ECE2049
Embedded Computing in Engineering Design

E-Term 2021

Instructor: N. DeMarinis

Lecture Workbook

Module 1. Intro to Number Representations

Topics

- How do we store (or "encode") information in digital systems?
- Specifically: how do we store numbers?

First things first: Remembering Digital Logic

Before we can talk about how computing systems are built, we first need to talk about their basic building block: digital logic. In digital logic, information is represented in binary *bits.*

Digital logic defines how we can process information using bits:

First things first: *n* bits differentiate among 2^n things.

Terminology: 1 byte = 8 binary digits = 8 bits (e.g. 10010011)

½ byte = 1 nibble = 4 bits

1 word = 2 (or more) bytes --> MSP430 word = 2 bytes

1 double word = 2 words (4 bytes on MSP430)

In computers, ***information*** and ***memory space*** is organized in to multiples of bytes.
But what do the bytes mean?

The meaning of bits and bytes <u>assigned by convention</u>!

>> Under a given coding convention, a byte can represent up to $2^8 = 256$ things

For example, 1 byte (8 bits) could encode:

- A letter in an alphabet

- One or more decimal numbers

- The state of eight individual things (one per bit)

- An *instruction* that tells the CPU to do something:

We call these conventions **encoding formats**. They represent a kind of contract on how data will be stored and used. As programmers, it is up to us to assign meaning to those bits—which defines what operations we perform on them.

Conversion between Bases and Formats: Binary

Positional Number Systems

We write numbers in a positional system, which can be defined as:

For binary numbers, we can write this definition as:

Unsigned integers = All bits used to convey magnitude (whole numbers ≥ 0)

Decimal to Binary Conversion – Successive Division

Note: To differentiate numbers in different formats, we use notation to denote the radix used wo write it. For binary: 1010_2 or 1010b; decimal: 1010_{10} or 1010d (or just 1010)

Hexadecimal: A common way to write binary numbers

Since working in binary can be cumbersome, we often write numbers in *hexadecimal*, which is base 16.

Simple rule for conversion:

--> 1 Hex character represents values from 0 to 15d using digits 0 – Fh

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

Ex. 158d =

<u>If you memorize anything in this class, memorize these!</u>

Notation: Numbers in hex are written as 1010h or 0x1010

Conversion between hex and binary is piece of cake! Just convert each hex digit to a binary nibble...

1001 1110b = 158d

Or vice versa:

8AC4h = 8 A C 4

Note: A modern computer always stores information in binary form. Writing in hex is just a faster way for us to read and write these numbers—the machine's representation is still binary.

How do we store negative numbers?

One way: Sign Magnitude integers = *n*-1 bits used to convey magnitude with "most significant bit" or MSB used for sign. Convention: 0 = +, 1 = -

Note: This format has 2 representations of 0 = +0 and -0 !

Another way: Two's Complement integers = More common format for signed integers. For *n* bits, values range from $-2^{(n-1)}$ to $2^{(n-1)}-1$

How it works:

Positive numbers: Follow same format as unsigned numbers

1026 = 0000 0100 0000 0010b = 0402h

Negative numbers: *Write* magnitude, *Complement* each bit, *Add* 1

-15 =

Range of Values for 2's Complement

```
0111 1111 1111 1111b
0111 1111 1111 1110b
       . . .

0000 0000 0000 0000b

        ...
1000 0000 0000 0001b
1000 0000 0000 0000b
```

Ex: Find the 8-bit two's complement representation of 104 and -80

Ex: What are the decimal equivalent values of these 2's complement values

0010 0011b

1000 0011b

Ex: What decimal value does 8008h represent as an…
(a) unsigned integer (b) 2's comp integer (c) sign-magnitude integer

What about things that aren't integers?

Characters

To handle letters and other displayable characters, we need an encoding format to describe how we can represent these values in binary. One very common format for this is ASCII (American Standard Code for Information Exchange), which defines a table of binary codes that represent various characters.

Note: Other formats exists for representing different types of characters (alphabets and character sets for all human languages, emoji, etc.). For information on this, see "Unicode".

Unicode Examples

Unicode Name	Bit representation	Character
U+00FC LATIN SMALL LETTER U WITH DIAERESIS	C3 BC	ü
U+1F602 FACE WITH TEARS OF JOY	F0 9f 98 82	
U+1F363 SUSHI	F0 9F 8D 83	

Non-integer Numbers

In future lectures we will talk about representing non-integer data. These are called fixed-point and floating-point data types, which we will cover soon!

Preview: How is data actually stored in a program?

C defines a set of standard data types to store information. Each datatype has a specific representation, which depends on the compiler and the CPU architecture.

For the MSP430 architecture, the standard datatypes are defined as follows:

```
int a;           // 16-bit two's-complement signed (2 bytes)
unsigned int b;  // 16-bit unsigned integer (2 bytes)
long int c;      // 32-bit signed integer (two's complement) (4 bytes)
char d;          // 8-bit unsigned integer (1 byte)

float e;         // 32-bit IEEE754 single-precision floating point value (4 bytes)
double f;        // 64-bit IEEE754 double-precision floating point value (8 bytes)
```

Note that the types `char`, `float`, and `double` have the same size on all architectures–these are part of the C standard.

We can use these standard datatypes to hold different kinds of information (signed/unsigned numbers, characters, decimal values), or compose more complex types (like arrays or structs).

Important: The size and type of a variable define the range of values they can represent!

- The value of a variable CANNOT exceed the fixed size of the variable
- Variables will "overflow" or "roll over" if the value exceeds the variable size!

Module 2. Data Representations & C Programming Basics

Topics for Today

- More on data representations
- C programming basics

Last Time

- Introduction and Policies
- Data representations for integers

Warmup: How many bits are in a byte?

So how are variables actually stored?

Each datatype has a specific representation, which depends on the compiler and the architecture.

For the MSP430 architecture, the standard datatypes are defined as follows:

```
int a;           // 16-bit two's-complement signed integer (2 bytes)
unsigned int b;  // 16-bit unsigned integer (2 bytes)
long int c;      // 32-bit signed integer (two's complement) (4 bytes)
char d;          // 8-bit unsigned integer (1 byte)
float e;         // 32-bit IEEE754 single-precision floating point value (4 bytes)
double f;        // 64-bit IEEE754 double-precision floating point value (8 bytes)
```

Note that the types char, float, and double have the same size on all architectures–these are part of the C standard.

Important: The size and type of a variable define the range of values they can represent!

- The value of a variable CANNOT exceed the fixed size of the variable
- Variables will "overflow" or "roll over" if the value exceeds the variable size!

Example: a `char` has a size of 8 bits (or one byte), and thus can hold values from 0 to 2^8 - 1 = 255.
What happens if we try to do the following?

```
char c = 253;
char a, h;

for(a = 0; a < 4; a ++) {
   h = c + a;
}
// As we run this:
// a = 0, h = 253
// a = 1, h = 254
// a = 2, h = 255
// a = 3, h = 0 <--- Rollover!
```

This is a very important takeaway about datatypes–you always need to make sure your datatypes are appropriately sized for your application!

Don't like how ints are different sizes on different architectures?

Yeah, me neither. And neither did the people who wrote later C standards. If you include stdint.h, you can use datatypes that look like these:

```
#include <stdint.h>

uint8_t  a;    // Unsigned, 8 bit integer (aka char)
uint16_t b;    // Unsigned 16-bit integer (aka unsigned int on the MSP430)
int16_t c;    // Signed 16-bit (2's comp) integer
uint32_t d;    // Unsigned 32-bit integer
int32_t e;     // Signed 32-bit integer (2's comp)
// Similar types exist for 64 bit integers, and 128-bit on some
architectures...
```

You are welcome to use these in your labs!

Recall: Characters

One common format for representing characters is ASCII (American Standard Code for Information Exchange), which defines a table of binary codes that represent various characters.

Example: in ASCII, the character the decimal value 68 (or 44h), represents the character 'D'.

In C, we can represent enter ASCII characters using 'single quotes', like so:

```
char a = 'D'; // Assigning c to the character value of 'D'


// This is the same as writing
char a = 68;
// or
char a = 0x44;
```

Note: Other formats exists for representing different ranges of characters (like other alphabets, emoji, etc.). For information on this, see "Unicode".

C Programming for Embedded Systems

Rule #1: a program will always do exactly what you tell it to do!

Here is an example of a simple C program:

```
#include <stdlib.h>

void main(void) {
    float degF, degC, degK;
    degF = 45.7;
    degC = 5.0 * (degF - 32.0) / 9.0;
    degK = degC + 273.15;
}
```

What does this code do? Is it correct? Is it useful?

Terminology: Compilation Process

When you write a C program and build it, the **compiler** and **linker** are responsible for turning your code into machine instructions that the MSP430 can execute and arranging your variables, code, and definitions in the program's memory. The output of the compilation process is an **executable** that runs on the MSP430.

Compiler:

Linker:

Why is it important to know these terms? They will help you debug compilation problems!

Basic data types

Our MSP430 compiler users the following sizes for basic data types:

```
int a;           // 16-bit two's-complement signed integer (2 bytes)
unsigned int b;  // 16-bit unsigned integer (2 bytes)
long int c;      // 32-bit signed integer (two's complement) (4 bytes)
char d;          // 8-bit unsigned integer (1 byte)
float e;         // 32-bit IEEE754 single-precision floating point value (4 bytes)
double f;        // 64-bit IEEE754 double-precision floating point value (8 bytes)
```

One of your first tasks when writing a program is creating variables of suitable size for your problem!

Declaring variables

```
int x;      // Reserve space for an int.  What is x's value?



char z = 5; // Store 5 in 8-bit

char array[10]; // Make space for 10 bytes
int ints[5] = {1, 2, 3, 4, 5}; //Initializing an array
```

Arithmetic Operators

If you've seen C before, you have used these.

Arithmetic Operators: `+ - * / % (% == Modulo, or the "Remainder operator"`)

```
int x, y, z, u;
float a;

u = (x + z) * y / z; /* y / z is the integer part of the division—truncation!*/

z = 47;

y = z / 10;


x = z % 10;
```

Casting is a way to change type of a variable. The compiler will add the appropriate routines to convert a variable from one representation to another for you.

```
// Let's say we want to divide an integer to get a
// decimal result?
float a;
int z = 47;

a = ((float)z)/10.0; // a = 4.7
```

Unary and assignment operators: `+= - += -= *= /=`

These operators are shorthand (also called "syntactic sugar") for other operations:

```
i++;      // i = i + 1
j--;      // j = j - 1
i += 2;   // i = i + 2;
k *= 4;   // k = k *4;
```

(These operators are called *unary* because they only take a single argument (eg, `a += 5`), as opposed to *binary* operators, which use two arguments (eg. `a + b`)

More shorthand with increment/decrement operators:

```
i = 5;
arr[i++] = 4;
```

Be careful with these!

Logical and Relational Operators

Logical operators, such as produce a Boolean result (true or false).
But how are Booleans represented in C?

Relational Operators: `>, >=, <, <=, ==, !`

These operators return a Boolean (1 or 0) result:

```
if (x > y) {
  z = z - x;
} else {
  x = z - y;
}
```

```
while(x != 0) {
 // ...
}

while(z < 5) {
 // ...
}

while(count) {
 count--;
 // . . .
}
```

Logical Operators: `&& (AND), || (OR), == (EQUAL), ! (NOT)`

```
if ((j == 0) || (x < 100)) {
  // ...
}

canContinue = 1;
while((i < 5) && (canContinue)) {
 // ...
 canContinue = 0;
}
```

Bitwise Operators

In systems-level (and embedded) programming, we often need to operate on individual bits of a variable.

Bitwise Operators: `& (AND), | (OR), ~ (NOT), ^ (XOR), >> (Right shift), << (Left shift)`

These operators operate on each bit of the data type (hence, bitwise):

```
char a = 0x85;   // 1000 0101b
char b = 0xF0;   // 1111 0000b

char k = a & b;









char m = k >> 2;









char c = ~b;
```

Control flow

If/else statements (also called "Conditionals")

Used for making decisions:

```
if ( k > 100) {
    k = 0;
} else {
    k += 1;
}
```

Alternate form for small statements, the conditional operator (also known as the *ternary* operator):

```
k = (k > 100) ? 0 : k + 1;
```

You can also have many conditional blocks:

```
if(x > 0) {
   y++;
   doSomething(x, y);
} else if ((x > 0) && (y != 2)){
    y = 100;
    // ...
} else if(x > 100) {
    // ...
} else {
   // ...
}
```

NOTE: Brackets around your if/else statements (or loops) are not required, but you should always use them!

Switch/Case statements

```
x = getValue();

switch(x)
{
case 1: // if x == 1
    doSomething(x, y);
    y = 0;
    break;  // Must have these at the end of each case.  Why?
case 2: // if x == 2
    doSomethingElse();
    break;
case 12: // if x == 12
    doSomeOtherThing();
    break;
default: // For all other values
    break;
}
```

Case statements can be useful for making decisions about a single value. They can also be useful for implementing complex control structures like state machines, which we will discuss later.

Loops

While loops

Can use to iterate over a set of values:

```
i = start_value;

while(i < end_value) {
    /* Body of loop */
    // ...
    i++;
}
```

Example: How many times will the body of the loop execute?

```
int a = 32;
while(a > 0) {
    // . . .
    a = a - 8;
}
```

Can also use a while loop to wait for something:

```
int data_is_ready = 0;
while(data_is_ready != 0) // Stay in loop until data is available
{
   data_is_ready = get_data();
}

// After the loop, use the data
do_stuff_with_data();
```

For loops

For loops are a different syntax for a simple while loop (like the first example):

```
int i;
for(i = start_value; i < end_value; i++)
{
    /* Body of loop */
}
```

Break and Continue

The break keyword will exit the current loop. The continue statement will skip the rest of the current iteration and start the next one.

```
int i;
int data[100];
for(i = 0; i < 100; i++)
{
    if(check_input(arr[i]) == -1) {
        break;
    }
    do_thing(arr[i]);
}
// . . .
```

```
int i;
int data[100];
for(i = 0; i < 100; i++)
{
    if(check_input(arr[i]) == -1) {
        continue;
    }
    do_thing(arr[i]);
}
// . . .
```

The "forever" loop

Infinite loops are not often desirable in programs. However, embedded programs use them all the time in certain circumstances, like your main function.

```
void main(void)
{
  /* Initialize variables, do setup tasks */

  while(1){
      // Perform tasks that your device needs to do!
  }
}
```

We will discuss more about how to write programs using this paradigm later.

More Data Representations

Arrays

Arrays are contiguous group of a certain data type, stored sequentially:

```
// Declare an array of 10 ints
int a[10];

// Initialize an array
int arr[4] = {1, 2, 3, 4};

// "Indexing" an array
int a0 = arr[0];
int a1 = arr[1];

int a_last = arr[3];

int *arr_ptr = arr; // Name of array is pointer to its first element
```

Strings

In C, we can also define arrays of characters using "double quote", which make up groups of displayable characters.

Convention: C-style strings (or "null-terminated strings"): arrays of ASCII characters followed by a special byte called a null-terminator (which has value 0x00, usually written as `'\0'`).
When you type a string in "double quotes," a null-terminator is automatically included.

The null terminator is used to tell functions that operate on strings when it reaches the end of the string.

For example, we can represent the string "ECE2049" as follows:

```
char *str = "ECE2049"; // The string "ECE2049"

// This is the same as writing out each character in array form
char str[8] = {'E', 'C', 'E', '2', '0', '4', '9', '\0'};

// Or we could write out each character in decimal or hex.
char str[8] = { 'E',  'C',  'E',  '2',  '0',  '4',  '9', '\0'};
char str[8] = {0x45, 0x43, 0x45, 0x32, 0x30, 0x34, 0x39, 0x00};

// All have the same meaning, we are just entering them differently!
```

We will discuss strings in more detail later, but you should know that they exist since you will see them in lab. You should also know about the existence of null terminators.

Pointers

A pointer gives the location of something of in program memory—this is also known as a memory address. We will discuss pointers in further detail later.

Complex data: `structs`

We can define complex data types called structs, which are composed of other data types:

```
// Defining a struct
struct point {
    int x;
    int y;
};

// Declaring variables of type "struct point"
struct point p1;

// Setting and accessing members of a struct
p1.x = 5;
p1.y = 2;
// . . .

struct point p2 = {1, 2}; // Declaring and initializing a struct

int z = p1.x + p2.x;
```

We will discuss these in more detail soon.

Data representations: Would you like to know more?

In the next segment, we will talk *even more* about data representations!

- Representing fractional numbers: fixed-point and floating point
- Machine code: the compiler turns your C code into a binary format to create instructions the CPU can understand

Program structure in C

In C (as in other programming languages), you can separate your programs into a series of smaller functions to complete certain tasks.

```
#define MAX_BUTTONS (4) // Constant for the number of buttons on the board

int some_value;         // Global variable (visible to whole program)

// Function prototypes
int check_button(char button_id);

void set_led(char led_id);


// Function definition
int check_button(int button_id)
{
    int button_state = 0; // Local variable for this function

    // ...actual function body goes here...

    return button_state;
}

// Need to implement other functions too!

void main(void)
{
   // Variables local to main
   int i, button;

   while (1)
   {
       for(i = 0; i < MAX_BUTTONS; i++) {
         int button = check_button(i);
         if(button == 1) {
             set_led(i);
         }
       }
       // Do other things.. perhaps wait for a while?
   }
}
```

Module 3. Of Integers and Endians & Floating Point Representations

Topics

- Memory organization and endianness
- More data representations: overview of floating point

Last Time

- C programming basics
- Data representations for characters

Warmup: try the following…

```
int z = 0x4007;

// a. What is the size of z (in bytes)?
// b. In C, how is z stored (unsigned, sign-magnitude, 2's comp)?

if (z & 0x8000) {
    alpha();
} else {
    beta();
}

// c. Based on the value of z, which function would get called?
```

Memory organization

What does it mean to type "`int a`" in C? This is called variable declaration, which allocates space in the program's memory to store an int.

