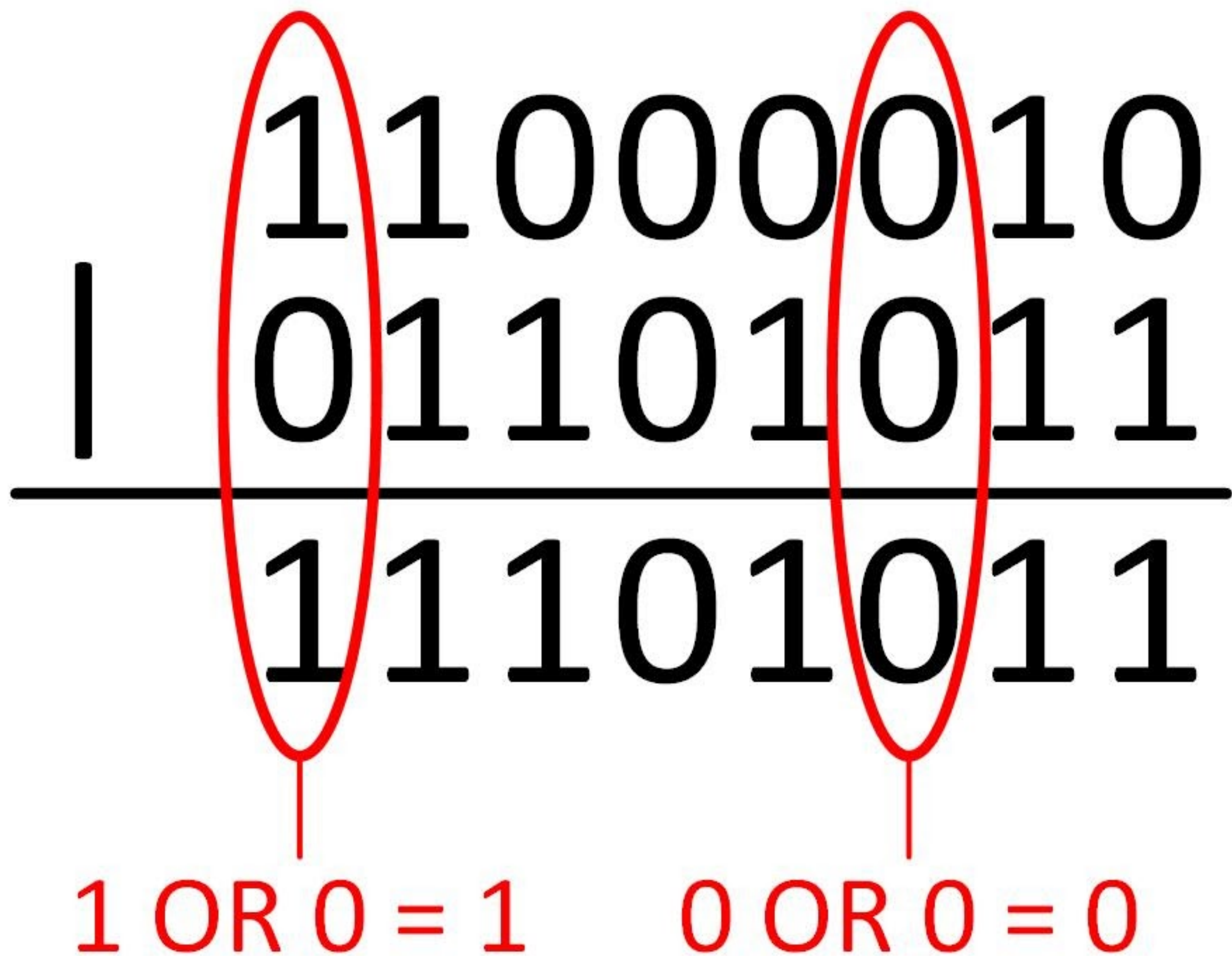
BIT MANIPULATION: BITWISE OPERATIONS

C can perform six different bitwise operations: OR, AND, XOR, NOT, left shift, and right shift. As the names seem to imply, these operations are very similar to their logical counterparts, but there are crucial differences. If an operation is **bitwise**, then it applies to each individual bit in the bitstring. If an operation is **logical**, then it applies to the bitstring as a whole.

