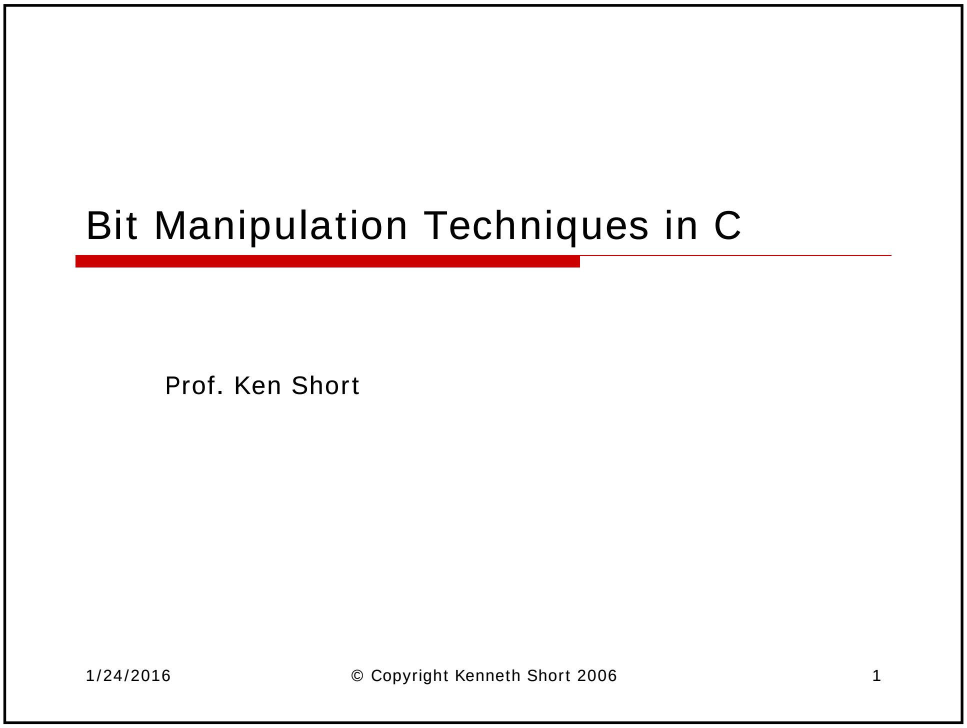
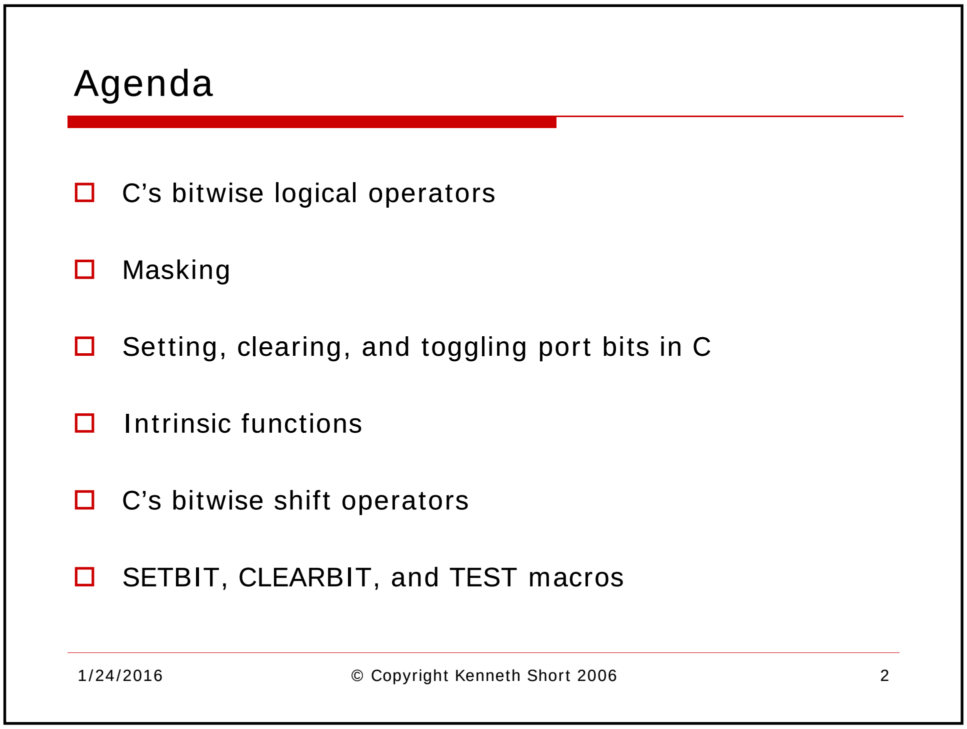


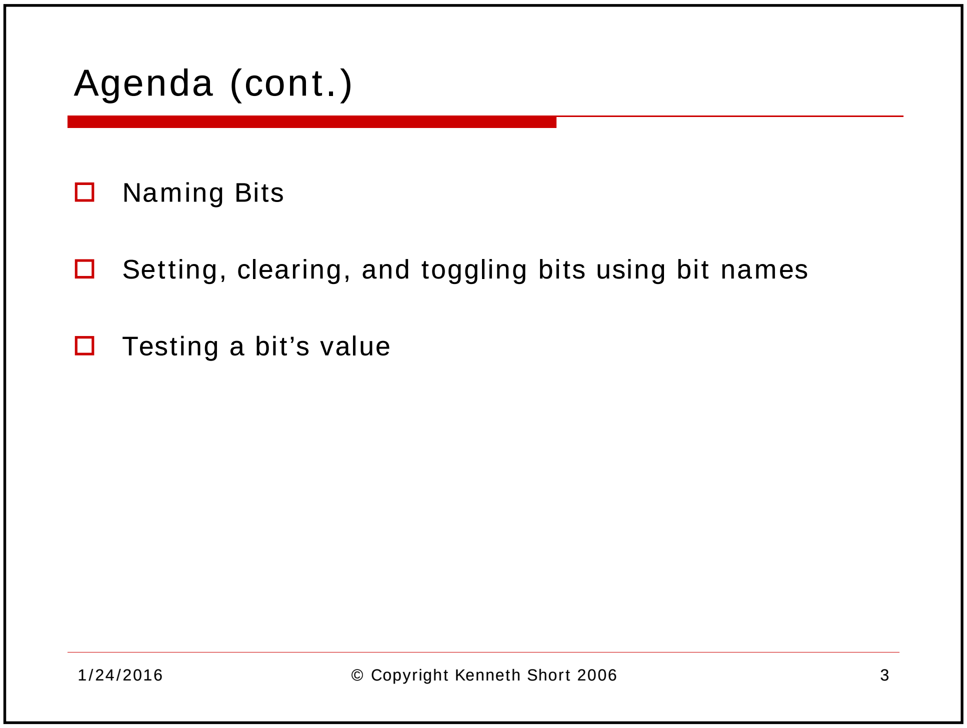
Bit Manipulation Techniques in C

Prof. Ken Short



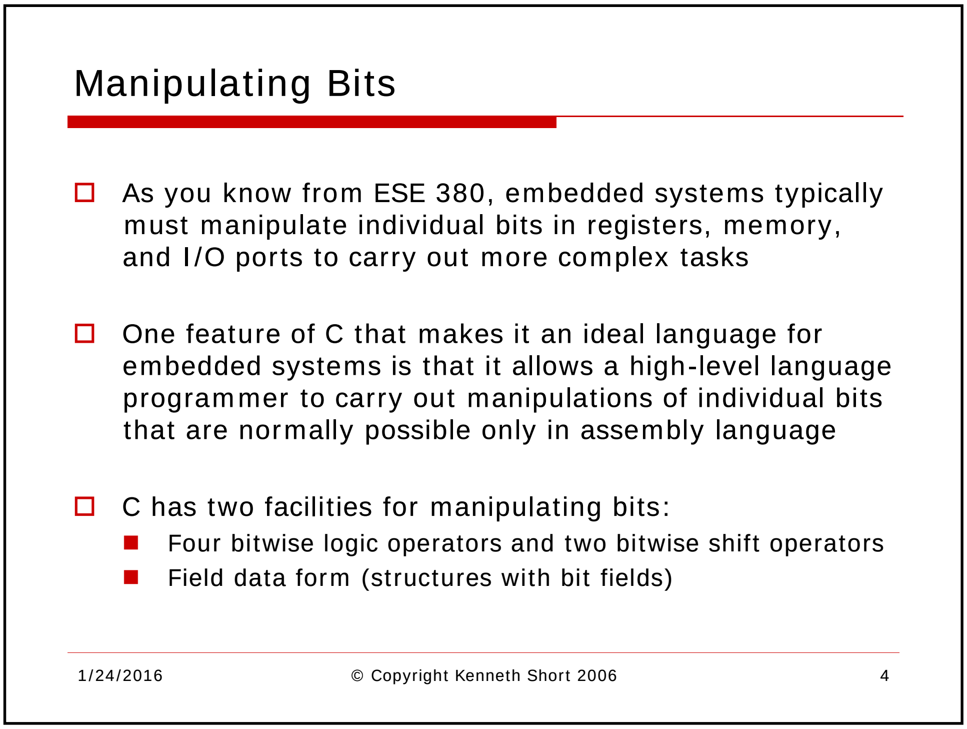
Agenda

- □ C’s bitwise logical operators
- □ Masking
- □ Setting, clearing, and toggling port bits in C
- □ Intrinsic functions
- □ C’s bitwise shift operators
- □ SETBIT, CLEARBIT, and TEST macros



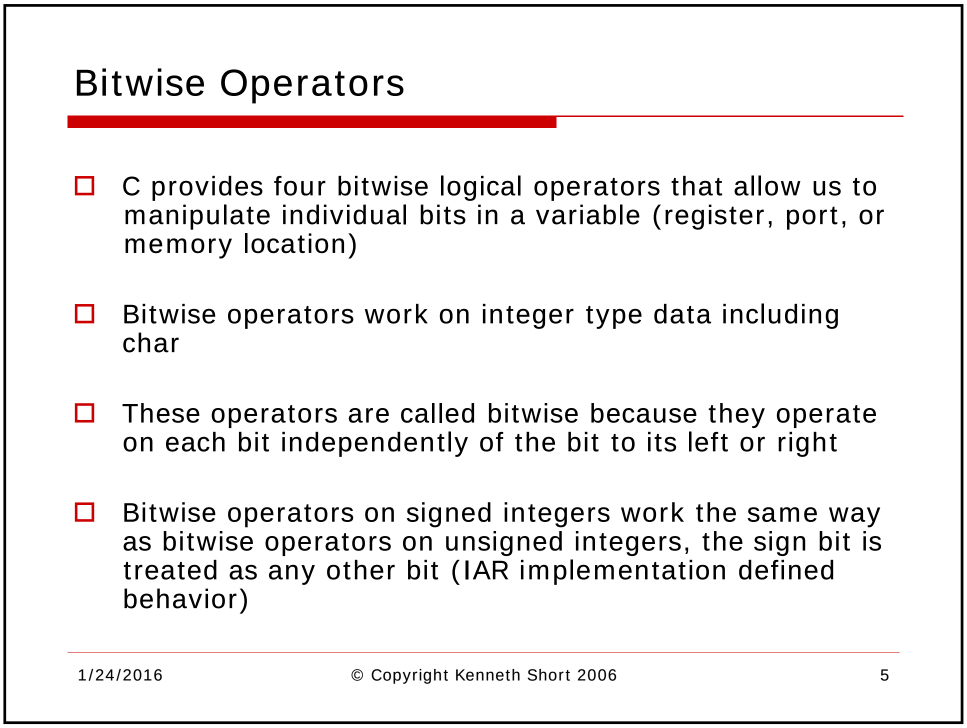
Agenda (cont.)

