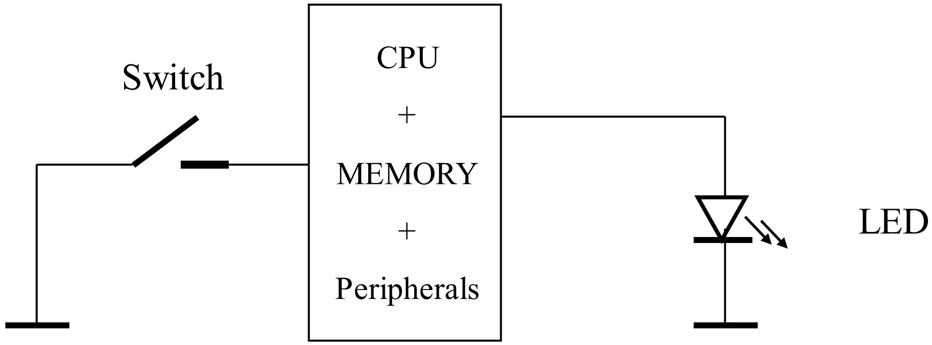
2.10 Basic Digital Port Input/Output Synchronisation

Hardware and software aspects of input/output synchronisation

Overview

- Aims
  - Introduce the basic concept of I/O synchronization and the polling method in particular.
  - Study I/O synchronization for digital ports
- Learning outcomes – you should be able to…
  - Explain I/O synchronization, why its needed, and the polling method for handling it
  - Differentiate between bus aware and non-bus aware devices
  - Explain the purpose of latches and buffers within a digital I/O port
  - Show how to handle various electrical issues, e.g. driving a high current device from a current limited port
  - Explain switch debounce and show how to handle it in hardware or software
  - Write pseudocode/code to poll I/O ports and take action continuously, only on change, only after a delay, etc.

What is I/O?

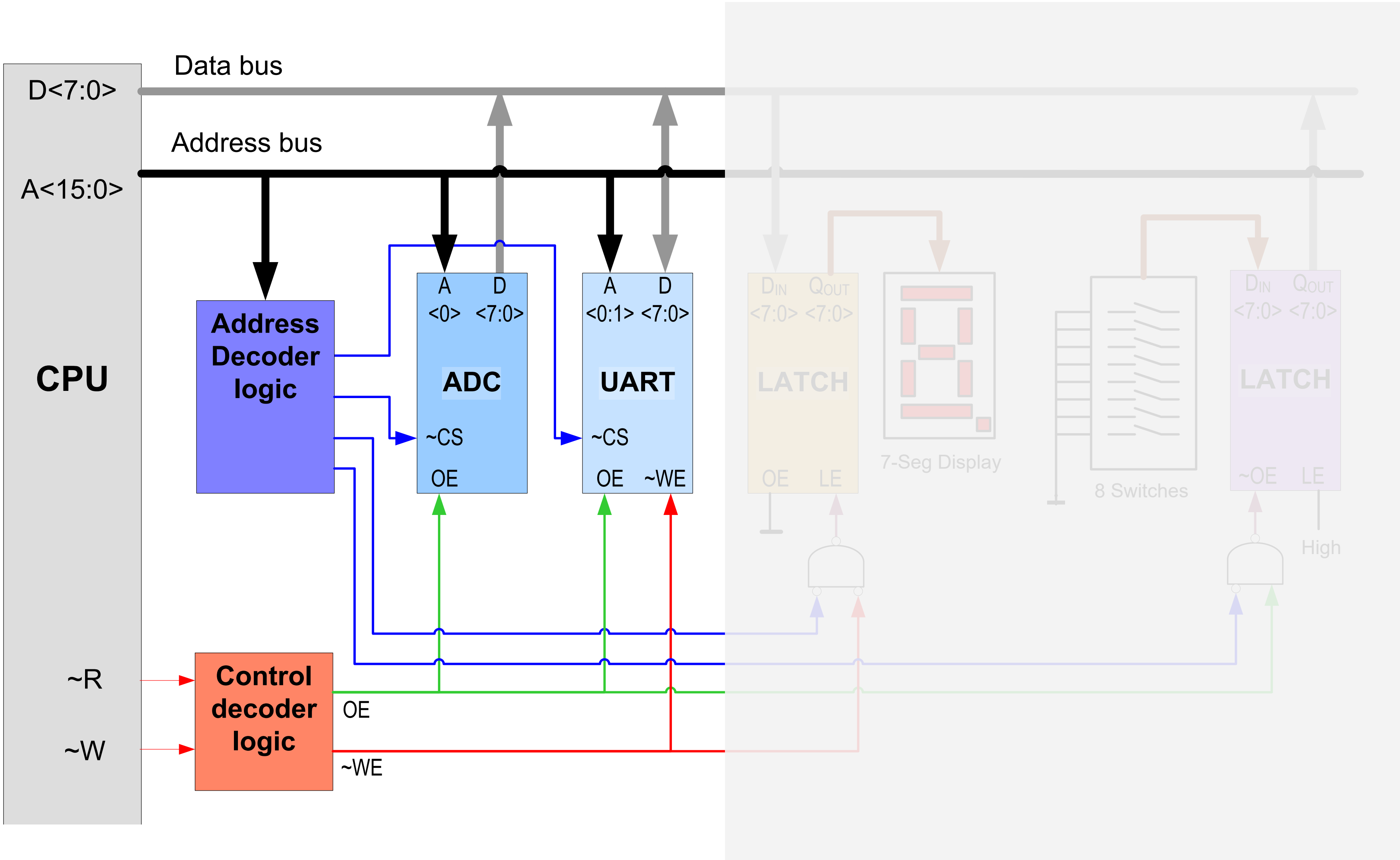


- To be useful a computer must be able to communicate with the outside world
- Simple example
  - input from a switch (one bit of information)
  - Indicate output via an LED (also one bit of information)

Interfacing to a CPU

- A CPU can only receive input **data** and send output data via the **data bus**
- Because the CPU may be connected to many devices…
  - For every I/O operation, the CPU outputs an **address** on the **address bus**
  - Decoder logic uses the address to determine which of the devices to select – all other devices ignore the operation
- The CPU's Interface to external world is mainly via peripheral devices attached to the data bus

