COE 115

Lecture 4

C and Embedded Systems

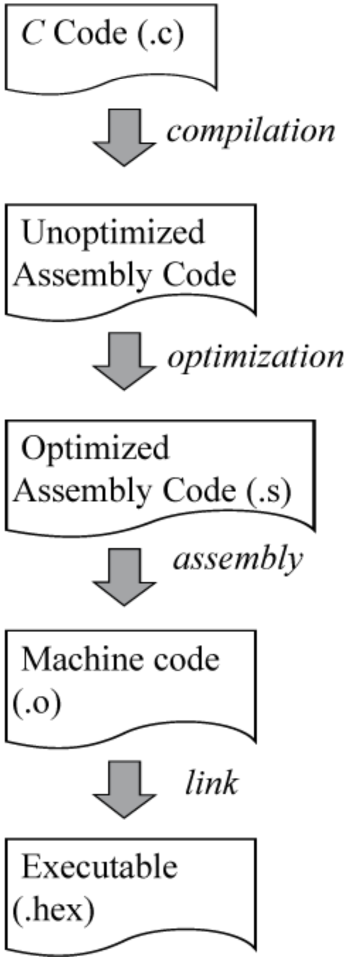
- A μC-based system used in a device (i.e., a car engine) performing control and monitoring functions is referred to as an **embedded system**.
  - The embedded system is invisible to the user
  - The user only indirectly interacts with the embedded system by using the device that contains the μC
- Many programs for embedded systems are written in C
  - Portable – code can be retargeted to different processors
  - Clarity – C is easier to understand than assembly
  - Modern 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 **uint32** data type is used where **uint8** would suffice, both performance and code size will suffer.
- Learning the assembly language, architecture of the target μC provides performance and code size clues for compiled C
  - Does the μC have support for multiply/divide?
  - Can the μC shift only one position each shift or multiple positions? (i.e, does it have a *barrel shifter*?)
  - How much internal RAM does the μC have?
  - Does the μC have floating point support?
- Sometimes have to write assembly code for performance reasons.

C Compilation

From .c to .hex



Example Optimization

```
i = i + j;
k = k + j;
```

compilation

```
mov    j,W0   ;W0 = j
add    i      ;i = i + W0 = i + j
mov    j,W0   ;W0 = j
add    k      ;k = k + W0 = k + j
```

optimization

```
mov    j,W0   ;W0 = j
add    i      ;i = i + W0 = i + j
add    k      ;k = k + W0 = k + j
```

W0 already contains `j`, remove second `mov` instruction

MPLAB PIC24 *C* Compiler

- Programs for hardware experiments are written in *C*
- Will use the MPLAB PIC24 *C* Compiler from Microchip
- **Excellent** compiler, based on GNU C, generates very good code
- Use the MPLAB example projects that come with the ZIP archive associated with the first hardware lab as a start for your projects

Referring to Special Function Registers

```
#include "pic24.h"
```

Must have this include statement at top of a *C* file to include the all of the header files for the support libraries.

Special Function Registers can be accessed like variables:

```
extern volatile unsigned int  PORTB __attribute__((__sfr__));
```

Defined in compiler header files

Register Name

Special function register

```
PORTB = 0xF000;
```

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

Referring to Bits within Special Function Registers

The compiler include file also has definitions for individual bits within special function registers. Can use these to access individual bits and bit fields:

```
PORTBbits.RB5 = 1;   //set bit 5 of PORTB
PORTBbits.RB2 = 0;   //clear bit 2 of PORTB

 if (PORTBbits.RB0) {
  //execute if-body if LSb of PORTB is '1'
....
}
```

A bit field in a SFR is a grouping of consecutive bits; can also be assigned a value.

```
OSCCONbits.NOSC = 2;    //bit field in OSSCON register
```

Referring to Bits within Special Function Registers

Using *registername.bitname* requires you to remember both the register name and the bitname. For bitnames that are UNIQUE, can use just *_bitname*.

```
_RB5 = 1;   //set bit 5 of PORTB
_RB2 = 0;   //clear bit 2 of PORTB

 if (_RB0) {
  //execute if-body if LSb of PORTB is '1'
....
}

_NOSC = 2;    //bit field in OSSCON register
```

Variable Qualifiers, Initialization

If a global variable does not have an initial value, by default the runtime code initializes it to zero – this includes static arrays. To prevent a variable from being initialized to zero, use the _PERSISTENT macro in front of it:

```
uint16  u16_k;      //initialized to 0
uint8   u8_k = 4;   //initialized to 4

_PERSISTENT uint8 u8_resetCount;  //uninitialized, value
                                  // is unknown
```

The C runtime code is run before **`main()`** entry, so run on every power-up, every reset. Use **`_PERSISTENT`** variables to track values across processor resets.

C Macros, Inline Functions

The support library and code examples makes extensive use of C macros and Inline functions. The naming convention is all uppercase:

```
#define DEFAULT_BAUDRATE    57600
#define LED1     _RB15
```

Macros, the left hand label is replaced by the right hand text

```
static inline void CONFIG_RB1_AS_DIG_INPUT(){
     DISABLE_RB1_PULLUP();
    _TRISB1 = 1;
    _PCFG3 = 1;
}
```

Inline functions expand without a subroutine call.

PIC24HJ32GP202 µC

Hardware lab exercises will use the PIC24HJ32GP202 µC (28-pin DIP)

Note that most pins have multiple functions.

Pin functions are controlled via special registers in the PIC.

