09-C Programming

Ahmad Abdullah, PhD
abdullah@uvic.ca
https://web.uvic.ca/~abdullah/csc230

Lectures: MR 10:00 – 11:20 am
Location: ECS 125

Outline

- Context
- hello_blink.c
- Variables
- Bitwise operations
- A little bit more control flow
- (What is really happening in assembler)
- Functions
- Interrupt handlers
- C and assembly
- Role played by linker

Context

- All of our coding so far has been in assembler
- For some problem domains, this is absolutely necessary
  - Where space and speed are overriding concerns
  - Where the only programming tool is an assembler
- For other problem domain, a high-level language is more convenient
- C programming language
  - Invented at Bell Labs in the early 1970s
  - Original purpose: coding kernel of an operating system
  - Writing the kernel in C made it easier to port OS to different hardware architectures

Context (Cont.)

- For CSC 230:
  - Just the rudiments of C
  - Language is covered (and used!) much more extensively in SENG 265
- Our interest here:
  - showing how tasks we have until now completed in assembly...
  - ... might be implemented in a higher-level language

Context (Cont.)

We will be introducing language features at a much faster pace than would occur in a first-year course.

From second-year onwards, we instructors assume our students have already completed several semesters of programming.

Therefore, please use relevant concepts you have learned in other languages in order to learn and understand any new language you are given.

Hello, blink

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

Hello, blink (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

All operations in C programs are contained within **functions**. And every full program must contain exactly one function named **main**.

Functions may take parameters. For this program, main() takes none. Functions may also return values. Here main() returns an integer. (The main() function only ever returns a value to its "environment".)

The code and the scope for a function are bounded by curly braces ({, }). Functions may not be nested.

Comments are surrounded by the /* and */ symbols (and therefore comments may span lines).

Hello, blink (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

Some of the things we can use in C depend upon the compiler, linker, and overall development environment. We are using AVR Studio in this course, and in our project we have specified the mega2560 board

Consequence: For this program, the C language knows the names of I/O registers.

Hello, blink (Cont.)

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

Also seen here is the assignment of values to some registers. The version of C in AVR Studio knows to use out or sts as is appropriately for the registers.

And notice that the "=" symbol here means assignment. Normally we would see this with variables, but we use it here with registers.

Hello, blink (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001[illegible]0;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b0000000[illegible]
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

C functions may call other functions. These functions may be those we write (and we'll look at these later) or those available to us in the development environment.

Naming conventions often help us determine if the function is "built in" or one of our own. Here the function begins with an underscore (i.e., provided by the development environment).

Hello, blink (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

The _delay_ms() function is "known" to the compiler in part because we've included a file that mentions its signature.

IMPORTANT!!

#include is not the same as Java's import.

#include does not always "include" the source code for functions (i.e., very different from behavior of #include in assembler). Instead it makes available constants and function signatures to this program.

When we talk about linking we'll learn how our compiled code really accesses library code.

Hello, blink (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

This particular constant (F_CPU) is needed by the code in _delay_ms().

Even though the AVR Studio project holding this code knows the code is meant for a mega2560, the library code still needs to be told by us of the board's clock rate.

#define supports a form of textual substitution. (Read this example as: When compiling code, replace the text "F_CPU" is found, replace it with the text "16000000UL".)

Note: the "UL" at the end of "16000000" means "unsigned long integer". More on this later...

Hello, blink (Cont.)

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

... and this include file gives the C compiler the meaning of DDRB, DDRL, PORTL and PORTB (amongst many many other items).

Hello, blink (Cont.)

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;

   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
      _delay_ms(500);
   }
   /* Never reached in this program.... */
   return 0;
}
```

C supports many kinds of control-flow constructions, including **for-loops**. For-loops in C are almost identical with for-loops in Java (i.e., format of header, use of curly braces, meanings of “break” and “continue”).

This particular for-loop example has a completely empty header! This means there are no conditions used to determine whether the loop runs or not. It is an infinite loop!

AVR programs will normally have the kind of structure present in this initial program: **one big infinite loop in which other code exists.**

Hello, blink (Cont.)

Integer literals may be expressed in several different formats and bases.