What do we mean by memory? You can think of memory as a big table of "addresses" that each map to a certain piece of data. This data could be a variable (as above), or it could be a piece of code, a portion of the hardware, etc., but for now let's focus on variables.

On the MSP430, addresses are 16-bits long, and each address refers to one byte.

Recall that the MSP430 is a 16-bit architecture,

Unfortunately, this is no longer completely true! Newer MSP430 variants (like ours MSP430F5529) utilize 20-bit addresses. Why?

Laying out variables in memory

When you declare variables in your program, they are arranged in memory starting at a certain address. For now, it is sufficient to know that variables in `main` start at address 0x4400. We will discuss why in an upcoming lecture.

When variables are declared, they are (usually) arranged in order from this starting address. For example:

```
char a = 0x11;
char b = 0x22;
```

…can be arranged in memory as follows:

Address	Data	Variable

In our class, we will arrange memory in a table like the one above, with the starting address at the bottom. We use this convention because we are typically representing variables on the program *stack*, which starts at a fixed base address and grows up.

Endianness: Ordering bytes

In the previous example, we have left out an important detail. How do you store variables that are larger than a byte?

As declared on the MSP430, a long is has a size of four bytes:

```
long v = 0xAABBCCDD; // AAh is the  most significant byte (MSB), and
                     // DDh is the least significant byte (LSB)
```

For multi-byte variables, we have a choice–do we arrange the data with the least significant byte first, or with the most significant byte first? Which is correct? Does it matter?

This concept is known as *endianness*, which governs how a processor orders bytes in memory. There are two forms of endianness:

Little Endian (LE)

Little Endian stores the *least significant byte first*, meaning that the memory in this example would be arranged as follows:

Address	Data	Variable
0x4403	AA	V
0x4402	BB	
0x4401	CC	
0x4400	DDh	

Big Endian (BE)

Big Endian stores the *most significant byte first*, as follows:

Address	Data	Variable
0x4403	DD	v
0x4402	CC	
0x4401	BB	
0x4400	AAh	

Important points on endianness

- Endianness is a fundamental part of the architecture's design. When a processor is designed, it is designed to use a specific byte order–you cannot change this with a compiler setting.
- Is one endianness better than the other? No, they simply reflect different design choices.
- Big endian is read "left to right", which is intuitively easier to read for those accustomed to languages written left to right
- Little endian makes it easier to slice out small portions of a variable (eg, what if you only want the first byte of a long?)

When will you deal with endianness?

Endianness becomes especially important when you need to transfer data between different architectures. Examples include any stored data format or network protocol.

More memory layout: Arrays

How do arrays work, anyway?

In C, we can declare arrays and use them as follows:

```
// Declare an array of 5 bytes
char arr[5];
// Declare an array of 5 bytes, and initialize it (set it with some initial values)
char arr[5] = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };

// You can access elements of an array by "indexing" into it
// In C, array indexes start at 0
char c = arr[0]; // The first element
char d = arr[4]; // The last element (arr has size of 5, so last index is 5 - 1 = 4
```

You can think of the elements of the array laid out like this:

Index	0	1	2	3	4
Element					
Value					

Why is it important that array elements are contiguous? (And must contain elements of the same type?)

What would happen if we tried to get the 6th element of `arr`?

How endianness affects arrays (or rather, how it does not)

A fundamental property of arrays is that their elements are stored contiguously in memory in order of their index (as discussed above).
Endianness does not change the order of array elements.

For example, if we laid out the array from the above example on a Big Endian (BE) and a Little Endian (LE) system, it would look like this:

Address	BE	LE
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

However, endianness *does* affect the ordering of the bytes in each element of the array! In the previous example, the elements were just 1 byte each!

Example: an array of `int`s:

```
int iarr[2] = {0x1122, 0x3344};
```

Here, the memory would be organized as follows:

Address	BE	LE
0x4403		
0x4402		
0x4401		
0x4400		

Using addresses as data

We can also have variables that contain memory addresses. These are called *pointers*.

You can get the address of a variable with the *"address-of" operator* (&):

```
long v = 0x11223344;
long *pv = &v;
```

In this example, we say that `pv` is declared as the type "pointer to long," which is indicated by the "*" before the name `pv`.

How big is `pv`?

,

What is the value of `pv`?

We can lay out these variables in memory as follows:

Address	BE	LE
0x4405		
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

How big is a pointer?

A pointer is the size of a memory address for a given architecture. On the MSP430, an address has a size of 2 bytes (16 bits).

Type	Size (bytes)
`int`	
`long`	
`char`	
`long long`	

Type	Size (bytes)
`int *`	
`long *`	
`char *`	
`long long *`	

This is one way in which pointers are powerful: a pointer can represent a larger data structure in the program—by passing around the pointer, we can avoid copying or moving the larger data structure.

How are pointers used with arrays?

Whenever you use arrays, you use pointers. Consider the following example:

```
int iarr[10];
int i = iarr[5];
```

When you index into an array, the program actually does the following:

```
int i = *(iarr + 5); // Equivalent to writing iarr[5]
```

Here, the ***** is the *dereference operator*, which **gets the value at the given address**. This is called *dereferencing* the pointer—it is the opposite of the *address-of* (&) operator.

Working with Pointers

Pointer math: When performing arithmetic operations on pointers, the address changes in increments based on the type of the pointer.

```
// Example 1: array of char
char carr[4];
// How big is the array?

// Say the starting address is 0x4400, what is the address of carr[3]?
```

```
// Example 1: array of int
int carr[4];
// How big is the array?

// Say the starting address is 0x4400, what is the address of iarr[3]?
```

Passing arrays: Further, when you use the name of an array (either to store or pass to a function), you are *passing a pointer to the first element of the array*. This is the "starting point" of the array used as input to calculate the index.

```
int *ptr = iarr;    // Could also write &iarr[0]
do_thing(iarr, 10); // Same here

void do_thing(int* arr, int size) { // Function takes pointer to array (+ size)
    // . . .
}
```

Memory organization example

Here's a larger example of memory organization. How would we organize the following variables?

```
unsigned int a = 0x00FF;
long int b[2] = { 65540, -5 };
char c = 'c'; // 'c' = 0x43
```

How many bytes of memory do we require?

So using the above information, we can make our table:

Address	BE	LE
0x440C		
0x440B		
0x440A		
0x4409		
0x4408		
0x4407		
0x4406		
0x4405		
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

How do you represent fractional numbers in binary form?

So far, we have only expressed integer values in binary. There are two conventions for representing fractions: fixed point and floating point.

Fixed Point

In a given data type, we can define a binary "radix point", which is a fixed point that denotes fractional bits.

In this format, the precision of the number is defined by the number of fractional bits.

For example, 4 fractional bits = $2^{(-4)} = 0.0625$ is the smallest fraction you can represent

Often, fixed-point representations are stored in scaled form as integers–it's up to you (the programmer) to treat them as fixed-point values.

Floating Point

Floating point is an IEEE standard used to approximate real-valued numbers to a certain number of decimal places.

There are two forms, *single precision* (32 bits) and *double precision* (64 bits). Each representation has three components:

- A sign bit (S)
- An exponent (E)
- A fractional part (F), which is also called the "mantissa" or "significand"

Format for single precision: S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
Exponent is 8 bits, fractional part is the remainder (23 bits)

$$\text{Value} = (-1)^S * 2^{(E-127)} * (1.F)$$

Example: Floating point to Decimal
What is the decimal equivalent of the floating point variable CAAA0000h?

Some features of floating point:

- Effective range: approximately +/- 10^{38}
- Single precision has ~7 decimal digits of precision
    - Double precision and others have more
- Special representations for +/- infinity, NaN
- Standard has conventions for rounding, normalization, etc.

Example: Decimal to floating point

Represent -5.375 as a single-precision floating-point number.

Module 4. MSP430 Architecture & Intro to Digital I/O

Topics

- Getting to know the MSP430 Hardware
- Start of Digital I/O

Last Time

- Memory Organization
- Floating point format

Getting to know the MSP430 Hardware

In a programming course, typically you focus on just the code:

- Learn a high level programming language and some algorithms
- Use a "computer" from a high level

A typical program does three things:

A few common points for all kinds of software:

- Use constructs like loops, conditional, algorithms like search, sort, data structures
- Software: write logic and syntax correctly and it will just work
- Library functions for I/O

... but what's inside the "computer"? How does it work? When you are writing in a high level programming language, do you care?

In contrast, developing software for embedded **requires** much more in depth knowledge about the microprocessor that is the target of your program. For instance, it's important to know:

A general software hierarchy

We can think of the software components in a system and the way they interact with the hardware as a hierarchy or *software stack*:

A general computing device (PC) might have a software hierarchy like this:

Applications
Operating System (Includes user interface)
System Software: Interface to hardware
Hardware (CPU, Memory, Peripherals)

On an embedded system, this stack gets “squashed”:

Application
System software / OS
Hardware (CPU, Memory, Peripherals)

- Application is closely integrated with the hardware layer
- Little or no operating system–usually only runs one task or a set of tasks
- Often little or no "wrapping" of functionality
- On larger systems, you may use a Real Time Operating System (RTOS) that provides some basic support for multitasking…

A general microprocessor hardware architecture

In general, any microprocessor system has the following components:

CPU (Central Processing Unit)

The "core" of the computer

Memory

Stores information

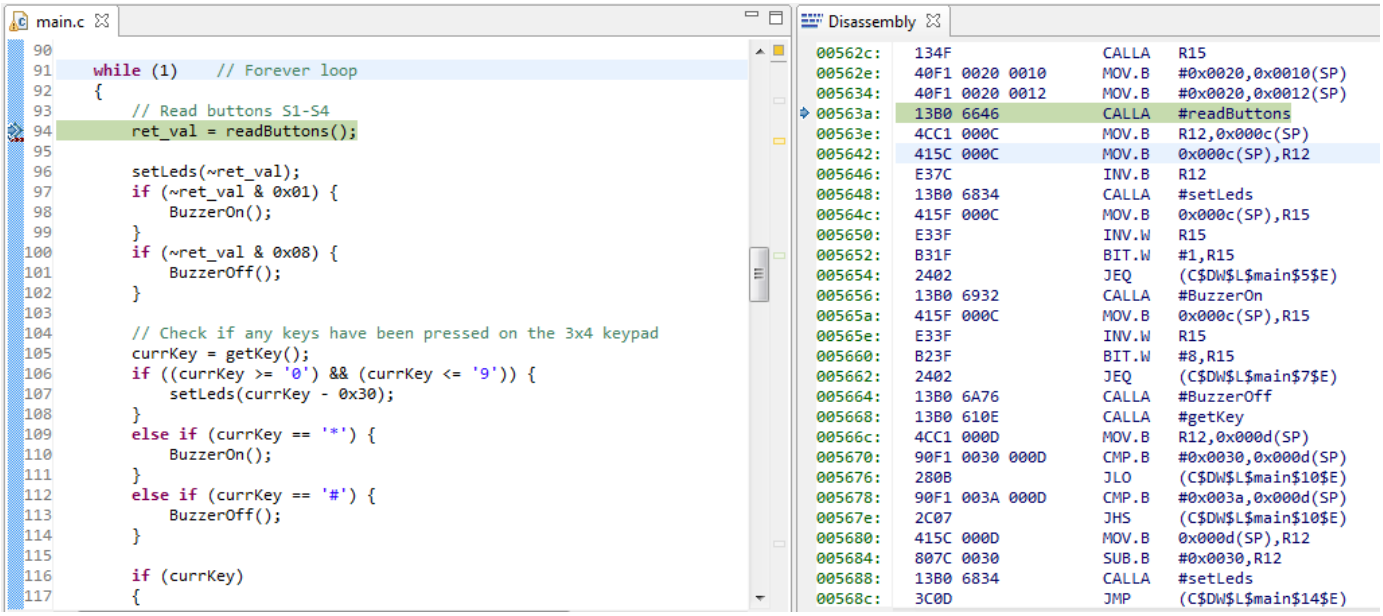
Peripherals

How does the CPU work?

The CPU executes *machine code*, which are low-level instructions directly run by the hardware. Machine code is a binary format seen by the CPU.

- Instructions perform very specific tasks
- Instruction set (ISA) is different for every CPU type (MSP430, ARM, x86, …)

- **Compiler is responsible for figuring out how to build all programs using these instructions!**



We will never write in assembly in this class. However, it is important that you understand that these instructions exist!

CPU instructions operate on…

- **Internal Registers**: 16 general purpose registers (R0-R15)
  - Storage locations inside the CPU used for recent instructions
    All registers are 16-bits wide (except R0 and R1, which are 20 bits)
  - Can be accessed very quickly (one clock cycle)
  - Some registers control program execution
    (R0 = Program counter, R1 = Stack pointer, R2 = Status register)
- **Memory**: Instructions read from and write to memory
  - Load and store data from the outside world using the memory bus!

What goes in memory?

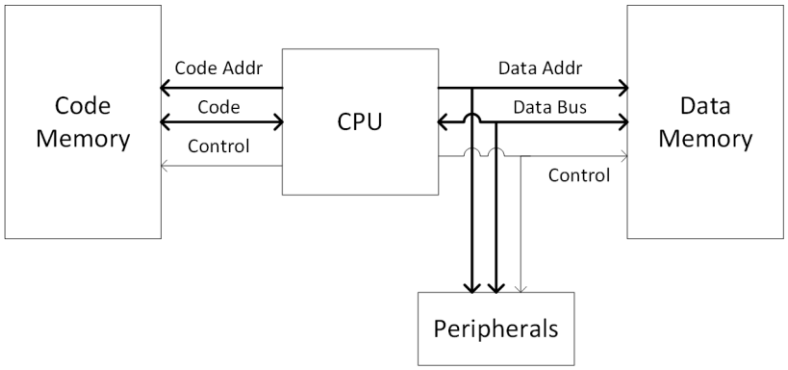
Remember that memory doesn't just store your variables–it stores the program's code as well!

- The CPU needs to load both code **and** data from memory

There are two generic types of memory architectures used by microprocessors and microcontroller systems:

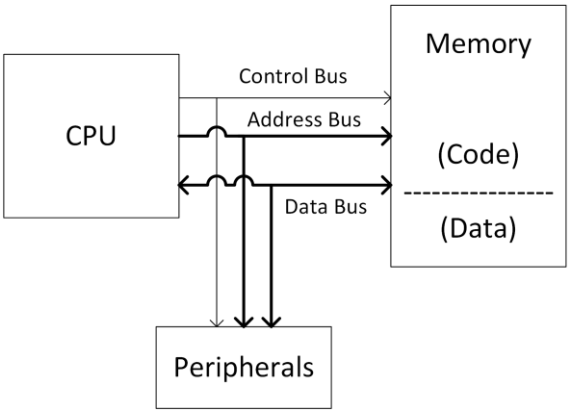
- **Von Neumann** Architecture (~1952)
- **Harvard** Architecture (~1944)

Harvard Architecture: Separate memory address spaces for code and data



Benefits: Instruction fetch and data read happen in parallel
Drawbacks: Separate instruction and data buses
In this form, the Harvard architecture is used today by highly-pipelined systems like DSP chips.

Von Neumann Architecture: Single memory address spaces for code and data



Benefits: Single address and data buses (simpler to interface)
Drawbacks: Implicit bottleneck since we have the same pipeline for code and data

The MSP430 Architecture

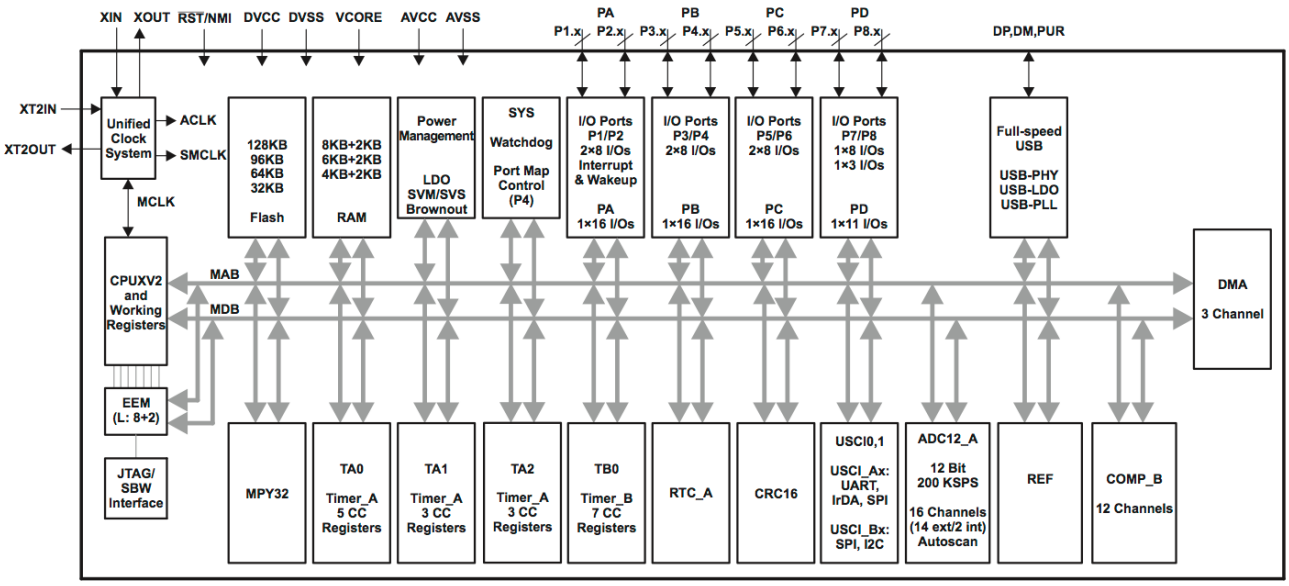
The MSP430 is a family of microcontrollers–there are hundreds of versions of this CPU with various configurations of memory and peripherals!

