TDDI11: Embedded Software

C for Embedded Systems II

Lecture outline

- Structures
- Unions
- Endianness
- Bitfield
- Bit manipulation

Lecture outline

- **Structures**
- Unions
- Endianness
- Bitfield
- Bit manipulation

Structures

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees
  - Can hold the data associated to a hardware device

Struct content

- A `struct` cannot contain an instance of itself
- Can contain a member that is a pointer to the same structure type
- A structure definition does not reserve space in memory
  - Instead creates a new data type used to define structure variables

Structure definitions, cont

- Valid operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the sizeof operator to determine the size of a structure

Using structures with functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

Let's do some code

A small interlude (typedefs)

- typedef
  - Creates synonyms (aliases) for previously defined data types
  - Use typedef to create shorter type names
  - Example:

    ```
    typedef struct Card *CardPtr;
    ```
  - Defines a new type name CardPtr as a synonym for type struct Card *
  - typedef does not create a new data type
    - Only creates an alias

typedefs increase readability

```
unsigned long int count ;
```

versus

```
typedef unsigned long int DWORD32 ;
DWORD32 count ;
```

typedefs and #defines

```
typedef unsigned char           BYTE8 ;
typedef unsigned short int      WORD16 ;
typedef unsigned long int       DWORD32 ;


typedef int                     BOOL ;
#define FALSE                   0
#define TRUE                    1
```

Lecture outline

- Structures
- **Unions**
- Endianness
- Bitfield
- Bit manipulation

Unions

- Memory that contains a variety of objects over time
- Members of a union share the same space
  - Only one data member at a time
  - Only the last data member defined can be accessed
  - Conserves storage

Size of unions

```
union Data
{
   int i;
   float f;
   char str[20];
} data;
```

The memory occupied by a union will be large enough to hold the largest member of the union.

For example, in above example Data type will occupy 20 bytes.

Unions

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

Variant access with pointers, casts, & subscripting

- Given an address, we can cast it as a pointer to data of the desired type, then deference the pointer by subscripting.
- Without knowing the data type used in its declaration, we can read or write various parts of an object named *operand* using:

((BYTE8 *) &operand)[k]

Variant access with pointers, casts, & subscripting, cont.

```
typedef struct KYBD_INFO
        {
        BYTE8     lo ;
        BYTE8     hi ;
        WORD16  both ;
        } KYBD_INFO ;
```

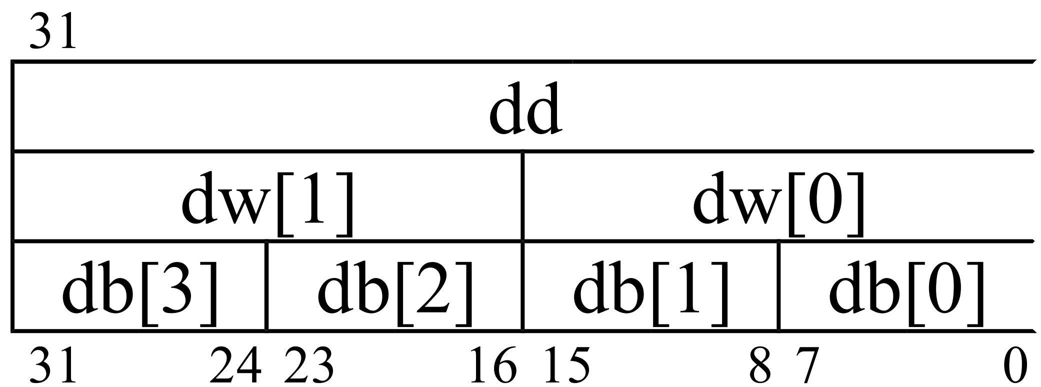
```
BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
        {
        …...
        kybd->both   = ((WORD16  *) &new_flags)[0] ;
        kybd->lo       = ((BYTE8  *)    & new_flags)[0] ;
        kybd->hi       = ((BYTE8  *)     &new_flags)[1] ;

        if (kybd->both == old_flags) return FALSE ;
        old_flags = kybd->both ;

        return TRUE ;
        }
```

