CENG-336 Introduction to Embedded Systems Development

Input/Output

Spring 2015

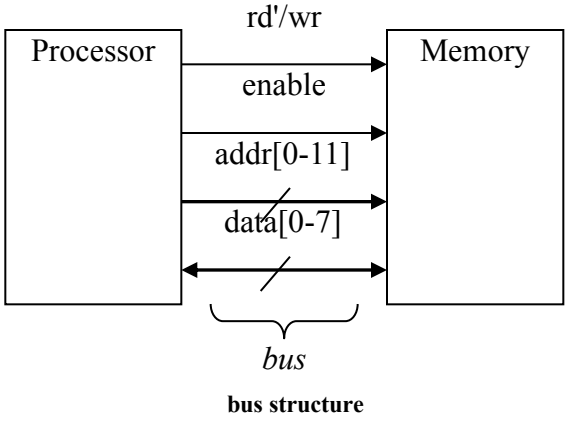
Introduction

- Embedded system functionality aspects
  - Processing
    - Transformation of data
    - Implemented using processors
  - Storage
    - Retention of data
    - Implemented using memory
  - Communication
    - Transfer of data between processors and memories
    - Implemented using I/O Ports and/or buses
    - Called interfacing

A simple bus

- Wires:
  - Uni-directional or bi-directional
  - One line may represent multiple wires

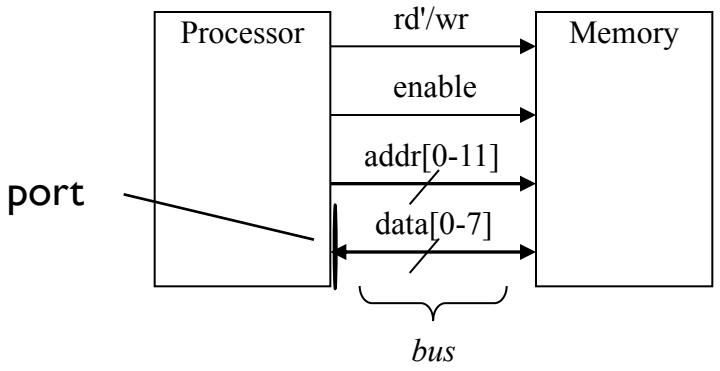
- Bus
  - Set of wires with a single function
    - Address bus, Data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication



bus structure

Ports

- Conducting device on periphery
- Connects bus to processor or memory

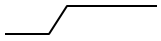


- Often referred to as a (collection of) pin
  - Actual pins on periphery of IC package that plug into socket on printed-circuit board
  - Sometimes metallic balls instead of pins
  - Today, metal "pads" connecting processors and memories within single IC

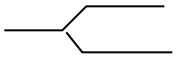
- Single wire or set of wires with single function
  - E.g., 12-wire address port

Timing Diagrams

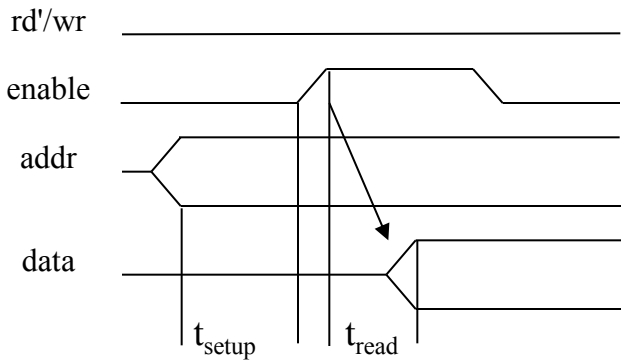
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis
- Control signal: low or high



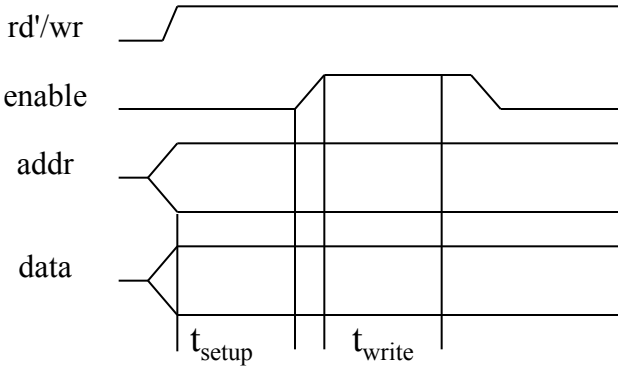
  - May be active low (e.g., go', /go, or go_L)
  - Use terms assert (active) and deassert
  - Asserting go' means go=0
- Data signal: not valid or valid