- You can think of it as a type of *System on a Chip* (SoC)

In our labs, we use the MSP430F5529

- 128KB of flash memory: Used for code storage
- 8 KB of RAM (+ 2KB RAM for USB controller): Used for data storage
- Lots of peripherals
  - 32 bit multiplier
  - Timers, comparator, USB controller
  - Much, much more!

How much more? Here's a block diagram:



Note the lines connecting all of the peripherals: this is the memory bus!

MAB: Memory Address Bus

MDB: Memory Data Bus

MSP430 Memory Organization

Memory: A group of sequential locations where binary data is stored

- On the MSP430, each memory location holds one byte
- Each byte has a unique address which the CPU uses to access it
- Multibyte data is stored in _____ Endian!

Two types of memory: Volatile and Non-Volatile

RAM (Random Access Memory)

- Our MSP430 has 8KB of RAM + 2KB for USB
- RAM is **volatile**, meaning that it loses its state when the chip is not powered
- Used as data memory
- Accessed via read and write instructions

Flash

- Used primarily for code memory
- Flash is **non-volatile**, meaning that its state persists even if the chip is not powered
- CPU fetches code from flash automatically
- Accessed via program control, but more difficult than RAM
    - Write time >> Read time
    - Writes must occur in large segments (512 bytes)

How are programs stored in memory?

When a program is compiled, the linker arranges different portions into various memory *segments*, which are stored in different contiguous memory regions. The most important segments are:

- The stack (`.stack`): Stores local variables and context information on each function call
- Constant data (`.data`, `.bss`): Stores global variables and other constant data (strings, lookup tables, etc.
- Text (`.text`): Compiled code for your program (code you write + libraries)
- Heap: Dynamically allocated memory (avoid using this!)

When compiling, the linker reads a script called a *command file*, which maps each section to a memory device. Usually, most code is stored in flash, while most data goes in RAM, though it may be necessary to adjust these requirements. Why?

Why should we avoid dynamically allocated memory?

Memory architecture and layout

The MSP430 is a 16-bit microcontroller, meaning that:

- The data bus is 16 bits wide
- Internal CPU registers are 16-bits

Note: MSP430 '5xxx and '6xxx families use a **20 bit address bus** to allow access to at most 1MB of memory.

However, memory isn't just one big block….

Mapping Memory

In practice, the “memory space” is *mapped* across the different types of memory and hardware devices connected to the CPU.

- This includes the different types of physical memory (RAM, flash), as well as *hardware peripherals*
- The mappings of which components use which addresses is based on the physical wiring in the IC (we know the mappings based on the header files)

The Memory Map (found in MSP430F5529 datasheet)

Table 5. Memory Organization(1)

		MSP430F5522 MSP430F5521 MSP430F5513	MSP430F5525 MSP430F5524 MSP430F5515 MSP430F5514	MSP430F5527 MSP430F5526 MSP430F5517	MSP430F5529 MSP430F5528 MSP430F5519
Memory (flash) Main: interrupt vector	Total Size	32 KB 00FFFFh–00FF80h	64 KB 00FFFFh–00FF80h	96 KB 00FFFFh–00FF80h	128 KB 00FFFFh–00FF80h
Main: code memory	Bank D	N/A	N/A	N/A	32 KB 0243FFh–01C400h
	Bank C	N/A	N/A	32 KB 01C3FFh–014400h	32 KB 01C3FFh–014400h
	Bank B	15 KB 00FFFFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h
	Bank A	17 KB 00C3FFh–008000h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h
RAM	Sector 3	2 KB(2) 0043FFh–003C00h	N/A	N/A	2 KB 0043FFh–003C00h
	Sector 2	2 KB(3) 003BFFh–003400h	N/A	2 KB 003BFFh–003400h	2 KB 003BFFh–003400h
	Sector 1	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h
	Sector 0	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h
USB RAM(4)	Sector 7	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h
Information memory (flash)	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
Bootstrap loader (BSL) memory (flash)	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
Peripherals	Size	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h

(1) N/A = Not available
(2) MSP430F5522 only
(3) MSP430F5522, MSP430F5521 only
(4) USB RAM can be used as general purpose RAM when not used for USB operation.

What can we learn from this?

- RAM starts at 0x2400, implemented in 2KB "Banks"
- Flash uses the address range 0x4400 to 0xFFFF
  - Code is written to flash starting from this address
- What about addresses 0x0010-0x0fff?

So what's the deal with addresses 0010h-0FFFh again?

These addresses are assigned to *peripherals:*

- Each peripheral has its own registers that are *mapped* as part of the memory that the CPU can access
- CPU can read or write data to peripherals just like any other memory address

This is how you make the CPU do I/O!

Input and Output

Consider this C code for a general-purpose system:

```
#include <stdio.h>

void main()
{
   char inKey = '-';  // declare variable named inKey
                        // and initialize to ASCII '-'

   while (inKey != 'X');
   {
         /* get character from keyboard */
         inKey = getchar();

         /* display character entered on screen */
         putchar(inKey);
   }
}
```

What is really happening here?

`getchar()` and `putchar()` are functions from the C standard library (part of stdio.h)

- Library for these functions is part of OS, and linked into code during build process
- These functions have always been part of the standard library because general purpose systems have always needed to use this type of I/O (eg. keyboard, screen, …)

Example: When a key is pressed, several layers below our little application, a byte has been placed on the microprocessor's data bus from a *port* connected to the keyboard:

Digital I/O: The Basics

Why do we use Digital I/O anyway?

Digital I/O is a method of directly inputting our outputting logic levels to the pins of the MSP430 Package.

You can use this functionality to implement almost anything!

- Simple devices: Buttons and LEDs
- Control signals for complex peripherals
- … and more!

Fun Facts about Digital I/O

- Eight independent, individually-configurable *ports*
- Ports 1-7 each have 8 configurable *pins*, and are thus 8 bits wide; Port 8 is 3 bits wide
- Each pin of each port can be configured individually as **input** or **output**
    - What makes something an input or an output? Inputs are devices from which you *read* a state, outputs require you to *write* a state to it.
- Ports 1 and 2 can generate *interrupts* on certain events, which are control signals that can be accepted or ignored by the MSP430
    - We will discuss these soon!
- Each port is controlled by **six** single-byte registers
- All the I/O port registers are *memory-mapped*, meaning that each register associated with a digital I/O port has a unique address in memory
    - How do you know what the addresses are? These are defined in msp430.h and msp430f5529.cmd. In these files, each register is given a specific name.
- Many more fun facts can be found in the User's Guide!

Digital I/O Registers (Part 1)

Each Digital I/O port has six registers to control its features. We will start by discussing three of them:

Direction Register (PxDIR)

Sets port pins as Input or Output
Set to 1 = Output
Set to 0 = Input

Input Register (PxIN)

Output Register (PxOUT)

The other registers are:

- **Function Select Register (PxSEL)**
- **Drive Strength (PxDS)**
- **Pull-up/Pull-Down Resistor Enable (PxREN)**

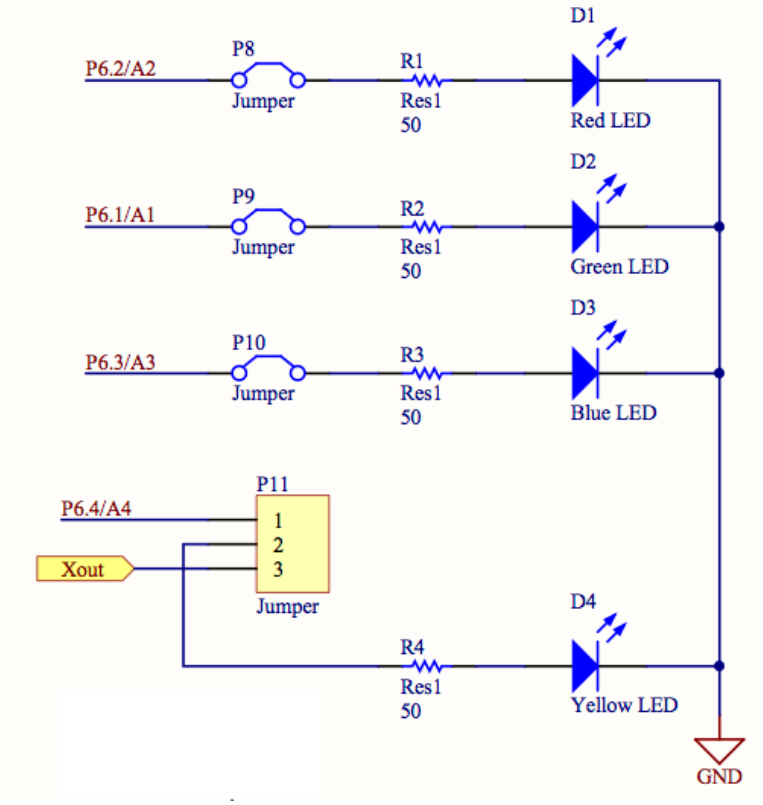
We will discuss these (using examples) later.

Conceptually, once you know which registers to use, using Digital I/O is pretty simple–all you need to do is read or write the desired values to the registers.

Digital I/O Concepts: Input or Output?

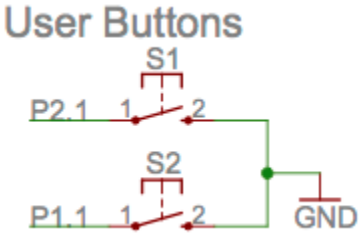
How do you know if something is an input or an output?

Consider the LEDs on our board. Are they inputs or outputs? What logic level lights the LED?



Example: Buttons

Consider the buttons on the Launchpad board:



Are the buttons inputs or outputs? **Inputs! We "read" values from them.**

Consider S1. What logic level indicates the button is pressed? What about when it is unpressed?

Module 5. Digital I/O

Topics

- More Digital I/O

About Digital I/O

Why do we use Digital I/O anyway?

Digital I/O is a method of directly inputting our outputting logic levels to the pins of the MSP430 Package.

You can use this functionality to implement almost anything!

- Simple devices: Buttons and LEDs
- Control signals for complex peripherals
- … and more!

Fun Facts about Digital I/O

- Eight independent, individually-configurable *ports*, named P1-P8
- Ports 1-7 each have 8 configurable *pins*, and are thus 8 bits wide; Port 8 is 3 bits wide
  Pins are referenced as P<port>.<pin>, eg. P1.4.
- Each pin of each port can be configured individually as input or output
- Most digital I/O pins share physical *package pins* with some other function on the device. This is called *pin multiplexing*.
- Each port is controlled by **six** single-byte **registers**

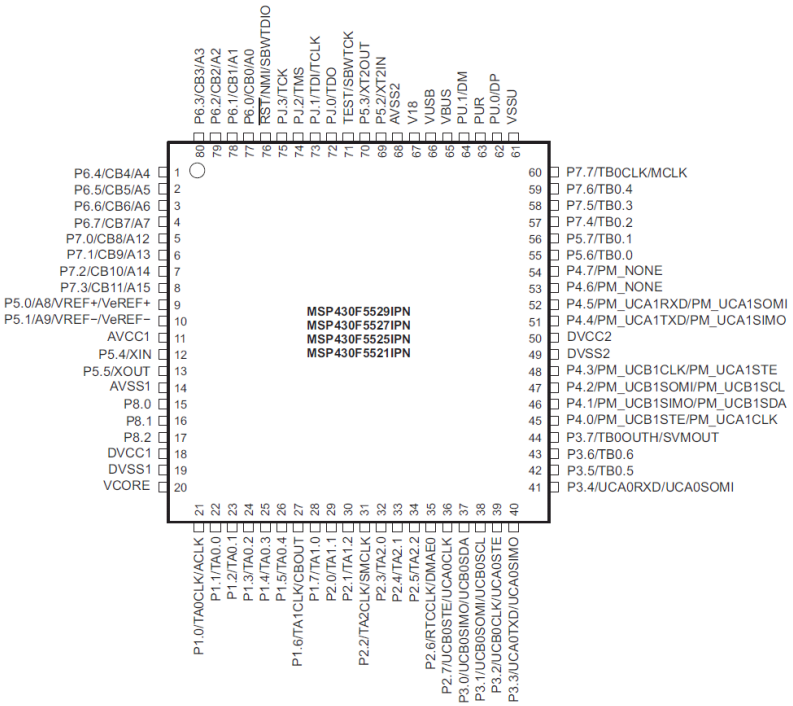
What is a register, anyway?

Register:

- Registers have addresses just like standard memory, so you can read and write to them
- Provide interface between hardware and software:
  - Reading from a register can get information about the hardware
  - Writing to a register can change how the hardware is configured, or send information to a component
- Functionality provided is defined by the hardware's design. When TI designs the MSP430, they define what registers are exposed to the programmer, which defines the functionality available on the chip.
- All the I/O port registers are *memory-mapped*: each register associated with a digital I/O port has a unique address in memory
  - How do you know what the addresses are? These are defined in the MSP430F5529 datasheet, as well as msp430.h and msp430f5529.cmd

Pins on the Microcontroller

Microcontrollers often pack lots of functionality in a small IC. However, the usage of all this functionality is limited by the physical pins on the IC *package*:



In order to maximize the usage of physical pins, most physical pins (also called “package pins”) are shared between multiple device functions.

Digital I/O Registers

The 6 registers controlling the digital I/O ports are as follows. **Each bit of the register controls the state for a specific pin.**

Function Select Register (PxSEL)

Selects the port pin for Digital I/O–remember multiplexing? This selects the function used on the pin.

Direction Register (PxDIR)

Sets port pins as Input or Output
Set to 1 = Output
Set to 0 = Input

Input Register (PxIN)

This is where the value input on the port appears (this is where you "read" the port)

Output Register (PxOUT)

This is where data to be output on the port should be "written"

Drive Strength (PxDS)
Pull-up/Pull-Down Resistor Enable (PxREN)

We will discuss these two (using examples) later.

Conceptually, once you know which registers to use, using Digital I/O is pretty simple–all you need to do is read or write the desired values to the registers.

Important Background: Bitwise manipulation

Because each bit in a register can control a different pin, we will make extensive use of C's *bitwise* operators (&, |, ~) to manipulate registers.

This is a very common practice when interacting directly with hardware!

Recall the truth tables for the bitwise operators AND (&), OR (|) and NOT (~):

A	B	Z = A & B
0	0	
0	1	
1	0	
1	1	

X	Y	Z = A | B
0	0	
0	1	
1	0	
1	1	

A	C = ~A
0	
1	

Where “X” is either 0 or

From these operators, **we can build a set of techniques for individually controlling specific bits in a variable while leaving the others unmodified**.

Common operations using bitwise operators

Setting individual bits to 1

We can do this by OR'ing a specific bit (or bits) with a 1. This is called "setting" a bit.

Setting individual bits to 0

We can do this by AND'ing a specific bit (or bits) with a 0. This is called "clearing" a bit.

"Selecting" specific bits from a variable

It is often necessary to check if certain bits of a field are set, or to only take the value of certain bits from a variable. We can do this by AND'ing a variable with only those bits that interest us set to 1–this is called *masking* bits.

You will use these techniques very frequently when working with digital I/O:

Configuration Example

Example: Configure Port 3 for Digital I/O with pins 1 and 0 as inputs and pins 7-4 as outputs.

There are two ways we can approach this problem:

An even better way: Lose the "magic numbers"

In this lecture, it's clear what the constants 0xF0 and 0xFC mean, but will you remember what's happening here 6 months from now? Probably not.

In C, as in many programming languages, it's good practice to avoid *magic numbers*, or hard coded numbers that appear in the code without explanation of their meaning or purpose. Instead, we can use constants to attach meaning to these values and allow them to be reused.

In this case, a set of constants for the individual bits are defined for us, we can just use them:

Name	Hex	Binary	Name	Hex	Binary
BIT0	0x01	0000 0001b	BIT4	0x10	0001 0000b
BIT1	0x02	0000 0010b	BIT5	0x20	0010 0000b
BIT2	0x04	0000 0100b	BIT6	0x40	0100 0000b
BIT3	0x08	0000 1000b	BIT7	0x80	1000 0000b

We can also combine these constants to refer to more than one bit:

Digital I/O Examples

Example 1: Input and output registers

Assume the following digital I/O pins are configured correctly. P3.1-0 are configured as inputs, and P3.7-4 are outputs.

