

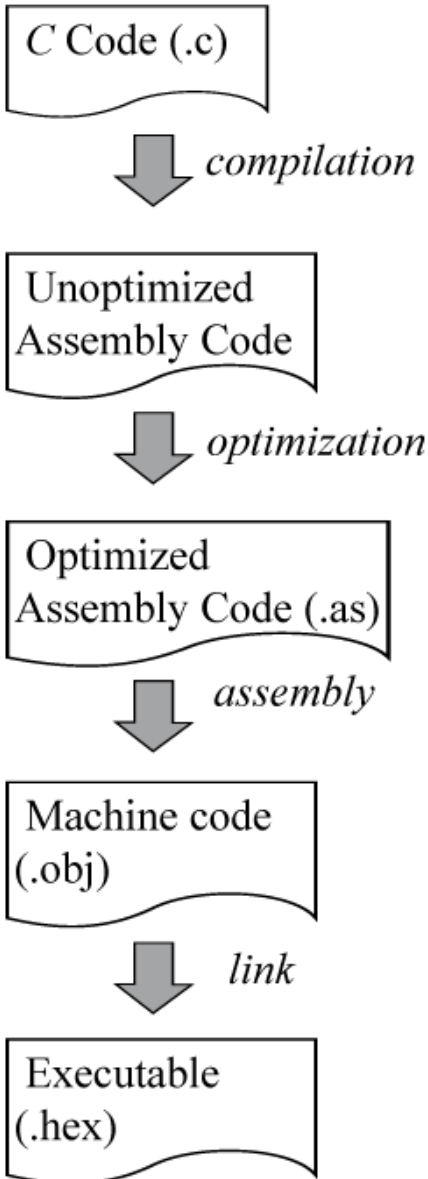
# C and Embedded Systems

- A  $\mu$ P-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  $\mu$ P
- 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

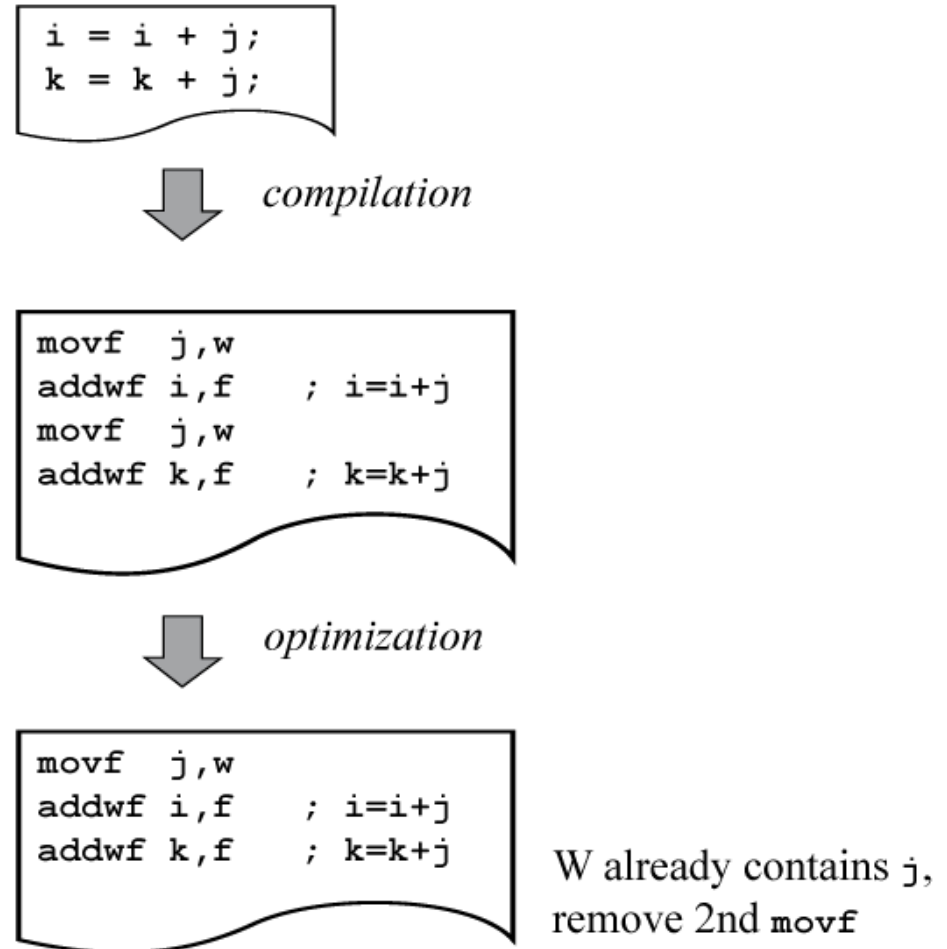
# So Why Learn Assembly Language?

- 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  $\mu$ P provides performance and code size clues for compiled C
  - Does the  $\mu$ P 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  $\mu$ P have?
  - Does the  $\mu$ P have floating point support?
- Sometimes have to write assembly code for performance reasons.

From .c to .hex



Example Optimization



# C Compilation

# PICC18 C Compiler

- Programs for hardware experiments (labs 6-13) are written in *C*
- Will use the PICC18 *C* Compiler
  - Company is Hi Tech ([www.htsoft.com](http://www.htsoft.com))
  - **Excellent** compiler, generates very good code
- See lab manual for instructions on *C* compilation.

# Using the PICC18 compiler

To compile a file called 'myfile.c', do:

Specifies the particular PIC18 device

```
picc18 -O -a200 -18F242 myfile.c
```

-O option turns on compiler optimizations (reduces number on instructions generated).

-a200 locates the code starting at location 0x200 in memory (must do this for code programmed by serial bootloader).

Output file produced is called 'myfile.hex' (A hex file is a ASCII-hex representation of the machine code).

Other compile options:

```
picc18 -O -a200 -lf -18F242 myfile.c
```

Required for *printf()* statements containing 'longs', 'floats'.  
The option -lf is letter L, letter F, in lowercase.

# Importing *.hex* files within the MPLAB<sup>®</sup> IDE

- If you have a *.hex* file produced outside of the MPLAB<sup>®</sup> IDE, it can be imported and executed within the MPLAB<sup>®</sup> IDE.
- Select the correct PIC18 device by using “Configure → Select Device”
- Use the command “File → Import” to import a *.hex* file
  - Browse to the directory that contains your *.hex* file, select it, and click on ‘OPEN’
  - If you get the error ‘Unexpected End of File’, then on a Unix machine such as ‘yavin.ece.msstate.edu’, do “*unix2dos myfile.hex*”
  - This converts the end-of-line format within the *hex* file from Unix-style to DOS style
- Once the *.hex* file is loaded, use ‘View → Program Memory’ to verify that memory contains valid instructions.

# Referring to Special Registers

```
#include <pic18.h>
```

Must have this include statement at top of a C file to include the processor header files for the PIC18 family. The

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

```
#static volatile near unsigned char PORTB @ 0xF81;
```

found in  
*/usr/local/hitech/include/pic18fxx2.h*

special  
register

memory  
location in  
PIC18

```
PORTB = 0x80;
```

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

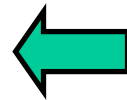
# *bittst*, *bitclr*, *bitset* Macros

```
#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 PICC18, these macros compile to the equivalent PIC18 *bsf*, *bcf*, *btfsc*, *btfss* instructions.

# Referring to Bits within Special Registers

The *pic18fxx2.h* include file also has definitions for individual bits within special function registers:

```
#static volatile near bit CARRY @((unsigned) &STATUS*8)+2;
```

*bit* data type

named bit

location that  
contains this  
bit

bit offset  
within register

```
CARRY = 1;
```

```
bitset(STATUS, 2);
```

Both do the same thing. The *bit* data type is not standard C – it is a non-standard extension of the language. But commonly done, so we will use it.

# Bit Testing within C

```
if (CARRY) {  
    // do if carry == 1  
}
```

```
if (!CARRY) {  
    // do if carry == 0  
}
```

```
if (bittst(STATUS,2) {  
    // do if carry == 1  
}
```

```
if (!bittst(STATUS,2) {  
    // do if carry == 1  
}
```

The above are all valid code fragments. Using the named bit symbols improves code clarity.

However, must still know that 'CARRY' refers to a bit and not a register!!!!

Is *PIR1* a bit or a special function register? How do you know? Look in the data sheet!!!!

# Runtime Code Produced by PICC18

The code produced by PICC18 C compiler first executes run-time start-up code before jumping to your *main()* routine.

The runtime code begins at the reset vector (location 0x0000), and it clears any uninitialized values to zero, or initializes variables to the value specified in the C code.

Initialized to '0' by reset code, which is the C default value for global variables.