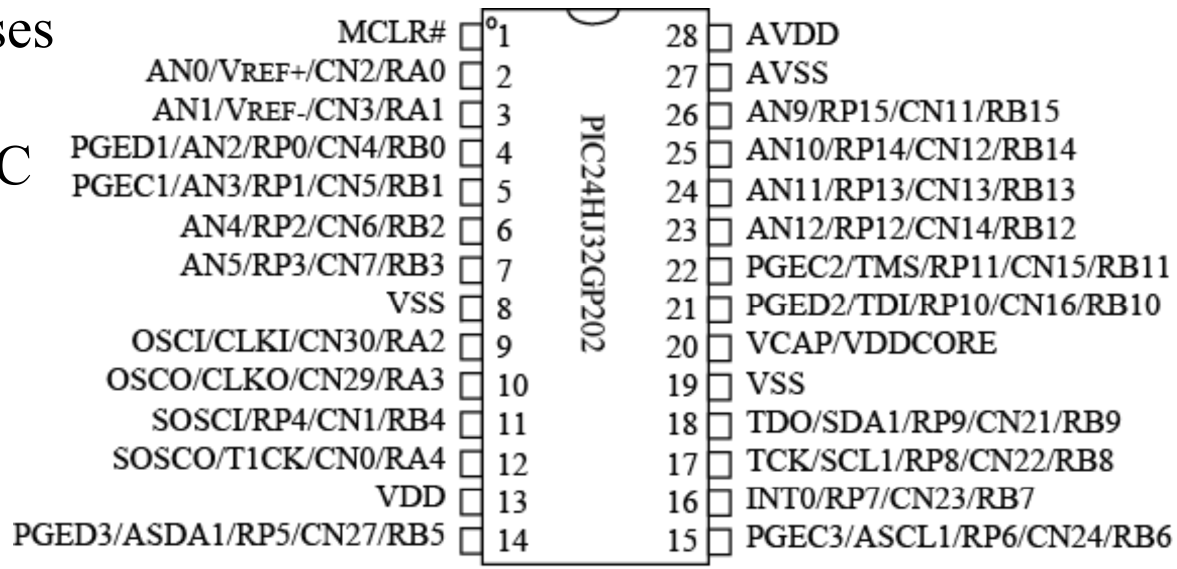
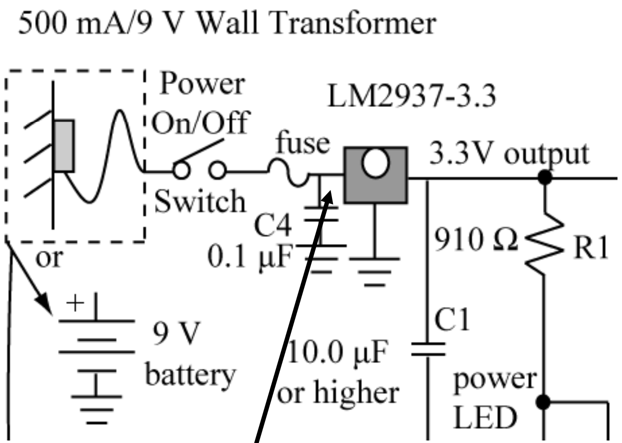


Figure redrawn by author from PIC24HJ32GP202/204 datasheet (DS70289A), Microchip Technology Inc.

Will download programs into the PIC24 µC via a serial bootloader that allows the PIC24 µC to program itself.

Powering the PIC24 μC



The POWER LED provides a visual indication that power is on.

A Wall transformer provides 15 to 6V DC unregulated (unregulated means that voltage can vary significantly depending on current being drawn). The particular wall Xfmr in the parts kit provides 6V with a max current of 1000 mA.

The LM2937-3.3 voltage regulator provides a regulated +3.3V. Voltage will stay stable up to maximum current rating of device.

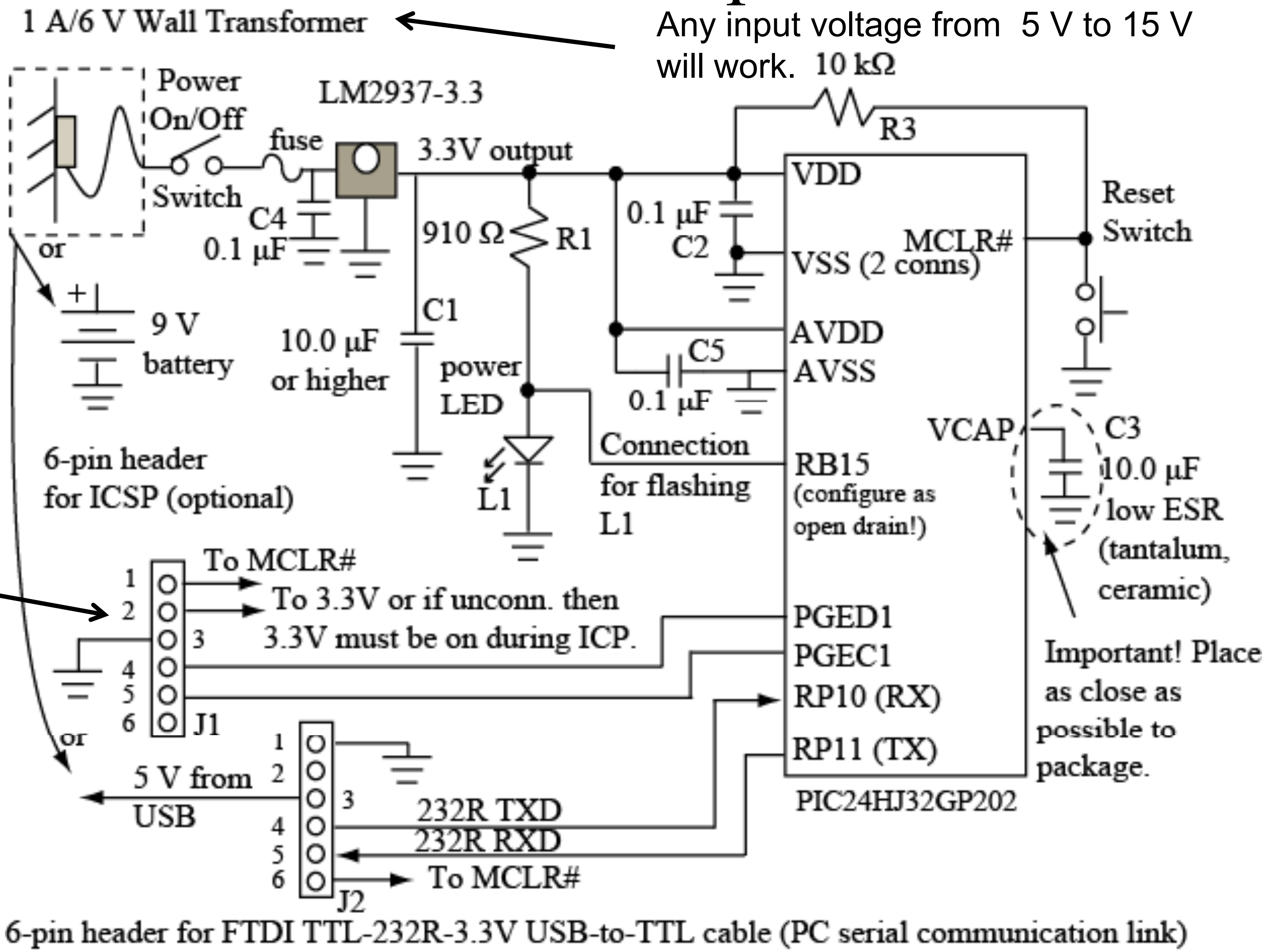
With writing on device visible, input pin (+9 v) is left side, middle is ground, right pin is +3.3V regulated output voltage.

