Microcomputers

PIC24 Startup

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 `uint32` data type is used where `uint8` 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
  - How many general purpose registers does the uP have?
  - What are the capabilities of the instruction set?
    - For example, is there hardware support for multiply/divide?
  - 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

From .c to .hex

- C Code (.c)
- Compilation
- Unoptimized Assembly Code
- Optimization
- Optimized Assembly Code (.o)
- Assembly
- Machine code (.s)
- Link
- Executable (.hex)

Example Optimization

- \( k = k + y \)
- \( k = k + y \)
- \( \text{compilation} \)
- \( \text{optimization} \)
- \( \text{assembly} \)
- \( \text{link} \)
- \( \text{executable} \)

```
mov j,MO : MO = j
add a
rs = k + MO = k + y
mov j,MO : MO = j
add k
rs = k + MO = k + y
```

- \( \text{W0 already contains } y, \text{ remove second mov instruction} \)
Referring to Special Function Registers

```c
#include "pic24_all.h"
```

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

Special Function Registers can be accessed like variables:

```c
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 function 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:

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

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

```c
OSCCONbits.NOSC = 2; // bit field in OSCCON 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_`.

```c
_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:

```c
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.
PIC24H128GP502 µC

Pin functions are controlled via special function registers in the PIC.

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

A Wall transformer (or battery) provides a variable DC supply voltage (unregulated means that voltage can vary significantly depending on current being drawn).

The LM2937-3.3 voltage regulator provides a regulated +3.3V. Voltage will stay stable up to maximum current rating of device.
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 ~ \( \frac{3.3V - 2.2V}{470 \, \Omega} = \frac{1.1V}{470 \, \Omega} = 2.36 \, mA \)
The PIC24 μC has many options for generating a clock; can use an external crystal or internal oscillator.

We will use the internal clock.
**Lecture 5-17**

**Internal Fast RC Oscillator + PLL**

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

**Configuration Bits**

- **Configuration bits** are stored at special locations in 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, 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 discuss the meaning of the configuration bit options as it is necessary.
The PC Serial Interface

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 from/to PC using the Putty terminal emulation program.

ledflash_nomacros.c (Modified for Microstick II)

```c
#include "pic24_all.h" // Simple program that flashes the LED.

void a_delay(void) { // A naive software delay function
    uint16_t ul6_i,ul6_k;
    // change count values to alter delay
    for (ul6_k=1800; --ul6_k;)
    {
        for (ul6_i = 1200; --ul6_i ;);
    }
}

int main(void) {
    configClock(); // Clock configuration
    /********** GPIO config **********/
    _ODCA0 = 0; // Disable open drain
    _TRISA0 = 0; // Config RA0 as output
    _LATA0 = 0; // RA0 initially low
    while (1) { // Infinite while loop
        a_delay(); // Call delay function
        _LATA0 = !_LATA0; // Toggle LED attached to RA0
    } // end while (1)
}
```

A simple subroutine for implementing a software delay
### ledflash.c (Modified for Microstick II)

```c
#include "pic24_all.h"

// A simple program
// that flashes an LED

define CONFIG_LED1()  CONFIG_RA0_AS_DIG_OUTPUT()
#define LED1  _LATA0   //_LATA0 is port register for RA0

int main(void) {
    configClock(); //Configure clock source for processor
    /********** GPIO config **********/
    CONFIG_LED1();
    LED1 = 0;
    while (1) {
        DELAY_MS(250); //delay long enough to see LED blink
        LED1 = !LED1;  // Toggle LED
    }  // end while (1)
}
```

Defined in header file in include/devices. MACRO CONFIG_RA0_AS_DIG_OUTPUT() includes the statements: _TRISA0 = 0;  _PCFG0 = 1;

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

Software delay MACRO to delay 250 ms

Original file c:\microchip\chap8\ledflash.c

---

### echo.c

```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\uart1.c. Configures heartbeat LED by default on RB15.

cfgDefaultsUART(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.
# echo.c (Modified for Microstick II)

```c
#include "pic24_all.h"

// "Echo" program which waits for UART1 RX character and echos it back +1. Use the program to test your UART connection.

int main(void) {
    uint8_t u8_c;
    configClock();
    configPinsForLowPower();
    configDefaultUART(9600);
    printResetCause(); // print statement about what caused reset
    outString(HELLO_MSG);
    while (1) {
        u8_c = inChar(); // get character
        u8_c++; // increment the character
        outChar(u8_c); // echo the character
    } // end while (1)
}
```

---

## echo.c Output

- Echo.c output with user input “123abc”
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 PIC24HJ128GP502 has summary information, you will need to refer to the family reference manual most often.

PIC24 Reset

- MCLR# -- external reset button brings input low causes reset.
- RESET instruction causes reset.
- Power-on causes reset after voltage stabilizes.
What RESET type occurred?

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

<table>
<thead>
<tr>
<th>Flag Bit</th>
<th>Set by:</th>
<th>Cleared by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAPR (RCON&lt;15&gt;)</td>
<td>Trap conflict event</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>IOPUWR (RCON&lt;14&gt;)</td>
<td>Illegal opcode or initialized W register access</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>CM (RCON&lt;9&gt;)</td>
<td>Configuration Mismatch</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>EXTR (RCON&lt;7&gt;)</td>
<td>MCLR# Reset</td>
<td>POR</td>
</tr>
<tr>
<td>SWR (RCON&lt;6&gt;)</td>
<td>reset instruction</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>WDTO (RCON&lt;4&gt;)</td>
<td>WDT time-out</td>
<td>_psav instruction,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_clrwdt instruction,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>POR, BOR</td>
</tr>
<tr>
<td>SLEEP (RCON&lt;3&gt;)</td>
<td>_psav #0 instruction</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>IDLE (RCON&lt;2&gt;)</td>
<td>_psav #1 instruction</td>
<td>POR, BOR</td>
</tr>
<tr>
<td>BOR (RCON&lt;1&gt;)</td>
<td>BOR</td>
<td>n/a</td>
</tr>
<tr>
<td>POR (RCON&lt;0&gt;)</td>
<td>POR</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

Simplified printResetCause() function