- Protocol may have subprotocols
  - Called bus cycle, e.g., read and write
  - Each may be several clock cycles
- Read example
  - rd'/wr set low, address placed on addr for at least tsetup time before enable asserted, enable triggers memory to place data on data wires by time tread



read protocol

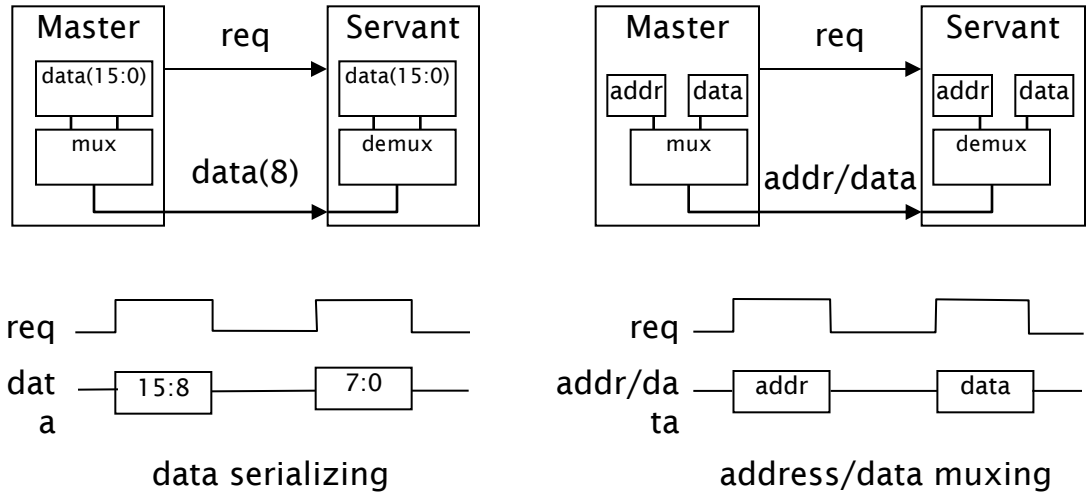


write protocol

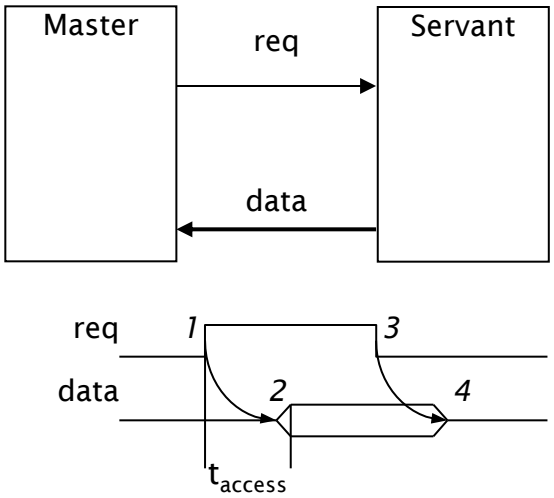
Protocol Concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses: special kind of data
  - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time

Time-multiplexed data transfer

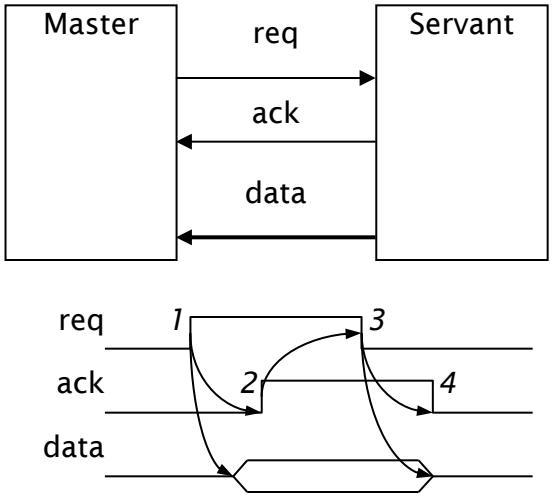


Protocol Concepts: Control Methods



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

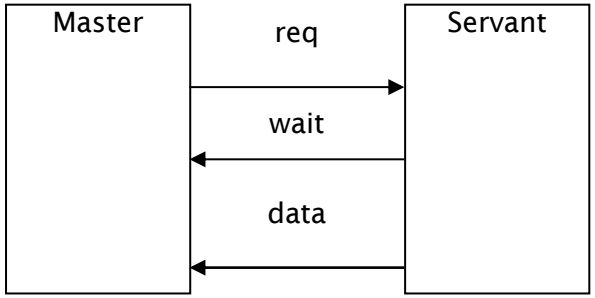
Strobe protocol

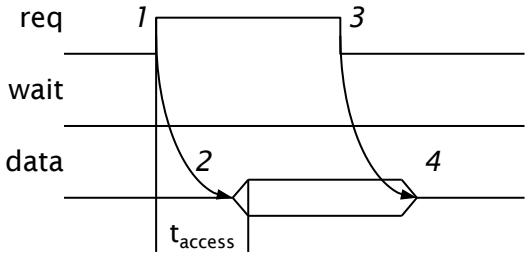


1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

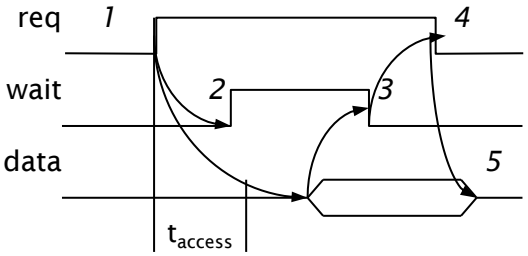
A strobe/handshake compromise





1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
   (wait line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