Initial Hookup

There are multiple VDD/VSS pins on your PIC24 µC; hook them all up!!!

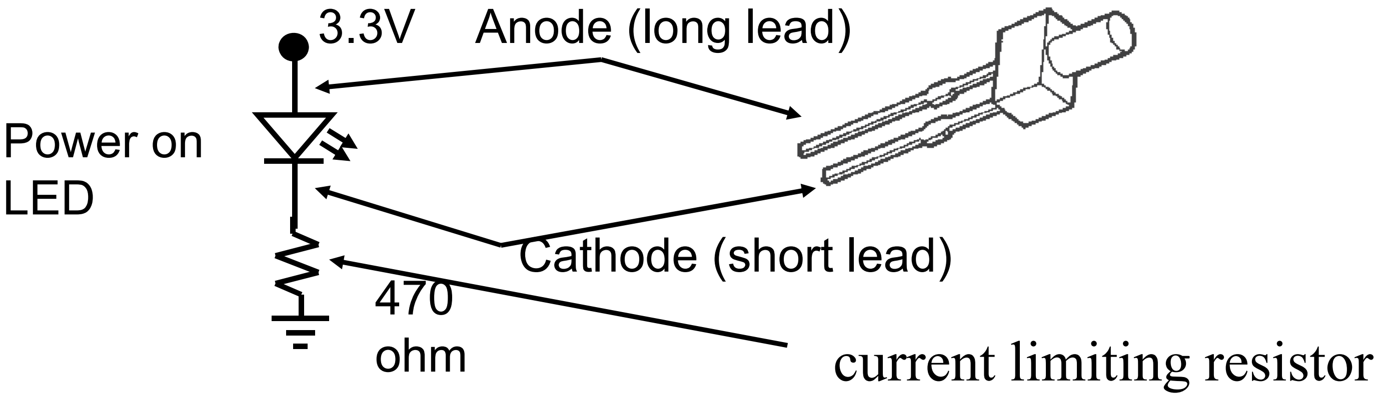
Any input voltage from 5 V to 15 V will work.

Not included in your board.



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

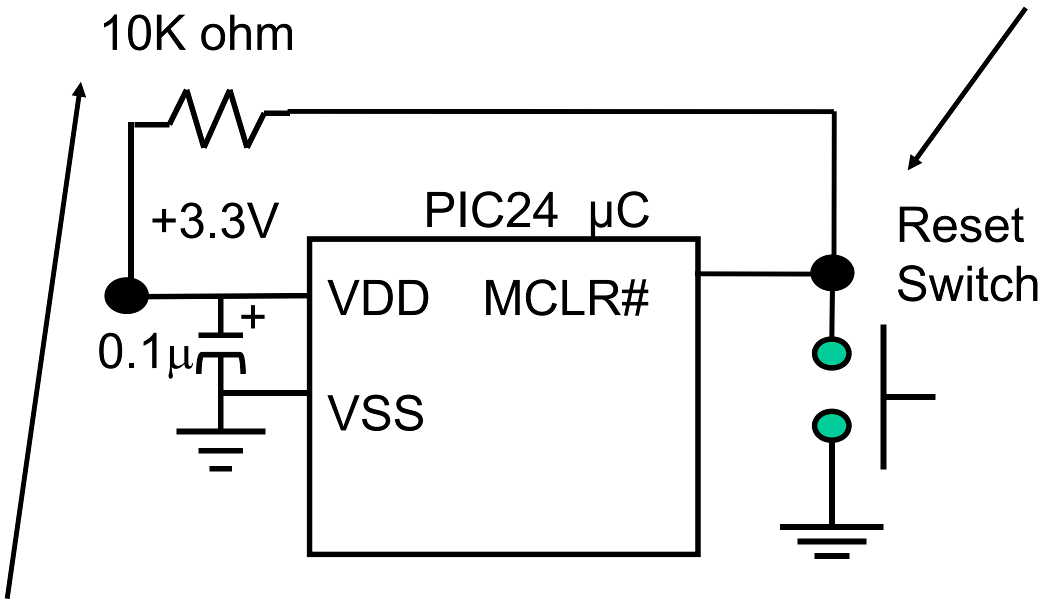
Aside: How does an LED work?



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. The voltage drop across LEDs used in the lab is about 2V.

Current = Voltage/Resistance ~ (3.3v – LED voltage drop)/470 Ω
= (3.3v – 2.2V)/470 = 2.7 mA

Reset



10K resistor used to limit current when reset button is pressed.

When reset button is pressed, the 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 µCs have a reset line in order to force the µC to a known state.