```
char a;  
int k;
```

The initial values for these variables are stored in program memory. Reset code copies initial values from program memory to data memory.

```
int j = 10;  
char *astring = "hello";
```

*persistent* qualifier keeps reset code from touching this variable; initial value is undefined.

```
persistent int s;
```

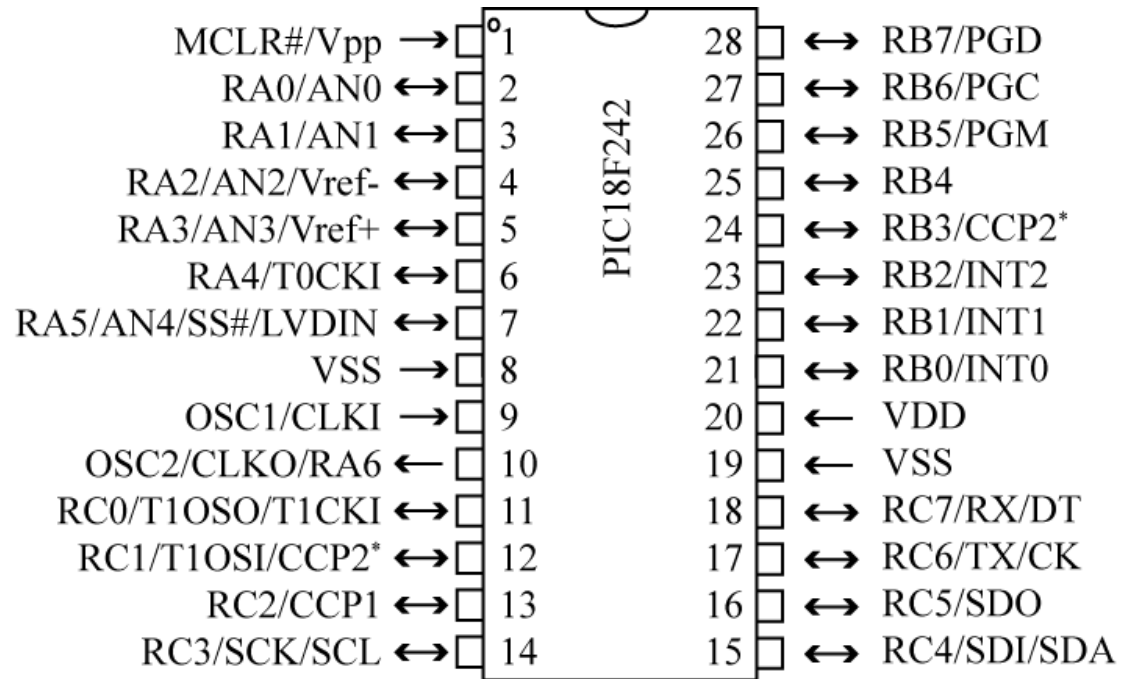
```
main() {  
    // your code  
}
```

# PIC18F242

Hardware lab exercises will use the PIC18F242 (28-pin DIP)

Note that most pins have multiple functions.

Pin functions are controlled via special registers in the PIC18F242.



\*RB3 is the alternate pin for the CCP2 pin multiplexing

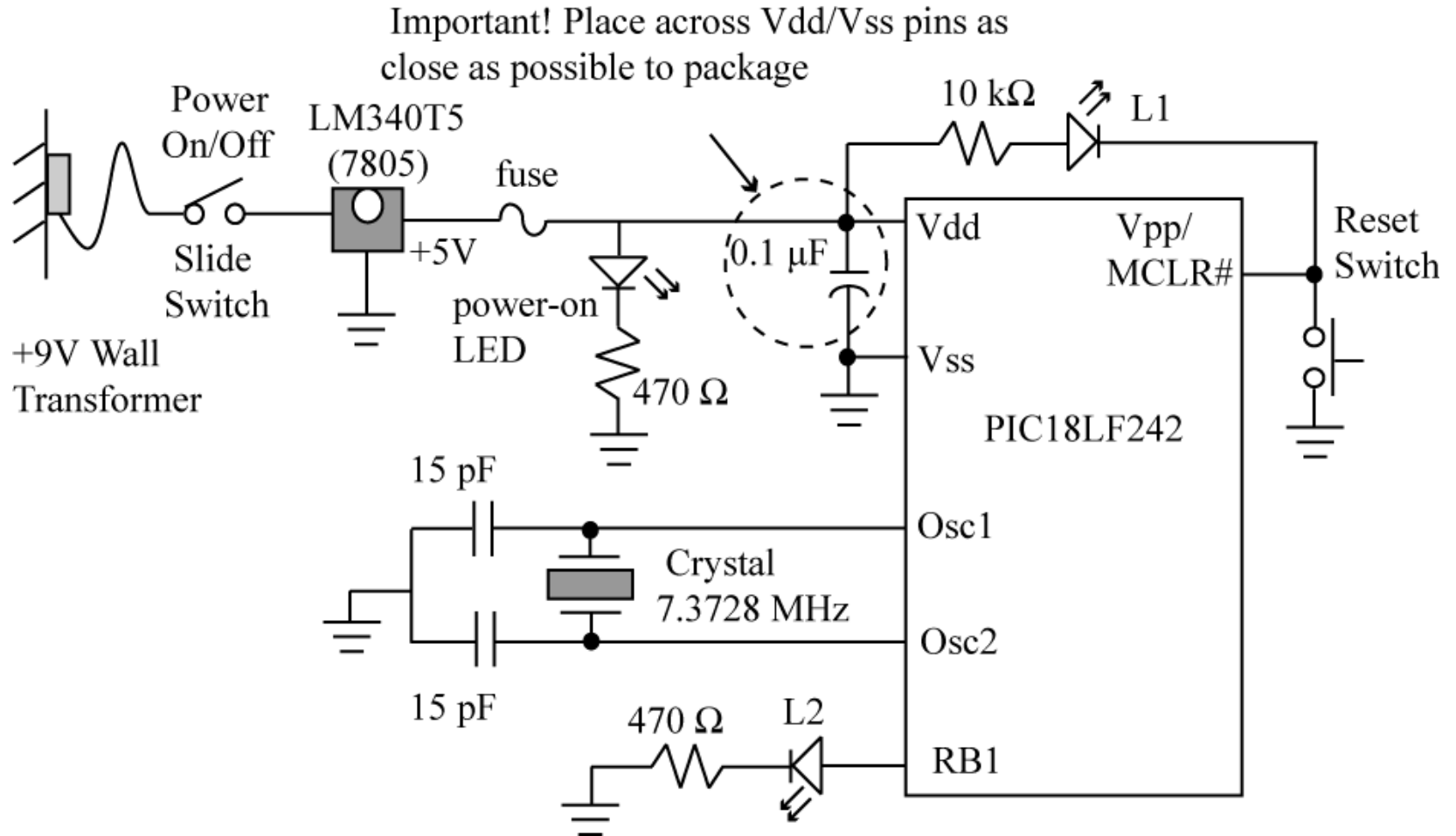
Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

Will download programs into the PIC18F242 via a serial bootloader that allows the PIC18F242 to program itself.

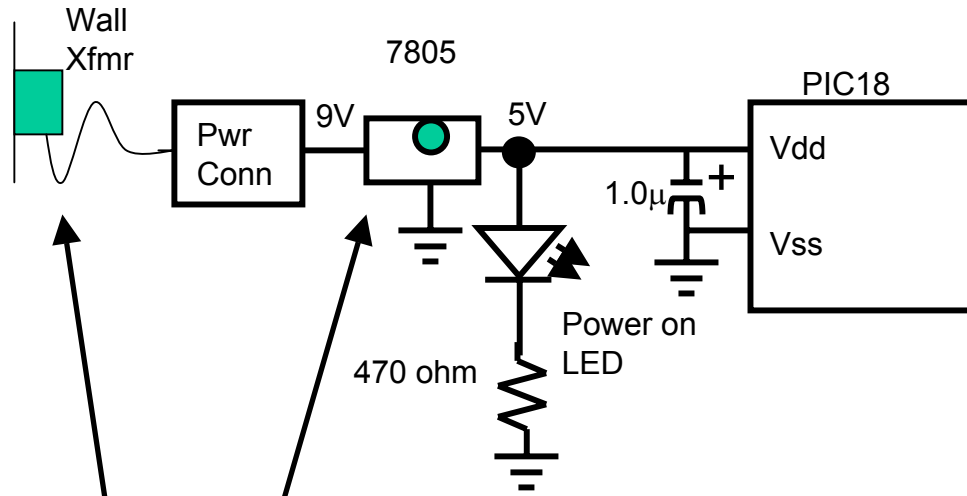
# Initial Hookup

If there are multiple VDD/VSS pins on your PIC18F242, hook them all up!!!

Note polarity of LED!!  
Should turn on when reset button is pressed.



# Powering the PIC18F242



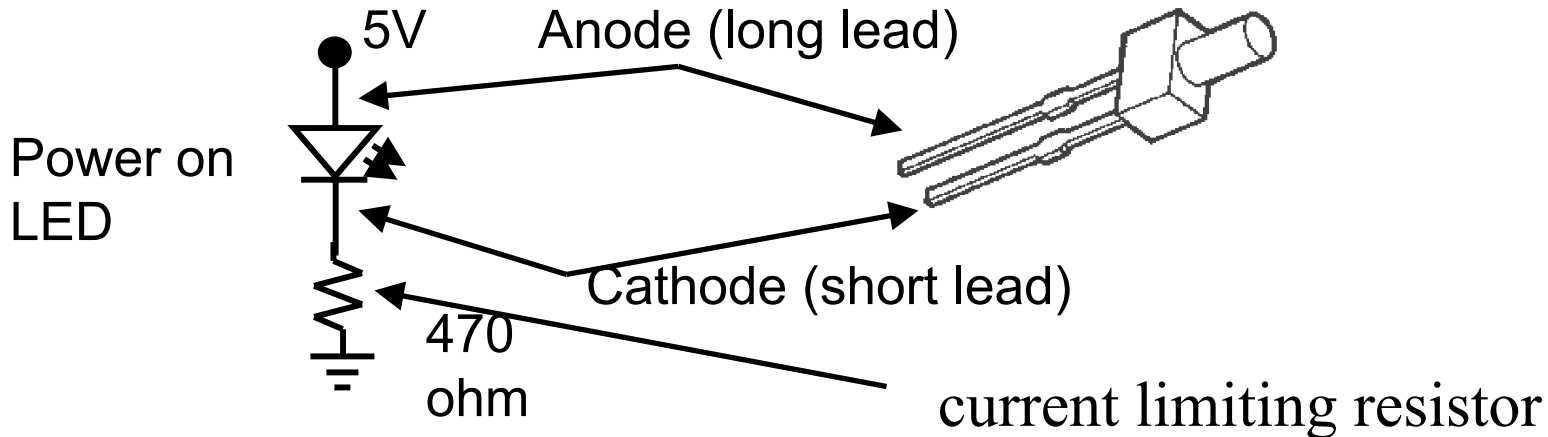
Wall transformer provides 9V DC unregulated (unregulated means that voltage can vary significantly depending on current being drawn). Maximum current from Xfmr is 650 mA.

The LM340T5 (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.

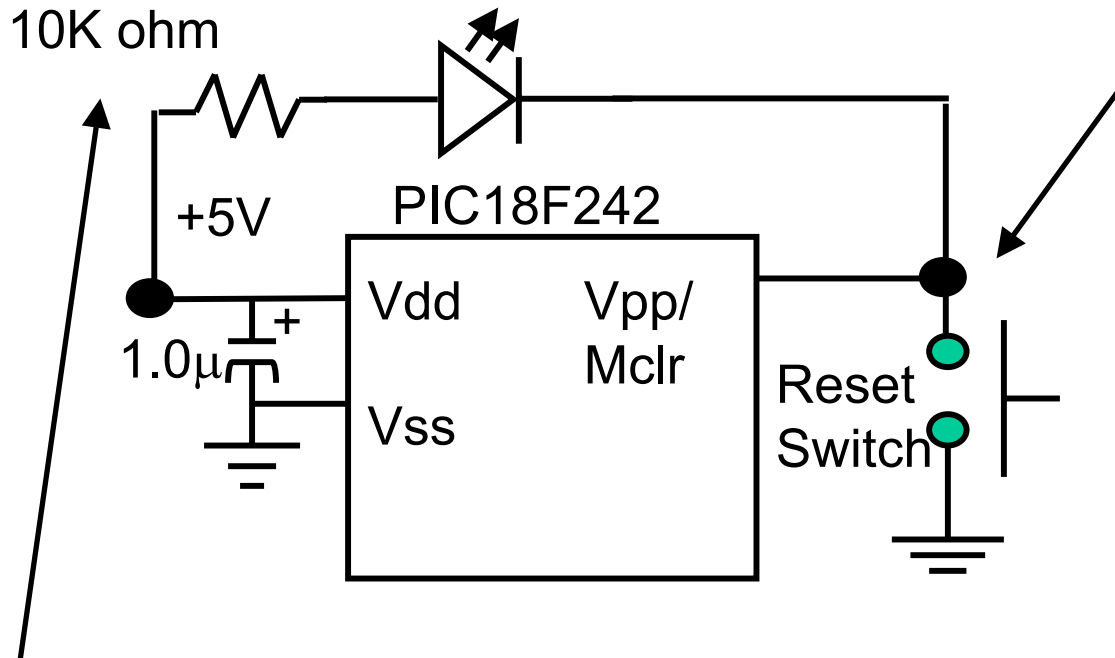
# Aside: How does an LED work?



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.

$$\text{Current} = \text{Voltage}/\text{Resistance} \sim (5\text{v} - 0.7\text{v})/470 \Omega = 9.1 \text{ mA}$$

# Reset

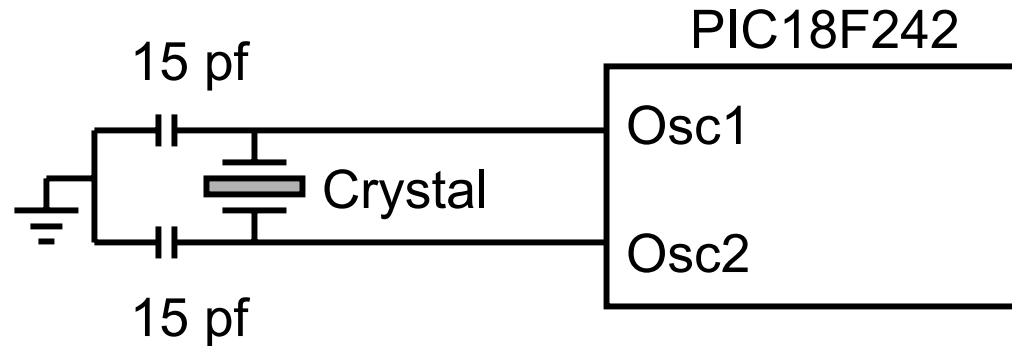