Fast-response case

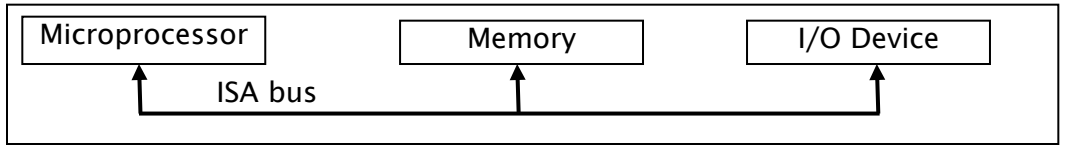


1. Master asserts *req* to receive data
2. Servant can't put data within t_{access}, **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

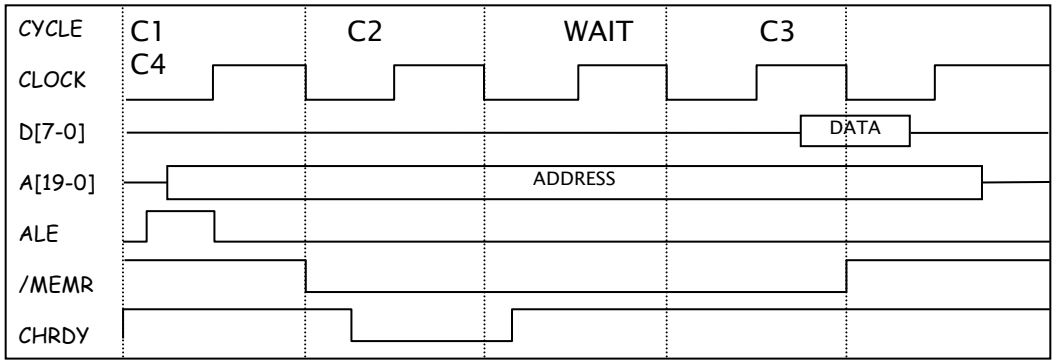
Slow-response case

ISA bus protocol – memory access

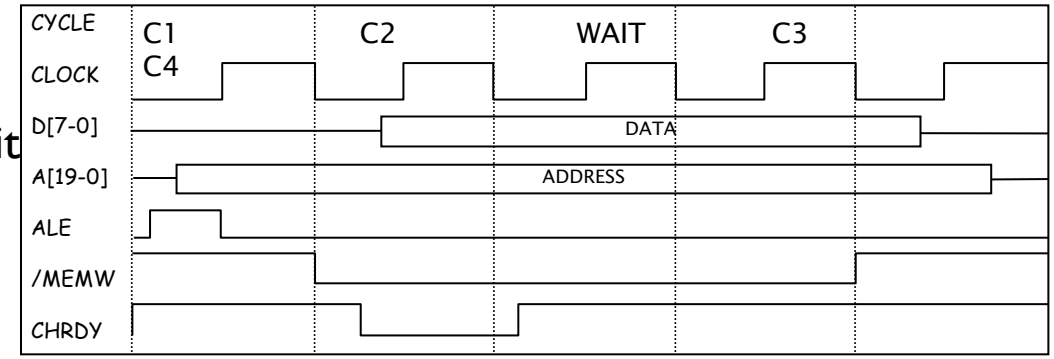
- ISA: Industry Standard Architecture
  - Common in 80x86's
- Features
  - 20-bit address
  - Compromise strobe/handshake control
    - 4 cycles default
    - Unless CHRDY deasserted – resulting in additional wait cycles (up to 6)



memory-read bus cycle



memory-write bus cycle

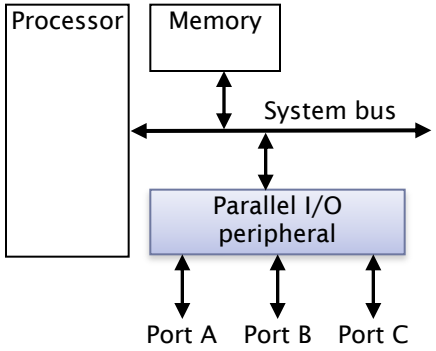


Interfacing: I/O addressing

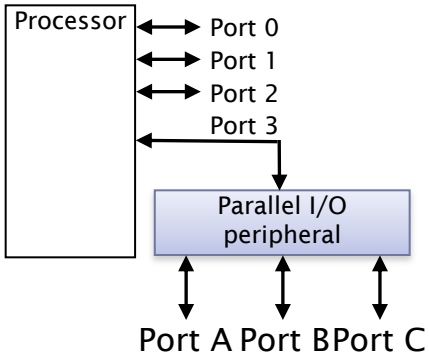
- A microprocessor communicates with other devices using some of its pins
  - Port-based I/O (parallel I/O)
    - Processor has one or more N-bit ports
    - Processor's software reads and writes a port just like a register
    - E.g., P0 = 0xFF; v = P1.2; -- P0 and P1 are 8-bit ports
  - Bus-based I/O
    - Processor has address, data and control ports that form a single bus
    - Communication protocol is built into the processor
    - A single instruction carries out the read or write protocol on the bus

Compromises / Extensions

- Parallel I/O peripheral
  - When processor only supports bus-based I/O but parallel I/O needed
  - Each port on peripheral connected to a register within peripheral that is read/written by the processor
- Extended parallel I/O
  - When processor supports port-based I/O but more ports needed
  - One or more processor ports interface with parallel I/O peripheral extending total number of ports available for I/O
  - e.g., extending 4 ports to 6 ports in figure