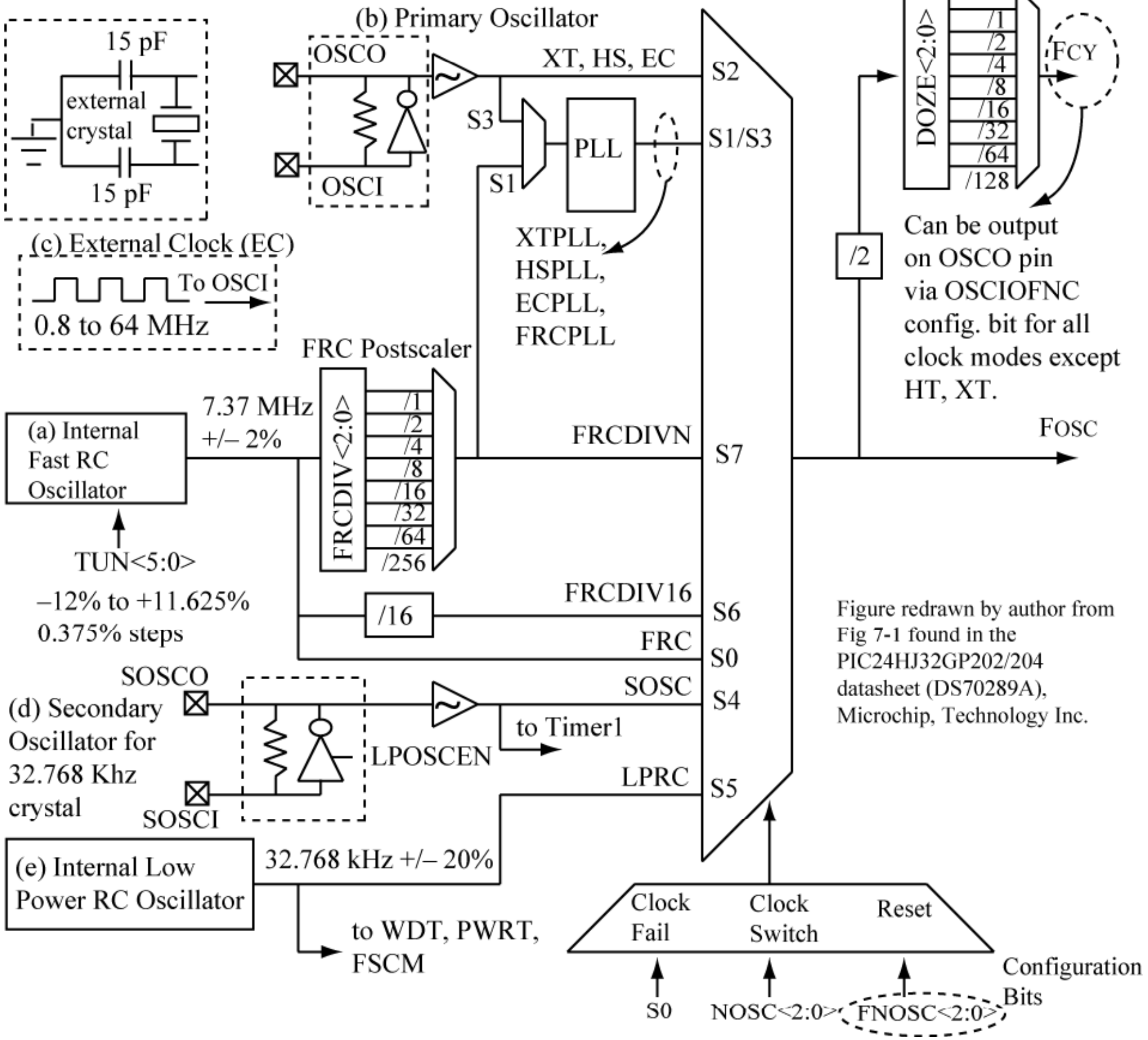


Figure redrawn by author from Fig 7-1 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip, Technology Inc.

The Clock

The PIC24 μC has many options for the primary clock; can use an (a) internal oscillator, (b) external crystal, or (c) an external clock.

We will use the internal clock.

Internal Fast RC Oscillator + PLL

(a)

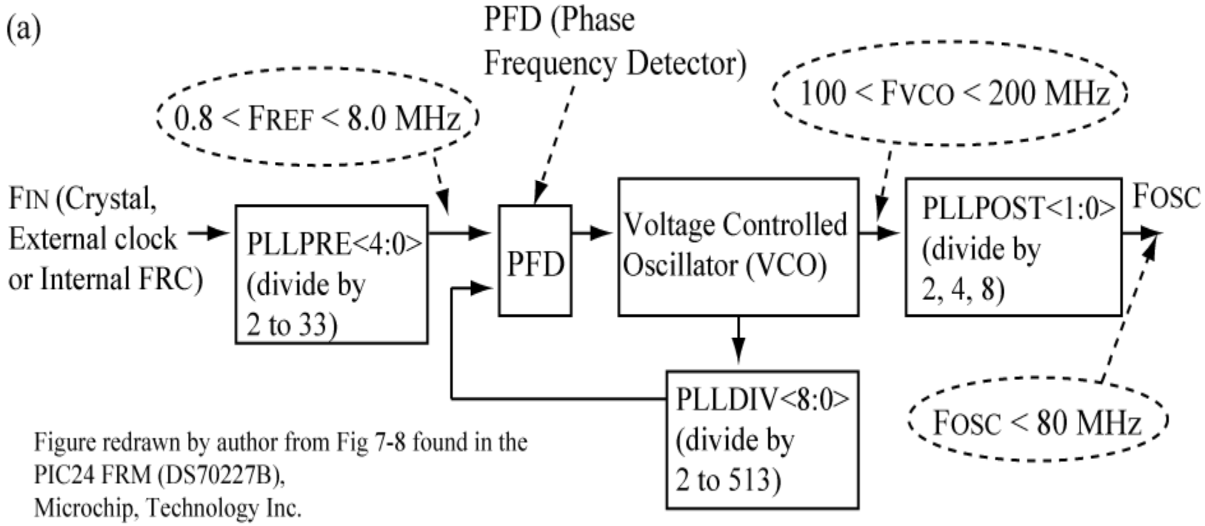


Figure redrawn by author from Fig 7-8 found in the PIC24 FRM (DS70227B), Microchip, Technology Inc.

(b)

$$F_{OSC} = F_{IN} \times \left(\frac{PLLDIV + 2}{(PLLPRE + 2) \times 2(PLLPOST + 1)}\right)$$

Sample Calculations:

	TUN	FIN	PLL Calculation	FOSC
(1) FRC 7370000	–19	6844888	$6844888 \times \left(\frac{185 + 2}{(6 + 2) \times 2(0 + 1)}\right)$	79999623
(2) Crystal 8000000	n/a	8000000	$8000000 \times \left(\frac{38 + 2}{(0 + 2) \times 2(0 + 1)}\right)$	80000000

Our examples use this! Internal FRC + PLL configured for 80MHz.