```c
#define F_CPU 16000000UL
```
decimal

```c
#include <avr/io.h>
#include <util/delay.h>

int main() {
   /* Set direction for bits on port */
   DDRB = 0xff;
   DDRL = 0xff;
```
hexadecimal

```c
   for (;;) {
      /* Turn lights ON for half-a-second. */
      PORTL = 0b10101010;
      PORTB = 0b00001010;
```
binary

```c
      _delay_ms(500);

      /* Turn lights OFF for half-a-second */
      PORTL = 0b00000000;
      PORTB = 0b00000000;
```
binary

```c
      _delay_ms(500);
```
decimal

```c
   }
   /* Never reached in this program.... */
   return 0;
}
```

Variables

- In our hello_blink program we did not use any variables!!!
  - All assignments were to AVR registers!
- In C (as in Java), variables must be declared before use
  - Declaration indicates the type of the variable.
  - The type implies the amount of memory needed for the variable...
  - ... as well as legal operations permitted with the variable.
- Example: integer types uint8_t and int

flash 01

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRL = 0xff;
    uint8_t pattern = 0b00000010;

    for (;;) {
        PORTL = pattern;
        pattern = pattern << 2;
        if (pattern == 0) {
            pattern = 0b00000010;
        }
        _delay_ms(500);
    }

    /* Never reached in this program....
     * but keep the compiler happy.
     */
    return 0;
}
```

Programs causes four LEDs to turn on in sequence, one at a time, over and over again.

flash 01 (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRL = 0xff;
    uint8_t pattern = 0b00000010;

    for (;;) {
        PORTL = pattern;
        pattern = pattern << 2;
        if (pattern == 0) {
            pattern = 0b00000010;
        }
        _delay_ms(500);
    }

    /* Never reached in this program....
     * but keep the compiler happy.
     */
    return 0;
}
```

One variable is declared in function scope. Only the code with main() is able to access the pattern variable.

The name is pattern and it is an unsigned integer of 8 bits.

pattern is a local variable.

An initial value is given to the variable (i.e., turn on fourth light from the right – i.e., pin 48).

flash 01 (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRL = 0xff;
    uint8_t pattern = 0b00000010;

    for (;;) {
        PORTL = pattern;
        pattern = pattern << 2;
        if (pattern == 0) {
            pattern = 0b00000010;
        }
        _delay_ms(500);
    }

    /* Never reached in this program....
     * but keep the compiler happy.
     */
    return 0;
}
```

We can assign the value of a variable to a register.

Bit shifts left and right may be performed on contents of the variable.

flash 01 (Cont.)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRL = 0xff;
    uint8_t pattern = 0b00000010;

    for (;;) {
        PORTL = pattern;
        pattern = pattern << 2;
        if (pattern == 0) {
            pattern = 0b00000010;
        }
        _delay_ms(500);
    }

    /* Never reached in this program....
     * but keep the compiler happy.
     */
    return 0;
}
```

Here we see our first conditional statement (i.e., selection).

The syntax is identical to that in Java. There is one twist, however.

When a conditional expression is evaluated: zero means false, all other values means true.

flash 02

```c
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

uint8_t pattern = 0b00000010;

int main() {
    DDRL = 0xff;
    for (;;) {
        PORTL = pattern;
        pattern = pattern << 2;
        if (pattern == 0) {
            pattern = 0b00000010;
        }
        _delay_ms(500);
    }

    /* Never reached in this program....
     * but keep the compiler happy.
     */
    return 0;
}
```

This program behaves precisely the same as flash_02...

... but it differs in than pattern is no longer local to main, but is a program-scope variable (i.e., sometimes called a global variable).

All functions in the program can use this declaration of pattern.

mathy 01

```
int main() {
   int m = 16791;
   int n = 2958;
   int p;

   p = m +  n;

   long int s = 3141592;
   long int t = 5358979;
   long int u;

   u = s + t;