10K resistor used to limit current when reset button is pressed. Diode will be very dim when reset switch is pressed because current  $\sim 0.5$  mA

When reset button is pressed, the Vpp/Mclr pin is brought to ground. This causes the PIC18 program counter to be reset to 0, so next instruction fetched will be from location 0. All  $\mu$ Ps have a reset line in order to force the  $\mu$ P to a known state.

# The Clock

7.3728 MHz



Will use an external crystal and 2 capacitors to provide the clock for the PIC18F242. A circuit internal to the PIC18 causes the crystal to begin oscillating after power up. This ‘weird’ frequency provides common baud rates for serial communication when divided down internally.

Internally, we will use the HSPLL option (High Speed, Phased Locked Loop to multiply this clock frequency by 4)

$$7.3728 \text{ MHz} * 4 = 29.4912 \text{ MHz (actual internal clock freq.)}$$

The PIC18F242 can also use an external RC network (cheap, but not very accurate) or an external oscillator (less components, but expensive).

# Configuration Bits

**Configuration bits** are stored beginning at location 0x300000 in program memory to control various processor options. Configuration bits are only read at power up.

Processor options controlled by configuration bits relate to Oscillator options, Watchdog timer operation, RESET operation, Interrupts, Code protection, etc.

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

# Specifying Configuration Options in C

The file *config.h* included by the sample programs used in lab contains the following statements that specifies configuration bits used for all lab exercises:

```
___CONFIG(1, HSPLL) ;  
___CONFIG(2, BORDIS & PWRTDIS & WDTDIS) ;  
___CONFIG(4, DEBUGDIS & LVPDIS) ;
```

HSPLL: use external crystal with the internal PLL

BORDIS: disables brownout reset (disables auto reset if voltage drops too low)

PWRTDIS: disables the power up timer, when power applied, begin execution immediately.

WDTDIS: disables hardware enable of the watchdog timer (allows the watchdog timer to be turned on/off in software)

LVPDIS: disables low voltage programming, pin RB5 can be used as I/O pin.

# Programming the PIC18F242 Flash Memory

The TA will program your PIC18F242 with a program called “ledflash” that will blink the LED attached to port RB1.

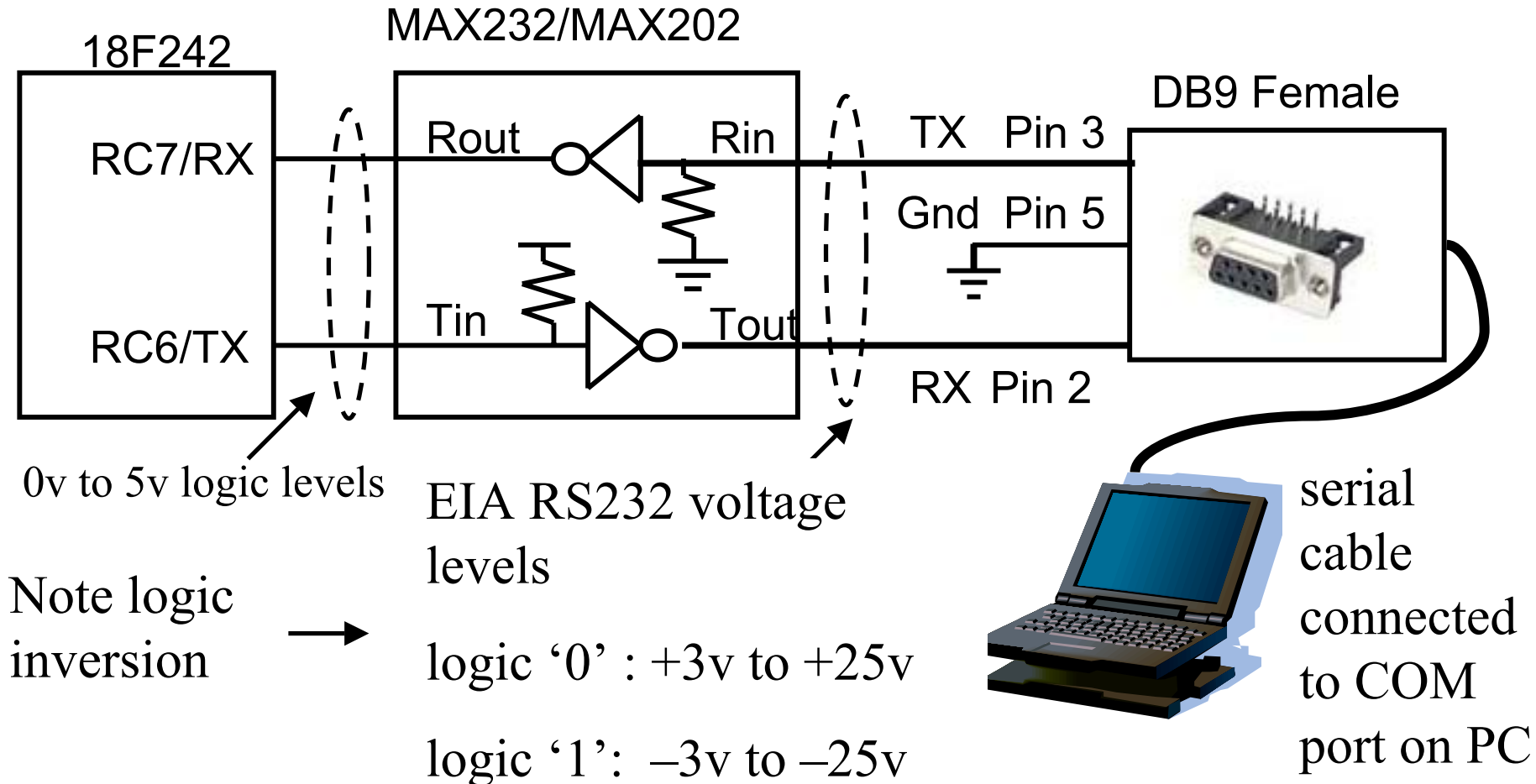
Do this to verify that your PIC18F242 is working. The TA will use an external programmer to program your PIC18F242, which must be removed from your board to program.

Then connect the serial port interface shown on the next page so that the PIC18F242 can program itself without removing it from the board.

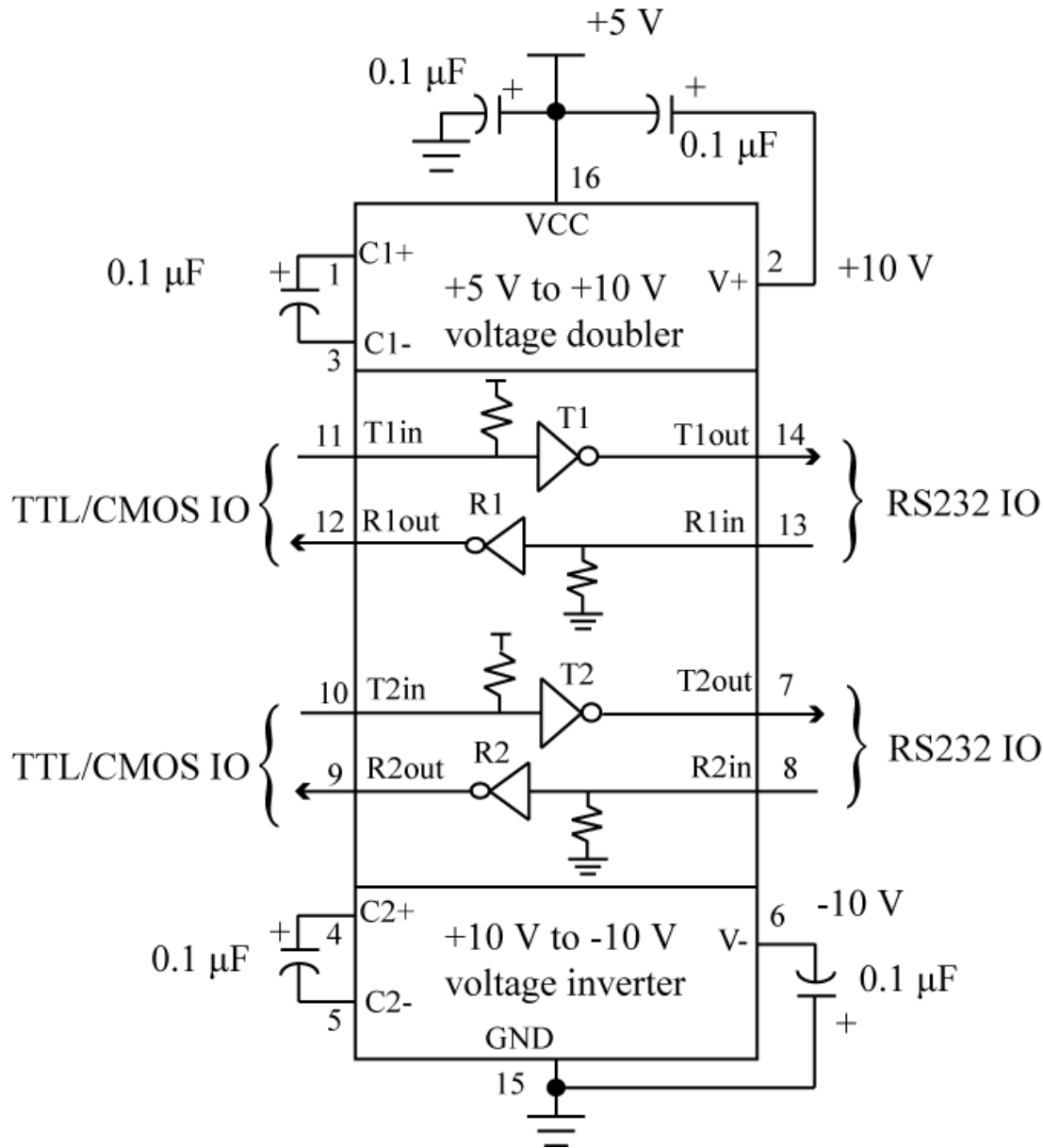
# A Serial Bootloader

The PIC18F242 can program itself by without removal from the board by downloading a program via a serial port connection to an external PC.

The serial port connection is shown below, will discuss operation in detail later, just wire it up.



# MAXIM 232/202 driver/receiver



Converts RS232 voltage levels to digital levels and vice-versa

External capacitors used with internal charge pump circuit to produce +/- 10V from 5V supply

# Hyperterminal

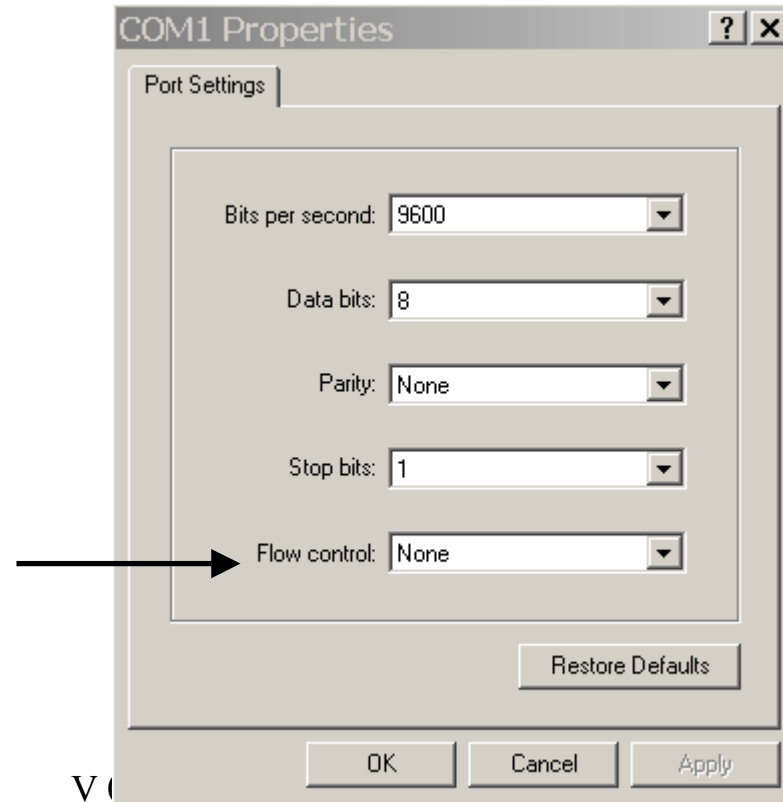
Will use Hyperterminal program on PC to communicate with PIC18F242.

Under Programs→Accessories →Communications → Hyperterminal

When configuring Hyperterminal connection, must know port number (COM1/COM2/etc), baud rate, data bits (8), parity (none), stop bits (1), and flow control(none)

On PC lab machines, use COM1

Very important to set flow control to **none** since we are only using a 3-wire connection and not using the handshaking lines in the RS232 standard. If you forget this, then will not receive any characters.

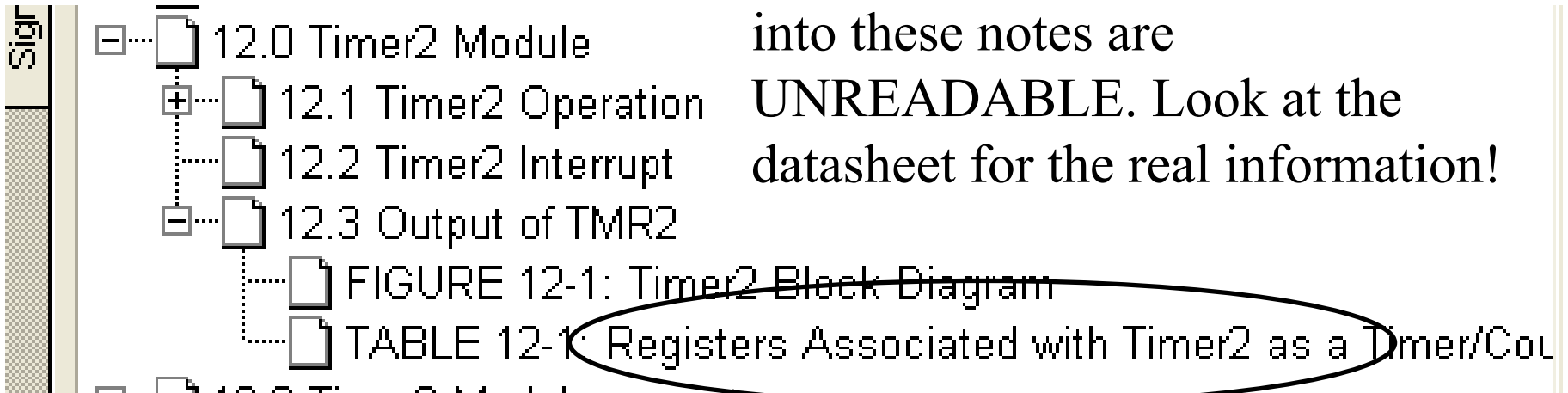


# Reading the PIC18Fxx2 Datasheet

- You **MUST** be able to read the PIC18Fxx2 datasheet and find information in it.
  - These notes refer to bits and pieces of what you need to know, but **DO NOT** duplicate everything that is contained in the datasheet.
- The datasheet chapters are broken up into functionality (I/O Ports, Timer0, USART)
  - In each chapters are sections on different capabilities (I/O ports have a section on each PORT).
