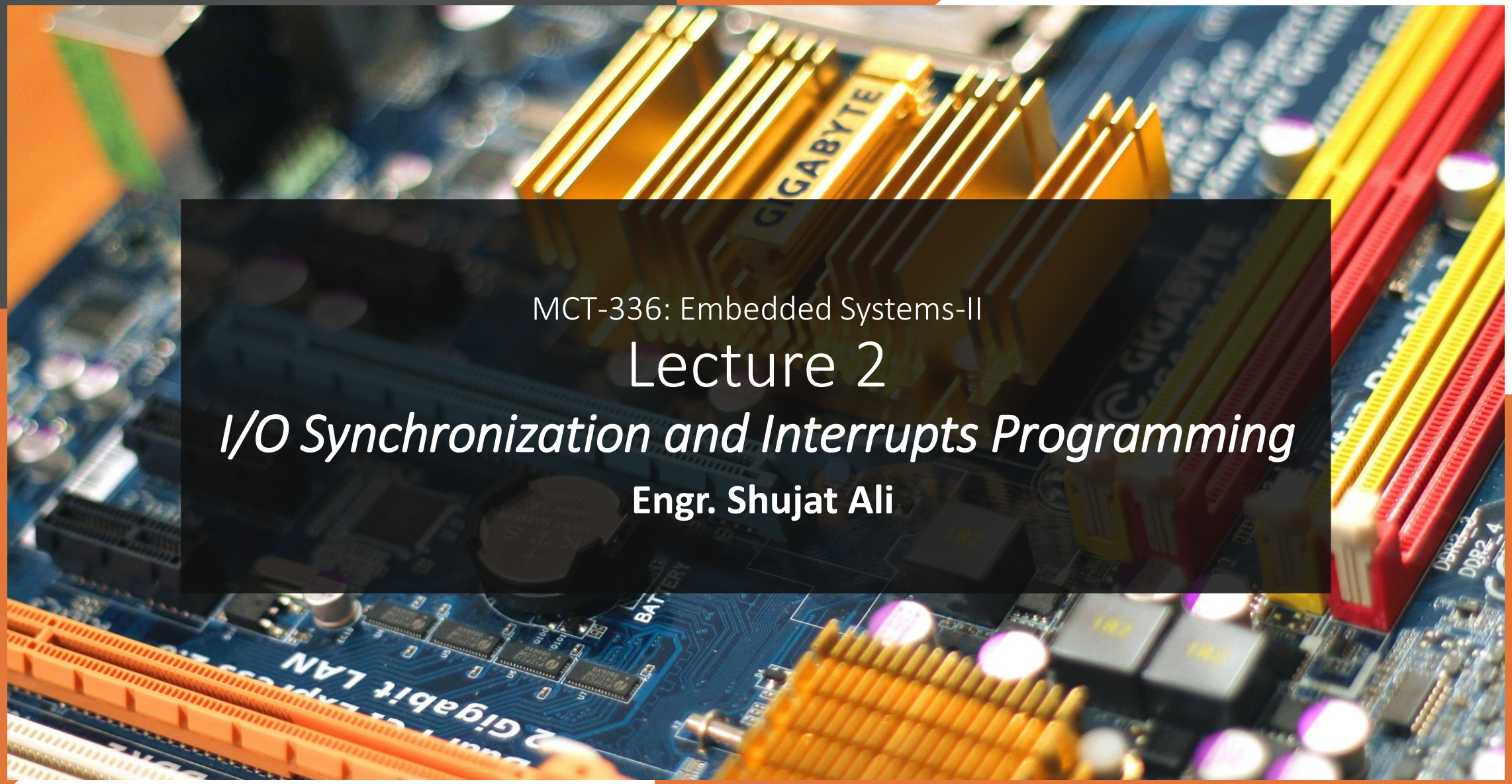
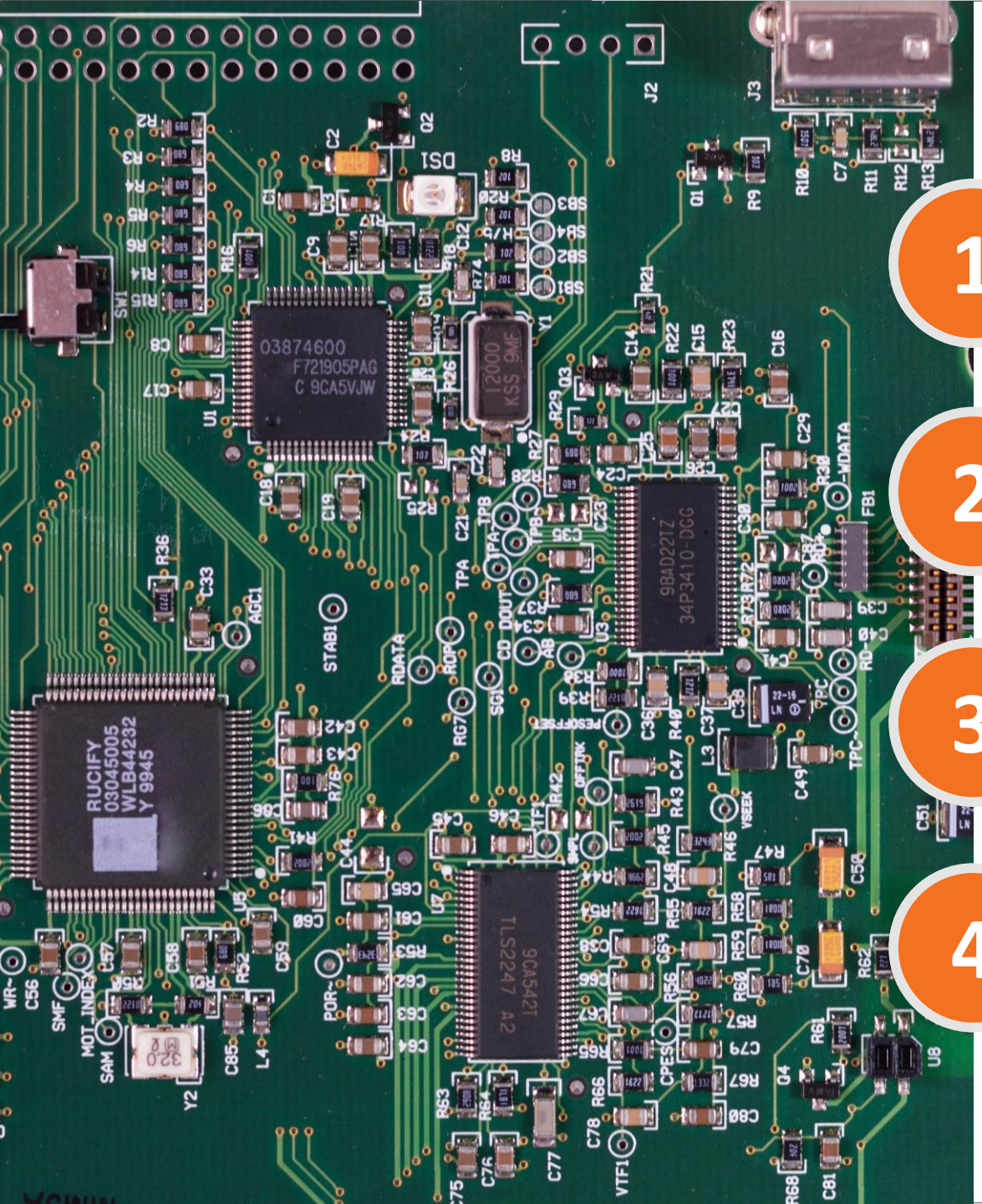


MCT-336: Embedded Systems-II

Lecture 2

I/O Synchronization and Interrupts Programming

Engr. Shujat Ali



1. INTRODUCTION TO I/O SYNCHRONIZATION
2. METHODS FOR I/O SYNCHRONIZATION
3. TYPES OF EXCEPTIONS OR INTERRUPTS
4. CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

INTRODUCTION TO I/O SYNCHRONIZATION

Peripheral Devices Operational States and State Transition, Input/Output Device Synchronization

Why is I/O Synchronization is Needed?

