MSP430 Intro
Why embedded systems?

• Big bang-for-the-buck by adding some intelligence to systems.
• Embedded Systems are ubiquitous.
• Embedded Systems more common as prices drop, and power decreases.
Which Embedded System?

• We will use Texas Instruments MSP-430
  + TI has large market share
  + 16 bits (instead of 8)
  + low power
  + clean architecture
  + low cost (free) development tools
  - relatively low speed/capacity (i.e., no video or fancy audio)
  - low level tools (compared to Arduino...)
  - 16 bits (instead of 32)
This lecture

• Brief overview of
  o logic, true (logical 1, 3.3V) or false (logical 0, 0V).
  o numbers
  o C
  o MSP430 digital I/O
Number Systems

**Binary:** \(00001101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13\)

**Hex:** \(00101010_2 = 2A_{16} = 0x2A = 2 \cdot 16^1 + 10 \cdot 16^0 = 32 + 10 = 42\)
(check \(1 \cdot 2^5 + 1 \cdot 2^3 + 2 \cdot 2^1 = 32 + 8 + 2 = 42\))

8 bits = 1 byte

- \(0000\ 0000_2 \rightarrow 1111\ 1111_2\)
- \(0x00 \rightarrow 0xff\)
- \(0 \rightarrow 2^8-1=255\) (or -128 \(\rightarrow\) 127 (\(-(2^7) \rightarrow 2^7-1)\))

16 bits = 2 bytes = 1 word

- \(0000\ 0000\ 0000\ 0000_2 \rightarrow 1111\ 1111\ 1111\ 1111_2\)
- \(0x0000 \rightarrow 0xffff\)
- \(0 \rightarrow 2^{16}-1=65535\) (or -32768 \(\rightarrow\) 32767 (\(-(2^{15}) \rightarrow 2^{15}-1)\))

4 bits = 1 nybble  \(\ (0 \rightarrow 2^4-1=15)\)
## C Data Types

### (that we will use)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bits)</th>
<th>Representation</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char</td>
<td>8</td>
<td>ASCII</td>
<td>-128</td>
<td>+127</td>
</tr>
<tr>
<td>unsigned char, bool</td>
<td>8</td>
<td>ASCII</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>int, signed int</td>
<td>16</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16</td>
<td>Binary</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>long, signed long</td>
<td>32</td>
<td>2s complement</td>
<td>-2 147 483 648</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>Binary</td>
<td>0</td>
<td>4 294 967 295</td>
</tr>
<tr>
<td>enum</td>
<td>16</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>IEEE 32-bit</td>
<td>±1.175 495e-38</td>
<td>±3.40 282 35e+38</td>
</tr>
</tbody>
</table>
## C Operators

### (Arithmetic)

<table>
<thead>
<tr>
<th>Arithmetic Operator name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic assignment</td>
<td><code>a = b</code></td>
</tr>
<tr>
<td>Addition</td>
<td><code>a + b</code></td>
</tr>
<tr>
<td>Subtraction</td>
<td><code>a - b</code></td>
</tr>
<tr>
<td>Unary plus</td>
<td><code>+a</code></td>
</tr>
<tr>
<td>Unary minus (additive inverse)</td>
<td><code>-a</code></td>
</tr>
<tr>
<td>Multiplication</td>
<td><code>a * b</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
</tr>
<tr>
<td>Modulo (remainder)</td>
<td><code>a % b</code></td>
</tr>
</tbody>
</table>