- At the end of each chapter is a summary of all registers and bits affecting the operation of that component.
  - This summary is **VERY HELPFUL**. It should one of the first places you look.
- Reading the datasheet is required if you expect to pass the tests in this course.

# PIC18 Datasheet: Example Register Summary

Pieces of the datasheet cut/pasted into these notes are UNREADABLE. Look at the datasheet for the real information!



**TABLE 12-1: REGISTERS ASSOCIATED WITH TIMER2 AS A TIMER/COUNTER**

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on All Other RESETS
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	0000 000u
PIR1	PSPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
PIE1	PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
IPR1	PSPIP <sup>(1)</sup>	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	0000 0000	0000 0000
TMR2	Timer2 Module Register								0000 0000	0000 0000
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
PR2	Timer2 Period Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented read as '0'. Shaded cells are not used by the Timer2 module.

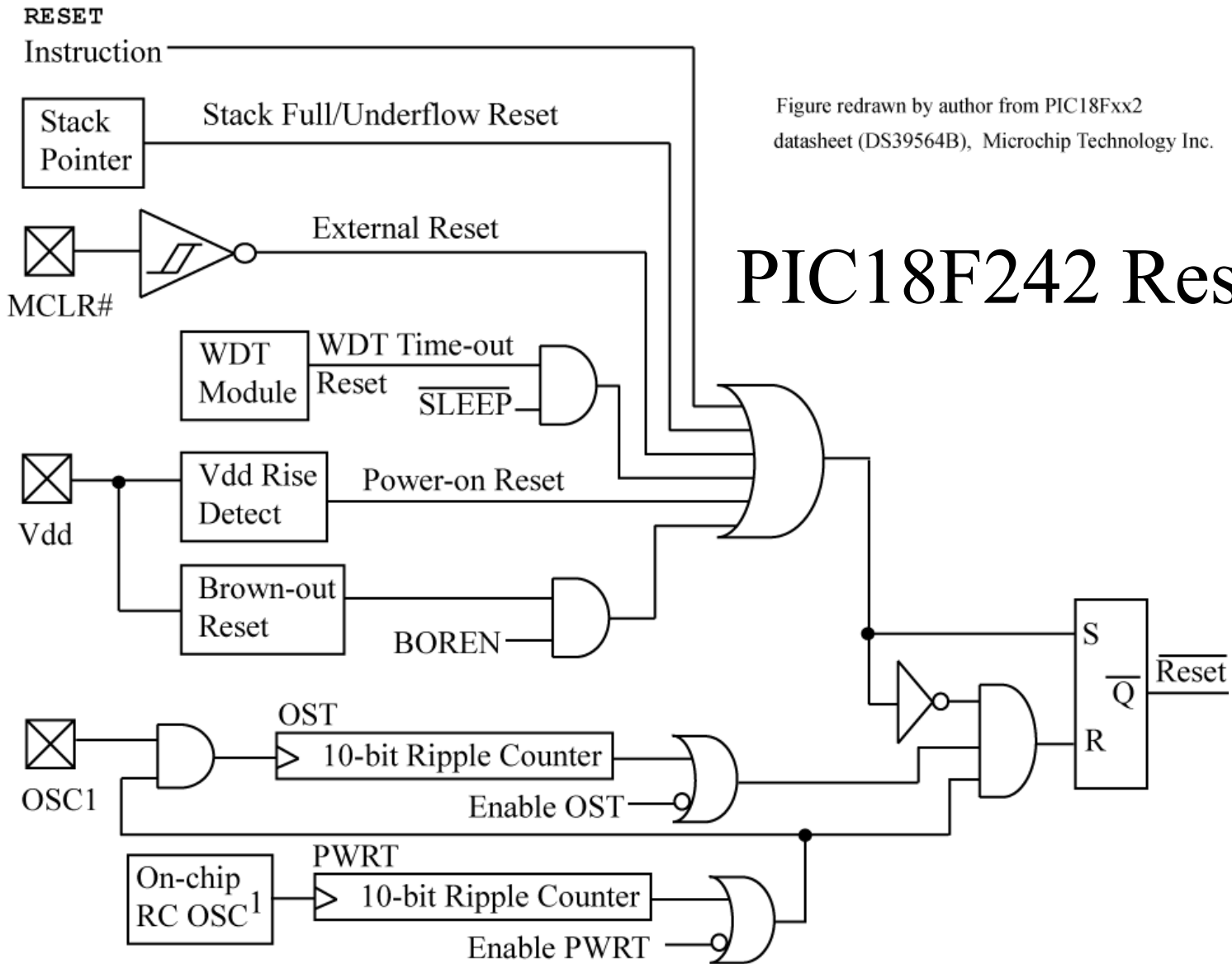


Figure redrawn by author from PIC18Fxx2  
datasheet (DS39564B), Microchip Technology Inc.

# PIC18F242 Reset

**Note 1:** This is a separate oscillator from the RC oscillator of the CLKI pin.

# PIC18Fxx2 Reset Sources

- RESET instruction (software reset)
- MCLR reset (external pin, pushbutton)
- Stack Underflow/Overflow
- Watchdog timer (WDT)
  - A timer is a counter; when WDT wraps around, generates a reset.
- Power-On Reset (POR) – reset automatically applied when power applied (do not have to use pushbutton).
- Brownout Reset (BOR) – if VDD falls below certain value, auto reset
- Power-up Timer (PWRT) – after power up detected, wait an additional time period for external power to stabilize (optional).
- Oscillator Startup (OST) – after power up timer is expired, wait an additional time period for external crystal oscillator to stabilize

# What RESET type occurred?

RCON Register	IPEN	–	–	RI#	TO#	PD#	POR#	BOR#
	bit 7							bit 0

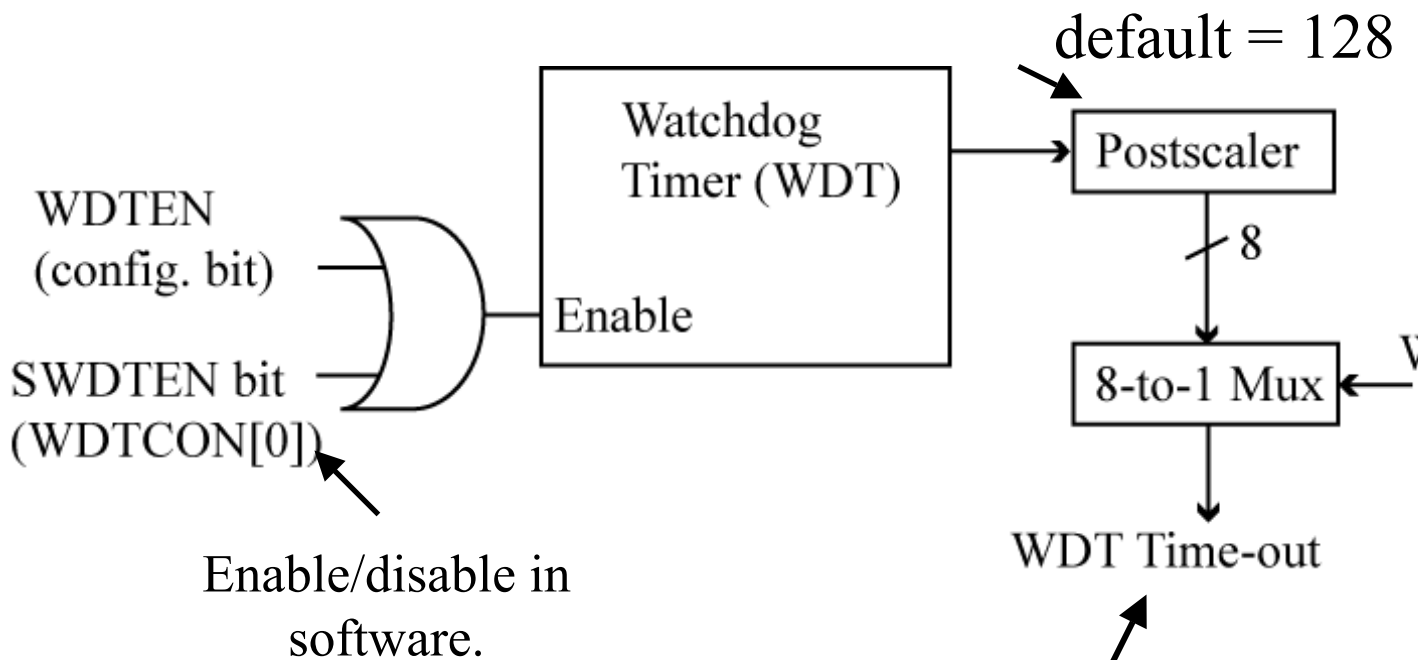
- bit 7 IPEN : Interrupt Priority Enable  
1 - Interrupt priorities enabled  
0 - Interrupt priorities disabled
- bit 4 RI# : **RESET** Instruction Flag Bit  
1 - **RESET** instruction not executed  
0 - **RESET** instruction executed causing device reset  
(must be set in software after **RESET** instruction occurs)
- bit 3 TO# : Watchdog Time-out Flag Bit  
1 - after power-up, **CLWRDT** instruction, or **SLEEP** instruction  
0 - a WDT time-out occurred
- bit 2 PD# : Power-down Detection Flag bit  