INTRODUCTION TO I/O SYNCHRONIZATION

- In an embedded system, the speed of I/O devices mismatches that of the microprocessor.
- Therefore, there is an inherent need for synchronization for an efficient embedded system
- In the absence of synchronization,
  - The processor is highly underutilized when the I/O is too slow
  - potential data loss in case of a too fast I/O device
- I/O synchronization ensures efficient communication between system software and peripheral hardware to prevent delays and data loss.

Input/Output Devices

INTRODUCTION TO I/O SYNCHRONIZATION

Based on data handling a microcontroller may have following categories of peripherals (or devices).

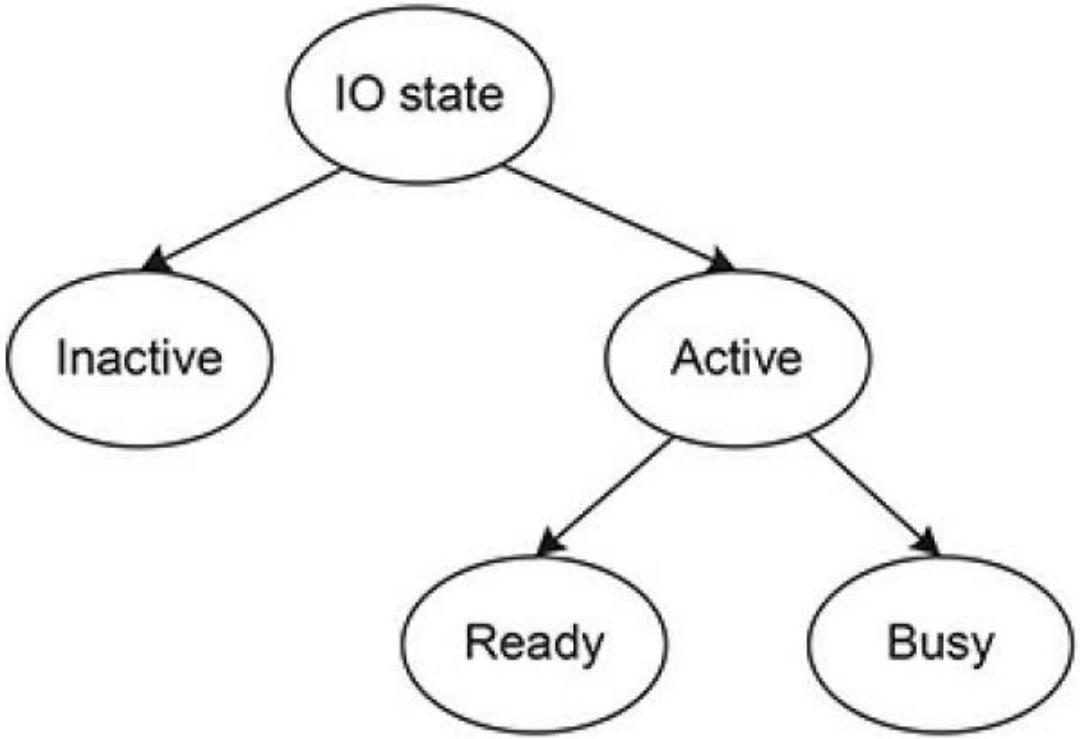
Input Device: A peripheral operates as an input device if it is used for data acquisition/reception, e.g., ADC, CCP of timer, GPIO Pin used as input pin, etc.

Output Device: A peripheral is termed as an output device if it is used for data generation/ transmission, e.g., DAC, PWM, GPIO Pin used as output pin, etc.

Communication interfaces (e.g., UART, USB, Ethernet, etc.) are used for data transmission as well as reception among different devices, therefore these are both input and output devices.

Peripheral Device Operation States

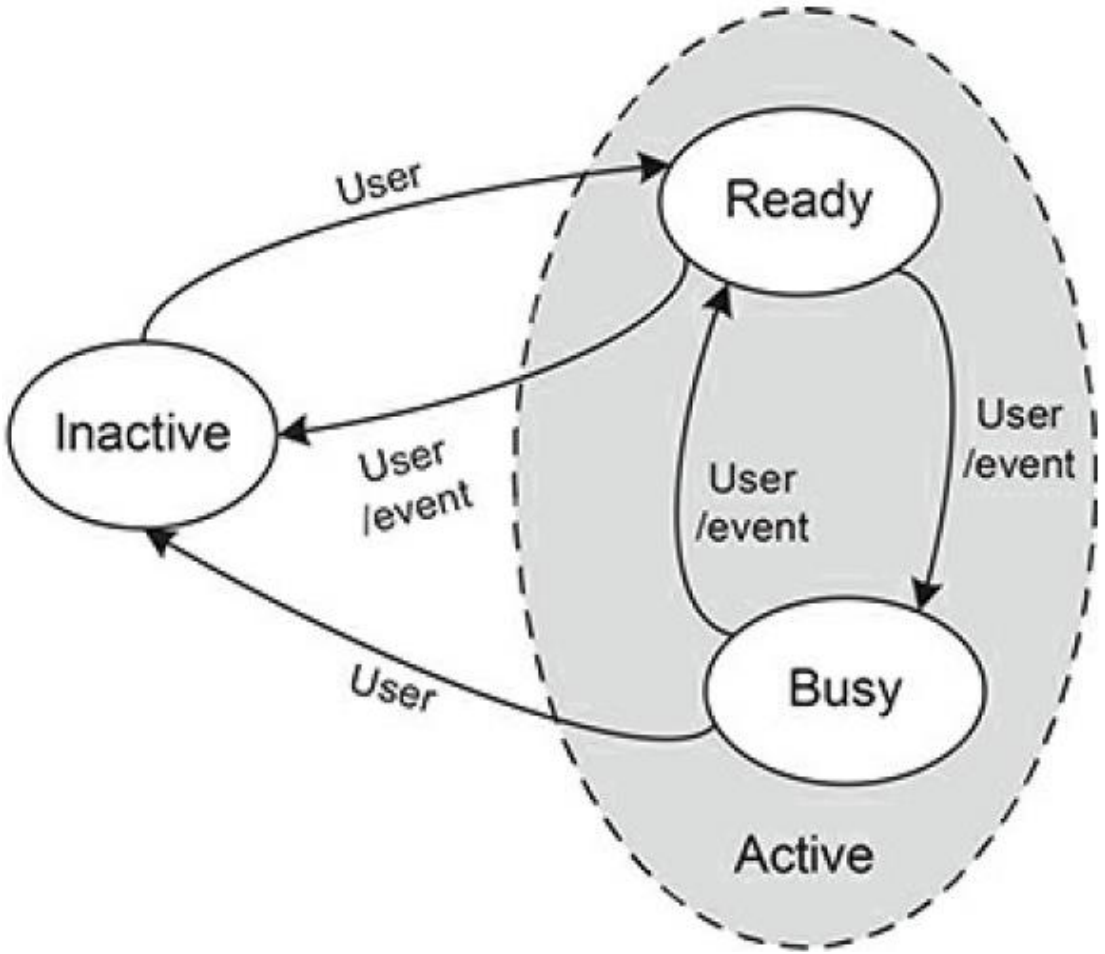
INTRODUCTION TO I/O SYNCHRONIZATION



- From an operational perspective, the hardware device can be in one of the two possible states
  1. ***Inactive state****:* No input-output operation can be performed by the device in an inactive state.
  2. ***Active state****:* When the device is active, it can be either in ***busy state*** or in ***ready state***.