   /* calculations involving p, u ...
    *
    */
   return 0;
}
```

We have three int variables (i.e. signed integers...)

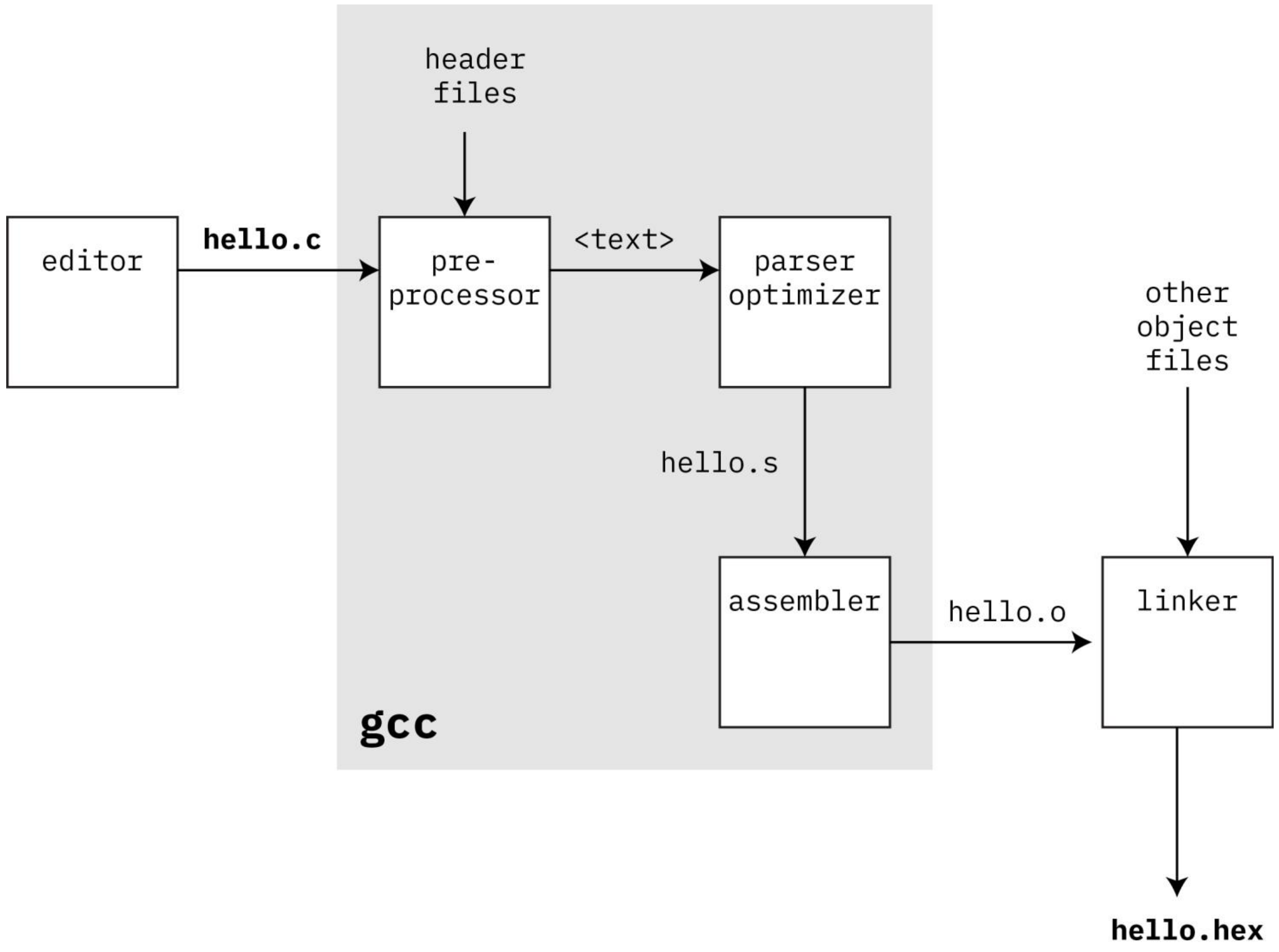
We have three long int variables (i.e., twice as large as an int ...)

But int and long int do not exist in the AVR architecture! So, what is happening?

Variables

- Higher level languages (such as C) create the abstraction of new types on top of the computer architecture’s existing types
  - An int in AVR C is a 16-bit signed integer (i.e., two bytes)
  - A long int in AVR C is a 32-bit signed integer (i.e., four bytes)
- The abstraction also includes the implementation of operations on the type
  - And these use the underlying architecture’s operations
- It is time for us to peek at what the compiler creates

Data flow through compiler



mathy 01