- Naming Bits
- Setting, clearing, and toggling bits using bit names
- Testing a bit’s value



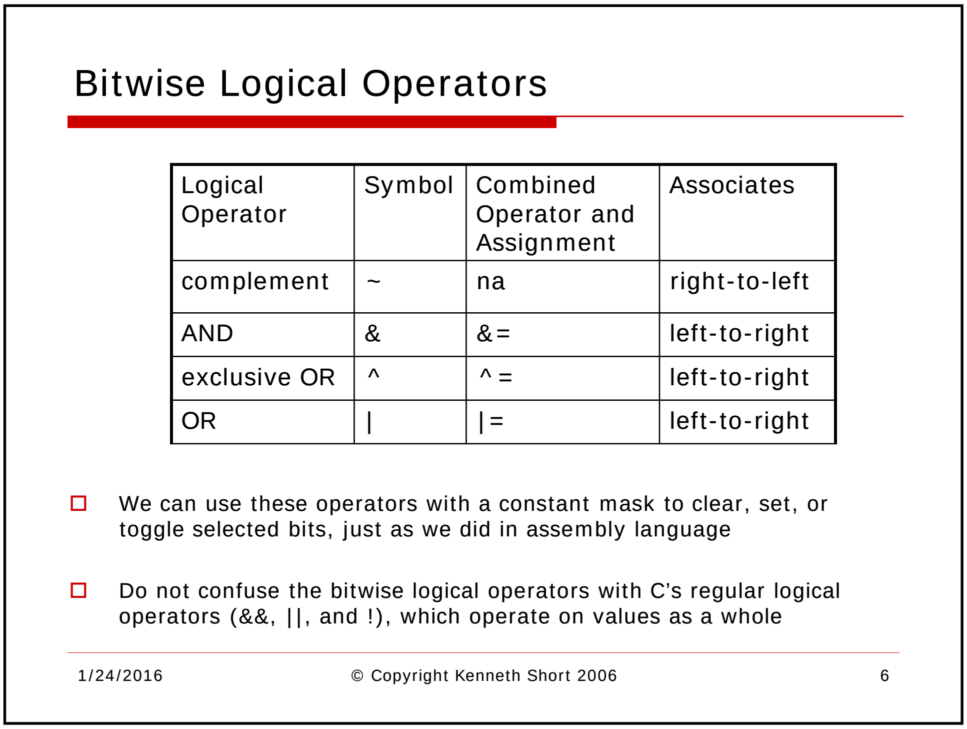
Manipulating Bits

- As you know from ESE 380, embedded systems typically must manipulate individual bits in registers, memory, and I/O ports to carry out more complex tasks
- One feature of C that makes it an ideal language for embedded systems is that it allows a high-level language programmer to carry out manipulations of individual bits that are normally possible only in assembly language
- C has two facilities for manipulating bits:
  - Four bitwise logic operators and two bitwise shift operators
  - Field data form (structures with bit fields)



Bitwise Operators

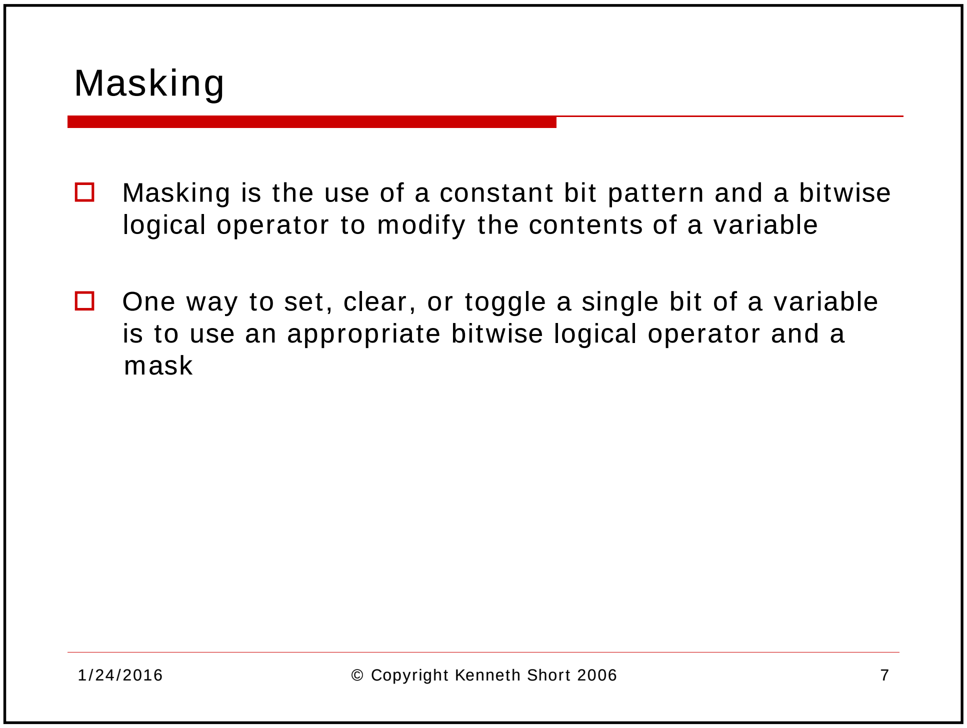
- C provides four bitwise logical operators that allow us to manipulate individual bits in a variable (register, port, or memory location)
- Bitwise operators work on integer type data including char
- These operators are called bitwise because they operate on each bit independently of the bit to its left or right
- Bitwise operators on signed integers work the same way as bitwise operators on unsigned integers, the sign bit is treated as any other bit (IAR implementation defined behavior)



Bitwise Logical Operators

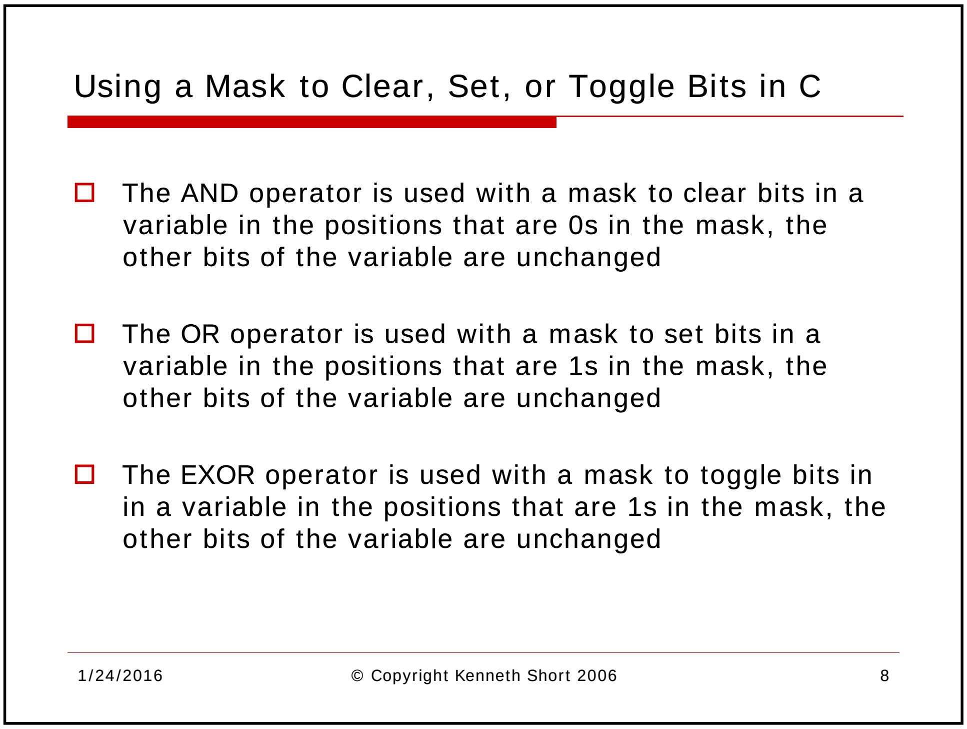
Logical Operator	Symbol	Combined Operator and Assignment	Associates
complement	~	na	right-to-left
AND	&	&=	left-to-right
exclusive OR	^	^=	left-to-right
OR	|	|=	left-to-right

- We can use these operators with a constant mask to clear, set, or toggle selected bits, just as we did in assembly language
- Do not confuse the bitwise logical operators with C's regular logical operators (&&, ||, and !), which operate on values as a whole



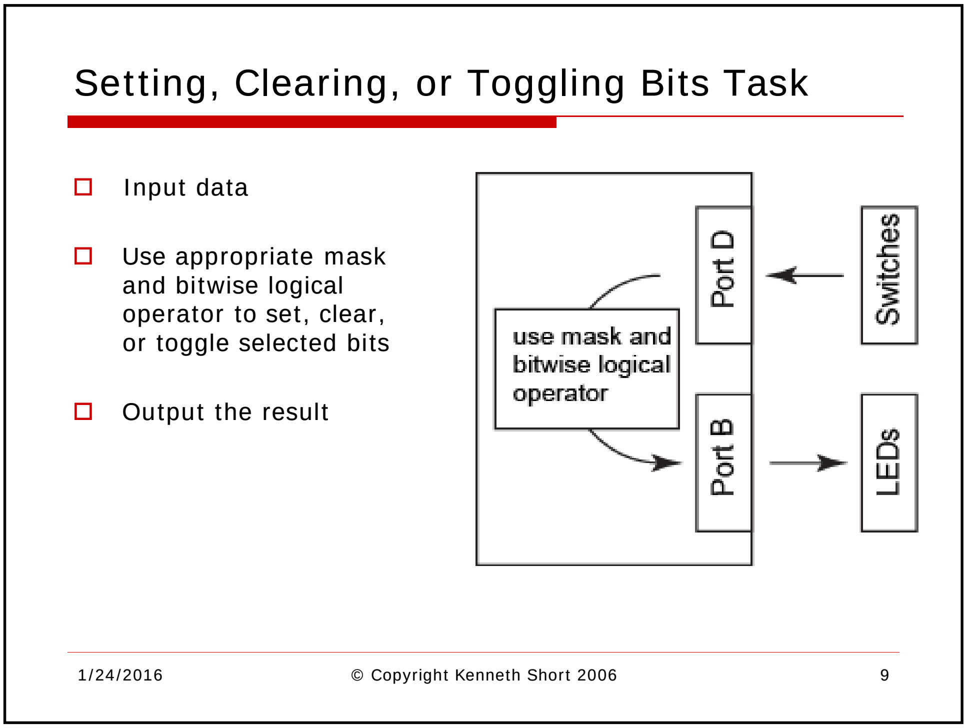
Masking

- Masking is the use of a constant bit pattern and a bitwise logical operator to modify the contents of a variable
- One way to set, clear, or toggle a single bit of a variable is to use an appropriate bitwise logical operator and a mask