Peripheral Device State Transition

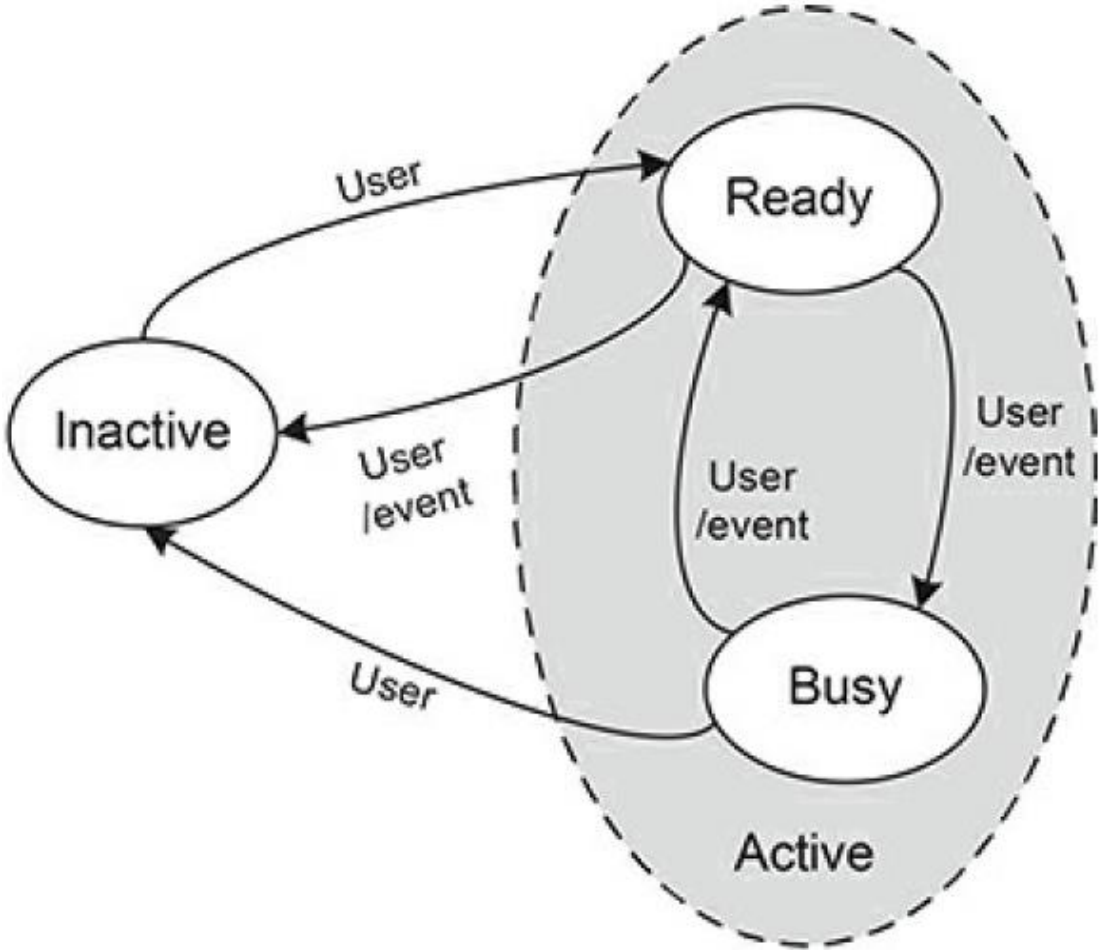
INTRODUCTION TO I/O SYNCHRONIZATION



- A transition from inactive to active state can be generated by the software (i.e., **user application program**)
- A transition from ready to busy state occurs either due to **user task assignment** or due to an **event**.
- The device remains in the busy state when performing the assigned task.
- When the device is finished with the current task a transition from busy to ready state occurs.
- The device remains in ready state, waiting for another task assignment.

Peripheral Device State Transition

INTRODUCTION TO I/O SYNCHRONIZATION

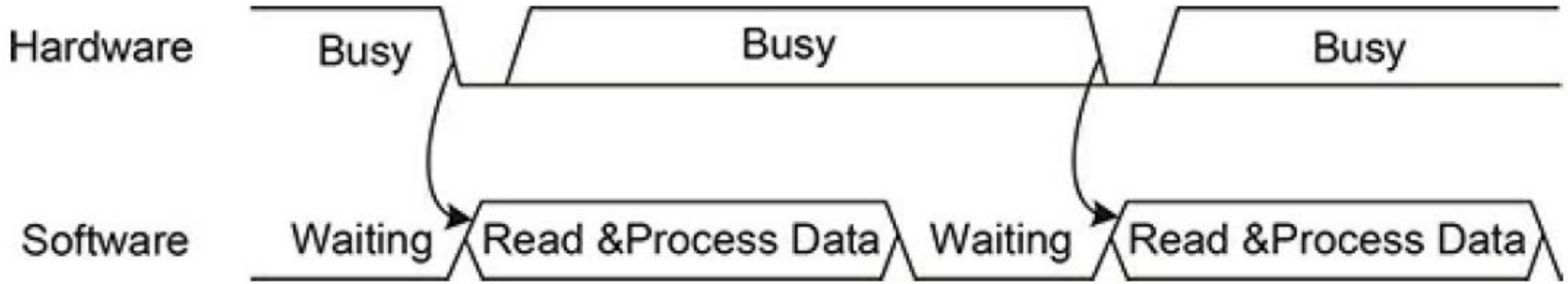


- A transition from ready to inactive state can be triggered either by the **user program** or it can happen due to **expiration of predefined inactivity time interval**.
- A state transition from busy state to inactive state can be triggered by the software (user program), which can lead to data loss.
- The state transitions are equally valid for both input as well as output modes of operation of a peripheral device.

Input Device Synchronization

INTRODUCTION TO I/O SYNCHRONIZATION

In the case of an **input device**, hardware is the **data producer**, while software is responsible for receiving and processing data and is labeled as a **data consumer**.



Once the hardware has produced data, the software reads it and allows the hardware device to start producing new data, while the software is busy processing that data.

If the time required by the hardware to produce data is higher than the time it takes the software to consume data, then the software needs to wait for the hardware, i.e., ***time loss***

The arrows from hardware to the software, in the timing diagram, mark the synchronizing instances.

Input Device Synchronization

INTRODUCTION TO I/O SYNCHRONIZATION

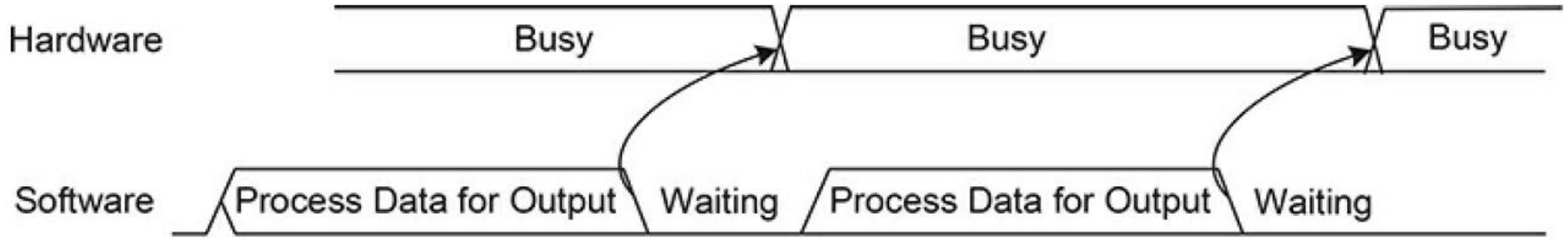
What if hardware produces data faster than the software can consume? There is always a possibility of ***data loss***. Different mechanisms, including buffering, hand shaking, etc. are used to avoid any data loss.

In an ideal scenario, the hardware and software data producing and consuming speeds should match exactly, which is highly unlikely in practical situation and some methodology for their synchronization is required.

Output Device Synchronization

INTRODUCTION TO I/O SYNCHRONIZATION

In the case of an output device, hardware is the **data consumer**, while software is responsible for processing and producing data, i.e., **data producer**.



