C and Embedded Systems

- A µP-based system used in a device (i.e, a car engine) performing control and monitoring functions is referred to as an **embedded system**.
  - The embedded system is invisible to the user
  - The user only indirectly interacts with the embedded system by using the device that contains the µP
- Most programs for embedded systems are written in C
  - Portable – code can be retargeted to different processors
  - Clarity – C is easier to understand than assembly
  - compilers produce code that is close to manually-tweaked assembly language in both code size and performance

So Why Learn Assembly Language?

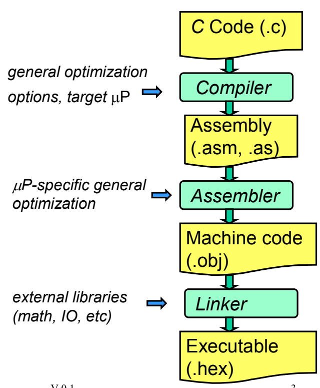
- The way that C is written can impact assembly language size and performance
  - i.e., if the **int** data type is used where **char** would suffice, both performance and code size will suffer.
- Learning the assembly language, architecture of the target µP provides performance and code size clues for compiled C
  - Does the uP have support for multiply/divide?
  - Can it shift only one position each shift or multiple positions? (i.e, does it have a *barrel shifter*?)
  - How much internal RAM does the µP have?
  - Does the µP have floating point support?
- Sometimes have to write assembly code for performance reasons.

C Compilation

This general tool chain is used for all high-level programming languages.

C is portable because a different compiler can target a different processor. Generally, some changes are always required, just fewer changes than if trying port an assembly language program to a different processor.

Assembly language or machine code is not portable.



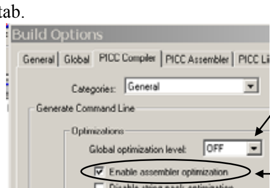
PICC Lite C Compiler

- Programs for hardware experiments (labs 6-13) will be written in *C*
- Will use the PICC Lite *C* Compiler
  - Demo version of professional *C* compiler from Hi-Tech Software (www.htsoft.com)
  - **Excellent** compiler, generates very good code
- When creating a project, select the "Hi-Tech PICC Toolsuite" as the language toolsuite
  - See Experiment #5 in Lab manual for full instructions on using PICC Lite

PICC *C* Optimizations

By default, all code optimizations are turned off in during compilation.

To enable assembly level optimizations (-O flag), do
Project →Build →Options Project
to open the build options window. Click on the PICC Compiler tab.



Set value between 1 (lowest effort) to 9 (highest effort). Generally, > 3 does not help much.

Check this to enable the –O option

PICC Lite *C* Optimization Results (Lab #13)

Optimization	Code Size (words)	Bank 0 Ram (bytes)	Bank 1 Ram (bytes)
None (default)	1425	94	76
-O	1228	94	76
-O –Zg3 Level 3 global optimization	1198	94	76
-O –Zg9 Level 9 global optimization	1198	94	76

Referring to Special Registers

```
#include <pic.h>
```

Must have this include statement at top of each file. Will include a processor-specific header file based on device chosen in MPLAB.

This header file contains *#defines* for all special registers:

```
#static volatile unsigned char   PORTB  @ 0x06;
```

found in *pic1687x.h* in PICC Lite installation directory — special register — memory location in PIC

```
PORTB = 0x80;
```

In C code, can refer to special register using the register name

bittst, *bitclr*, *bitset* Macros

```
#define bitset(var,bitno) ((var) |= (1 << (bitno)))
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
#define bittst(var,bitno) (var & (1 << (bitno)))
```

Include these utility *C* macros at the top of all of your *C* files (does not matter where, just have them defined before you use them).

Example usage:

```
bitset(PORTB,7); /* MSB ← 1 */
bitclr(PORTB,0); /* LSB ← 0 */

if (bittst(PORTB, 0)) {
  /* do something */
}
```

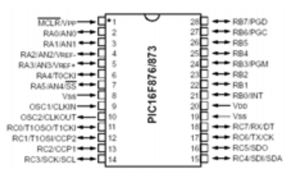
Under PICC Lite, these macros compile to the equivalent PIC *bsf*, *bcf*, *btfsc*, *btfss* instructions.

