

CHAPTER 4

GETTING THE MOST OUT OF C

Copyright © 2000, Daniel W.
Lewis. All Rights Reserved.

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

| <i>Data Type</i> | <i>Size</i> | <i>Range</i> |
|------------------|-------------|---------------|
| unsigned | char | 8 bits |
| | short int | 16 bits |
| | int | 16 or 32 bits |
| | long int | 32 bits |
| signed | char | 8 bits |
| | short int | 16 bits |
| | int | 16 or 32 bits |
| | long int | 32 bits |

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

Minimum and Maximum Integer Values defined in LIMITS.H

| Data Type | Minimum | Maximum |
|------------------------|-----------|------------|
| signed char | SCHAR_MIN | SCHAR_MAX |
| signed short int | SHRT_MIN | SHRT_MAX |
| signed long int | LONG_MIN | LONG_MAX |
| signed long long int | LLONG_MIN | LLONG_MAX |
| unsigned char | 0 | UCHAR_MAX |
| unsigned short int | 0 | USHRT_MAX |
| unsigned long int | 0 | ULONG_MAX |
| unsigned long long int | 0 | ULLONG_MAX |

Contents of ANSI File (limits.h)

Notice anything odd?

```
#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
```

1999 ISO (C99) New Data Types

#include <stdint.h>

| | Signed | Unsigned |
|---------|---------|----------|
| 8-bits | int8_t | uint8_t |
| 16-bits | int16_t | uint16_t |
| 32-bits | int32_t | uint32_t |
| 64-bits | int64_t | uint64_t |

Note: If your compiler doesn't already provide these data types, you can create your own using `typedefs`.

If Your Compiler Doesn't Support C99

```
#ifndef __STDC_VERSION__  
  
typedef signed     char      int8_t ;  
typedef signed     short int  int16_t ;  
typedef signed     long int   int32_t ;  
typedef signed     long long int int64_t ;  
  
typedef unsigned   char      uint8_t ;  
typedef unsigned   short int  uint16_t ;  
typedef unsigned   long int   uint32_t ;  
typedef unsigned   long long int uint64_t ;  
  
#endif
```

Using #defines to simplify changes

```
#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!

```
#define SUM(a, b) a  
+ b
```

```
y = 5 * SUM(t1, t2) ;
```

Becomes...

```
y = 5 * t1 + t2 ;
```

```
#define TIMES(a, b) (a *  
b)
```

```
y = TIMES(t1-2, t2+3) ;
```

Becomes...

```
y = (t1 - 2 * t2 + 3) ;
```

C99 Boolean Data Type

```
#include  
<stdbool.h>
```

```
#define bool _Bool      // New C99 data type  
#define false ((Bool) 0)  
#define true  ((Bool) 1)
```

Other Alternative Booleans

```
#ifndef __STDC_VERSION__  
#define bool unsigned char      // using macros  
#define false 0  
#define true 1  
#endif
```

```
#ifndef __STDC_VERSION__  
typedef unsigned char bool ;      // using unsigned char's  
const unsigned char false = 0 ;  
const unsigned char true = 1 ;  
#endif
```

```
#ifndef __STDC_VERSION__  
typedef enum {false = 0, true = 1} bool ; // using an  
enumerated type  
#endif
```

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

Boolean and Binary Operators

| Operation | Boolean Operator | Bitwise Operator |
|-----------|-------------------------|--------------------|
| AND | <code>&&</code> | <code>&</code> |
| OR | <code> </code> | <code> </code> |
| XOR | <i>unsupported</i> | <code>^</code> |
| NOT | <code>!</code> | <code>~</code> |

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

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 | ~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

Packed Operands



MS/DOS packed representation of time.

| | | | | | | | |
|----------------|----------------|----------------|---------------|------------------|----------------|----------------|-----------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Carrier Detect | Ring Indicator | Data Set Ready | Clear To Send | Δ Carrier Detect | Ring Indicator | Δ Data Set Rdy | Δ Clear To Send |

UART modem status port

| | | | | | | | |
|----------|----------|----------|------------|----------------|--------------------|---------------|-------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | Reserved | Reserved | Enable IRQ | Select Printer | Initialize Printer | Auto LineFeed | Data Strobe |

IBM-PC printer control port

Interpreting the Bitwise-AND

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

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 */
```

$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ & 01000000 \rightarrow $0b_6000000$

Testing Bits

- 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

- 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)) != 0) /* 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.