Interfacing devices to a CPU

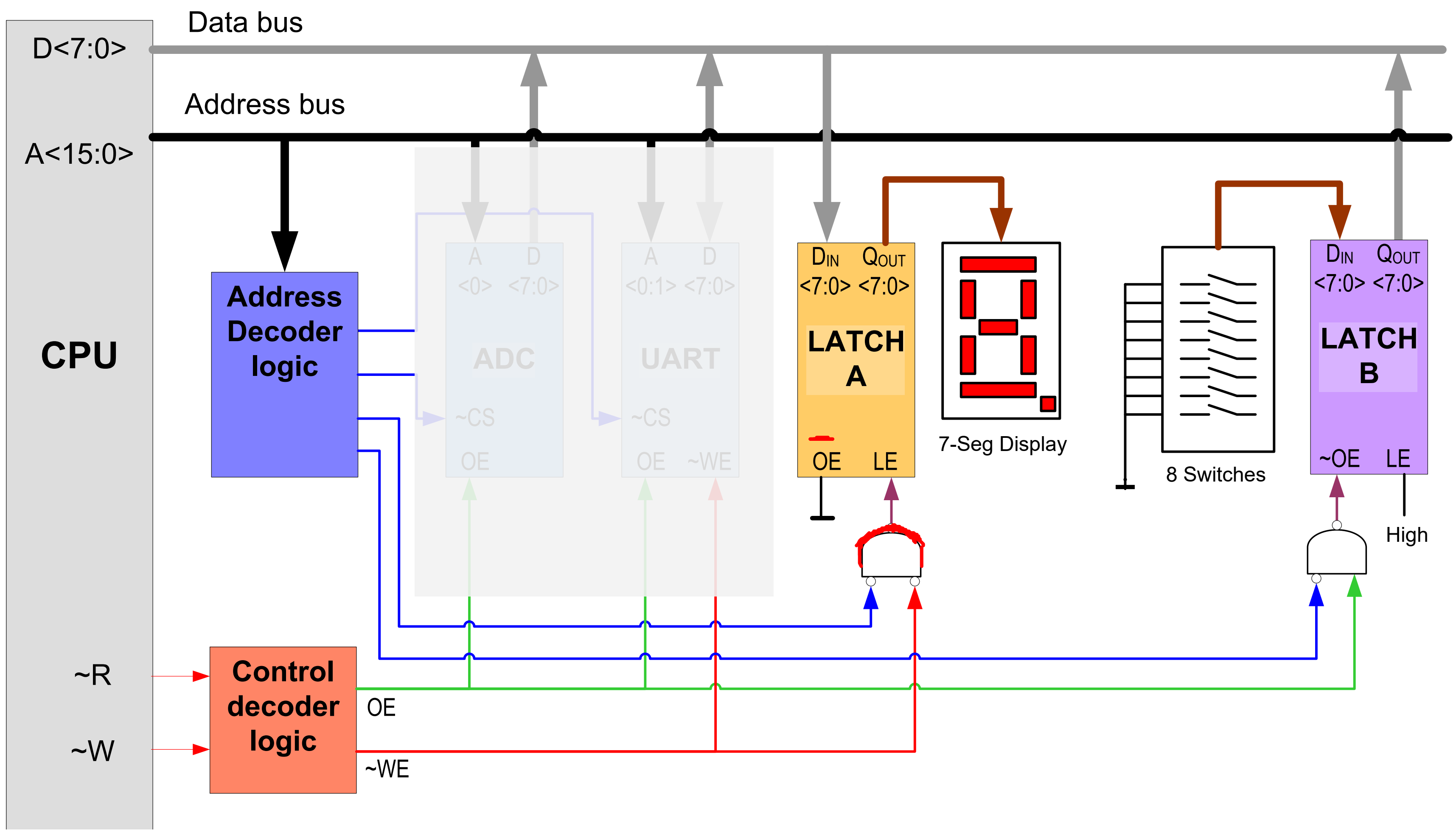


Bus-aware devices

Interfacing to a CPU

- **Bus-aware** devices
  - Can usually be connected directly to data and address bus of CPU
  - E.g. I/O ports, timers & clocks, communications devices, A/D and D/A convertors, etc.
  - May be internal to processor package (e.g. in a microcontroller) or external (e.g. in a GPP)
- Devices and hardware that is **not bus-aware**
  - Cannot be connected directly to data and address bus of CPU. Connected instead via a bus-aware peripheral, usually an I/O port
  - E.g. switches and LEDs are “dumb” devices that are not bus-aware

Interfacing devices to a CPU



Non bus-aware devices

Interfacing devices to a CPU – hardware signalling

For the sake of example consider how the processor/CPU would read the state of the input switches (connected via LATCH B)

1. The CPU outputs a valid address on the address bus. In this case it must be the address of the LATCH B.
2. The address decoder logic decodes the address (on the address bus lines) such that at most one device is selected, in this case LATCH B, by activating the appropriate chip select (CS) line.
3. After a short time the CPU activates its read control line (R), indicating that it wants to read a value from an external device via the data bus
4. The control decoder logic maps this control input to activating the output enable (OE) line.
5. The combination of OE and CS activates the OE of LATCH B which causes the latch to pass the switch state on it's inputs (D_{IN}) to its outputs (Q_{OUT}). Thus the state of the switches now appears on the data bus lines.
6. Finally the CPU reads the current state of the data bus lines (thereby reading the switch states).

The next CPU instruction will change the address and ensure that the latch "freezes" its current values and does not output any values onto the data bus.

I/O Synchronization

- Managing the time and manner in which input/output takes place is called **I/O sychronization**
- There are various synchronization methods that can be used but our initial focus will be on **polling I/O**
- Polling is appropriate for a wide range of I/O devices including digital I/O ports, analog I/O ports (via the ADC), the USART, etc.

Polling - analogy

- You need a colleague to do a job for you (e.g. get some information)
  - The “job” might be something once-off, something intermittent (done as needed), or something that needs to be done regularly
  - Whether the job is repeated or not and how regularly, has little effect on how the polling operation is performed each time
- Go ask them to do the job – this begins the polling operation
- Then there are a number of different possibilities for how the polling operation proceeds and completes…
  1. Job done immediately – move on to other work
  2. Assume job will get done – no need to check back, move on to other work
  3. Wait until job done – do no other work until done
  4. Check back regularly until job done – work ahead between checks

Polling

- Characteristics of polling
  - All transfers to and from I/O devices are performed by the software at times of its choosing (often at regular intervals)
  - the software “polls” (checks) the device to see if an I/O operation is possible/required
    - For simple I/O operations, polling may only be required to check the current “value” or “flag bits” in a device register
    - More complex I/O may consist of initiating some I/O and then polling until it has been completed
- Examples
  - Example 1: poll port B, bit 0, every 20ms to see if active low button is closed (=0) or open (=1)
  - Example 2: poll ADC sample conversion flag to see if next digital sample is ready. If it is, then read the sample value.

Polling approach

- There are two main questions to answer when designing a polling solution:

1. When to begin an I/O operation?

2. How to complete the I/O operation?

Contd.

1. The possibilities for when to begin an I/O operation…

- Once off
  - This is largely the same as intermittent (see next point)
- Intermittently (as needed)
  - The I/O operation is initiated only when needed
  - Often used for certain kinds of outputs (e.g. change the state of a LED in response to something else, transmit some data over serial, etc.)
- At regular intervals
  - The I/O operation is initiated at regular intervals
  - This is typically used to monitor inputs such as button press, received serial communication, etc.
  - It is also often used to sample analogue inputs with a steady sample rate

Contd.

2. How to detect completion of an I/O operation depends on the how quickly/reliably it can be completed

- **Poll once and forget**
  - In this case, we don't wait or check for completion – once initiated, we assume it will complete quickly enough. E.g. Digital port input/output
  - Other work can continue immediately.
- **Busy wait polling (blocking)**
  - Wait for operation to complete, checking (polling) constantly. E.g. waiting for ADC conversion complete once initiated.
  - No other work can be done while we wait
- **Multi-task polling (non-blocking)**
  - Other work can proceed while waiting for operation to complete
  - check back (poll) at regular intervals to detect I/O operation completion

Poll once and forget


```
// General app structure
global gButtonState = OPEN

Main…

setup()
  setup button and LEDS…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK
```

```
pollSwitch()
  …

updateLED()
  if gButtonState is CLICKED
    // poll once and forget --
    // Note how the code doesn't wait
    // to see if the LED changed state
    // or not
    toggle LED_PIN
```

In this example, the updateLED function is writing to a digital I/O port pin – i.e. polling an output

Busy wait polling - ADC


```
// General app structure
global gButtonState = OPEN
global gAdcValue = 0

Main…

setup()
  setup button and ADC…

loop()
    pollSwitch()
    checkADC()
    delay SUPERLOOP_TICK
```

Note: If a programme has real time contraints, busy wait polling should only be used for short delays that cannot cause deadlines to be missed

```
pollSwitch()
  …

checkADC()
  if gButtonState is CLICKED
    // initiate ADC conversion
    set ADC_GO_DONE_BIT = GO

    // busy wait polling:
    // The code checks continuously
    // until conversion is done
    // and no other useful work can
    // be done while waiting
    while (ADC_GO_DONE_BIT is not DONE)
      doNothing

    // conversion is done and result
    // is in adc register, so copy it
    // to global variable
    set gAdcValue = adcRegister
```

Busy wait polling – transmit string


```
// General app structure
global gButtonState = OPEN

Main…

setup()
  setup button and serial…

loop()
    pollSwitch()
    transmitIfNeeded()
    delay SUPERLOOP_TICK
```

other tasks (delayed a long time)

```
pollSwitch()
  …

transmitIfNeeded()
  if gButtonState is CLICKED
    transmitStringCompletely("hello")

transmitStringCompletely(str)
  while not at end of str
    // busy wait polling:
    // The code checks continuously
    // until TXREG is empty
    // and no other useful work can
    // be done while waiting
    while TXREG is full
      doNothing

    // TXREG is empty, send next char
    set TXREG = next char from str
```