```c
int m = 16791;
int n = 2958;
int p;

p = m +  n;

long int s = 3141592;
long int t = 5358979;
long int u;

u = s + t;
```

```
 ...
lds r18,n
lds r19,(n)+1
lds r24,m
lds r25,(m)+1
add r18,r24
adc r19,r25
 ...

lds r20,t
lds r21,(t)+1
lds r22,(t)+2
lds r23,(t)+3
lds r24,s
lds r25,(s)+1
lds r26,(s)+2
lds r27,(s)+3
add r20,r24
adc r21,r25
adc r22,r26
adc r23,r27
 ...
```

We focus here on just the assembly code for addition operations.

The assembler uses the Gnu format (i.e., not exactly the same as AVR assembler syntax).

Rather than using HIGH and LOW, the byte at the variable's address (low endian!!) is directly referenced.

To compute an address, notice the use of parentheses around name of variable.

Array variables

- Arrays are simply contiguous regions of memory storing values of the same type
- C array usage is similar to that in Java
  - When declaring an array, we specify its type and number of elements.
  - When reading or writing array elements, we use the square-bracket notation
- Intuition: when accessing array elements, underlying machine code must calculate element address
  - C takes care of those calculations for us!

Array Example

```
int m[10];   /* Each int is two bytes */

int main() {
   for (int i = 0; i < 10; i++) {
      m[i] = i + 4;
   }
}
```

Notice that the compiler is clever enough to recognize our loop is only ever used to access the array (therefore it tests loop exit by examining array addresses).

The assembly generated by a compiler is often optimized like this...

Bitwise operations

- Our coding this semester has often been concerned with bit sequences
  - Assigning them to I/O registers
  - Reading them into general-purpose registers
  - Setting and clearing individual bits
  - Testing whether individual bits are set or cleared
- We can express bit operations in C
  - but these are not quite as tight and specific as AVR assembler...
  - ... but recall that C has to work for all kinds of architectures (and not just one specific kind)

Bitwise != logical operations

- Some symbols used for bitwise operations look similar to others seen in Java
- Must remember that:
  - bitwise operations work at the level of individual bits in our arguments
  - logical operations work at the level of the integer values represented by arguments

Bitwise AND vs. logical AND

- C’s bitwise AND is the same as AVR’s and instruction
  - The AVR instruction, however, can only work on bytes
  - The C instruction can also work with larger values

```
uint8_t a    = 0b10101100;
uint8_t mask = 0b00000110;

uint8_t result 1 = a & mask;  /* bitwise AND */
uint8_t result_2 = a && mask; /* logical AND */
```

```
result_1 → 4 (i.e., 0b00000100)

result_2 → 1
```

Bitwise AND vs. logical AND

- C's bitwise AND is the same as AVR's and instruction
  - The AVR instruction, however, can only work on bytes
  - The C instruction can also work with larger values

```
int b = 0x8f1d;
int c = 59;

int result_3 = b & c; /* which means the same
as c & b */
```

```
result_3 → 0x19
```

Bitwise != logical operations

- Before looking at more bitwise operations...
- ... we must examine again the significance of C treating zero as false.
  - Any value that is non-zero is true
- This means we can use the result of expressions from bitwise operations in the same places as logical operations
- That is:
  - Even though bitwise != logical ...
  - ... we can exploit the meaning of false to write less code!

Some example

This works...

```
int d = 0x88ff;
int mask_2 = 0x0002;

if ((d & mask_2) == mask_2) {
....
}
```

But this is much cleaner

```
int d = 0x88ff;
int mask_2 = 0x0002;

if (d & mask_2) {
....
}
```

But, be Careful!

This compiles...

```
int e = 4;
int f = 2;

if (e && f) {
    /* will always arrive here */
}
```

As does this...