Clearing Bits

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

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

$(1 \ll 7)$ 0000000

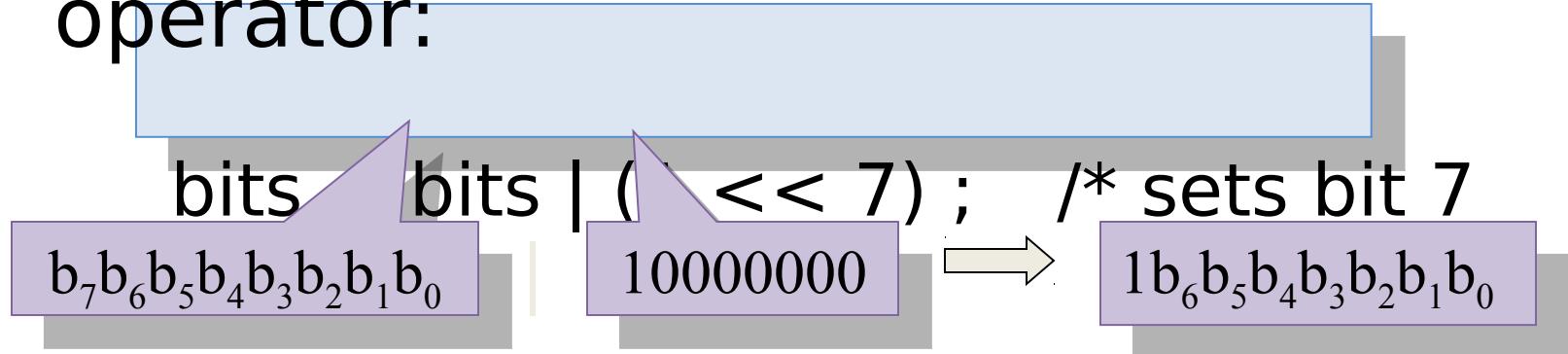
$\sim(1 \ll 7)$ 1111111

Interpreting the Bitwise-OR

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

Setting Bits

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



- This would usually be written more succinctly as:

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

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 | 1 | | |
| 1 | 0 | 1 | $\sim p$ | If bit m of the mask is 1, bit p is passed through to the result inverted . |
| | 1 | 0 | | |

Inverting Bits

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

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

Extracting Bits

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

Inserting Bits

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 |

Automatic Insertion and Extraction Using Structure Bit Fields

```
struct {  
    uint16_t 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!*

Packed Structures