PICF16873

Hardware lab exercises will use the PICF16873 (28-pin DIP)

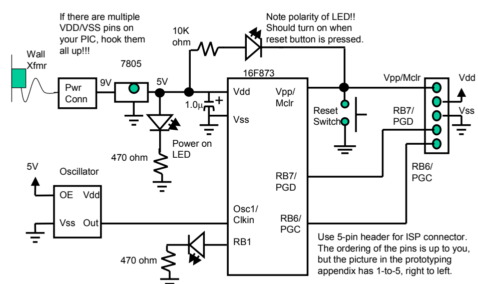
Note that most pins have multiple functions.

Pin functions are controlled via special registers in the PIC.

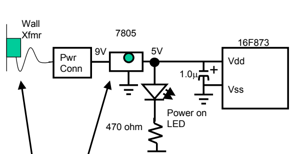


In-Circuit Programming (ICP) will be used to program memory contents from a PC without removing the device from the protoboard.

Initial Hookup



Powering the PIC

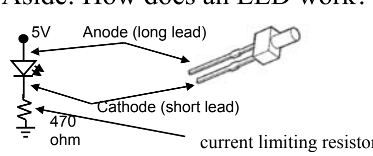


Wall transformer provides 9V DC unregulated (unregulated means that voltage can vary significantly depending on current being drawn). Maximum current from Xfmr is 650 mA.

The 7805 voltage regulator provides a regulated +5V. Voltage will stay stable up to maximum current rating of device.

With writing on device visible, input pin (+9 v) is left side, middle is ground, right pin is +5V regulated output voltage.

Aside: How does an LED work?

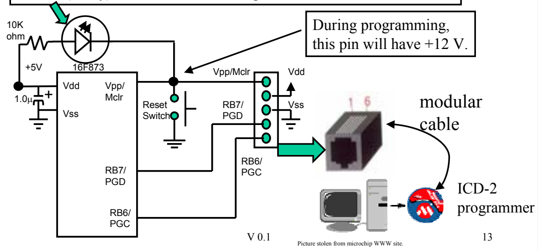


A diode will conduct current (turn on) when the anode is at approximately 0.7V higher than the cathode. A Light Emitting Diode (LED) emits visible light when conducting – the brightness is proportional to the current flow.

Current = Voltage/Resistance ~ (5v – 0.7v)/470 Ω = 9.1 mA

In-Circuit Programming

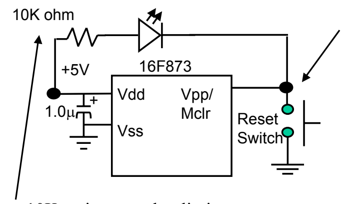
This diode is **very important** – it protects the other devices connected to the +5V supply from the +12 V that is applied during programming. The diode does not conduct if the cathode voltage > anode voltage. Be sure you have the polarity correct; the diode should turn on (dimly) when the reset button pressed.



V 0.1

Picture stolen from microchip WWW site.

Reset

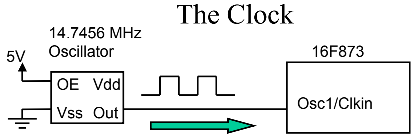


When reset button is pressed, the Vpp/Mclr pin is brought to ground. This causes the PIC program counter to be reset to 0, so next instruction fetched will be from location 0. All μPs have a reset line in order to force the μP to a known state.

10K resistor used to limit current when reset button is pressed. Diode will be very dim when reset switch is pressed because current ~ 0.5 mA

V 0.1

The Clock



Will use an external oscillator IC to provide the clock for the PIC. The 'weird' frequency provides common baud rates for serial communication when divided down internally.

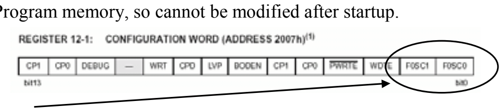
The internal instruction frequency is:
14.7456 MHz /4 = 3.6863 MHz

The PIC can also use an external RC network (cheap, but not very accurate) or an external crystal (more components, usually needs two external capacitors as well).