Configuration Bits

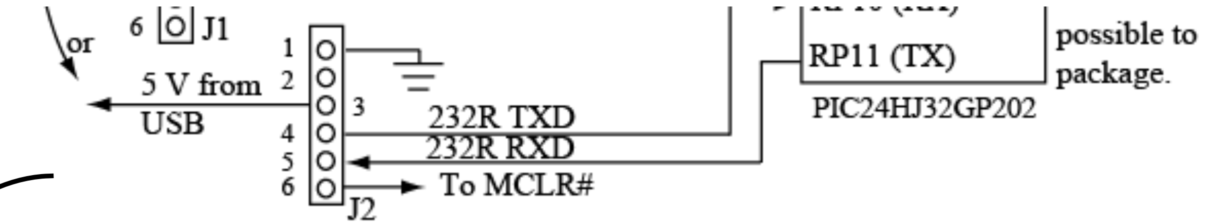
Configuration bits are stored at special locations in program memory to control various processor options. Configuration bits are only read at power up.

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

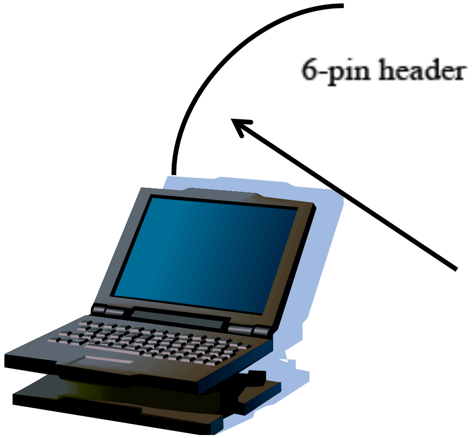
The file *pic24_config.c* file included by the sample programs used in lab specifies configuration bits used for all lab exercises.

We will not cover configuration bit details in this class; refer to the PIC24 datasheet for more information if interested.

The PC Serial Interface



6-pin header for FTDI TTL-232R-3.3V USB-to-TTL cable (PC serial communication link)



We use a special USB-to-Serial cable to connect our board to the PC. This serial interface outputs 3.3 V levels compatible with the PIC24 µC pins (careful, most USB-to-Serial cables use +/- 9V levels!!).

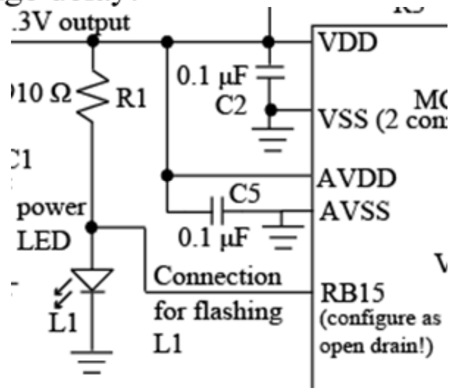
The serial interface will be used for ASCII input/output to PIC24 µC, as well as for downloading new programs via the Bully Serial Bootloader (winbootldr.exe).

ledflash_nomacros.c

```
#include "pic24_all.h"

/**
A simple program that flashes the Power LED.
*/

//a naive software delay function
void a_delay(void){
  uint16 u16_i,u16_k;
  // change count values to alter delay
  for (u16_k = 1800; --u16_k;) {
    for(u16_i = 1200 ; --u16_i ;);
  }
}

int main(void) {
  configClock();      //clock configuration
  /********** PIO config **********/
  _ODCB15 = 1;            //enable open drain
  _TRISB15 = 0;           //Config RB15 as output
  _LATB15 = 0;            //RB15 initially low
  while (1) {             //infinite while loop
   a_delay();            //call delay function
   _LATB15 = !_LATB15;   //Toggle LED attached to RB15
  } // end while (1)
}
```

Includes several header files, discussed later in this chapter.

A subroutine for a software delay. Change `u16_i`, `u16_k` initial values to change delay.



Infinite loop that blinks the LED. Only exit is through MCLR# reset or power cycle.

ledflash.c

Defined in device-specific header file in *include\devices* directory in the book source distribution.
Macro **CONFIG_RB15_AS_DIG_OD_OUTPUT()** configures RB15 as an open drain output and contains the statements **_TRISB15=0, _ODCB15 = 1**

LED1 macro makes changing of LED1 pin assignment easier, also improves code clarity.

DELAY_MS(ms) macro is defined in *include\pic24_delay.h* in the book source distribution, **ms** is a **uint32** value.

```
#include "pic24_all.h"

/**
A simple program that
flashes an LED.
*/

#define CONFIG_LED1() CONFIG_RB15_AS_DIG_OD_OUTPUT()

#define LED1  _LATB15

int main(void) {

  configClock();    //clock configuration
  /********** PIO config **********/
  CONFIG_LED1();   //config PIO for LED1
  LED1 = 0;

  while (1) {
    DELAY_MS(250);  //delay
    LED1 = !LED1;   // Toggle LED
  } // end while (1)
}
```

echo.c

```
#include "pic24_all.h"
/**
"Echo" program which waits for UART RX character and echos it back +1.
Use the echo program to test your UART connection.
*/


int main(void) {
  uint8 u8_c;

  configClock();
  configHeartbeat();
  configDefaultUART(DEFAULT_BAUDRATE);
  printResetCause();
  outString(HELLO_MSG);

  /** Echo code ********/
  // Echo character + 1
  while (1) {
    u8_c = inChar();   //get character
    u8_c++;            //increment the character
    outChar(u8_c);     //echo the character
  } // end while (1)
}
```

`configHeartbeat(void)` function defined in *common\pic24_util.c*. Configures heartbeat LED by default on RB15.

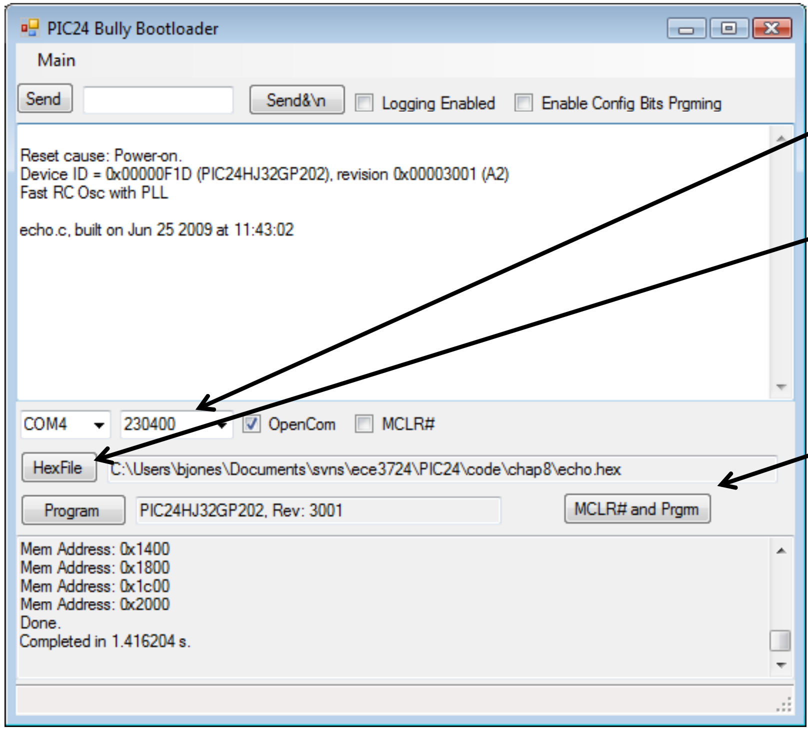
`configDefaultUART(uint32 u32_baudRate)` function defined in *common\pic24_serial.c*. This initializes the UART1 module for our reference system.