1 - After power-up or by the **CLRWDT** instruction  
0 - By execution of the **SLEEP** instruction
- bit 1 POR#: Power-on Reset (POR) Status Bit  
1 - A POR has not occurred  
0 - A POR occurred (must be set in software after a POR occurs)

# What RESET type occurred? (cont)

Condition	Program Counter	RCON Register	RI#	TO#	PD#	POR#	BOR#	STKFUL	STKUNF
Power-on Reset	0x0000	0--1 1100	1	1	1	0	0	u	u
MCLR# Reset during normal operation	0x0000	0--u uuuu	u	u	u	u	u	u	u
Software Reset during normal operation	0x0000	0--0 uuuu	0	u	u	u	u	u	u
Stack Full Reset during normal operation	0x0000	0--u uu11	u	u	u	u	u	u	1
Stack Underflow Reset during normal operation	0x0000	0--u uu11	u	u	u	u	u	1	u
MCLR# Reset during SLEEP	0x0000	0--u 10uu	u	1	0	u	u	u	u
WDT Reset	0x0000	0--u 01uu	1	0	1	u	u	u	u
WDT Wake-up	PC+2	u--u 00uu	u	0	0	u	u	u	u
Brown-out Reset	0x0000	0--1 11u0	1	1	1	1	0	u	u
Interrupt wake-up from SLEEP	PC+2 <sup>(1)</sup>	u--u 00uu	u	1	0	u	u	u	u

Legend: **u** : unchanged, **x** = unknown, **-** = unimplemented bit, read as '0'

# Watchdog Timer



WDTEN  
(config. bit)

SWDTEN bit  
(WDTCON[0])

Enable/disable in  
software.

On-chip, free-running RC oscillator independent of main clock used to clock WDT. Typical time out value is 18 ms.

If WDT enabled, code must reset the WDT before it times out to avoid reset.

Useful if code gets hung up talking to a failed external device – watch dog reset will force PIC18Fxx2 to restart and code can detect that this happened, and take action.

# WDT Specifics

Using free-running RC oscillator, a typical WDT timeout value is 18 ms (no scaling). Default postscaler is 128 (multiplies timeout value), so default timeout is about 2.3 seconds.

WDT free-running RC oscillator runs even if normal oscillator clock is stopped!!

This means that the WDT can be used to **wake-up** out of **sleep mode**.

The PIC18 instruction **CLRWDT** is used to clear the WDT and prevent a time-out.

If code is in a loop waiting for a response from external device that has an arbitrarily long wait (like a human!), the loop should include 'CLRWDT' instruction so that the watchdog timer reset does not occur.

# Power Consumption

Power is measured in watts,  $P = V_{dd} * I_{dd}$ , where  $I_{dd}$  is the power supply current.

Datasheet gives values of  $I_{dd}$  for different values of  $V_{dd}$ .

PIC18LF242 is low power version of PIC18F242 – can operate over wider range of  $V_{DD}$  values, has lower power-down current. The PIC18LF242 is the PIC18 version in the lab parts kit.

Total Power = Static power + Dynamic Power

