Subsections

Low Level Operators and Bit Fields

We have seen how pointers give us control over low level memory operations.

Many programs (***e.g.*** systems type applications) must actually operate at a low level where individual bytes must be operated on.

NOTE: The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code. (Only about 10 % of UNIX is assembly code the rest is C!!.)

Bitwise Operators

The ***bitwise*** operators of C a summarised in the following table:

Table: Bitwise operators

&	AND
|	OR
$\wedge$	XOR
$\sim$	One's Compliment
	$0 \rightarrow 1$
	$1 \rightarrow 0$
<<	Left shift
>>	Right Shift

DO NOT confuse & with &&: & is bitwise AND, && logical AND. Similarly for $\vert$ and $\vert\vert$.

$\sim$ is a unary operator -- it only operates on one argument to right of the operator.

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (***i.e.*** There is **NO** wrap around).

For example: x << 2 shifts the bits in x by 2 places to the left.

So:

if x = 00000010 (binary) or 2 (decimal) then:

$x >>= 2 \Rightarrow x = 00000000$ or 0 (decimal)

Also: if *x* = 00000010 (binary) or 2 (decimal)

$x <<= 2 \Rightarrow x = 00001000$ or 8 (decimal)

Therefore a shift left is equivalent to a multiplication by 2.

Similarly a shift right is equal to division by 2

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 ***use shifts***.

To illustrate many points of bitwise operators let us write a function, `Bitcount`, that counts bits set to 1 in an 8 bit number (`unsigned char`) passed as an argument to the function.

```
int bitcount(unsigned char x)

        { int count;

                for (count=0; x != 0; x>>=1)
                        if ( x & 01)
                           count++;
                return count;
        }
```

This function illustrates many C program points:

- `for loop not used for simple counting operation`
- x$>>=1 \Rightarrow$ `x = x >> 1`
- `for loop will repeatedly shift right x until x becomes 0`
- `use expression evaluation of x & 01 to control if`
- `x & 01` ***masks*** `of 1st bit of x if this is 1 then count++`

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. ***e.g.*** 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. ***E.g.*** 9 bit integers.

C lets us do this in a structure definition by putting :***bit length*** after the variable. ***i.e.***

```
struct packed_struct {
                 unsigned int f1:1;
                 unsigned int f2:1;
                 unsigned int f3:1;
                 unsigned int f4:1;
                 unsigned int type:4;
                 unsigned int funny_int:9;
} pack;
```

Here the `packed_struct contains 6 members: Four 1 bit` ***flags*** `f1..f3, a 4 bit type and a 9 bit funny_int.`

`C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit fiels portability below).`

`Access members as usual via:`

```
   pack.type = 7;
```

NOTE:

- `Only` *n* `lower bits will be assigned to an` *n* `bit number. So type cannot take values larger than 15 (4 bits long).`
- `Bit fields are always converted to integer type for computation.`
- `You are allowed to mix ``normal'' types with bit fields.`
- `The unsigned definition is important - ensures that no bits are used as a` $\pm$ `flag.`

Bit Fields: Practical Example

Frequently device controllers (*e.g.* disk drives) and the operating system need to communicate at a low level. Device controllers contain several ***registers*** which may be packed together in one integer (Figure 12.1).

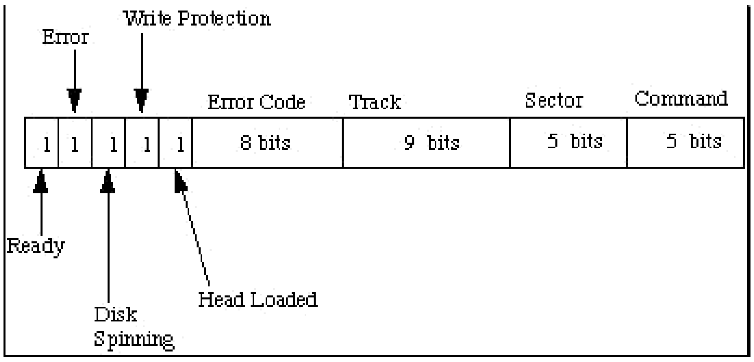

Fig. 12.1 Example Disk Controller Register We could define this register easily with bit fields:

```
struct DISK_REGISTER  {
     unsigned ready:1;
     unsigned error_occured:1;
     unsigned disk_spinning:1;
     unsigned write_protect:1;
     unsigned head_loaded:1;
     unsigned error_code:8;
     unsigned track:9;
     unsigned sector:5;
     unsigned command:5;
};
```

To access values stored at a particular memory address, `DISK_REGISTER_MEMORY` we can assign a pointer of the above structure to access the memory via:

```
struct DISK_REGISTER *disk_reg = (struct DISK_REGISTER *) DISK_REGISTER_MEMORY;
```

The disk driver code to access this is now relatively straightforward:

```
/* Define sector and track to start read */

disk_reg->sector = new_sector;
disk_reg->track = new_track;
```

```
disk_reg->command = READ;

/* wait until operation done, ready will be true */

while ( ! disk_reg->ready ) ;

/* check for errors */

if (disk_reg->error_occured)
  { /* interrogate disk_reg->error_code for error type */
    switch (disk_reg->error_code)
    ......
  }
```

A note of caution: Portability

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an `integer` which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

If portability of code is a premium you can use bit shifting and masking to achieve the same results but not as easy to express or read. For example:

```
unsigned int  *disk_reg = (unsigned int *) DISK_REGISTER_MEMORY;

/* see if disk error occured */

disk_error_occured = (disk_reg & 0x40000000) >> 31;
```

Exercises

Exercise 12507

Write a function that prints out an 8-bit (unsigned char) number in binary format.

Exercise 12514

Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of an unsigned char variable y (leaving other bits unchanged).

E.g. if x = 10101010 (170 decimal) and y = 10100111 (167 decimal) and n = 3 and p = 6 say then you need to strip off 3 bits of y (111) and put them in x at position 10*xxx*010 to get answer 10111010.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
   x = 10101010 (binary)
   y = 10100111 (binary)
   setbits n = 3, p = 6 gives x = 10111010 (binary)
```

Exercise 12515

Write a function that inverts the bits of an unsigned char x and stores answer in y.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10101010 (binary)
x inverted = 01010101 (binary)
```

Exercise 12516

Write a function that rotates (**NOT shifts**) to the right by n bit positions the bits of an unsigned char x.ie no bits are lost in this process.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10100111 (binary)
x rotated by 3 = 11110100 (binary)
```

Note: All the functions developed should be as concise as possible

---

Dave Marshall
1/5/1999