`printResetCause(void)` function defined in *common\pic24_util.c*. Prints info string about reset source.

`outString(char* psz_s)` function defined in *common\pic24_uart1.c*. Sends string to UART. `HELLO_MSG` macro default is file name, build date.

Testing your PIC24 System

After you have verified that your hookup provides 3.3 V and turns on the power LED, the TA will program your PIC24 μC bootloader firmware. Use to program your PIC24 with the hex file produced by the echo.c program and verify that it works.



(a) Select correct COM port, baud rate of 230400, open the COM port.

(b) Browse to hex file

(c) To program, press the 'MCLR# and Prgm' while power is on.

Reading the PIC24 Datasheets

- You MUST be able to read the PIC24 datasheets and find information in them.
  - The notes and book 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).
- The PIC24 Family reference manual has difference sections for each major subsystem.
- The component datasheet for the PIC24HJ32GP202 has summary information, you will need to refer the family reference manual most often.

PIC24 Reset

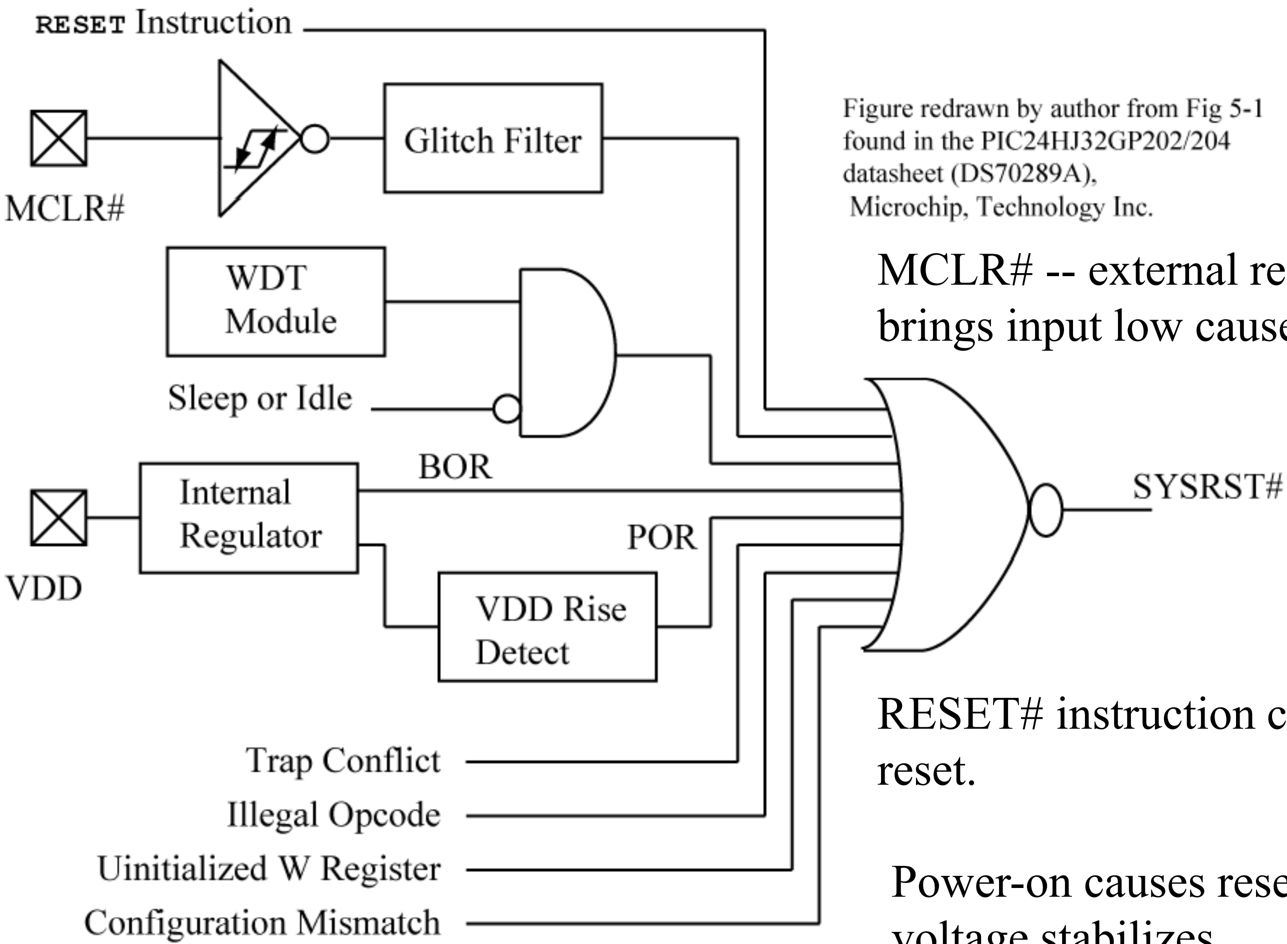


Figure redrawn by author from Fig 5-1 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip, Technology Inc.

MCLR# -- external reset button brings input low causes reset.

RESET# instruction causes reset.

Power-on causes reset after voltage stabilizes.

What RESET type occurred?

Figure redrawn by author from Table 5-1 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip, Technology Inc.

