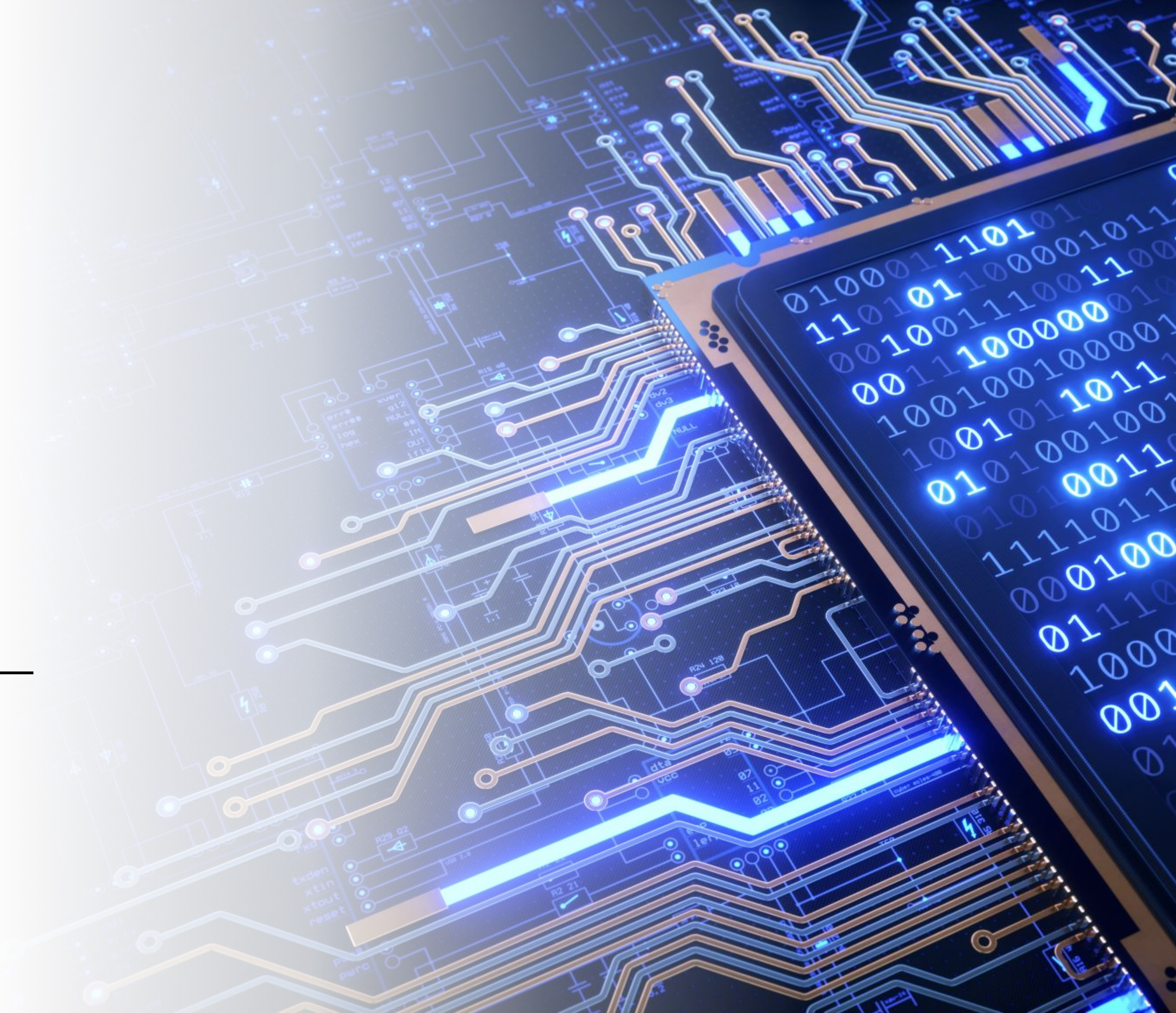
ANSI C

Review/Intro to Basic ANSI C Principles and Terminology



Motivation for this Review

- We will be using ANSI C to program the Arduino
- You should be familiar with the basics of C
- Programming microcontrollers requires using parts of C that are not typically taught in introductory courses
- We will focus on the parts of C that we will be using in this class
- We will revisit as needed – this is just a quick intro to get us started

Preview of Coding the Arduino

- Assembly Code

```
rjmp START                ; the reset vector: jump to "main"
START:
ldi r16, low(RAMEND)      ; set up the stack
out SPL, r16
ldi r16, high(RAMEND)
out SPH, r16
ldi r16, 0xFF             ; load register 16 with 0xFF (all bits 1)
out DDRB, r16             ; write the value in r16 (0xFF) to Data
                          ; Direction Register B
LOOP:
  sbi PortB, 5            ; switch off the LED
  rcall delay_05          ; wait for half a second
  cbi PortB, 5            ; switch it on
  rcall delay_05          ; wait for half a secon
  rjmp LOOP               ; jump to loop

DELAY_05:                 ; the subroutine:
  ldi r16, 31             ; load r16 with 31
OUTER_LOOP:               ; outer loop label
  ldi r24, low(1021)      ; load registers r24:r25 with 1021, our new
                          ; init value
  ldi r25, high(1021)     ; the loop label
DELAY_LOOP:               ; "add immediate to word": r24:r25 are
                          ; incremented
  adiw r24, 1             ; if no overflow ("branch if not equal"), go
                          ; back to "delay_loop"
  brne DELAY_LOOP
  dec r16                 ; decrement r16
  brne OUTER_LOOP         ; and loop if outer loop not finished
  ret                     ; return from subroutine
```