#### Increment
- **Prefix**
  - `++a`
- **Suffix**
  - `a++`

#### Decrement
- **Prefix**
  - `--a`
- **Suffix**
  - `a--`
## More C Operators

### (Relational, Logical, Bitwise and Compound)

<table>
<thead>
<tr>
<th>Relational Operator name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td>a == b</td>
</tr>
<tr>
<td>Not equal to</td>
<td>a != b</td>
</tr>
<tr>
<td>Greater than</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>Less than</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>a &lt;= b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical Operator name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical negation (NOT)</td>
<td>!a</td>
</tr>
<tr>
<td>Logical AND</td>
<td>a &amp;&amp; b</td>
</tr>
<tr>
<td>Logical OR</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bitwise Operator name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise NOT</td>
<td>~a</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>a</td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>a ^ b</td>
</tr>
<tr>
<td>Bitwise left shift</td>
<td>a &lt;&lt; b</td>
</tr>
<tr>
<td>Bitwise right shift</td>
<td>a &gt;&gt; b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compound Operator name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition assignment</td>
<td>a += b</td>
</tr>
<tr>
<td>Subtraction assignment</td>
<td>a -= b</td>
</tr>
<tr>
<td>Multiplication assignment</td>
<td>a *= b</td>
</tr>
<tr>
<td>Division assignment</td>
<td>a /= b</td>
</tr>
<tr>
<td>Modulo assignment</td>
<td>a %= b</td>
</tr>
<tr>
<td>Bitwise AND assignment</td>
<td>a &amp;</td>
</tr>
<tr>
<td>Bitwise OR assignment</td>
<td>a</td>
</tr>
<tr>
<td>Bitwise XOR assignment</td>
<td>a ^= b</td>
</tr>
<tr>
<td>Bitwise left shift assignment</td>
<td>a &lt;&lt;= b</td>
</tr>
<tr>
<td>Bitwise right shift assignment</td>
<td>a &gt;&gt;= b</td>
</tr>
</tbody>
</table>
Manipulating bits (1)

All variables are “int”, though we’ll only use 8 bits

```
x = 0x33;   // 0011 0011
y = 0x5a;  // 0101 1010
```

```
z = y & x;  // (and) 0001 0010   hex=0x12

z = y | x;  // (or)  0111 1011   hex=0x7b

z = y ^ x;  // (xor) 0110 1001   hex=0x69

z = ~x;     // (not) 1100 1100   hex=0xcc
```

Remember: Use “&,” “|,” “^,” “~” for bitwise operations.
Use “&&,” “||,” “!” for logical operations
Manipulating Bits in C (2)

MSP430 has some built in constants for manipulating bits

<table>
<thead>
<tr>
<th>Hex</th>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT0 = 0x01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BIT1 = 0x02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BIT2 = 0x04</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BIT3 = 0x08</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BIT4 = 0x10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BIT5 = 0x20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BIT6 = 0x40</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BIT7 = 0x80</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Setting bits**

\[
x = \text{BIT6} \mid \text{BIT3} \mid \text{BIT0}; // 0100 1001 = 0x49
\]

\[
x = x \mid \text{BIT4}; // 0101 1001 = 0x59
\]

**Clearing bits**  (assume x = 0101 1001 = 0x59)

\[
y = x \& \sim\text{BIT3}; // 0101 0001 = 0x51 (bit 3 is cleared)
\]

\[
y = x \& \sim(\text{BIT3} \mid \text{BIT4}); // 0100 0001 = 0x41
\]
Some C shorthand

• There are some C constructs that can be convenient.
  – increment by one: “x++” is equivalent to “x = x+1”
  – decrement by one: “x--” is equivalent to “x = x-1”
  – Perform operation on variable and reassign to same variable.
    • “x += 3” is equivalent to “x = x+3”
    • “x *= y” is equivalent to “x = x*y”
    • “x |= BIT3” is equivalent to “x = x|BIT3” (this sets bit 3)
    • …
Logical vs. Bitwise

• When dealing with “logical” quantities, anything that is not zero is interpreted as true.

• Let $x=0x03$ (0011), $y=0x08$ (1000)
  – if ($x$)       // “$x$” is interpreted as true, and the
                   statement; // statement is executed.
  – if ($x$ && $y$)... // “$x$ && $y$” is interpreted as true
  – if ($x$ & $y$)... // “$x$ & $y$” is interpreted as false (zero)
  – if (!$x$)...     // “!$x$” is interpreted as false
  – if ($\sim x$)... // “$\sim x$” is interpreted as true (not zero)
  – if ($x==$y)...   // “$x==$y” is interpreted as false
  – if ($x=$y)...    // “$x=$y” assigns the $x$ the value of 8 and
                     // is interpreted as true
Basic Architecture of MSP430 (from Family User’s Guide)
MSP430G2533
(from device specific datasheet)

Functional Block Diagram, MSP430G2x53

NOTE: Port P3 is available on 28-pin and 32-pin devices only.
MSP430FG2533, 20 pin DIP
(from datasheet)
Device Pinout, MSP430G2x13 and MSP430G2x53, 20-Pin Devices, TSSOP and PDIP
8.3 Digital I/O Registers

The digital I/O registers are listed in Table 8-2.