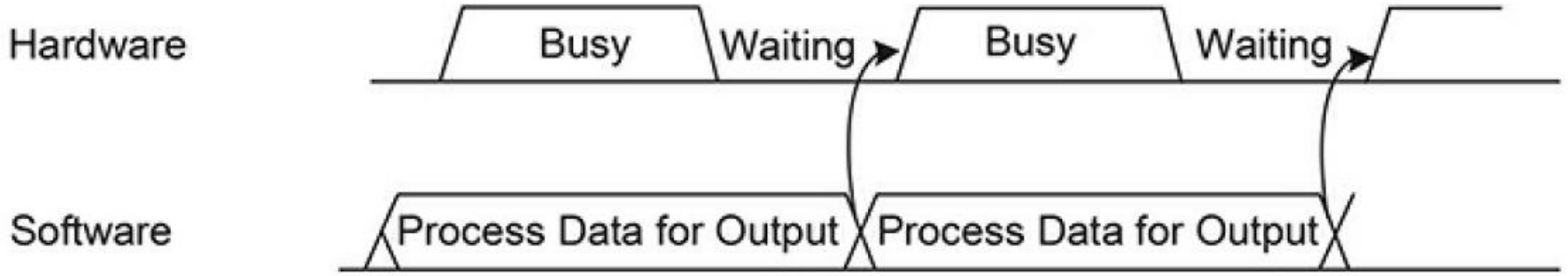
If the time taken by the software to produce data is less than the time required by the hardware to consume that data (i.e., to transmit or generate corresponding signal for that data) then software needs to wait for synchronization.

The synchronizing instances are marked by the arrows from the software to the hardware.

Output Device Synchronization

INTRODUCTION TO I/O SYNCHRONIZATION

In case of an output device, hardware is the **data consumer**, while software is responsible for processing and producing data, i.e., **data producer**.



The case, when hardware is faster and needs to wait for the software. The synchronizing instances are marked by the arrows from the software to the hardware.

From the both cases, we can observe that for output device synchronization, the mismatch in the hardware and software speeds results in **time loss** only and unlike an input device, there is no possibility of **data loss**.

METHODS FOR I/O SYNCHRONIZATION

Performance Evaluation Parameters,

Blind Cycle, Polling-based Methods, Interrupt-driven Methods

Performance Evaluation Parameters

METHODS FOR I/O SYNCHRONIZATION

Following are the important parameters that are used to quantify the performance of different synchronization approaches.

1. **Bandwidth** or **throughput** – the maximum data processing/transfer rate (measured in bytes/bits per second) that can be handled by the system. It is limited by the I/O device or the execution speed of the processor.
2. **Priority** – determines the order in which the processor processes the requests from different I/O devices, when two or more devices have made requests simultaneously. It also determines if high-priority request can preempt the low priority task.
3. **Latency** – the delay between time instance when the I/O device indicates that it needs to be processed till the time instance when the required processing is initiated.
   - For an **input device**, the latency can be considered as the time elapsed between new data available till the time when software reads the data for processing.
   - For an **output device**, latency can be defined in terms of time interval from the instance the output device becomes idle till the software (processor) provides new data to the output device

Types Of Different I/O Synchronization Methods

METHODS FOR I/O SYNCHRONIZATION

Different I/O synchronization methods are grouped into the following 3 main categories.

1. Blind cycle
2. Polling based methods
   - Continuous polling or busy waiting
   - Periodic polling
3. Interrupt-driven methods
   - I/O synchronization using Interrupts
   - Direct memory access

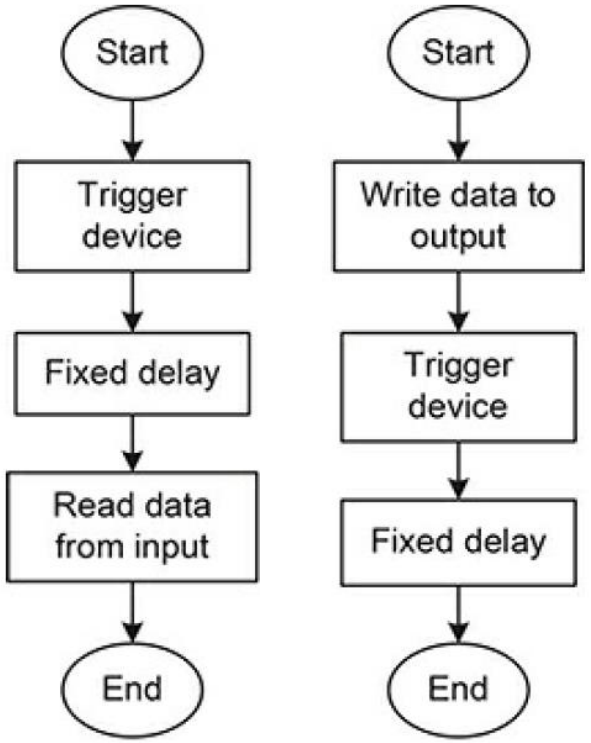
Let’s discuss these methods and their pros/cons.

Methods For I/O Synchronization

1: **Blind Cycle**

2: Polling-based Methods

3: Interrupt-driven Methods

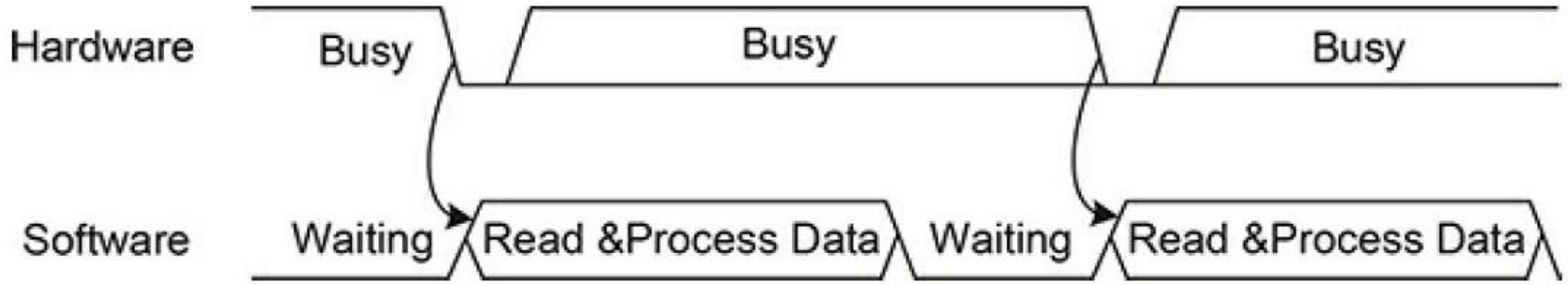


In this method, the software waits for a fixed time interval for the I/O device hardware to complete the operation

The choice of fixed delay depends on the maximum possible time that the device hardware may require to complete the operation.

This method is called blind cycle, as no information about the current state of the hardware is available to software

Useful for I/O operations where the device hardware speed (or wait delay) is highly predictable.



Methods For I/O Synchronization

1: **Blind Cycle**

2: Polling-based Methods

3: Interrupt-driven Methods

Advantage:

The key advantage is that its implementation simplicity.

Disadvantages:

We take maximum time a hardware device may take to complete a task, and it incurs unnecessarily large delays resulting in wastage of processing time

The software and hardware operations are performed sequentially in case of blind cycle method resulting in poor system performance by wasting processing resource.

Methods For I/O Synchronization

1: *Blind Cycle*

2: **Polling-based Methods**

3: Interrupt-driven Methods

Large delays caused by the blind cycle method can be reduced by continuously or periodically checking the hardware's status.

This allows the software to avoid unnecessary delays and only wait for the needed time. This process of checking the hardware status is called **polling**.

Methods For I/O Synchronization

1: *Blind Cycle*

2: **Polling-based Methods**

- ❑ **Continuous Polling or Busy Wait**
- ❑ Periodic Polling

3: Interrupt-driven Methods

The **continuous polling** (also called **busy waiting**) is implemented by the software using a loop, where the status of the device hardware is continuously checked for operation completion.

As soon as the device hardware is done with the operation, the corresponding status is changed, and the software immediately knows about it and can initiate new operation.

