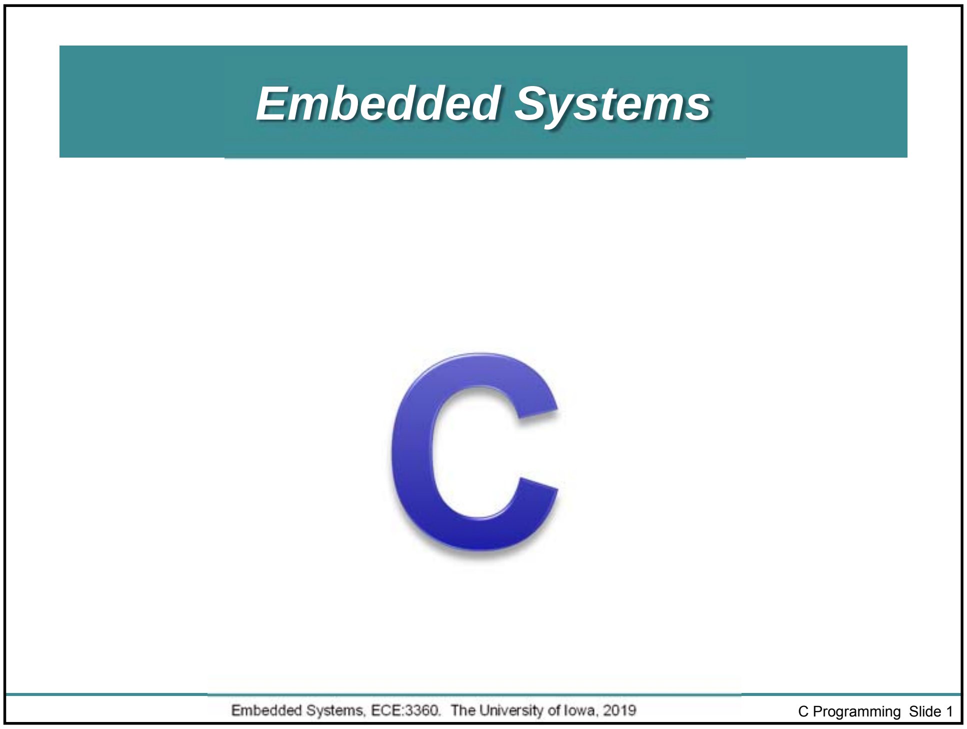
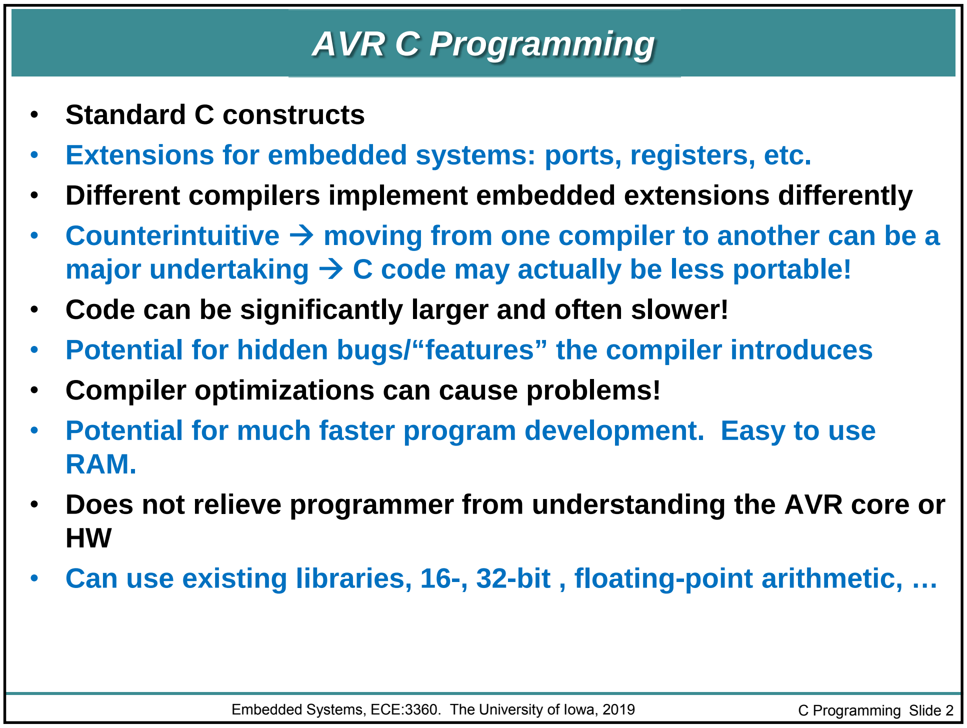


