ELEC3730 Embedded Systems
Lecture 7: C for Embedded Programming

- **Bitwise Operations**
- **Setting,Inverting,Toggling,Clearing,Extracting + Inserting bits**
- **Structures**
- **Unions**

Boolean and Binary Operators

Operation	Boolean Operator	Bitwise Operator
AND	&&	&
OR	||	|
NOT	!	~
XOR	N/A	^
LEFT SHIFT	N/A	<<
RIGHT SHIFT	N/A	>>

- Boolean operators used to for conditional expressions (eg: *if* statement)
  - C does not contain a Boolean data type.
  - Boolean operators yield results of type *int*, with true and false represented by 1 and 0.
  - Any numeric data type may be used as a Boolean operand.
  - Zero is interpreted as false; any non-zero value is interpreted as true.
- Bitwise operators are used to manipulate bits.
  - Bitwise operators operate on individual bit positions within the operands;
  - The result in one bit position is independent of all the other bit positions.

Boolean versus bitwise operations

(5 || !3) && 6

= (true OR (NOT true)) AND true
= (true OR false) AND true
= (true) AND true
= true
= 1

(5 | ~3) & 6

= (00..0101 OR ~00..0011) AND 00..0110
= (00..0101 OR 11..1100) AND 00..0110
= (11..1101) AND 00..0110
= 00..0100
= 4

Bitwise-AND: Forces 0's where they occur in mask

m	p		m AND p	Interpretation
0	0	0	0	If bit *m* of the mask is 0, bit *p* is **cleared to 0** in the result.
	1	0		
1	0	0	p	If bit *m* of the mask is 1, bit *p* is passed through to the result **unchanged**.
	1	1		

Bitwise-OR: Forces 1's where they occur in mask

m	p		m OR p	Interpretation
0	0	0	p	If bit *m* of the mask is 0, bit *p* is passed through to the result **unchanged**.
	1	1		
1	0	1	1	If bit *m* of the mask is 1, bit *p* is **set to 1** in the result.
	1	1		

Bitwise-XOR: Forces bit flip where 1's occur in mask

m	p		m XOR p	Interpretation
0	0	0	p	If bit *m* of the mask is 0, bit *p* is passed through to the result **unchanged**.
	1	1		
1	0	1	~p	If bit *m* of the mask is 1, bit *p* is passed through to the result **inverted**.
	1	0		

Testing Bits

- Form a mask with 1 in the bit position of interest;
- Bitwise AND the mask with the operand.
- The result is non-zero if and only if the bit of interest was 1:

```
if  ((bits  &  64) != 0)   /* check to see if bit 6 is set */
```

- Same as:

```
if  (bits  &  0x64)        /* check to see if bit 6 is set *
```

- Same as

```
if  (bits & (1 << 6))      /* check to see if bit 6 is set */
```

```
if  ((bits  &  64) != 0)     /* check to see if bit 6 is set */
```

$b_7b_6b_5b_4b_3b_2b_1b_0$ & 01000000 ➡ $0b_6000000$

Setting bits, Inverting bits

- Setting a bit to 1 is easily accomplished with the bitwise-OR operator:

```
bits = bits | (1 << 7) ;   /* sets bit 7 */
```

$b_7b_6b_5b_4b_3b_2b_1b_0$ | 10000000 ➡ $1b_6b_5b_4b_3b_2b_1b_0$

- This would usually be written more succinctly as:

```
bits |= (1 << 7) ;        /* sets bit 7 */
```

- Inverting (toggling) is accomplished with bitwise-XOR:

```
bits  ^=  (1 << 6) ;       /* flips bit 6 */
```


Clearing Bits

- Clearing a bit to 0 is accomplished with the bitwise-AND operator:

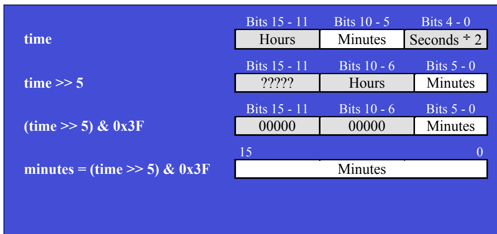
```
bits &= ~(1 << 7) ; /* clears bit 7 */
```

```
 (1 << 7)   ➡   10000000
~(1 << 7)   ➡   01111111
```

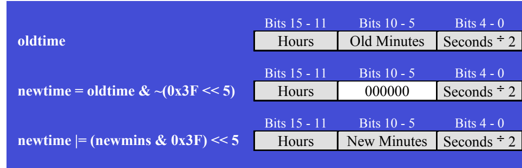
- Mask must be as wide as the operand!
- if `bits` is a 32-bit data type, the assignment must be 32-bit:

```
bits &= ~(1L << 7) ;     /* clears bit 7 */
```

Extracting Bits



Inserting Bits



Structures in C

```
#include<stdio.h>

struct{
  float x;
  int y;
  char c;
} brett;

int main(void)
{
  brett.x=2.345;
  brett.y=5;
  brett.c='b';
  printf("x,y,c in brett is %f,%d,%c\n",brett.x,brett.y,brett.c);
}
```

Structures in C - Use typedef to define

```
#include<stdio.h>

typedef struct{
  float x;
  int y;
  char c;
} elec3730;

int main(void)
{

  elec3730 brett;

  brett.x=2.345;
  brett.y=5;
  brett.c='b';
  printf("x,y,c in brett is %f,%d,%c\n",brett.x,brett.y,brett.c);

}
```