[Sample code from https://medium.com/jungletronics/meeting-assembly-hello-world-arduino-blinking-code-330386652309](https://medium.com/jungletronics/meeting-assembly-hello-world-arduino-blinking-code-330386652309)

Preview of Coding the Arduino (2)

- ANSI C (no libraries)

```
// CPE 301 - REGISTER-LEVEL Blink Example
// Written By Frank Mascarich, Spring 2018

// Define Port B Register Pointers
volatile unsigned char* port_b = (unsigned char*) 0x25;
volatile unsigned char* ddr_b  = (unsigned char*) 0x24;
volatile unsigned char* pin_b  = (unsigned char*) 0x23;

void setup()
{
  //set PB7 to OUTPUT
  *ddr_b |= 0x80;
}

void loop()
{
  // drive PB7 HIGH
  *port_b |= 0x80;
  // wait 500ms
  delay(500);
  // drive PB7 LOW
  *port_b &= 0x7F;
  // wait 500ms
  delay(500);
}
```

Preview of Coding the Arduino (3)

- C/C++ with the Wired Library

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Why ANSI C?

- ANSI C and C++ are used in Arduino programming
- C is a *low-level* language
  - Closer to the hardware than Java / Python / Ruby
- Compiles down to binary instructions for the computer
  - No virtual machine required
- Easier to use than assembly language
- Very fast execution compared to most higher level languages

Why ANSI C? (2)

- The Arduino *Wired* library is written in C++
- Why are we only (well, mostly) using C and not C++?
  - C++ and the Wired library make programming easier, but is not compatible with many other embedded systems
  - Using C without the libraries develops skills that will translate to other systems
- One issue: most code you find on the web uses the Wired library, so you have to translate it to remove the library calls

C++ to C

- C++ is a superset of C, so
  - If you know C++, you know C
- Some differences between C and C++
  - Not object-oriented
    - No convenience classes such as String
  - No exception handling
  - I/O is different
    - C uses scanf/printf where C++ uses cin/cout

Data Types in C

- In programming typical applications, you don't need to worry *too* much about data type sizes
- In embedded and low-level programming, you must keep track of the size of data types used
- NOTE: `int` is machine dependent
    - On a Mac, for instance, `int` is 4 bytes (32 bits)

Type	Width
Char (signed/unsigned)	8 bits
Int (signed/unsigned)	16 bits*
Float	32 bits
Double	64 bits

Assignment Operators

- Shortcut assignment operators are the same as those in C++
- Shortcuts exist for both arithmetic and bit level operators
  - More on this in a minute

`+=`	`i += j;`	`i = (i + j);`
`-=`	`i -= j;`	`i = (i - j);`
`*=`	`i *= j;`	`i = (i * j);`
`/=`	`i /= j;`	`i = (i / j);`
`%=`	`i %= j;`	`i = (i % j);`
`&=`	`i &= j;`	`i = (i & j);`
`|=`	`i |= j;`	`i = (i | j);`
`^=`	`i ^= j;`	`i = (i ^ j);`
`<<=`	`i <<= j;`	`i = (i << j);`
`>>=`	`i >>= j;`	`i = (i >> j);`

Control Structures

- Control structures are the same as those in C++
  - Conditionals
    - `if-else and else-if`
    - `switch-case`
  - Loops
    - `for`
    - `while`
    - `do while`

Functions

- C does not contain classes, so there is no such thing as class methods
- Unlike in C++, functions cannot be overloaded
- As in C++, functions must have a return type
  - The *void* type is used for functions that do not need to return a value
- Values to functions can be passed either by value or by reference