```c
void printResetCause(void) {
    // Check each bit, print a message, clear the bit after checking.
    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
            if (_SWR) { outString("Software Reset.\n"); _SWR = 0; }
            if (_WDTO) { outString("Watchdog Timeout: "); _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
    }
}
```

Complete function defined in c:\microchip\lib\common\pic24_util.c
Watchdog Timer

WDTPRE selects either divide-by-32 or divide-by-128 operation. The prescaler is set by the WDTPRE Configuration bit (configuration bits (FWDT<4>).

WDTPOST<3:0>, (configuration bits (FWDT<3:0>), allow the selection of 16 settings, from 1:1 to 1:32,768. Using the prescaler and postscaler, time-out periods ranging from 1 ms to 131 seconds can be achieved.

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.
Sample WDT Timeout Calculation

• Give possible values for the WDTPRE and WDTPOST fields to enable a watchdog timer (WDT) reset of approximately 1 second for the PIC24 processor.
  – Assume the WDT is clocked using a 32.768kHz oscillator.
  – See Table 25-2 in PIC24HJ128GP502 datasheet for WDTPRE and WDTPOST definitions

\[
WDT\text{ timeout } = \frac{1}{32.768kHz} \times (WDTPRE) \times (WDTPOST)
\]

Assume WDTPRE=0 => 1:32 division

\[
WDTPOST = \frac{WDT\text{ timeout} \times 32768}{WDTPRE}
\]

\[
= 1 \times 32768/32 = 1024
\]

• Therefore, setting WDTPRE=0 and WDTPOST=1010 would suffice.

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 resetting 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 awoken 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 awoken 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.

```c
#include "pic24_all.h"

uint8_t printMenuGetChoice() {
    uint8_t u8_c;
    outString("'1' enable watchdog timer\n");
    outString("'2' enter sleep mode\n");
    outString("'3' enter idle mode\n");
    outString("'4' enable watchdog timer and enter sleep mode\n");
    outString("'5' doze = divide by 2\n");
    outString("'6' doze = divide by 128\n");
    outString("'7' execute reset instruction\n");
    outString("Choice: ");
    u8_c = inChar();
    outChar(u8_c);  // echo character
    outString("\n");  // newline
    return(u8_c);
}
```

_reset_.c (Modified for Microstick II)

_persistent_ variables are not initialized by C runtime code
```c
int main(void) {
    configClock(); // clock configuration
    configPinsForLowPower(); // config pins for low power since
        // we are measuring current
    configDefaultUART(9600); // serial port config
    outString(HELLO_MSG); // say Hello!

    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);
    outString("\n");
}
```

---

```c
while (1) {
    uint8_t u8_c;
    u8_c = printMenuGetChoice();
    DELAY_MS(1); // let characters clear the UART before 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
            outString("after sleep\n"); // never executed.
            break;
        case '3': // idle mode
            asm("pwrsav #1"); // idle
            outString("after idle\n"); // never executed.
            break;
        case '4':
            _SWDTEN = 1; // WDT Enable bit = 1
            asm("pwrsav #0"); // sleep
            outString("after WDT enabled, sleep.\n"); // executed on wakeup
            break;
    }
}
```
reset.c (Modified for Microstick II)

```c
reset.c Operation

reset.c, built on Feb 3 2017 at 15:11:37

Reset cause: MCLR assertion.
Device ID = 0x0000067D (PIC24HJ128GP502), revision 0x00003004 (unknown)
Fast RC 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
'5' doze = divide by 2
'6' doze = divide by 128
'7' execute reset instruction

Choice: 1
... Menu is reprinted
... 2 seconds elapse
Reset cause: Watchdog Timeout:
... Device ID info
The reset count is 0x01
... Menu is reprinted
```

Reduces current draw

3.3 V

Vdd

PIC24

ammeter

Menu printed by `printMenuGetChoice()`

Enable WDT

WDT timer reset

Reset count is now 1
reset.c Operation

Choice: 2
... Nonresponsive, press
... MCLR# button to wakeup
Device has been in sleep mode
Reset cause: MCLR assertion.
... Device ID info
The reset count is 0x02
... Menu is reprinted
Choice: 4
... Enters sleep mode
... WDT expires after 2 seconds causing wakeup
after WDT enable, sleep.
... 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
Reset count is now 2
Reset count is now 3

What do you have to know?

- Understand initial hookup schematic for the PIC24 μC
- Watchdog timer operation
- Sleep mode operation
- ledflash.c, echo.c, reset.c basic operation