V 0.1

Configuration Word

The **configuration word** contains 13 bits that specifies various PIC processor options that affect operation. This is located in Program memory, so cannot be modified after startup.



If these lower 2 bits are 01 (XT option), then PIC will expect an external oscillator to provide the clock.

We will discuss the meaning of the other options as it is necessary.

V 0.1

Setting the Configuration Word

```
#define DATA_EEMEM_PROTECT_DISABLE 0x0100
#define XT_OSC 0x0001
#define DISABLE_DEBUG 0x0800

#define CP0              0x1010
#define CP1              0x2020

/* CP0,CP1  code protect off*/
/* other bits zero which means WDT disabled
   Low Voltage prm disabled, brownout disabled
   power up timer enabled */

__CONFIG((CP1 | CP0) | DATA_EEMEM_PROTECT_DISABLE   |
XT_OSC | DISABLE_DEBUG);
```

This *C* code causes the configuration word to be programmed as 0x3931. This is what should be used for all hardware labs unless specified otherwise.

V 0.1

Parallel Port I/O

The simplest type of I/O via the PIC external pins is **parallel port** I/O.

The PICF873 has three parallel ports:

- PORTA – 6 bits, bidirectional
- PORTB – 8 bits, bidirectional (except RB0, input only)
- PORTC – 8 bits, bidirectional

We will use PORTB pins most of the time because the PORTA, PORTC pins will be used for other functions beside parallel I/O.

Each pin on these ports can either be an input or output – the data direction is controlled by the corresponding bit in the TRISA, TRISB, TRISC registers ('1' = input, '0' = output).

V 0.1

PORTB Example

Set the upper four bits of PORTB to outputs, lower four bits to be inputs:

```
TRISB = 0x0f;
```

Drive RB4, RB5 high; RB6, RB7 low:

```
PORTB = 0x30;
```

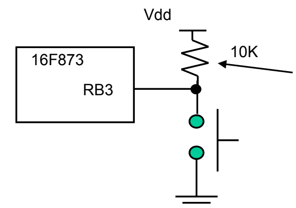
Wait until input RB2 is high:

```
while (!bittst(PORTB,2)) ;
```

Wait until input RB3 is high:

```
while (bittst(PORTB,3)) ;
```


Switch Input



External pullup

When switch is pressed RB3 reads as ‘0’, else reads as ‘1’.

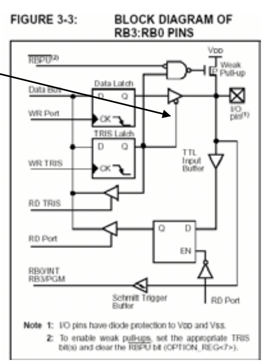
If pullup not present, then input would float when switch is not pressed, and input value may read as ‘0’ or ‘1’ because of system noise.

PORTB Pin Diagram

If TRIS bit a 0, output active

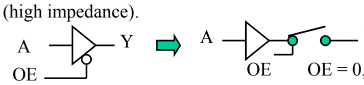
If pin is programmed to be an OUTPUT (TRIS bit = 0), and a read is done, will read the *last value* written to the PORT.

If pin is programmed to be in INPUT (TRIS bit = 1), will always read what the external pin digital value is. A write to an input pin has no effect.

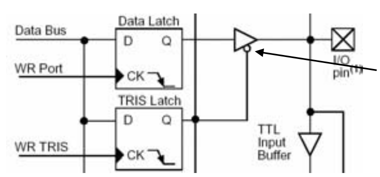


Aside: Tri-State Buffer (TSB) Review

A tri-state buffer (TSB) has input, output, and output-enable (OE) pins. Output can either be ‘1’, ‘0’ or ‘Z’ (high impedance).



OE = 0, then switch closed
OE = 1, then switch open

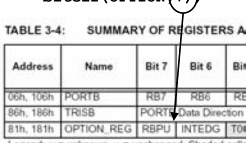


Output Enable (OE) of TSB. If asserted, output = input. If negated, output is high impedance (output disconnected)

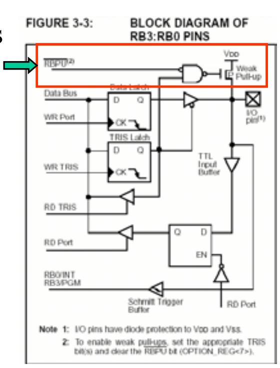
PORTB weak pullups

Can enable weak pullups on all RB pins configured to be inputs by clearing the RBPU bit in the OPTION register

```
bitclr(OPTION,7);
```



Removes the need for an external pullup.