Adding parallel I/O to a bus-based I/O processor



Extended parallel I/O

Memory-mapped vs. Standard I/O

- Processor talks to both memory and peripherals using same bus – two ways to talk to peripherals
  - Memory-mapped I/O
    - Peripheral registers occupy addresses in same address space as memory
    - e.g., Bus has 16-bit address
      - lower 32K addresses may correspond to memory
      - upper 32k addresses may correspond to peripherals
  - Standard I/O (I/O-mapped I/O)
    - Additional pin (M/IO) on bus indicates whether a memory or peripheral access
    - e.g., Bus has 16-bit address
      - all 64K addresses correspond to memory when M/IO set to 0
      - all 64K addresses correspond to peripherals when M/IO set to 1

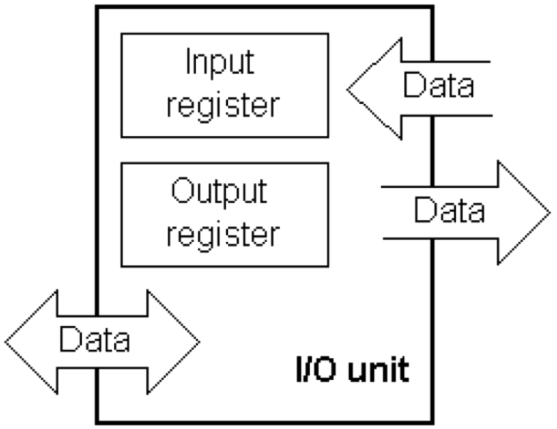
Memory-mapped vs. Standard I/O

- Memory-mapped I/O
  - Requires no special instructions
    - Assembly instructions involving memory like MOV and ADD work with peripherals as well
    - Standard I/O requires special instructions (e.g., IN, OUT) to move data between peripheral registers and memory
- Standard I/O
  - No loss of memory addresses to peripherals
  - Simpler address decoding logic in peripherals possible
    - When number of peripherals much smaller than address space then high-order address bits can be ignored
      - smaller and/or faster comparators

I/O on PIC 18F8722

Ports / Registers

- Term "port" refers to a group of pins on a microcontroller which can be accessed simultaneously (in parallel), or on which we can set the desired combination of zeros and ones, or read from them an existing status.
- Physically, port is a register inside a microcontroller which is connected by wires to the pins of a microcontroller.
- Ports represent physical connection of Central Processing Unit with an outside world.
- Microcontroller uses them in order to monitor or control other components or devices.



Example of a simplified input-output unit that provides communication with external world

PIC I/O

- All input and output on the PIC is performed via the I/O ports.

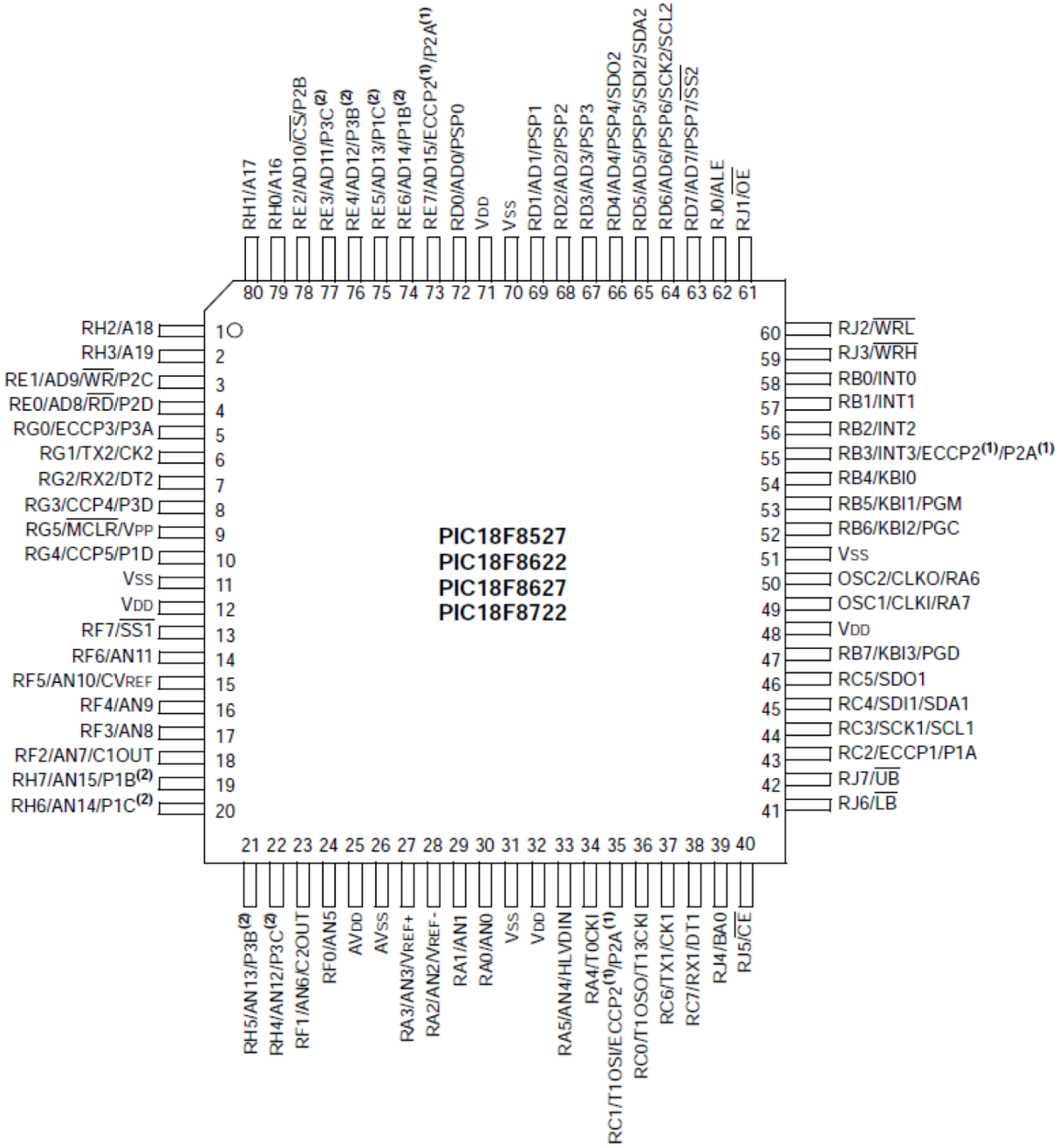
- I/O ports are like RAM locations with wires leading from the bits to the pins of the microchip.
  - Memory-mapped