Variant access with unions

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



Example of variant access with unions

```
typedef union VARIANT {
      BYTE8          b[2] ;
      WORD16         w ;
} VARIANT ;

BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
      {
      static WORD16 old_flags = 0xFFFF ;
      VARIANT *flags = (VARIANT *) malloc(sizeof(VARIANT)) ;

      dosmemget(0x417, sizeof(VARIANT), (void *) flags) ;

      status->both         = flags->w ;
      status->lo           = flags->b[0] ;
      status->hi           = flags->b[1] ;
      free(flags) ;

      if (status->both == old_flags) return FALSE ;
      old_flags = status->both ;
      return TRUE ;
      }
```

One more small code example

Lecture outline

- Structures
- Unions
- **Endianness**
- Bitfield
- Bit manipulation

Endianness

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

31.......................................0

dd			
dw[1]		dw[0]	
db[3]	db[2]	db[1]	db[0]

31......24|23......16|15........8|7........0

Little endian

vs

Big endian

31.......................................0

dd			
dw[0]		dw[1]	
db[0]	db[1]	db[2]	db[3]

31......24|23......16|15........8|7........0

Endianness differ depending on architecture.
X86: little
Motorola, sparc: big

Endianness

- **Big-endian** systems are systems in which the *most significant byte* of the word is stored in the *smallest address* given and the least significant byte is stored in the largest. In contrast, **little endian** systems are those in which the *least significant byte* is stored in the *smallest address*.

Why is Endianness important for embedded software developers?

- Think about communication between two machine that have different Endianness
- One machine writes integers to a file and another machine with opposite Endianness reads it.
- Sending numbers over network between two machines with different Endianess. Think about serial communication when we split the data into mulitple chunks !!

Trivia!

- Where does this term ‘Endian’ come from?

Trivia!

- Excellent read: http://www.ietf.org/rfc/ien/ien137.txt
- Quote:
- “It may be interesting to notice that the point which Jonathan Swift tried to convey in Gulliver's Travels in exactly the opposite of the point of this note.
- Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.
- We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made.”

Lecture outline

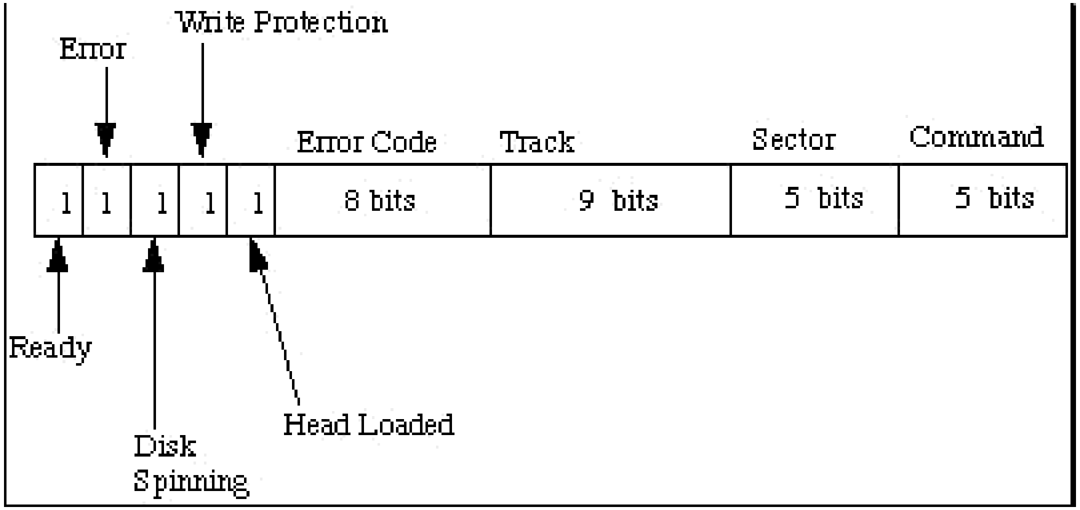
- Structures
- Unions
- Endianness
- **Bitfield**
- Bit manipulation

Bitfield

- In embedded systems, storage is at a premium
- It may be necessary to pack several objects into one word
- Bitfields allow single bit objects
- They must be part of structure

Bitfield example

- Embedded systems must communicate with peripherals at low-level.
- A register of a disk controller, for example, has several fields.



