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
  - logic, true (logical 1, 3.3V) or false (logical 0, 0V).
  - numbers
  - C
  - MSP430 digital I/O

Number Systems

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

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 → 0xff
$0 \rightarrow 2^8-1=255 \left(\text{or } -128 \rightarrow 127 \ \left(-(2^7) \rightarrow 2^7-1\right)\right)$

16 bits = 2 bytes = 1 word

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

4 bits = 1 nybble $(0 \rightarrow 2^4-1=15)$

C Data Types
(that we will use)

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

C Operators
(Arithmetic)

Arithmetic Operator name		Syntax
Basic assignment		a = b
Addition		a + b
Subtraction		a - b
Unary plus		+a
Unary minus (additive inverse)		-a
Multiplication		a * b
Division		a / b
Modulo (remainder)		a % b
Increment	Prefix	++a
	Suffix	a++
Decrement	Prefix	--a
	Suffix	a--

More C Operators
(Relational, Logical, Bitwise and Compound)

Relational Operator name	Syntax
Equal to	a ==b
Not equal to	a !=b
Greater than	a > b
Less than	a < b
Greater than or equal to	a >=b
Less than or equal to	a <=b

Bitwise Operator name	Syntax
Bitwise NOT	~a
Bitwise AND	a & b
Bitwise OR	a | b
Bitwise XOR	a ^ b
Bitwise left shift	a <<b
Bitwise right shift	a >>b

Logical Operator name	Syntax
Logical negation (NOT)	!a
Logical AND	a &&b
Logical OR	a ||b

Compound Operator name	Syntax
Addition assignment	a += b
Subtraction assignment	a -= b
Multiplication assignment	a *= b
Division assignment	a /= b
Modulo assignment	a %= b
Bitwise AND assignment	a &= b
Bitwise OR assignment	a |= b
Bitwise XOR assignment	a ^= b
Bitwise left shift assignment	a <<=b
Bitwise right shift assignment	a >>=b

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
```

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Remember: Use "&, |, ^, ~" for bitwise operations.
Use "&&, ||, !" for logical operations

Manipulating Bits in C (2)

MSP430 has some built in constants for manipulating bits

Hex	Bit 7	6	5	4	3	2	1	Bit 0
BIT0 = 0x01	0	0	0	0	0	0	0	1
BIT1 = 0x02	0	0	0	0	0	0	1	0
BIT2 = 0x04	0	0	0	0	0	1	0	0
BIT3 = 0x08	0	0	0	0	1	0	0	0
BIT4 = 0x10	0	0	0	1	0	0	0	0
BIT5 = 0x20	0	0	1	0	0	0	0	0
BIT6 = 0x40	0	1	0	0	0	0	0	0
BIT7 = 0x80	1	0	0	0	0	0	0	0

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Setting bits

```
x = BIT6 | BIT3 | BIT0; // 0100 1001 = 0x49
x = x | BIT4;           // 0101 1001 = 0x59
```

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

Note: ~BIT3 = 1111 0111_{binary} = $F7_{hex}$

```
y = x & ~BIT3; // 0101 0001 = 0x51  (bit 3 is cleared)
y = x & ~(BIT3 | 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