Typical Ports

- Ports A,B,C,D,E,F,H,J are a 8-bit wide, bi-directional ports.
- PORTG is a 6-bit wide bi-directional port.

- Pin functionality "overloaded" with other features
- Each pin can be individually configured for input or output.

- Configuration controlled through TRISx Register
- Pin input/output accessed through PORTx register
- Port data latch is accessed through LATx register


PIC18F8527
PIC18F8622
PIC18F8627
PIC18F8722
1 RH2/A18
2 RH3/A19
3 RE1/AD9/WR/P2C
4 RE0/AD8/RD/P2D
5 RG0/ECCP3/P3A
6 RG1/TX2/CK2
7 RG2/RX2/DT2
8 RG3/CCP4/P3D
9 RG5/MCLR/VPP
10 RG4/CCP5/P1D
11 VSS
12 VDD
13 RF7/SS1
14 RF6/AN11
15 RF5/AN10/CVREF
16 RF4/AN9
17 RF3/AN8
18 RF2/AN7/C1OUT
19 RH7/AN15/P1B(2)
20 RH6/AN14/P1C(2)
21 RH5/AN13/P3B(2)
22 RH4/AN12/P3C(2)
23 RF1/AN6/C2OUT
24 RF0/AN5
25 AVDD
26 AVSS
27 RA3/AN3/VREF+
28 RA2/AN2/VREF-
29 RA1/AN1
30 RA0/AN0
31 VSS
32 VDD
33 RA5/AN4/HLVDIN
34 RA4/T0CKI
35 RC1/T1OSI/ECCP2(1)/P2A(1)
36 RC0/T1OSO/T13CKI
37 RC6/TX1/CK1
38 RC7/RX1/DT1
39 RJ4/BA0
40 RJ5/CE
41 RJ6/LB
42 RJ7/UB
43 RC2/ECCP1/P1A
44 RC3/SCK1/SCL1
45 RC4/SDI1/SDA1
46 RC5/SDO1
47 RB7/KBI3/PGD
48 VDD
49 OSC1/CLKI/RA7
50 OSC2/CLKO/RA6
51 VSS
52 RB6/KBI2/PGC
53 RB5/KBI1/PGM
54 RB4/KBI0
55 RB3/INT3/ECCP2(1)/P2A(1)
56 RB2/INT2
57 RB1/INT1
58 RB0/INT0
59 RJ3/WRH
60 RJ2/WRL
61 RJ1/OE
62 RJ0/ALE
63 RD7/AD7/PSP7/SS2
64 RD6/AD6/PSP6/SCK2/SCL2
65 RD5/AD5/PSP5/SDI2/SDA2
66 RD4/AD4/PSP4/SDO2
67 RD3/AD3/PSP3
68 RD2/AD2/PSP2
69 RD1/AD1/PSP1
70 VSS
71 VDD
72 RD0/AD0/PSP0
73 RE7/AD15/ECCP2(1)/P2A(1)
74 RE6/AD14/P1B(2)
75 RE5/AD13/P1C(2)
76 RE4/AD12/P3B(2)
77 RE3/AD11/P3C(2)
78 RE2/AD10/CS/P2B
79 RH0/A16
80 RH1/A17

