

# 4.20 Advanced I/O synchronization

Interrupt driven I/O

Other I/O synchronization methods

EE302 – Real time and embedded systems

1

## Overview

2



- Aims
  - To introduce alternatives to polling I/O methods, particularly interrupt driven I/O
- Learning outcomes – you should be able to...
  - Explain the handling of an interrupt using polled interrupts or vectored interrupts
  - Explain/show how to implement interrupt priority (when multiple interrupts occur simultaneously) using polled interrupt software, interrupt daisy chaining, and a programmable interrupt controller
  - Convert between single interrupt service routine (ISR) implementing polled interrupt priority vs. multiple ISRs (and associated hardware) implementing vectored interrupt priority

25 November 2020

2

## Recall - I/O Synchronization

3

- Why refer to it as I/O synchronisation? 
- Some specific cases...
  1. The CPU is faster than the I/O device
  2. The I/O device is faster than the CPU
  3. The I/O device needs to transfer data at regular, predictable times
  4. The I/O device needs to transfer data at irregular, unpredictable times 

25 November 2020

3

## Interrupt driven I/O

4

I/O devices “interrupt” CPU via an “interrupt request” (IRQ) line/pin as follows...

1. When an I/O device wants service it causes a transition on the CPU’s IRQ pin (or internal IRQ line when it is an internal peripheral)
2. CPU immediately(\*) pauses main program execution
3. CPU executes Interrupt Service Routine (ISR) and ISR code services the device (e.g. by reading or writing data)
4. When ISR completes, CPU resumes program execution

*NOTE: see 4.10 Timer notes (where interrupts were introduced) for more detail*

25 November 2020

4

## Example – interrupt driven timer

5

```
global gTimerExpired = FALSE

main:...

setup():
    set TMR0 = TMR0_START_VALUE

loop():
    while gTimerExpired is FALSE, do nothing // wait
    set gTimerExpired = FALSE

    doWork...
    // NOTE: no delay or sleep required

doWork():
    ...

isr(): // Interrupt Service Routine implementing polled interrupts
    if TMR0IF interrupt
        set TMR0IF = 0 // clear interrupt flag since we've responded
        set TMR0 = TMR0_START_VALUE // reset timer
        set gTimerExpired = TRUE // signal superloop
```

25 November 2020

5

## Example – interrupt driven UART

6

```
// NOTE: this pseudocode reads commands from the UART but does
// nothing useful with them.
main:...

setup():
    configure initial device/peripheral settings
    unmask Rx interrupt and enable interrupts

loop():
    processSerialCommand()
    ...
    delay or sleep

processSerialCommand():
    if gCommandComplete:
        process command...

isr(): // Interrupt Service Routine implementing polled interrupts
    if (UART Rx interrupt)
        buffer[i] = value of UART Rx register
        if buffer[i] is END_OF_COMMAND // e.g. a newline
            set gCommandComplete = TRUE
            increment i (if buffer still has space)
```

25 November 2020

6

## Interrupt synchronisation features

8

- Best for
  - I/O which occurs at infrequent or unpredictable intervals
  - Asynchronous I/O which takes place “in the background”
- Advantages
  - Software doesn't “waste” time polling I/O devices
- Disadvantages
  - Software can be complex

25 November 2020

8

## Some interrupt issues

9

- If there are many I/O devices...
  - Which one caused the interrupt (and which ISR should be executed)?
  - What happens if 2 or more devices try to interrupt at once?
- Solutions...
  - Polled interrupts use software based priority (used by PIC Microcontroller)
  - Vectored interrupts with Interrupt controller priority or Daisy chained priority
- Additional considerations
  - Masking and disabling interrupts
  - Multiplexing a single interrupt line

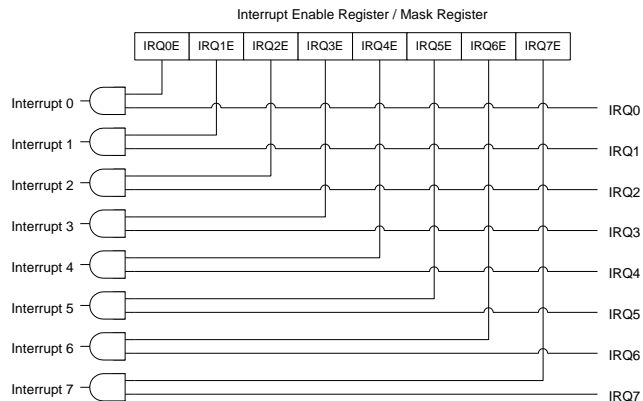
25 November 2020

9

## Masking and disabling interrupts

10

- With multiple interrupt systems there has to be a way to enable and disable all or individual interrupts.

