University of British Columbia
Electrical and Computer Engineering
EECE281/282

Module 6 - Interfacing With Transistors

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 09, 2015

Objectives

- Understand the current/voltage capability of microcontroller digital I/O pins.
- Drive high voltage/current loads using BJTs or MOSFETS.
- Optically isolate digital I/O pins.
- Use Interrupt Service Routines in C for the 8051 microcontroller.
- Understand and use PWM.

Logic Levels: Input/output margins

- Both logic zero and logic one have input and output margins:
  - For digital outputs:
    - V_{OH}: Output high voltage
    - V_{OL}: Output low voltage
  - For digital inputs:
    - V_{IH}: Input high voltage
    - V_{IL}: Input low voltage
- V_{OH}, V_{OL}, V_{IH}, and V_{IH} depend on the logic family used as well as the voltage used to power the ICs.

Logic Levels: Input/output margins

For TTL For CMOS

@V_{CC}/V_{DD}=5V		
	CMOS/HC	TTL
VOH	≥ 4.7V (4.2V)	≥ 3.3V
VIH	≥ 3.7V	≥ 2.0V
VIL	≤1.3V	≤0.8V
VOL	≤0.2V	≤0.35V

Logic Levels: Maximum output currents

- Also, there is a maximum amount of current available from the outputs:

	CMOS/HC	TTL
Source	20 mA	0.4 mA
Sink	20 mA	8 mA

Generally, a digital output can sink more current than it can source!

Logic Levels: The C8051F38C microcontroller

C8051F380/1/2/3/4/5/6/7/C

Table 5.3. Port I/O DC Electrical Characteristics

V_{DD} = 2.7 to 3.6 V, –40 to +85 °C unless otherwise specified.

Parameter	Test Condition	Min	Typ	Max	Unit
Output High Voltage	I_{OH} = –3 mA, Port I/O push-pull I_{OH} = –10 µA, Port I/O push-pull I_{OH} = –10 mA, Port I/O push-pull	V_{DD} – 0.7 V_{DD} – 0.1 —	— — V_{DD} – 0.8	— — —	V
Output Low Voltage	I_{OL} = 8.5 mA I_{OL} = 10 µA I_{OL} = 25 mA	— — —	— — 1.0	0.6 0.1 —	V
Input High Voltage		2.0	—	—	V
Input Low Voltage		—	—	0.8	V
Input Leakage Current	Weak Pullup Off Weak Pullup On, V_{IN} = 0 V	— —	— 15	±1 50	µA

If you need to connect a heavy load (lots of current) or a load with a high voltage requirement to the AT89LP51RB2, some sort of interfacing is required!

C8051F38C microcontroller pin configuration.

SFR Definition 20.10. P1MDOUT: Port 1 Output Mode

Bit	7	6	5	4	3	2	1	0
Name	P1MDOUT[7:0]							
Type	R/W							
Reset	0	0	0	0	0	0	0	0

SFR Address = 0xA5; SFR Page = All Pages

Bit	Name	Function
7:0	P1MDOUT[7:0]	**Output Configuration Bits for P1.7–P1.0 (respectively).** These bits are ignored if the corresponding bit in register P1MDIN is logic 0. 0: Corresponding P1.n Output is open-drain. 1: Corresponding P1.n Output is push-pull.

For example, to set P1.0 as output in C:

```
P1MDOUT |= 0B_0000_0001;
```


C8051F38C microcontroller pin configuration.

SFR Definition 20.9. P1MDIN: Port 1 Input Mode

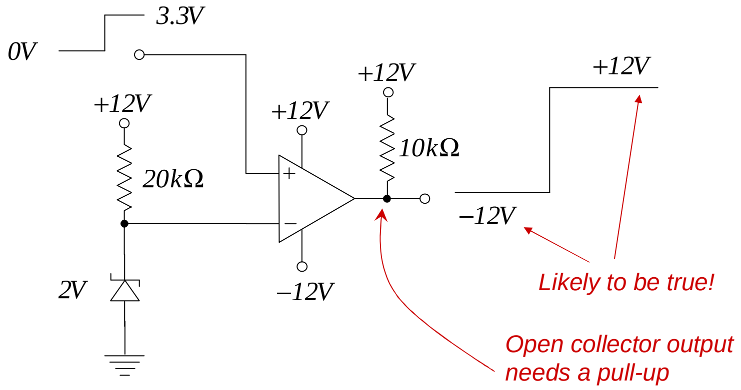
Bit	7	6	5	4	3	2	1	0
Name	P1MDIN[7:0]							
Type	R/W							
Reset	1*	1	1	1	1	1	1	1