<table>
<thead>
<tr>
<th>Port</th>
<th>Register</th>
<th>Short Form</th>
<th>Address</th>
<th>Register Type</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Input</td>
<td>P1IN</td>
<td>020h</td>
<td>Read only</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>P1OUT</td>
<td>021h</td>
<td>Read/write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>P1DIR</td>
<td>022h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Flag</td>
<td>P1IFG</td>
<td>023h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Edge Select</td>
<td>P1IES</td>
<td>024h</td>
<td>Read/write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Interrupt Enable</td>
<td>P1E</td>
<td>025h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select</td>
<td>P1SEL</td>
<td>026h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select 2</td>
<td>P1SEL2</td>
<td>041h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Resistor Enable</td>
<td>P1REN</td>
<td>027h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td>P2</td>
<td>Input</td>
<td>P2IN</td>
<td>028h</td>
<td>Read only</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>P2OUT</td>
<td>029h</td>
<td>Read/write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>P2DIR</td>
<td>02Ah</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Flag</td>
<td>P2IFG</td>
<td>02Bh</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Interrupt Edge Select</td>
<td>P2IES</td>
<td>02Ch</td>
<td>Read/write</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Interrupt Enable</td>
<td>P2E</td>
<td>02Dh</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select</td>
<td>P2SEL</td>
<td>02 Eh</td>
<td>Read/write</td>
<td>0C0h with PUC</td>
</tr>
<tr>
<td></td>
<td>Port Select 2</td>
<td>P2SEL2</td>
<td>042h</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
<tr>
<td></td>
<td>Resistor Enable</td>
<td>P2REN</td>
<td>02Fh</td>
<td>Read/write</td>
<td>Reset with PUC</td>
</tr>
</tbody>
</table>
8.2.1 *Input Register PxlN*

Each bit in each PxlN register reflects the value of the input signal at the corresponding I/O pin when the pin is configured as I/O function.
- Bit = 0: The input is low
- Bit = 1: The input is high

---

**NOTE: Writing to Read-Only Registers PxlN**

Writing to these read-only registers results in increased current consumption while the write attempt is active.

---

8.2.2 *Output Registers PxlOUT*

Each bit in each PxlOUT register is the value to be output on the corresponding I/O pin when the pin is configured as I/O function, output direction, and the pullup/down resistor is disabled.
- Bit = 0: The output is low
- Bit = 1: The output is high

If the pin's pullup/pulldown resistor is enabled, the corresponding bit in the PxlOUT register selects pullup or pulldown.
- Bit = 0: The pin is pulled down
- Bit = 1: The pin is pulled up
A typical I/O pin

- Analog switch
- Inverted input
- Electronically controlled digital switch
- 2-1 Mux (multiplexer)
- 4-1 Mux (multiplexer)
- Tri-State Output
- Schmitt trigger (hysteresis)
- Active low Transparent latch
Effect of P1DIR

P1SEL, and P1SEL2 = 0 for I/O

Also – P1REN
A simple C program

```c
#include <msp430.h>

void main(void) {
    volatile int i;

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= 0x01;            // Set P1.0 to output direction

    while (1) {  //Do this forever
        P1OUT = P1OUT | 0x01;         // Set P1.0 with "or", |
        for (i=0; i<0x5000; i++) {} // Delay
        P1OUT = P1OUT & ~0x01;  // Clear P1.0
        for (i=0; i<0x5000; i++) {} // Delay
    }
}
```

Constants associated with our chip
Every program needs a “main” routine (between braces)
Declare “i” as volatile so compiler doesn’t optimize it out of existence (or turn optimizations off).
All variables must be declare before they are used.

Don’t worry about for now.

Set bit 0 high (connected to LED)
Loop to waste time
Set bit 0 low (LED turns off)
Loop to waste time

Set bit 0 in “P1DIR” - this makes it an output (next page).