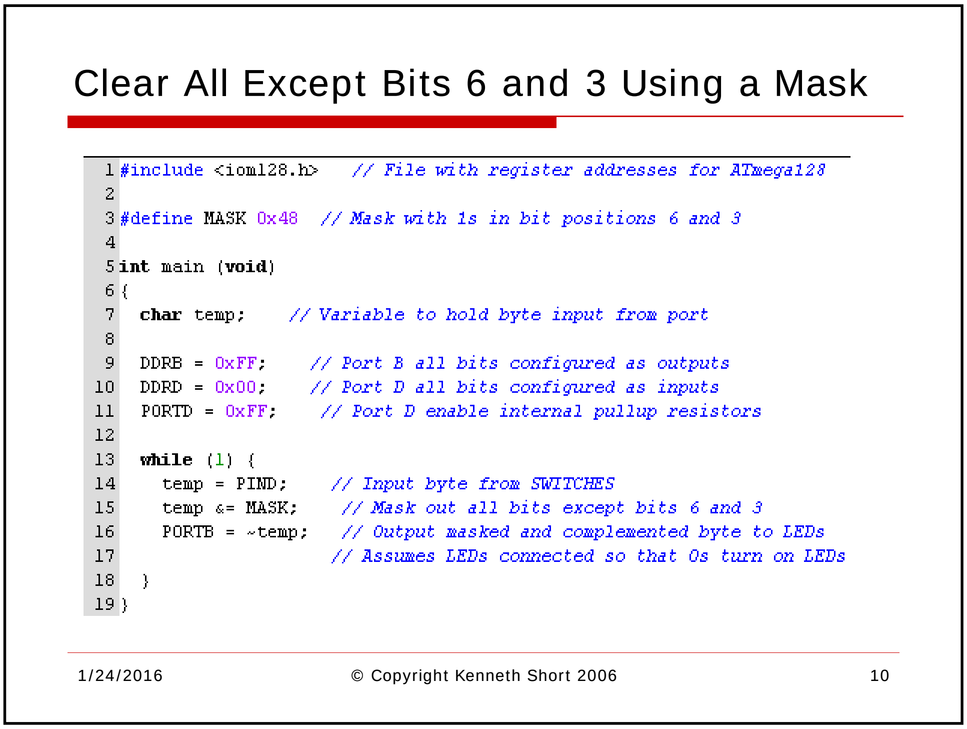
Using a Mask to Clear, Set, or Toggle Bits in C

- The AND operator is used with a mask to clear bits in a variable in the positions that are 0s in the mask, the other bits of the variable are unchanged
- The OR operator is used with a mask to set bits in a variable in the positions that are 1s in the mask, the other bits of the variable are unchanged
- The EXOR operator is used with a mask to toggle bits in in a variable in the positions that are 1s in the mask, the other bits of the variable are unchanged



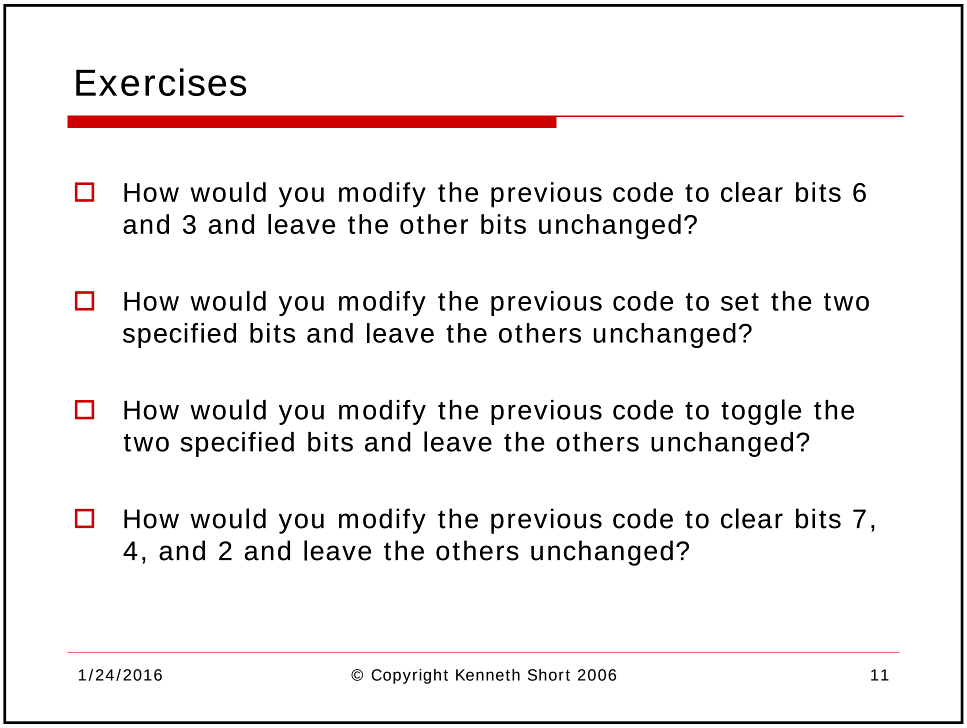
Setting, Clearing, or Toggling Bits Task

- Input data
- Use appropriate mask and bitwise logical operator to set, clear, or toggle selected bits
- Output the result



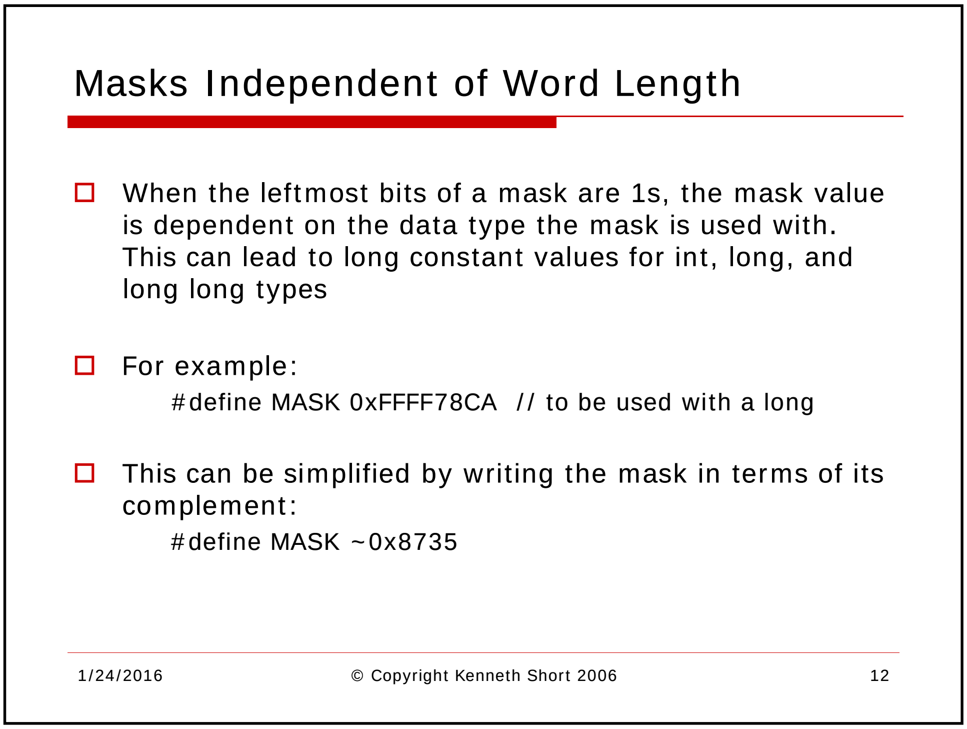
Clear All Except Bits 6 and 3 Using a Mask

```
#include <iom128.h>   // File with register addresses for ATmega128

#define MASK 0x48   // Mask with 1s in bit positions 6 and 3

int main (void)
{
  char temp;    // Variable to hold byte input from port

  DDRB = 0xFF;    // Port B all bits configured as outputs
  DDRD = 0x00;    // Port D all bits configured as inputs
  PORTD = 0xFF;    // Port D enable internal pullup resistors

  while (1) {
    temp = PIND;    // Input byte from SWITCHES
    temp &= MASK;    // Mask out all bits except bits 6 and 3
    PORTB = ~temp;    // Output masked and complemented byte to LEDs
                    // Assumes LEDs connected so that 0s turn on LEDs
  }
}
```



Exercises

- How would you modify the previous code to clear bits 6 and 3 and leave the other bits unchanged?
- How would you modify the previous code to set the two specified bits and leave the others unchanged?
- How would you modify the previous code to toggle the two specified bits and leave the others unchanged?
- How would you modify the previous code to clear bits 7, 4, and 2 and leave the others unchanged?



Masks Independent of Word Length

- When the leftmost bits of a mask are 1s, the mask value is dependent on the data type the mask is used with. This can lead to long constant values for int, long, and long long types
- For example:

```
#define MASK 0xFFFF78CA  // to be used with a long
```

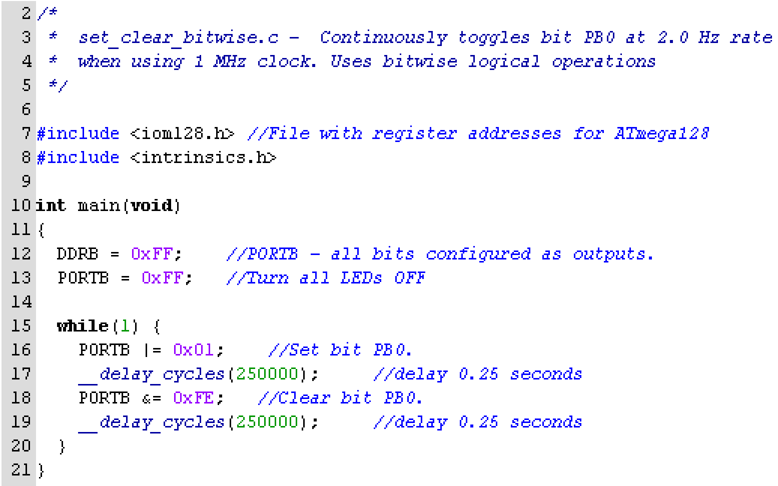
- This can be simplified by writing the mask in terms of its complement:

```
#define MASK ~0x8735
```