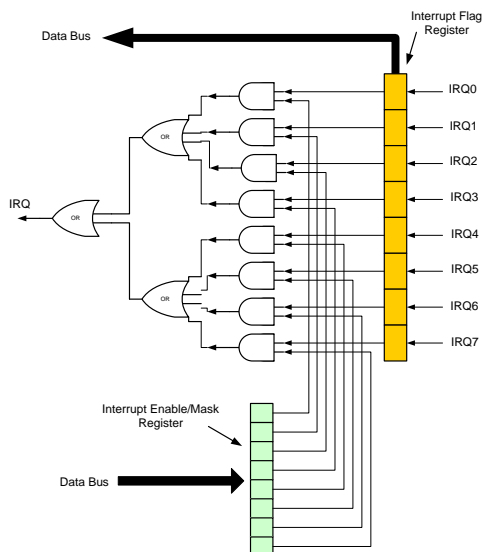


25 November 2020

10

## Multiplexing a single interrupt line

11



This figure shows how 8 I/O devices may be connected to a single IRQ pin.

Each of the 8 interrupts is maskable (mask register).

Software priority (polled interrupts) is made possible with the inclusion of an Interrupt Flag Register.

25 November 2020

11

## [software] Polled Interrupts

12

- Used when there are multiple devices connected to a single IRQ pin and no additional information transmitted during interrupt
- There is just one ISR and all interrupt code located there (both an advantage and disadvantage)
- ISR code polls (checks) some register(s) to determine which interrupt occurred
- Easy to set priority: Interrupt priority is determined by order in which the code polls (checks) the set of possible interrupts
- Simplifies CPU and peripherals compared to vectored interrupts, but can be slightly slower
- used in PIC microcontroller

```
isr()
{
    if SSP interrupt flag set
        handle SSP Interrupt
    else if TMR0 interrupt flag set
        handle Timer0 Interrupt
    else if ADC interrupt flag set
        handle ADC Interrupt


    // Note: often not a good idea
    // to call functions from isr
    // - instead embed handler code
    // directly in isr
}
```

25 November 2020

12

## [hardware] Vectored Interrupts

13

- Using vectored interrupts, there are multiple ISRs, one per interrupt source
- Each I/O Device is configured with a unique number called a vector, which can be either
  - the address (or part thereof) of its particular ISR
  - an index into a table (array) of ISR addresses, called the interrupt vector table or interrupt dispatch table 
- Hardware signalling is used to get the vector from the device which generated interrupt
- Compared to (software) polled interrupts
  - Faster to identify interrupt source
  - Requires hardware to implement interrupt priority
  - Both CPU and peripherals are more complex to handle vector signalling and automatic dispatch to correct interrupt service routine

25 November 2020

13

## Vectored interrupt ISR code

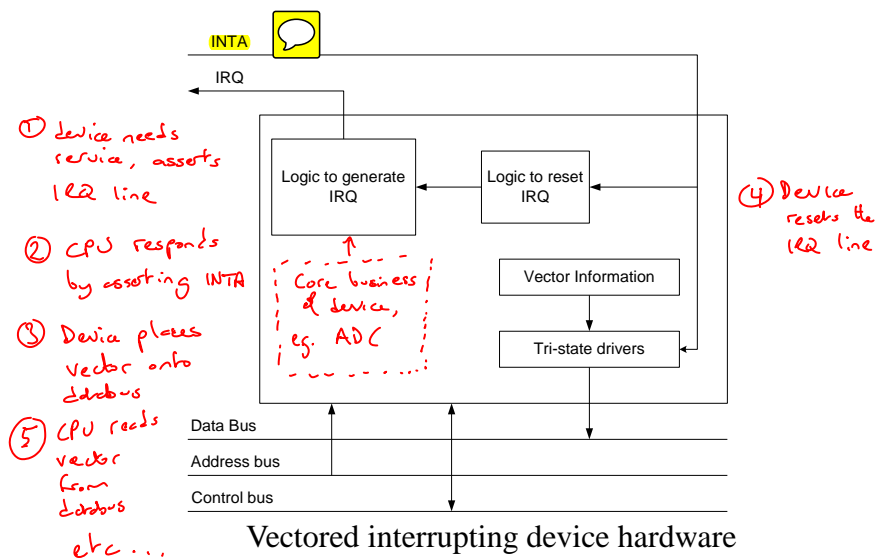
```
// Separate ISRs for each possible interrupt.  
// No checking of flags required to identify  
// which interrupt occurred - the vector which  
// causes particular ISR to be invoked is sufficient.  
  
ssp_isr()  
    handle SSP Interrupt  
  
tmr0_isr  
    handle Timer0 Interrupt  
  
adc_isr  
    handle ADC Interrupt
```