- How can we represent this is memory compactly?

Bitfield example

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

- Bit fields must be part of a structure/union – stipulated by the C standard

Lecture outline

- Structures
- Unions
- Endianness
- Bitfield
- **Bit manipulation**

Boolean and binary operators

<table>
<tr><th>Operation</th><th>Boolean Operator</th><th>Bitwise Operator</th></tr>
<tr><td>AND</td><td>&&</td><td>&</td></tr>
<tr><td>OR</td><td>||</td><td>|</td></tr>
<tr><td>XOR</td><td>unsupported</td><td>^</td></tr>
<tr><td>NOT</td><td>!</td><td>~</td></tr>
</table>

- Boolean operators are primarily used to form conditional expressions (as in an *if* statement)
- Bitwise operators are used to manipulate bits.

Boolean values

- Most implementations of C don't provide a Boolean data type.
- Any numeric data type may be used as a Boolean operand.
- Boolean operators yield results of type *int*, with true and false represented by 1 and 0.
- Zero is interpreted as false; any non-zero value is interpreted as true.

Boolean expressions

```
(5 || !3) && 6
```

True / False ?

= (true OR (NOT true)) AND true

= (true OR false) AND true

= (true) AND true

= true

= 1

Bitwise operators

- Bitwise operators operate on individual bit positions within the operands

- The result in any one bit position is entirely independent of all the other bit positions.

Interpreting the bitwise-AND

<table>
<tr><th>m</th><th>p</th><th colspan="2">m AND p</th><th>Interpretation</th></tr>
<tr><td rowspan="2">0</td><td>0</td><td>0</td><td rowspan="2">0</td><td rowspan="2">If bit m of the mask is 0, bit p is cleared to 0 in the result.</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>0</td><td rowspan="2">p</td><td rowspan="2">If bit m of the mask is 1, bit p is passed through to the result unchanged.</td></tr>
<tr><td>1</td><td>1</td></tr>
</table>

Interpreting the bitwise-OR

<table>
<tr><th>m</th><th>p</th><th colspan="2">m OR p</th><th>Interpretation</th></tr>
<tr><td rowspan="2">0</td><td>0</td><td>0</td><td rowspan="2">p
.</td><td rowspan="2">If bit m of the mask is 0, bit p is passed through to the result unchanged.</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>1</td><td rowspan="2">1
.</td><td rowspan="2">If bit m of the mask is 1, bit p is set to 1 in the result.</td></tr>
<tr><td>1</td><td>1</td></tr>
</table>

Interpreting the bitwise-XOR

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

Bitwise expressions

$$(5 \mid \sim 3) \; \& \; 6$$

= (00..0101 OR ~00..0011) AND 00..0110

= (00..0101 OR 11..1100) AND 00..0110

= (11..1101) AND 00..0110

= 00..0100

= 4

Testing bits

- A 1 in the bit position of interest is AND'ed 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 */
```

$$b_7b_6b_5b_4b_3b_2b_1b_0 \;\&\; 01000000 \Rightarrow 0b_6000000$$

Testing bits, cont.

- Since any non-zero value is interpreted as *true*, the redundant comparison to zero may be omitted, as in:

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

Testing bits, cont.

- The mask (64) is often written in hex (0x0040), but a constant-valued shift expression provides a clearer indication of the bit position being tested:

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

- Almost all compilers will replace such constant-valued expressions by a single constant, so using this form almost never generates any additional code.

Testing keyboard flags using bitwise operators