Example:

Multi task polling – transmit string


```
// General app structure
global gButtonState = OPEN
global gTxBuffer

Main…

setup()
  setup button and serial…

loop()
    pollSwitch()
    transmitIfNeeded()
    completeTransmission()
    // other tasks…
    delay SUPERLOOP_TICK
```

Multi-task polling means quickly checking if I/O is required or ready and returning immediately if it is not rather than waiting. This allows us to proceed with other tasks in the superloop.

```
pollSwitch()
  …

transmitIfNeeded()
  if gButtonState is CLICKED
    startTransmission("hello")

startTransmission(str)
  copy str to gTxBuffer

completeTransmission()
 // multi-task polling:
 // returns quickly whether char
 // can be transmitted or not and
 // does not wait
 if not at end of gTxBuffer
   if TXREG is full
      return

   // TXREG is empty, send next char
   set TXREG = next char from str
```

Polling features

- Advantages
  - Simple to implement
  - I/O is synchronous, taking place in the application foreground
  - Device priority is easy to change
    - Determined by the order in which software polls (checks) the devices
    - Device priority is used to decide what to handle first when several I/O devices need servicing at the same time
- 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

Standard I/O Synchronization issues

- Some standard I/O synchronization issues and the possible solutions when using polling I/O…
  1. The CPU/app is faster than the I/O device
  2. The I/O device is (temporarily) faster than the CPU/app –
  3. The I/O device needs or supplies data at regular, predictable times
  4. The I/O device needs or supplies data at irregular, unpredictable times

contd

1. The CPU/app is faster than the I/O device
   - The app needs to buffer output data so that it can be written at the slower device rate over the course of multiple superloops
   - The app can do other work while not servicing the IO device
2. The I/O device is (temporarily) faster than the CPU/app
   - This only works if the fast IO device is only active in short bursts of activity
   - The app needs to buffer input data that is arriving quickly without doing any substantial processing work. The app must respond to device input with low latency (delay between stimulus and response) to avoid missing data
   - Later when the input burst has ended, the app can process the buffered data

contd

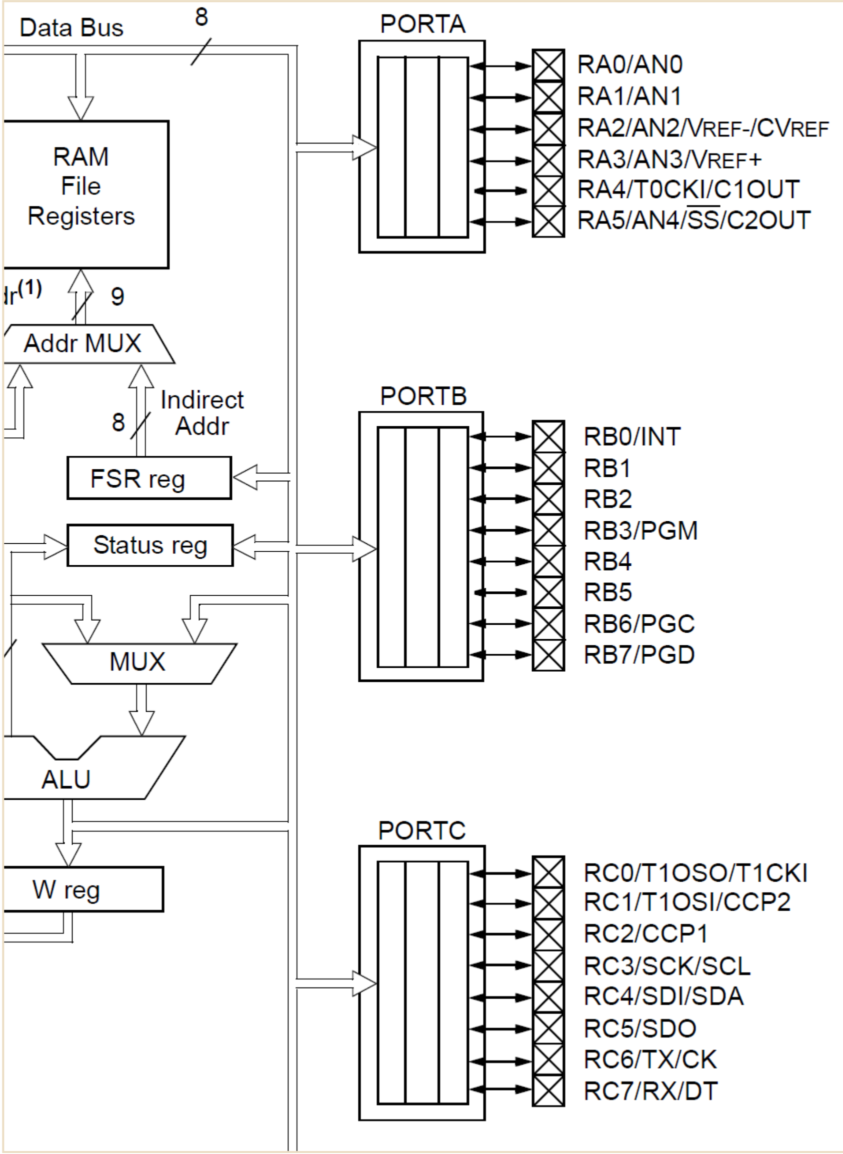
1. The I/O device needs or supplies data at regular, predictable times
   - Poll at regular intervals corresponding to the predictable IO times
2. The I/O device needs or supplies data at irregular, unpredictable times
   - Poll at regular intervals corresponding to the minimum latency required to handle the irregular IO times. E.g. a 10 ms minimum latency on unpredictable button presses requires polling at 10 ms (or smaller) intervals

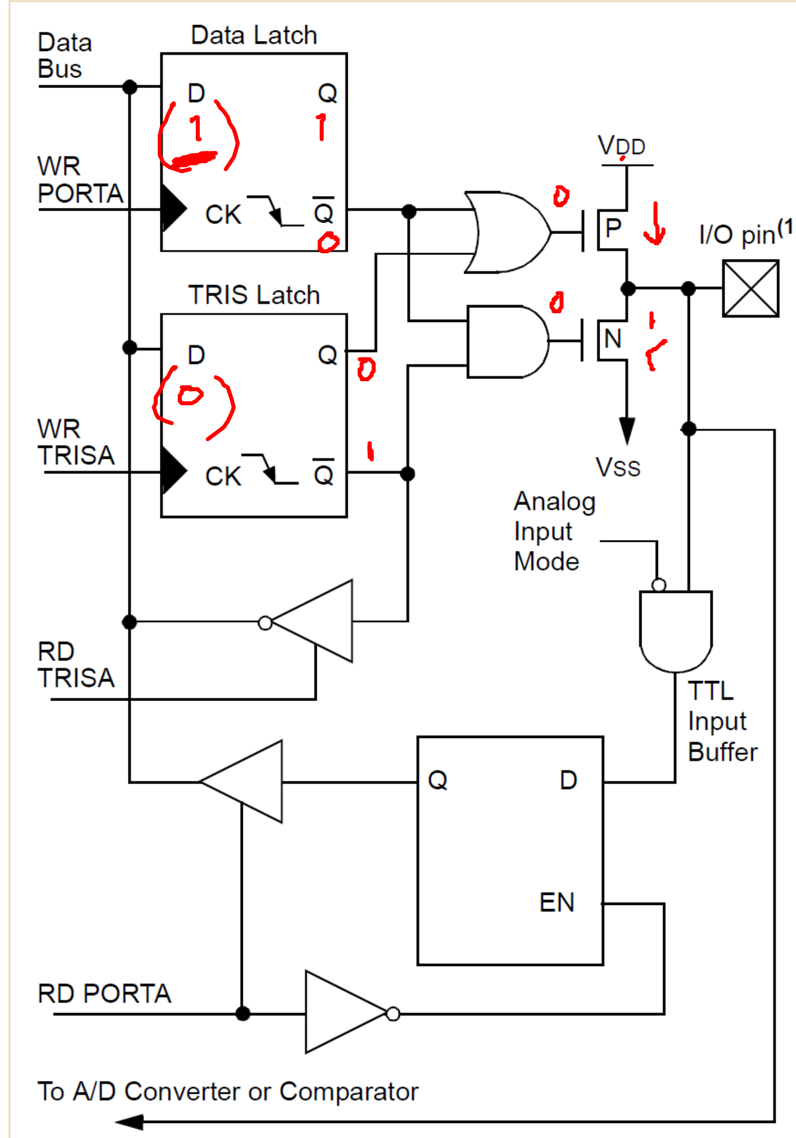
Digital Port Input/Output

Basic digital input/output

- Basic digital I/O devices include
  - Switch (input)
  - Simple keypad (e.g. 3 x 4 phone keypad, input)
  - LED (output)
  - 7 segment display (output)
  - Speaker/beeper (output)
  - Etc.
- All the devices above are not bus aware, so they may be connected to CPU via a digital input or output port as appropriate