Pointers to Structures

```
#include<stdio.h>
#include<stdlib.h>

typedef struct{
  float x;
  int y;
  char c;
} elec3730;

int main(void)
{

  elec3730 *brett;

  brett=malloc(sizeof(elec3730));
  if (brett != NULL)
    {
      (*brett).x=2.345;
      brett->y=5;
      brett->c='b';
      printf("x,y,c in brett is %f,%d,%c\n",brett->x,(*brett).y,(*brett).c);
      free(brett);
    }
}
```


Testing Keyboard Flags Using Bitwise Operators.

```
#define    FALSE      0
#define    TRUE       1

typedef unsigned char  BOOL ;

typedef struct
{
     BOOL  right_shift    ;
     BOOL  left_shift     ;
     BOOL  ctrl      ;
     BOOL  alt       ;
     BOOL  left_ctrl;
     BOOL  left_alt ;
} KEYS ;
```

```
BOOL get_keys(KEYS  *) ;

void main(){
     KEYS  kybd ;
     do{ /* wait for both shift keys*/
          get_keys(&kybd)
     }
      while (!kybd.left_shift || !kybd.right_shift) ;
}
```

```
void get_keys(KEYS  *kybd)
{
int hit, *addr;

addr=0x417; hit=*addr;

kybd->right_shift   = (hit & (1 << 0)) != 0 ;
kybd->left_shift    = (hit & (1 << 1)) != 0 ;
kybd->ctrl          = (hit & (1 << 2)) != 0 ;
kybd->alt           = (hit & (1 << 3)) != 0 ;
kybd->left_alt      = (hit & (1 << 9)) != 0 ;
kybd->left_ctrl     = (hit & (1 << 8)) != 0 ;

}
```

Automatic Insertion/Extraction Using Structure Bit Fields

```
struct {
unsigned      seconds       :5 ,   /* secs divided by 2 */
              minutes       :6 ,
              hours         :5 ;
} time ;

time.hours = 13 ;
time.minutes = 34 ;
time.seconds = 18 / 2 ;
```

Leave the insertion (or extraction) problems to the compiler!

Keyboard Revisited - Using Structure Bit Fields

```
#define    FALSE     0
#define    TRUE      1

typedef unsigned char  BOOL ;

typedef struct
{
unsigned   right_shift    :1,
           left_shift     :1,
           ctrl           :1,
           alt            :1,
                          :4,
           left_ctrl      :1,
           left_alt       :2;
} KYBD_BITS;
```

```
BOOL get_keys(KEYS  *) ;

void main(){
     KEYS  kybd ;
     do{ /* wait for both shift keys*/
          get_keys(&kybd)
     }
      while (!kybd.left_shift || !kybd.right_shift) ;
}
```

```
void get_keys(KEYS  *kybd)
{
int hit, *addr;
KYBD_BITS *bits ;

addr=0x417; hit=*addr;

bits = (KYBD_BITS *) &hit ;
kybd->right_shift   = bits->right_shift != 0 ;
kybd->left_shift    = bits->left_shift  != 0 ;
kybd->ctrl          = bits->ctrl        != 0 ;
kybd->alt           = bits->alt         != 0 ;
kybd->left_alt      = bits->left_alt    != 0 ;
kybd->left_ctrl     = bits->left_ctrl   != 0 ;

}
```


Variant Access with Pointers, Casts, & Subscripting

- Given an address:
  - *Cast it as a pointer to data of the desired type,*
  - *Then dereference the pointer by subscripting.*
- Without knowing the data type used in its declaration:
  - *Read or write various parts of an object named* `operand` *using:*

((char *) &operand)[k]

Keyboard Revisited - Using Pointers+Cast+Subscripting

```
#define    FALSE      0
#define    TRUE       1

typedef unsigned char  BOOL ;

typedef struct
{
     char   lo ;
     char   hi ;
     short  both ;
} KEYS;
```

```
BOOL get_keys(KEYS  *) ;

void main(){
     KEYS  kybd ;
     do{ /* wait for both shift keys*/
          get_keys(&kybd)
     }
      while (!kybd.left_shift || !kybd.right_shift) ;
}
```

```
void get_keys(KEYS  *kybd)
{
int hit, *addr;

addr=0x417; hit=*addr;

kybd->both = ((short *) &hit)[0] ;
kybd->lo   = ((char  *) &hit)[0] ;
kybd->hi   = ((char  *) &hit)[1] ;

}
```


Variant Access with Unions

```
union {
    unsigned long  dd ;
    unsigned short dw[2] ;
    unsigned char  db[4] ;
} ;
```

31 … 24	23 … 16	15 … 8	7 … 0
dd			
dw[1]		dw[0]	
db[3]	db[2]	db[1]	db[0]

Keyboard Revisited - Using Unions

```
#define    FALSE      0
#define    TRUE       1

typedef unsigned char  BOOL ;

typedef union
{
     char  b[2] ;
     short int w ;
} VARIANT ;
```

```
BOOL get_keys(KEYS  *) ;

void main(){
     KEYS  kybd ;
     do{ /* wait for both shift keys*/
          get_keys(&kybd)
     }
      while (!kybd.left_shift || !kybd.right_shift) ;
}
```

```
void get_keys(KEYS  *kybd)
{
VARIANT  *hit;

hit=0x417;

kybd->both = hit->w ;
kybd->lo   = hit->b[0] ;
kybd->hi   = hit->b[1] ;

}
```