SFR Address = 0xF2; SFR Page = All Pages

Bit	Name	Function
7:0	P1MDIN[7:0]	**Analog Configuration Bits for P1.7–P1.0 (respectively).** Port pins configured for analog mode have their weak pullup, digital driver, and digital receiver disabled. 0: Corresponding P1.n pin is configured for analog mode. 1: Corresponding P1.n pin is not configured for analog mode.

Voltage Level Conversion

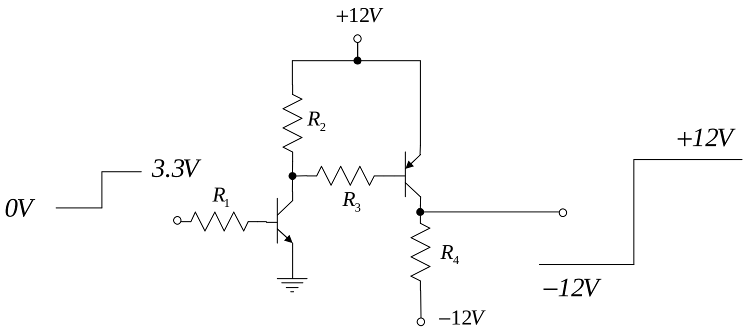
- Using an comparator, such as the LM393 or LM339:



Voltage Level Conversion

- Using BJTs:

BJT stands for Bipolar Junction Transistor

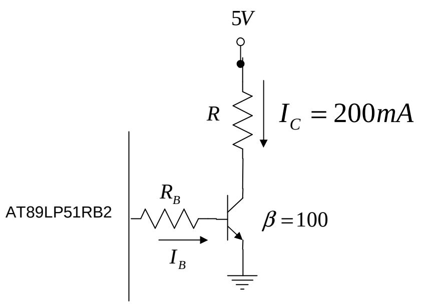


Modes of Operation of BJTs

- BJTs are current amplifying devices. Therefore, I use currents to assess the operation mode of the BJT:
  - Cut off: $I_C=0$
  - Active: $I_C=\beta x I_B$
  - Saturation: $I_C<\beta x I_B$
- β is the current gain factor. Look for h_{fe} in the datasheets. Assume it as constant…
- For interfacing with the 8051 we will be using the BJTs in cut-off or saturation only!

Driving High Currents

- The "classic" way:



$$R_B = \frac{3.3V - 0.7V}{I_B}$$

$$I_B > \frac{200mA}{\beta}$$

$$I_B > 2mA$$

WARNING: Way too much source current for many microcontrollers! Ok for the C8051F38C.

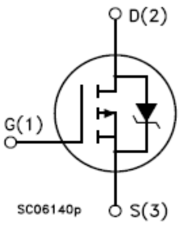
Using MOSFETs

- MOSFETs (Metal Oxide Semiconductor Field Effect Transistors) are voltage controlled devices.
- There are many kinds of MOSFETs. The ones often used are channel enhancement PMOS and NMOS.
- The two parameters used to design with MOSFETs (as switches) are the Threshold Voltage V_t, and the resistance between drain and source R_{DS}.

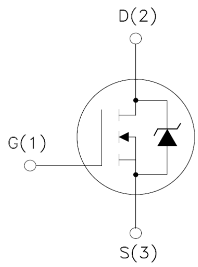
MOSFET operation modes

- As with the BJTs, there are three operating modes:
  - Cutoff: VGS<Vt
  - Triode: $V_{DS}<(V_{GS}-V_t)$
  - Saturation: $V_{DS}\geq(V_{GS}-V_t)$
- To use the MOSFET as switch, operate it in the cutoff and triode modes. If you operate it in saturation it will get really-really hot!

