ECE2049 Lecture 2

- Overview of C Programming Concepts
- Intro to Lab 0

Administriva

- Lab today 2-4PM, AK113
  - Tutorial, warmup w/ C Programming.
  - Due next Thurs.
- HW1 Due next Tues, By 2PM.

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?

↳ 1 BYTE = 8 BITS.

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 Interchange), 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;
}
```

MAYBE ADD... DISPLAY(DEG K)

NOTHING HAPPENS W/ RESULT!

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

✓ ✗

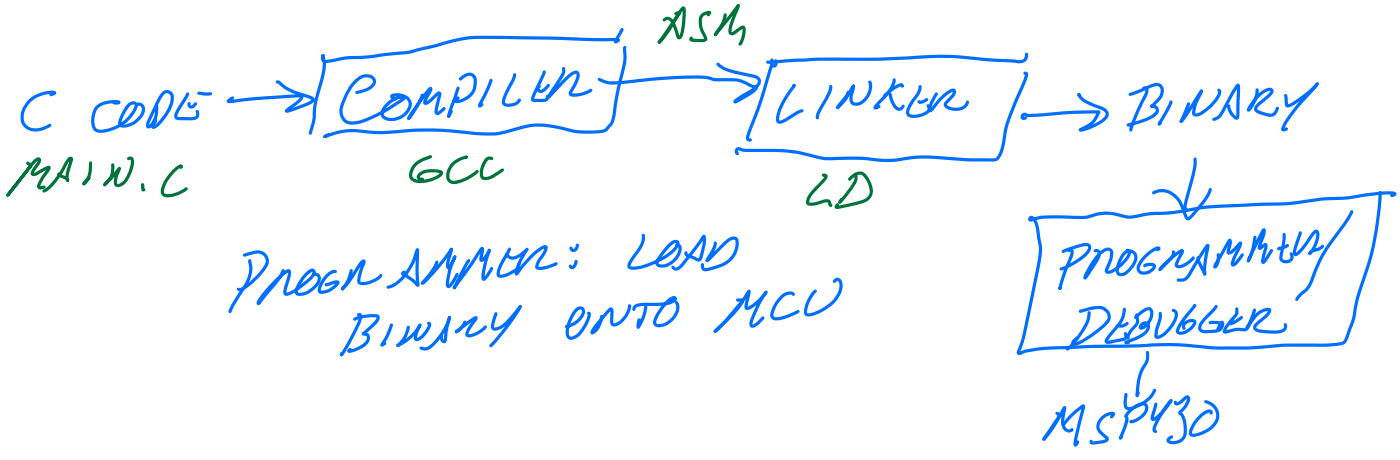
Terminology: Compilation Process

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: TRANSLATES C CODE INTO MACHINE-READABLE FORM (MACHINE CODE/ASSEMBLY) (WHAT CPU UNDERSTANDS)

Linker: JOINS ALL COMPILED FILES INTO ONE EXECUTABLE → ("IMAGE" OR "BINARY")

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!

DECLARATION

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
```

W/O INITIALIZATION (SETTING A VALUE), X IS UNDEFINED! (COULD HAVE ANY VALUE)

DECLARATION + INITIALIZATION

Z = 5 "STORE 5 INTO Z" ⇐ ASSIGNMENT

Y = X "STORE X INTO Y"

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;
```

47/10 = 4 ⇐ INTEGER

47%10 = 7 ⇐ REMAINDER

OPERATIONS USE DATATYPE OF THE INPUTS

Y, Z ARE INTS, SO RESULT GETS "TRUNCATED"

WHAT IF THIS ISN'T WHAT YOU WANT?

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
```

FORCES COMPILER TO CONVERT INT TO FLOAT (WHICH CAN HOLD NUMBERS LIKE 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:

"SIDE EFFECTS"

```
i = 5;
arr[i++] = 4;
// i = 6
```

ARR[5] = 4, i++ ← TWO THINGS HAPPEN ON THIS LINE

Be careful with these!

++i i++

Logical and Relational Operators

Logical operators, such as produce a Boolean result (true or false).

But how are Booleans represented in C?

IF X IS ZERO => FALSE

IF X IS NOT ZERO => TRUE

IF (X) {
// ...

- C DOES NOT HAVE A BOOLEAN TYPE

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

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

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

TRUE OR FALSE

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

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

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

LOOP WHILE COUNT > 0

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

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

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

IF (x == y) {

TRUE IF x AND y HAVE SAME VALUE.

IF (x = y) {

DOESN'T DO WHAT YOU WANT

}

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;
}
```

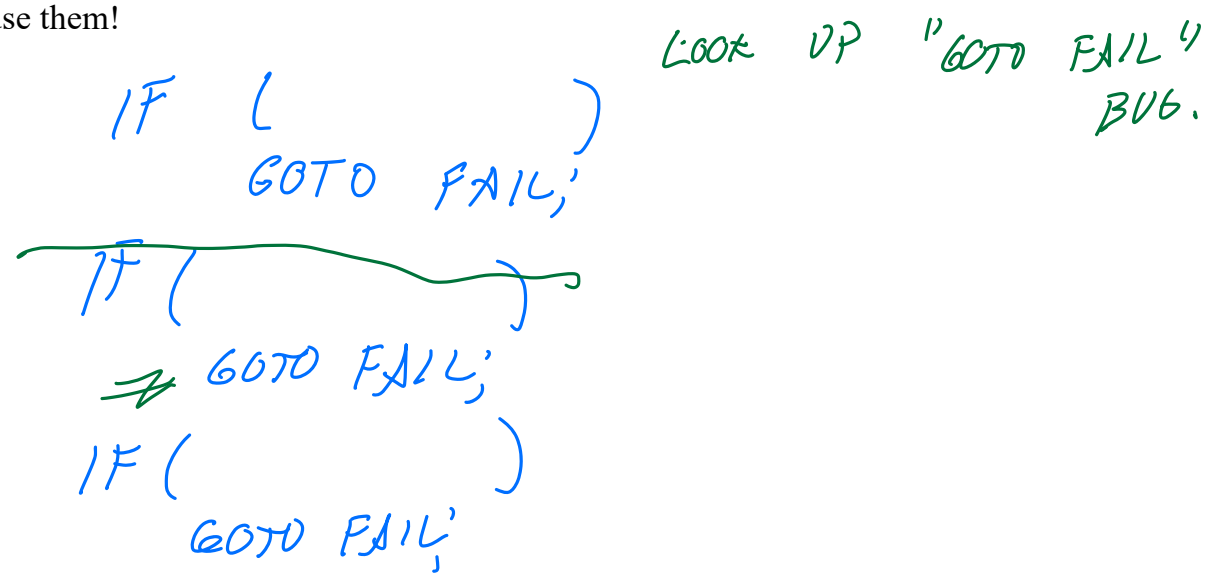
Alternate form for small statements, the conditional operator (also known as the *ternary* operator):

```
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.

IF YOU USE ARDUINO:

```
MAIN(){
    SETUP();
    WHILE(1){
        LOOP();
    }
}
```

A 1000 0101

B 1111 0000

&

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?
   }
}
```