Compare with polled interrupts example

14

## Basic vectored interrupt hardware (each device)

15



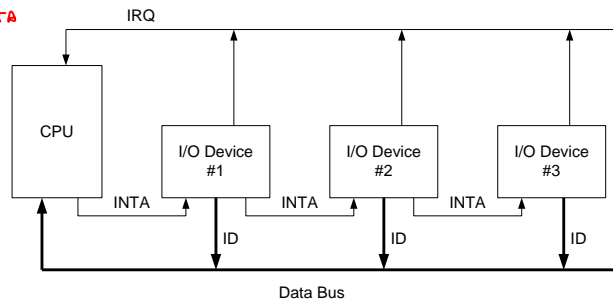
25 November 2020

15

## Vectored Interrupt Priority – Daisy Chain

16

- Suppose device 2 and device 3 generate simultaneous interrupts
- ① IRQ line asserted by device 2,3
  - ② CPU responds INTA
  - ③ Device 1 has no interrupt pending so forwards INTA
  - ④ Device 2 has interrupt pending so responds with ID (vector)
  - ⑤ CPU handles interrupt
  - ⑥ IRQ still asserted (because of device 2) so do it again ....



Daisy Chained Interrupts for Prioritization

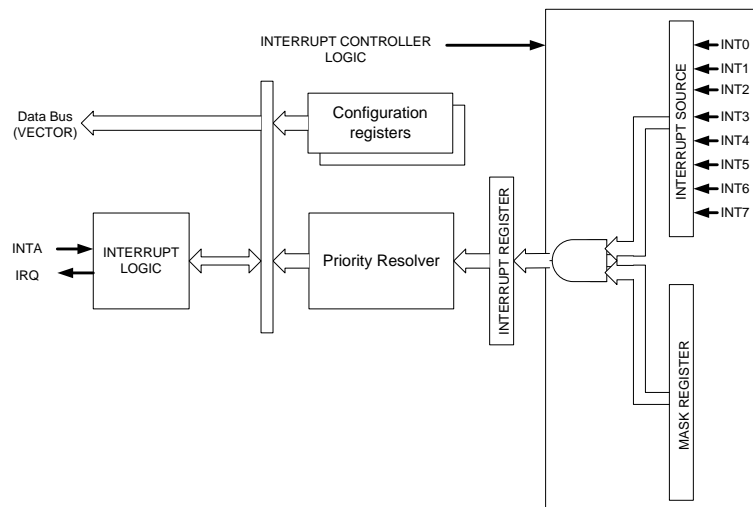
Fast but inflexible

25 November 2020

16

## Vectored Interrupt Priority – Programmable Interrupt Controller

17



25 November 2020

17



# Synchronization methods

18

There are four synchronization methods used to manage I/O data transfers

1. Polling
2. Handshaking
3. Interrupts
4. Direct Memory Access – DMA

*Note 1: Different methods may be used for different I/O devices (in a single embedded system)*

*Note 2: Combinations of methods may be used with a single I/O device*

25 November 2020

18

## 1. Polling features (looked at previously)

19

### □ Advantages

- Simple to implement
- I/O is synchronous, taking place in the foreground
- Device priority is easy to change
  - Determined by the order in which software polls (checks) the devices

### □ Disadvantages

- CPU time may be wasted continually polling
- Response time/throughput is often a compromise
- Not best choice when I/O takes place infrequently or at unpredictable intervals