Comments start with “//” and go to end of line.
Also note that every statement ends with “;” or “}”
#include <msp430.h>
#define LED 0x01

void main(void) {
volatile int i;
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= LED; // Set P1.0 (LED bit) to output

    while (1) { // Do this forever
        P1OUT |= LED; // Turn on LED
        for (i=0; i<0x5000; i++) {} // Delay
        P1OUT &= ~LED; // Turn off LED
        for (i=0; i<0x5000; i++) {} // Delay
    }
}
Variant 2 (macros)

```c
#include <msp430.h>
#define LED BIT0
#define SETBIT(p,b) (p |= (b))
#define CLRBIT(p,b) (p &= ~(b))

void main(void) {
    volatile int i;
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= LED;            // Set P1.0 to output direction

    while (1) {  //Do this forever
        SETBIT(P1OUT,LED);       // Set P1.0
        for (i=0; i<0x5000; i++) {} // Delay
        CLRBIT(P1OUT,LED);       // Clear P1.0
        for (i=0; i<0x5000; i++) {} // Delay
    }
}
```

- Use Macros sparingly, but they can make code look much cleaner (see below)
- Can call bits by location

Expands to: \((P1OUT |= (0x01))\)
Note “;” must be added.
Variant 3 (shorter)

```c
#include <msp430.h>
#define LED BIT0

void main(void) {
    volatile int i;
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= LED;            // Set P1.0 to output direction

    while (1) {             // Do this forever
        P1OUT ^= LED;        // Toggle LED
        for (i=0; i<0x5000; i++) {} // Delay
    }
}
```
More C

Statements

• a simple statement is a single statement that ends in a “;”
• a compound statement is several statements inside braces:

```c
{  
simple statement;  
...
  
simple statement;  
}
```

Indenting

There are no rules about indenting code, but if you don’t adopt a standard style, your code becomes unreadable. Development system will do this for you.

```c
while (x == y) {  
something();  
somethingelse();  
if (some_error)  
  do_correct();  
else  
  continue_as_usual();  
}
```

```c
while (x == y) {  
something();  
somethingelse();  
}
```

```c
if (x < 0)  
{  
  printf("Negative");  
  negative(x);  
}  
else  
{  
  printf("Positive");  
  positive(x);  
}
```
Array definition
int a [100];      //Array elements are a[0] to a[99].  Don’t use a[100]!

if...then
if (<expression>)
  <statement>
<statement> may be a compound statement.

if...then...else
if (<expression>)
  <statement1>
else
  <statement2>

if...then...else (shorthand)
x = (y > 2) ? 3 : 4;    // if y>2, then x=3, else x=4.
Yet more C

Iteration (do...while  while...  for...)

```c
do
    <statement>
while ( <expression> );
```

```c
while ( <expression> )
    <statement>
```

```c
for ( <expression> ; <expression> ; <expression> )
    <statement>
```

Recall:  for (i=0; i<0x5000; i++) {} // Delay

```c
for (e1; e2; e3)
    s;
```

is equivalent to

```c
e1;
while (e2) {
    s;
    e3;
}
```

The break statement is used to end a for loop, while loop, do loop, or switch statement. Control passes to the statement following the terminated statement.
Again with the C

switch (one choice of many)

```
switch (<expression>) {
    case <label1> :
        <statements 1>
    case <label2> :
        <statements 2>
            break;
    default :
        <statements 3>
}
```

- `<expression>` is compared against the label, and execution of the associated statements occur (i.e., if `<expression>` is equal to `<label1>`, `<statements 1>` are executed.
- No two of the case constants may have the same value.
- There may be at most one default label.
- If none of the case labels are equal to the expression in the parentheses following `switch`, control passes to the default label, or if there is no default label, execution resumes just beyond the entire construct.
- Switch statements can "fall through", that is, when one case section has completed its execution, statements will continue to be executed downward until a `break;` statement is encountered. This is usually not wanted, so be careful
Material taken from:

- http://www.ti.com/lit/ug/slau144i/slau144i.pdf (Family User’s Guide; 658 pages)
- http://www.ti.com/lit/ds/symlink/msp430g2553.pdf (Datasheet; 70 pages)