Example Function

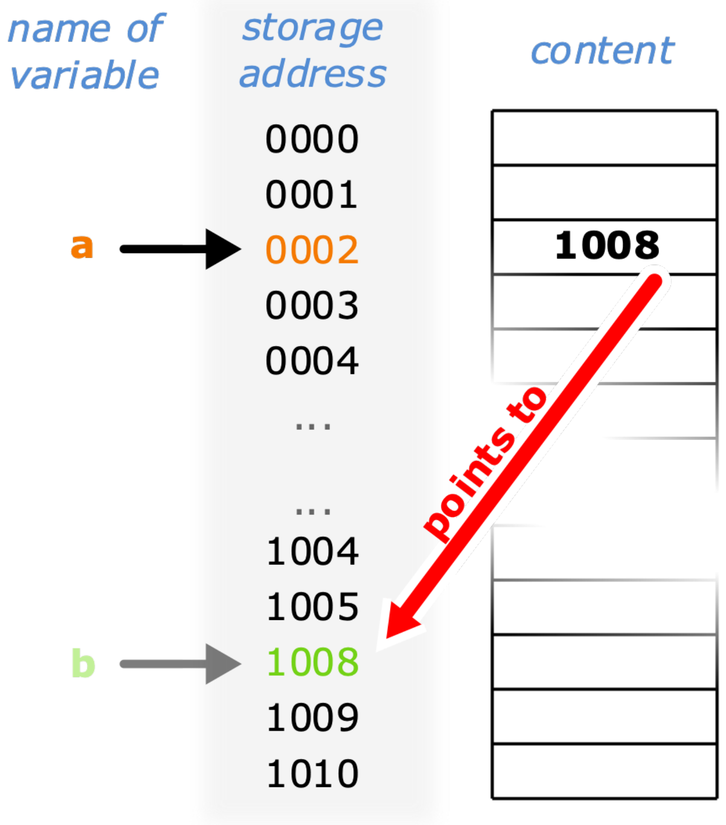
```
void what_does_this_do(char c){
    for(int i = sizeof(c)*8 - 1; i >= 0; i--){
        printf("%c", c & (1 << i) ? '1':'0');
    }
    printf("\n");
}
```

```
char x = 'a';
what_does_this_do(x);
```

We will come back to this after we discuss bitwise operators

Pointers

- A pointer is a variable that holds the *address* in memory
- All languages use pointers, but only low level (C, C++, etc.) expose them to the developer
- To access the contents of the address pointed to by the pointer, the pointer is *de-referenced*
- Pointers can point to many different data types (char, int, etc.) but pointers themselves *are always the same size*



https://en.wikipedia.org/wiki/Pointer_(computer_programming)

Pointer Examples

- What is the output from the print statement?

```c
    int* py;
    int y = 10;
    py = &y;
    *py = 100;

printf("%d",y);
```

```c
int array[5] = {1,2,3,4,5};
*(array + 3) = 10;
for(int i = 0; i < 5; i++){
    printf("%d\n",array[i]);
}
```

Passing by Value vs Passing by Reference

- Passing by value means the actual value of the argument is passed
  - The value of the argument cannot be changed inside the function
- Passing by reference means that the address of the argument is passed to the function
  - The value of the argument can then be changed
  - What is actually passed is a pointer. If you use *, it can be altered to point to a different memory location. If you use &, the value can be changed, but not the memory address it points to

Bitwise Operators

- Many functions used in embedded systems require manipulating individual bits
- Must be able to read, set, or clear individual bits to manipulate registers
- *Be careful that you use the correct operators*
- *The compiler will not give you a warning!*

Table 2.9: Bitwise Operators

Operator	Operation
&	AND (boolean intersection)
|	OR (boolean union)
^	XOR (boolean exclusive-or)
<<	left shift
>>	right shift
~	NOT (boolean negation, i.e., ones' complement)

Statement	x	y	z After	Operation
`z = (x & y);`	1	2	0	Bitwise AND
`z = (x && y);`	1	2	1	Logical AND
`z = (x | y);`	1	2	3	Bitwise OR
`z = (x || y);`	1	2	1	Logical OR

Bitwise Operation Examples

- A *mask* (sometimes called a *bit mask)* is a number that is used to target one or more bits in a bitwise operation

Statement	c	mask	d	Embedded usefulness
`d = (c & mask);`	0x55	0x0F	0x05	Clear bits that are 0 in the mask
`d = (c | mask);`	0x55	0x0F	0x5F	Set bits that are 1 in the mask
`d = (c ^ mask);`	0x55	0x0F	0x5A	Invert bits that are 1 in the mask
`d = (c << 3);`	0x55		0xA8	Multiply by a power of 2
`d = (c >> 2);`	0x55		0x15	Divide by a power of 2
`d = ~c;`	0x55		0xAA	Invert all bits

```
c    = 0x99 = b10011001
mask = 0x0F = b00001111
```

```
   10011001
&  00001111
-----------
   00001001
```

```
   10011001
^  00001111
-----------
   10010110
```

XOR

A Bit* More About << and >> Operators

- The bit shift operators can be used for more than just multiplying
- In our case, we will use frequently use them create masks

```
char mask;
mask = 1 << 4;
printf("Mask in Decimal: %d\n", mask);
printf("Mask in Binary: ");
print_as_binary(mask);
```

Output

```
Mask in Decimal: 16
Mask in Binary: 00010000
```

The `volatile` Keyword

- Compilers assume that *only* the CPU can modify a value in a variable
- In embedded systems, this may not be the case
  - Memory locations may be mapped to ports or devices that may alter their contents without CPU intervention
- The compiler optimization algorithm may remove calls accessing the memory location and instead use cached values
- The solution: declare variables as *volatile*
  - This prevents the compiler from “optimizing out” calls to the memory location

Revisiting the Function Example

Now that we know bitwise operators, what does this function do?

```
void what_does_this_do(char c){
    for(int i = sizeof(c)*8 - 1; i >= 0; i--){
        printf("%c", c & (1 << i) ? '1':'0');
    }
    printf("\n");
}
```

```
char x = 'a';
what_does_this_do(x);
```

Reading

- Jimenez: 5.1-5.3
- Mazidi: 7.1