STP12PF06 and STP16NF06 MOSFETS



STP12PF06

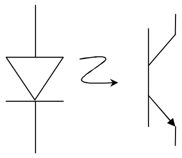


STP16NF06

Optocouplers

- An optocoupler is a combination of a light source and a photosensitive element
- You use an optocoupler when you want to isolate high or very high voltages, inductive circuits, or “noisy” circuits from the microcomputer system.
- The typical optocoupler consists of an infrared LED and a NPN BJT.
- The BJT usually doesn’t have a base pin! Instead it is the light from the LED what is used to saturate the transistor.

Designing with Optocouplers



Some optocouplers include a base pin!

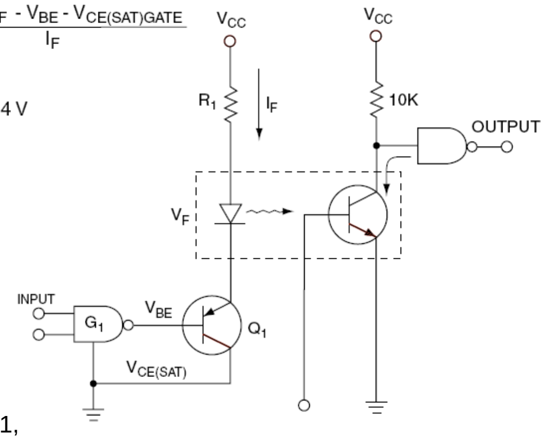
- When designing with optocouplers you take into consideration the following parameters:
  - The current transfer ratio (CTR) is a parameter similar to the DC current amplification ratio of a transistor (β) and is expressed as a percentage indicating the ratio of the output current (I_C) to the input current (I_F).
    $CTR(\%)=(I_C/I_F) \times 100$
  - The Diode forward voltage (1.2 to 1.4V).
  - The maximum diode forward current (around 50mA max).
  - The BJT saturation voltage (0.1 to 0.4V).
  - The voltage isolation between the diode and the transistors (a few hundred volts to thousands of volts)