A Hypothetical Specification:

Input: Read a 2-bit binary value *a* on P3.1-0

Output: Given *a*, set P3.7-4 based on the table:

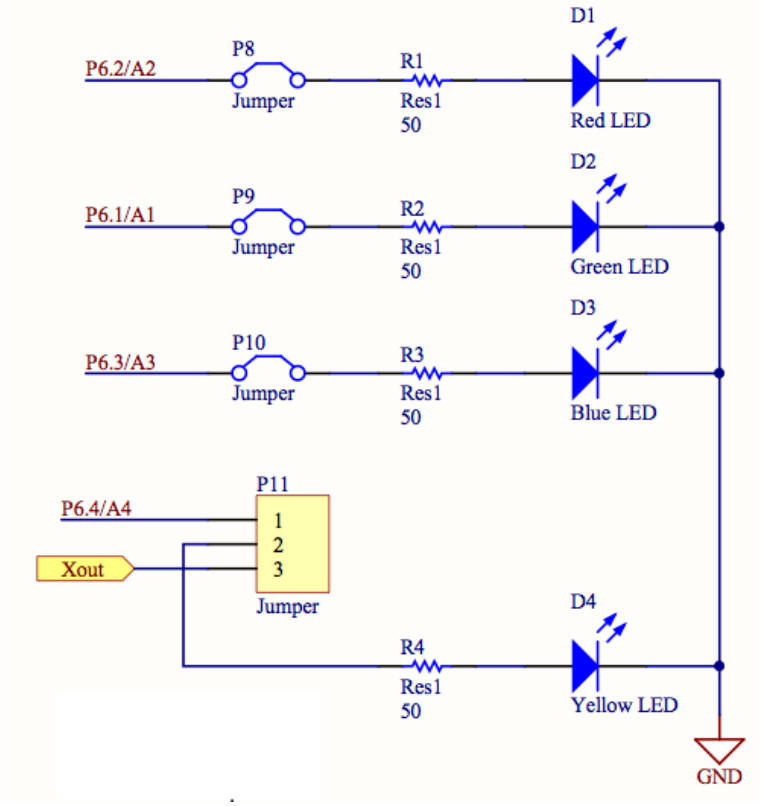
Input		**Output**			
a_1	a_0	z_3	z_2	z_1	z_0
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Digital I/O Concepts: Input or Output?

How do you know if something is an input or an output?

- If we are "reading" state from a hardware device, it is an **input**
- If we are "writing" or "setting" the state of a device, it is an **output**

Consider the LEDs on our board. Are they inputs or outputs? What logic level lights the LED?



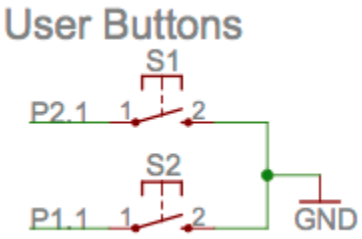
In an application program like our demo project, we use the digital I/O ports repeatedly to use the buttons and LEDs. In these programs, it's a good idea to *wrap* the functionality for hardware components into useful functions.

See `setLeds()` in the demo project for an example!

Dealing with Inputs

As we discussed briefly last lecture, inputs may require some special handling.

Consider the buttons on the Launchpad board:



Digital I/O Registers (cont.)

Pull-up/Pull-Down Resistor Enable (PxREN)
Activates pull-up or pull-down resistors when a pin is configured as a digital input.

What controls whether to use a pull-up or pull-down resistor?
The output register (PxOUT) is actually re-used for this purpose!
Set the appropriate bits to 1 for pull-up resistors, and to 0 for pull-down. See p. 408 of the user's guide for details.

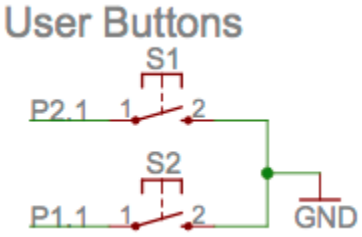
You will also see one more Digital I/O register…

Drive Strength (PxDS)
Controls "drive strength", or amount of current that is sourced from the pin when used as an output. We will always use the default setting for this.
Set to 0 = Reduced drive strength (default) Set to 1 = Full drive strength

Important to note: all I/O pins have *limits* on the amount of current that can pass through them (usually on the order of milliamps). See the MSP430F5529 datasheet for details.

As an embedded developer, it's always important to remember the requirements of the hardware as well as the software!

Example: Launchpad Buttons (cont.)



We can configure these buttons as inputs and using pull-up resistors, as follows:

```
void initButtonsLecture(void)
{
      // Configure buttons as outputs using internal pull up resistors
      // Button 1:  P1.1
      P1SEL &= ~BIT1;
      P1DIR &= ~BIT1;
      P1REN |=  BIT1;
      P1OUT |=  BIT1;

      // Button 2:  P2.1
      P2SEL &= ~BIT1;
      P2DIR &= ~BIT1;
      P2REN |=  BIT1;
      P2OUT |=  BIT1;
}
```

Note that the buttons are on different ports, so we need to configure them separately!

```
// Read buttons S2 and S1 and return their state in the
// lower two bits of the return value such that
// ret = 0 0 0 0    0 0 S2 S1
unsigned char readButtonsLecture(void)
{









}
```

Polling

How can you monitor and use your properly-configured digital I/O functions?

- … by repeatedly checking if the button status has changed!
- Since this just involves reading a memory address, it is very fast to execute (on the order of microseconds!)

Example:

```
// Inside your demo project...
while(1)
{
    ret_val = readButtons();
    setLeds(~retVal);
}
```

Another, similar example:

```
ret_val = 0x0f; // Default value for all buttons unpressed
while(ret_val == 0x0f)
{
    ret_val = read_buttons();
}

setLeds(~ret_val);
```

Without a delay, this loop executes in microseconds!

This process is called **polling**–we constantly check the buttons and do something when they change.

At the moment, it's all the program needs to do, so it's fine. But what if we wanted to perform more tasks? What if we wanted the processor to sleep while it was waiting?

Module 6. Review: The Story So Far

Writing Numbers

Be comfortable with standard conventions for writing numbers used in class and in C:

- Decimal: 42
- Hex: 0x2A or 2Ah
- Binary: 0010 1010b

You should be able to convert from binary to hex easily (and vice versa)!

Basic C Instructions and Syntax:

>> Know layout of C source file (Lecture 2)
>> Some Data types (as they are defined in CSS for the MSP430)

```
// What are the sizes for each datatype?
    int             a;        //
    float           b;        //
    char            c;        //
    unsigned int    d;        //
    long int        e;        //
    double          f;        //
    int         arr[5];       //
```

Arrays: Are blocks of memory where multiple values are stored contiguously. Storing elements successively (in order) makes it easy to access each element given its index.

Standard C Operators:

Math: + - * / = % (modulo)
Unary: ++ -- (also |= &= += etc.)
Relational and Logical: > >= < <= == != && | |
Bitwise: & (AND) | (OR) ^ (XOR) >> (R shift) << (L shift) ~ (NOT)

Quick Questions:

```
int a = 0x0101;
int w = a + 12;
int x = a << 1;

unsigned char b = 0xff;
unsigned char y = b + 2;

int d = 42;
int z = d / 10;
```

1) What value is assigned to x?
a) 0x0202 b) 0x1010 c) 0x2020 c) 0x0080

2) What value is assigned to y?
a) -1 b) 0 c) 1 d) 256

3) What value is assigned to z?
a) 2 b) 4 c) 4.2 d) 10

Decisions, looping, etc:

```
if  (kk > 100) {
   kk = 0;
} else {
   z = 2*z+kk;
   kk++;
}

while  (j < 100) {
   /* Body of  loop  */
   j++;
}

for (i = strt; i < end_pt; i++) {
  /* Body of loop. Do something  */
}
```

--> The ***"Forever Loop"***

```
while  (1) {
   /* Body of loop. Do something */
}
```

Basic Structure of a C program

```
#define  MAX_SZ  100;

 // Determines max value of an array
 unsigned int arrayMax(unsigned int* in_arr, int num_pts);

 void main()
 {
     unsigned int     big[MAX_SIZ];
     unsigned int     maximum=0;
     unsigned int     i, other_val;

     /* Do some stuff */
    i = 0;
     while (i < MAX_SZ)
    {
         big[i] = (i % 10);
         i++;
    }
     maximum = arrayMax(big, MAX_SZ);
    /* Do more stuff */

 }  // end of main()
```

Quick Questions:

1) How many times does the while loop execute?

a) 99 b) 100 c) 101

2) To what value is big[47] assigned?

a) 40 b) 0.47 c) 7 d) 470

3) What is the range of *valid* indices for the big array?

a) big[1] to big[100] b) big[0] to big[99]
c) big[0] to big[100] d) big[0] to big[9]

4) To what value is maximum assigned?

a) 99 b) 100 c) 10 d) 9

Data Representations (HW #1):

>> Integer representations:
--Unsigned, sign-magnitude, two's complement and BCD

>> Expect Conversion Between Bases and Formats!

Unsigned integers = all bits used to convey magnitude (whole numbers) – For *n* bits, values run from 0 to $2^n - 1$ (i.e. *N*=16, 0 to 65535)

1026 = 00000100 00000010b = 0402h

Sign Magnitude integers = *n*-1 bits used to convey magnitude with "most significant bit" or MSB used for sign (0 = +, 1 = -). For *n* bits, values run from $-2^{(n-1)}-1$ to $2^{(n-1)}-1$

1026 = 0000 0100 0000 0010b = 0402h

-1026 = 1000 0100 0000 0010b = 8402h

** Has 2 representations of 0 >>> +0 and -0!

Two's Complement integers = Common format for signed integers (int). For *n* bits, values run from $-2^{(n-1)}$ to $2^{(n-1)}-1$. (i.e. *n*=16, -32768 to 32767). Used by C.

Positive numbers: Same as Unsigned

1026 = 0000 0100 000 00010b = 0402h

Negative numbers (ONLY!!): ***Encode*** magnitude, ***Complement*** each bit, ***Add*** 1

-15 = 0000 0000 0000 1111 = 15
1111 1111 1111 0000 complement
+1
1111 1111 1111 0001 = 0FFF1h = -15 in two's complement

Binary Coded Decimal = Each decimal digit expressed in binary nibble

367 = 0000 0011 0110 0111b

Fractional Number representations:

Fixed point: Binary radix point assigned a fixed location in byte (or word)

$0101.1010 = 5 + 2^{-1} + 2^{-3} = 5.625$

Precision is function of number of fractional bits assigned
--> 4 fractional bits = 2^(-4) = 0.0625 = smallest fraction

Floating Point (IEEE Standard) : Used to better approximate real valued decimal values to a prescribed number of decimal places

Single Precision (32 bits): S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF

Value = $(-1)^S$ $2^{(E-127)}$ * (1.F)

Why are floating point operations computationally expensive?

For the exam, you do not need to remember how to convert to/from floating-point, but you should understand what it is and how it differs from fixed-point.

Character Representations

ASCII: Standard for representing characters in Roman alphabet and some control characters

- You will have an ASCII table on the exam. Know how to read one and when you need it!

Quick Questions:

1) The decimal equivalent of unsigned integer 8002h is
a) 32770 b) 65538 c) -2 d) 16386

2) The decimal equivalent of two's complement integer 8002h is
a) -2 b) 32770 c) -32766 d) -65538

3) The decimal equivalent of two's complement integer 0002h is

a) -2 b) 32770 c) 2 d) -65538

4) The decimal equivalent of BCD integer 8002h is

a) -2 b) 32770 c) 8002 d) 2008

Little Endian: The MSP430, like Intel processors, is "Little Endian" (HW1)

-- The lower byte of each 16 bit word is stored first then the higher byte
"*Low Byte, High Byte*"

-- For double words the lower word is stored first then the upper word

Ex: How 65340 decimal = 00 01 00 04h is stored in memory at address 0400h

Little Endian

Address	***Byte Value***
02403h	00h
02402h	01h
02401h	00h
02400h	04h
....	....

A memory dump from CCS shows contents of addresses from left to right starting at 02400h

`02400h = 04 00 01 00` ... <= Bytes appear "out of order" when read left to right

Big Endian: Many other RISC processors

-- The higher byte (big end) of each 16 bit word is stored first then the lower byte

BIG Endian

Address	***Byte Value***
02403h	04h
02402h	00h
02401h	01h
02400h	00h
....	....

A memory dump from a big endian processor (also left to right)
`02400h= 00 01 00 04`... <= Bytes appear "in order" when read left to right

Network Byte Order = BIG ENDIAN!!!

Microprocessor Systems Architecture:

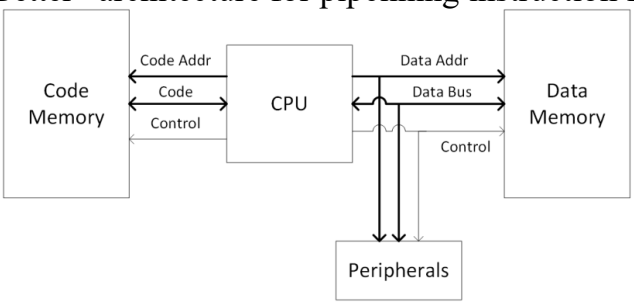
>> General Computing Hardware/Software Hierarchy

Applications
Operating System = User Interface
System SW = Interface to HW
HW Layer = CPU, Mem., peripherals

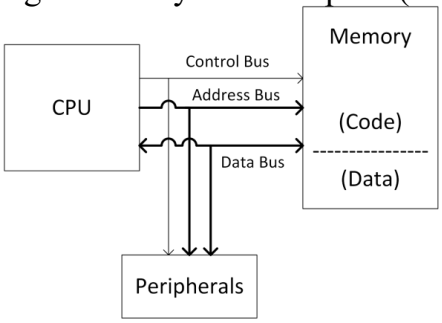
>> Gets "squashed" in an embedded system...

Applications
Maybe some System SW functions
HW Layer = CPU, Mem., peripherals

Harvard Architecture – Separate memory address spaces (and busses) for code and data ("Better" architecture for pipelining instruction fetches)



Von Neumann Architecture – Single memory address space (and bus) for code & data



>> MSP430x55xx uses ***Von Neumann*** architecture

>> We're using ***MSP430F5529***
-- 128 KB Flash memory (code)
-- 8 KB RAM (data) + 2 kB USB RAM
-- LCD controller
-- Hardware multiply, UART, and a slew of other peripherals (Timers, ADC, comparator, general digital IO ports...)

Memory Organization:

>> Memory = group of sequential locations where binary data is stored

-- In MSP430, a memory location holds 1 byte

-- Each byte has unique address which CPU uses to read to and write from that location
-- Multibyte data is stored Little Endian!

-- *2 types of memory: Volatile and Non-volatile*
RAM = 8KB = DATA memory = Volatile
FLASH = 128KB = CODE memory (primarily!) = Non-volatile

Memory Operations

- Read and Write: retrieving or writing DATA to/from RAM (under programmer control)
- Fetch: retrieving of instruction from CODE (Flash) memory (automatic CPU function)

>> Flash is NOT byte writable!
-- Must be erased in multi-byte (e.g. 512 byte) segments
>> A flash write cycle takes much longer than read cycle

MSP430 is 16 bit Microcontroller
>> 16 bit word size = 16 bit internal registers
>> Also has 20 bit address bus (can access up to 1 MB = 2^{20} addresses)
>> Know Memory Map for MSP430x5529x Processors (from HW)
-- Addresses for RAM & FLASH, (good thing to have in notes!)
>> Know how to figure memory addresses

Memory Mapped I/O

What does it mean for I/O to be *memory-mapped*?

Quick Questions:

1) The long int i = 0x00081230 is stored in memory by a microprocessor as

Address	*Contents*
0213h	30h
0212h	12h
0211h	08h
0210h	00h

The microprocessor must be

a) Little Endian b) Big Endian c) Running Linux d) Running Windows 10

2) In the MSP430F5529, the RAM is

a) non-volatile system memory b) volatile data memory

c) non-volatile code memory d) consists only of the 16 CPU registers

3) In the MSP430F5529, the FLASH memory is

a) non-volatile code memory b) volatile data memory

c) volatile code memory d) not available in this model

MSP430F5529 Basic Digital I/O (HW3-4):

>> Eight independent, individually configurable digital I/O ports
-- Ports 1-7 are 8-bit wide and Port 8 is 3 bits wide

>> Each pin of each port can be configured individually as an input or an output

>> Each pin of each port can be individually read or written to

Function Select Register: Sets function of each pin in the port (i.e. P4SEL)
-- Bit = 0 = Selected for Digital I/O
-- Bit = 1 = Not selected for digital I/O (multiplexed pin functions)

Direction Register: Sets direction of each pin in the port (i.e. P2DIR)
-- Bit = 0 = Corresponding pin is an ***Input***
-- Bit = 1 = Corresponding pin is an ***Output***

Input Register: Where input to the port is read from (i.e. P2IN)
-- Bit = 0 = Logic low
-- Bit = 1 = Logic high

Output Register: Where data to be output from the port is written (i.e. P5OUT)
-- Bit = 0 = Logic low
-- Bit = 1 = Logic high

Drive Strength: Sets drive strength of port (we will usually leave as default)
--Bit = 0 = reduced drive strength (default)
--Bit = 1 = full drive strength

Pull-up/down Resistor Enable: Enable internal pull-up resistors (can be used for inputs)
--Bit = 0 = Not enabled (default)
--Bit = 1 = Enabled (see User's Guide)

>> All I/O port registers are ***memory mapped.*** Register names defined in ***msp430x4xx.h***
(Read from and Write to defined names as if writing to C variables...)