Advantage:

We don't need to use un-necessary delays.

Disadvantage:

One major drawback of busy waiting is the possibility of blocking software execution due to some hardware malfunctioning, which halts the corresponding status information updating and as a result the software execution is blocked.

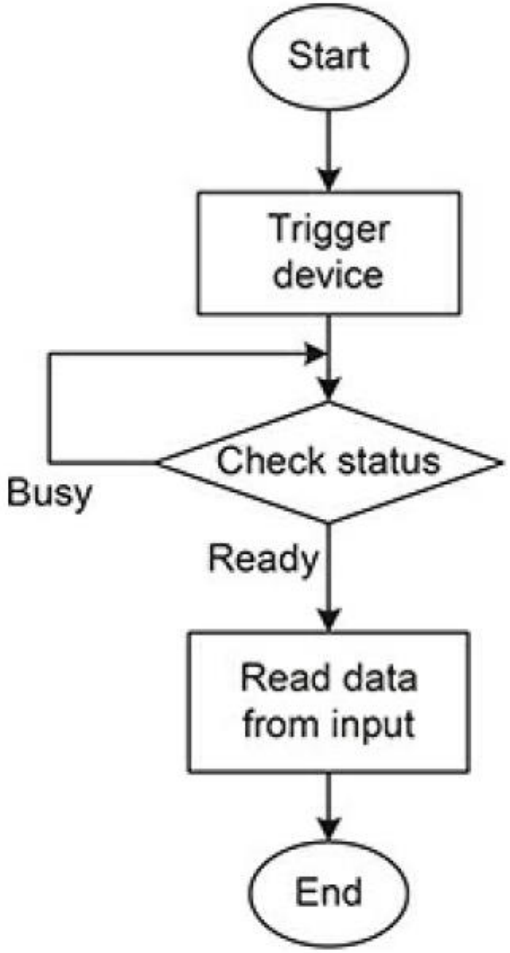
Methods For I/O Synchronization

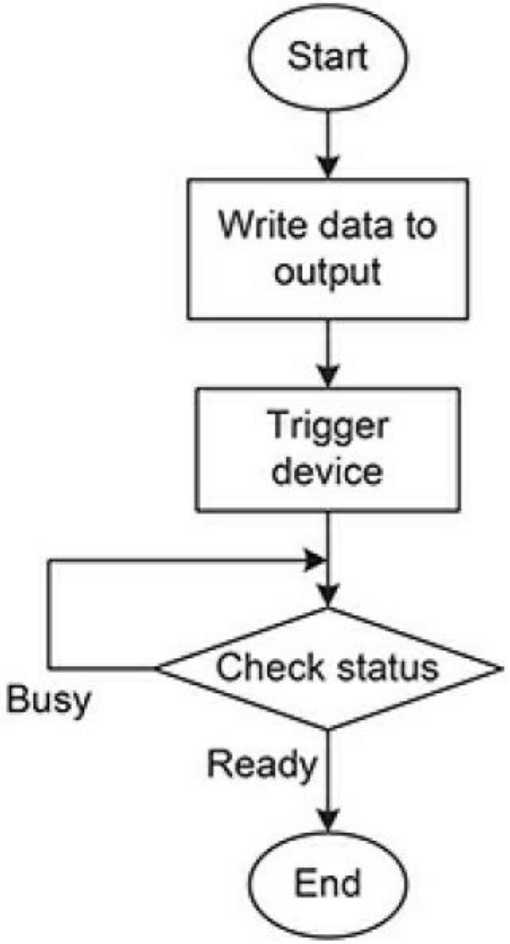
1: *Blind Cycle*

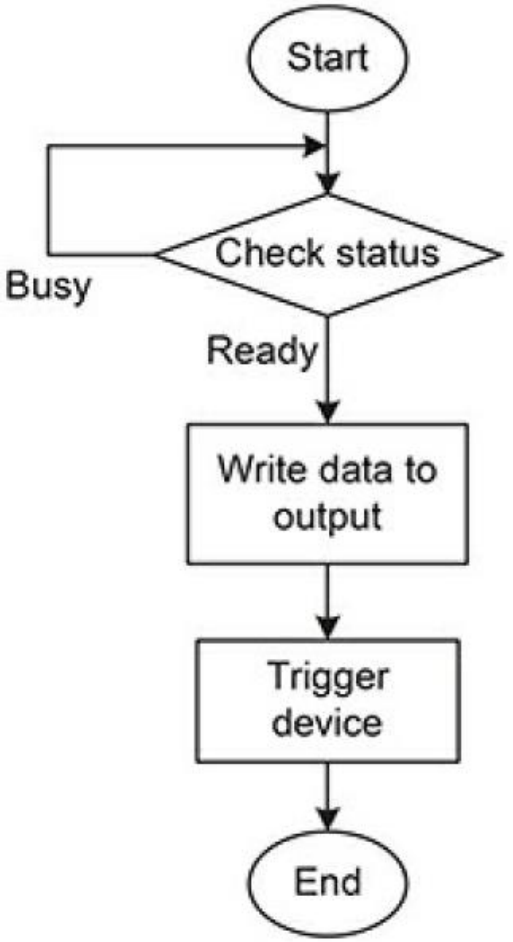
2: **Polling-based Methods**

- ❑ **Continuous Polling or Busy Wait**
- ❑ Periodic Polling

3: Interrupt-driven Methods







Methods For I/O Synchronization

1: *Blind Cycle*

2: **Polling-based Methods**

- *Continuous Polling or Busy Wait*
- **Periodic Polling**

3: Interrupt-driven Methods

The key disadvantage of continuous polling is the possibility of blocking the software execution which is resolved by using periodic polling.

In this method device status is checked periodically by using a timer interrupt

Limitation

Limitation of periodic polling is an extra delay, compared to continuous polling, that might occur in the I/O operation due to the timer interrupt latency