Optocouplers circuits

$$R_1 = \frac{V_{CC} - V_F - V_{BE} - V_{CE(SAT)GATE}}{I_F}$$

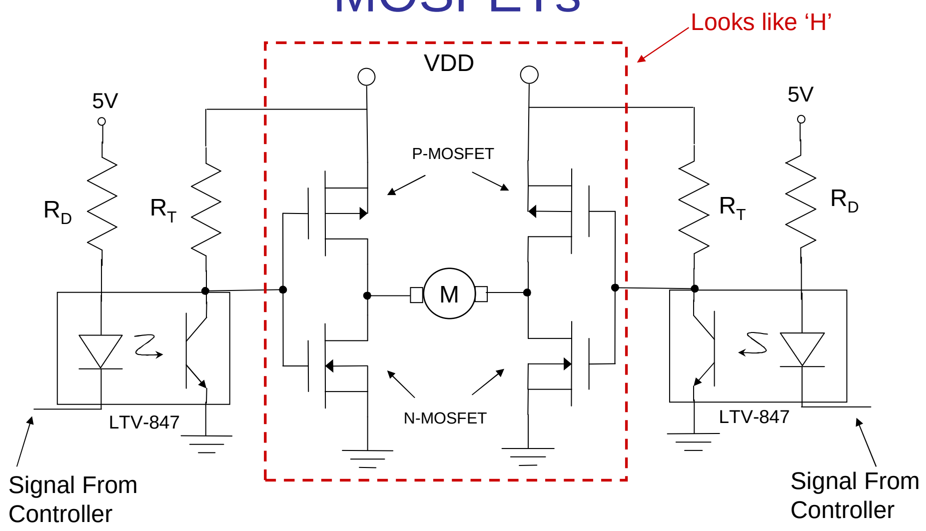
$V_{BE(Q_1)} = 0.6\ V$

$V_{CE(SAT)(G_1)} = 0.4\ V$

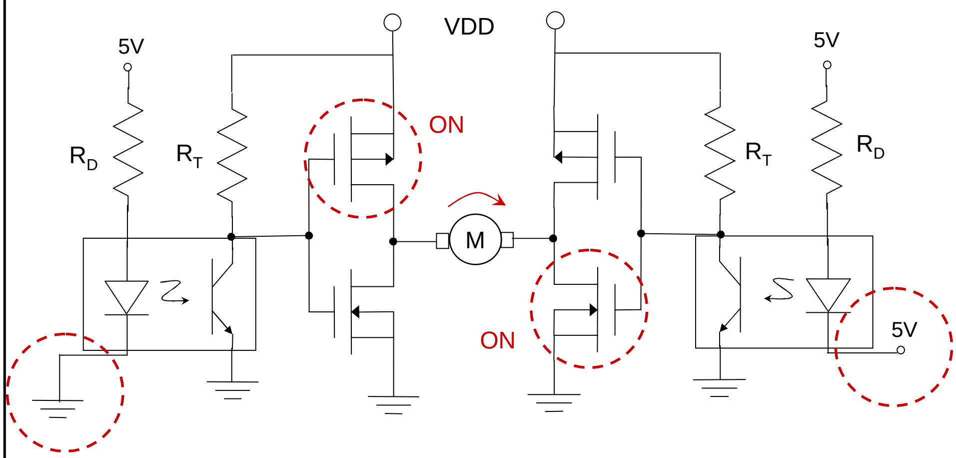


This is from AN-3001, Fairchild Semiconductors

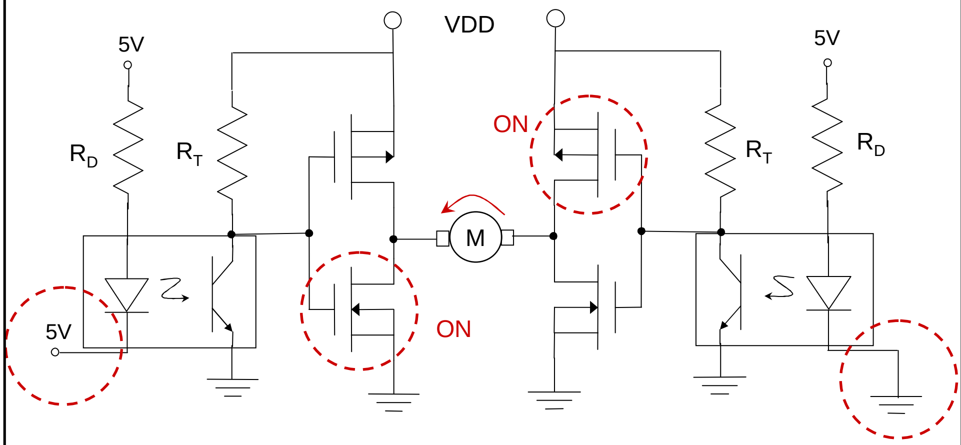
H-Bridge with Optocouplers and MOSFETs



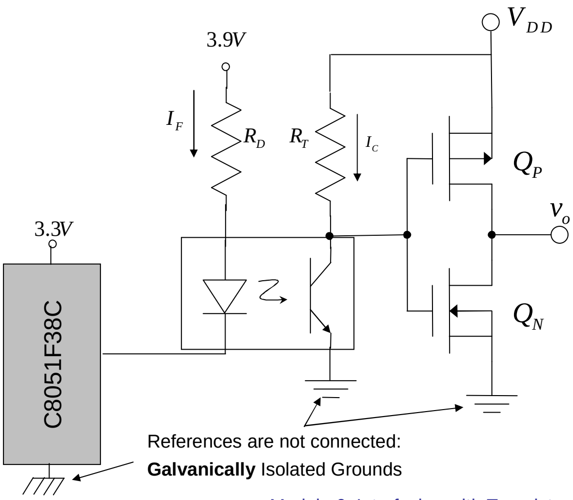
H-Bridge with Optocouplers and MOSFETs CW Rotation



H-Bridge with Optocouplers and MOSFETs CCW Rotation



H-Bridge with Optocouplers and MOSFETs: Calculations.