I/O Ports: Access

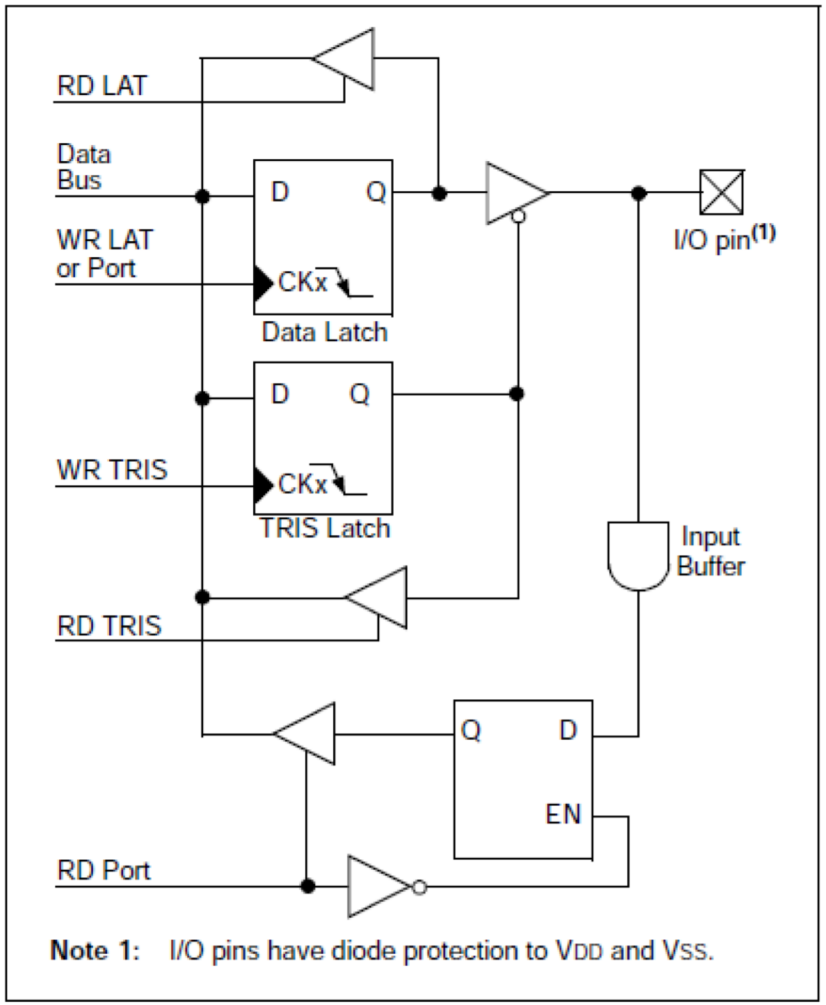
- I/O ports on a PIC are memory mapped.
  - This means that to read from or write to a port on a PIC the program must read from or write to a special RAM location.
  - To access PORTA, the program must access the RAM at address location F80h (or access bank 80h)

I/O Ports: Direction

- The I/O ports on the PIC can be used as either inputs or outputs.
- Configuring a port for input or output is done by setting or clearing the data direction register for the port (TRISx).
- On the PIC each bit of each port has a data direction bit.
  - Therefore it is possible to set some bits within a port as inputs and others within the same port as outputs.
- I/O pin direction (input or output) is controlled by the data direction register, called the TRIS register.
  - TRIS<x> controls the direction of PORT<x>.
  - A '1' in the TRIS bit corresponds to that pin being an input, while a '0' corresponds to that pin being an output.
  - An easy way to remember is that a '1' looks like an I (input) and a '0' looks like an O (output).
  - Note that these registers are all in bank 1

I/O Ports: General Structure

- TRIS controls direction
- PORT register
  - Write on PORT changes output value (Data latch)
  - Read on PORT read from external PIN
- LAT register
  - Both Read and Write accesses the value being output (Data latch)
  - Used for read/modify/write operations



PORTA / TRISA

- PORTA 0 0 0 0 0 0 0 0
- TRISA 0 0 1 0 0 1 0 1
- DIRECTION OUT OUT IN OUT OUT IN OUT IN

- TRISA is located at address F92h (Bank 15, address 92), or access bank address 92h
- PORTA in F80h (or access bank 80h)
- LATA in F89h (or access bank 89h)

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
PORTA	RA7(1)	RA6(1)	RA5	RA4	RA3	RA2	RA1	RA0	61
LATA	LATA7(1)	LATA6(1)	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0	60
TRISA	TRISA7(1)	TRISA6(1)	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	60

PORTA Analog/Digital setup

- Since PORTA can be analog or digital, you also need to tell the PIC that PORTA is digital.
- This is done by writing 0x0F to ADCON1
- PCFG bits control in detail which pins are analog

PCFG<3:0>	AN15(1)	AN14(1)	AN13(1)	AN12(1)	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
0000	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	D	D	A	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	D	D	D	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	D	D	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	D	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	D	D	D	A	A	A	A	A	A	A	A	A
0111	D	D	D	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

REGISTER 21-2: ADCON1: A/D CONTROL REGISTER 1

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

Port Configuration Example 1

```
        clrf    0x85,A    ; TRISA = 0
        clrf    0x86,A    ; TRISB = 0
        clrf    0x87,A    ; TRISC = 0

        movlw   0x0f
        movwf   0xC1,A    ; ADCON1 = 0x0f

        clrf    0x82,A    ; PORTC = 0
        clrf    0x81,A    ; PORTB = 0
        clrf    0x80,A    ; PORTA = 0
L0:
        goto    L0        ; stop (infinite loop)
```

- The above example configures Ports A, B and C as outputs, all Port A pins as digital and clears all bits.