>> *Polling*: Repeated checking of IO ports to see if they have data or need servicing (usually inside main loop)

```
#include "msp430.h"
#include <stdlib.h>

void  configPort()
{
   P5SEL = 0x00;
   P5DIR = (BIT7|BIT5|BIT3|BIT1);
}

void main()
{
   configPort();

   while (1)
   {
     char in = P5IN;
     P5OUT = (in & 0x55) << 1;
   }
}
```

a) Which port(s) and which pins are being used as digital inputs?

b) Which port(s) and which pins are being used as digital outputs?

c) Assume that the port 5 input register holds the value 6Dh. What value is written to the port 5 output register?

Module 7. Intro to Clocks and Timers

Clocks

A microcontroller and its peripherals are just sequential logic circuits. Remember that sequential logic circuits need a *clock signal*. Before a CPU can operate, it must have power, a clock signal, and ground.

What does a clock signal look like?

Clocks on the MSP430: The Unified Clock System (UCS)

Microprocessors usually allow you to configure the clocks used by the system. On the MSP430, this tasks is handled by the **Unified Clock System** (UCS), which is billed as "full featured and capable" (read: complex and confusing)!

Like most microcontrollers, the MSP430 has a variety of configurable *clock sources* and clock *signals*:

There are two types of clock sources:

- External sources:

- Internal sources:

Why is all of this configurability important?

The MSP430F5529 has 5 possible clock sources:
XT1CLK

XT2CLK

DCOCLK

REFOCLK
VLOCLK

These provide 3 clock signals to the CPU and peripherals:
ACLK - Auxiliary Clock:

MCLK - Main or Master Clock:

SMCLK - Sub-main Clock:

The three clock signals are *software selectable*, meaning that the user can configure the clock sources and speeds for the CPU and peripherals **at runtime**.

Configuring the UCS: The Gist

In general, configuring the UCS boils down to connecting the various clock sources (XT1, XT2, DCO, etc.) to the 3 clock signals (ACLK, MCLK, SMCLK):

In addition, you also need to configure some parameters for the sources (like the DCO), and the signals (like clock dividers).

Configuration notes

Configuring XT1 and XT2

The low frequency and high frequency crystals XT1 and XT2 are connected via pins on the MSP430. On the MSP430F5529, these pins are multiplexed with P5.4-5 (for XT1) and P5.2-3 (for XT2).

If you want to use XT1 or XT2, you need to configure these pins for **function mode** (as opposed to digital I/O mode) by setting their corresponding bits in P5SEL to 1:

```
P5SEL |= (BIT5|BIT4|BIT3|BIT2);
```

In our lab, this is already done for us in the template in the `configDisplay` function.

The DCO (Digitally-controlled oscillator)

The DCO is a *digitally-controlled oscillator*, which means that you can configure its frequency in software. The UCS module provides a frequency-locked loop (FLL) to stabilize the DCO. The frequency for the DCO is defined by the following formula:

Default clock configuration

After decoding the default register values, we know that **by default**, SMCLK = MCLK, and both use DCOCLK as their source. In addition, ACLK = XT1CLK (if enabled). From this, we can conclude that the default clock settings are as follows:

- ACLK (Auxiliary clock) =
- MCLK (Master/CPU clock) =
- SMCLK (Sub-main clock) =

In our labs, we will keep it simple and use these default settings! These are important. Remember them!

Buttons Challenge (from Homework)

What is wrong with this program? What can we do about it?

Module 8. Timers: Theory and Practice

Topics

- Intro to Interrupts
- Intro to Timers

But first… what is a timer?

Most microcontrollers have timers in some form. Timers can be used to generate **interrupts** at particular intervals, generate PWM signals, measure frequency of input signals, and more! In this course, we will focus on the generation of timer interrupts, which tell the CPU that a certain amount of time has passed.

Fundamental timer counting modes

Most timers have a number of *counting modes*:
Unidirectional mode (called "Up mode" on the MSP430)
Count from 0 to a *programmer set* maximum count value (which we call MAX_CNT).

Continuous mode
Count from 0 to full count of timer (8, 12, 16 bits, etc.) For a 16 bit timer, this means:

Up/Down Mode
Counts from 0 to *programmer set* maximum count, then back down to zero

In each mode, most timers (like those on the MSP430) will trigger an interrupt when the count transitions back to 0.
Most timer peripherals have two "operating modes", which control how they use the counter:

- **Capture mode**: Records the counter value when a certain input changes
- **Compare mode**: Performs an operation when the counter value reaches a certain value

Interrupts

Interrupt: A signal sent to the CPU from a peripheral or external source

- Typically, an interrupt is either a request for the CPU to do something or a notification that the peripheral has something (ie, data) available for the CPU to use.
- The CPU can choose to accept (or to "service") the interrupt, or ignore it. Certain interrupts, called "non-maskable interrupts" (NMIs) cannot be ignored.

Interrupts on the CPU are handled by a special function called an **Interrupt Service Routine (ISR)**.

Note: Interrupts are not just for timers!
Many peripherals on the MSP430 can generate interrupts for different reasons:

What does the CPU do when it receives interrupts?

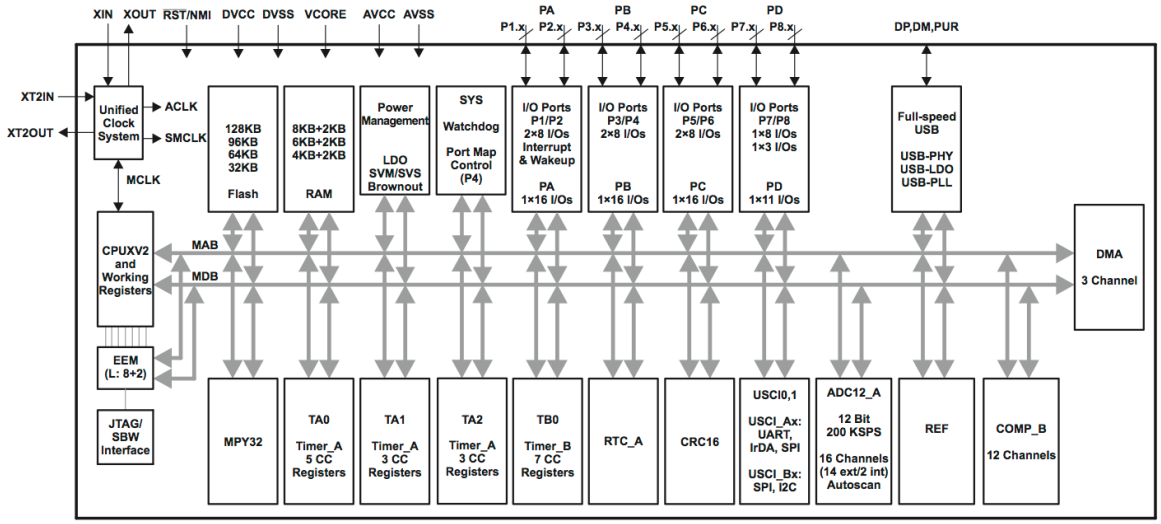
Arrival of interrupts is *asynchronous* to the program's execution.

How do interrupts work internally?

Peripherals that can trigger interrupts so do by issuing a request to the CPU's *interrupt controller*, often over a dedicated wire called an interrupt request line. The interrupt controller decides how the CPU processes each interrupt:

Timers on the MSP430

The MSP430F5529 has a number of timer peripherals, as shown in the system block diagram:



MSP430s have two main types of timers, Timer A, and Timer B; both types function in very similar ways, but have some subtle differences. Each chip can have multiple Timer A's and B's, as shown in the block diagram.

The MSP430F5529 has the following timers:

Timer B: Has 7 capture/compare units, can generate PWM signals

Timer A0: Multiple capture compare modules, can generate PWM

Timer A1: Functionally the same as A0

Timer A2: 3 capture compare registers

Additionally, the MSP430F5529 has the following other peripherals that contain timers:

- A Basic Timer, which has some real-time clock features
- The **Watchdog timer** (WDT)
  - When the WDT is on, it must continuously have its count reset within the program
  - If the count reaches zero, it **resets the MSP430!**

Why does the WDT exist? To prevent your program from getting stuck in some kind of unrecoverable state. We don't want to deal with the WDT in your labs, which is why the first line of every program we write is:

```
WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
```

This disables the watchdog timer. If you wait too long to stop the watchdog timer, your program will reset only a few milliseconds after startup!

Configuring timers on the MSP430

Like the UCS module, timers are highly configurable!

We will stick to the basic configurations and **only add complexity when we need it**! This is a good design practice, and also makes our lives easier!

Timer A has the following registers:

17.3 Timer_A Registers

Timer_A registers are listed in Table 17-3 for the largest configuration available. The base address can be found in the device-specific data sheet.

Table 17-3. Timer_A Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	TAxCTL	Timer_Ax Control	Read/write	Word	0000h	Section 17.3.1
02h	TAxCCTL0	Timer_Ax Capture/Compare Control 0	Read/write	Word	0000h	Section 17.3.3
04h	TAxCCTL1	Timer_Ax Capture/Compare Control 1	Read/write	Word	0000h	Section 17.3.3
06h	TAxCCTL2	Timer_Ax Capture/Compare Control 2	Read/write	Word	0000h	Section 17.3.3
08h	TAxCCTL3	Timer_Ax Capture/Compare Control 3	Read/write	Word	0000h	Section 17.3.3
0Ah	TAxCCTL4	Timer_Ax Capture/Compare Control 4	Read/write	Word	0000h	Section 17.3.3
0Ch	TAxCCTL5	Timer_Ax Capture/Compare Control 5	Read/write	Word	0000h	Section 17.3.3
0Eh	TAxCCTL6	Timer_Ax Capture/Compare Control 6	Read/write	Word	0000h	Section 17.3.3
10h	TAxR	Timer_Ax Counter	Read/write	Word	0000h	Section 17.3.2
12h	TAxCCR0	Timer_Ax Capture/Compare 0	Read/write	Word	0000h	Section 17.3.4
14h	TAxCCR1	Timer_Ax Capture/Compare 1	Read/write	Word	0000h	Section 17.3.4
16h	TAxCCR2	Timer_Ax Capture/Compare 2	Read/write	Word	0000h	Section 17.3.4
18h	TAxCCR3	Timer_Ax Capture/Compare 3	Read/write	Word	0000h	Section 17.3.4
1Ah	TAxCCR4	Timer_Ax Capture/Compare 4	Read/write	Word	0000h	Section 17.3.4
1Ch	TAxCCR5	Timer_Ax Capture/Compare 5	Read/write	Word	0000h	Section 17.3.4
1Eh	TAxCCR6	Timer_Ax Capture/Compare 6	Read/write	Word	0000h	Section 17.3.4
2Eh	TAxIV	Timer_Ax Interrupt Vector	Read only	Word	0000h	Section 17.3.5
20h	TAxEX0	Timer_Ax Expansion 0	Read/write	Word	0000h	Section 17.3.6

We will use a subset of these registers:

- **TA2CTL**: Control register for Timer A2
- **TA2CCTLx**: Control register for a capture/compare block
- **TA2CCRx**: Capture/compare register (data register)

We will discuss how to use these registers in detail using an example.

Timer configuration example: A stopwatch

Example: Implement a stopwatch that measures seconds and hundredths of seconds on our development board.

First, how do we measure the passage of 0.01 seconds? By counting clock ticks. We will do this by configuring a timer for the job. But how do we start?

In our labs, we can break any problem involving timers into a set of steps:

1. Select a timer to use: How about Timer A2?

2. Map desired behavior to an operating mode (Up, Continuous, Up/Down)

3. Select a clock source and configure registers appropriately

How do we configure the TimerA2 control registers?

Using this information, we can write the register configuration:

Parameters we know	Relevant register field

We can use this to write:

```
TA2CTL =

TA2CCR0 =

TA2CCTL0 =
```

Step 4: Write Interrupt Service Routine (ISR) and enable interrupts

… how do we write interrupts in our code, anyway?

An ISR for Timer A2 looks like this:

```
// Example syntax for TimerA2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{
    // Do something
    // ...
}
```

In addition, in your `main()` you must **enable interrupts** to tell the CPU to handle them:

```
// Using pre-defined macros in msp430.h

_BIS_SR(GIE);          // Global interrupt enable
// ... OR ...
__enable_interrupt();
```

(The above macros are equivalent. You will see both of them in example code and notes in this class.)

Back to the example: what does it mean when we get an interrupt from Timer A2?
What should the ISR do?

Each interrupt means that the timer has reached MAX_CNT, meaning that 328 ticks of ACLK ~= 0.01s have elapsed.

Thus, the ISR should count how many interrupts have occurred… and do nothing else:

```
// Global count of clock ticks
unsigned long int timer = 0;

#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{



}
```

Important note: ALWAYS keep your ISRs short. Why?
What happens if your ISR hasn't completed before the next ISR arrives?

In general, NEVER do any of the following in an ISR:

- Write to the display
- Flush the display
- Do floating point math
- Call expensive functions like `sin()` or `sprintf()`

Examples: Using the timer variable

Stopwatch

Now that our ISR is properly configured, what does the variable timer represent? How do you use it to actually display the time?

The timer represents the number of 0.01 second intervals that have elapsed since Timer A2 was started. To use it, we need to convert this to minutes and seconds in order to display it.

How do we do this? Note that we want to do it using *integer math*, since floating point is slow and we eventually want to put this information on the display.

Timer accuracy

How accurate will our stopwatch be? Is that accuracy acceptable?
The duration of one ACLK tick = 1/32768 Hz = 3.05e-5 seconds

Since our stopwatch will run slow, how long until it is off by 0.01 second?
Here is how we can add a leap count for this example:

```
#pragma vector=TIMER2_A0_VECTOR
__interrupt void TIMER_A2_ISR(void)
{









}
```

When using leap counting, we can use the following general rule:

In our example, is our leap counting solution perfect? For how long will it be accurate to 0.01 sec?

Configuration example: 1ms resolution

What if we wanted our stopwatch to have 1ms resolution? What would need to change?
We would need to change the value of MAX_CNT. Would we need leap counting?

Configuration example

Configure Timer A2 for 0.5 sec resolution. Do you need to use leap counting?

Another configuration example

What if you wanted 0.0001 second resolution? What do we do now?

Module 9. Analog to Digital Conversion

Topics

- More on timers
- Starting Analog to Digital Conversion

Warmup: Analyzing a timer configuration

1. What is the period of the timer with the configuration below? (How often are interrupts generated?)

```
void runtimerA2(void)
{
    TA2CTL = TASSEL_2 | MC_1 | ID_3;
    TA2CCR0 = 32767;
    TA2CCTL0 = CCIE; // Enable timer A2 interrupt
}
```

2. The ISR for the timer above increments a counter called `timer` on each interrupt. If `timer = 2447`, how much time has elapsed since the timer was started?

Analog to Digital Converters (ADCs)

Analog to Digital Converters (ADCs), or A/D converters, have become ubiquitous in embedded applications.

- ADCs return a binary code to represent a measured voltage from within a fixed range of voltages
- Small voltages return "low valued" codes, greater voltages return "larger" codes

For example, within the range of 0–3V, a 10-bit ADC could return codes like these:

```
00 0000 0000b = 000h = 0d
01 1111 1111b = 1FFh = 512d
11 1111 1111b = 3FFh = 1023d
```

- It is **very** likely that you will use an ADC when you take ECE2799, do your MQP, or work on a robotics project!

ADC Concepts

When an analog signal is “read” by an ADC, the analog value of the signal is *sampled* to obtain a k-bit digital representation of that value at that time.

Realities of ADCs

No ADC is perfect–there will always be some error between the analog voltage and the one measured with the ADC. There are several reasons for this, including:

- Output codes are "quantized": the closest ADC code will differ somewhat from the analog voltage, depending on the resolution
- Our MSP430 uses a "sample and hold" type of ADC, which means the analog circuitry that samples the waveform can "hold" the analog value at a certain level—this means that it might miss certain changes in the waveform
- Transients from switching circuitry inside the ADC can affect the output code, which may introduce some non-linearity in the output values

When sampling at faster rates, these effects tend to get worse!

We will not deal with these issues much in this class, but it is important to know they exist.

ADCs on Microcontrollers

Small ADCs like the 12-bit "Sample and Hold" type ADC on our MSP430 often come standard on small microcontrollers.
Are they any good?

These ADCs are best suited to measuring from sensors with low to moderate data rates with a *fixed dynamic range*. Some examples:

These small ADCs are likely not suitable for applications with higher data rates or a larger dynamic range. Examples:

As always, however, the application will determine the type of ADC you need!
What is *dynamic range*, anyway?

Key Concepts for using Analog to Digital Converters

(or performing any measurements)

1. Full Scale Range (FSR): The maximum range of analog values that can be represented

This is defined as the total range of voltages between V_{REF+} and V_{REF-}

2. Resolution (for a single bit): The smallest change in value that can be measured

You can think of this as the "value of 1 bit" in an output code.