```
#pragma pack(1) // turn off structure padding
typedef struct // This structure occupies 6 bytes
{
    uint8_t  a ; // 1 byte (would otherwise pad to 4 bytes)
    uint32_t b ; // 4 bytes
    uint8_t  c ; // 1 byte (would otherwise pad to 4 bytes)
} ;
#pragma pack() // turn on structure padding
```

Pros/Cons of Structure Bit-Fields

Pros:

- Cleaner syntax
- Fewer lines of C code
- Easier to read
- Compiles into bitwise operations

Cons:

- No standard regarding left-to-right vs. right-to-left Bit-Field assignment

What does compiler need to know?

Consider:

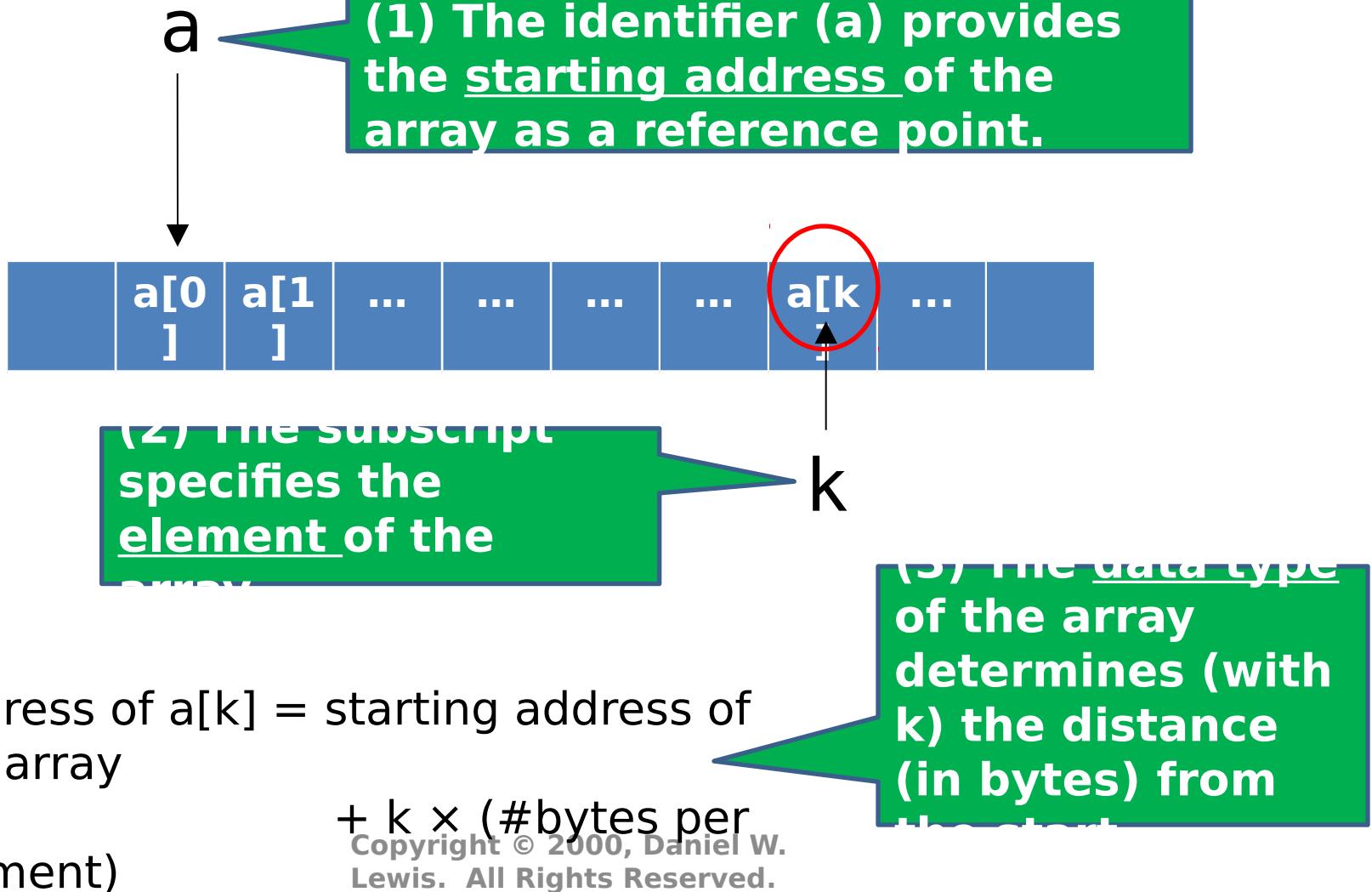
```
x = 0 ;
```

Compiler must know:

- (1) Address of memory location
(Provided by the identifier)

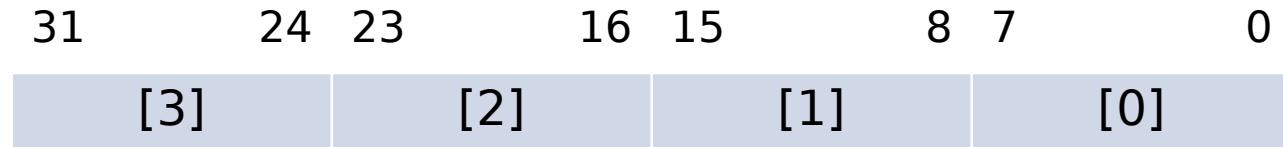
- (2) Size (in bytes) of memory location
(Provided by the data type)

What does it take to subscript?



Variant Access with Pointers, Casts, & Subscripting

