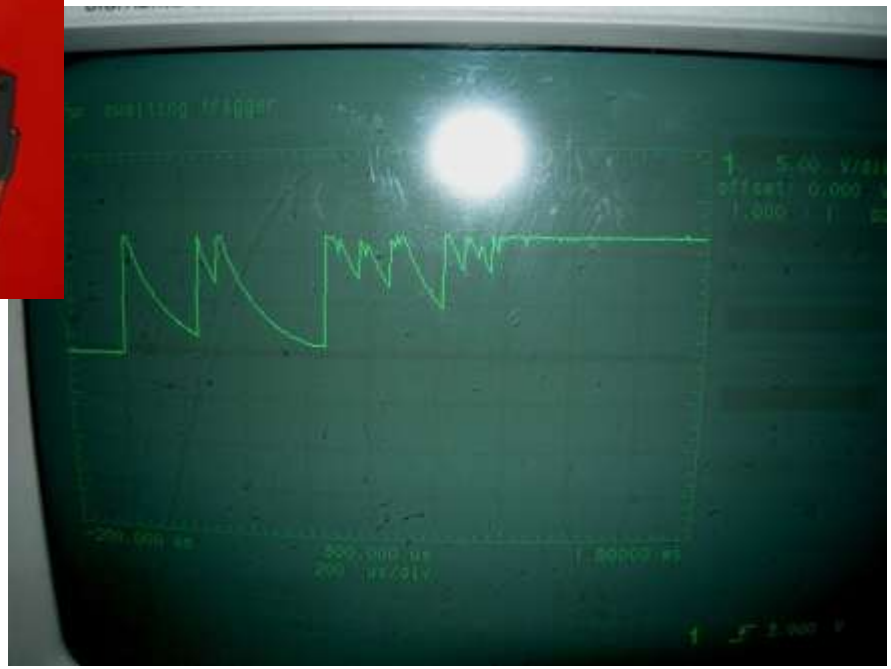
Switches, power savings, concurrency

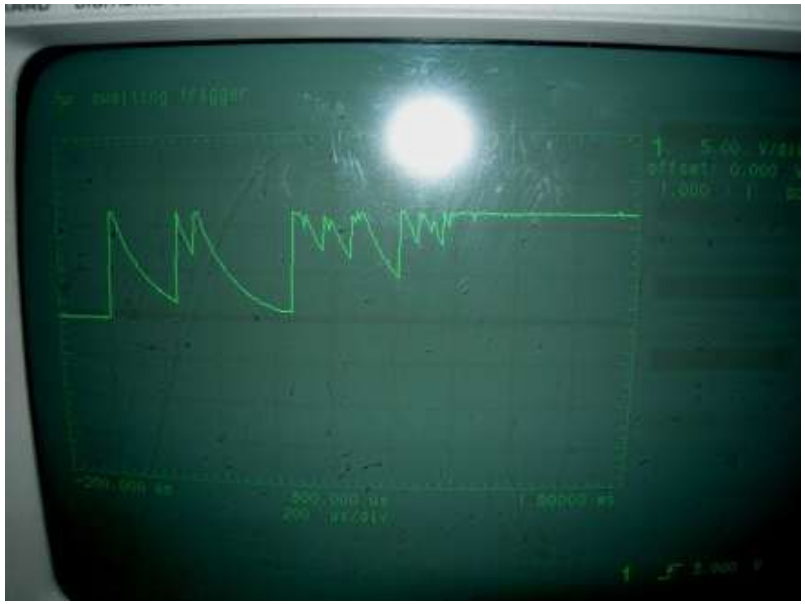


Overview

- Switch bounce
  - Handling physical realities of a switch
  - Get a trigger single event for a single button push
- 8052 power savings modes
- C bit operations
- Concurrency
  - Staying safe with preemptive interrupts

Contact bounce

- Mechanical switches
  - Don't make or break contact cleanly in time scale of a microcontroller
  - Signal may bouncing between high and low several times