Assume R_T=98kΩ, V_{DD}=10V, $V_{CE(sat)}$=0.2, LED forward voltage=1.2V, CTR=50%. Find R_D!

$$I_C = \frac{10V - 0.2V}{98k\Omega} = 0.1mA$$

$$CTR = \frac{I_C}{I_F} \times 100$$

$$I_F = \frac{I_C \times 100}{CTR} = \frac{0.1mA \times 100}{50} = 0.2mA$$

$$R_D < \frac{3.9V - 1.2V}{0.2mA} = 13.5k\Omega$$

LTV-847 Optocoupler

- CTR=50%
- Diode forward voltage=1.4 max.
- Maximum diode forward current is 50mA
- The BJT saturation voltage is less than 0.12V!
- Voltage isolation 5000V_{RMS}

Interrupt Service Routines in C

C51 allows *Interrupt Service Routines* (ISRs) to be coded in C, by using a predefined format, for example:

```
void timer_isr (void) interrupt (1) using (1)
{
   ...
}
```

Number associated with the interrupt source

Optional register bank (0 to 3)

Interrupt # (Standard 8051)

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000b
2	External 1	0x0013
3	Timer 1	0x001b
4	Serial	0x0023
5	Timer 2	0x002b

Interrupt # (C8051F38x)

C8051F380/1/2/3/4/5/6/7/C

Table 16.1. Interrupt Summary

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag	Bit Address?	Cleared by HW?	Enable Flag	Priority Control
Reset	0x0000	Top	None	N/A	N/A	Always Enabled	Always Highest
External Interrupt 0 (INT0)	0x0003	0	IE0 (TCON.1)	Y	Y	EX0 (IE.0)	PX0 (IP.0)
Timer 0 Overflow	0x000B	1	TF0 (TCON.5)	Y	Y	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (INT1)	0x0013	2	IE1 (TCON.3)	Y	Y	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	0x001B	3	TF1 (TCON.7)	Y	Y	ET1 (IE.3)	PT1 (IP.3)
UART0	0x0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	Y	N	ES0 (IE.4)	PS0 (IP.4)
Timer 2 Overflow	0x002B	5	TF2H (TMR2CN.7) TF2L (TMR2CN.6)	Y	N	ET2 (IE.5)	PT2 (IP.5)
SPI0	0x0033	6	SPIF (SPI0CN.7) WCOL (SPI0CN.6) MODF (SPI0CN.5) RXOVRN (SPI0CN.4)	Y	N	ESPI0 (IE.6)	PSPI0 (IP.6)

Interrupt # (C8051F38x) Cont.

SMB0	0x003B	7	SI (SMB0CN.0)	Y	N	ESMB0 (EIE1.0)	PSMB0 (EIP1.0)
USB0	0x0043	8	Special	N	N	EUSB0 (EIE1.1)	PUSB0 (EIP1.1)
ADC0 Window Compare	0x004B	9	AD0WINT (ADC0CN.3)	Y	N	EWADC0 (EIE1.2)	PWADC0 (EIP1.2)
ADC0 Conversion Complete	0x0053	10	AD0INT (ADC0CN.5)	Y	N	EADC0 (EIE1.3)	PADC0 (EIP1.3)
Programmable Counter Array	0x005B	11	CF (PCA0CN.7) CCFn (PCA0CN.n)	Y	N	EPCA0 (EIE1.4)	PPCA0 (EIP1.4)
Comparator0	0x0063	12	CP0FIF (CPT0CN.4) CP0RIF (CPT0CN.5)	N	N	ECP0 (EIE1.5)	PCP0 (EIP1.5)
Comparator1	0x006B	13	CP1FIF (CPT1CN.4) CP1RIF (CPT1CN.5)	N	N	ECP1 (EIE1.6)	PCP1 (EIP1.6)
Timer 3 Overflow	0x0073	14	TF3H (TMR3CN.7) TF3L (TMR3CN.6)	N	N	ET3 (EIE1.7)	PT3 (EIP1.7)
VBUS Level	0x007B	15	N/A	N/A	N/A	EVBUS (EIE2.0)	PVBUS (EIP2.0)
UART1	0x0083	16	RI1 (SCON1.0) TI1 (SCON1.1)	N	N	ES1 (EIE2.1)	PS1 (EIP2.1)
Reserved	0x008B	17	N/A	N/A	N/A	N/A	N/A
SMB1	0x0093	18	SI (SMB1CN.0)	Y	N	ESMB1 (EIE2.3)	PSMB1 (EIP2.3)
Timer 4 Overflow	0x009B	19	TF4H (TMR4CN.7) TF4L (TMR4CN.6)	N	N	ET4 (EIE2.4)	PT4 (EIP2.4)
Timer 5 Overflow	0x00A3	20	TF5H (TMR5CN.7) TF5L (TMR5CN.6)	Y	N	ET5 (EIE2.5)	PT5 (EIP2.5)