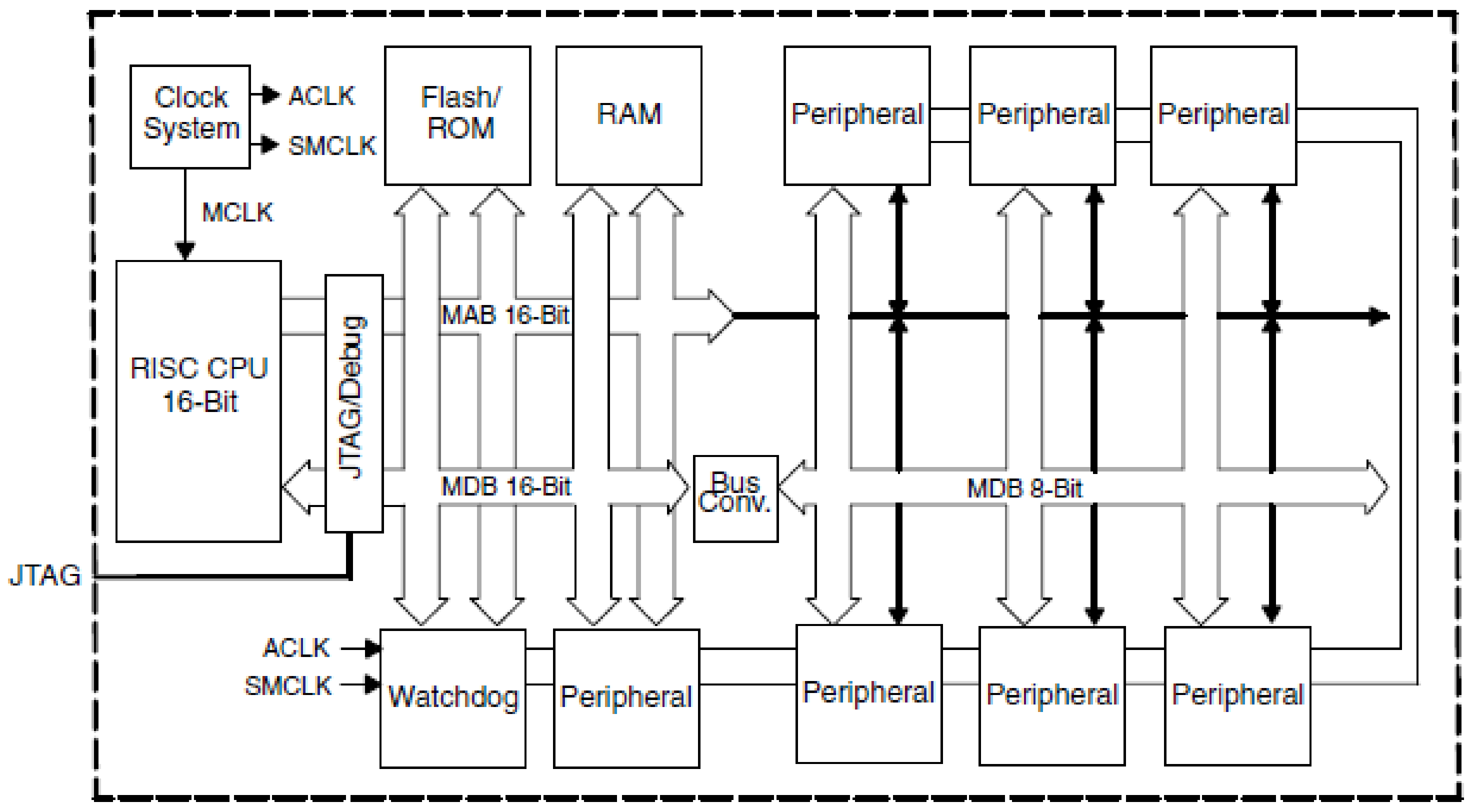
- When dealing with "logical" quantities, anything that is not zero is interpreted as true.
- Let x=0x03 (0011), y=0x08 (1000)

```
– 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 (~x)…          // "~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
```