Port Configuration Example 2

```
movlw   0x0F            ;Defining input and output pins
movwf   TRISB           ;Writing to TRISB register

bsf     PORTB, 4, A     ;PORTB <7:4>=1
bsf     PORTB, 5, A
bsf     PORTB, 6, A
bsf     PORTB, 7, A
```

- The above example shows how pins 0, 1, 2, and 3 are designated input, and pins 4, 5, 6, and 7 for output, after which PORTB output pins are set to one.

initialize_portc Function Example

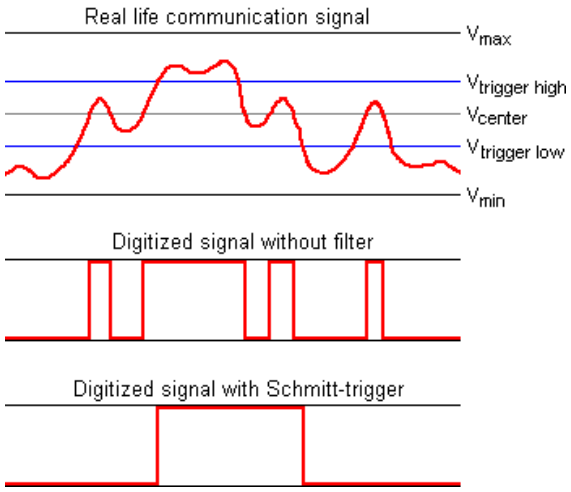
```
clrf    PORTC       ; clear the data register to
                    ; cause outputs to be 0
movlw   0xBF        ; each bit with a 1 is input,
                    ; otherwise output – Bit 6 is output
                    ; B '10111111'
movwf   TRISC       ; setup the port
return              ; exit
```

Electrical Properties for Ports

- The RA4 pin is a Schmitt Trigger input and an open drain output. All other RA port pins have TTL input levels and full CMOS output drivers.

- All pins on PORTC, PORTD, PORTE, PORTF, PORTG, PORTH, PORTJ are also Schmitt Trigger inputs

- Schmitt Trigger is an input filtering mechanism for more robust digital operation.



Alternate Functions

- To add flexibility and functionality to a device, some pins are multiplexed with an alternate function(s).
  - These functions depend on which peripheral features are on the device.
  - In general, when a peripheral is functioning, that pin may not be used as a general purpose I/O pin.

Alternate Functions

- PORTA is a 8-bit I/O port (shared with A/D converter)
  - The operation of each pin is selected by clearing/setting the control bits in the ADCON1 register (A/D Control Register1).

- Pin RA4 is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin.

PortD Parallel Slave Port

- PORTD can be configured as an 8-bit wide microprocessor port (parallel slave port) by setting control bit PSPMODE (PSPCON<4>).
- 8-bit Parallel Slave Port - function allow another processor to read from a data buffer in the PIC.
- In Slave mode, it is asynchronously readable and writable by the external world through RD control input pin RE0/RD and WR control input pin RE1/WR.

Basic I/O Programming

Basic Embedded Code Structure

- Simplest form for embedded code is a simple loop, called Round-Robin (or Cyclic Executive):

In C:

```
while (1) {
        task_1();
        ...
        task_n();
}
```

In assembly:

```
LOOP:
        call task_1
        ...
        call task_n
        bra LOOP
```

- Executes n tasks in sequence, within an infinite loop that never exits.

Round-Robin Architecture

- Embedded software is often "event driven"
  - An "external" event triggers software to respond (e.g. a digital input went from 0 to 1)
  - An action is taken, which might trigger additional events
  - When no events are present, software simply waits, possibly putting the CPU into sleep mode
- In the Round-Robin architecture
  - Each task should "listen" to a specific set of events through polling (i.e. explicit check with an if statement)
  - When a task notices an event, it responds by doing whatever is necessary, which might result in other events

Round-Robin Example

- Simple example: Use a button to toggle an LED
- Goal: Write a program that will monitor a button connected to Port B bit 0 and when it becomes true, toggle an LED connected to Port B pin 1
- Two tasks:
  - Button task: Monitor the button state
  - LED task: Toggle the LED when the button tasks says so

Main Program

```
unsigned char toggle_flag;      /* Tells led_task() to toggle LED */
unsigned char button_state;     /* Remembers previous state of the button */

void init_system() {
        toggle_flag = 0;        /* Initialize global variables. */
        button_state = 0;
        TRISB = 0xFD;           /* Pin 1 output, rest is input */
}
void main() {
        init_system();
        while(1) {              /* Here is the Round-Robin structure */
                button_task();
                led_task();
        }
}
```

Button Task

```
void button_task() {
        switch (button_state) {
        case 0:  /* Previous button state was 0 */
                if  (PORTB & 0x01 != 0) {
                        button_state = 1;
                        toggle_flag = 1;
                }
                break;

        case 1: /* Previous button state was 1 */
                if  (PORTB & 0x01 == 0) button_state = 0;
                break;
        }
}
```

- Implements a "State Machine". More on this later.