```
int e = 4;
int f = 2;

if (e & f) {
   /* will never arrive here */
}
```

Confusing bitwise and logical operators is the source of subtle and difficult-to-find bugs.

Therefore, be **very careful** with mixing and matching the two.

Shorter forms involving assignment

- A common idiom is to apply a bitwise operator on some operand/variable...
- ... and then store the result back into that variable ...
- ... and to do this in one statement.
- These statements are sometimes said to involve **bitwise assignment operators**

```c
uint8_t a    = 0b10101100;
uint8_t mask = 0b00000110;

a &= mask;  /* equivalent to "a = a & mask" */
```

```c
uint8_t a    = 0b10101100;
uint8_t mask = 0b00000110;

mask &= a;  /* NOT equivalent to "a = mask & a" */
```

When using these short forms, ensure you know precisely what variable you want to be modified...

Some example of bit-wise operations

```
uint8_t a = 0;
uint8_t flag_X = 0b10000000;
uint8_t flag_Y = 0b10001000;

a = a | flag_X;  (also a |= flag_X)
a = a | flag Y;  (also a |= flag_Y)
```

bitwise OR

```
uint8_t a = 0b10100111;
uint8_t b = 0b10100111;
uint8_t flip_all = 0xff;
uint8_t flip_one = 0x08;

a = a ^ flip_all; (also a ^= flag_X)
b = b ^ flip_one; (also b ^= flag_Y)
```

bitwise XOR

```
uint8_t a = 0xCC;

a = ~a;
```

bitwise NOT

Some example of bit-wise operations (Cont.)

```
uint8_t a = 0b10100111;
int shift = 3;
a = a << shift;  (also a <<= shift)
```

shift left

```
uint8_t a = 0b10100111;
int shift = 2;
a = a >> shift;  (also a >>= shift)
```

shift right

```
uint8_t a = 0xCC;

a = a & ~077;
```

combinations (e.g., set the last six bits of a to zero)

```
uint8_t x = 0b11010010;

x = (x >> 1) | ((x << 7) & 0x80);
```

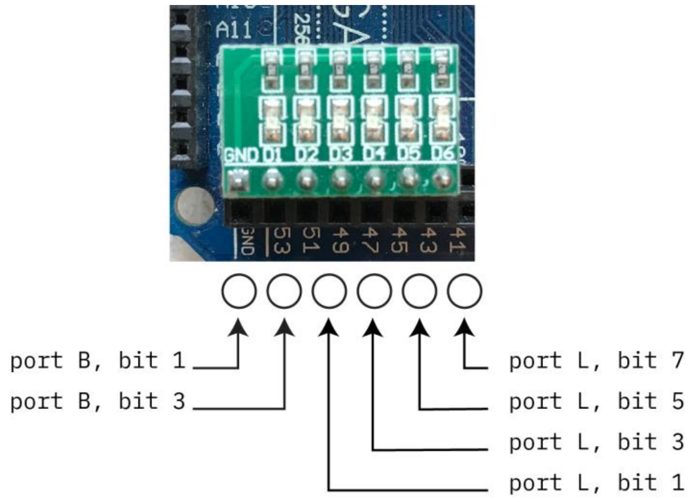
roll-your-own rotate right

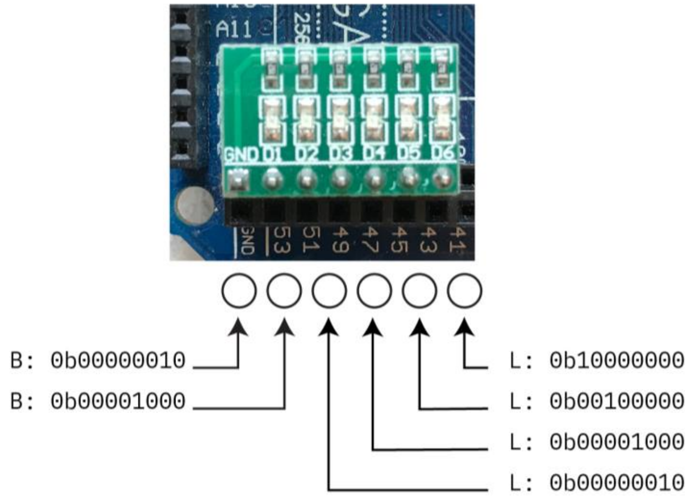
A little bit more control flow

- C has the “usual suspects”
  - for loop
  - top-tested while
  - bottom-tested do-while
  - break and continue
  - if, if-else, multiway if (i.e., if ... else if ... else if)
  - switch
- These look exactly the same as in Java...
- ... and again the big difference is in the way zero is interpreted as false, and all non-zero values interpreted as true

Example: leds_on

- This little programming study will look at multi-way ifs and the switch statement
- It was also show how we define functions in C.
- Consider a function we decide to call leds_on:
  - Given some integer value up to and including 6...
  - ... turn on that number of LEDs
- We turn on the LEDs from right to left.





Main loop

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

/* leds_on code here... */

int main() {
/* Set direction for bits on port */
DDRB = 0xff;
DDRL = 0xff;

uint8_t counter = 0;

for (;;) {
    leds_on_C(counter % 7);
    _delay_ms(500);
    counter++;
}

    /* Never reached in this program.... */
    return 0;
}
```