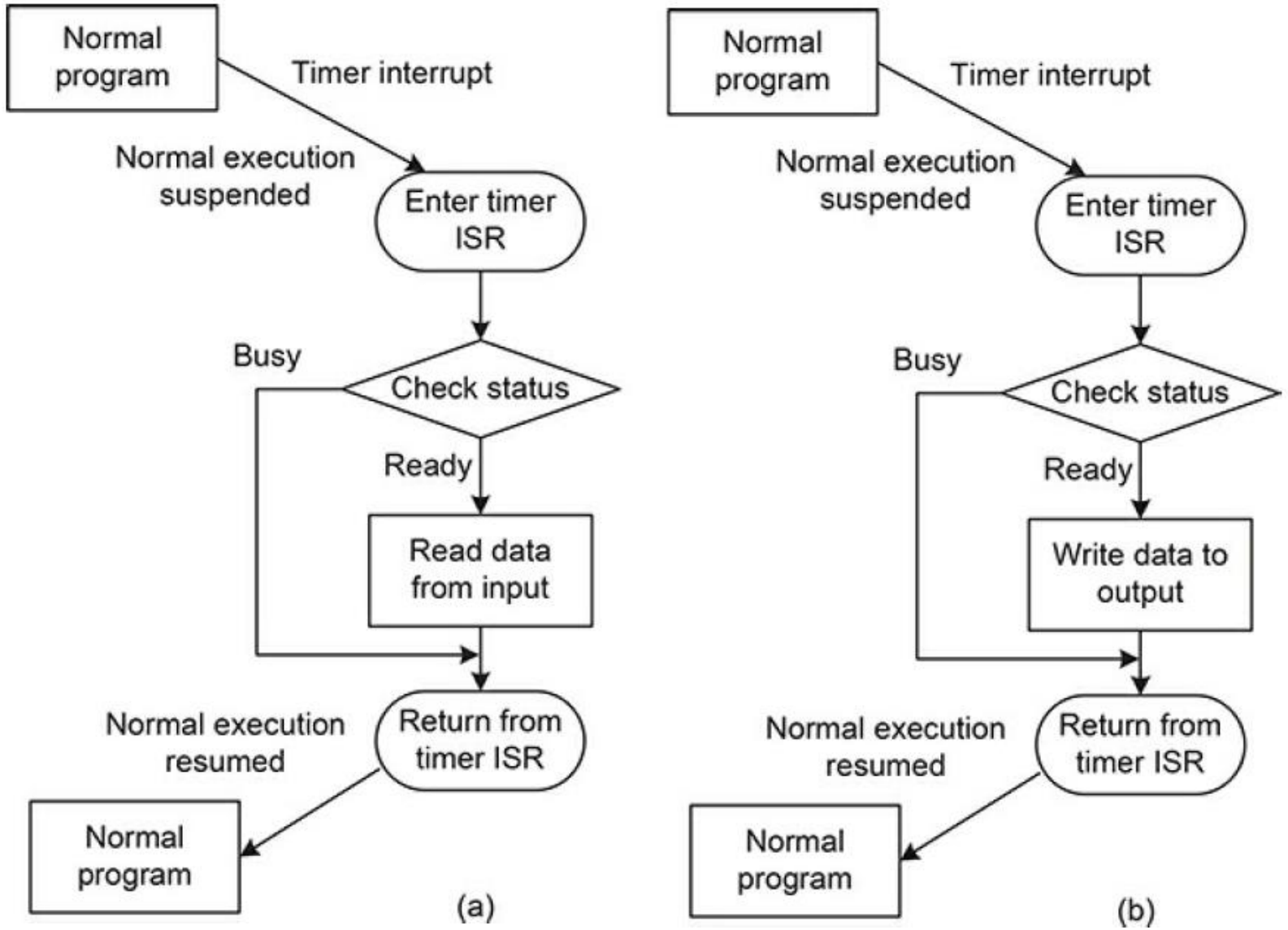
Methods For I/O Synchronization

1: *Blind Cycle*

2: **Polling-based Methods**

- *Continuous Polling or Busy Wait*
- **Periodic Polling**

3: Interrupt-driven Methods



Methods For I/O Synchronization

1: *Blind Cycle*

2: *Polling-based Methods*

- *Continuous Polling or Busy Wait*
- *Periodic Polling*

3: **Interrupt-driven Methods**

- **Interrupts**
- Direct Memory Access

Key limitations of before-mentioned I/O synchronization methods include software blocking and serial execution of tasks.

While waiting for hardware, processing time is wasted.

Interrupt-based I/O synchronization solves these issues by allowing the software to perform other tasks during hardware operations.

For an input device, the hardware will generate an interrupt when the data is ready and the corresponding software ISR will read the data.

In case of an output device, the hardware will generate an interrupt on the completion of assigned operation to inform the software. The software then can initiate another output operation while executing the corresponding ISR.

Methods For I/O Synchronization

1: *Blind Cycle*

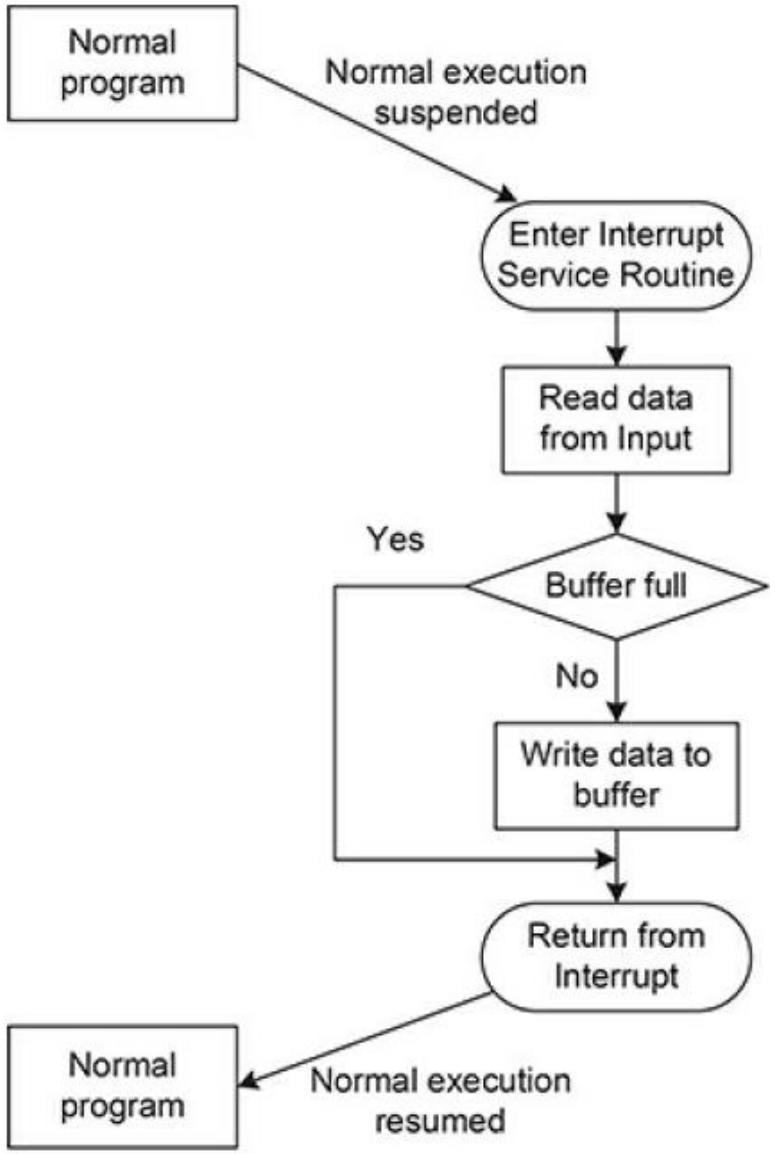
2: *Polling-based Methods*

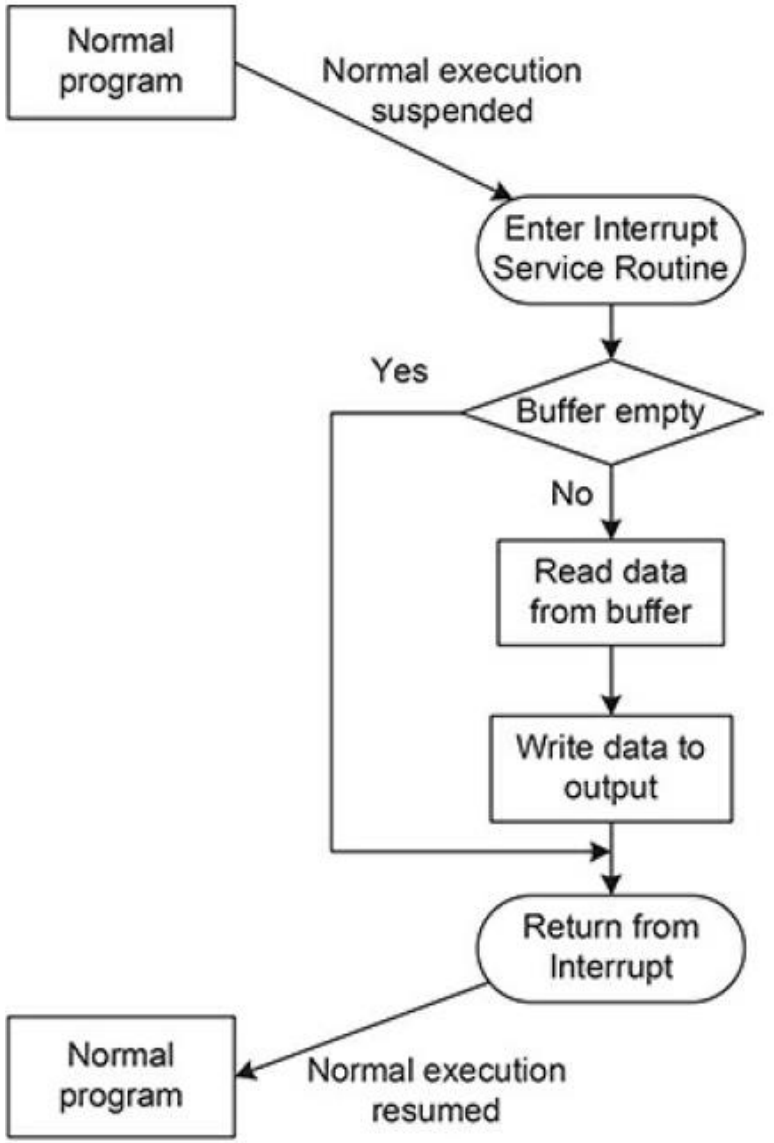
- *Continuous Polling or Busy Wait*
- *Periodic Polling*

3: **Interrupt-driven Methods**

- **Interrupts**
- Direct Memory Access

Interrupts are generated by the hardware, and the corresponding software ISR will read or write the data.