3. Dynamic range: Ratio of largest to smallest values that can be measured

The dynamic range is usually expressed in decibels (dB), and can be computed as follows:

Thinking about data representations

As an embedded systems engineer, you get to decide how to make your sensors interface with the ADC! Knowing how your external sensor works and how to "map" it to the ADC you're using is as critical as knowing how to make the MSP430 read the value!
Here’s a way to think about how sensor measurements are represented as digital values:

Example: Current sensor

You can make a simple digital current meter by measuring the voltage across a small sensing resistor.

Can we use the ADC12 on the MSP430 to measure current in the range 0–1A with 1mA accuracy? How about to 0.1mA accuracy?

For now, let's assume we have an FSR of 2.5V.

On the MSP430: Using the ADC12

Our MSP430 provides includes a 12-bit ADC, called the ADC12.

About the ADC12

- 16 channel, 12-bit sample-and-hold ADC
- Maximum sample rate of 200k samples/second
- 12 External analog inputs A0-A7, A12-A15; shared with Digital I/O ports 6 and 7
- You configure and use them by setting values in various control registers

Overall ADC operation

An ADC's job is to perform a *conversion* by sampling an analog voltage into a digital value.

The ADC12 has the following components:

- Inputs from analog input channels
- Core unit to perform conversions
- *Core configuration registers* that configure how the conversion happens
- Can define multiple *channels* to perform multiple conversions at once
    - *Memory control registers* that configure how each channel should be converted
    - *Memory registers* that store the conversion results for each channel

ADC12 Control and Data registers

You can find the ADC12 register definitions in the MSP430 User's Guide (Ch. 28).

Core configuration registers

The ADC12 conversion core is configured using ADC12CTL0 and ADC12CTL1.

ADC12CTL0 controls the following options:

- Sample and Hold Time (ADC12SHT1x, ADC12SHT0x): Controls sampling period
- Multiple sample conversion method (ADC12MSC)
- **Reference voltages (ADC12REF2_5V and ADC12REF_ON)**
- **ADC12ON bit**: Turns on the ADC12! (It's off by default!)
- **Enable conversions (ADC12ENC)**: Must be set to 1 before ADC will perform conversions! When set to 0, ADC can be configured.
- **Start conversion (ADC12SC)**: Starts a conversion!
- Overflow/conversion time interrupt enables (ADC12OVIE, ADC12TVIE)

ADC12CTL1 controls the following options:

- Conversion start address (ADC12STARTADDx)
- Sample and hold source select (ADC12SHS):
- Sample and hold pulse mode select (ADC12SHP): Always set this to 1
- Invert signal sample and hold (ADC12ISSH)
- ADC12 clock divider (ADC12DIVx): Typically use 1
- ADC12 clock source select (ADC12SSELx):
- Conversion mode select (ADC12CONSEQx): Can select single, multi-channel, or repeated conversions
- **ADC12 busy bit (ADC12BUSY)**

Results from each channel are stored in the low 12 bits of one 16 bit **Conversion Memory Register (ADC12MEMx)**.

Each memory register has a corresponding **Conversion Memory Control Register (ADC12MCTLx)**.

Each **ADC12MCTLx** controls one channel on which a conversion can occur. The conversion parameters for channel x is controlled by Memory Control Register x, and the result gets stored in memory register x.

Each ADC12MCTLx controls the following options:

- **Reference voltage select (ADC12SREFx)**: Important settings are as follows:

- **Analog input channel select (ADC12INCH_x)**:

- **End of Sequence (EOS)**: Set to 1 if this channel is the end of a sequence of channels. Used for multi-channel conversions.

So, as a programmer, what do you need to use the ADC12?

ADC configuration: Key steps

Step 0: Disable the ADC for configuration

- Before you can modify any ADC12 register settings, conversions must be disabled by setting ADC12ENC = 0.

Step 1: Select ADC core behavior (ADC12CTL0 and ADC12CTL1)

- Set clock source and divider
- Configure sample and hold behavior
- Select trigger source (ADC12SHS)

- **Reference voltages**

Step 2: Select conversion mode for your application

- Configure using ADC12CONSEQx in ADC12CTL1 register
- There are four possible conversion modes:

Table 28-2. Conversion Mode Summary

ADC12CONSEQx	Mode	Operation
00	Single-channel single-conversion	A single channel is converted once.
01	Sequence-of-channels (autoscan)	A sequence of channels is converted once.
10	Repeat-single-channel	A single channel is converted repeatedly.
11	Repeat-sequence-of-channels (repeated autoscan)	A sequence of channels is converted repeatedly.

Step 3: Select input channel(s)

- What analog inputs do we need to read?
- Configure using ADC12INCHx in ADC12MCTLx registers

- Analog inputs A0–A7 and A12–A15 are external analog inputs–these are **multiplexed** with Digital I/O pins on Port 6 and Port 7!
    - To use them, we need to configure the digital I/O pins for **function mode**!
      Ex. P6SEL | BIT7|BIT6;

- Analog inputs 8, 9, 11 are connected to the various on-chip reference voltages–you can use these to monitor the "health" of the microcontroller
- Input channel 10 is connected to an internal temperature sensor (ADC12INCH_10)

Step 4: Enable ADC interrupts, if desired (ADC12IE register)

- Using interrupts is NOT required, but useful if you are doing repeated measurements
- Also need to write ISR

Step 5: Enable ADC and start conversions

- Need to re-enable ADC so it will perform conversions (opposite of step 0)
- Start conversion process by setting ADC12SC.
- If not using interrupts, need to **poll** ADC12BUSY bit in ADC12CTL1 until conversion has finished!

Example: Current measurement sensor

You can make a simple digital current meter by measuring the voltage across a small sensing resistor. Can we use the ADC12 on the MSP430 to measure current in the range 0–1A with 1mA accuracy? (Yes!) How about to 0.1mA accuracy? (No!)
Assume we have an FSR of 2.5V and the analog voltage is connected to input A0.

What parameters do we need?

```
// Current sensor conversion example
void config_adc(void) {
  /* ***** Core configuration ***** */

  // Reset REFMSTR to enable control of reference voltages by ADC12
  REFCTL0 &= ~REFMSTR;

  /*
   * Initialize control register ADC12CTL0
   * STH0x   = 9 => 384 clock cycles; MSC = 0 => no multisample mode
   * REF2_5V = 1 => Reference is 2.5V, REFON = 1 => Use internel reference generator
   * ADC12ON = 1 => Turn on ADC12
   */
  ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12REF2_5V | ADC12ON;

  /*
   * Initialize control register ADC12CTL1
   * STARTADDx = 0 => Start conversion at ADC12MEM0
   * SHSx      = 0 => Conversion trigger:  Start when ADC12SC is set to 1
   * SHP       = 1 => SAMPCON sourced from sampling timer (default)
   * ISSH      = 0 => Input signal not inverted
   * SSEL      = 0 => ADC12clock = ADC12OSC (~5 MHz)
   * DIVx      = 0 => Divide ADC12CLK by 1
   * CONSEQx   = 0 => Single channel, single conversion mode
   */
  ADC12CTL1 = ADC12SHP;

  /* ***** Channel configuration ***** */

  // Set conversion memory control register ADC12MCTL0
  // SREF = 001b => Voltage refs:
  // EOS  =   0  => End of sequence not set (not a multi-channel conversion, so ig-
  nore)
  ADC12MCTL0 = ADC12SREF_1 | ADC12INCH_0;

  // Set P6.0 to FUNCTION mode
  // This connects the physical pin P6.0/A0 to the ADC input A0
  P6SEL |= BIT0;

  // Enable the ADC.  This means we are done configuring it,
  // so we can start the conversion.
  ADC12CTL0 |= ADC12ENC;
}
```

```
unsigned int read_adc(void) {
  // Input voltage has range 0-2.5V, which corresponds to 0 to 1A.
  unsigned int in_value;

  ADC12CTL0 &= ~ADC12SC;
  // Enable and start a single conversion
  ADC12CTL0 |= ADC12SC;


  // Wait for the conversion to finish by polling the busy bit
  // The busy bit is automatically set to 0 when the conversion is done
  while(ADC12CTL1 & ADC12BUSY) {
      __no_operation(); // Could also just leave the loop empty
  }

  // Now that the conversion has completed, we can read the result
  // from the memory register
  in_value = ADC12MEM0 & 0x0FFF; // Keep only the low 12 bits
  return in_value;
}
```

Now what do we do with the return value?

Example: The internal temperature sensor

To use any sensor, you need to understand how the sensor output (in this case, voltage) corresponds to the quantity it measures, which is documented by the designers.

Our MSP430 contains a built-in sensor to measure the internal chip temperature. It has a linear mapping from voltage to temperature:

A typical temperature sensor transfer function is shown in Figure 28-11 . The transfer function shown in Figure 28-11 is only an example—the device-specific data sheet contains the actual parameters for a given device. When using the temperature sensor, the sample period must be greater than 30 µs. The temperature sensor offset error can be large and may need to be calibrated for most applications. Temperature calibration values are available for use in the TLV descriptors (see the device-specific data sheet for locations).

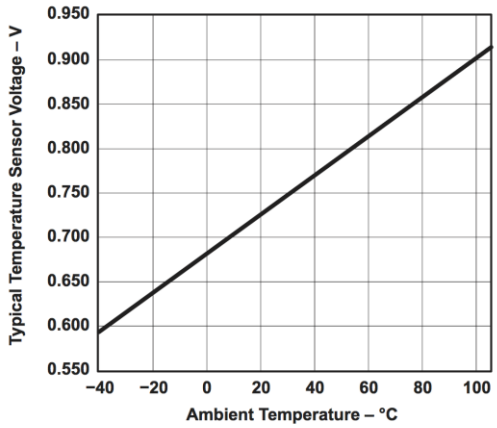


Figure 28-11. Typical Temperature Sensor Transfer Function

To use the sensor, we need to read some *calibration* data from the device, which we use in the computation for the resolution. This information is stored in the *Tag-Length-Value Table (TLV Table),* which is a portion of flash memory that contains some device-specific settings and constants—we need to read from the addresses specified in this table to get the calibration data. For more information on how this works, see p. 102 of the datasheet.

According to the datasheet, the calibration data provided is based on a 1.5V reference.

More ADC features

Multi-channel conversion

What if we wanted to read data from two sensors? Consider the following sensors:

- Our current sensor example from earlier (connected to input channel A0, 2.5V reference)
- A barometric pressure sensor (Input channel A4, 3.3V reference)

These sensors require different settings for reference voltages and inputs.
We *could* reconfigure the ADC every time we wanted to take a measurement, but this would be annoying. Instead, the ADC12 provides several different conversion modes to solve this problem. We will discuss the most straightforward: *multiple channel, single conversion* mode.

To perform readings from two sensors, we will need to use **two** ADC12MEMx registers, one for each channel.

Like the previous examples, we need to:

- Configure the ADC12 core, this time selecting *multiple-channel, single conversion* mode
- Configure one ADC12MEMx register for each reading we want to perform with the appropriate settings for each channel (ie, analog channel and reference voltage)

Configuration example

```
// Multi-sample conversion example:  current sensor + pressure sensor
// Input voltage has range 0-2.5V, which corresponds to 0 to 1A.
// The temperature sensor uses a 3.3V reference, using the following calibration
data


// Resolution for current sensor:  1.0A/4096
// (Here, we add the f suffix so the compiler will know this constant is a float)
#define MA_PER_BIT        (0.0244f)

void config_adc(void)
{
  unsigned int in_current, in_temp;
  /* ***** Core configuration ***** */

  // Reset REFMSTR to enable control of reference voltages
  REFCTL0 &= ~REFMSTR;

  /*
   * Initialize control register ADC12CTL0
   * This is similar to single-channel conversion, except we add the following:
   * MSC = 1 => Burst mode (convert multiple channels) * <- *NEW*
   * REF2_5V = 1 => Reference is 2.5V
   */
  ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12REF2_5V | ADC12ON | ADC12MSC;
  //                                                             ^----NEW!

  /*
  * Initialize control register ADC12CTL1
  * This is similar to the previous example,
  * except we need to consider the following:
  * STARTADDx = 0     => Start conversion at ADC12MEM0
  * CONSEQx   = **1** => ***Sequence of channels, converted once***
  */
  ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;
  //                     ^------NEW!

  /* ***** Channel configuration ***** */
  /* Here, we pick one MCTLx register per reading we need
   * Let's pick MCTL0 for the current sensor, and MCTL1 for the pressure sensor
   */

  // Current sensor:  Input channel A0, SREF_1 => Internal ref => 2.5V
  ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_0;

  // Pressure sensor:  Input channel A4, SREF_0 => 3.3V Reference, ADC12EOS = 1
  ADC12MCTL1 = ADC12SREF_0 + ADC12INCH_4 + ADC12EOS;
  //                                       ^- NEW!
  // For the last channel in our sequence, we set the ADC12EOS bit.  This tells
  // the ADC12 that our sequence is finished!

  ADC12CTL0 |= ADC12ENC; // Done configuring, enable conversion
```

```
  /* ***** I/O configuration ***** */
  // Set P6.0 and P6.4 to FUNCTION mode
  // This connects the physical pins P6.0/A0 P6.4/A4 to the ADC inputs A0 and A4
  P6SEL |= BIT4|BIT0;
}

void read_adc(void)
{
  // Enable the ADC and start the conversion
  ADC12CTL0 |= ADC12ENC;
  ADC12CTL0 &= ~ADC12SC;
  ADC12CTL0 |= ADC12SC;

  // Wait for the conversion to finish by polling the busy bit
  while(ADC12CTL1 & ADC12BUSY) {
      __no_operation();
  }

  // Now that the conversion has completed, we can read the results
  // from the memory registers
  in_current = ADC12MEM0 & 0x0FFF; // Keep only the low 12 bits
  in_pressure    = ADC12MEM1 & 0x0FFF;

  // Finally, we would convert the results to current (amps)
  // and pressure (atm, kPa, mmHg. ...), respectively, and store them somewhere
}
```

ADC12 Interrupts

Our examples so far have used the `ADC12SC` bit to start conversions, and then we poll the `ADC12BUSY` bit to see when a conversion is complete:

```
ADC12CTL0 |= ADC12SC;       // Start the conversion

// Poll busy bit waiting for conversion to complete
while (ADC12CTL1 & ADC12BUSY) {
    __no_operation();       // Could just leave body of loop empty
}

in_value = ADC12MEM0 & 0x0FFF; // Read result
```

This is another form of *busy-waiting*, which is like `swDelay`:

- The CPU isn't doing any useful work–it's just sitting in a loop!
- While ADC conversions happen very quickly (<< 1 ms), the CPU executes faster, so we do need to wait for a result

The main purpose of on-chip peripherals like the Timer and ADC is to remove burdens from or provide services to the CPU.

How useful is a peripheral if the CPU is just waiting for it to finish?

How do we use interrupts on the ADC12?

- Configure control registers same as before (as if you were polling)
- Set the ADC12IE register bit corresponding to the **last** MEMx register used in the conversion. (If you are only doing single-channel conversions, this is ADC12MEM0.)

Here is an example configuration with two channels:

```
ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12ON | ADC12MSC;
ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;

// Here, we are performing conversions for two channels
ADC12MCTL0 = ADC12SREF_0 + ADC12INCH_5;
ADC12MCTL1 = ADC12SREF_1 + ADC12INCH_6 + ADC12EOS;

// Because we are converting for two channels, we want the interrupt
// to occur after BOTH conversions are complete, so we enable the
// interrupt for MEM1.
ADC12IE = BIT1;

__enable_interrupt(); // Globally enable interrupts

ADC12CTL0 |= ADC12SC + ADC12ENC; // Enable ADC and start conversion
```

What should the ISR do? It's triggered when a conversion is finished, so it just needs to read the memory registers!

```
// Global variables for storing data
// (could also store into an array!)
volatile unsigned int in_value1, in_value2;

#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{




}
```

Timer-triggered ADC Measurements

It's also possible to have the ADC perform conversions automatically. An easier option is to trigger ADC conversions from a timer and use ADC interrupts to read the results.

For example, say we wanted to take measurements every 0.25 seconds:

```
void config_timerA2(void)
{




}
```

Inside the timer ISR, start the ADC12 conversion:

```
// NOTE:  this example assumes the ADC has already been configured.

// Global variables for storing data (could also store into an array!)
volatile unsigned int value1, value2;

// Timer A2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2_ISR(void)
{
    timer++; // You probably still want to keep track of time


    ADC12CTL0 |= ADC12SC;
}

// ADC 12 ISR
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{

    // Move the result(s) into global variables
    value1 = ADC12MEM0 & 0x0FFF;
    // . . .
}

void main(void) {
    do_all_setup();
    while(1) {
        display(value1);
    }
}
```

Aside: Data logging

A common practice when working with sensors is to record past data. Since memory is finite resource, it is common to only keep the last N values.
For example: if we are building a sensor that logs temperature and wind speed, we may measure data every second and keep only the results from the last hour.

An efficient way to keep track of the "last N values" in an array is by using *circular indexing*. When used this way, the array is often called a *circular buffer*.

Circular Indexing: How it works

- Use a fixed-size array ("buffer") to store values
- Keep track of the *tail*, which is the "end" index of data in the array (also called the "write pointer")
  Using this definition, the tail always represents the location of the oldest element, which is also the next index to be filled with new data
- When adding a new element, add 1 to the index.
  When the index reaches the end of the array, wrap around to the beginning

How do we implement this?

For more information

- See "Data Logging Example" on the course website
- The Wikipedia entry on Circular Buffer has great animations!