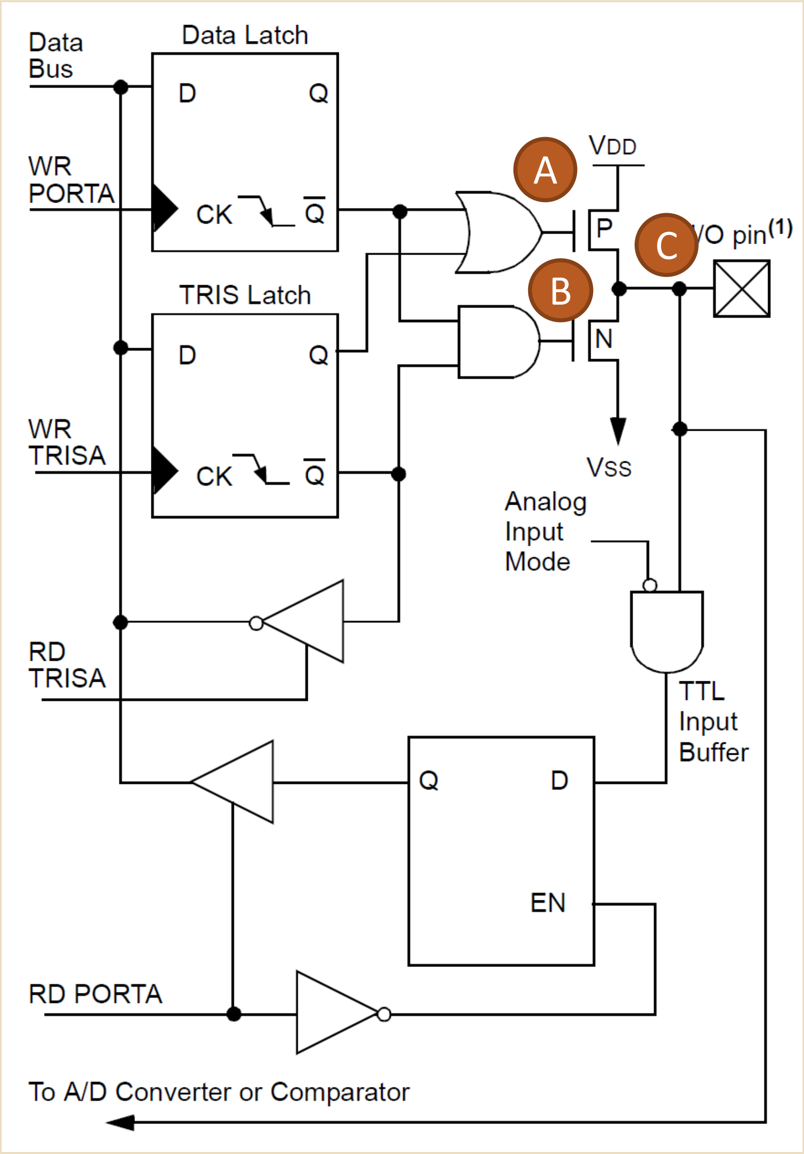
Architecture of “simple” I/O port





Figures reproduced from PIC16F87x data sheet

Architecture of “simple” I/O port



Figures reproduced from PIC16F87x data sheet

Contd.

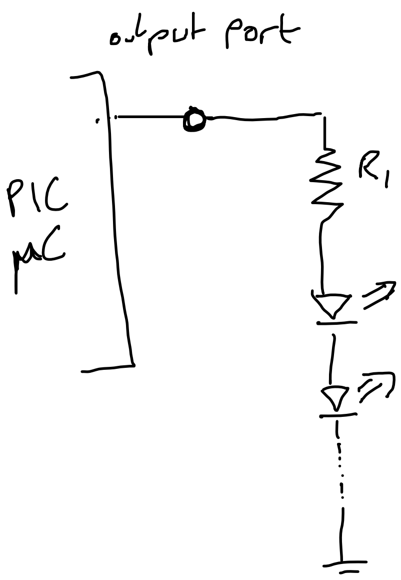
- Input port comprises
  - Buffer: isolate input latch from input current source
  - Input latch: to briefly hold (*) the current value on the latch output (connected to the data bus) so that the CPU can read it.
- Output port comprises
  - Output latch: briefly enabled (*) so that CPU can set output level via the data bus; then disabled so that output is held stable at that value (while data bus reused for next instruction)
- Bidirectional port
  - Contains additional latch and logic to enable just one of the input or output directions

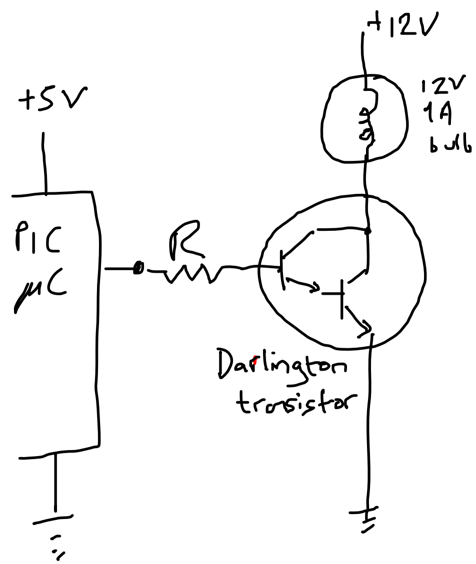
* NOTE: When a latch is enabled values at the latch inputs appear at the outputs almost immediately. When disabled, the latch output is disconnected from the latch input and held steady – changes to the latch input do not appear at the output.

I/O port hardware considerations

- Electrical issues and characteristics
  - Current limits
    - PIC Microcontroller limits current sourced or sunk on any one I/O pin to 25 mA; total current sourced/sunk by any port is 200 mA
    - Affects fan out/fan in for connection to TTL logic gates etc.
    - Generally requires addition of a current limiting resistor when driving low impedance output (like an LED)
  - How are “floating” pins handled?
    - Particularly relevant to switch inputs which are open circuit when not pressed
    - Pull up and pull down resistors
  - Interfacing to different voltage levels
    - PIC microcontroller is normally based on 5V TTL levels
    - Open collector can allow connection of different devices which use a different signal level (e.g. 12 V)

Current limiting





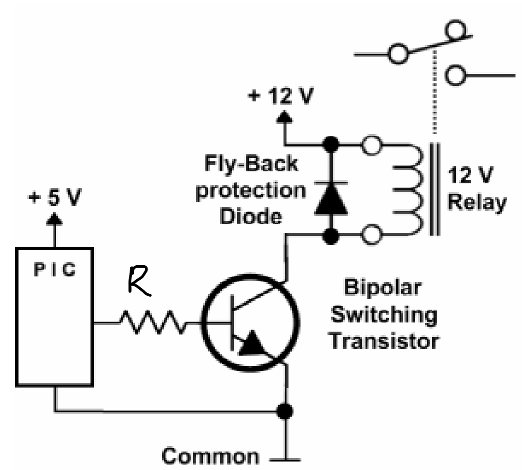
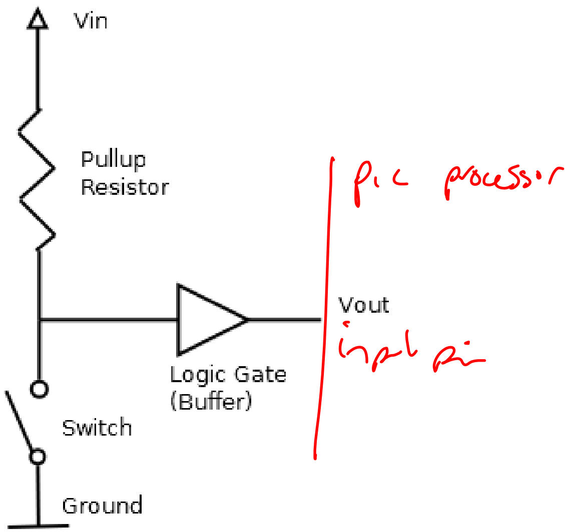