Embedded Systems

C



AVR C Programming

- Standard C constructs
- Extensions for embedded systems: ports, registers, etc.
- Different compilers implement embedded extensions differently
- Counterintuitive → moving from one compiler to another can be a major undertaking → C code may actually be less portable!
- Code can be significantly larger and often slower!
- Potential for hidden bugs/“features” the compiler introduces
- Compiler optimizations can cause problems!
- Potential for much faster program development. Easy to use RAM.
- Does not relieve programmer from understanding the AVR core or HW
- Can use existing libraries, 16-, 32-bit , floating-point arithmetic, …

AVR C Programming

- We will use the WinAVR/GCC compiler and libraries
- Windows port of GNU gcc compiler
- GNU copyright/copyleft

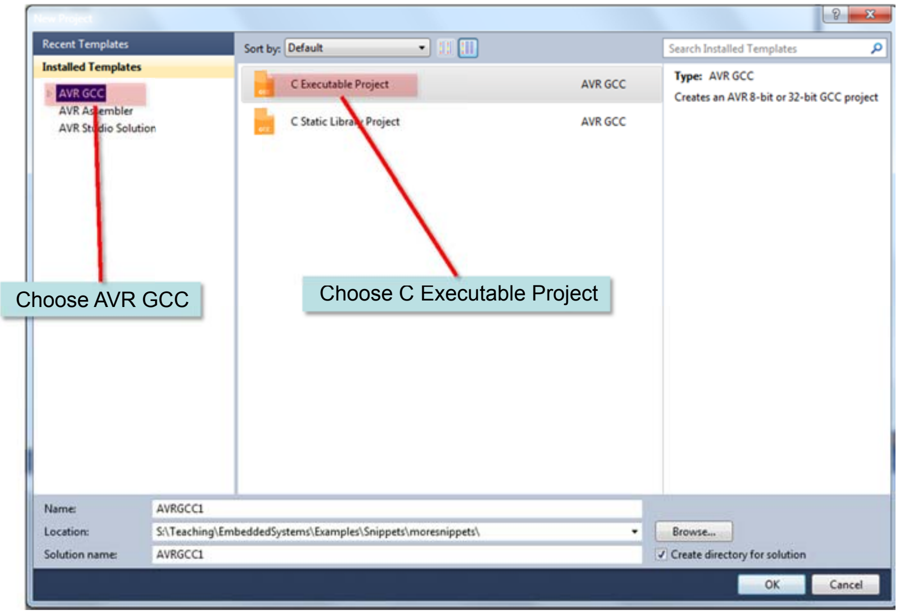
Compiler documentation:
https://www.microchip.com/webdoc/AVRLibcReferenceManual