```
#define        FALSE       (0)
#define        TRUE        (1)

typedef unsigned char  BOOL ;

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

```
BOOL      Kybd_Flags_Changed(SHIFTS  *) ;
void  Display_Kybd_Flags(SHIFTS  *) ;

void main()
        {
        SHIFTS  kybd ;

        do
                { /* repeat until both shift keys are pressed */
                if (Kybd_Flags_Changed(&kybd))
                                Display_Kybd_Flags(&kybd) ;
                } while (!kybd.left_shift || !kybd.right_shift) ;
        }
```

continued …

```
typedef unsigned int WORD16 ;

BOOL Kybd_Flags_Changed(SHIFTS  *kybd)
        {
        static WORD16 old_flags = 0xFFFF ;
        WORD16 new_flags ;

        dosmemget(0x417, sizeof(new_flags), &new_flags) ;
        if (new_flags == old_flags) return FALSE ;
        old_flags = new_flags ;

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

        return TRUE ;
        }
```

Setting bits

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

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

$b_7b_6b_5b_4b_3b_2b_1b_0 \;|\; 10000000 \Rightarrow 1b_6b_5b_4b_3b_2b_1b_0$

- This would usually be written more succinctly as:

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

Setting bits, cont.

- Note that we don't *add* (+) the bit to the operand! That only works if the current value of the target bit in the operand is known to be 0.

- Although the phrase *"set a bit to 1"* suggests that the bit was originally 0, most of the time the current value of the bit is actually unknown.

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**

- Note that we don't *subtract* the bit from the operand!

Clearing bits, cont.

- When clearing bits, you have to be careful that the mask is as wide as the operand. For example, if *bits* is changed to a 32-bit data type, the right-hand side of the assignment must also be changed, as in:

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

Inverting bits

- Inverting a bit (also known as toggling) is accomplished with the bitwise-XOR operator as in:

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

- Although *adding* 1 would invert the target bit, it may also propagate a carry that would modify more significant bits in the operand.

Extracting bits

Extract minutes from time

	Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
time	Hours	Minutes	Seconds ÷ 2
	Bits 15 - 11	Bits 10 - 6	Bits 5 - 0
time >> 5	?????	Hours	Minutes
	Bits 15 - 11	Bits 10 - 6	Bits 5 - 0
(time >> 5) & 0x3F	00000	00000	Minutes
	15		0
minutes = (time >> 5) & 0x3F	Minutes		

Inserting bits

Updates minutes in time

oldtime

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	Old Minutes	Seconds ÷ 2

newtime = oldtime & ~(0x3F << 5)

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	000000	Seconds ÷ 2

newtime |= (newmins & 0x3F) << 5

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	New Minutes	Seconds ÷ 2

Extra material

Structure definitions

- Example

```
struct card {
 char *face;
 char *suit;
};
```

  - struct introduces the definition for structure card
  - card is the structure name and is used to declare variables of the structure type
  - card contains two members of type char *
    - These members are face and suit

Defining structure variables

- Defined like other variables:

```
struct card oneCard, deck[ 52 ], *cPtr;
```

- Can use a comma separated list:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

Initializing structures

- Initializer lists
  - Example:

```
struct card oneCard = { "Three", "Hearts" };
```

- Assignment statements
  - Example:

```
card threeHearts = oneCard;
```

  - Could also define and initialize threeHearts as follows:

```
card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

Accessing members of structures

- Accessing structure members
  - Dot operator (**.**) used with structure variables
    ```
    struct card myCard;
    printf( "%s", myCard.suit );
    ```
  - Arrow operator (**->**) used with pointers to structure variables
    ```
    struct card *myCardPtr = &myCard;
    printf( "%s", myCardPtr->suit );
    ```
  - myCardPtr->suit is equivalent to
    ```
    ( *myCardPtr ).suit
    ```

Union definitions

- Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```