The bitwise OR operator is represented as | in C. As we know, an OR gate outputs 1 if and only if at least one of its inputs is 1. From the properties of Boolean algebra, we know that 1 OR X = 1, and 0 OR X = X. Since this is the case, we can say that ORing any bit with 1 sets it to 1, whereas ORing any bit with 0 does not change the bit. As such, you may see the OR operation referred to as a **set** operation. To demonstrate, if we had the bitstrings 11000010 and 01101011, and bitwise OR'd them, the result would be `11000010 | 01101011` = 11101011. Notice that Bits 0, 3, and 5 of the first bitstring went from 0 to 1, since the respective bits in the second bitstring were also 1. Going back to the eight LEDs example, we had the bitstring 00101101, and we wanted to turn LED #1 on. In order to do that, we could bitwise OR that bitstring with 00000010, resulting in `00101101 | 00000010` = 00101111. Notice that the only Bit 1 was changed from the original bitstring; everything else remained unchanged.



The bitwise AND operator is represented as & in C. As previously known, an AND gate outputs 1 if and only if both of its inputs are 1. The properties of Boolean algebra also state that 0 AND X = 0 and 1 AND X = X. This presents opposite behavior as the OR operation. ANDing any bit with 0 clears it to 0, whereas ANDing any bit with 1 leaves it unaffected. Thus, the AND operation can also be called a **clear** or **reset** operation. To put it into focus, let's take the same example as before, but with a bitwise AND instead. So, `11000010 & 01101011` = 01000010. Notice that Bit 7 of the first bitstring went from 1 to 0, since Bit 7 of the second bitstring was also 0. Following up from the eight LEDs example, we turned on LED #1, but now we want to turn off LED #3. Given that our bitstring is 00101111, we could bitwise AND that with 11110111, resulting in `00101111 | 11110111` = 00100111. Notice that only Bit 3 was changed from the original bitstring; everything else remained unchanged.

```
  11000010
& 01101011
----------
  01000010
```

1 AND 0 = 0

1 AND 1 = 1

The bitwise XOR operator is represented as `^` in C. From prior knowledge, we know that a two-input XOR gate outputs 1 if and only if one of its inputs is 1, but not both. Given this nature, we can derive that 1 XOR X = NOT X and 0 XOR X = X. We now have more curious behavior than the previous operations. XORing any bit with 1 returns the complement of that bit, and XORing any bit with 0 leaves the bit unchanged. In other words, the XOR operation is a **toggle** operation. With the same example as with the previous operations, applying a bitwise XOR operation to 11000010 and 01101011 leads to `11000010 ^ 01101011` = 10101001. Notice that Bits 0, 1, 3, 5, and 6 of the original bitstring have the opposite values now, since the respective bits of the second bitstring were 1. Returning to the eight LEDs example, suppose we wanted to arbitrarily toggle the state of LED #4. Now, from our bitstring 00100111, we know that LED #4 is off, so toggling its state would be the same as setting Bit 4 to 1. The usefulness in the toggle operation comes in the case where we either *don't know or don't care about the current state* of LED #4, we just want the *opposite* of it. So, in order to accomplish the task of toggling LED #4, we could bitwise XOR that bitstring with 00010000. Thus, `00100111 ^ 00010000` = 00110111. Notice that only Bit 4 was changed from the original bitstring; everything else remained unchanged.

```
  11000010
^ 01101011
----------
  10101001
```

1 XOR 0 = 1

1 XOR 1 = 0

Bitwise operation	C operator	Function	Output with X and 0	Output with X and 1
OR	|	Sets	X	1
AND	&	Clear	0	X
XOR	^	Toggle	X	NOT X

The bitwise NOT operation is represented as `~` in C. Simply put, a NOT gate outputs the complement of its input. So applying a NOT operation to a bit simply flips it. Thus, a bitwise NOT operation just flips each bit in a bitstring. Unlike the previous operations, the NOT operation only needs one operand (as opposed to two). For example, taking 11000010 and applying a bitwise NOT operation returns `~11000010` = 00111101. Notice that every bit in the original bitstring is now the opposite of its original value. You may notice a similarity with the bitwise XOR operation in that the NOT operation toggles each bit. Indeed, bitwise NOT is the same as bitwise XOR with a bitstring of all 1's (e.g., `~11000010` is the same as `11000010 ^ 11111111`). The important difference is that bitwise XOR is useful for toggling *specific* bits, whereas bitwise NOT is useful for toggling *every* bit. So, for the eight LEDs example, given 00110111, if you want to toggle the state of *every* LED, you could apply a bitwise NOT to that bitstring, resulting in `~00110111` = 11001000. So, LEDs #0, #1, #2, #4, and #5 are all turned off, while LEDs #3, #6, and #7 are all on.