Button counter: take 1

- Goal: Increment binary counter on P0 whenever P2.0 is pressed
  - First attempt:
    - If button is down (bit is 0)
      - Wait for button to be released (bit goes to 1)
      - Return 1
    - Else return 0

Button counter: take 1

```
#include <REG52.H>

sbit BUTTON0 = P1^0;
sbit BUTTON1 = P1^1;
sbit BUTTON2 = P1^2;
sbit BUTTON3 = P1^3;

bit getButton0()
{
   if (!BUTTON0)
   {
      while (!BUTTON0)
         ;
      return 1;
   }
   return 0;
}

void main()
{
   unsigned char count = 1;
   while (1)
   {
      P0 = ~count;
      if (getButton0())
         count++;
   }
}
```

Only use this ^ notation outside functions, otherwise you'll get C XOR!

Loop that spins until button pin goes high

Stick to one byte variables unless you actually need more.

Problem: counter occasionally seems to skip through several values!

Button counter: take 2

- Goal: Increment binary counter on P0 whenever P2.0 is pressed
  - Second attempt:
    - If button is down (bit is 0)
      - while button not released
        - » Increment counter
      - if counter too small return 0 otherwise return 1
    - Else return 0

Button counter: take 2

```
...

const unsigned int DEBOUNCE = 6600;    // Takes around 1/20 of a second

bit getButton0()
{
   unsigned int i = 0;
   if (!BUTTON0)
   {
      while (!BUTTON0)
         i++;
      if (i < DEBOUNCE)
         return 0;
      return 1;
   }
   return 0;
}

void main()
{
   unsigned char count = 1;
   while (1)
   {
      P0 = ~count;
      if (getButton0())
         count++;
   }
}
```

Doesn't count as "real" button press unless we spent enough time continuously at low signal.

Problem 1: Counter can overflow leading to a failure to detect long press.

Problem 2: Microcontroller expending a lot of power spinning around.

Power saving modes

- 8052 has two power saving modes:
  - Idle mode
    - Program execution stops
    - Timers, serial port, interrupts remain active
    - Consumes 1/3 - 1/4 normal current
    - Sleeps until any enabled interrupt occurs
  - Power-down mode
    - Power down most aspects of microcontroller
    - Only wakes up on external interrupt, low level only
    - Consumes 1/500 normal current

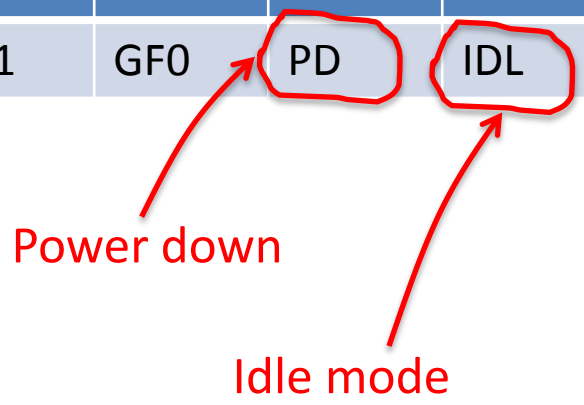
Enabling power saving

- Set one of two bits in PCON SFR:

PCON	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	SMON	-	-	-	GF1	GF0	PD	IDL

```
PCON = 1;    // Go to idle mode

PCON = 2;    // Power down
```



Button counter: take 3

```
// Fire up timer0 in 16-bit reload mode with
// interrupts enabled.  Set up for 0.01s delay.
void startTimer0()
{
   TMOD = TMOD & 252; // Mask out lowest 2 bits
   TMOD++;            // Set T0M1/T0M0 to 01
   TH0  = 219;        // Setup for ~0.01 delay
   TR0  = 1;          // Start timer0 running
   ET0  = 1;          // Enable timer0 interrupt
   EA   = 1;          // Global interrupt enable
}

// Timer0 ISR, provides a 0.01s heartbeat
void timer0ISR() interrupt 1
{
   TH0  = 219;
}

// Go to sleep for 0.01s
void sleep()
{
   TH0  = 219;
   PCON = 1;
}
```

```
// Returns 1 if P2.0 is pushed
// Returns 0 otherwise.
bit getButton0()
{
   unsigned int i = 0;
   if (!BUTTON0)
   {
      sleep();
      if (BUTTON0)
         return 0;
      while (!BUTTON0)
         PCON = 1;
      return 1;
   }
   return 0;
}
void main()
{
   unsigned char count = 1;
   startTimer0();
   while (1)
   {
      sleep();
      P0 = ~count;
      if (getButton0())
         count++;
   }
}
```