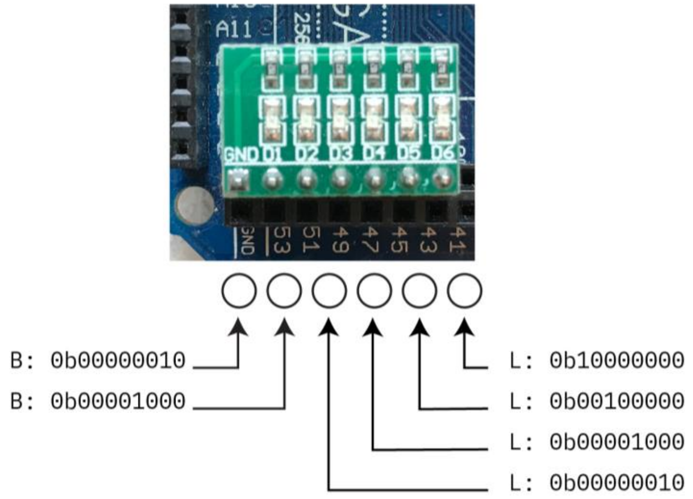
Programs causes six LEDS to turn on in sequence, one at a time, over and over again.

The behavior depends upon an implementation of leds_on (which accepts a value from 0 up to and including 6, and uses this to determine what LEDs are turned on).

Main loop

```c
void leds_on_A (uint8_t num) {
    if (num == 6) {
        PORTL = 0b10101010; PORTB = 0b00001010;
    } else if (num == 5) {
        PORTL = 0b10101010; PORTB = 0b00001000;
    } else if (num == 4) {
        PORTL = 0b10101010; PORTB = 0b00000000;
    } else if (num == 3) {
        PORTL = 0b10101000; PORTB = 0b00000000;
    } else if (num == 2) {
        PORTL = 0b10100000; PORTB = 0b00000000;
    } else if (num == 1) {
        PORTL = 0b10000000; PORTB = 0b00000000;
    } else {
        PORTL = 0b00000000; PORTB = 0b00000000;
    }
}
```

Notice function declaration syntax (i.e., similar to main() – but here there is a single parameter expected with a type of uint8_t).



Example: leds_on (Cont.)

- Our first attempt works nicely
  - The multiway-if checks for the value of the number
  - If the number is from 1 to 6, the correct PORTL and PORTB bits are set.
  - Anything else (include zero and numbers greater than 6) results in clearing all PORTL and PORTB bits
- We can do a little bit better
  - The code structure is quite regular...
  - ... as it uses some given value ...
  - ... and in effect “looks up” the code that should run.
- If we have this structure, and the possible values to be looked up are known at compile time...
- ... then we can use a switch.

Attempt B (switch)

```c
void leds_on_B (uint8_t num) {
    switch (num) {
        case 6:
            PORTL = 0b10101010; PORTB = 0b00001010;
            break;
        case 5:
            PORTL = 0b10101010; PORTB = 0b00001000;
            break;
        case 4:
            PORTL = 0b10101010; PORTB = 0b00000000;
            break;
        case 3:
            PORTL = 0b10101000; PORTB = 0b00000000;
            break;
        case 2:
            PORTL = 0b10100000; PORTB = 0b00000000;
            break;
        case 1:
            PORTL = 0b10000000; PORTB = 0b00000000;
            break;
        default:
            PORTL = 0b00000000; PORTB = 0b00000000;
            break;
    }
}
```

The expression in the switch parentheses can be computed at run time...

... but each of the case labels must be known at compile time (i.e., cases labels cannot include variables or function calls).

Each of the cases has an explicit break statement (i.e., after code in the case is complete, break out of the switch).

Example: leds_on (Cont.)