Finally, the shift operations are represented in C as `<<` and `>>` for left shift and right shift, respectively. As a note, these shifts are logical shifts, thus a shifted bit is *replaced with a 0* (as opposed to arithmetic shifts, where right shifts copies the signed bit). Even though the operation itself only requires one operand, in C, the operations take two operands: the bitstring to be shifted, and how many times it should be shifted. For example, `11111111 << 1` = 11111110. This is a left shift performed on 11111111 only once. As opposed to `1111111 << 3` = 11111000, which is a left shift on the same number done three times. This is the same for the right shift operation, where `11111111 >> 1` = 01111111 and `11111111 >> 4` = 00001111. These shift operations aren't as applicable to the eight LEDs example; however, they are very useful for creating bitmasks.

BIT MANIPULATION: AFFECTING A SINGLE BIT

A **bitmask** is a bitstring that can be <u>used in bitwise operations to only affect certain bits</u>. We have already seen examples of a bitmask with the eight LEDs problem. When we wanted to turn LED #1 without affecting the other LEDs, we bitwise OR'd the LED bitstring with the bitmask 00000010. What if we wanted to toggle LED #1 without affecting the other LEDs? We would bitwise XOR the LED bitstring with that same bitmask, 00000010. This bitmask corresponds to Bit 1, which in turn corresponds to LED #1. Similarly, 00010000 is a bitmask for Bit 4, which corresponds to LED #4. In each bitmask, *all the bits are 0 except for the corresponding bit*. In other words, a bitmask can "mask" the other bits so that they are not affected by whatever operation you perform on the desired bit.

Now, what if you wanted to turn off LED #1? Bitwise ANDing the LED bitstring with the bitmask for LED #1 would not work; it would just turn all the other LEDs off while leaving LED #1 unaffected. Instead, we could bitwise AND it with the *complement* of the LED #1 bitmask. We can get the complement of the bitmask by doing a bitwise NOT operation. Thus, `~00000010` = 11111101. If we bitwise AND this with the LED bitstring, only LED #1 is turned off while the others stay the same.

So now we know how to use bitmasks to manipulate the value of each bit. What if we wanted to get the value of a specific bit? As humans, we can easily glance at a bitstring and know what the value of, say, Bit 2 is. For a computer, "looking" at a specific bit equates to **reading** its value. In order to read a specific bit's value, we turn to bitmasks once again, this time using the bitwise AND operation to isolate the bit. This makes all other bits 0 while retaining the value of the bit you want. Let us say that the state of the LEDs is stored in variable a and you need to see if LED #2 is on or off, then the following expression isolates bit 2 of the LED status. If the expression becomes zero, it means LED #2 is off and if the expression become non-zero, it means LED #2 is on. Note that since we are not writing back this value to a, it does not affect a and therefore the lights are unchanged.

```
(a & 0b00000100)
```

The below table summarizes the bitwise operations with masks. Note that in all operations except when clearing a bit, the mask is used directly. However, when we need to clear a certain bit, we need to use the complement of the mask for that bit and AND it with the original bitstring.

Operation	Bitstring A, Bitmask B
Set	A | B
Clear	A & ~B
Toggle	A ^ B
Read	A & B

BIT MANIPULATION: CREATING BITMASKS

Creating a bitmask for a specific bit is as simple as starting with a bitstring of 0's and flipping the desired bit. So for a bitmask for Bit 0, it is as simple as starting with 00000000 (for an 8-bit bitmask), then flipping the 0th bit, resulting in 00000001. Although this starts off as simple for shorter bitstrings, it's easy to make mistakes when working with larger bitstrings. For instance, it may be difficult to create a bitmask for Bit 22 of a 32-bit bitstring using this method without possibly making errors.