Static Power – power consumed when clock is stopped (power down, i.e., SLEEP mode).

Dynamic Power – power consumed when PIC18Fxx2 is operating

# Static Power/SLEEP Mode

Power-down Current <sup>(3)</sup>					
PIC18LFXX2	—	.08	.9	μA	VDD = 2.0V, +25°C
	—	.1	4	μA	VDD = 2.0V, -40°C to +85°C
	—	3	10	μA	VDD = 4.2V, -40°C to +85°C
PIC18FXX2	—	.1	.9	μA	VDD = 4.2V, +25°C
	—	3	10	μA	VDD = 4.2V, -40°C to +85°C
	—	15	25	μA	VDD = 4.2V, -40°C to +125°C

Copyright Microchip, from PIC18xx2 Datasheet DS39564B

Typical | Max

The PIC18 instruction SLEEP is used to enter the power down mode. Power requirements in SLEEP mode can be reduced by 100x over normal operation!

If the WDT is enabled, it is cleared when SLEEP is executed. A WDT timeout will then wake up the processor, and the processor will continue normal operation (continues at next instruction).

# Dynamic Power

$$\text{Dynamic Power} = V_{dd} * V_{dd} * F_{osc} * C$$

where

$V_{dd}$  : power supply voltage

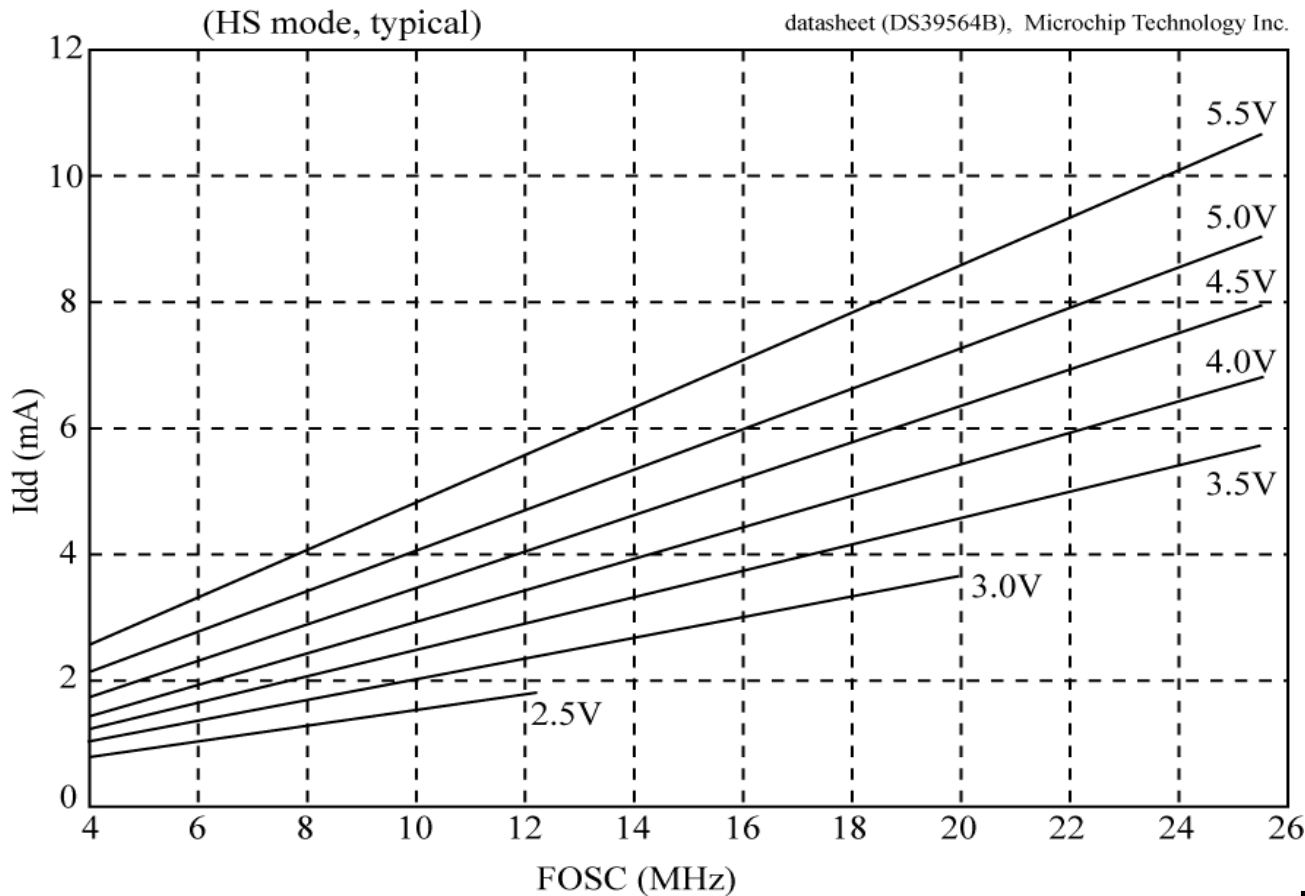
$F_{osc}$  : oscillator frequency,

$C$  : chip capacitance that switches each clock cycle.

The  $C$  is fixed, cannot change, dependent upon number of transistors in operation during a clock cycle.

Both  $V_{dd}$  and  $F_{osc}$  can be adjusted during use.

Reducing  $V_{dd}$  has the largest impact on reducing power requirement, power is proportional to the square of the voltage!!!



$V_{dd} = 5.0 \text{ V}$ ,  $F_{osc} = 20 \text{ MHz}$ ,  $I_{dd} \approx 7.9 \text{ mA}$  ←

$V_{dd} = 5.0 \text{ V}$ ,  $F_{osc} = 10 \text{ MHz}$ ,  $I_{dd} \approx 4.5 \text{ mA}$

~1.75 Ratio,  
expected 2.0

$V_{dd} = 5.5 \text{ V}$ ,  $F_{osc} = 12 \text{ MHz}$ ,  $I_{dd} \approx 6.0 \text{ mA}$  ←

$V_{dd} = 2.5 \text{ V}$ ,  $F_{osc} = 12 \text{ MHz}$ ,  $I_{dd} \approx 2.0 \text{ mA}$

~3.0 Ratio,  
expected 4.8

# Frequency versus Voltage

Why does the previous graph for  $V_{dd} = 2.5$  V stop at 12 MHz? Or for  $V_{dd} = 3.0$  V, stop at 20 MHz?

For lower supply voltages, transistors do not switch as fast, so maximum operating frequency of 40 MHz cannot be achieved!

**Tradeoff:** lower voltage, lower power consumption, but reduces maximum achievable clock frequency.

# *reset.c* Program

Allows experimentation with sleep mode, watch dog timer.

Want to detect if Power-ON Reset (POR), MCLR reset, Watchdog Timer Reset occurs.

Increment a variable each time a non-Power-ON reset occurs, clear this variable to 0 if a power-on reset occurs.

Allow the user to enable the watchdog timer.

Allow the user to enter sleep mode.

Allow the user to both enable watchdog timer and enter sleep mode.

# *reset.c* Program Listing