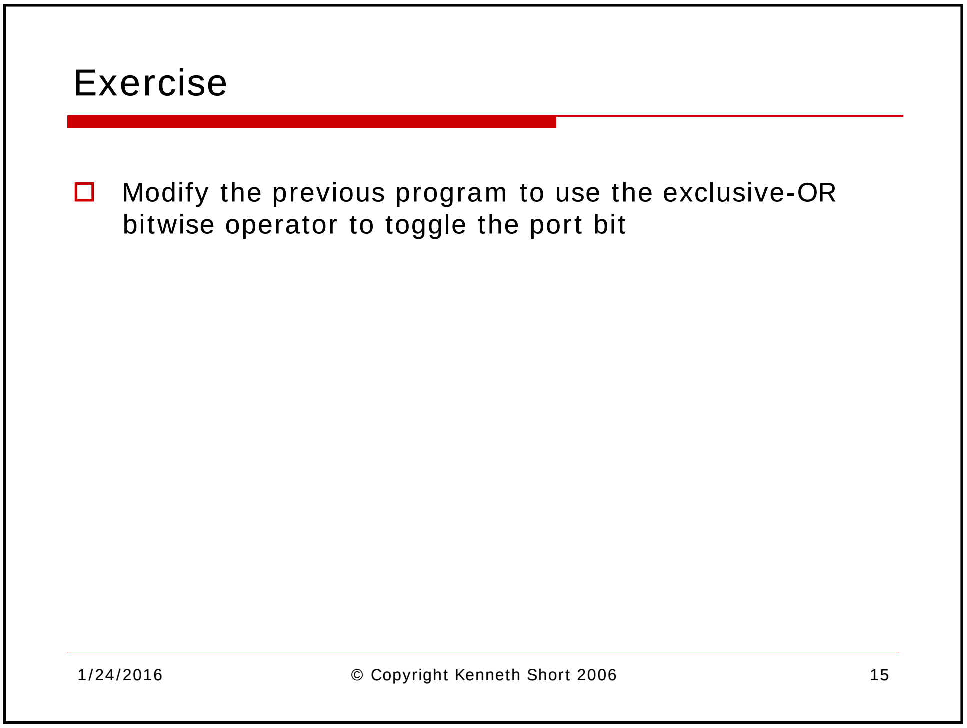
Other Ways to Set, Clear, or Toggle a Port Bit

- There are several ways to set, clear, or toggle a port bit, these include using:
  - A bitwise logical operator and a constant mask (previously discussed)
  - A bitwise logical operator and a mask created using a shift operation
  - Predefined compiler specific macros (SETBIT and CLEARBIT)
  - Bit names defined in the header file iom128.h

Toggling a Port Bit using Bitwise Logical Operators

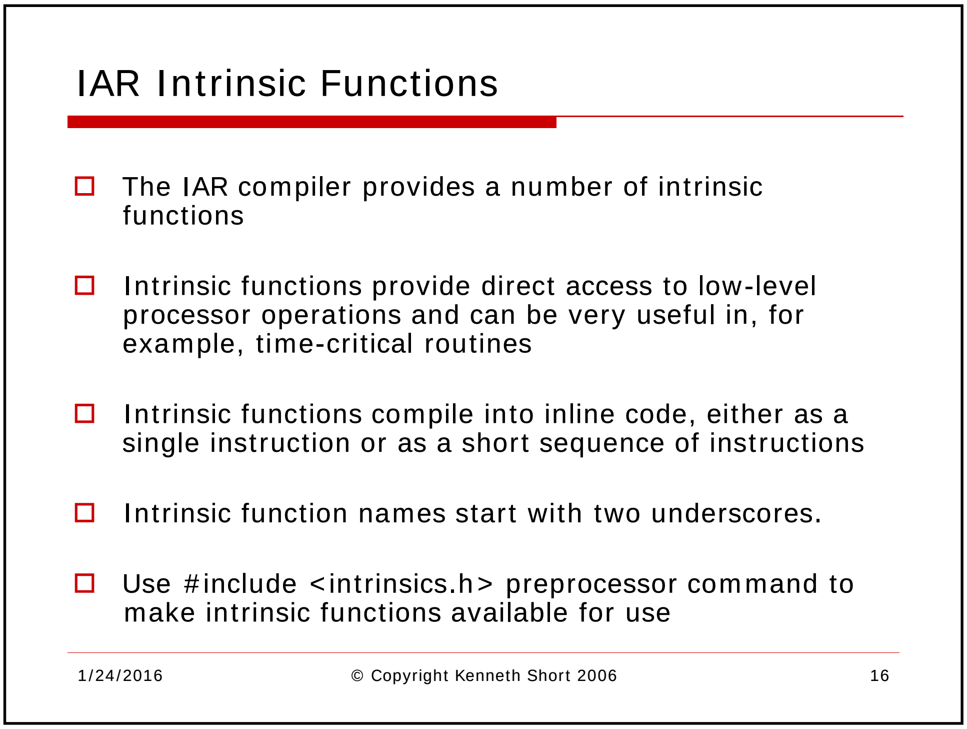


```
/*
 *  set_clear_bitwise.c -  Continuously toggles bit PB0 at 2.0 Hz rate
 *  when using 1 MHz clock. Uses bitwise logical operations
 */

#include <iom128.h> //File with register addresses for ATmega128
#include <intrinsics.h>

int main(void)
{
  DDRB = 0xFF;     //PORTB - all bits configured as outputs.
  PORTB = 0xFF;    //Turn all LEDs OFF

  while(1) {
    PORTB |= 0x01;     //Set bit PB0.
    __delay_cycles(250000);      //delay 0.25 seconds
    PORTB &= 0xFE;    //Clear bit PB0.
    __delay_cycles(250000);      //delay 0.25 seconds
  }
}
```



Exercise

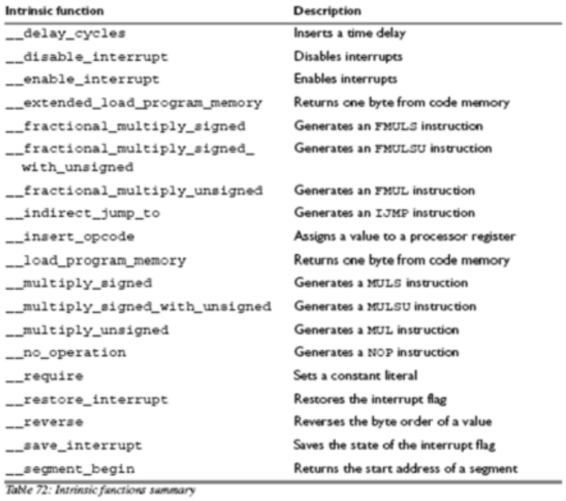
- Modify the previous program to use the exclusive-OR bitwise operator to toggle the port bit



IAR Intrinsic Functions

- The IAR compiler provides a number of intrinsic functions
- Intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines
- Intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions
- Intrinsic function names start with two underscores.
- Use #include <intrinsics.h> preprocessor command to make intrinsic functions available for use

List of IAR Embedded C Intrinsic Functions



Intrinsic function	Description
`__delay_cycles`	Inserts a time delay
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__extended_load_program_memory`	Returns one byte from code memory
`__fractional_multiply_signed`	Generates an `FMULS` instruction
`__fractional_multiply_signed_with_unsigned`	Generates an `FMULSU` instruction
`__fractional_multiply_unsigned`	Generates an `FMUL` instruction
`__indirect_jump_to`	Generates an `IJMP` instruction
`__insert_opcode`	Assigns a value to a processor register
`__load_program_memory`	Returns one byte from code memory
`__multiply_signed`	Generates a `MULS` instruction
`__multiply_signed_with_unsigned`	Generates a `MULSU` instruction
`__multiply_unsigned`	Generates a `MUL` instruction
`__no_operation`	Generates a `NOP` instruction
`__require`	Sets a constant literal
`__restore_interrupt`	Restores the interrupt flag
`__reverse`	Reverses the byte order of a value
`__save_interrupt`	Saves the state of the interrupt flag
`__segment_begin`	Returns the start address of a segment

Table 72: Intrinsic functions summary

__delay_cycles() Intrinsic Function

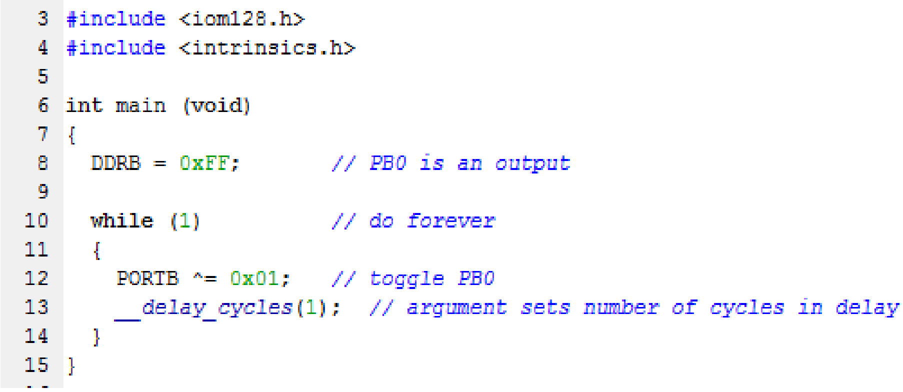
- In the bit toggling example we used the intrinsic function:

  __delay_cycles(unsigned long)

- This function makes the compiler generate code that takes the given amount of clock cycles to execute. That is, it inserts a time delay that lasts the specified number of cycles.

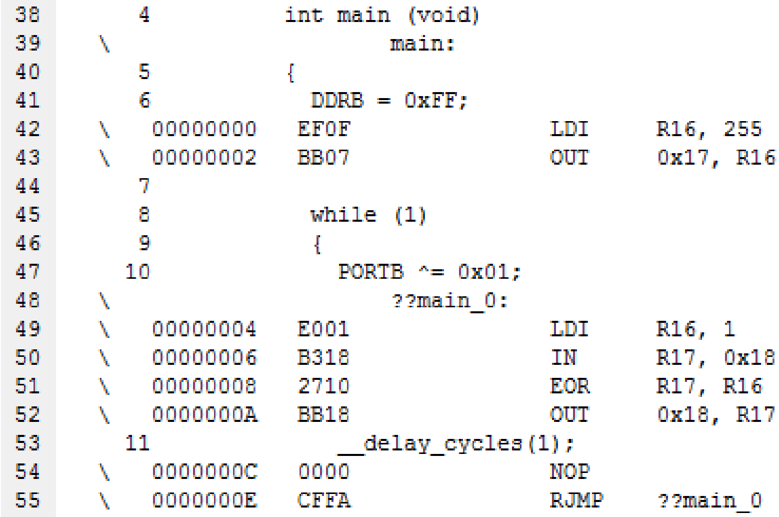
- Note: The specified unsigned long value must be a constant integer expression and not an expression that is evaluated at runtime.