- Switch statements highlight the importance of individual cases
  - New cases can be easily added.
  - Existing cases can be quickly found and modified.
- Important: the case value/label must be something that is known at compile time
  - If the value/label cannot be known, then we have to go back to a multi-way if.
- break statements are important in the code for each case...
  - ... although we can do fun things when carefully re-think our code...

Attempt C (very compact switch)

```
void leds_on_C (uint8_t num) {
    uint8_t bits_L = 0;
    uint8_t bits_B = 0;

    switch (num) {
        case 6: bits_B |= 0b00000010;
        case 5: bits_B |= 0b00001000;
        case 4: bits_L |= 0b00000010;
        case 3: bits_L |= 0b00001000;
        case 2: bits_L |= 0b00100000;
        case 1: bits_L |= 0b10000000;
        default: PORTL = bits_L; PORTB = bits_B;
    }
}
```

Notice how the absence of breaks causes the code to enter the switch and then fall through the remaining switch statements...

C and interrupts

- Many of the benefits of C can help us when implementing interrupt handlers
  - The handler code may be written in C
  - All of the interrupt dispatch and return behavior is available to us.
- There are one or two subtle “gotchas” however:
  - Finding the name of the interrupt
  - Dealing with program-scope variables

The LED counter

```
#define __DELAY_BACKWARD_COMPATIBLE__ 1
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define DELAY1 0.5
#define PRESCALE_DIV 1024
#define TOP1 ((int)(0.5 + (F_CPU/PRESCALE_DIV*DELAY1)))

uint8_t count = 0;
```

Idea: lower four bits of a global counter value (8-bit unsigned integer) are to be displayed on the LEDs.

Note that count is a program-scope variable (i.e., like a global).

```
ISR(TIMER1_COMPA_vect) {
    count++;
}
```

Given in parentheses is the particular interrupt for which we want a handler.

The ISR routine not only updates the interrupt vector, but also permits us to specify the handler code.

Notice that reti is not needed (as the compiler inserts that for us at the end of our handler's code.)

The LED counter (Cont.)

```
int main() {
    /* Set up registers for LED. */
    DDRB = 0xff;
    DDRL = 0xff;


    cli();



    /* Set up timer 1 */
    int top = TOP1;
    OCR1A  = top;
    TCCR1A  = 0;
    TCCR1B  = (1 << WGM12);
    TCCR1B |= (1 << CS12);
    TCCR1B |= (1 << CS10);
    TIMSK1 |= (1 << OCIE1A);

    /* Turn on global interrupts */
    sei();

    uint8_t bits;
```

```
        /* Main loop */
        for (;;) {
            bits = 0;
            bits |= ((count & 0x01) ? 0b10000000 : 0);
            bits |= ((count & 0x02) ? 0b00100000 : 0);
            bits |= ((count & 0x04) ? 0b00001000 : 0);
            bits |= ((count & 0x08) ? 0b00000010 : 0);
            PORTL = bits;
        }
    }
```

It's a good idea to turn off interrupts before configuring devices that will use them...

Timer 1 is configured into CTC mode. (TOP1 is described on the previous slide.)

Flip the switch...

Code for converting lower-four bits of counter into PORTL values (LED on == bit set at the position)

Something wrong!

“volatile”

- This is a keyword that can be applied to a variable at declaration time.
    - It gives the compiler advice.
    - The advice is: The variable may change at any time!
- The meaning of “any time”:
    - Hardware may change the value asychronously (i.e., via an interrupt handler)
- If the compiler follows the advice...
    - ... then it avoids certain optimizations, such as those keeping the variable’s value in registers as long as possible.
    - (Moving values from memory to registers and back takes extra cycles, and the compiler often wants to minimize this extra.)

The LED counter (revised)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define DELAY1 0.5
#define PRESCALE_DIV 1024
#define TOP1 ((int)(0.5 +
(F_CPU/PRESCALE_DIV*DELAY1)))

volatile uint8_t count = 0;
```

The LED counter (revised)

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define DELAY1 0.5
#define PRESCALE_DIV 1024
#define TOP1 ((int)(0.5 + (F_CPU/PRESCALE_DIV*DELAY1)))

volatile uint8_t count = 0;
```

This program-scope variable is now interrupt-handler friendly.

```
ISR(TIMER1_COMPA_vect) {
    count++;
}
```

Any Questions?