Module 10. Operating Modes and More

Topics

- Revisiting interrupts
- Operating modes

ADC12 Interrupts

Our examples so far have used the `ADC12SC` bit to start conversions, and then we poll the `ADC12BUSY` bit to see when a conversion is complete:

```
ADC12CTL0 |= ADC12SC;


while (ADC12CTL1 & ADC12BUSY) {
    __no_operation();       // Could just leave body of loop empty
}


in_value = ADC12MEM0 & 0x0FFF;
```

This is another form of *busy-waiting*, which is like `swDelay`:

- The CPU isn't doing any useful work–it's just sitting in a loop!
- While ADC conversions happen very quickly (<< 1 ms), the CPU executes faster, so we do need to wait for a result

The main purpose of on-chip peripherals like the Timer and ADC is to remove burdens from or provide services to the CPU.

The solution to this is to use an **Interrupt**, an external signal sent by a peripheral requesting that the CPU do something. Interrupts are hard-wired into certain peripherals. Fortunately for us, the ADC12 can trigger interrupts.

On the ADC12, an interrupt can signal the end of a conversion, **meaning that it is ready for the CPU to read data from its memory registers**.

Here is an example configuration with two channels:

```
ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12ON | ADC12MSC;
ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;

// Here, we are performing conversions for two channels
ADC12MCTL0 = ADC12SREF_0 + ADC12INCH_5;
ADC12MCTL1 = ADC12SREF_1 + ADC12INCH_6 + ADC12EOS;

// Because we are converting for two channels, we want the interrupt
// to occur after BOTH conversions are complete, so we enable the
// interrupt for MEM1.
ADC12IE = BIT1;

__enable_interrupt(); // Globally enable interrupts

ADC12CTL0 |= ADC12SC + ADC12ENC; // Enable ADC and start conversion
```

What should the ISR do? It's triggered when a conversion is finished, so it just needs to read the memory registers!

```
// Global variables for storing data
// (could also store into an array!)
volatile unsigned int in_value1, in_value2;

#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{

    // Move the results for both channels into global variables
    in_value1 = ADC12MEM0 & 0x0FFF;
    in_value2 = ADC12MEM1 & 0x0FFF;

}
```

"Scheduling" ADC measurements

It's also possible to have the ADC perform conversions automatically. An easier option is to trigger ADC conversions from a timer and use ADC interrupts to read the results.

```
// NOTE:  this example assumes the timer and ADC have already been configured.



volatile unsigned int in_value1, in_value2;


// Timer A2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2_ISR(void)
{
    timer++;


    ADC12CTL0 |= ADC12SC;
}

// ADC 12 ISR
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    in_value1 = ADC12MEM0 & 0x0FFF;
    // . . .
}

void main(void)
{
    setup_everything();
    _enable_interrupt();

    while(1){
        do_something_with_adc_values(in_value1, in_value2);
    }
}
```

Here, we now have two ISRs, one for the ADC, and one for the timer. Thus, our main program can simply use `in_value1` and `in_value2` without needing to explicitly start conversions.

Polling versus Interrupts Revisited

So far most of our code has relied heavily on *Polling*. While we have been using the timer to keep track of fixed time intervals, our main functions are still busy-waiting in some form of loop:

```
while (1)
{
   if (global_time_cnt > last_time)
   {
      take_ADC_meas();
      last_time = global_time_count;
   }
   button = checkButtons();
     . . .

   // Other task(s) can occur at different intervals
   if ((global_time_cnt % cnt_per_second) == 0)
   {
       toggleLED(LED2)
        displayHHMMSS(global_time_cnt);
    }
      . . .
}
```

This is not efficient: while it is waiting, the CPU is using precious energy to check if it needs to do something!

As an alternative, we can organize our code to *schedule* tasks to occur at specific "real" times, and even assign priorities to tasks.

--> We do this by implementing the Scheduler INSIDE Timer ISR!

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
      // First priority:  maintain time base
      global_time_cnt++;


      // Some of the app task(s) may execute every time slice
      take_ADC_meas();
      button = checkButtons();

     . . .

     // Other application task(s) at different intervals
      if ((global_time_cnt % cnt_per_second) == 0)
      {
         toggleLED(LED2)
          displayHHMMSS(global_time_cnt);
     }
       . . . .

}
```

Here, main() would consist of initialization followed by an "empty" loop:

```
void main()
{
  WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
  init_sys();                  // Initialize the MSP430
  setupButton();              // configure buttons
  setupADC();                 // configure ADC

    . . .

  _enable_interrupt();       // Global Interrupt enable
  runTimerA2();              // Start scheduler

  // An empty forever loop!
  // All application tasks are scheduled and dispatched
  // from within TimerA2 ISR
  while(1)
  {
     __no_operation();
  }
```

What are we assuming by organizing or scheduling our application's tasks like this?

There some very important assumptions here:

Remember the rule of interrupts!

- We can have multiple interrupt sources for different peripherals—we want to maintain this behavior when scheduling tasks.

- By default, interrupts are disabled inside an ISR. Therefore, one interrupt cannot normally interrupt another one, but we can change this behavior when we need it.

Interrupt Priorities

Interrupt Vector Addresses

The interrupt vectors and the power-up start address are located in the address range 0FFFFh to 0FF80h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

Table 4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV)(1)(2)	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV)(1)	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV)(1)(2)	(Non)maskable	0FFFAh	61
Comp_B	Comparator B interrupt flags (CBIV)(1)(3)	Maskable	0FFF8h	60
TB0	TB0CCR0 CCIFG0(3)	Maskable	0FFF6h	59
TB0	TB0CCR1 CCIFG1 to TB0CCR6 CCIFG6, TB0IFG (TB0IV)(1)(3)	Maskable	0FFF4h	58
Watchdog Timer_A Interval Timer Mode	WDTIFG	Maskable	0FFF2h	57
USCI_A0 Receive or Transmit	UCA0RXIFG, UCA0TXIFG (UCA0IV)(1)(3)	Maskable	0FFF0h	56
USCI_B0 Receive or Transmit	UCB0RXIFG, UCB0TXIFG (UCB0IV)(1)(3)	Maskable	0FFEEh	55
ADC12_A	ADC12IFG0 to ADC12IFG15 (ADC12IV)(1)(3)(4)	Maskable	0FFECh	54
TA0	TA0CCR0 CCIFG0(3)	Maskable	0FFEAh	53
TA0	TA0CCR1 CCIFG1 to TA0CCR4 CCIFG4, TA0IFG (TA0IV)(1)(3)	Maskable	0FFE8h	52
USB_UBM	USB interrupts (USBIV)(1)(3)	Maskable	0FFE6h	51
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)(1)(3)	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0(3)	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV)(1)(3)	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)(3)	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV)(1)(3)	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV)(1)(3)	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0(3)	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV)(1)(3)	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)(3)	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV)(1)(3)	Maskable	0FFD2h	41
Reserved	Reserved(5)		0FFD0h	40
			⋮	⋮
			0FF80h	0, lowest

(1) Multiple source flags
(2) A reset is generated if the CPU tries to fetch instructions from within peripheral space or vacant memory space. (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
(3) Interrupt flags are located in the module.
(4) Only on devices with ADC, otherwise reserved.
(5) Reserved interrupt vectors at addresses are not used in this device and can be used for regular program code if necessary. To maintain compatibility with other devices, it is recommended to reserve these locations.

Operating Modes

What's the advantage of scheduling tasks?

>>Our CPU doesn't need to be running all the time!!

This means we can use the MSP430's <u>operating modes</u> to save energy when not performing tasks.

The MSP430F5529 has 6 Operating Modes:

Active Mode => "Normally Active" = CPU is active, all enabled clocks are active

Low Power Mode 0 => CPU, MCLK are disabled , SMCLK , ACLK are active

Low Power Mode 1 => CPU, MCLK, DCO osc. are disabled , DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode, SMCLK, ACLK are active

Low Power Mode 2 => CPU, MCLK, SMCLK, DCO osc. are disabled , DC generator remains enabled, ACLK is active

Low Power Mode 3 => CPU, MCLK, SMCLK, DCO osc. are disabled , DC generator disabled, ACLK is active

Low Power Mode 4 => CPU and all clocks disabled (RAM retention mode)

Low Power Mode 4.5 => CPU and all clocks disabled (no RAM retention), PWR management off, Digital IO pin configuration retained

Operating Mode Basics

>> Most embedded systems run on batteries

-- Enter Low Power Mode to conserve batteries

On MSP430F5529 Low Power Modes are indicated/selected thru the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the *Status Register (SR)*

Mode	***SCG0***	***SCG0***	***OCSOFF***	***CPUOFF***
Active	0	0	0	0
LPM0	0	0	0	1
LPM1	0	1	0	1
LPM2	1	0	0	1
LPM3	1	1	0	1
LPM4	1	1	1	1
LPM4.5 *	1	1	1	1

*PMMREGOFF bit = 1, as well

Status Register Layout:

Reserved	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C

How do you enter a Low Power Mode?
Can set with the `_BIS_SR()` function:

Ex:
```
_BIS_SR(LPM3_bits) ;       // set bits to enter LMP3

_BIS_SR(LPM3_bits|GIE) ;  // set bits to enter LMP3
                          // and enable interrupts
```

When to enter LPM and how do you exit?

--> Ideally want to enter LPM whenever not executing tasks

--> This is made simple if program is organized as a "Scheduler"

--> Enter LPM after starting timer in `main()`

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
      // First priority = maintain the time base
      global_time_cnt++;

      // Some app task(s) may execute every time slice
      take_ADC_meas();
      button  =checkButtons();
     . . .

     // Other application task(s) execute less frequently
      if ((global_time_cnt % cnt_per_second) == 0)
          displayHHMMSS();
      . . . .

}
```

How do you exit LPM if CPU is OFF?

>> **INTERRUPTS!**

When an interrupt is received from a certain source, the CPU automatically does the following:

1. Finishes its current (assembly) instruction
2. Saves Status Register (SR) and Program Counter (PC) to stack
3. Clears Status Register (set to 0)
4. Loads address of ISR that was triggered from Interrupt Vector Table (IVT), loads it into PC
5. Execution continues in ISR
6.

Returning from an ISR restores the Status Register to its previous values meaning that your program will <u>automatically return to Low Power Mode</u> when exiting the ISR!

```
void main(void)
{
   WDTCTL = WDTPW | WDTHOLD;

   init_sys();     // initialize system
   _BIS_SR(GIE);

      . . .
   runtimerA2();   // task scheduler runs in Timer A2 ISR
      . . .
   _BIS_SR(LPM0_bits|GIE);  // Enter low power mode
    // Arrival of timer interrupt will cause MSP430 to exit low
    // power mode and enable it to execute all tasks within the
    // Timer A2 ISR

    while(1) {
      // Main program loop does nothing!
    }

}
```

Which LPM you enter depends on which clocks you are using for your scheduling Timer.

-- Also, on how quickly you need other clocks

>> Low Power Modes are great in practice, but can make debugging painful!

-- It's simple so add it last when debugging is complete

-- Then test to see if '430 is waking and executing tasks properly

Computing Power Usage: Example

An MSP430F5529 is being powered by a Duracell 2/3A $LiMnO_2$ battery, which has a nominal voltage of 3V and a listed capacity of 1550 mAh.

The MSP430F5529 is running an application that is in Active Mode for 5.6 % of the time, in LPM0 for 16.7% of the time, and in LPM4 for the rest of the time.
How long can the application run using this battery?

Module 11. Intro to Digital Interfaces

Digital Interface: Connection between two or more digital devices.

Types of Digital Interfaces

There are two classifications for how data is transmitted in a digital interface: serial and parallel

Parallel Interfaces = Each bit has its own electrical connection (interconnect, trace, wire)

>> Advantages = Fast, easier to synchronize (1 clock edge transfers all bits together)

>> Used almost exclusively inside a CPU (or other) chip

>> Disadvantage = Each bit must have its own electrical connection

Serial Interfaces – Bits sent one after another along a single connection

>> Used almost exclusively to make connections off-chip (and off-computer, through the Internet, out to the Mars Rover, etc.)

>> Advantages = Simpler/fewer connections between CPU and peripheral (2-4 lines)

Examples include:

Device Select/Enable (CS)

Synchronizing CLK (SCLK)

Data Line(s) (SDI and SDO)

Common ground (GND): Often already established if devices are on the same board or IC

>> "Connection" = PCB trace, wire, RF, acoustic, optical, etc.
>> Disadvantages = "Slower", more complicated synchronization, potential timing issues

How are digital interfaces implemented?

Most microcontrollers contain hardware peripherals that implement the interface in hardware. On the MSP430, we have hardware peripherals to implement a few types of serial interfaces.

On the MSP430: Universal Serial Communications Interface (UCSI)

>> Basically acts as a parallel-to-serial and serial-to-parallel converter

>> Most modern microprocessors/microcontrollers will have built-in Serial interface
-- MSP430F5229 has a total of four, in two types

>> Role of Serial interfaces has grown with growing sophistication and speed of serial links (SPI and I^2C to USB and others)

We will discuss three types of interfaces: UART, SPI, and I2C. There are many more types of digital interfaces!

What kind of information is exchanged?

Just like in a CPU, information in a digital interface is represented in bits. The interface defines the manner in which bits are transmitted:

Types of Interfaces

UART

>> UART: Universal Asynchronous Receiver/Transmitter
>> UART mode configures basic 2-wire *asynchronous* serial communications
>> Connect to UCSI with 2 external pins (MSP430: `UCAxRXD` and `UCAxTXD`)

>> Not synchronous (no shared clock) = Asynchronous

>> To use serial communications both devices must know data format and baud rate

- These are set using UARTs control registers
- Implies make data format & baud rate decisions at design time

UART: Fundamental Parameters

In order to use UART for an application, you need to determine the following parameters:

- Start Bit = 1 bit (“low”)
- Data Bits = 7 or 8 bits
- Parity = Even, Odd, or None
  >> Even Parity = 1 when number of 1's including parity is even
  >> Odd Parity = 1 when number of 1's including parity is odd
- Stop bit(s) = 1 or 2 bits (“high”)
- Baud rate (bits/sec): Common rates are 200, 2400, 9600, 19200, 115200

RS-232 – “Old standby” for serial format →Actually is a specific standard with associated voltage ranges and baud rated now often misused to mean any asynchronous serial communication

-->Data sent Least Significant Bit (LSB) first**

** Most UARTs send asynchronous serial data LSB first (because RS-232 is LSB first) but MSP430F5229 UCSI_A is configurable and can be set to send MSB first, so that it can be compatible with various types of serial interfaces.

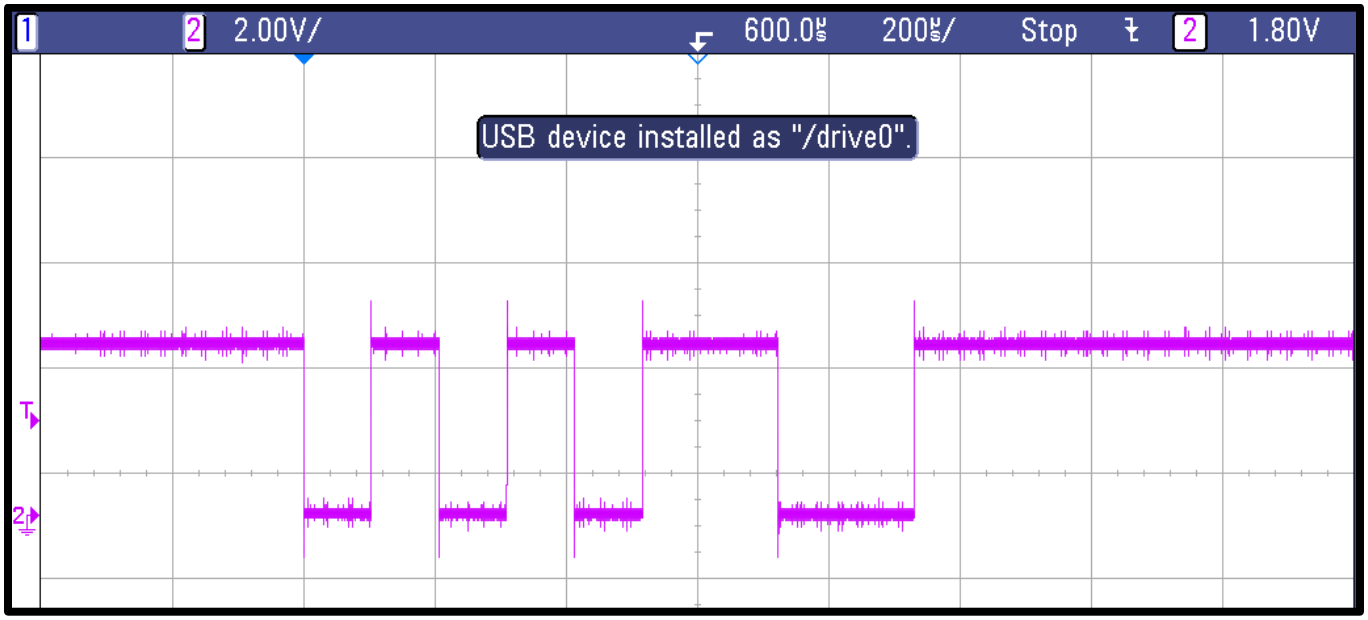
Common Baud Rates

Table 36-4. Commonly Used Baud Rates, Settings, and Errors, UCOS16 = 0

BRCLK Frequency (Hz)	Baud Rate (baud)	UCBRx	UCBRSx	UCBRFx	Maximum TX Error (%)		Maximum RX Error (%)	
32 768	1200	27	2	0	-2.8	1.4	-5.9	2.0
32 768	2400	13	6	0	-4.8	6.0	-9.7	8.3
32 768	4800	6	7	0	-12.1	5.7	-13.4	19.0
32 768	9600	3	3	0	-21.1	15.2	-44.3	21.3
1 000 000	9600	104	1	0	-0.5	0.6	-0.9	1.2
1 000 000	19200	52	0	0	-1.8	0	-2.6	0.9
1 000 000	38400	26	0	0	-1.8	0	-3.6	1.8
1 000 000	57600	17	3	0	-2.1	4.8	-6.8	5.8
1 000 000	115200	8	6	0	-7.8	6.4	-9.7	16.1
1 048 576	9600	109	2	0	-0.2	0.7	-1.0	0.8
1 048 576	19200	54	5	0	-1.1	1.0	-1.5	2.5
1 048 576	38400	27	2	0	-2.8	1.4	-5.9	2.0
1 048 576	57600	18	1	0	-4.6	3.3	-6.8	6.6
1 048 576	115200	9	1	0	-1.1	10.7	-11.5	11.3
4 000 000	9600	416	6	0	-0.2	0.2	-0.2	0.4
4 000 000	19200	208	3	0	-0.2	0.5	-0.3	0.8
4 000 000	38400	104	1	0	-0.5	0.6	-0.9	1.2
4 000 000	57600	69	4	0	-0.6	0.8	-1.8	1.1
4 000 000	115200	34	6	0	-2.1	0.6	-2.5	3.1
4 000 000	230400	17	3	0	-2.1	4.8	-6.8	5.8
4 194 304	9600	436	7	0	-0.3	0	-0.3	0.2

Example: How long would it take to transmit “Term is over!” at 9600 baud with 1 start bit 1 stop bit and even parity assume 8-bit ASCII encoding?