Flag Bit	Set by:	Cleared by:
TRAPR (RCON<15>)	Trap conflict event	POR, BOR
IOPUWR (RCON<14>)	Illegal opcode or initialized W register access	POR, BOR
CM (RCON<9>)	Configuration Mismatch	POR,BOR
EXTR (RCON<7>)	MCLR# Reset	POR
SWR (RCON<6>)	`reset` instruction	POR, BOR
WDTO (RCON<4>)	WDT time-out	`pwrsav` instruction, `clrwdt` instruction, POR,BOR
SLEEP (RCON<3>)	`pwrsav #0` instruction	POR,BOR
IDLE (RCON<2>)	`pwrsav #1` instruction	POR,BOR
BOR (RCON<1>)	BOR	n/a
POR (RCON<0>)	POR	n/a

Note: All Reset flag bits may be set or cleared by the user software.

Bits in the RCON special function register tell us what type of reset occurred.

`printResetCause()` function

Simplified version of `printResetCause()`, see book CD-ROM for full version.

```
void printResetCause(void) {
  if (_SLEEP) {
    outString("\nDevice has been in sleep mode\n"); _SLEEP = 0;
  }
  if (_IDLE) {
    outString("\nDevice has been in idle mode\n");  _IDLE = 0;
  }
  outString("\nReset cause: ");
  if (_POR) {
    outString("Power-on.\n");  _POR = 0; _BOR = 0; //clear both
  } else { //non-POR causes
   if (_SWR) {
    outString("Software Reset.\n");              _SWR = 0;    }
   if (_WDTO) {
    outString("Watchdog Timeout. \n");           _WDTO = 0;   }
   if (_EXTR) {
    outString("MCLR assertion.\n");              _EXTR = 0;   }
   if (_BOR) {
    outString("Brown-out.\n");                   _BOR = 0;    }
   if (_TRAPR) {
    outString("Trap Conflict.\n");               _TRAPR = 0;  }
   if (_IOPUWR) {
    outString("Illegal Condition.\n");           _IOPUWR = 0; }
   if (_CM) {
    outString("Configuration Mismatch.\n");      _CM = 0;     }
  }//end non-POR causes
  checkDeviceAndRevision();
  checkOscOption();
}
```

Check each bit, print a message, clear the bit after checking it.

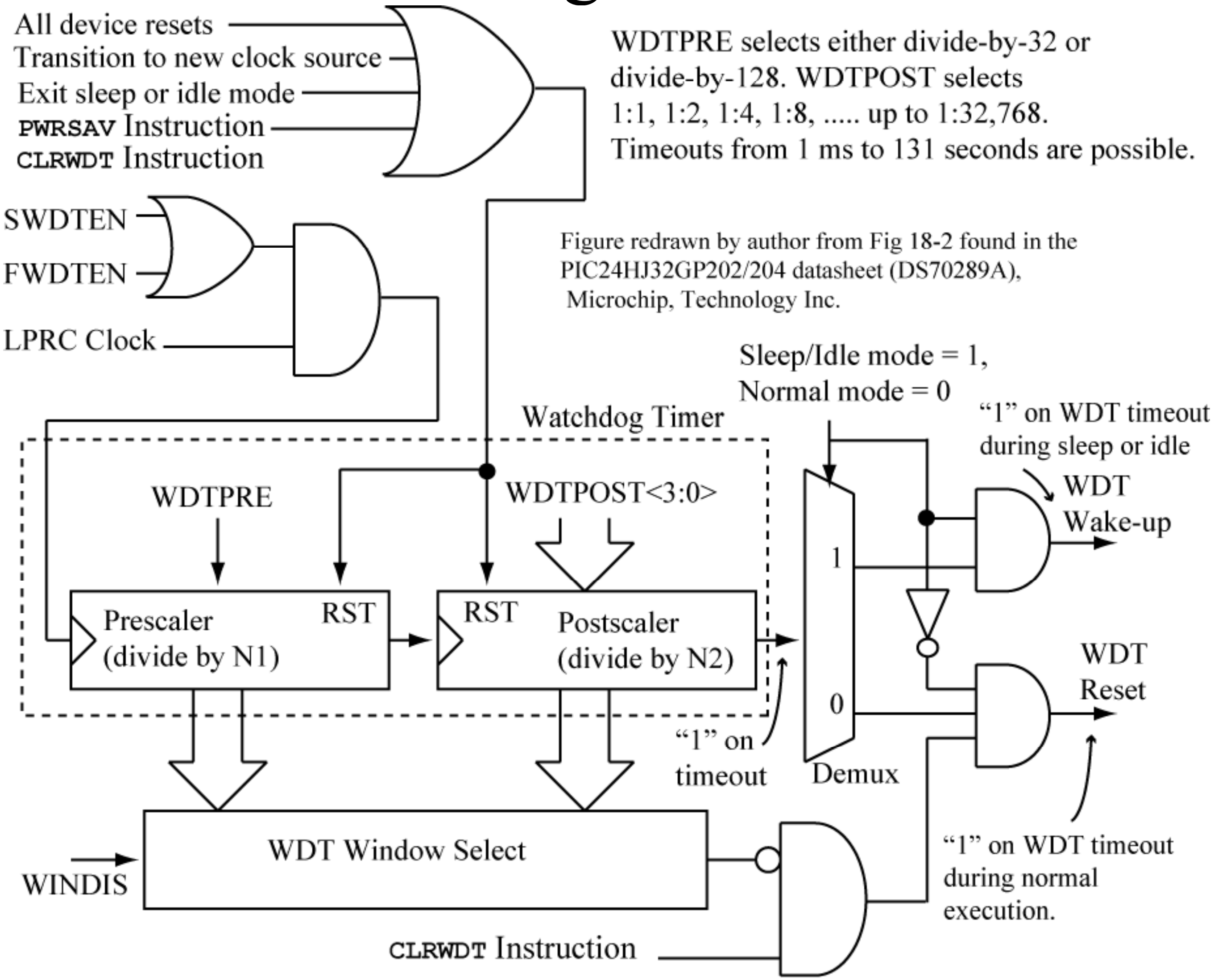
A status bit is cleared if it has been set.

Print status on processor ID and revision, and clock source.

Watchdog Timer

WDTPRE selects either divide-by-32 or divide-by-128. WDTPOST selects 1:1, 1:2, 1:4, 1:8, ... up to 1:32,768. Timeouts from 1 ms to 131 seconds are possible.

Figure redrawn by author from Fig 18-2 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip, Technology Inc.