Using __delay_cycles()



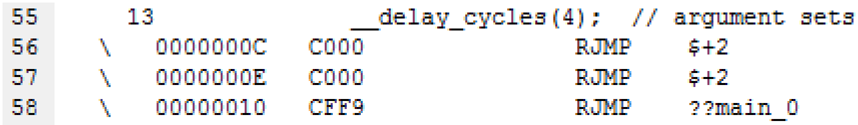
```
#include <iom128.h>
#include <intrinsics.h>

int main (void)
{
  DDRB = 0xFF;        // PB0 is an output

  while (1)           // do forever
  {
    PORTB ^= 0x01;    // toggle PB0
    __delay_cycles(1);  // argument sets number of cycles in delay
  }
}
```

Compiler Generated In-Line Code

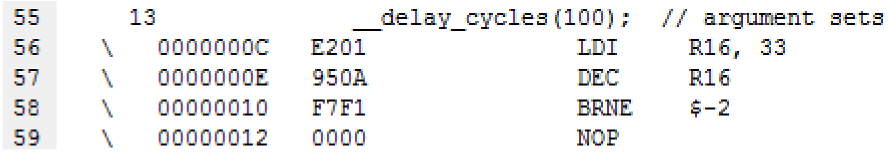


```
       4            int main (void)
  \                         main:
       5            {
       6              DDRB = 0xFF;
  \     00000000   EF0F                 LDI     R16, 255
  \     00000002   BB07                 OUT     0x17, R16
       7
       8              while (1)
       9              {
      10                PORTB ^= 0x01;
  \                         ??main_0:
  \     00000004   E001                 LDI     R16, 1
  \     00000006   B318                 IN      R17, 0x18
  \     00000008   2710                 EOR     R17, R16
  \     0000000A   BB18                 OUT     0x18, R17
      11                __delay_cycles(1);
  \     0000000C   0000                 NOP
  \     0000000E   CFFA                 RJMP    ??main_0
```

Code Generated for Different Cycle Values

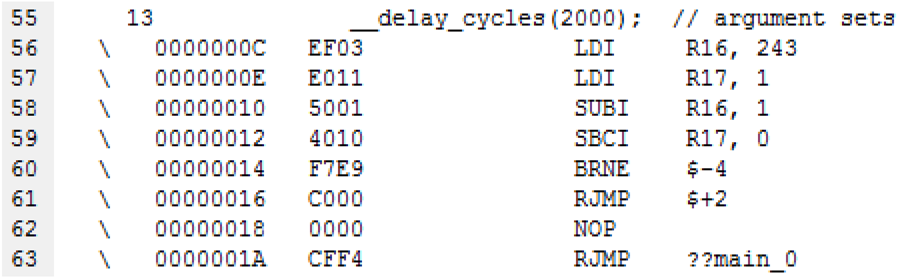


```
    13              __delay_cycles(4);  // argument sets
 \   0000000C   C000                RJMP    $+2
 \   0000000E   C000                RJMP    $+2
 \   00000010   CFF9                RJMP    ??main_0
```



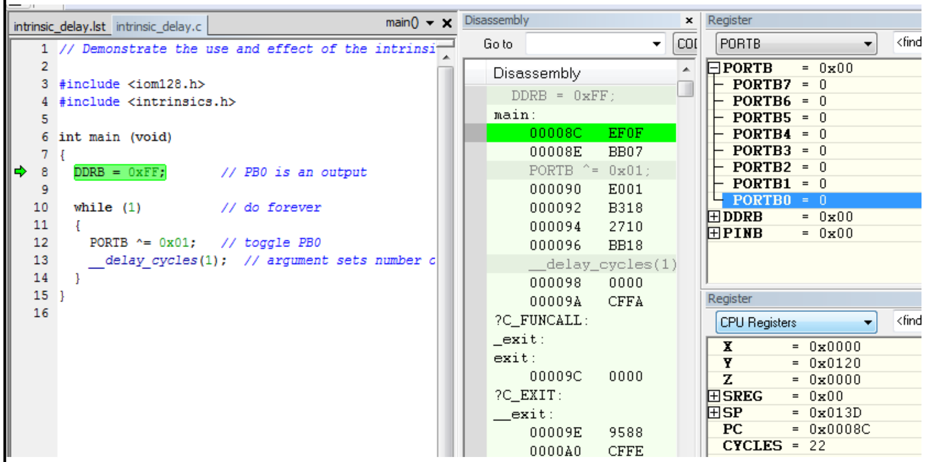
```
    13              __delay_cycles(100);  // argument sets
 \   0000000C   E201                LDI     R16, 33
 \   0000000E   950A                DEC     R16
 \   00000010   F7F1                BRNE    $-2
 \   00000012   0000                NOP
```

More Code Generated for a Different Cycle Value



```
    13              __delay_cycles(2000);  // argument sets
 \   0000000C   EF03                LDI     R16, 243
 \   0000000E   E011                LDI     R17, 1
 \   00000010   5001                SUBI    R16, 1
 \   00000012   4010                SBCI    R17, 0
 \   00000014   F7E9                BRNE    $-4
 \   00000016   C000                RJMP    $+2
 \   00000018   0000                NOP
 \   0000001A   CFF4                RJMP    ??main_0
```

Simulation in IAR Embedded Workbench



```
// Demonstrate the use and effect of the intrinsi

#include <iom128.h>
#include <intrinsics.h>

int main (void)
{
  DDRB = 0xFF;        // PB0 is an output

  while (1)           // do forever
  {
    PORTB ^= 0x01;    // toggle PB0
    __delay_cycles(1);  // argument sets number c
  }
}
```

```
Disassembly
   DDRB = 0xFF;
main:
       00008C    EF0F
       00008E    BB07
       PORTB ^= 0x01;
       000090    E001
       000092    B318
       000094    2710
       000096    BB18
       __delay_cycles(1)
       000098    0000
       00009A    CFFA
?C_FUNCALL:
_exit:
exit:
       00009C    0000
?C_EXIT:
__exit:
       00009E    9588
       0000A0    CFFE
```

```
⊟PORTB     = 0x00
 ├ PORTB7  = 0
 ├ PORTB6  = 0
 ├ PORTB5  = 0
 ├ PORTB4  = 0
 ├ PORTB3  = 0
 ├ PORTB2  = 0
 ├ PORTB1  = 0
 └ PORTB0  = 0
⊞DDRB      = 0x00
⊞PINB      = 0x00
```

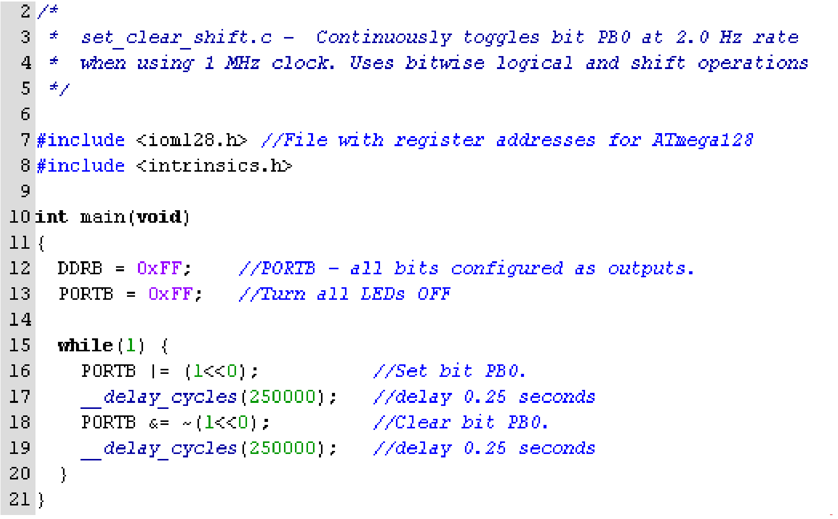
```
 X        = 0x0000
 Y        = 0x0120
 Z        = 0x0000
⊞SREG     = 0x00
⊞SP       = 0x013D
 PC       = 0x0008C
 CYCLES   = 22
```

Bitwise Shift Operators

Shift Operator	Symbol	Combined Operator and Assignment	Associates
Left shift	<<	<<=	left to right
Right shift	>>	>>=	left to right

- □ The shift operators shift the bits of the value of the left operand in the specified direction by the number of places given by the right operand.
- □ For a left shift the vacated positions are filled with 0s.
- □ For a right shift of an unsigned operand the vacated positions are filled with 0s. For signed types the result is compiler dependent. The vacated positions may be filled with 0s or they may be filled with copies of the sign bit. (IAR fills the vacated position with a copy of the sign bit)
- □ Result is undefined if right operand is negative or > than width of left operand

Setting and Clearing a Bit using Bitwise Shift Operators to Create a Mask



```
/*
 *  set_clear_shift.c -  Continuously toggles bit PB0 at 2.0 Hz rate
 *  when using 1 MHz clock. Uses bitwise logical and shift operations
 */

#include <iom128.h> //File with register addresses for ATmega128
#include <intrinsics.h>

int main(void)
{
  DDRB = 0xFF;     //PORTB - all bits configured as outputs.
  PORTB = 0xFF;    //Turn all LEDs OFF

  while(1) {
    PORTB |= (1<<0);           //Set bit PB0.
    __delay_cycles(250000);    //delay 0.25 seconds
    PORTB &= ~(1<<0);          //Clear bit PB0.
    __delay_cycles(250000);    //delay 0.25 seconds
  }
}
```

C Macro Review

- The C preprocessor is a macro processor that processes the source program text before it is read by the compiler
- Preprocessor command lines begin with the character #
- The preprocessor expands all macro calls that it encounters. After the preprocessor expands a macro it rescans the macro to determine if the expansion generated additional macro calls. If so, it expands these macro calls and then rescans the result. This process is repeated until no more macro calls are generated

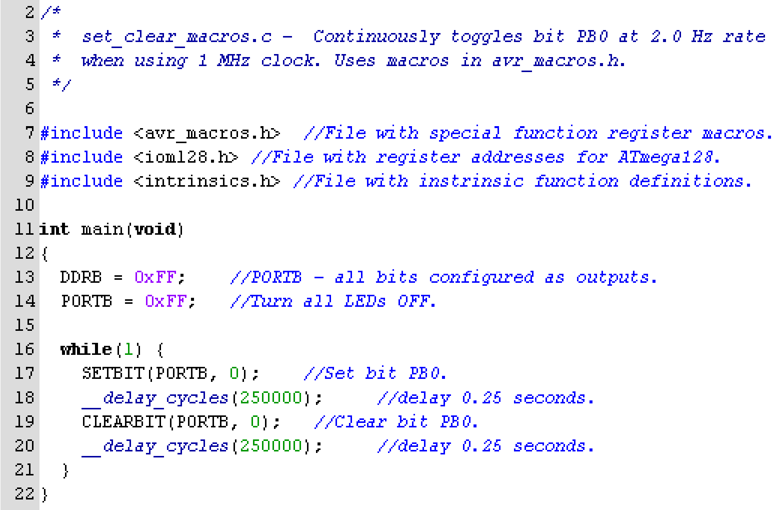
SETBIT, CLEARBIT, and TESTBIT Macros

- The file avr_macros.h provides a number of helpful macros including SETBIT and CLEARBIT

```
#define SETBIT(ADDRESS,BIT) ((ADDRESS) |= (1<<(BIT)))

#define CLEARBIT(ADDRESS,BIT)  ((ADDRESS) &= ~(1<<(BIT)))

#define TESTBIT(ADDRESS,BIT)  ((ADDRESS) & (1<<(BIT)))
```