An Example UART Transmission



What does this mean?

We have no idea unless we know the parameters used for the transmission. In this case, the parameters are: 9600 baud, 1 start bit, 1 stop bit, and no parity bits.
What data is being transmitted?

Serial Peripheral Interface Bus (SPI)

>> Used primarily for synchronous serial communications between a CPU and peripherals "within the box"

-- Synchronous = shared clock (supplied by master device)

>> Usually a 4 wire connection (sometimes 3-wire)

SIMO = Slave In/Master Out data line

SOMI = Slave Out/ Master In data

SCLK = Serial Clock (Called `UCAxCLK` in MSP430 Documentation)

CS = Chip Select

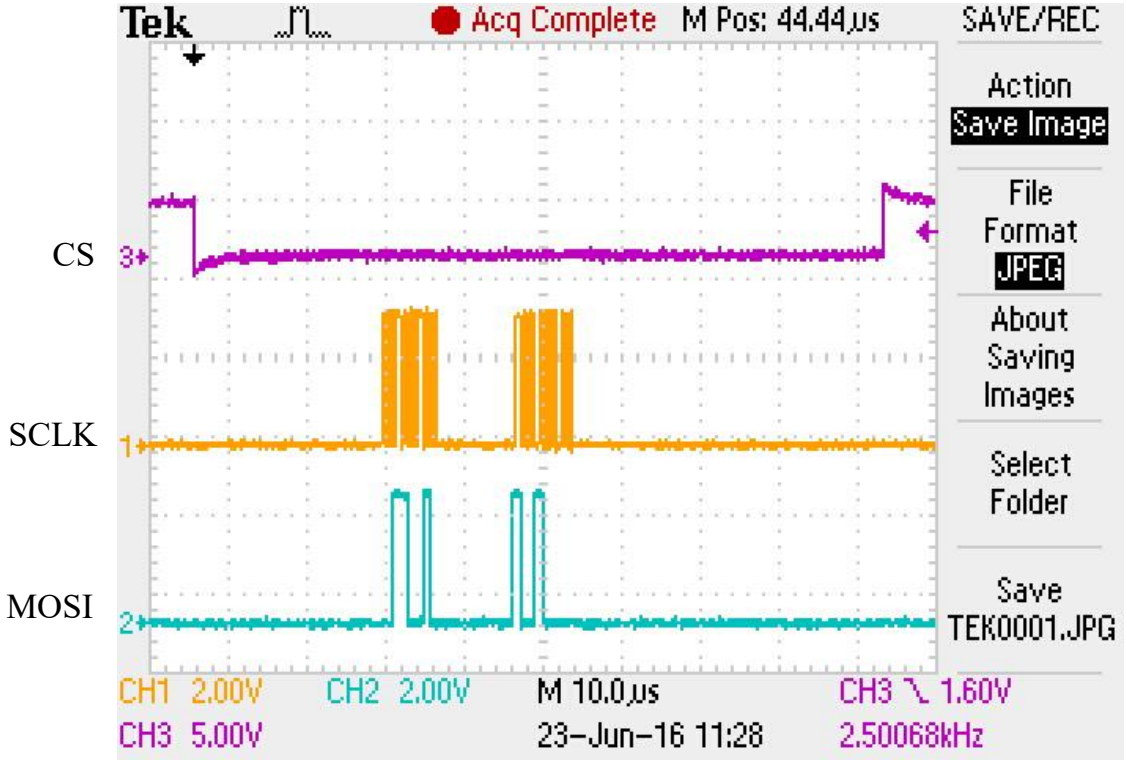
>> SPI is somewhat loose standard --> *Different* from I^2C

How will you know what to use?

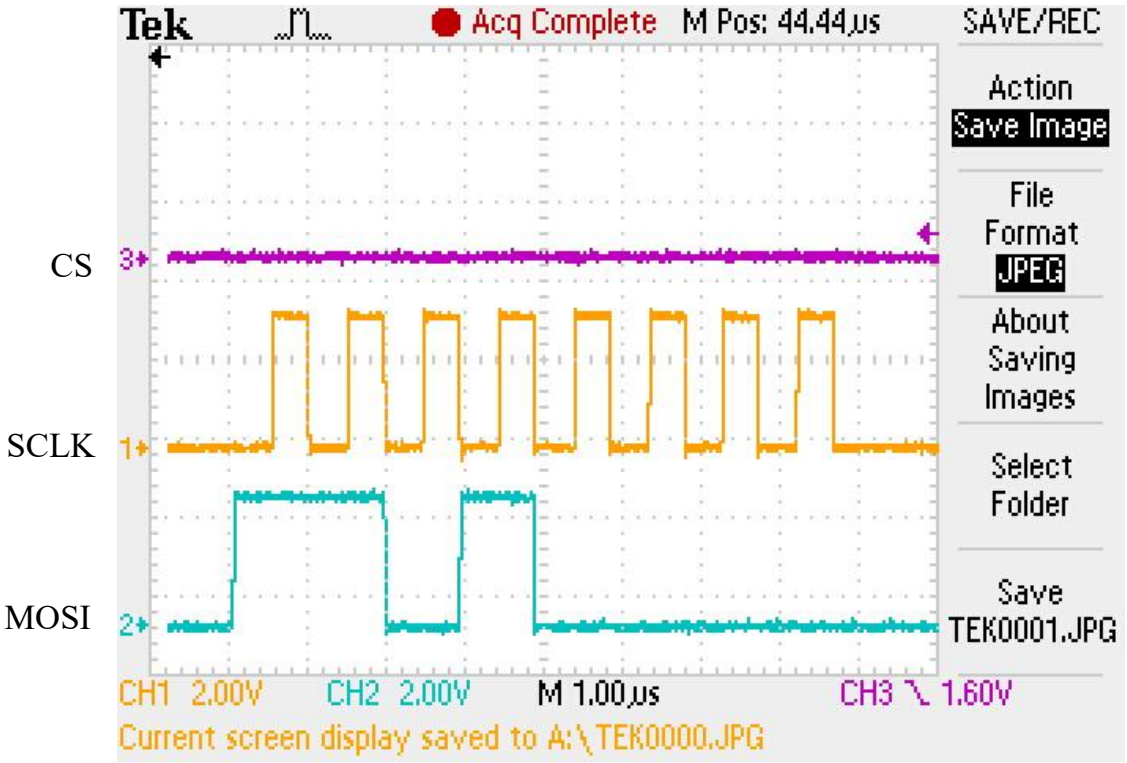
--> SPI, I^2C, asynchronous serial (RS-232), other?
Sensors or other peripheral devices will specify the interfaces with which they are compatible

An Example SPI Transmission

From a high level:



If we zoom in on just 8 bits…



What do we need to know to decode this?

>> ***To use SPI the programmer must...***

1) Enable USCI_A or USCI_B for SPI mode (and set pins for function mode)

2) Select data format (data size, clock edge)

3) Configure a clock frequency

When communicating with a device, the programmer must also ...

4) Select desired SPI peripheral using its chip select (CS)

First, what is the function of a CS?

>> Likely to be multiple peripherals using SPI bus

>> Only 1 Slave device and Master (MSP430) can use SPI bus at a time

>> CS is typically an ACTIVE LOW signal implemented using a Digital IO pins
CS = 0 = Device is Enabled (will read and write to SPI data lines)
CS = 1= Device is Disabled (outputs are high impedance)

Example SPI Peripherals

Our lab boards have 2 peripherals that are SPI devices, the LCD screen and the digital-to-analog converter (DAC). However, we could readily connect others as the USCI pins and some digital IO pins are available through the headers

Sharp 96x96 LCD Display

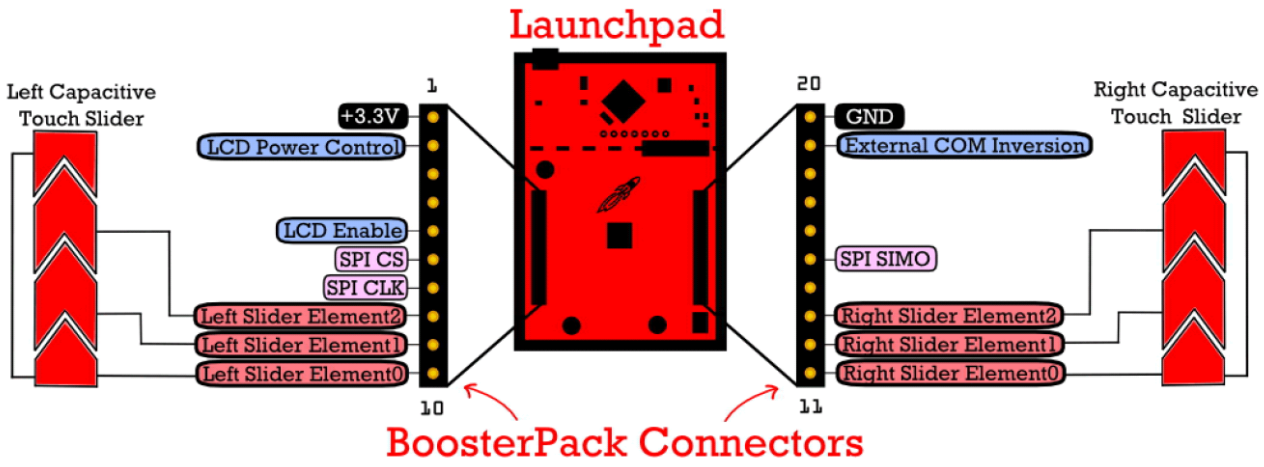


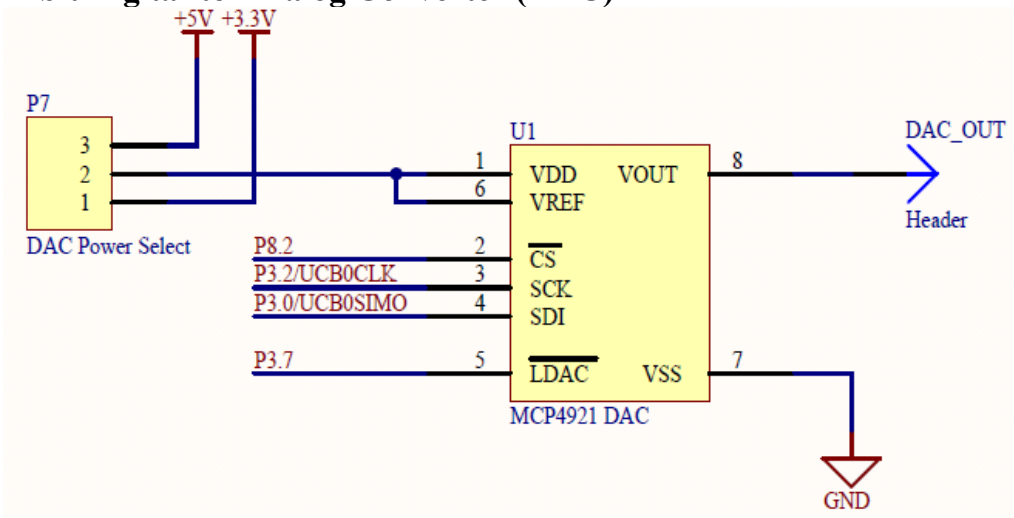
Figure 3. BoosterPack Default Pinout

Interface Pins

- **SPI CS: P6.6 (Digital I/O)**
- **SPI CLK: P3.2 (Function mode, UCB0CLK)**
- **SPI SIMO: P3.0 (Function mode, UCB0SIMO)**
- Two additional digital I/O pins to provide power and enable LCD (P6.5 and P1.6)

Note: LCD does not send data to the MSP430, so the SOMI line is not used!

MCP4921 12-bit Digital-to-Analog Converter (DAC)



Digital to Analog Converter: Converts 12-bit digital code into an analog voltage (in some range V_{Ref+} to V_{Ref-}). Sound familiar?
Can use this to generate analog signals!

Example Data format: MCP4921 Digital to Analog Converter (DAC)

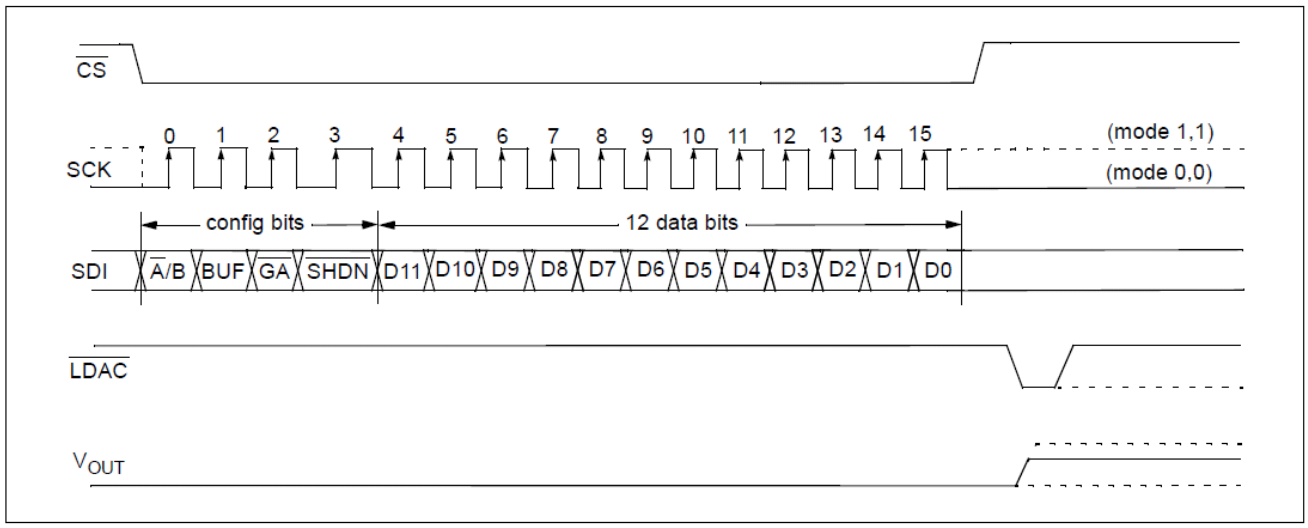


FIGURE 5-1: *Write Command.*

How do we connect multiple devices to the SPI bus?

- The "bus" lines (SCLK, MOSI, MISO) can be shared across all connected devices
- Each device needs its own chip select (CS) line, which tells which device when to wake up and watch the bus for input (or to write some output)

Inter-Integrated Circuit (I^2C)

I^2C (also, I2C) is another increasingly common serial interface. I2C provides a synchronous interface using only two wires!

An I2C interface is comprised of two wires, called the "I2C bus":

- SCL: Serial clock
- SDA: Serial data

I2C bus fundamentals:

- All peripherals are connected to the same two wires
- Both SCL and SDA require *pull-up* resistors such that both lines default to logic high.
- Both lines are *bidirectional*
- Speeds are standardized: typically 100kbps (standard mode) or 400kbps (fast mode)

Unlike SPI, I2C is more rigidly standardized and has strict timing requirements on how the two lines can be used. There are a few standard operations:

- START
- STOP
- ACK: Acknowledge transmission
- NACK: Negative acknowledgement

Every device datasheet will tell you how and when it expects to receive these commands.

How are devices selected?
With only two wires, we have nothing like a chip select (CS) to wake up individual devices! In I2C, every device has an *address*, which is often programmed at the factory.

Before any transmission, the master sends the address of the device it wants to use—the device should only respond to the request if it has a matching address.