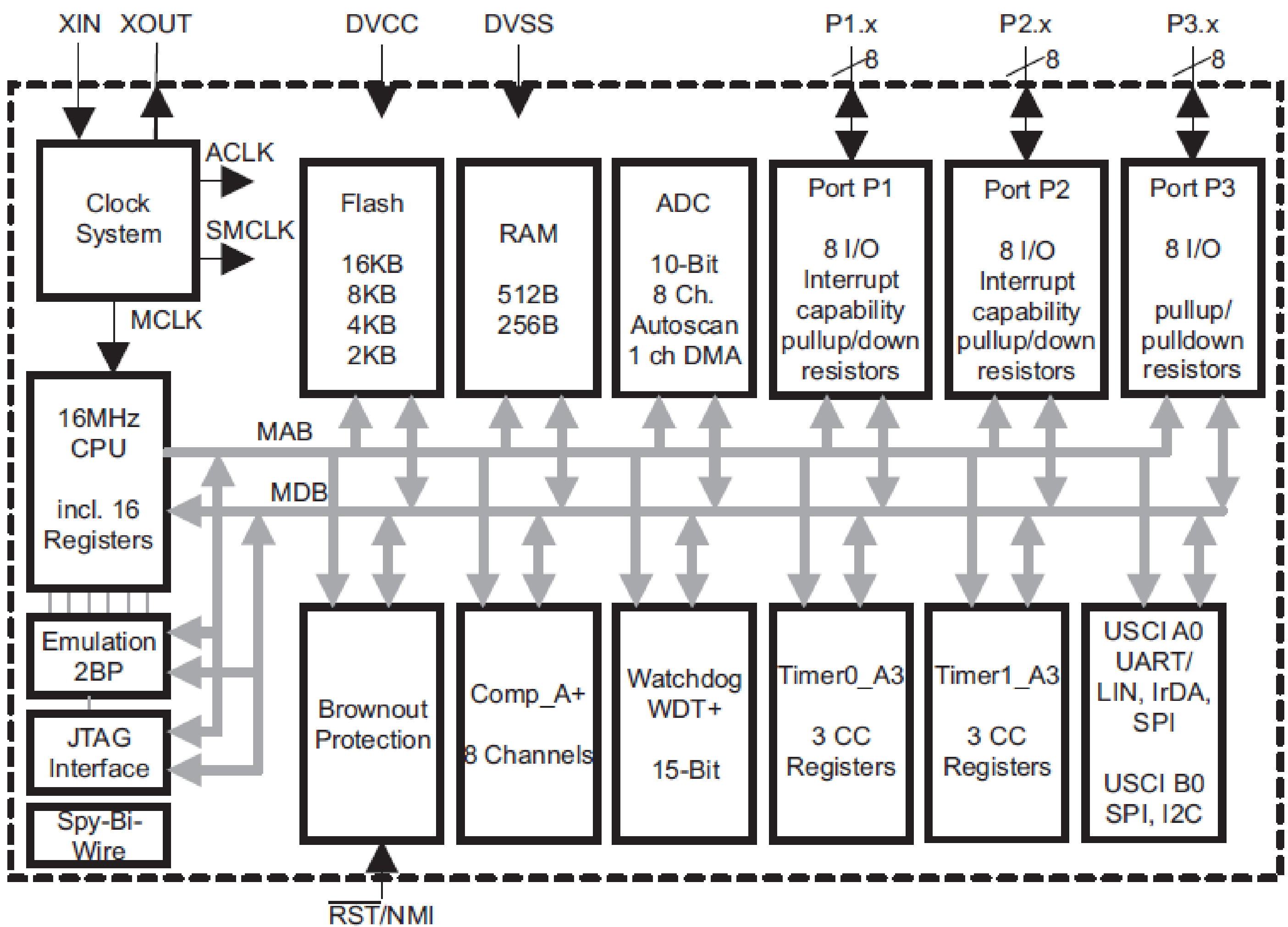
Basic Architecture of MSP430 (from Family User's Guide)

Figure 1–1. MSP430 Architecture



MSP430G2533
(from device specific datasheet)