Bitwise operators in C

- Using unsigned char data types:

AND

		0	1	1	0	1	1	0	0
	&	1	0	1	1	0	1	0	1
		0	0	1	0	0	1	0	0

108 & 181 = 36

OR

		0	1	1	0	1	1	0	0
	|	1	0	1	1	0	1	0	1
		1	1	1	1	1	1	0	1

108 | 181 = 253

NOT

	~	1	0	1	1	0	1	0	1
		0	1	0	0	1	0	1	0

~181 = 74

XOR

		0	1	1	0	1	1	0	0
	^	1	0	1	1	0	1	0	1
		1	1	0	1	1	0	0	1

108 ^ 181 = 217

Why bit-whack?

- Often used for bit masking
- Example:
  - One byte used to store 8 on/off user settings

```
const unsigned char AUTO_SAVE        = 1;
const unsigned char CONFIRM_EXIT     = 2;
const unsigned char SHOW_TOOLBAR     = 4;
const unsigned char STARTUP_TIP      = 8;
const unsigned char COMPRESS_FILES   = 16;
const unsigned char ENCRYPT_FILES    = 32;
const unsigned char PASSWORD_PROTECT = 64;
const unsigned char TRACK_CHANGES    = 128;
```

```
0000 0001 = 1
0000 0010 = 2
0000 0100 = 4
0000 1000 = 8
0001 0000 = 16
0010 0000 = 32
0100 0000 = 64
1000 0000 = 128
```

Why bit-whack?

```
const unsigned char AUTO_SAVE          = 1;
const unsigned char CONFIRM_EXIT       = 2;
const unsigned char SHOW_TOOLBAR       = 4;
const unsigned char STARTUP_TIP        = 8;
const unsigned char COMPRESS_FILES     = 16;
const unsigned char ENCRYPT_FILES      = 32;
const unsigned char PASSWORD_PROTECT   = 64;
const unsigned char TRACK_CHANGES      = 128;
```

```
unsigned char setting = AUTO_SAVE | COMPRESS_FILES | TRACK_CHANGES;
```

Start with three of the settings enabled.

```
setting = setting | SHOW_TOOLBAR;
```

Enable the toolbar if it isn't already enabled.

```
if (setting & TRACK_CHANGES) { /* do something */ }
```

Do something if track changes is turned on.

```
setting ^= SHOW_TOOLBAR;
```

Toggle the show toolbar setting.

Left bit shifting

- Left shift, **x << y**
  - Bits in x move y positions left
  - Zeros stuffed on right side (least significant bit)
  - Every move left doubles values
  - Left shift of value 3:

```
0000 0000 0000 0011 << 1
0000 0000 0000 0110          = 6
```

```
0000 0000 0000 0011 << 2
0000 0000 0000 1100          = 12
```

```
0000 0000 0000 0011 << 3
0000 0000 0001 1000          = 24
```

Right bit shifting

- Right shift, **x >> y**
  - Bits in x move y positions right
  - Zeros stuffed on left side (most significant bit)
  - Every move right halves value (integer div by 2)
  - Right shift of value 35:

```
0000 0000 0010 0011 >> 1
0000 0000 0001 0001          = 17
```

```
0000 0000 0010 0011 >> 2
0000 0000 0000 1000          = 8
```

```
0000 0000 0010 0011 >> 3
0000 0000 0000 0100          = 4
```