Methods For I/O Synchronization

1: *Blind Cycle*

2: *Polling-based Methods*

- *Continuous Polling or Busy Wait*
- *Periodic Polling*

3: **Interrupt-driven Methods**

- **Interrupts**
- Direct Memory Access

Advantages

It provides the **parallelization** of software and hardware operations with minimal overhead.

Useful for real-time applications as well as for event driven systems

Interrupt priorities and masking may be provided to different peripheral devices when synchronized using interrupts

It also avoids any **software blocking** and improves the systems overall performance.

Methods For I/O Synchronization

1: *Blind Cycle*

2: *Polling-based Methods*

- *Continuous Polling or Busy Wait*
- *Periodic Polling*

3: **Interrupt-driven Methods**

- *Interrupts*
- **Direct Memory Access**

Direct Memory Access (DMA) allows data transfers between peripheral devices and system memory without involving the main processor.

Cortex-M microcontrollers use a DMA controller to handle these transfers, including memory-to-memory transfers, offloading the processor for large data volumes.

The main processor is only notified when the transfer is complete

DMA is used for high-speed data transfers, ensuring efficient bandwidth use and low latency.

Methods For I/O Synchronization

1: *Blind Cycle*

2: *Polling-based Methods*

- *Continuous Polling or Busy Wait*
- *Periodic Polling*

3: **Interrupt-driven Methods**

- *Interrupts*
- **Direct Memory Access**

In DMA-based transfers, the DMA controller directly accesses peripheral devices and system memory, making it a hardware-driven process without involving the main processor or requiring software execution.

When an **input device** has new data, it sends a request to the DMA controller, which then transfers the data to a specified memory location without involving any software execution.

When an **output device** is ready, it sends a request to the DMA controller, which retrieves data from the specified memory location and transfers it to the output device.

TYPES OF EXCEPTIONS OR INTERRUPTS

Synchronous and Asynchronous Exceptions

Synchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

Synchronous exceptions are generated by the processor while executing instructions and are requested only after the completion of execution of the instruction. List of synchronous exceptions are

1. Memory management (MemManage) exception
2. BusFault exception
3. UsageFault exception
4. SVCall exception

The first three are system exceptions that are executed because of fault.

The last exception is also called a supervisor call (SVC), which is initiated by SVC instruction.

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

An **asynchronous exception** is the one that can occur, at an arbitrary time, during the execution of an instruction. These exceptions are generated by events other than the processor execution.

Below is the list of asynchronous exceptions:

1. *Reset exception* – used by the system on power up or in response to warm reset for restarting the system and is the highest priority exception
2. *NMI exception* – second highest priority exception used by the system to handle a highly critical situation
3. *PenSV exception* – a software triggered exception and unlike other software interrupts is an asynchronous exception
4. *Systick exception* – generated by the system timer when its count has reached zero
5. *Interrupt* – an exception triggered by a peripheral device, or it can be generated by the associated software and is also called a ***peripheral interrupt*** or ***external interrupt***

For asynchronous exceptions apart from reset exception, it is possible that the processor can complete the execution of currently being executed instruction before entering the exception handler.

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

Peripheral (or External) Interrupts

- The maximum number of external interrupts that are supported by Cortex-M microcontrollers are 240.
- They are assigned to different peripherals including timers, communication interfaces (UART, SPI, I2c, CAN bus etc.), data converter modules, general purpose I/Os etc.
- A specific microcontroller can use an arbitrary number of external interrupts depending on its peripherals.
- A total of 139 interrupts are supported by the NVIC controller of Texas Instrument's TM4C123 microcontroller and are labeled as IRQ0 to IRQ138.

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

Peripheral (or External) Interrupts

- An interrupt from each peripheral device is assigned a specific IRQ number.
- This unique IRQ number is used to determine the address of corresponding interrupt service routine (ISR) in the interrupt vector table.
- Assuming interrupt vector table base address is 0x00000000, then nth IRQ vector address is obtained as 4n + 64.
- This address when defined in terms of exception number m, is obtained as 4m, providing the relation between exception number and IRQ number as m = n +16.
- The use of GPIO pins as external interrupts will be discussed in detail in this lecture.

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

IRQ no.	Peripheral	IRQ no.	Peripheral
IRQ0	GPIO port A	IRQ1	GPIO port B
IRQ2	GPIO port C	IRQ3	GPIO port D
IRQ4	GPIO port E	IRQ5	UART0
IRQ6	UART1	IRQ7	SPI0
IRQ8	I2C0	IRQ9	PWM0 Fault
IRQ10	PWM0 Generator 0	IRQ11	PWM0 Generator 1
IRQ12	PWM0 Generator 2	IRQ13	QEI0
IRQ14	ADC0 Sequencer 0	IRQ15	ADC0 Sequencer 1
IRQ16	ADC0 Sequencer 2	IRQ17	ADC0 Sequencer 3
IRQ18	WD Timer 0 & 1	IRQ19	16/32 Bit Timer 0A
IRQ20	16/32 Bit Timer 0B	IRQ21	16/32 Bit Timer 1A
IRQ22	16/32 Bit Timer 1B	IRQ23	16/32 Bit Timer 2A
IRQ24	16/32 Bit Timer 2B	IRQ25	Analog comparator 0
IRQ26	Analog comparator 1	IRQ27	Reserved
IRQ28	System control	IRQ29	Flash and EEPROM control

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

IRQ no.	Peripheral	IRQ no.	Peripheral
IRQ30	GPIO port F	IRQ31	Reserved
IRQ32	Reserved	IRQ33	UART2
IRQ34	SPI1	IRQ35	16/32 Bit Timer 3A
IRQ36	16/32 Bit Timer 3B	IRQ37	I2C1
IRQ38	QEI1	IRQ39	CAN0
IRQ40	CAN1	IRQ41	Reserved
IRQ42	Reserved	IRQ43	Hibernation module
IRQ44	USB	IRQ45	PWM0 Generator 3
IRQ46	uDMA Software	IRQ47	uDMA error
IRQ48	ADC1 Sequence 0	IRQ49	ADC1 Sequence 1
IRQ50	ADC1 Sequence 2	IRQ51	ADC1 Sequence 3
IRQ52-56	Reserved	IRQ57	SPI2
IRQ58	SPI3	IRQ59	UART3
IRQ60	UART4	IRQ61	UART5
IRQ62	UART6	IRQ63	UART7

Asynchronous Exceptions

TYPES OF EXCEPTIONS OR INTERRUPTS

IRQ no.	Peripheral	IRQ no.	Peripheral
IRQ64-67	Reserved	IRQ68	I2C2
IRQ69	I2C3	IRQ70	16/32 Bit Timer 4A
IRQ71	16/32 Bit Timer 4B	IRQ72-91	Reserved
IRQ92	16/32 Bit Timer 5A	IRQ93	16/32 Bit Timer 5B
IRQ94	32/64 Bit Timer 0A	IRQ95	32/64 Bit Timer 0B
IRQ96	32/64 Bit Timer 1A	IRQ97	32/64 Bit Timer 1B
IRQ98	32/64 Bit Timer 2A	IRQ99	32/64 Bit Timer 2B
IRQ100	32/64 Bit Timer 3A	IRQ101	32/64 Bit Timer 3B
IRQ102	32/64 Bit Timer 4A	IRQ103	32/64 Bit Timer 4B
IRQ104	32/64 Bit Timer 5A	IRQ105	32/64 Bit Timer 5B
IRQ106	System Exception	IRQ107-133	Reserved
IRQ134	PWM1 Generator 0	IRQ135	PWM1 Generator 1
IRQ136	PWM1 Generator 2	IRQ137	PWM1 Generator 3
IRQ138	PWM1 Fault		

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

GPIO Interrupts

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

Configuring different peripheral devices to use interrupts for synchronization is performed during their initialization and is considered as one aspect of device initialization.