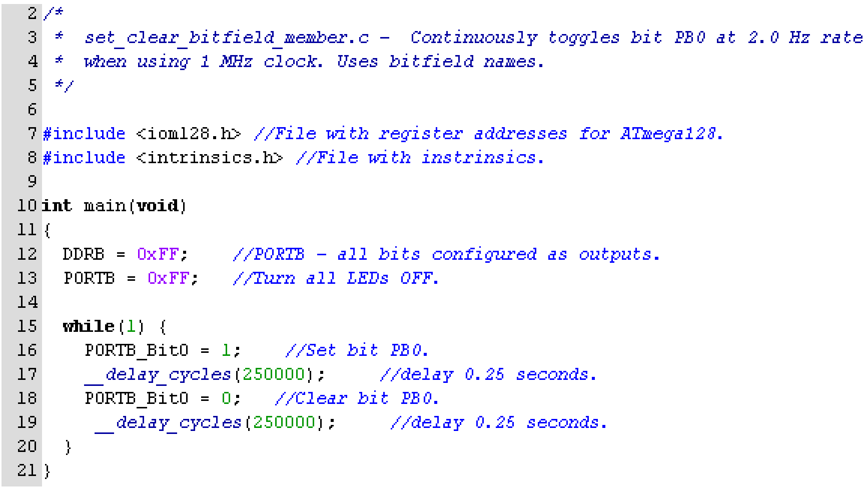
- Use of SETBIT and CLEARBIT produce the same code as programs using bitwise shift operators
- Note that TESTBIT does not cause an assignment and is used only to create a boolean result.

Toggling a Bit using Macros



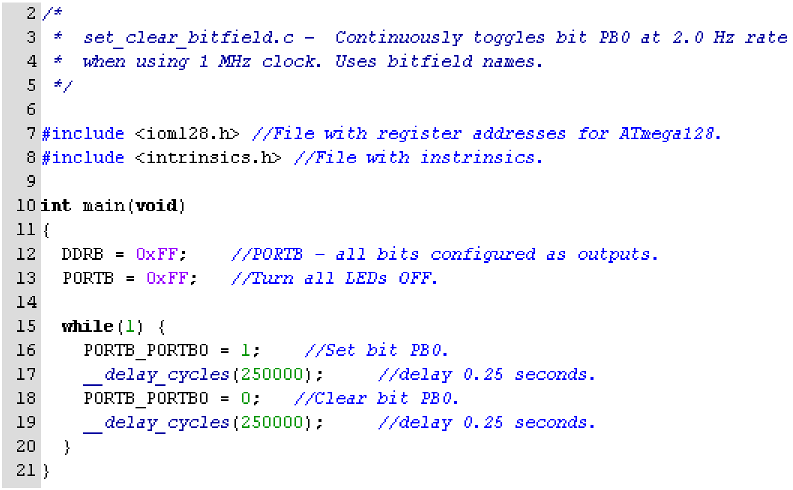
```
/*
 *  set_clear_macros.c -  Continuously toggles bit PB0 at 2.0 Hz rate
 *  when using 1 MHz clock. Uses macros in avr_macros.h.
 */

#include <avr_macros.h>  //File with special function register macros.
#include <iom128.h> //File with register addresses for ATmega128.
#include <intrinsics.h> //File with instrinsic function definitions.

int main(void)
{
  DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  PORTB = 0xFF;   //Turn all LEDs OFF.

  while(1) {
    SETBIT(PORTB, 0);    //Set bit PB0.
    __delay_cycles(250000);     //delay 0.25 seconds.
    CLEARBIT(PORTB, 0);   //Clear bit PB0.
    __delay_cycles(250000);     //delay 0.25 seconds.
  }
}
```

Toggling a Bit Using Bit Names

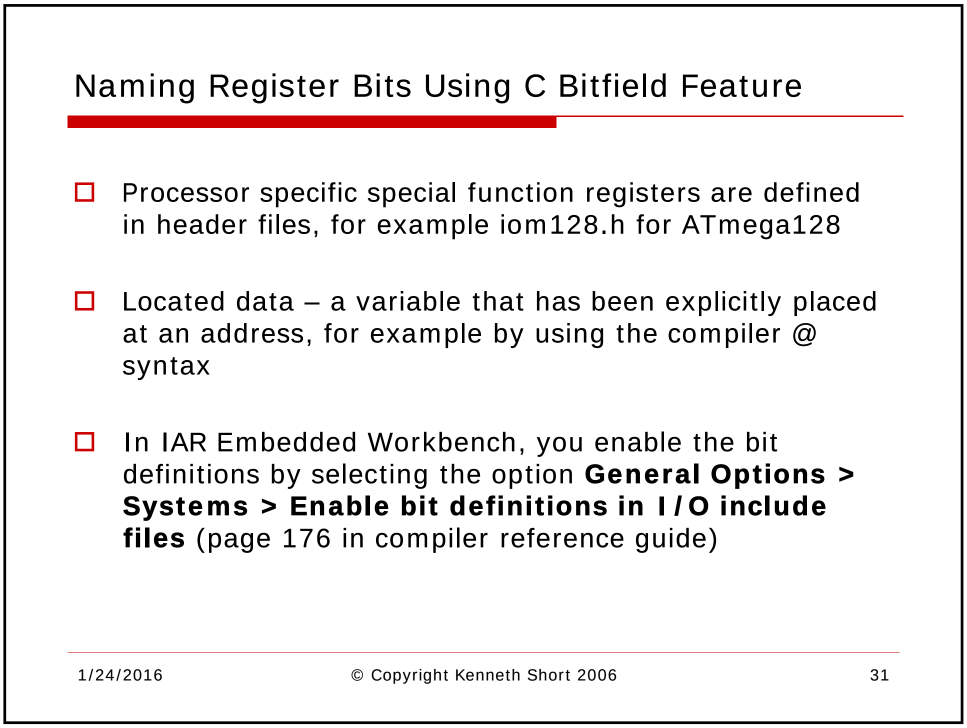


```
/*
 *  set_clear_bitfield_member.c -  Continuously toggles bit PB0 at 2.0 Hz rate
 *  when using 1 MHz clock. Uses bitfield names.
 */

#include <iom128.h> //File with register addresses for ATmega128.
#include <intrinsics.h> //File with instrinsics.

int main(void)
{
  DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  PORTB = 0xFF;   //Turn all LEDs OFF.

  while(1) {
    PORTB_Bit0 = 1;    //Set bit PB0.
    __delay_cycles(250000);     //delay 0.25 seconds.
    PORTB_Bit0 = 0;   //Clear bit PB0.
     __delay_cycles(250000);     //delay 0.25 seconds.
  }
}
```

Toggling a Bit Using Bit Names (2)



```
/*
 *  set_clear_bitfield.c -  Continuously toggles bit PB0 at 2.0 Hz rate
 *  when using 1 MHz clock. Uses bitfield names.
 */

#include <iom128.h> //File with register addresses for ATmega128.
#include <intrinsics.h> //File with instrinsics.

int main(void)
{
  DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  PORTB = 0xFF;   //Turn all LEDs OFF.

  while(1) {
    PORTB_PORTB0 = 1;    //Set bit PB0.
    __delay_cycles(250000);     //delay 0.25 seconds.
    PORTB_PORTB0 = 0;   //Clear bit PB0.
    __delay_cycles(250000);     //delay 0.25 seconds.
  }
}
```

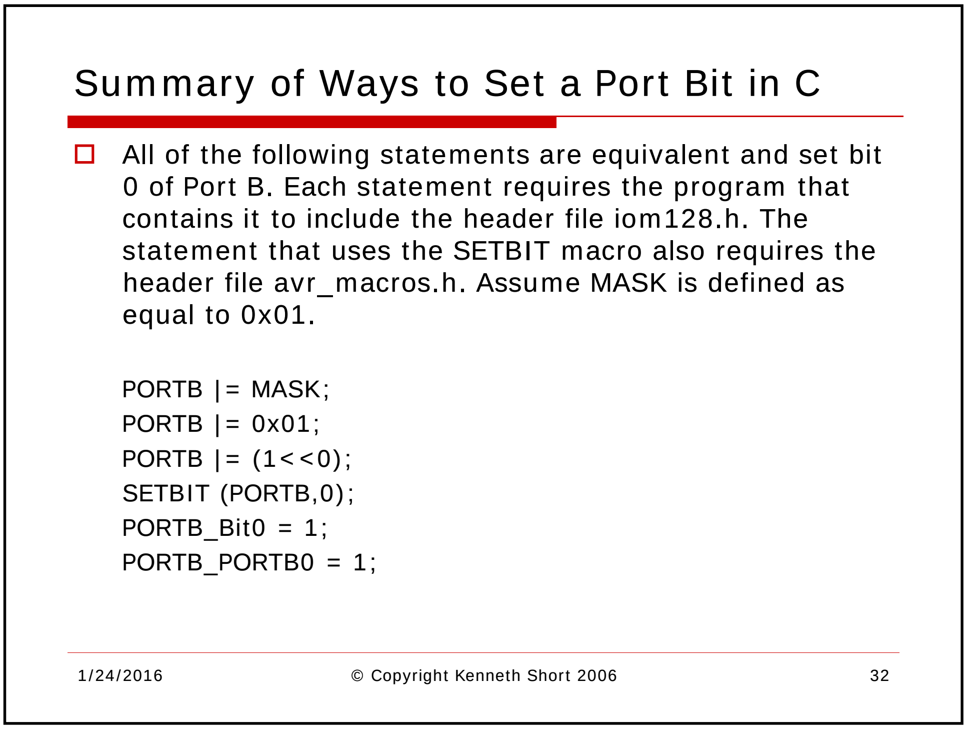


Naming Register Bits Using C Bitfield Feature

- Processor specific special function registers are defined in header files, for example iom128.h for ATmega128

- Located data – a variable that has been explicitly placed at an address, for example by using the compiler @ syntax