Compiler documentation: http://www.nongnu.org/avr-libc/user-manual/index.html

WinAVR home: http://winavr.sourceforge.net

Other resources: AVRFreaks: http://www.avrfreaks.net

C Project in AVR Studio



Type: AVR GCC

Creates an AVR 8-bit or 32-bit GCC project

Name: AVRGCC1

Location: S:\Teaching\EmbeddedSystems\Examples\Snippets\moresnippets\

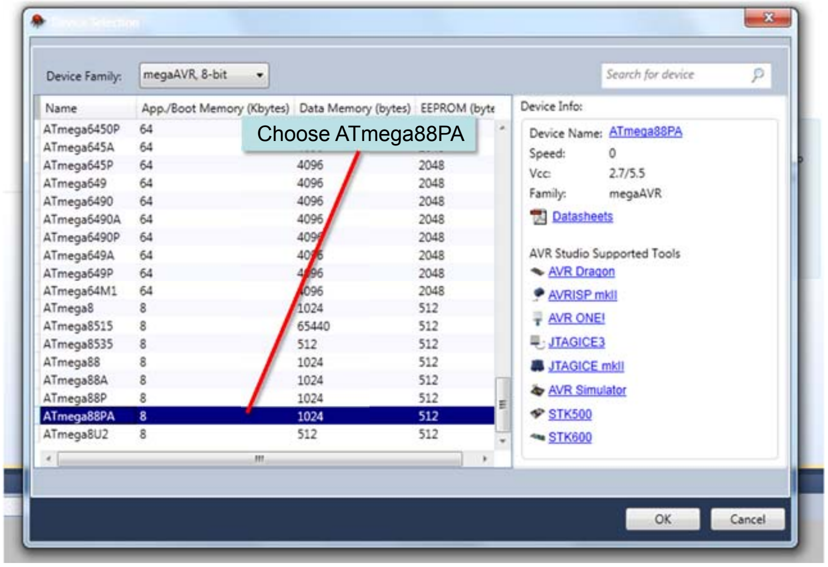
Solution name: AVRGCC1

☑ Create directory for solution

Choose AVR GCC

Choose C Executable Project

C Project in AVR Studio



Device Family: megaAVR, 8-bit

Name	App./Boot Memory (Kbytes)	Data Memory (bytes)	EEPROM (byte
ATmega6450P	64	[illegible]	[illegible]
ATmega645A	64	[illegible]	[illegible]
ATmega645P	64	4096	2048
ATmega649	64	4096	2048
ATmega6490	64	4096	2048
ATmega6490A	64	4096	2048
ATmega6490P	64	4096	2048
ATmega649A	64	4096	2048
ATmega649P	64	4096	2048
ATmega64M1	64	4096	2048
ATmega8	8	1024	512
ATmega8515	8	65440	512
ATmega8535	8	512	512
ATmega88	8	1024	512
ATmega88A	8	1024	512
ATmega88P	8	1024	512
ATmega88PA	8	1024	512
ATmega8U2	8	512	512

Device Info:

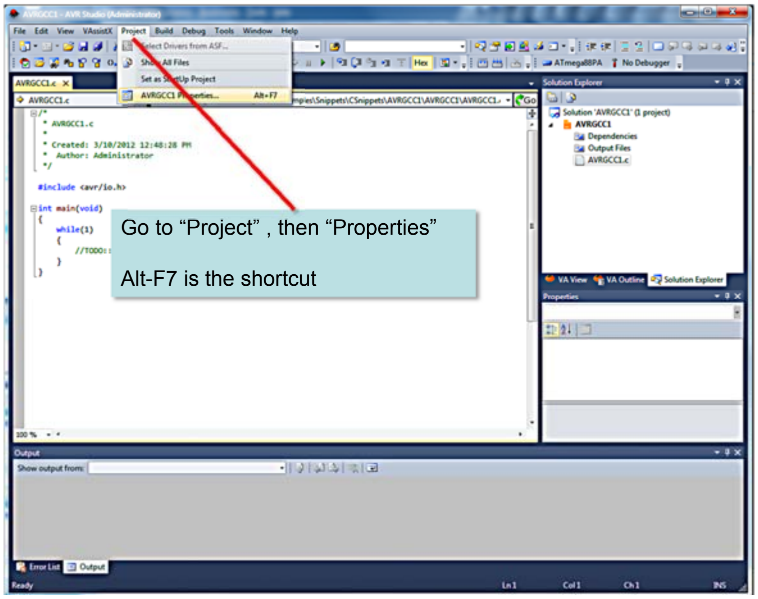
Device Name: ATmega88PA
Speed: 0
Vcc: 2.7/5.5
Family: megaAVR

Datasheets

AVR Studio Supported Tools
- AVR Dragon
- AVRISP mkII
- AVR ONE!
- JTAGICE3
- JTAGICE mkII
- AVR Simulator
- STK500
- STK600