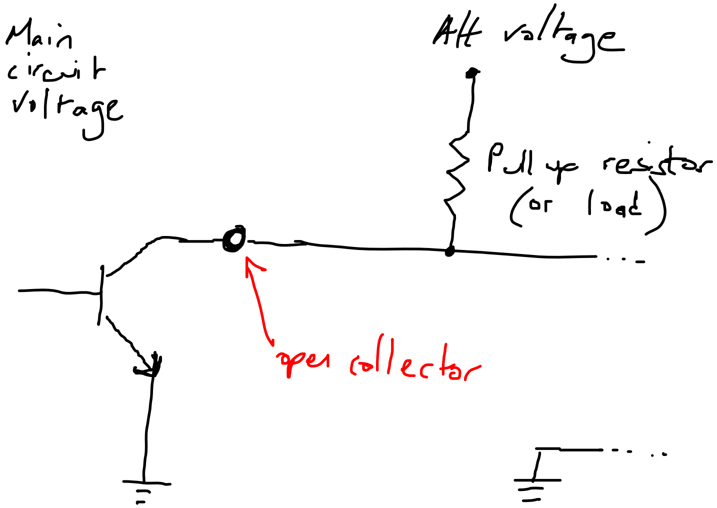
Figure reproduced from The Electronics Club

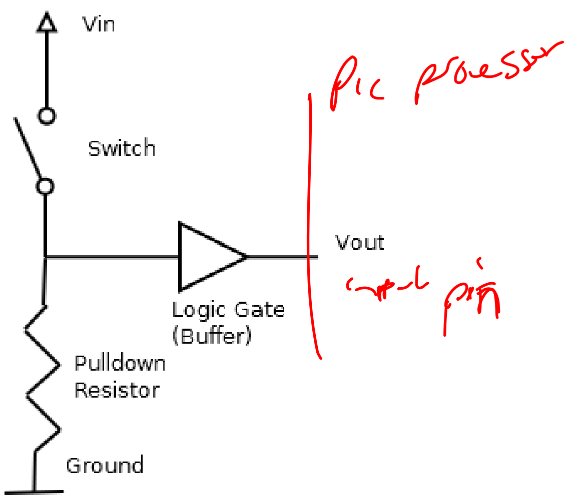
Current flows when output port is at logic high voltage. How should the value of R1 be chosen?
(A LED typically requires about 25mA current and may have about 2V voltage drop.)

If a high current load must be driven, then a relay and/or buffer transistor is needed.
NOTE: resistor still needed to limit port current.

Pull up, pull down, open collector/drain







Figures reproduced from wikimedia

Code examples

- What follows are some simple polling examples that focus on Digital Port I/O
- To keep things simple we assume a single button switch connected to an input pin and a LED connected to an output pin

Example 1

Light LED continuously while button pressed…

```
// General app structure
global gButtonState = OPEN

Main…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK
```

```
setup()
  configure port direction to
   set BUTTON_PIN as input
   set LED_PIN as output
  configure internal pull up resistors
  set LED_PIN to LOW // initially off

pollSwitch()
  if BUTTON_PIN is LOW
    set gButtonState = CLOSED
  else
    set gButtonState = OPEN

updateLED()
  if gButtonState is CLOSED
    set LED_PIN = HIGH // i.e. on
  else
    set LED_PIN = LOW // i.e. off
```

Example 2

Toggle LED on/off continuously while button is pressed…

```
// General app structure
global gButtonState = OPEN

Main…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK
```

```
setup()
  // as before…

pollSwitch()
  if BUTTON_PIN is LOW
    set gButtonState = CLOSED
  else
    set gButtonState = OPEN

updateLED()
  if gButtonState is CLOSED
    toggle LED_PIN
```

else
set LED_PIN = LOW // off

Give 2 possible implementations of toggle LED_PIN
Hint: XOR and if-statement

A	B	xor
0	0	0
0	1	1
1	0	1
1	1	0

toggle:
A ^ 1

Example 3

Toggle LED on/off only when button is released (i.e. is let go after being pressed)

```
// General app structure
global gButtonState = OPEN

Main…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK

setup()
  // as before…
```

```
pollSwitch()
  static prevButtonValue = LOW

  if BUTTON_PIN is LOW
    if prevButtonValue is LOW
      set gButtonState = HELD
    else // button value has changed
      set gButtonState = JUST_PRESSED
      set prevButtonValue = LOW
  else // BUTTON_PIN was HIGH
    if prevButtonValue is HIGH
      set gButtonState = OPEN
    else // button value has changed
      set gButtonState = JUST_RELEASED
      set prevButtonValue = HIGH

updateLED()
  if gButtonState is JUST_RELEASED
    toggle LED_PIN
```

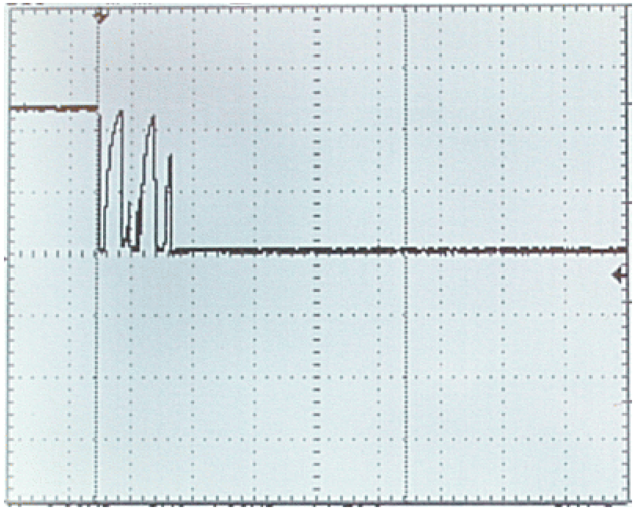
Questions

1. Based on example 1, how would you light the LED continuously when the button was not pressed and darken it when the button was pressed?
2. Based on example 2, how would you flash the LED repeated only when the button was not pressed?
3. Based on example 2, how fast will the LED flash?
4. Based on example 3, how would you toggle the LED only when the button is just pressed (and not while it is held down)?

Digital Port Input/Output

Switch Debounce

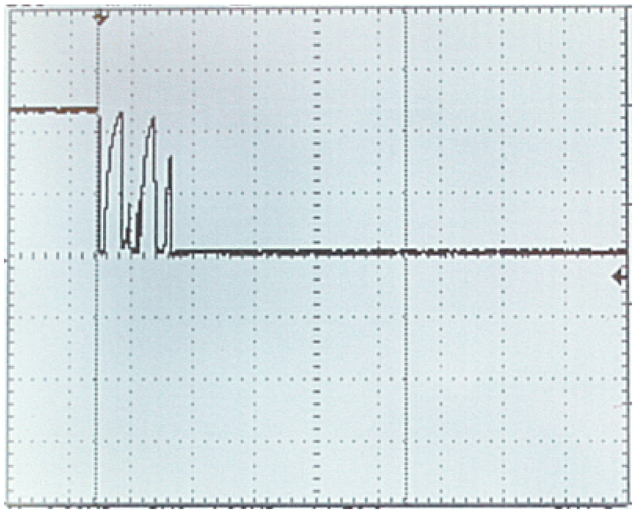
Switch debounce in hardware



Voltage from a switch that has not been "debounced"