PORTA Parallel IO

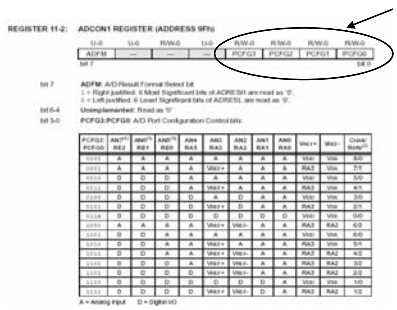
On the PIC16F873, the PORTA RA0:RA3 and RA5 pins are also used for as the inputs to the analog-to-digital converter module.

By default, they are analog input pins, not bi-directional digital I/O pins. If a read is done on these pins while they are configured as analog inputs, a ‘0’ will always be returned.

To enable RA0:RA3, RA5 pins to functions as digital pins, the ADCON1 (A/D configuration register) must be set to the value 0x06:

```
/* configure port A to be all digital inputs */
TRISA = 0xff;
ADCON1 = 0x06;
```

PORT A Pin Configuration



Bits 3:0 of ADCON1 control the configuration of the PORTA pins in terms of digital vs. analog.

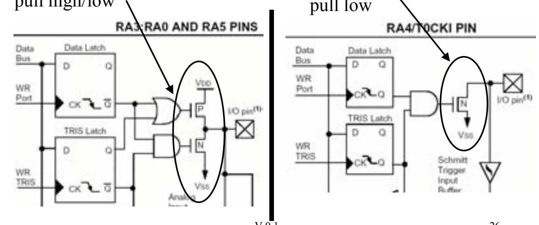
The datasheet in the A/D section has a complete description.

RA4 Pin: Open Drain Output

RA4 is different from RA0:RA3, RA5 in that it is an open drain output. RA4 can only pull LOW, it cannot pull high.

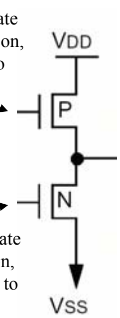
P/N transistors both present, can pull high/low

Only N present, can only pull low

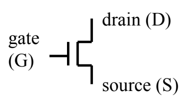


Aside: P/N CMOS Transistor Review

A ‘0’ (Vss) on gate turns this device on, pulls output up to VDD

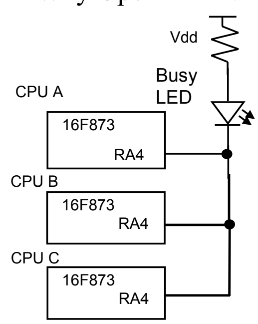


A ‘1’ (VDD) on gate turns this device on, pulls output down to VSS



The logic that drives the gate inputs of the P/N transistors will never turn both transistors on at the same time.

Why Open Drain?



Useful because can tie open-drain outputs together without external logic, only an external pullup.

Assume CPUs A,B,C are all working on different tasks, and want to know when all are finished.

When working on a task, a CPU asserts RA4 low. When finished, negate RA4. If any CPU is busy, LED will turn on. If all CPUs finished, LED will turn off.

Cannot do this with non-open drain output because of clash of some outputs driving low, some high.

This type of connection is often called a **wired-or**. If CPU A *or* B *or* C is busy, then LED is on.

PORTC Parallel IO

- We will not look at PORTC Parallel IO
- PORTC pins shared with many other functions of the PICF873
- If parallel IO is needed, will always use PORTB first, then PORTA if needed
  - Do not use RB0 because this pin has a special interrupt function.

What do you have to know?

- How LEDs, Switches work
- How pullup resistors work and when they are needed.
- How to use an external oscillator with the PIC
- Parallel port usage of PORTA, PORTB
- How to use the weak pullups of PORTB
- How N/P type transistors work
- How a Tri-state buffer works
- How an open-drain output works and what it is useful for