C Constants, Defines and Integers

1 Review of C Constants

We will often need to specify the content of memory very precisely.

1.1 HEX and OCTAL review

Entering binary numbers can wear out the 1 and 0 keys on your computer! HEX is a good shorthand for binary since each HEX character maps to a specific bit pattern.

<table>
<thead>
<tr>
<th>bits</th>
<th>HEX</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>12</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>17</td>
</tr>
</tbody>
</table>

Hex Place value:
To convert 0xD2FB to decimal:

<table>
<thead>
<tr>
<th>D</th>
<th>2</th>
<th>F</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^12</td>
<td>2^8</td>
<td>2^4</td>
<td>2^0</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Octal can also be used and is easier to memorize because there are only 8 symbols (0-7). However, to fill a byte of memory requires 2.5 octal characters since each character only encodes 3 bits. For example,

HEX 9A = Binary 1011010 = Octal 132

- Ordinary numbers are interpreted by the C compiler as decimal values. Example: 25 = 1101
- Hex values are indicated to the C compiler by the prefix 0x for example: 0x2F
- Octal values are indicated by a leading zero: 028 is a syntax error (why?).

---

1.2 Constant Syntax in C

It is very important to use symbolic constants (#define’s) for all your numeric constants. Here are the main reasons:

- Code is more readable by humans.
- Code is much easier to change.

1.2.1 Example and Exercise

```
// Constants
#define EXAMPLE_CONST_D 1234 ; /* DECIMAL value */
#define EXAMPLE_CONST_H 0x4D2 ; /* HEX value */
#define EXAMPLE_CONST_O 02322 ; /* OCTAL value */

int x = EXAMPLE_CONST_D;
int y = EXAMPLE_CONST_H;
int z = EXAMPLE_CONST_O;

// test yourself here: (which of these print?)
if (x == y) printf("Hello there ...
");
else {
}

if (y == 1234) printf("What’s up doc?? 
");
else {
}

if (z == 2322) printf("The Rain in Spain ...
");
else {

}

if (z = 2322) printf("Falls mainly on the plains.
");
else {

}
```

1.2.2 Example 2

Suppose you are going to control a CD-ROM drive and you want to open and close the drive door. The bit to open and close the drawer is the 4th bit in a register located at address 0x2DA. Let’s say that to set that bit we use the function

```
IO_Register_Set([addr],[value])
```

where [addr] is the bus address of the I/O register you want to manipulate and [value] is the bit pattern you want to put there.

It is “technically” correct to use the following code:

```
\ open CD-ROM door
IO_Register_Set(0x2DA, 0x10) ;
```
However, if we come back to that code later (1 week or 10 years) we will not know what that statement does (or if the comment is correct) without a lot of research. Instead we use defines for much better code as follows:

```c
// (this part is in the preamble before "main ()"
// or in a special ".h" include file
#define CD_ROM_CONTROL_REG 0x2DA
#define CD_ROM_DOOR_OPEN 0x010

// .... skip to inside the code

// open CD-ROM door
IO_Register_Set(CD_ROM_CONTROL_REG, CD_ROM_DOOR_OPEN) ;
```

Now, isn’t that better?!! It is a requirement of EE472 that your code be written this way. We will take off points for any numerical constants in the main body of code.

# 2 Bitwise vs. Logical Operators

The familiar Boolean operations **AND**, **OR** and **NOT** are provided *two ways* in C.

## 2.1 Logical Operators

These work on *arithmetic* types or pointers.

- **True**: a non zero value.
- **False**: zero.
- **&&**: AND
  - A && B: True if *both* A and B are non-zero.
- **||**: OR
  - A || B: True if *either* A and B are non-zero.
- **!**: NOT

## 2.2 Bitwise Operators

A very important C feature for embedded microcomputer systems. These operators are only defined for integer-like variables i.e. **char**, **short**, **int**, and **long** signed or unsigned.

- **&**: bitwise AND
- **|**: bitwise OR
- **^**: bitwise XOR
- **<<**: left shift
- **>>**: right shift
- **~**: ones complement
2.3 Examples:

/* Use of bitwise logical operators */

#define Bit_Zero 0x01
#define Bit_One 0x02
#define Bit_Two 0x04
#define Bit_Three 0x08
#define Bit_Four 0x10
#define Bit_Five 0x20
#define Bit_Six 0x40

// etc etc etc ..

int x = 0x0B; // x = [... 0 0 0 0 1 0 1 1]
int y = 011; // y = [... 0 0 0 0 1 0 0 1]
int z = x << 2; // z = [... 0 0 1 0 1 1 0 0]
int z1 = y & Bit_Three; // z1 = [... 0 0 0 0 1 0 0 0]
int z2 = z | Bit_Four; // z2 = [... 0 0 1 1 1 1 0 0]
3 In-class Excercise:

Convert 0x1A to decimal: __________

Convert 0xA2 to decimal: __________

Convert 020 to Hex: _________________

Convert 0x20 to Octal: ________________

/* Give the binary value of the least significant 8 bits of the following values: */

int aa = 0xC5; // aa = [ ]

int ab = 017 + 1; // ab = [ ]

int a = x | y; // a = [ ]

int b = Bit_Two | Bit_Five | Bit_Seven;
    // b = [ ]

int c = ~(z | Bit_Zero) << 1;
    // c = [ ]

int d = !(z1 | Bit_Two);
    // d = [ ]
4 Binary Integers

4.1 Unsigned Integer

Typically just a 16 bit binary number. Straightforward, but cannot represent negative numbers.

If \( x \) is of type int, then

\[ 0 \leq x \leq 65535 \]

C declaration of unsigned integers:

```c
unsigned char a,b; // 8 bit unsigned
unsigned int x,y,z; // 16 bit unsigned
unsigned long int p,q,r; // 32 bit unsigned
```

4.1.1 Exercise

Write a C function using bitwise logical operators etc. to print the binary value of a 16 bit word.

4.2 2’s Complement Integers

Most common way to represent integers which can have both positive and negative values. Most significant bit (MSB) is the “sign bit” 0=positive, 1=negative. however, if MSB is 1, other bits are subject to “2’s complement”.

To convert a number to or from two’s complement,

1. complement all bits
2. add 1
Taking the 2’s complement of the entire $n$ bits is equivalent to negation.

For an $n$ bit number, $x$, the range of all possible integers in 2’s complement is

$$-2^{n-1} \leq x \leq (2^{n-1} - 1)$$

(see Lewis Figure 2-4, page 22 for a useful chart).

### 4.2.1 Exercise

Write a c code fragment which negates a signed (2’s complement) integer using bitwise logical operators and addition. Include declarations for your variables. (do not just multiply by -1)