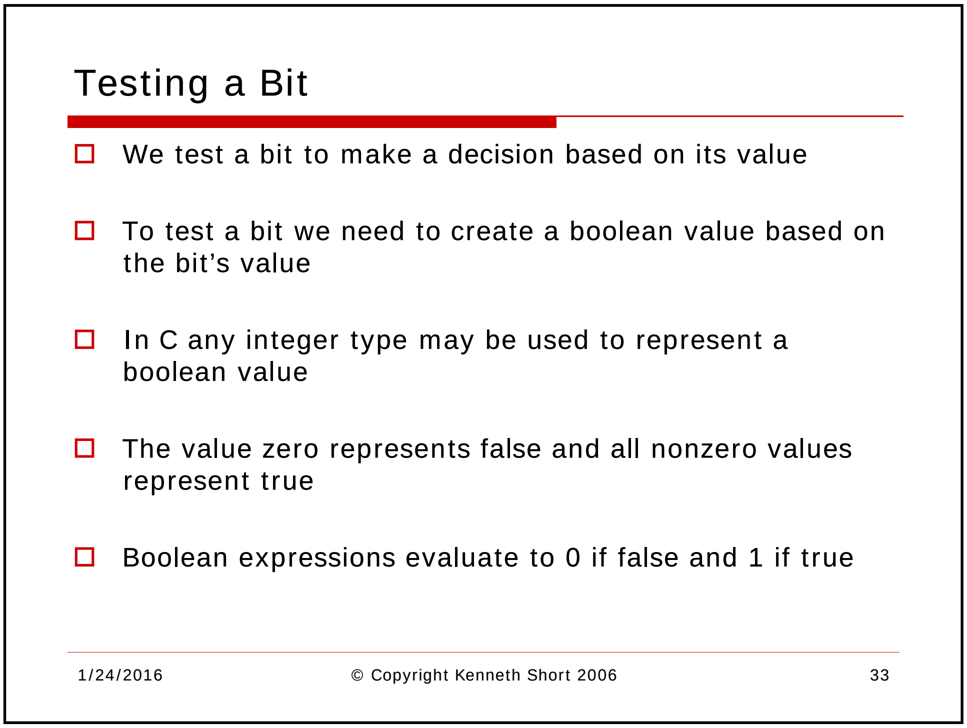
- In IAR Embedded Workbench, you enable the bit definitions by selecting the option **General Options > Systems > Enable bit definitions in I/O include files** (page 176 in compiler reference guide)



Summary of Ways to Set a Port Bit in C

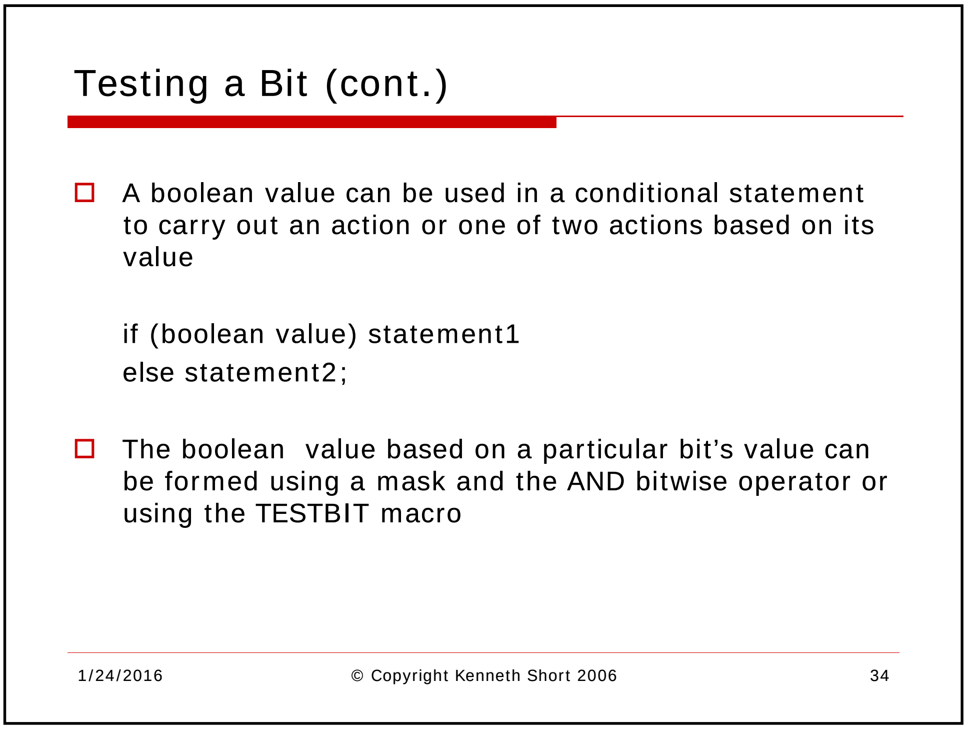
- All of the following statements are equivalent and set bit 0 of Port B. Each statement requires the program that contains it to include the header file iom128.h. The statement that uses the SETBIT macro also requires the header file avr_macros.h. Assume MASK is defined as equal to 0x01.

```
PORTB |= MASK;
PORTB |= 0x01;
PORTB |= (1<<0);
SETBIT (PORTB,0);
PORTB_Bit0 = 1;
PORTB_PORTB0 = 1;
```



Testing a Bit

- □ We test a bit to make a decision based on its value
- □ To test a bit we need to create a boolean value based on the bit’s value
- □ In C any integer type may be used to represent a boolean value
- □ The value zero represents false and all nonzero values represent true
- □ Boolean expressions evaluate to 0 if false and 1 if true



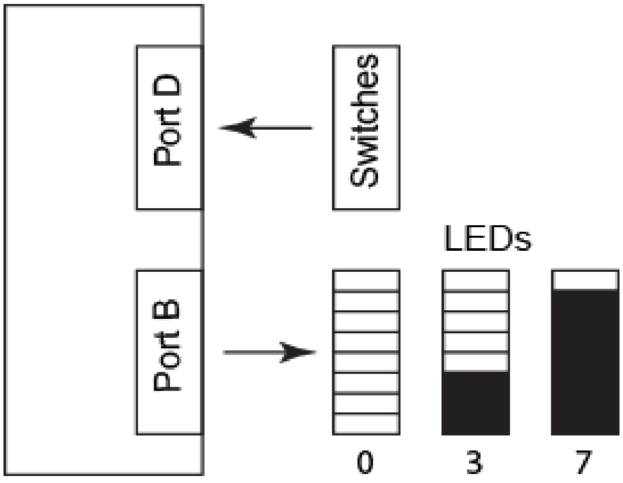
Testing a Bit (cont.)

- □ A boolean value can be used in a conditional statement to carry out an action or one of two actions based on its value

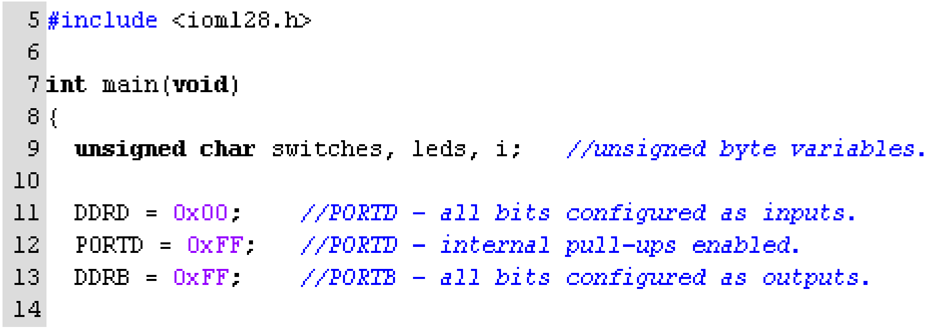
```
if (boolean value) statement1
else statement2;
```

- □ The boolean value based on a particular bit’s value can be formed using a mask and the AND bitwise operator or using the TESTBIT macro

Display The Number of Switches in Their 1s Positions As a Bargraph

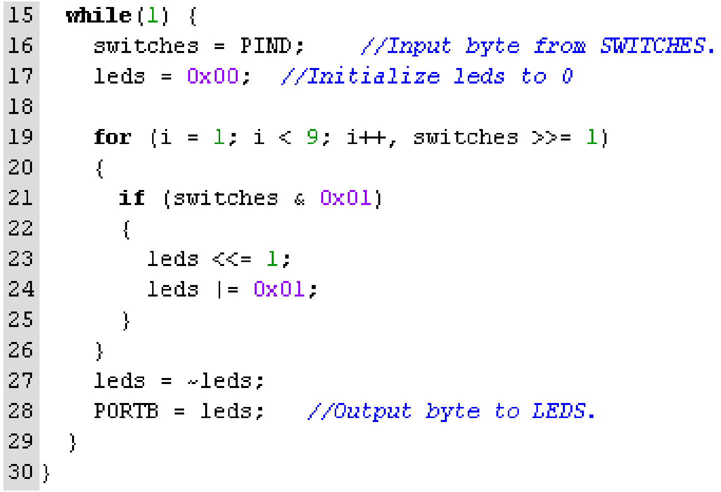


Switches Level Program in C



```
#include <ioml28.h>

int main(void)
{
  unsigned char switches, leds, i;   //unsigned byte variables.

  DDRD = 0x00;      //PORTD - all bits configured as inputs.
  PORTD = 0xFF;     //PORTD - internal pull-ups enabled.
  DDRB = 0xFF;      //PORTB - all bits configured as outputs.

```

Switches Level Program in C (cont.)

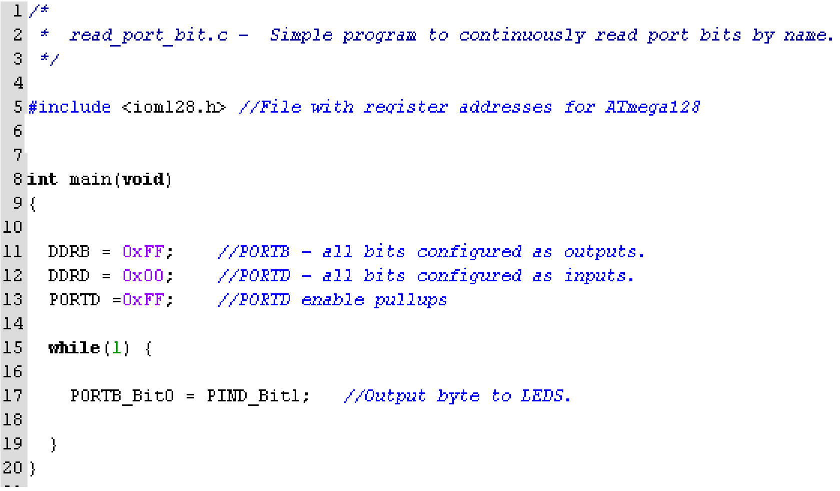


```
  while(1) {
    switches = PIND;    //Input byte from SWITCHES.
    leds = 0x00;  //Initialize leds to 0

    for (i = 1; i < 9; i++, switches >>= 1)
    {
      if (switches & 0x01)
      {
        leds <<= 1;
        leds |= 0x01;
      }
    }
    leds = ~leds;
    PORTB = leds;   //Output byte to LEDS.
  }
}
```

Exercise Problems

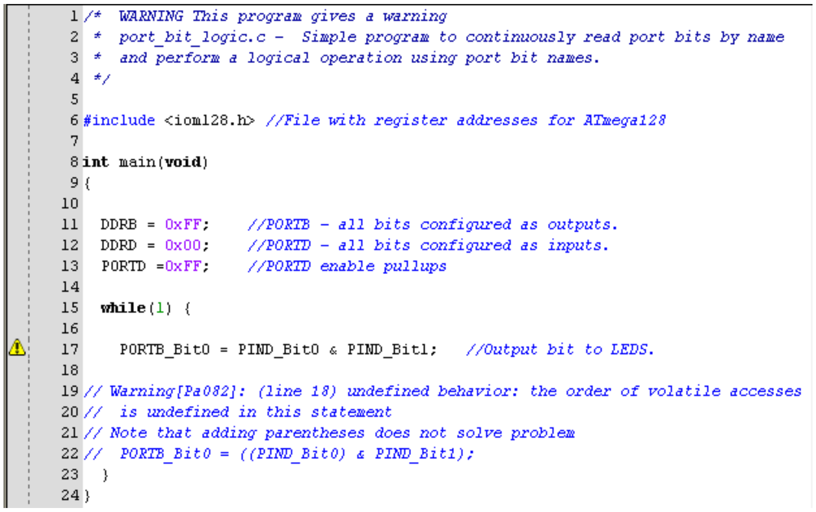
- Try single stepping the previous program using C-SPY in the simulator mode. It will give you a clear understanding of the semantics of the for statement in C.