```
persistent char reset cnt;
```

by default, any global variable cleared to '0' by reset code. 'persistent' prevents this from happening.

```
main(void) {
```

```
int i;
```

```
char c;
```

Init serial port to 19,200 baud, will discuss later.

```
serial_init(95,1);
```

```
pcrlf();
```

Check for power-on reset.

```
if (POR == 0) {
```

```
// POR is RCON(2), cleared to 0 on power-up reset
```

```
printf("Power-on reset has occurred.");
```

```
pcrlf();
```

```
// setting POR bit =1, will remain a '1' for MCLR reset
```

```
POR = 1;
```

```
reset_cnt = 0;
```

Clear reset count on POR.

```
}
```

```
if (TO == 0) { //TO is RCON(4)
```

Check for WDT reset

```
SWDTEN = 0; // disable watchdog timer
```

```
printf("Watchdog timer reset has occurred.\n");
```

```
pcrlf();
```

SWDTEN is WDCON(0) register, '0' disables.

```
}
```

# *reset.c* Program Listing (cont).

```
i = reset_cnt;
printf("Reset cnt is: %d",i); pcrLf();
reset_cnt++;
while(1) {
    printf("'1' to enable watchdog timer"); pcrLf();
    printf("'2' for sleep mode"); pcrLf();
    printf("'3 ' for both watchdog timer and sleep mode");
    pcrLf();
    printf("Anything else does nothing, enter keypress: ");
    c = getch();
    putchar(c); pcrLf();
    if (c == '1') SWDTEN = 1; // enable watchdog timer
    else if (c == '2') {
        asm("sleep");
    }
    else if (c == '3') {
        SWDTEN = 1; // enable watchdog timer
        asm("sleep");
    }
}
```

bit 0 in WDTCON register, '1' enables.

'asm' allows insertion of PIC18 instructions into C code – called 'in-line' assembly.

Typing '3' on keyboard chooses to both enable WDT and enter sleep mode.

# Sleep Mode and WDT

Sleep mode causes the main clock to stop, dramatically reduces power consumption.

The PC is frozen at the next instruction after sleep mode.

Because the WDT clock is separate from the main clock, the WDT keeps running.

When the WDT goes off, the processor ‘wakes up’ by restarting the clock, and the PC picks up where it left off, which is the instruction after the SLEEP instruction.

# More on *reset.c*

*reset.c* has subroutines called *getch()*, *putch()*, and *serial\_init()* (*serial\_init* is in the file ‘*serial.c*’). The functionality of these subroutines is discussed in detail later.

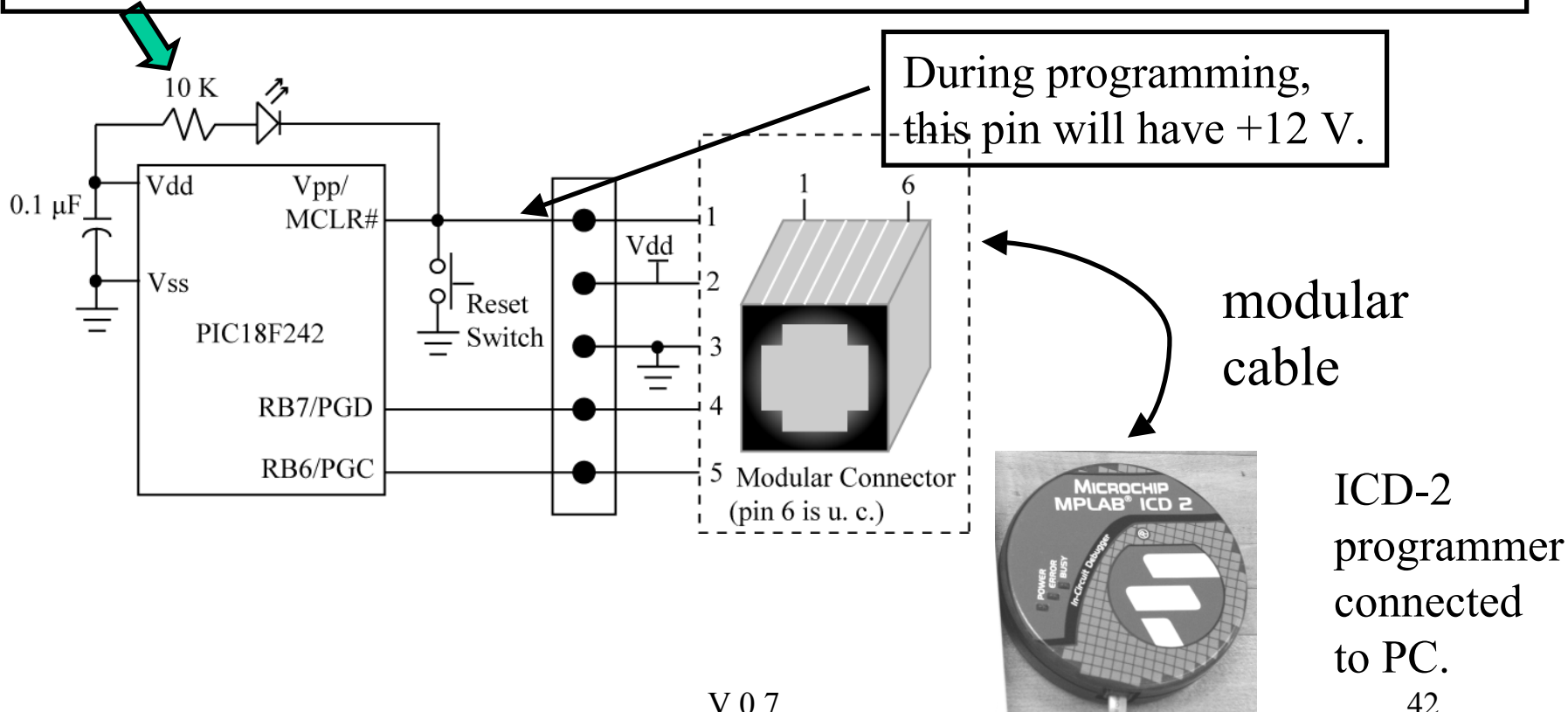
The *getch()* subroutine waits for a byte to be ready from the serial port, then returns it.

The *putch()* subroutine writes one byte to the serial port (this is used by the ‘*printf()*’ library call).

The *serial\_init()* subroutine initializes the serial port and sets the baud rate.

# Optional In-Circuit Programming

Connect this only if you want to use the 'hockey-puck' programmers. The 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.



# What do you have to know?

- Understand initial hookup schematic for the PIC18F242
- How pullup resistors work and when they are needed.
- What configuration bits are used for
- Watchdog timer operation
- Sleep mode operation
- Power consumption equation, static vs dynamic power
- *reset.c* operation