There is, however, a simpler method. Notice that the bitmask for Bit 1 is 00000010, which is equivalent to the decimal number 2, which is also 2^1. Similarly, the bitmask for Bit 2 is 00000100 = 4 = 2^2. The bitmask for Bit 3 is 00001000 = 8 = 2^3, and the pattern continues. Notice that the bitmask for a Bit n is the binary equivalent to 2^n. Recall that if you have an unsigned binary number and you logical shift it to the left, it is the same as multiplying that binary number by 2. If you logical shift it to the left twice, it is the same as multiplying by 4, or 2^2. Similarly, the logical shift to the left three times is the same as multiplying by 8, or 2^3. Notice that doing a logical shift left n times is the same as multiplying that number by 2^n. Thus, we can use the left shift operation in order to create bitmasks. Let's start with making a bitmask for Bit 0. The bitmask is 00000001 = 1 = 2^0 = `1 << 0`. To get the bitmask for Bit 1, which is 2, we need to multiply 1 by 2. In other words, `1 << 1` gives the bitmask for Bit 1. To get the bitmask for Bit 22, which is 2^22, we need to multiply 1 by 2^22, or `1 << 22`. In summary, creating bitmask in C for a Bit n is as simple as `1 << n`.

In the driverlib provided for MSP432, single-bit masks are defined as macros that you can readily use. See the below image. BIT0 is the macro that all the bits are 0 except at bit index 0. The first 16 bitmasks are defined as BIT* where * is the hex digit representing the index of the single bit that is 1 in the bitmask. For indices larger than 15, you need to use BIT(x) format where x is the index. For example, BIT(25) is the bitmask that all bits are 0 except for index 25.

Getting Started | msp432p401r.h

```
239 #include "system_msp432p401r.h"
240
241 /******************************************************************************
242 * Definition of standard bits                                                *
243 ******************************************************************************/
244 #define BIT0                                        (uint16_t)(0x0001)
245 #define BIT1                                        (uint16_t)(0x0002)
246 #define BIT2                                        (uint16_t)(0x0004)
247 #define BIT3                                        (uint16_t)(0x0008)
248 #define BIT4                                        (uint16_t)(0x0010)
249 #define BIT5                                        (uint16_t)(0x0020)
250 #define BIT6                                        (uint16_t)(0x0040)
251 #define BIT7                                        (uint16_t)(0x0080)
252 #define BIT8                                        (uint16_t)(0x0100)
253 #define BIT9                                        (uint16_t)(0x0200)
254 #define BITA                                        (uint16_t)(0x0400)
255 #define BITB                                        (uint16_t)(0x0800)
256 #define BITC                                        (uint16_t)(0x1000)
257 #define BITD                                        (uint16_t)(0x2000)
258 #define BITE                                        (uint16_t)(0x4000)
259 #define BITF                                        (uint16_t)(0x8000)
260 #define BIT(x)                                     ((uint16_t)1 << (x))
261
262 /******************************************************************************
```

You can also create bitmasks that control multiple bits. Using the eight LEDs example once more, say the bitstring is 10100110, and you wanted to turn on LED #3 and #4. While you could turn them on one at a time, a more efficient approach would be to turn them on simultaneously. In other words, we can bitwise OR the LED bitstring with 00011000, leading to `10100110 | 00011000` = 10111110. Notice that 00011000 is a combination of the bitmasks for Bits 3 and 4. In other words, 00011000 = `00010000 | 00001000`. So, in order to combine bitmasks, use a bitwise OR operation. Let's say you wanted to turn off LED #2 and #7. Combining the bitmasks for those LEDs gives you `00000100 | 10000000` = 10000100. Bitwise ANDing the LED bitstring with the complement of this combined bitmask gives you `10111110 & ~10000100` = `10111110 & 01111011` = 00111010.

In many cases, we are interested to create a bitmask for several adjacent bits. For example, let us say we are interested to LED #3, #4, #5, and #6. There are 4 '1's in this mask that start from position Bit 3 and go up. This is similar to having the binary number `00001111` and shifting it 3 times, which gives us `01111000` .This means we can create a mask by putting as many '1's as the mask needs at the left-most side of the bitstring and then shifting it as many times as the lowest bit position of the mask.