Choose ATmega88PA

C Project in AVR Studio

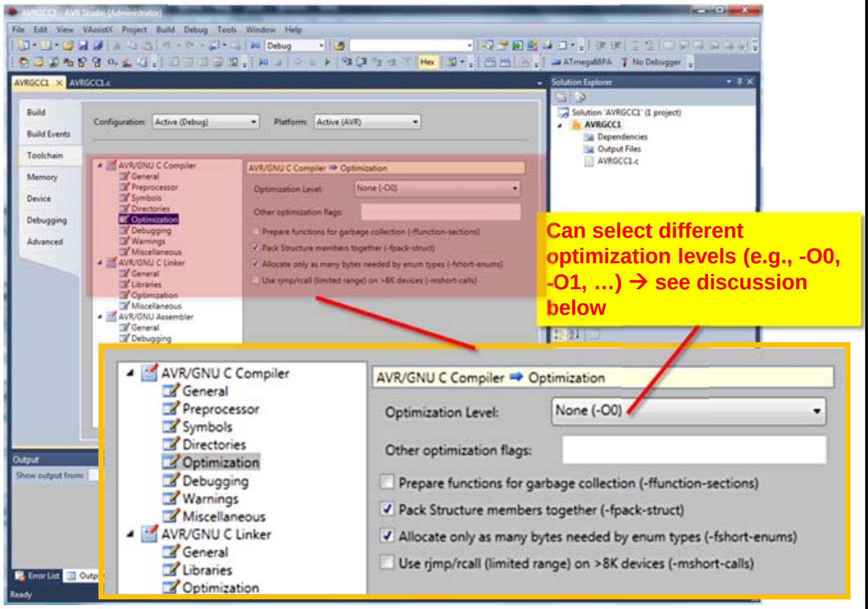


```
/*
 * AVRGCC1.c
 *
 * Created: 3/10/2012 12:48:28 PM
 *  Author: Administrator
 */

#include <avr/io.h>

int main(void)
{
    while(1)
    {
        //TODO::
    }
}
```

Go to “Project” , then “Properties”

Alt-F7 is the shortcut

C Project in AVR Studio



AVR/GNU C Compiler → Optimization

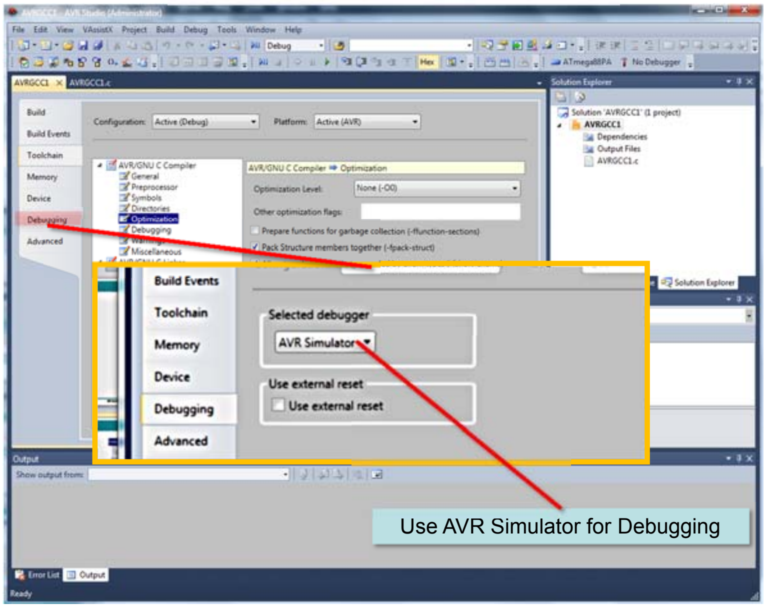
Optimization Level: None (-O0)

Other optimization flags:

- ☐ Prepare functions for garbage collection (-ffunction-sections)
- ☑ Pack Structure members together (-fpack-struct)
- ☑ Allocate only as many bytes needed by enum types (-fshort-enums)
- ☐ Use rjmp/rcall (limited range) on >8K devices (-mshort-calls)

Can select different optimization levels (e.g., -O0, -O1, …) → see discussion below