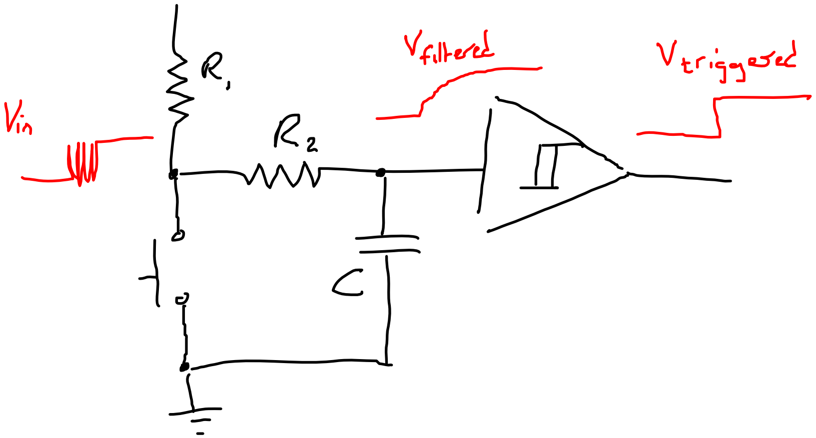
Figure reproduced from [Micah Carrick's blog]

Switch debounce in hardware



Voltage from a switch that has not been "debounced"

Figure reproduced from [Micah Carrick's blog]

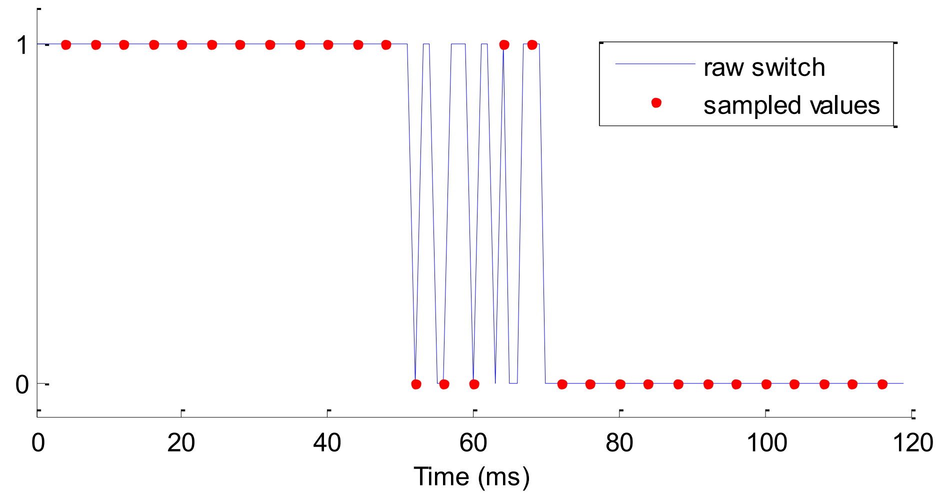


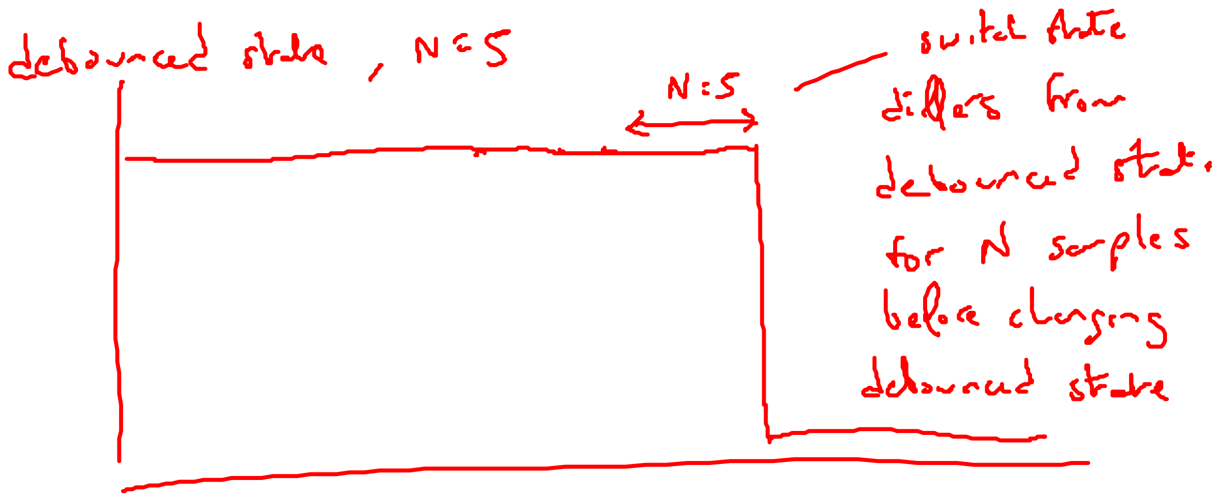
Hardware debounce solution using RC circuit and Schmitt trigger

For more details see: A guide to debouncing - http://www.ganssle.com/debouncing.htm

Software debounce using counting

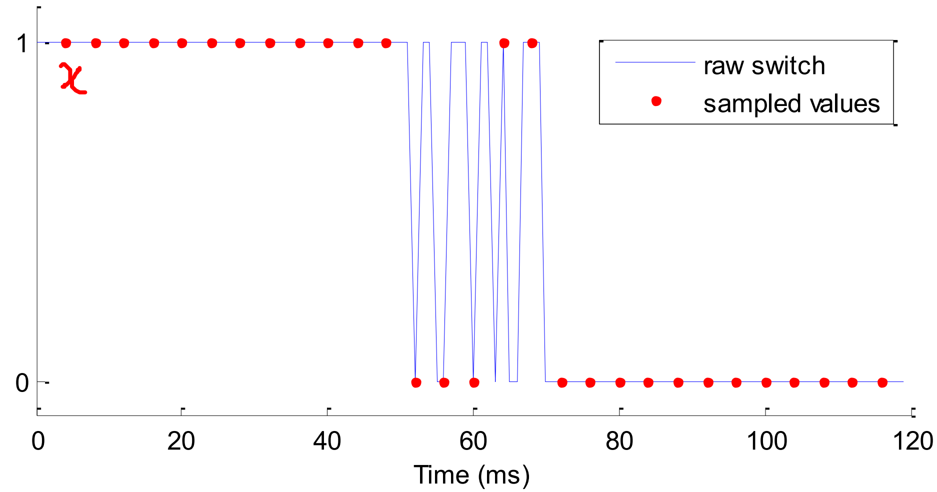
- count based
  - Measure switch value repeatedly
  - value must be same for N polls to be considered debounced/ stable

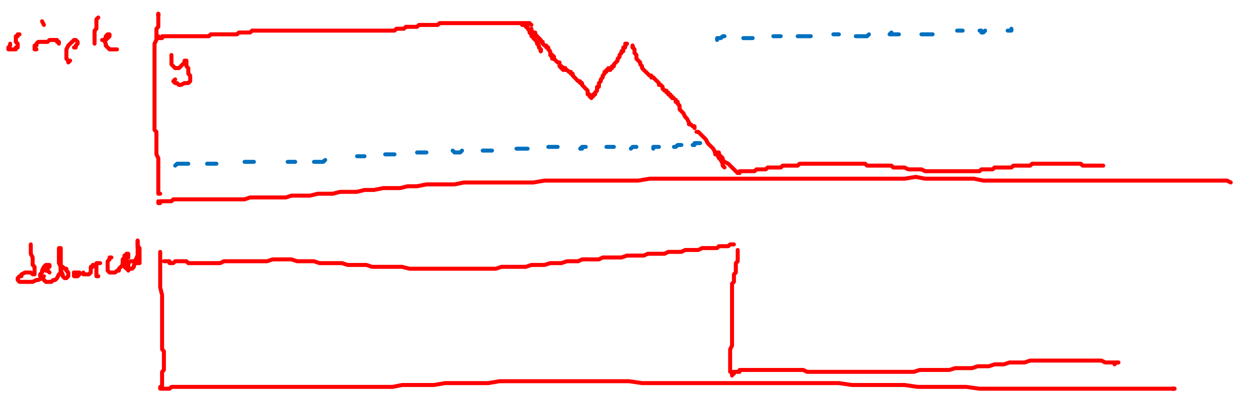




Software debounce using digital filter

- Digital filter based
  - Digital simulation of low pass filter with schmitt trigger
  - Has good EMI filtering properties





