Basic Names for Integer Types:

- **char** and **int**
  - *unsigned, signed, short,* and *long* are *modifiers* (adjectives).
  - If one or more modifiers is used, **int** is assumed and may be omitted.
  - If neither *signed* or *unsigned* is used, **signed** is assumed and may be omitted.
  - **int** with no modifier is a data type whose size varies among compilers.

---

**Integer Data Types**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>char</td>
<td>8 bits</td>
</tr>
<tr>
<td></td>
<td>short int</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>16 or 32 bits</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>32 bits</td>
</tr>
<tr>
<td>signed</td>
<td>char</td>
<td>8 bits</td>
</tr>
<tr>
<td></td>
<td>short int</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>16 or 32 bits</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

---

**Contents of ANSI File (limits.h)**

Notice anything odd?

```c
#define SCHAR_MIN            -127
#define SCHAR_MAX             127
#define SHRT_MIN           -32767
#define SHRT_MAX            32767
#define INT_MIN       -2147483647
#define INT_MAX        2147483647
#define LONG_MIN      -2147483647
#define LONG_MAX     2147483647
```
Mixing Signed & Unsigned Integers

Always TRUE or always FALSE?

unsigned int u;
...
if (u > -1) ...

The constant -1 is signed, but when combined with an unsigned, the bit pattern used to represent \(-1\) is interpreted as unsigned with a full-scale value! Thus regardless of the value in \(u\), the condition will always be FALSE.

typedefs

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

#defines, continued

#define ENTRIES(a) (sizeof(a)/sizeof(a[0]))
unsigned counts[100];
for (i = 0; i < 100; i++) ...

for (i = 0; i < ENTRIES(counts); i++) ...
Careful: #define is text substitution!

```c
#define SUM(a, b)  a + b
y = 5 * SUM(t1, t2) ;
Becomes...
y = 5 * t1 + t2 ;
```

```c
#define TIMES(a, b)  (a * b)
y = TIMES(t1-2, t2+3) ;
Becomes...
y = (t1 – 2 * t2 + 3) ;
```

---

Packed Operands

<table>
<thead>
<tr>
<th>Bits 15 - 11</th>
<th>Bits 10 - 5</th>
<th>Bits 4 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hours</td>
<td>Minutes</td>
<td>Seconds × 2</td>
</tr>
</tbody>
</table>

MS/DOS packed representation of time.

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carrier Detect</td>
<td>Ring Indicator</td>
<td>Data Set Ready</td>
<td>Clear To Send</td>
<td>Δ Carrier Detect</td>
<td>Ring Indicator</td>
<td>Δ Data Set Rdy</td>
<td>Δ Clear To Send</td>
</tr>
</tbody>
</table>

UART modem status port

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Enable IRQ</td>
<td>Select Printer</td>
<td>Initialize Printer</td>
<td>Auto LineFeed</td>
<td>Data Strobe</td>
</tr>
</tbody>
</table>

IBM-PC printer control port

---

Boolean and Binary Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Boolean Operator</th>
<th>Bitwise Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;&amp;</td>
<td>&amp;</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>
</tbody>
</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.
- 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.
Boolean Expressions

\[(5 || !3) \&\& 6\]

\[= (true \ OR \ (NOT \ true)) \ AND \ true\]
\[= (true \ OR \ false) \ AND \ true\]
\[= (true) \ AND \ true\]
\[= true\]
\[= 1\]

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.

Bitwise Expressions

\[(5 \mid \sim 3) \& 6\]

\[= (00..0101 \ OR \ \sim 00..0011) \ AND \ 00..0110\]
\[= (00..0101 \ OR \ 11..1100) \ AND \ 00..0110\]
\[= (11..1101) \ AND \ 00..0110\]
\[= 00..0100\]
\[= 4\]

Interpreting the Bitwise-AND

<table>
<thead>
<tr>
<th>m</th>
<th>p</th>
<th>m AND p</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0: 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>
<td>\text{10}</td>
<td>( p ): If bit ( m ) of the mask is 1, bit ( p ) is passed through to the result unchanged.</td>
</tr>
</tbody>
</table>
Interpreting the Bitwise-OR

<table>
<thead>
<tr>
<th>m</th>
<th>p</th>
<th>m OR p</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>If bit ( m ) of the mask is 0, bit ( p ) is passed through to the result <strong>unchanged</strong>.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>If bit ( m ) of the mask is 1, bit ( p ) is <strong>set to 1</strong> in the result.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>If bit ( m ) of the mask is 1, bit ( p ) is <strong>set to 1</strong> in the result.</td>
</tr>
</tbody>
</table>

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:

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

Testing Bits

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

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

• 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:

  ```c
  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.