Reading a Port Bit by Name



```
/*
 *  read_port_bit.c -  Simple program to continuously read port bits by name.
 */

#include <iom128.h> //File with register addresses for ATmega128


int main(void)
{

  DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  DDRD = 0x00;    //PORTD - all bits configured as inputs.
  PORTD =0xFF;    //PORTD enable pullups

  while(1) {

    PORTB_Bit0 = PIND_Bit1;   //Output byte to LEDS.

  }
}
```

Problematic Logic Operation on Named Port Bits

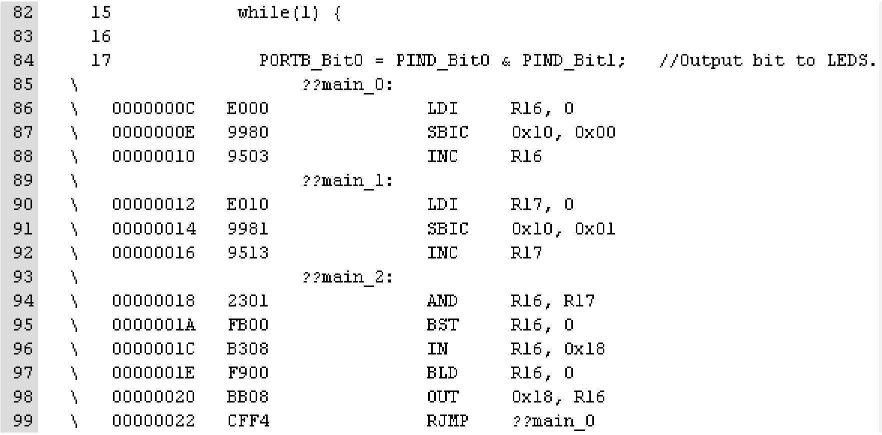


```
 1 /*  WARNING This program gives a warning
 2  *  port_bit_logic.c -  Simple program to continuously read port bits by name
 3  *  and perform a logical operation using port bit names.
 4  */
 5
 6 #include <iom128.h> //File with register addresses for ATmega128
 7
 8 int main(void)
 9 {
10
11   DDRB = 0xFF;    //PORTB - all bits configured as outputs.
12   DDRD = 0x00;    //PORTD - all bits configured as inputs.
13   PORTD =0xFF;    //PORTD enable pullups
14
15   while(1) {
16
17     PORTB_Bit0 = PIND_Bit0 & PIND_Bit1;    //Output bit to LEDS.
18
19 // Warning[Pa082]: (line 18) undefined behavior: the order of volatile accesses
20 //  is undefined in this statement
21 // Note that adding parentheses does not solve problem
22 //  PORTB_Bit0 = ((PIND_Bit0) & PIND_Bit1);
23   }
24 }
```

Volatile

- A register in a computer is defined as volatile if the value stored in the register can change even though the program itself is not storing a new value there.
- For example, an input port on a microcontroller is volatile. ATmega128 input ports are defined as volatile in the file iomacro.h, which is included in the compilation by the file iom128.h
- C provides a volatile type qualifier to allow you to inform the compiler if any registers are volatile. This prevents the compiler from performing optimizations that change the logic of the program.

Compiler Generated Code to Read Port Twice, Once for Each Bit



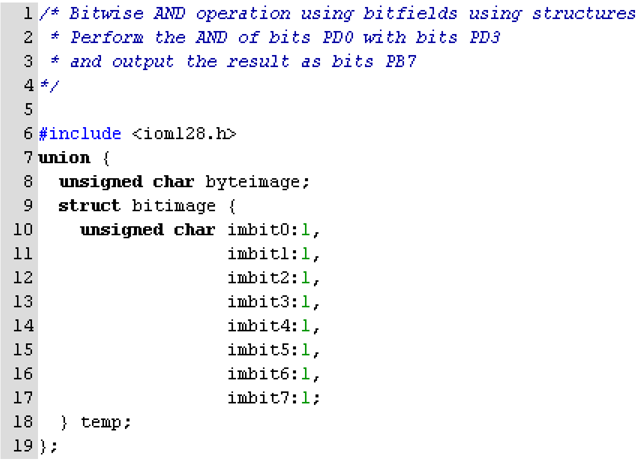
```
    15             while(1) {
    16
    17               PORTB_Bit0 = PIND_Bit0 & PIND_Bit1;   //Output bit to LEDS.
  \                      ??main_0:
  \   0000000C   E000                LDI     R16, 0
  \   0000000E   9980                SBIC    0x10, 0x00
  \   00000010   9503                INC     R16
  \                      ??main_1:
  \   00000012   E010                LDI     R17, 0
  \   00000014   9981                SBIC    0x10, 0x01
  \   00000016   9513                INC     R17
  \                      ??main_2:
  \   00000018   2301                AND     R16, R17
  \   0000001A   FB00                BST     R16, 0
  \   0000001C   B308                IN      R16, 0x18
  \   0000001E   F900                BLD     R16, 0
  \   00000020   BB08                OUT     0x18, R16
  \   00000022   CFF4                RJMP    ??main_0
```

Compiler Warning

- The compiler gives a warning because it generates code to read the port twice, once for each bit.
- Since the port is volatile, its value could change between readings.
- Note that adding parentheses does not solve the problem:

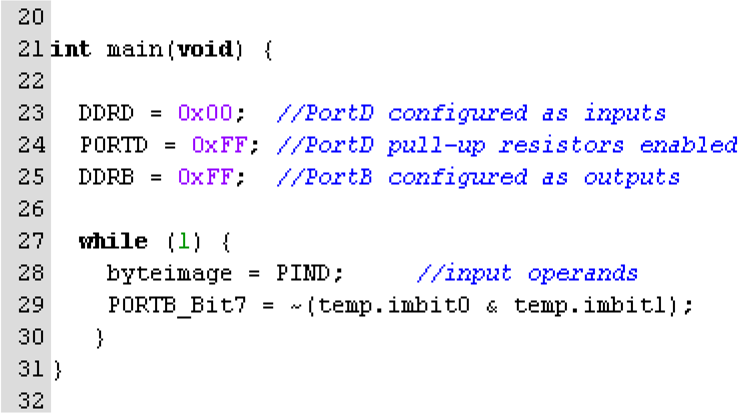
  PORTB_Bit0 = ((PIND_Bit0) & PIND_Bit1);

Accessing Bits From Structure Variable



```
/* Bitwise AND operation using bitfields using structures
 * Perform the AND of bits PD0 with bits PD3
 * and output the result as bits PB7
*/

#include <iom128.h>
union {
  unsigned char byteimage;
  struct bitimage {
    unsigned char imbit0:1,
                  imbit1:1,
                  imbit2:1,
                  imbit3:1,
                  imbit4:1,
                  imbit5:1,
                  imbit6:1,
                  imbit7:1;
  } temp;
};
```

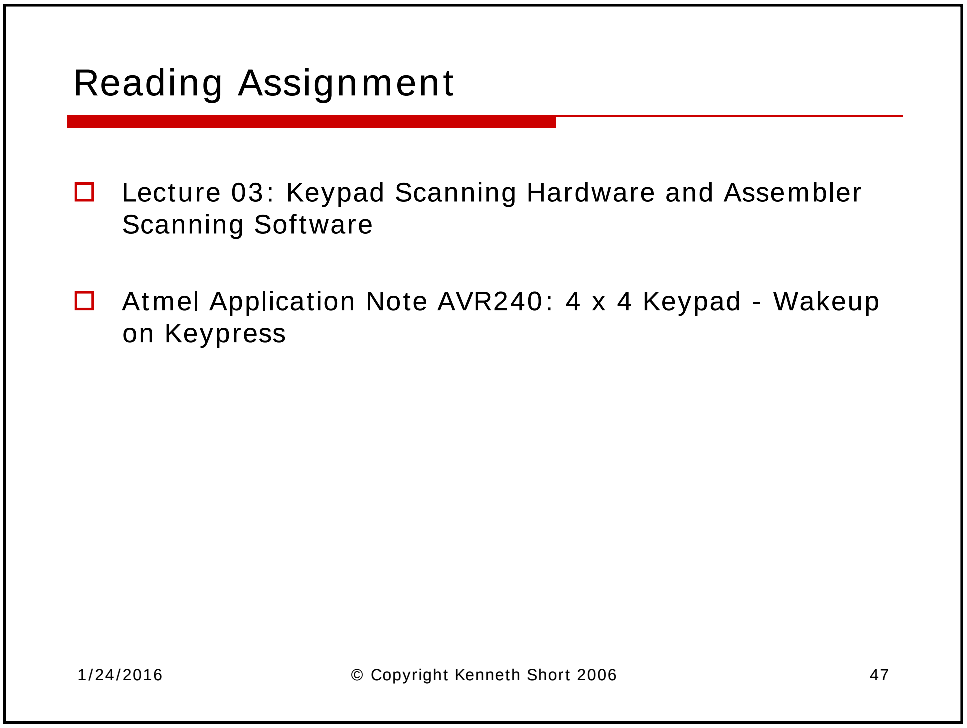
Accessing Bits From Structure (cont.)



```

int main(void) {

  DDRD = 0x00;  //PortD configured as inputs
  PORTD = 0xFF; //PortD pull-up resistors enabled
  DDRB = 0xFF;  //PortB configured as outputs

  while (1) {
    byteimage = PIND;      //input operands
    PORTB_Bit7 = ~(temp.imbit0 & temp.imbit1);
   }
}

```

Next Class

- Keypad Scanning



Reading Assignment

- □ Lecture 03: Keypad Scanning Hardware and Assembler Scanning Software
- □ Atmel Application Note AVR240: 4 x 4 Keypad - Wakeup on Keypress