C Project in AVR Studio



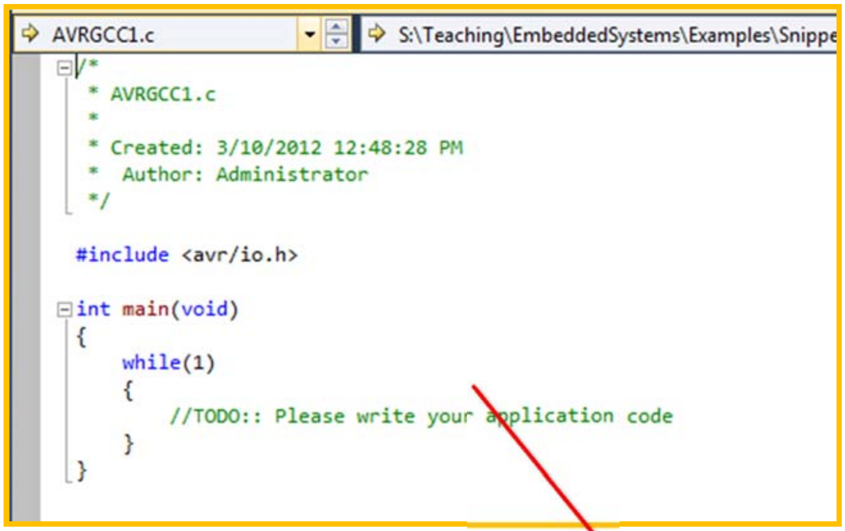
Selected debugger: AVR Simulator

Use external reset

- ☐ Use external reset

Use AVR Simulator for Debugging

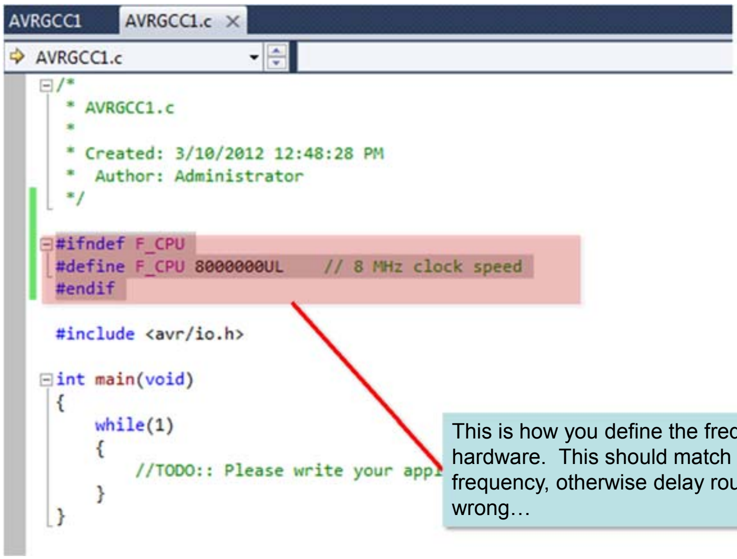
C Project in AVR Studio



```
/*
 * AVRGCC1.c
 *
 * Created: 3/10/2012 12:48:28 PM
 *  Author: Administrator
 */

#include <avr/io.h>

int main(void)
{
    while(1)
    {
        //TODO:: Please write your application code
    }
}
```