```
uint32_t operand ; // a 32-bit scalar
```



(`(uint8_t` *) `&operand`) [`k`]

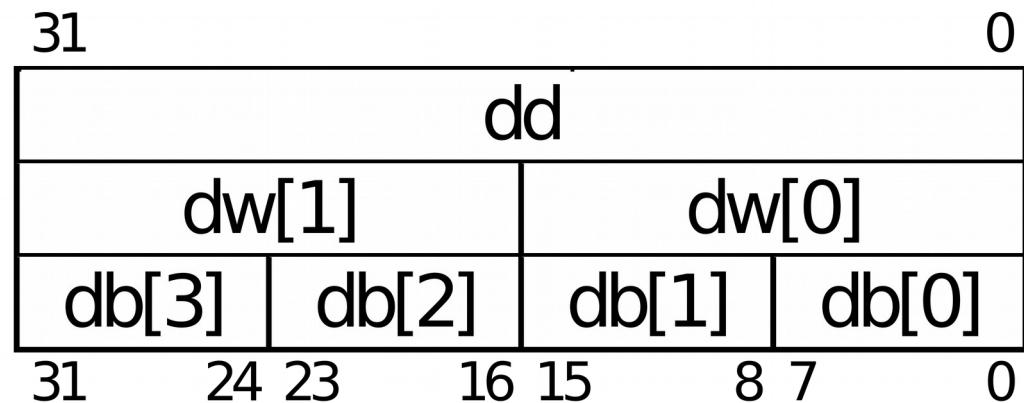
(3) The data type

(1) The starting address

(2) The selected element

Variant Access with Unions