```c
#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  *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_ctrl = (new_flags & (1 << 8)) != 0 ;
  return TRUE ;
}

void Display_Kybd_Flags(SHIFTS  *kybd) {
  printf( "Right Shift: %s \n", kybd->right_shift ? "on" : "off" ) ;
  printf( "Left Shift:  %s \n", kybd->left_shift ? "on" : "off" ) ;
  printf( "Control:    %s \n", kybd->ctrl ? "on" : "off" ) ;
  printf( "Alt:        %s \n", kybd->alt ? "on" : "off" ) ;
  printf( "Left Ctrl:  %s \n", kybd->left_ctrl ? "on" : "off" ) ;
  printf( "Left Alt:   %s \n", kybd->left_alt ? "on" : "off" ) ;
}

void main() {
  SHIFTS  kybd ;
  do {
    if (Kybd_Flags_Changed(&kybd))
      Display_Kybd_Flags(&kybd) ;
  } while (!kybd.left_shift || !kybd.right_shift) ;
}
```

Setting Bits

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

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

• This would usually be written more succinctly as:

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

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

\[
\begin{array}{c}
(1 \ll 7) \Rightarrow 10000000 \\
\neg(1 \ll 7) \Rightarrow 01111111
\end{array}
\]

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

Inverting Bits

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

```
bits ^= (1L << 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.
Importance of Matching Operand Sizes When ints Are 16-bits

0xFFFFFFFFL & ~(1L << 14) ⇒ FFFFBFFF (ok)
0xFFFFFFFFL & ~(1L << 15) ⇒ 00007FFF (error)

Extracting Bits

Bits 15 - 11 Bits 10 - 5 Bits 4 - 0
<table>
<thead>
<tr>
<th>Hours</th>
<th>Minutes</th>
<th>Seconds + 2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>time &gt;&gt; 5</th>
<th>Bits 15 - 11</th>
<th>Bits 10 - 6</th>
<th>Bits 5 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>????</td>
<td>Hours</td>
<td>Minutes</td>
<td></td>
</tr>
</tbody>
</table>

(time >> 5) & 0x3F

<table>
<thead>
<tr>
<th>Bits 15 - 11</th>
<th>Bits 10 - 6</th>
<th>Bits 5 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>000000</td>
<td>Minutes</td>
</tr>
</tbody>
</table>

minutes = (time >> 5) & 0x3F

<table>
<thead>
<tr>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Inserting Bits

oldtime

Bits 15 - 11 Bits 10 - 5 Bits 4 - 0
<table>
<thead>
<tr>
<th>Hours</th>
<th>Old Minutes</th>
<th>Seconds + 2</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Bits 15 - 11</th>
<th>Bits 10 - 5</th>
<th>Bits 4 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hours</td>
<td>000000</td>
<td>Seconds + 2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bits 15 - 11</th>
<th>Bits 10 - 5</th>
<th>Bits 4 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hours</td>
<td>New Minutes</td>
<td>Seconds + 2</td>
</tr>
</tbody>
</table>

Automatic Insertion and 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!
Reserved/Unused Bit Positions

typedef struct {
    unsigned CF :1, /* Bit 0: Carry Flag */
    :1, /* Bit 1: (unused) */
    PF :1, /* Bit 2: Parity Flag */
    :1, /* Bit 3: (unused) */
    AF :1, /* Bit 4: Auxiliary Carry Flag */
    :1, /* Bit 5: (unused) */
    ZF :1, /* Bit 6: Zero Flag */
    SF :1, /* Bit 7: Sign Flag */
    TF :1, /* Bit 8: Trap Flag */
    IF :1, /* Bit 9: Interrupt Enable Flag */
    DF :1, /* Bit 10: Direction Flag */
    OF :1, /* Bit 11: Overflow Flag */
    :4 ; /* Bits 12-15: (unused) */
} PSW ;

Testing Keyboard Flags Using Structure Bit Fields.

typedef struct KYBD_BITS {
    unsigned
    right_shift :1,
    left_shift :1,
    ctrl :1,
    alt :1,
    :4,
    left Ctrl :1,
    left Alt :2 ;
} KYBD_BITS ;

BOOL Kybd_Flags_Changed(SHIFTS *kybd) {
    ...bits = (KYBD_BITS *) &new_flags ;
    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 Ctrl = bits->left Ctrl != 0 ;
    return TRUE ;
}

Variant Access with Pointers, Casts, & Subscripting

- Given an address, we can 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, we can read or write various parts of an object named operand using:

  ((BYTE8 *) &operand)[k]

Variant Access with Pointers, Casts, & Subscripting

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

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

```c
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 ;
}
```

I/O Addresses Used by One of the IBM/PC Serial Ports.