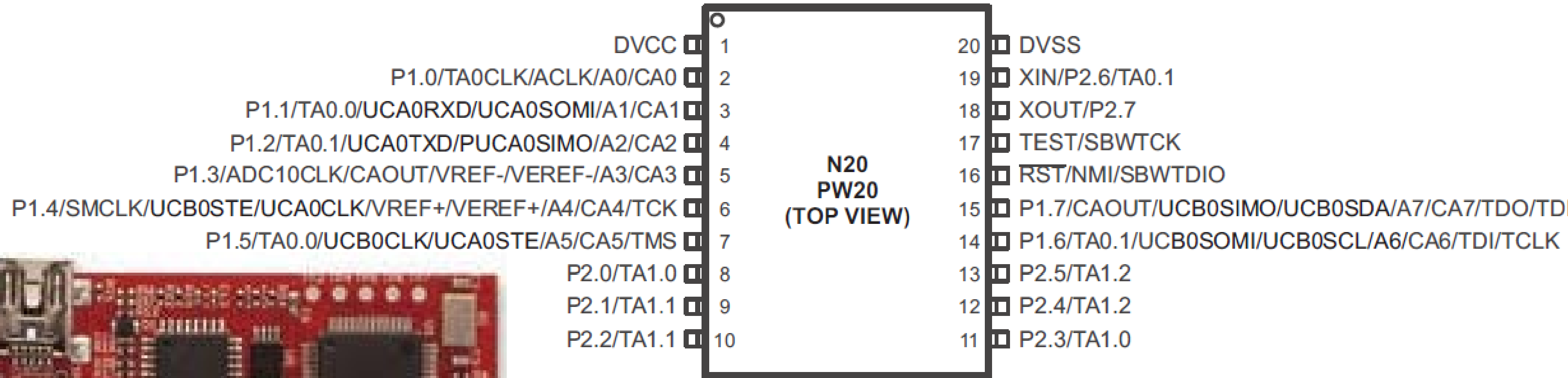
Functional Block Diagram, MSP430G2x53



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



Digital I/O
(Family User's Guide)

8.3 Digital I/O Registers

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

Table 8-2. Digital I/O Registers

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	-
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC
P2	Input	P2IN	028h	Read only	-
	Output	P2OUT	029h	Read/write	Unchanged
	Direction	P2DIR	02Ah	Read/write	Reset with PUC
	Interrupt Flag	P2IFG	02Bh	Read/write	Reset with PUC
	Interrupt Edge Select	P2IES	02Ch	Read/write	Unchanged
	Interrupt Enable	P2IE	02Dh	Read/write	Reset with PUC
	Port Select	P2SEL	02Eh	Read/write	0C0h with PUC
	Port Select 2	P2SEL2	042h	Read/write	Reset with PUC
	Resistor Enable	P2REN	02Fh	Read/write	Reset with PUC

Some register functions
(Family User's Guide)

8.2.1 Input Register PxIN

Each bit in each PxIN register reflects the value of the input signal at the corresponding I/O pin when th
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 PxIN