WDT Specifics

Using free-running RC oscillator, frequency of about 32.768 kHz, runs even when normal clock is stopped.

Watchdog timeout occurs when counter overflows from max value back to 0. The timeout period is

WDT timeout = 1/32.768kHz x (WDTPRE) x (WDTPOST)

Times from 1 ms to 131 seconds are possible, bootloader firmware set for about 2 seconds.

A WDT timeout during normal operation RESETS the PIC24.

A WDT timeout during sleep or idle mode (clock is stopped) wakes up the PIC24 and resumes operations.

The `clrwdt` instruction clears the timer, prevents overflow.

WDT Uses

Error Recovery: If the CPU starts a hardware operation to a peripheral, and waits for a response, can break the CPU from an infinite wait loop by reseting the CPU if a response does not come back in a particular time period.

Wake From Sleep Mode: If the CPU has been put in a low power mode (clock stopped), then can be used to wake the CPU after the WDT timeout period has elapsed.

Power Saving Modes

Sleep: Main clock stopped to CPU and all peripherals. Can be awoke by the WDT. Use the `pwrsav #0` instruction.

Idle: Main clock stopped to CPU but not the peripherals (UART can still receive data). Can be awoke by the WDT. Use the `pwrsav #1` instruction.

Doze: Main clock to CPU is divided by Doze Prescaler (/2, /4, … up to /128). Peripheral clocks unaffected, so CPU runs slower, but peripherals run at full speed – do not have to change baud rate of the UART.

Current Measurements

Mode	PIC24HJ32GP202 @40MHz (mA)	PIC24FJ64GA002 @16 MHz (mA)
Normal	42.3	5.6
Sleep	0.030	0.004
Idle	17.6	2.0
Doze/2	32.2	4.0
Doze/128	17.9	2.0

Doze current(/N mode) = Idle current + (Normal current − Idle current)/N

The idle current is the base current of the chip with the CPU stopped and the clock going to all of the peripherals. So any doze mode current adds to this base.

reset.c Program

```
#include "pic24_all.h"
//Experiment with reset, power-saving modes

_PERSISTENT uint8 u8_resetCount;
int main(void) {

 configClock();
 configPinsForLowPower();
 configHeartbeat();
 configDefaultUART(DEFAULT_BAUDRATE);
 outString(HELLO_MSG);

 if (_POR) {
    u8_resetCount = 0;      // if power on reset, init the reset count variable
  } else {
    u8_resetCount++;        //keep track of the number of non-power on resets
 }

 if (_WDTO) {
    _SWDTEN = 0;            //If Watchdog timeout, disable WDT.
 }
 printResetCause();        //print statement about what caused reset
 //print the reset count
 outString("The reset count is ");
 outUint8(u8_resetCount);  outChar('\n');
 while (1) {
    ...See the next figure...
  }
}
```

`_PERSISTENT` variables are not initialized by *C* runtime code.

`configPinsForLowPower(void)` function defined in *common\pic24_util.c*. Configs parallel port pins as all inputs, with weak pull-ups enabled.

`_POR bit` is set to a "1" by power-on reset. The function `printResetCause()` clears `_POR` to a "0".

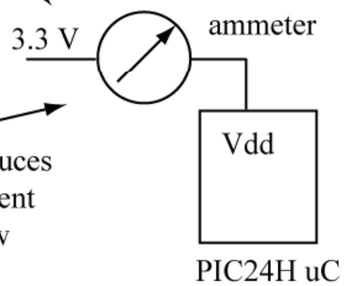
`_WDTO bit` is set to a "1" by watch dog timer timout. The function `printResetCause()` clears `_WDTO` to a "0".

reset.c Program (cont)

```
//...see previous figure for rest of main()
while (1) {
    uint8 u8_c;
    u8_c = printMenuGetChoice();  //Print menu, get user's choice
    delayMs(1);  //let characters clear the UART executing choice
    switch (u8_c) {
      case '1':                //enable watchdog timer
        _SWDTEN = 1;           //WDT ENable bit = 1
        break;
      case '2':               //sleep mode
        asm("pwrsav #0");     //sleep
        break;
      case '3':               //idle mode
        asm("pwrsav #1");     //idle
        break;
      case '4':
        _SWDTEN = 1;          //WDT ENable bit = 1
         asm("pwrsav #0"); //sleep
        outString("after WDT enable, sleep.\n"); //executed on wakeup
        break;
      case '5':       //doze mode
        _DOZE = 1;    //chose divide by 2
        _DOZEN= 1;    //enable doze mode
         break;
      case '6':       //doze mode
        _DOZE = 7;    //chose divide by 128
        _DOZEN= 1;    //enable doze mode
       break;
      case '7':               //software reset
        asm("reset");         //reset myself
        break;
      default:
        break;
    }
  } // end while (1)
  return 0;
}
```

Reduces current draw

Reduces current draw



reset.c Operation

```
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003001 (A2)
FastRC Osc with PLL
The reset count is 0x00
'1' enable watchdog timer
'2' enter sleep mode
'3' enter idle mode
'4' enable watchdog timer and enter sleep mode      Menu printed by
'5' doze = divide by 2                              printMenuGetChoice()
'6' doze = divide by 128
'7' execute reset instruction
Choice: 1                                           (a) Enable WDT timer

...Menu is reprinted...
...2 seconds elapse...
Reset cause: Watchdog Timeout:                      (b) WDT timer reset
...Device ID info...
The reset count is 0x01                             (c) Reset count is now 1
...Menu is reprinted...
Choice: 2                                           (d) Sleep mode selected,
...non responsive, press                            program hangs
...MCLR button to wakeup...
Device has been in sleep mode                       (e) from printResetCause()
Reset cause: MCLR assertion.                        (f) pressed MCLR to escape
...Device ID info...                                sleep mode.
The reset count is 0x02                             (g) Reset count is now 2
...Menu is reprinted...
Choice: 4                                           (h) WDT enabled, sleep
...enters sleep mode...                             mode entered.
...WDT expires after 2 second causing wakeup
after WDT enable, sleep.                            (i) After WDT wakeup
...menu is reprinted from loop, then after 2 more seconds
...WDT expires again, causing WDT reset.
Device has been in sleep mode
Reset cause: Watchdog Timeout:
...Device ID info...
The reset count is 0x03                             (j) Reset count is now 3
```