To enable interrupts for a peripheral device requires configuration at the following three different levels.

1. ***Processor interrupt configuration*** – provides global enabling and disabling of interrupts
2. ***NVIC configuration*** – allows device level interrupt configuration, priority assignment and pending behavior management
3. ***Device interrupt configuration*** – provides device level interrupt configurations

The NVIC and processor interrupt configurations are quite similar for different peripheral devices. However, the device interrupt configuration varies from device to device and the configuration for one device may not be used for a different peripheral device.

1. Processor Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

There are three processor registers that are used for interrupt masking:

1. **Priority Masking Register** (PRIMASK) – enable/disable interrupts and exceptions with programmable priority
2. **Fault Mask Register** (FAULTMASK) – enable/disable interrupts and exceptions except Reset and NMI
3. **Base Priority Masking Register** (BASEPRI) – blocks all the interrupts of either the same or lower priority

The default values of these registers are such that all interrupts are enabled.

However, in case of critical code execution, that should not be interrupted, global interrupt enabling and disabling can be performed.

2. NVIC Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

Following are that are used for NVIC Configurations:

1. **Interrupt Enable Register** (ENx) – enables device interrupts
2. **Interrupt Disable Register** (DISx) – disables devices interrupts
3. **Priority Configuration Register** (PRIy) – assign priority to device interrupts

Depending on the specific peripheral interrupt allocation in the vector table

- x can take values from 0 to 4
- y can take values from 0 to 34

Separate NVIC registers are allocated for enabling and disabling of interrupts (IRQ0-IRQ138 or correspondingly exceptions 16 to 154).

2. NVIC Configuration

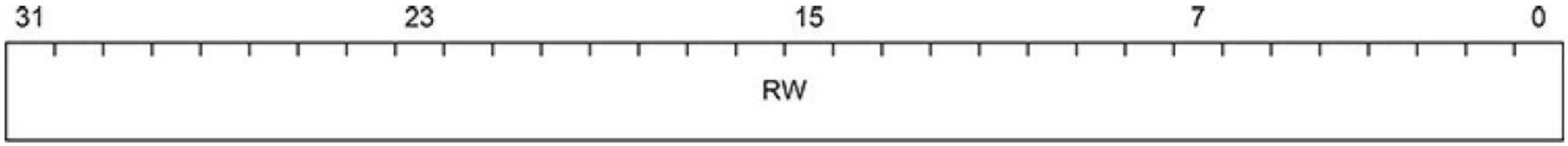
CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

Register label	Address	Reset value	Brief description
Interrupt enable registers			
EN0	0xE000E100	0x00000000	Used to enable interrupts 0 to 31.
EN1	0xE000E104	0x00000000	Used to enable interrupts 32 to 63.
⋮	⋮	⋮	⋮
EN4	0xE000E110	0x00000000	Used to enable interrupts 128 to 138.
Interrupt disable registers			
DIS0	0xE000E180	0x00000000	Used to disable interrupts 0 to 31.
DIS1	0xE000E184	0x00000000	Used to disable interrupts 32 to 63.
⋮	⋮	⋮	⋮
DIS4	0xE000E190	0x00000000	Used to disable interrupts 128 to 138.
Priority configuration registers			
PRI0	0xE000E400	0x00000000	Priority for interrupts 0 to 3.
PRI1	0xE000E404	0x00000000	Priority for interrupts 4 to 7.
⋮	⋮	⋮	⋮
PRI34	0xE000E488	0x00000000	Priority for interrupts 136 to 138.

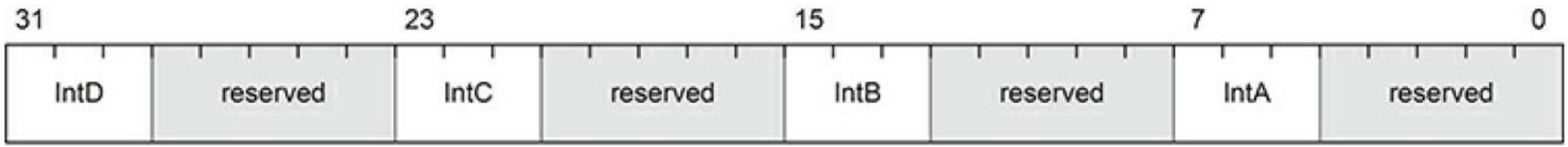
2. NVIC Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

Interrupt Enable/Disable Registers Bit-field



Interrupt Priority Registers Bit-field



3. Device Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

Each device interrupt configuration has following two components.

1. **Interrupt Vector Table Configuration and ISR Implementation**
2. **Device Interrupt Register Configurations**

3. Device Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

1. Interrupt Vector Table Configuration and ISR Implementation

The interrupt vector table holds the addresses of interrupt service routines (ISRs) for each interrupt source.

For handling an interrupt from a device like PortF, all vector entries must be included or reserved before adding the ISR for PortF.

This is because fixed IRQ numbers and ISR addresses are assigned. As a result, vector table entries can't be skipped, even if unused.

In Cortex-M, ISRs are implemented like regular C function calls.

3. Device Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

2. Device Interrupt Register Configurations for GPIO

Each device can have multiple possible sources of interrupts that can be enabled/disabled (unmasked/masked) individually using device interrupt related registers.

The device level configuration of GPIO as a source of external interrupts is discussed here, specifically, the configuration of port F pin 4 as source of external interrupt.

When a GPIO port pin is configured as an external interrupt, then interrupt triggering can be performed by one of the following two possible activations.

- Level triggered
- Edge triggered

3. Device Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

A **level-triggered interrupt** remains active until the peripheral deactivates it by de-asserting the interrupt signal.

In case of level triggered interrupt, the processor executes the ISR and on the exit if the interrupt condition is still valid, the processor immediately enters the ISR again, provided no higher priority interrupt occurs in the meantime.

In case of an **edge triggered interrupt**, the interrupt signal is generated by the rising or falling edge (or both edges) on the corresponding GPIO port pin.

There are seven 32-bit registers associated with GPIO port for interrupt configuration. Only least significant 8-bits are used by all these registers to configure each GPIO port pin individually.

3. Device Interrupt Configuration

CONFIGURING INTERRUPTS FOR CORTEX-M DEVICES

#	Configuration Register	Configuration Setting
1	**Interrupt Sense (IS) Register** Configures interrupt triggering as edge or level	Clear a bit to configure corresponding GPIO port pin as edge triggered otherwise level triggered
2	**Interrupt Event (IE) Register** Configures the type of the interrupt event	When a bit is cleared the corresponding GPIO port pin is configured as falling edge (or level low) triggered interrupt
3	**Interrupt Both Edge (IBE) Register** Configures both edges as the source of interrupt	Set a bit to configure corresponding GPIO port pin as both edge triggered
4	**Interrupt Mask (IM) Register** Enables or disables individual port pin interrupts	Writing 1 to a bit enables the corresponding port pin interrupt.
5	**Masked Interrupt Status (MIS) Register** Indicate the occurrence of an interrupt condition	The occurrence of interrupt on a particular pin is indicated by turning the corresponding bit of this register High (1).
6	**Interrupt Clear (ICR) Register** Used to clear an interrupt flag in the status registers	Writing ‘1’ to a bit of ICR will clear the corresponding bit (status flag) in MIS register.

Thank You!

Any Questions?