25 November 2020

19

## 2. Handshaking I/O

20

### □ A hardware method of synchronizing with a slow(ish) I/O device

- Not strictly an alternative to polling and invisible to software 
- Can use with other synchronisation methods
- I/O device “handshakes” with CPU to say when ready
- Often used to interface to “slow” memory

### □ Advantages

- Simpler for programmer (than software polling)
- Potentially faster (than software polling)

### □ Disadvantages

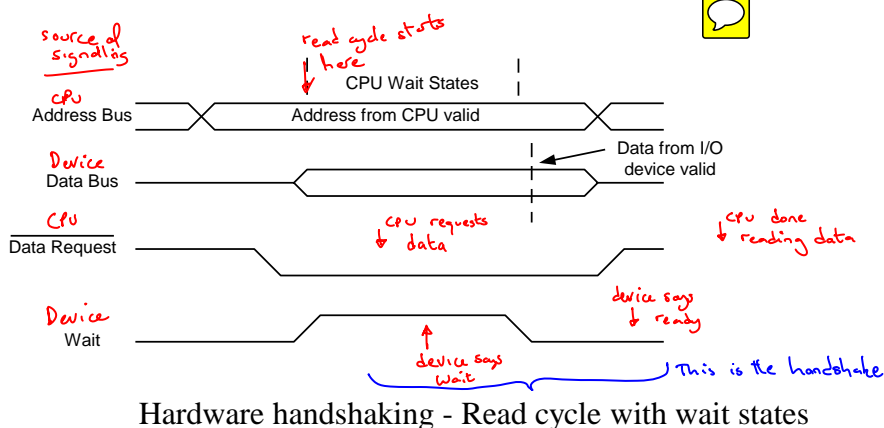
- Requires additional pins/lines from the CPU, e.g. READY/WAIT
- When CPU is in the wait state it can not do anything else

25 November 2020

20

## Handshaking example

21



25 November 2020

21

### 3. Interrupt driven I/O

22



- Best for
  - I/O which occurs at infrequent or unpredictable intervals
  - Asynchronous I/O which takes place “in the background”
- Advantages
  - Software doesn’t “waste” time polling I/O devices when no service is required
- Disadvantages
  - Software can be complex

25 November 2020

22

### 4. Direct Memory Access - DMA

23

- A DMA Controller (DMAC) is a single purpose processor designed to perform high speed data transfers between memory and I/O devices. 
- A DMAC does not require the CPU to accomplish the data transfer
  - The DMAC takes control of the Address and Data buses to effect the data transfer
    - May suspend or halt the CPU (temporarily)
    - May steal some memory cycles (Cycle Stealing) 
    - Often I/O is slow-ish so DMAC doesn’t interfere much
  - Used for data intensive I/O such as hard disk or memory card transfers (e.g. in a digicam)

25 November 2020

23

### *Exam: 2009-2010 Q3 (c)*

25

*Briefly describe the terms interrupt vector and interrupt priority.*

*Consider a system with 3 peripheral devices (A, B, and C) which may generate interrupts. The priority order from highest to lowest is A, B, C. The system uses the daisy chain implementation of vectored interrupts.*

*With the aid of a diagram, describe in detail how the scheme would operate if device A and device C both required service simultaneously.*

25 November 2020

25

26

25 November 2020

26

## *Exam: 2011-2012 Q4 (a)*

27

*List the key differences between software polling, polled interrupts and vectored interrupts.*

*For a system using polled interrupts, show (using pseudocode) how you would prioritize interrupts from the following devices in the order given: Timer, ADC, and UART.*

25 November 2020

27

28

25 November 2020

28

*Consider a polled interrupt routine that must use the timer interrupt (TMR0IF) to determine when to start an ADC conversion and use the end of ADC conversion interrupt (ADIF) to copy the ADC sample into set the variable **gSample** and then set the variable **gSampleToProcess** true.*

*Write the pseudocode implementation of the polled interrupt routine.*

*Assume that the Timer0 register is TMR0, that the period between timeouts is ADC\_SAMPLE\_TIMEOUT, that ADC conversion is started by setting the ADGO bit, and that we are only interested in 8-bit ADC samples from register ADRESH.*