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

8.2.2 Output Registers PxOUT

Each bit in each PxOUT 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

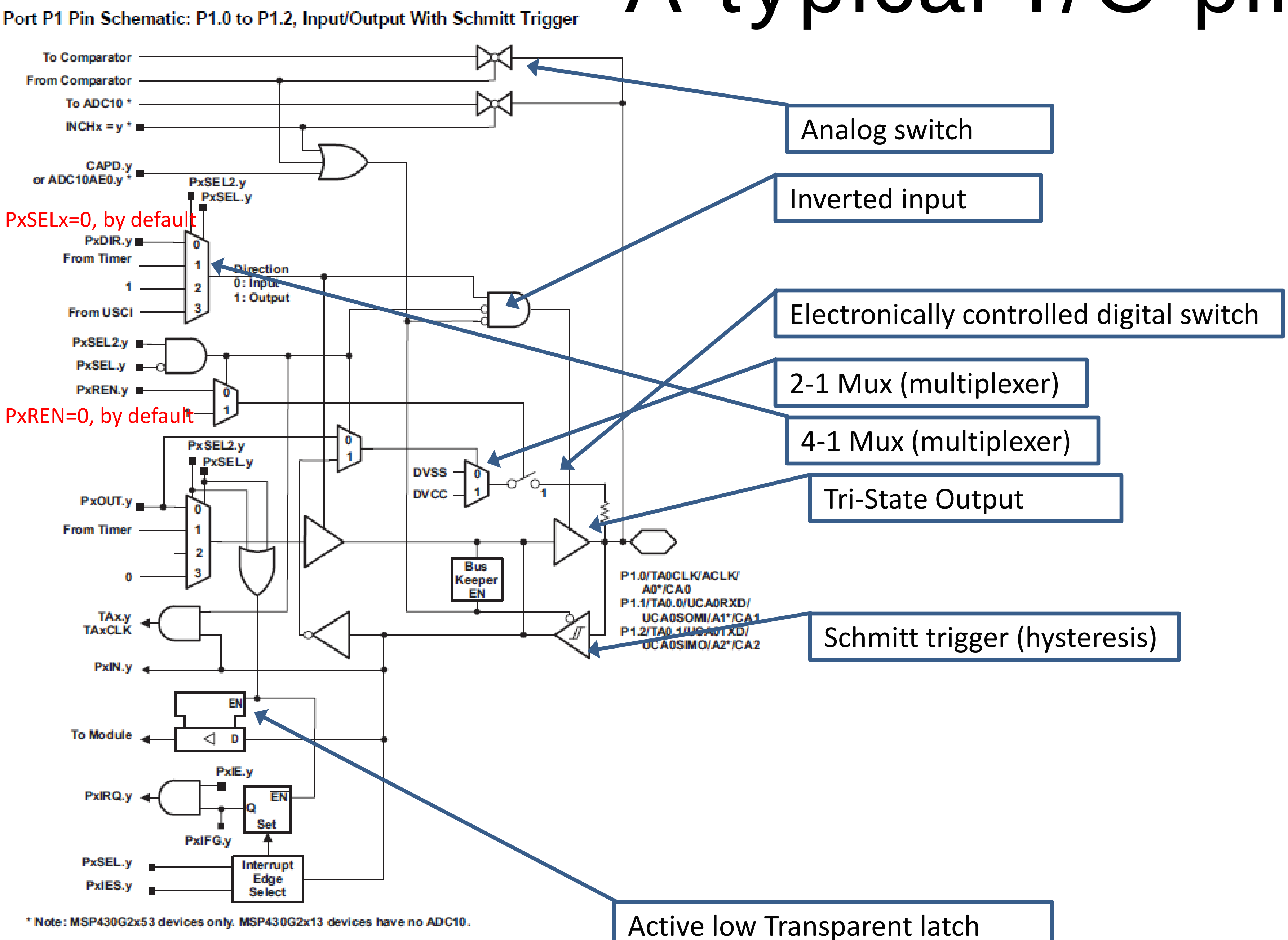
Bit = 1: The output is high

If the pin's pullup/pulldown resistor is enabled, the corresponding bit in the PxOUT register selects pullup or pulldown.