<table>
<thead>
<tr>
<th>Addr.</th>
<th>DLAB</th>
<th>Restriction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>02F8</td>
<td>0</td>
<td>Write-Only</td>
<td>Transmitter holding register (holds character to be sent)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Read-Only</td>
<td>Receiver buffer register (contains the received character)</td>
</tr>
<tr>
<td>02F9</td>
<td>1</td>
<td></td>
<td>Least significant byte of baud rate divisor latch</td>
</tr>
<tr>
<td>02F9</td>
<td>0</td>
<td></td>
<td>Most significant byte of baud rate divisor latch</td>
</tr>
<tr>
<td>02FA</td>
<td>n/a</td>
<td>Read-Only</td>
<td>Interrupt ID register</td>
</tr>
<tr>
<td>02FB</td>
<td>n/a</td>
<td></td>
<td>Line control register (Bit 7 is the DLAB bit)</td>
</tr>
<tr>
<td>02FC</td>
<td>n/a</td>
<td></td>
<td>Modem control register</td>
</tr>
<tr>
<td>02FD</td>
<td>n/a</td>
<td>Read-Only</td>
<td>Line status register</td>
</tr>
<tr>
<td>02FE</td>
<td>n/a</td>
<td>Read-Only</td>
<td>Modem status register</td>
</tr>
<tr>
<td>02FF</td>
<td>n/a</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Manipulating Bits in I/O Ports

- I/O ports must be accessed using functions.
- It is often common for I/O ports to be either read-only or write-only.
- It is often common for several I/O ports to be assigned to a single I/O address.
I/O Addresses Used by One of the IBM/PC Serial Ports.

![Diagram of I/O Addresses](image)

Read-Only

Write-Only

THR  RBR  BRD  IER  IIR  LCR  MCR  LSR  MSR

2F8  2FA  2FB  2FC  2FD  2FE

Read-Only

DLAB=1

Modifying Bits in Write-Only I/O Ports

- The read-modify-write techniques presented earlier for manipulating bits can’t be used.

- Solution: Keep in memory a copy of the last bit-pattern written to the port. Apply the read-modify-write techniques to the memory copy, and then write that value to the port.

Modifying Bits in Write-Only I/O Ports

```c
typedef struct LPT_CTRL {
    unsigned
data_strobe :1, /* structure bit fields simplify */
auto_linefeed :1, /* access to the printer’s packed */
initialize :1, /* control port. Note, however, */
select :1, /* we must later cast it as a byte */
enable_irq :1, /* because the outportb function */
/* unused */
/* requires a byte parameter. */
} LPT_CTRL;

static LPT_CTRL lpt1 = {0, 0, 0, 0}; /* static! (and initialized to zeroes) */
...
lpt1.data_strobe = 1; /* change bit in memory copy; others remain unchanged */
...
Byte2Port(0x3BE, (BYTE8) lpt1); /* then copy data to the printer port */
```

Sequential Access to I/O Ports

- Sometimes multiple read-only ports assigned to one address are distinguished by the order in which the data is read.

- Multiple write-only ports assigned to one address can also be distinguished by order, or by the value of specific bits in part of the data that is written.
### I/O Ports of the 8259 Interrupt Controller in the IBM/PC.

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Data</th>
<th>Restriction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0020</td>
<td>xxx10xxx</td>
<td>Read-Only</td>
<td>Interrupt request or in-service register (ICW1)</td>
</tr>
<tr>
<td></td>
<td>xxx00xxx</td>
<td>Write-Only</td>
<td>Initialization command word 1 (ICW1)</td>
</tr>
<tr>
<td></td>
<td>xxx01xxx</td>
<td>Write-Only</td>
<td>Operation Control Word 2</td>
</tr>
<tr>
<td>0021</td>
<td></td>
<td>Write-Only</td>
<td>Operation Control Word 1 (Interrupt mask register)</td>
</tr>
<tr>
<td>0022-003F</td>
<td></td>
<td>Write-Only</td>
<td>ICW2, ICW3, ICW4 (in order, only just after writing ICW1)</td>
</tr>
</tbody>
</table>

**Note:**
- Only after writing ICW1

---

**Diagram:**
- ISR
- ICW1
- OCW2
- OCW3
- OCW1
- ICW2
- ICW3
- ICW4

- XXX10000
- XXX00000
- XXX01100

1st 2nd 3rd

Only after writing ICW1