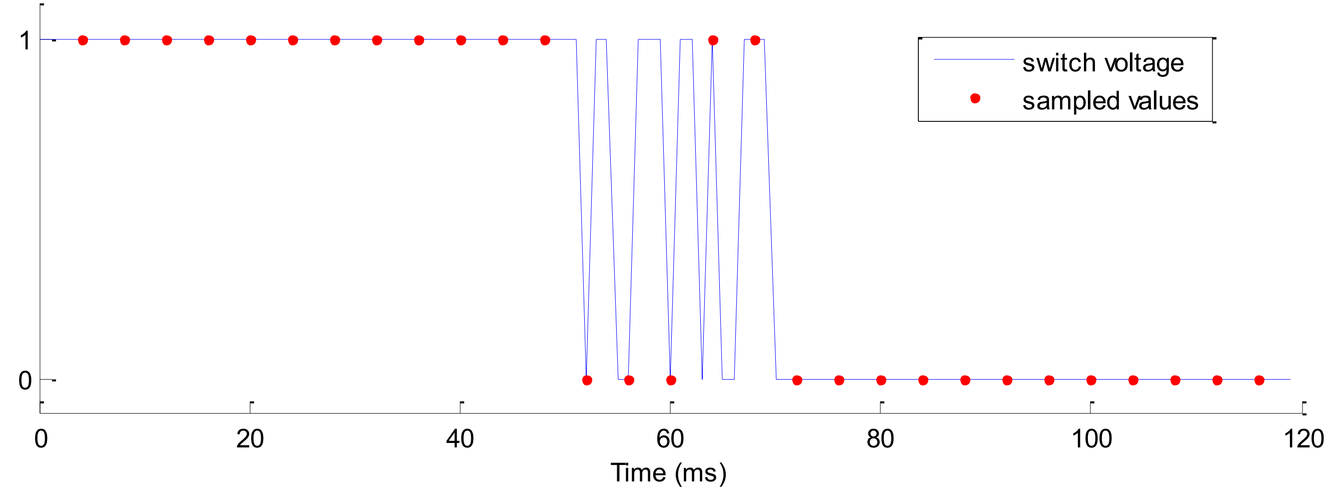
Simple digital filter is $y_n = \alpha x + (1-\alpha) y_{n-1}$

Switch (button) debounce in software

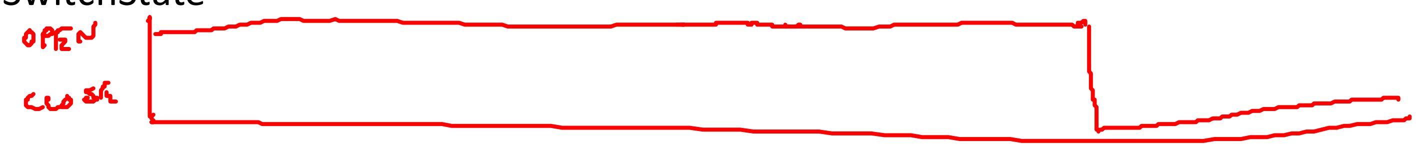

```
// Simple edge trigger algorithm (there are many possible variations)
// Verify signal is stable using a counter
// DEBOUNCE_TICKS should be chosen to give enough time for bounces
// to disappear (often 10-20ms but depends on button mechanics)
// checkSwitch would be called once per superloop
global gDebouncedSwitchState = OPEN // default value at start of app

// get debounced state of switch and indicate whether it has changed
checkSwitch()
  static debounceCounter = DEBOUNCE_TICKS // initial state
  debouncedSwitchChanged = FALSE // default unless change verified
  rawSwitchState = read switch input pin from IO port

  if rawSwitchState == gDebouncedSwitchState // switch state stable?
    debounceCounter = DEBOUNCE_TICKS // prepare to debounce next change
  else // switch state changed, so wait for it to become stable
    decrement debounceCounter
    if debounceCounter is 0 // debouncing all done?
      gDebouncedSwitchState = rawSwitch // accept the change
      debounceCounter = DEBOUNCE_TICKS // prepare to debounce next change
      debounceSwitchChanged = TRUE

  return debounceSwitchChanged
```

Software debounce – example walk through

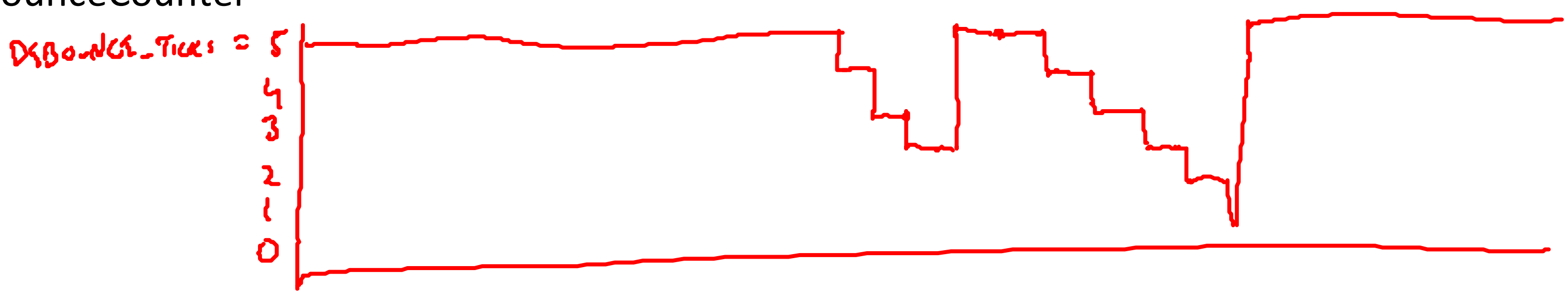
rawSwitchState
(red dots)



gDebouncedSwitchState



debounceCounter



debouncedSwitchChanged

Digital Port Input/Output

Common/useful polling patterns

General port I/O polling pseudocode


```
// General app structure
Main…

setup()
  configure port direction
  configure internal pull up resistors

loop()
    pollInputPort()
    pollOutputPort()
    …
    delay SUPERLOOP_TICK

pollInputPort()
  // see next slides

pollOutputPort()
  // see next slides
```

Polling design patterns

- What follows are some very common polling problems with some typical solutions on following slides
- Timing or repeating problems occur when the rate at which your superloop repeats and I/O function is called doesn't match the rate or delay or repetitions that is suitable for the I/O operation you want to do

- Typical patterns…
  1. Poll to detect the instantaneous input state. Do something continuously (every superloop) while the input is in a particular state (e.g. while a button is pressed)
  2. Poll to detect specific input changes. Take action once only when the specific change has occurred (e.g. edge triggered only when a button changes from not-pressed to pressed)
  3. Poll with a period of N times the superloop period

Pattern 1: poll to detect instantaneous state, act continuously


```
// General app structure
global gSwitchClosed = FALSE

Main…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK

setup()
  setup button and LED…
```

```
pollSwitch()
  // active low button (assuming
  // hardware debounce)
  if BUTTON_PIN is LOW
    set gSwitchClosed = TRUE
  else
    set gSwitchClosed = FALSE

updateLED()
  if gSwitchClosed is TRUE
    set LED_PIN = HIGH // i.e. on
  else
    set LED_PIN = LOW // i.e. off
```

Note that the updateLED code sets the LED value on every single superloop (i.e. continuously based on the instantaneous state of the switch).