```
union {
    uint32_t    dd ;
    uint16_t    dw[2] ;
    uint8_t     db[4] ;
};
```



Variant Access vs. Unions

Both:

- Bit-fields must lie on byte boundaries

Variant Access:

- Syntax is somewhat complex.
- No union declaration required.

Unions:

- Cleaner syntax
- Easier to read
- Compiles into variant access code

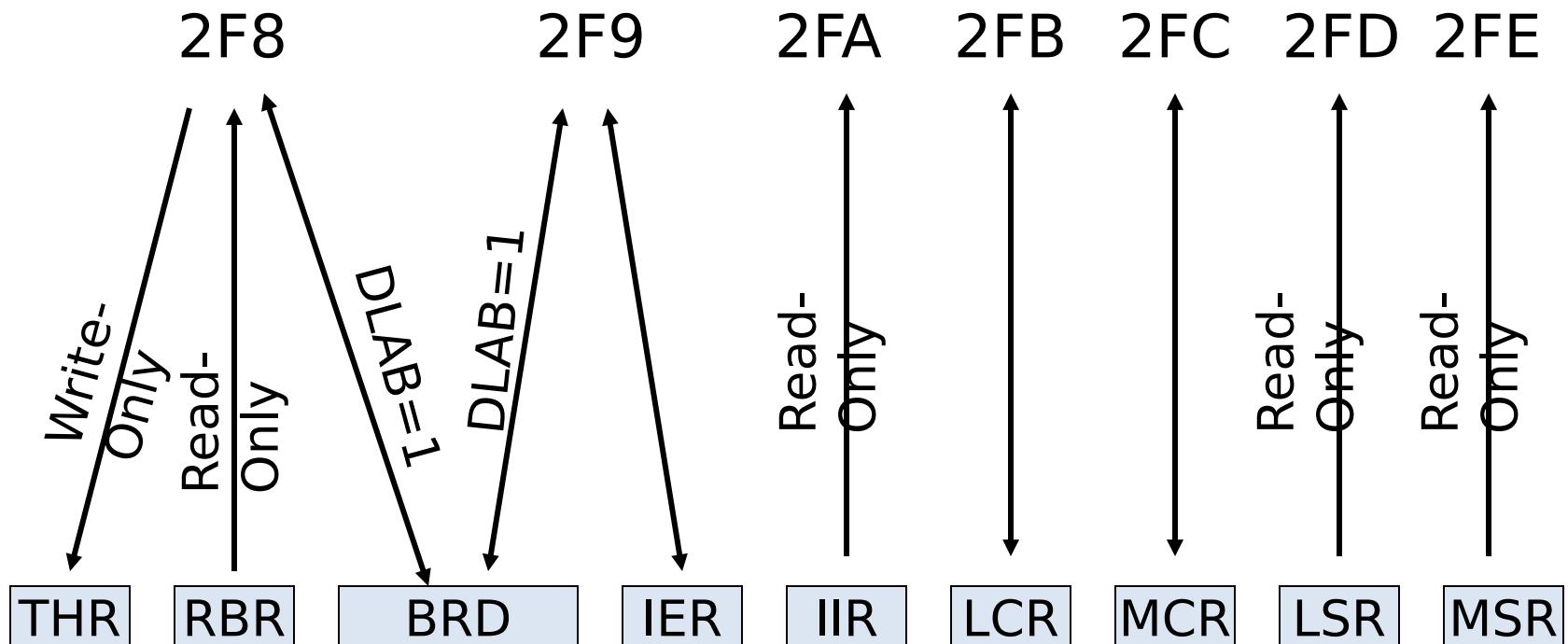
Manipulating Bits in I/O Ports

- 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.

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

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



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.

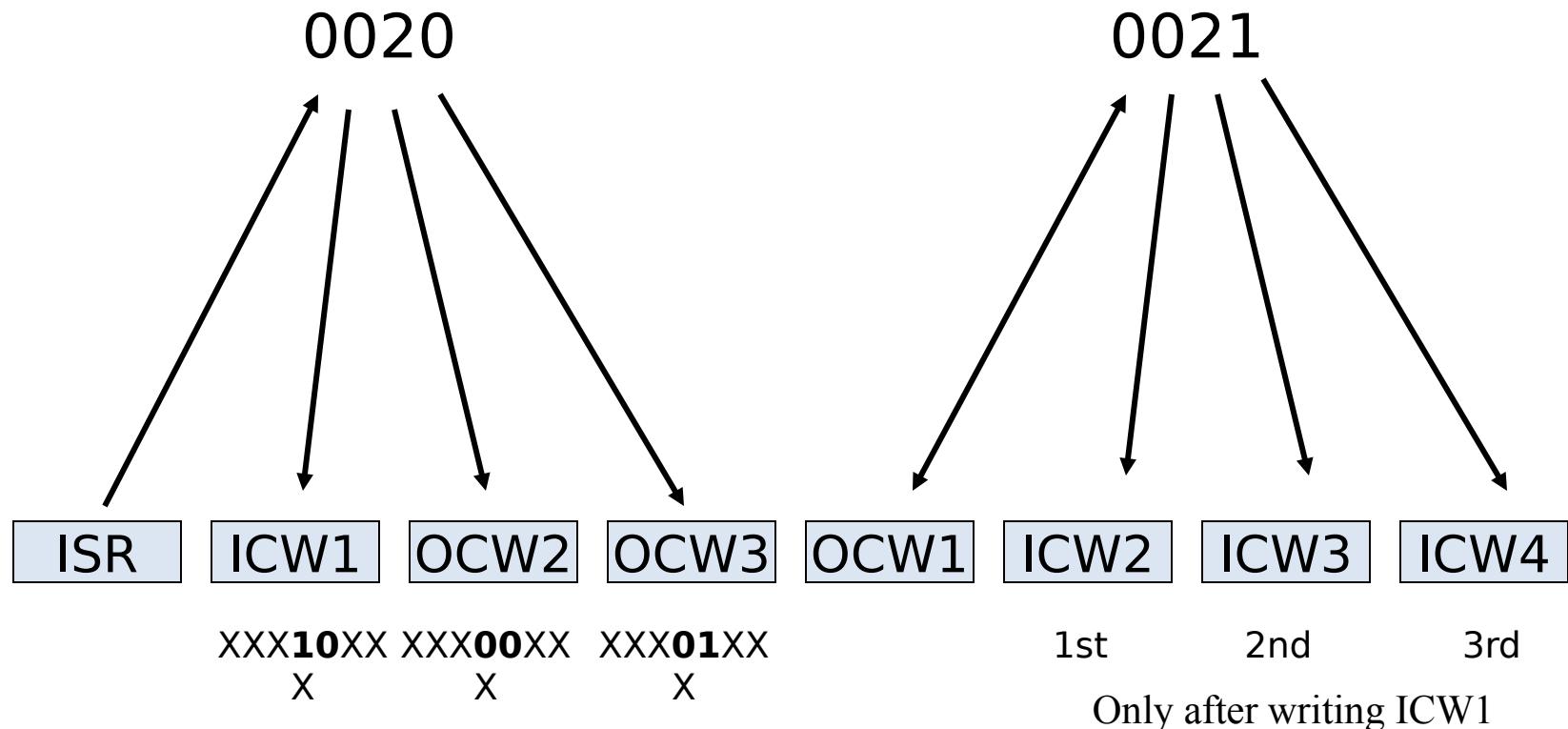
Sequential Access to I/O Ports

- 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 Programmable Interrupt Controller.

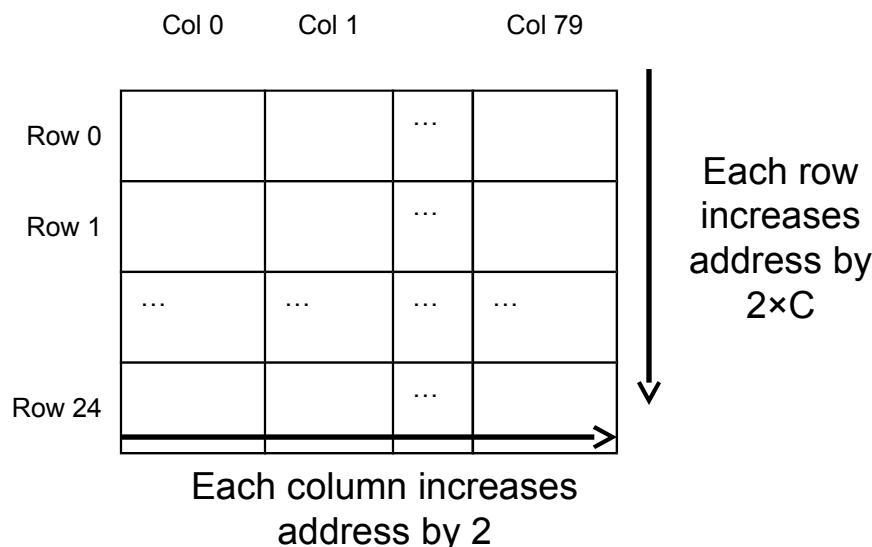
| Addr. | Data | Restriction | Description |
|-----------|-------------------|-------------|--|
| 0020 | | Read-Only | Interrupt request or in-service register |
| | xxx 1 xxxx | Write-Only | Initialization command word 1 (ICW1) |
| | xxx 00 xxx | Write-Only | Operation Control Word 2 |
| | xxx 01 xxx | Write-Only | Operation Control Word 3 |
| 0021 | | | Operation Control Word 1 (Interrupt mask register) |
| | | Write-Only | ICW2, ICW3, ICW4 (in order, only just after writing ICW1) |
| 0022-003F | | | Reserved |

I/O Ports of the 8259 Interrupt Controller in the IBM/PC.



Accessing a Display Buffer

Display buffer memory starts at address $B8000_{16}$. Each cell contains 2 bytes: ASCII code and an attribute byte



```
uint8_t *p = (uint8_t *) 0xB8000  
+ 2 * (80 * row + col);
```

```
*(p + 0) = ascii_code;  
*(p + 1) = attribute_bits;
```

-- or --

```
p[0] = ascii_code;  
p[1] = attribute_bits;
```

-- or --

```
typedef uint_8 CELL[2];  
typedef CELL ROW[80];  
ROW *buffer = (ROW *) 0xB8000;
```

```
buffer[row][col][0] = ascii_code;  
buffer[row][col][1] = attribute_bits;
```