Bit masking

- Bit-whacking to obtain part of number
- Example:
  - 16-bit number, get least or most significant nibble

```
unsigned short number     = 12345;
unsigned short lowNibble  = (unsigned short)  (number & 0x000F);
unsigned short highNibble = (unsigned short) ((number & 0xF000) >> 12);
```

```
  0011 0000 0011 1001 = 12345
& 0000 0000 0000 1111 = 0x000F
  0000 0000 0000 1001 = 9
```

lowNibble

```
  0011 0000 0011 1001 = 12345
& 1111 0000 0000 0000 = 0xF000
  0011 0000 0000 0000 = 12288

  0011 0000 0000 0000 >> 12
  0000 0000 0000 0011 = 3
```

highNibble

Concurrency

- Cooperative multitasking
  - Processes explicitly programmed to yield when they don't need the system
- Preemptive multitasking
  - Interrupt mechanism suspends currently executing process
  - New process is selected for execution
    - Based on some criteria, e.g. priority, round-robin
  - How the 8052 does it

LED cycler

- Goal: Cycle one LED left-to-right
  - Cycling happens every time timer1 overflows
    - Overflow triggers interrupt service routine
  - Main program
    - Monitors for push of P2.0 button
    - After P2.0 released, extra shift of LED

LED cycler: failure

```
sbit BUTTON0 = P2^0;
unsigned char lights = 1;

// Fire up timer1 and enable interrupt
void startTimer1()
{
   TMOD = TMOD & 207; // Mask out T1M1/T1M0
   TMOD = TMOD + 16;  // Set T1M1/T1M0 to 01
   TR1  = 1;          // Start timer1 running
   ET1  = 1;          // Enable interrupt
   EA   = 1;          // Global enable
}

// Timer1 ISR - moves LED to the left
void timer1ISR() interrupt 3
{
   // Check if time to roll around
   if (lights == 16)
      lights = 1;
   else
      lights = lights << 1;
   P0 = ~lights;
}
```

```
void main()
{
   unsigned int i = 0;
   startTimer1();

   while (1)
   {
      if (!BUTTON0)
      {
         // Wait for button release
         while (!BUTTON0)
            ;
         if (lights == 16)
            lights = 1;
         else
         {
            for (i = 0; i < 1000; i++)
               ;
            lights = lights << 1;
         }
         P0 = ~lights;
      }
   }
}
```

Added to make bug easier to notice.

Critical section

- Critical section
  - Section of code that accesses a shared resource
  - Resource should not be concurrently accessed by more than one thread of execution

```
void main()
{
   unsigned int i = 0;
   startTimer1();

   while (1)
   {
      if (!BUTTON0)
      {
         // Wait for button release
         while (!BUTTON0)
            ;
         if (lights == 16)
            lights = 1;
         else
         {
             for (i = 0; i < 1000; i++)
                    ;
            lights = lights << 1;
         }
         P0 = ~lights;
      }
   }
}
```

LED cycler: success

```
sbit BUTTON0 = P2^0;
unsigned char lights = 1;

// Fire up timer1 and enable interrupt
void startTimer1()
{
   TMOD = TMOD & 207; // Mask out T1M1/T1M0
   TMOD = TMOD + 16;  // Set T1M1/T1M0 to 01
   TR1  = 1;          // Start timer1 running
   ET1  = 1;          // Enable interrupt
   EA   = 1;          // Global enable
}

// Timer1 ISR - moves LED to the left
void timer1ISR() interrupt 3
{
   // Check if time to roll around
   if (lights == 16)
      lights = 1;
   else
      lights = lights << 1;
   P0 = ~lights;
}
```

```
void main()
{
   unsigned int i = 0;
   startTimer1();

   while (1)
   {
      if (!BUTTON0)
      {
         // Wait for button release
         while (!BUTTON0)
            ;
         EA = 0;
         if (lights == 16)
            lights = 1;
         else
         {
             for (i = 0; i < 1000; i++)
                    ;
            lights = lights << 1;
         }
         EA = 1;
         P0 = ~lights;
      }
   }
}
```

Turn off interrupts

Turn on interrupts

Summary

- Learned how to:
  - Handle switch bounce
  - Use power saving features
    - Also can be used to delay during switch debounce
  - Use bit operations in C
  - Protect critical code sections
    - If ISR and main program use shared variables, temporarily disable interrupts