Both the button (input) and LED (output) are polled once per superloop

Pattern 2: poll to detect specific changes, act once only when specific change has occurred


```
// General app structure
global gSwitchJustClosed = FALSE

Main…

loop()
    pollSwitch()
    updateLED()
    delay SUPERLOOP_TICK

setup()
  setup button and LED…
```

```
pollSwitch()
  static prevButtonPin = HIGH

  // detect HIGH->LOW transition
  // on active low button (assuming
  // hardware debounce)
  if BUTTON_PIN is LOW
        AND prevButtonPin is HIGH
    set gSwitchJustClosed = TRUE
  else
    set gSwitchJustClosed = FALSE

  // remember prev for next time
  set prevButtonPin = BUTTON_PIN

updateLED()
  if gSwitchJustClosed is TRUE
    toggle LED_PIN
```

Note that the updateLED code only toggles the LED once on the button HIGH to LOW transition. In all other cases (including while the button stays LOW) the LED is untouched.
This pattern requires a static variable to detect the previous input state so as to detect transitions.

Self test questions


```
global gState = 0
const BITMASK =                                   // TODO: choose suitable value

// assume the following functions are called from the superloop…

// Q1. set global gState to 1 continuously while
// PORTB, bit 3 is 0 and to 0 otherwise. (Hint: see bitmask)
pollInputActContinuously()




// Q2. set global gState to 1 only when PORTB, bit 3 transitions from
// 1 to 0 and set it to 0 otherwise
pollInputActOnChange()
```

C coding self test question


```
// Q. How would you implement the following line of pseudocode in C
//    using a bitmask
// if PORTB, bits 4 and 6 matches binary x0x1 xxxx





// Q. how would you implement the following lines of pseudocode in C
// static prevValue = STARTUP_VALUE
// if PORTB, bit 6 differs from prevValue
// …
// set prevValue bit 6 from inputPort
```

Pattern 3: poll on every Nth superloop only


```
// General app structure

Main…

loop()
    …
    slowPoll()
    delay SUPERLOOP_TICK

setup()
  setup button and LED…
```

```
slowPoll()
  static periodCounter = N

  // only do the real work every
  // Nth time this function is called
  decrement periodCounter
  if (periodCounter is 0)
     readFromDevice()
     reset periodCounter = N


readFromDevice()
  …
```

Note the real work of polling the device is done in readFromDevice but this function is only called every Nth time that slowPoll is called.

Self test software questions

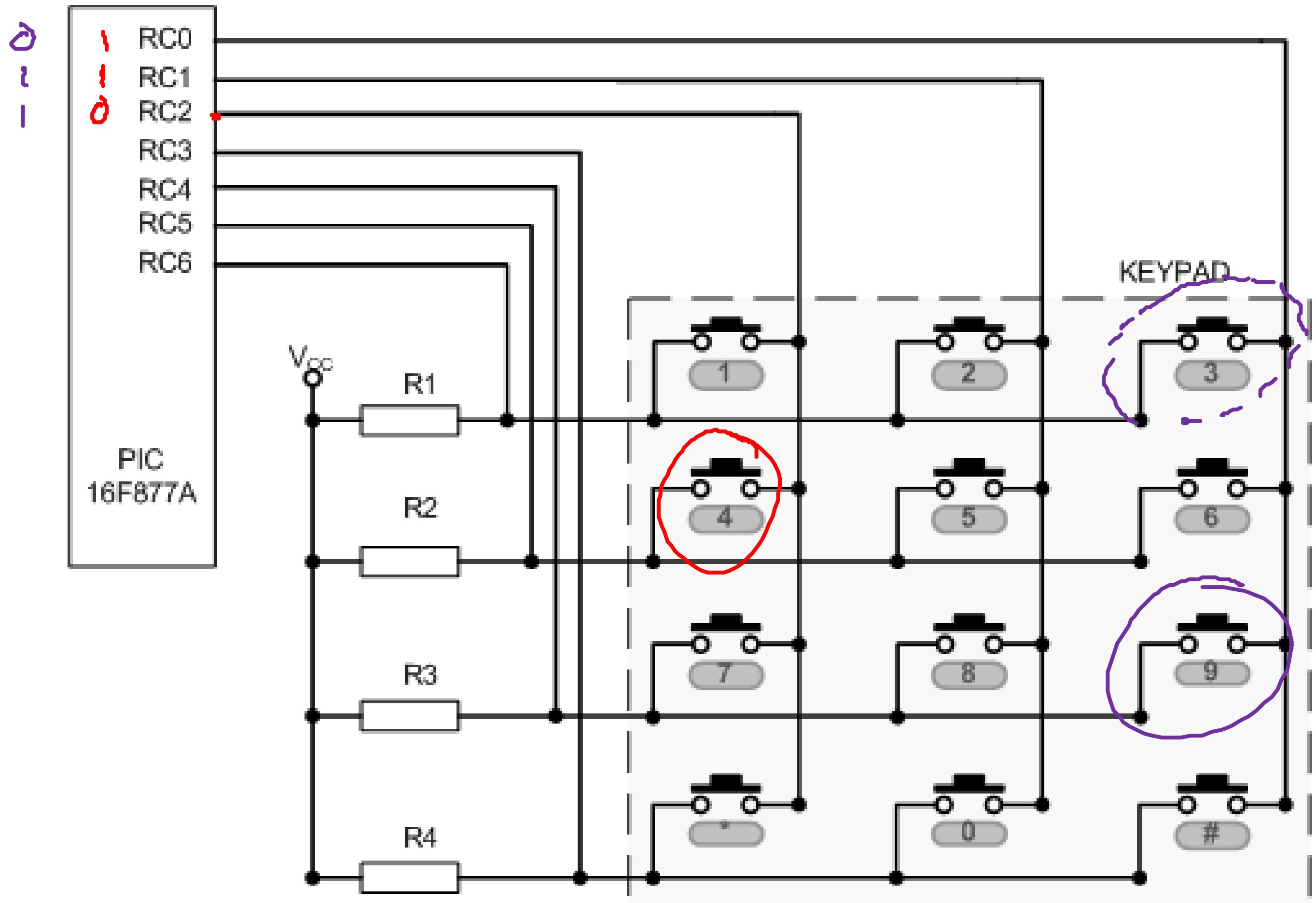
In the following

- *assume the superloop runs (and calls the corresponding polling function) once every 5 ms*
- *Make sure the superloop runs every 5 ms – i.e. do not place a delay in your polling function which would upset the superloop timing*

Q. show how set a global variable true only when an active low switch transitions from open to closed assuming hardware debounce

Q. show how, whenever a global variable become true, to light an LED for 50 ms and then set the variable false again

Exam 2012-2013 Q3



The 4x3 keypad is organised as 4 rows and 3 columns of switches. Port C is configured such that RC0 to RC2 are outputs connected to the keypad columns and RC3 to RC6 are inputs connected to the keypad rows.

contd

1. If RC2 (column 0) is driven low, other columns are high, and no key is pressed, what values would you expect to read in RC3 to RC6? Explain your answer.
2. If RC2 is driven low, other columns are high, and the '4' key is pressed, what values would you expect to read in RC3 to RC6? Explain your answer.
3. Explain the purpose of resistors R1 to R4. What is the significance the resistor values?

(1) All 4 inputs read as high as no current flowing and no voltage drop across R1 .. R4

(2) RC5 = 0, RC3, 4, 6 = 1 because SW4 ties R2 to RC2 = 0
other rows still tied to Vcc (Logic 1)

(3) Pull up resistors
resistor values determines how current is limited

contd

Write the pseudocode for a scanKeys function which detects the row and column of the key pressed (if any). To detect whether a key on a particular column has been pressed, the MCU drives only that column low. With the column driven low, the MCU reads the value of the 4 rows to determine whether a key was pressed or not. By repeating the procedure for all 3 columns any single key press can be detected.

(Note: the identity, e.g. ASCII code, of the detected key could be looked up in a simple 2 dimensional array, using the row and column returned from the scanKeys function as indexes into the array. This is outside the question scope)

```
scanKeys():
  for col = 0 to 2
    set RC0, RC1, RC2 = HIG
    if col is 0, set RC2 LOW
    if col is 1, set RC1 LOW
    if col is 2, set RC0 LOW
    set rowPins = PORTC & 0x78
    if rowPins & 0x08, row = 4
    if rowPins & 0x10, row = 3
    :
    else row = NONE
```

pins 3, 4, 5, 6 ⇒ 0b01111000 = 0x78