C8051F38x.h

```
// Interrupt Vector Numbers

#define INTERRUPT_INT0        0  // External Interrupt 0
#define INTERRUPT_TIMER0      1  // Timer 0 Overflow
#define INTERRUPT_INT1        2  // External Interrupt 1
#define INTERRUPT_TIMER1      3  // Timer 1 Overflow
#define INTERRUPT_UART0       4  // UART0
#define INTERRUPT_TIMER2      5  // Timer 2 Overflow
#define INTERRUPT_SPI0        6  // SPI0
#define INTERRUPT_SMBUS0      7  // SMBus0 Interface
#define INTERRUPT_USB0        8  // SMBus0 Interface
#define INTERRUPT_ADC0_WINDOW 9  // ADC0 Window Comparison
#define INTERRUPT_ADC0_EOC    10 // ADC0 End Of Conversion
#define INTERRUPT_PCA0        11 // PCA0 Peripheral
#define INTERRUPT_COMPARATOR0 12 // Comparator 0 Comparison
#define INTERRUPT_COMPARATOR1 13 // Comparator 1 Comparison
#define INTERRUPT_TIMER3      14 // Timer 3 Overflow
#define INTERRUPT_VBUS        15 // VBus Interrupt
#define INTERRUPT_UART1       16 // UART1
#define INTERRUPT_SMBUS1      18 // SMBus1 Interface
#define INTERRUPT_TIMER4      19 // Timer 4 Overflow
#define INTERRUPT_TIMER5      20 // Timer 5 Overflow
```

ISR Common Pitfalls.

1) Long and complex functions.
2) Calling standard library functions that are not reentrant.
3) Not clearing the interrupt flag.
4) Not making variables "volatile".
5) Using non-atomic variables.

1) Long and complicated functions

- Keep ISRs simple and quick!
- For periodic timer ISRs, the function must take less time than the interrupt rate it is programmed to serve! For example if the timer ISR takes 2 ms to run, the timer must be set to interrupt every 2 ms or more.
- It is customary to calculate the percentage of processor time used by an ISR. For example, if your ISR takes 2 ms, and is set to interrupt every 10 ms, then the ISR is using 20% of the processor time. You have 80% of the processor time available for other tasks!

2) Calling standard library functions that are not reentrant.

- Q: What is a reentrant function?
- A: In the context of this course, a reentrant function is function which does not write to global or static variables. That means that all the variables of the functions must be allocated in the stack!
  - Most microcontrollers have very little memory.
  - Having stack variables requires indirect access, which in turn is slow and requires more code memory.
  - Almost none of the standard library functions provided with C51 are reentrant. Even worst: that includes all basic integer arithmetic functions (+, -, *, /)!

2) Calling standard library functions that are not reentrant.

```
void Ext0_ISR (void) interrupt 0 using 2
{
    myvar++;
    printf( “myvar=%d\n” , myvar);
}
```

This may or may not work… If this is the only function using printf(), then it will probably work. Otherwise, it will crash your code!