Bit = 0: The pin is pulled down

Bit = 1: The pin is pulled up

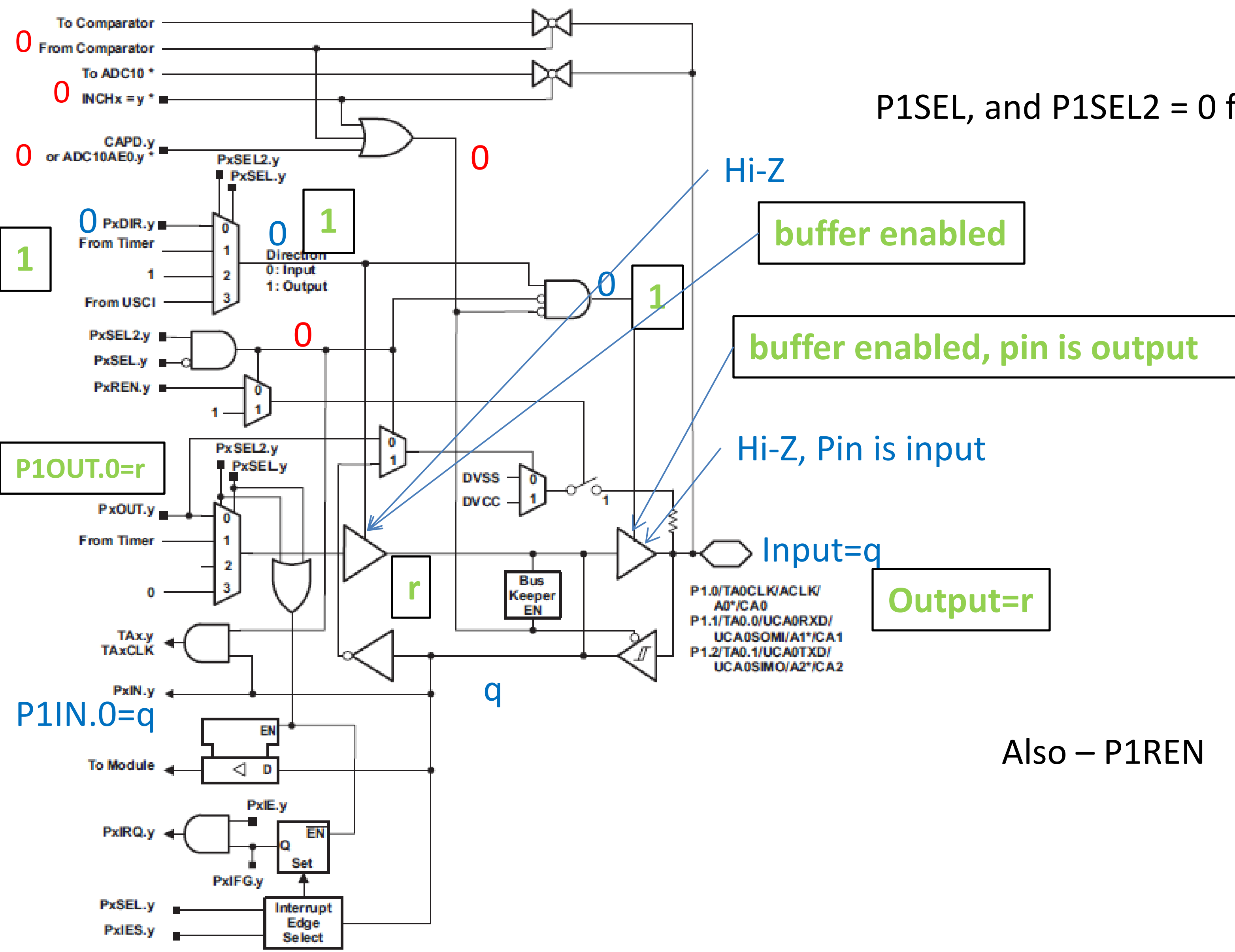
A typical I/O pin



Effect of P1DIR

P1SEL, and P1SEL2 = 0 for I/O

Port P1 Pin Schematic: P1.0 to P1.2, Input/Output With Schmitt Trigger



* Note: MSP430G2x53 devices only. MSP430G2x13 devices have no ADC10.

Also – P1REN

A simple C program

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.

```
#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
  }
}
```

"1" is always true, so loop forever.

Set bit 0 high (connected to LED)

Loop to waste time

Set bit 0 low (LED turns off)

Loop to waste time

Don't worry about for now.

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 "}"

Variant 1 (more readable)

```
#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
  }
}
```

Give constants meaningful names.

Equivalent to: `P1OUT = P1OUT | LED;`

Variant 2 (macros)

```
#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  }
}
```

Can call bits by location

Use Macros sparingly, but they can make code look much cleaner (see below)

Expands to: `(P1OUT |= (0x01))`
Note ";" must be added.

Variant 3 (shorter)

```
#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
  }
}
```

Loop is half as long as before

More C

Statements

- a **simple statement** is a single statement that ends in a ";"
- a **compound statement** is several statements inside braces:

```
{
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.

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

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

```
if (x < 0)
{   printf("Negative");
    negative(x);
}
else
{   printf("Positive");
    positive(x);
}
```

Even more C

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…)

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

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

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

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

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

is equivalent to

```
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 exectuted.
- 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://en.wikipedia.org/wiki/C_syntax
- http://en.wikipedia.org/wiki/Indent_style
- http://en.wikipedia.org/wiki/Operators_in_C_and_C++
- 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)