LED Task

```
void led_task() {
        if (toggle_flag != 0) {
                PORTB = PORTB ^ 0x02;
                toggle_flag = 0;
        }
}
```

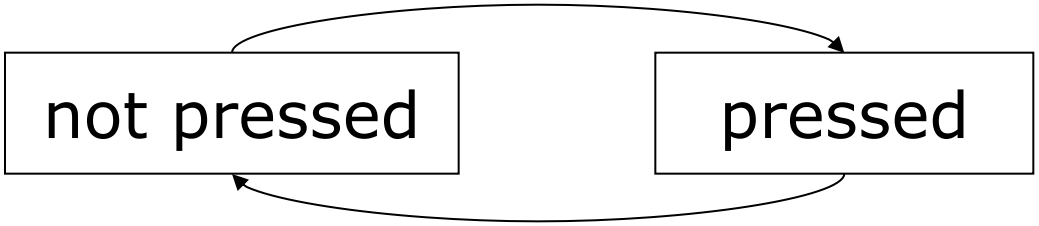
- Monitors the toggle flag, performs the requested toggle if detected.
- Event sources in this design:
  - Button connected to Port B (externally triggered)
  - toggle_flag variable (inter-task communication)

Timeline of Events

- Nothing happens until the button is pressed. If statements in both tasks are false
- When the button is pressed, the button task switches to state 1, and alerts the LED task through the toggle flag
- The LED task toggles the output and resets the flag.
- Once again, nothing happens until the next event, which is the button release!
- When the button is released, the button task switches to state 0, and starts waiting for another press.

Observations: State Machines

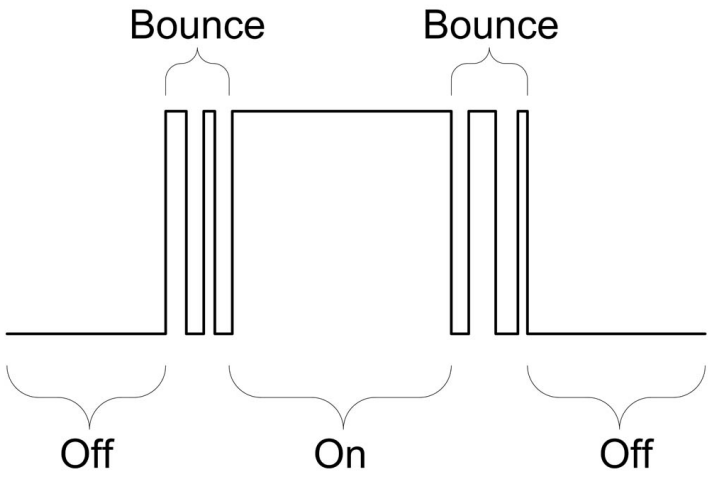
- The button task adopts what is called a "state machine" structure. It remembers past events through a carefully selected set of states, implemented in a switch case statement



- This represents an internal state, that attempts to track what the physical state of the button looks like
- State machines are essential to embedded software design. Use them!

Observations: Button Behavior

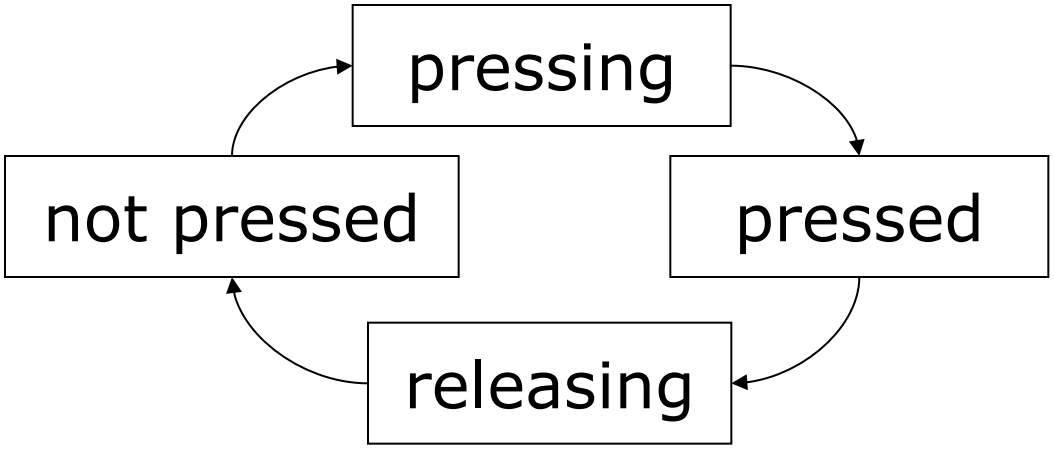
- Normally, buttons do not behave this nicely. The "bounce", generating multiple pulses



- Additional logic and states are needed in the button task for "debouncing", which involves waiting for the bounce to end.

Observations: Button Behavior

- For example, a debouncing button task might have a more complex state machine:



- "Pressing" and "releasing" states should wait for a short period for bouncing to end.
- WARNING: Waiting here does not mean another while loop!

Avoiding Busy Waits

- Ideally, the only busy wait (i.e. waiting for something to happen in a while loop) should be the while(1) in main().
- All other waiting should take the form of counters, or monitoring of timers (more on timers later)

```
void button_task() {
        ...
        case PRESSING:
                if (++wait_counter > DELAY_COUNT)
                        button_state = PRESSED;
                break;
        case PRESSED:
                ...
}
```

Avoiding Busy Waits

- Eliminating all busy waits except main() greatly helps debugging
  - When something is stuck, you know it's because no events are received
  - You can still use your debugger to single-step through all tasks, inspecting their internal states
- Even when a single task is "stuck" in some undesired state, remaining tasks will keep working
- This is similar to the difference between a preemptible and a non-preemptible operating system.