WARNING: Very common mistake!!!

3) Not Clearing The Interrupt Flag

- Your ISR should clear any interrupt flag or the ISR will be immediately executed again after the reti instruction is processed.

```
void My_Serial_ISR (void) interrupt 4 using 1
{
    char x;
    if (RI) { /*Reception part of ISR comes here*/ }
    if (TI) {
        TI=0;
        if (*tx!=0) {
            tx_busy=1;
            SBUF=*tx;
            tx++;
        }
        else tx_busy=0;
    }
}
```

Q: For example, what is wrong with this code?

A: We must set RI to zero somewhere!

3) Not Clearing The Interrupt Flag

- Your ISR should clear any interrupt flag or the ISR will be immediately executed again after the reti instruction is processed.

```
void My_Serial_ISR (void) interrupt 4 using 1
{
    char x;
    if (RI) {
        RI=0;
        rx_ready=1; //Global-volatile-bit variable
    }
    if (TI) {
        TI=0;
        if (*tx!=0) {
            tx_busy=1;
            SBUF=*tx;
            tx++;
        }
        else tx_busy=0;
    }
}
```

Much better!

4) Not Making Variables "volatile"

```
char foo;
.
.
.
void bar (void)
{
    foo = 0;
    while (foo != 255) ;
}
```

Compiler assumes 'foo' is never changed!

```
char foo;
.
.
.
void bar (void)
{
    foo = 0;
    while(1);
}
```

```
void Inc_foo (void) interrupt 1 using 1
{
    foo++;
}
```

BAD!

4) Not Making Variables "volatile"

```
volatile char foo;
.
.
.
void bar (void)
{
    foo = 0;
    while (foo != 255) ;
}
```

Compiler assumes 'foo' is changed somehow!

```
volatile char foo;
.
.
.
void bar (void)
{
    foo = 0;
    while (foo != 255);
}
```

```
void Inc_foo (void) interrupt 1 using 1
{
    foo++;
}
```

GOOD!

5) Using non-Atomic Variables.

- Q: What is an atomic variable?
- A: variable that can be accessed by a single assembly instruction.
- In the 8051 there are only two possible atomic variables: bit and char.

5) Using non-Atomic Variables.

```
volatile unsigned int foo;
.
.
.
void bar (void)
{
    foo = 0;
    while (foo != 1000) ;
}
```

The problem is way easier to understand when we see the generated assembly code for these functions…

```
void Inc_foo (void) interrupt 1 using 1
{
    foo++;
}
```

5) Using non-Atomic Variables.

```
        rseg R_DSEG
_foo:
    .ds 2

_bar:
        clr     a
        mov     _foo,a
        mov     (_foo + 1),a
L003003?:
        mov     a,#0xE8
        cjne    a,_foo, L003003?
        mov     a,#0x03
        cjne    a,(_foo + 1), L003003?
        ret
```

```
_Inc_foo:
        push    acc
        push    psw
        mov     psw,#0x08
        mov     a,#0x01
        add     a,_foo
        mov     _foo,a
        clr     a
        addc    a,(_foo + 1)
        mov     (_foo + 1),a
        pop     psw
        pop     acc
        reti
```

If the interrupt just happens to occur here, this code will be using a corrupted 'foo'!

5) Using non-Atomic Variables.

- To use non-atomic variables in an ISR we have two options:
  1. Disable interrupts before accessing the variable. (May not be all that practical!)
  2. Use an atomic variable flag to signal changes made to a non-atomic variable. This a job for which a 'bit' variable in the 8051 could excel!
- For bigger microprocessors (32-bit & 64-bit: x86, Coldfire, ARM, etc.) all standard "C" variables are atomic, but the same problem arises when using structures!

5) Using non-Atomic Variables.

```
volatile unsigned int foo;
volatile bit foo_flag;

void Set_foo (unsigned int x)
{
    do {
        foo_flag=0;
        foo=x;
    } while (foo_flag==1);
}

unsigned int Get_foo (void)
{
    volatile int x;
    do {
        foo_flag=0;
        x=foo;
    } while (foo_flag==1);
    return x;
}
```

```
void bar (void)
{
    Set_foo(0);
    while (Get_foo() != 1000) ;
}

void Inc_foo (void) interrupt 1 using 1
{
    foo++;
    foo_flag=1;
}
```

Bit variable ‘foo_flag’ is used to check if the access to ‘foo’ was valid!

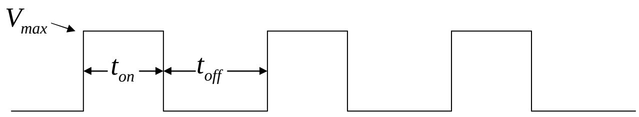
Example: Interrupt Based Timer Delay

```
#include <p89LPC9351.h>
volatile unsigned int msCount;
volatile bit msCount_flag;

#define XTAL 7373000L
//We want timer 1 to interrupt every
//millisecond (1/1000Hz)=1 ms
#define TIMER1_RELOAD_VALUE \
(65536L-((XTAL)/(2*1000L)))

void Set_msCount (unsigned int x) {
    do {
        msCount_flag=0;
        msCount=x;
    } while (msCount_flag);
}

unsigned int Get_msCount (void){
    volatile unsigned int x;
    do {
        msCount_flag=0;
        x=msCount;
    } while (msCount_flag);
    return x;
}
```

```
void WaitXms (unsigned int x) {
    Set_msCount(0);
    while (Get_msCount()< x) ;
}

void SetTimer1 (void) {
    TR1=0; // Stop timer 1
    TMOD=(TMOD&0x0f)|0x10; // 16-bit timer
    TH1=TIMER1_RELOAD_VALUE/0x100;
    TL1=TIMER1_RELOAD_VALUE%0x100;
    TR1=1; // Start timer 1 (bit 6 in TCON)
    ET1=1; // Enable timer 1 interrupt
    EA=1;  // Enable global interrupts
}

void T1ISR (void) interrupt 3 using 2 {
    TR1=0; // Stop timer 1
    TH1=TIMER1_RELOAD_VALUE/0x100;
    TL1=TIMER1_RELOAD_VALUE%0x100;
    TR1=1; // Start timer 1
    msCount++;
    msCount_flag=1;
}

void main (void) { ...
```

Example: Pulse Width Modulation (PWM) Using a Timer Interrupt.

- Pulse Width Modulation is a technique used to create an analog signal from a digital signal by changing the duty cycle of a constant frequency pulse waveform:



$$V_{mean} = V_{max} \times \frac{t_{on}}{t_{on} + t_{off}}$$

Use a low pass filter to get V_{mean}!

PWM

- Advantages of PWM over “analog” outputs:
  – Easy to generate with microcontrollers.
  – Can be easily optically isolated.
  – Operates transistors in cutoff/saturation (BJTs) or cutoff/triode (MOSFETs). For most applications, considerably less power wasted heating-up transistors!

Generating PWM signals with a generic 8051 using interrupts

- Using a timer interrupt to generate PWM:

```
//Timer 2 ISR: Use register bank 1 for this interrupt
void PwmGeneration (void) interrupt 5 using 1
{
   static unsigned char dutcnt;

   dutcnt++;
   if (dutcnt==100) dutcnt=0;
   P2_0=dutcnt<pwm?0:1;
}
```

Prevents variable re-initialization.

This variable must be declared volatile

- Works for any pin!
- In the example above, timer 2 is configured to interrupt every 0.1 ms.

PWM with the C8051F38C …

- This technique has some advantages over hardware generated PWM signals:

  1. You may have as many PWM signals as you need
  2. Frequency is good for small motors and coils
  3. Works with any microcontroller!

Provided example: square.c

Motors for Lab 6

- Any small DC motor will do.
- If you buy the project 2 kit, there are two motors there your team can share.
- It should be obvious how useful this lab is for the project!
- It could be very useful for EECE380 next year as well: ask the current EECE380 students! By the way keep the F38x, it may be used in future courses.