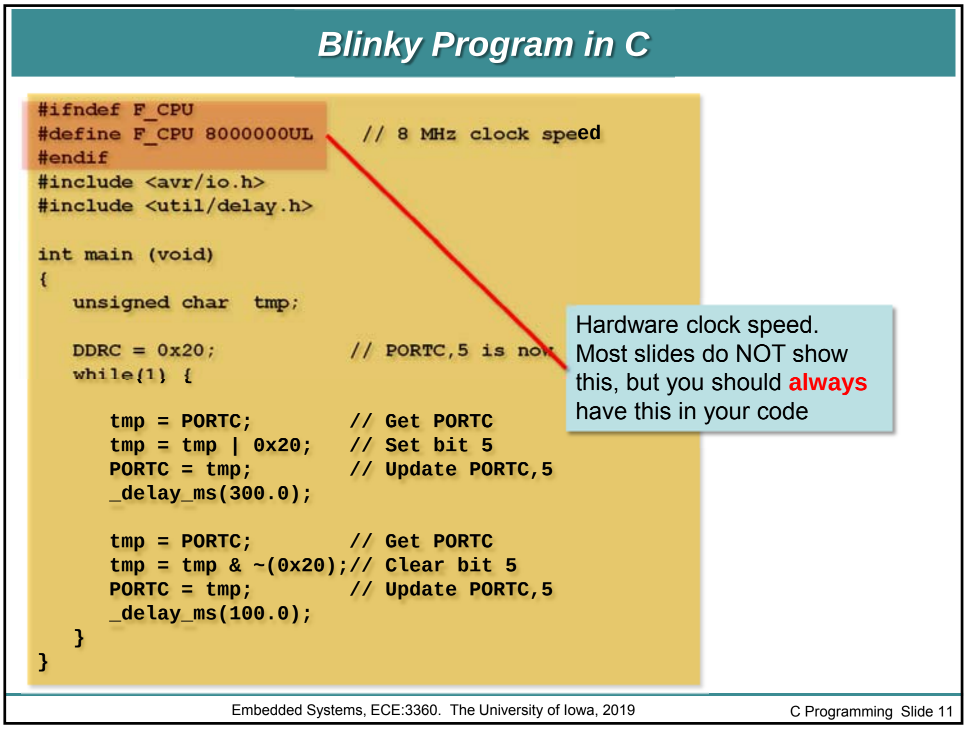
AVR Studio creates a skeleton program

C Project in AVR Studio



```
/*
 * AVRGCC1.c
 *
 * Created: 3/10/2012 12:48:28 PM
 *  Author: Administrator
 */

#ifndef F_CPU
#define F_CPU 8000000UL     // 8 MHz clock speed
#endif

#include <avr/io.h>

int main(void)
{
    while(1)
    {
        //TODO:: Please write your appl
    }
}
```

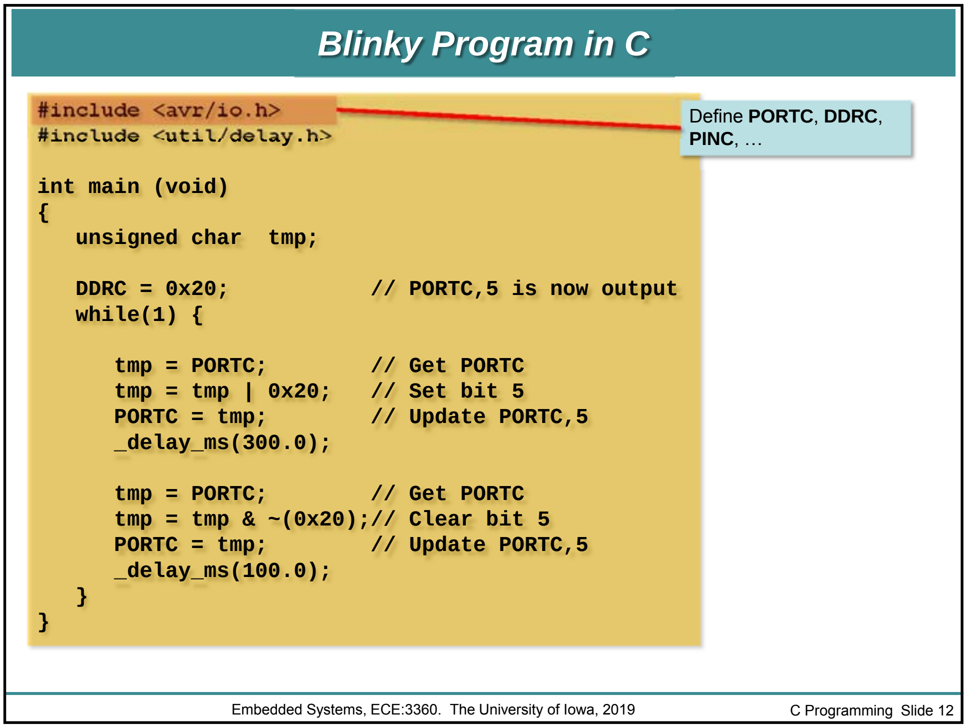
This is how you define the frequency of your hardware. This should match the actual clock frequency, otherwise delay routines will be wrong…



Blinky Program in C

```
#ifndef F_CPU
#define F_CPU 8000000UL     // 8 MHz clock speed
#endif
#include <avr/io.h>
#include <util/delay.h>

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;             // PORTC,5 is now
   while(1) {

      tmp = PORTC;          // Get PORTC
      tmp = tmp | 0x20;     // Set bit 5
      PORTC = tmp;          // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;          // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;          // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

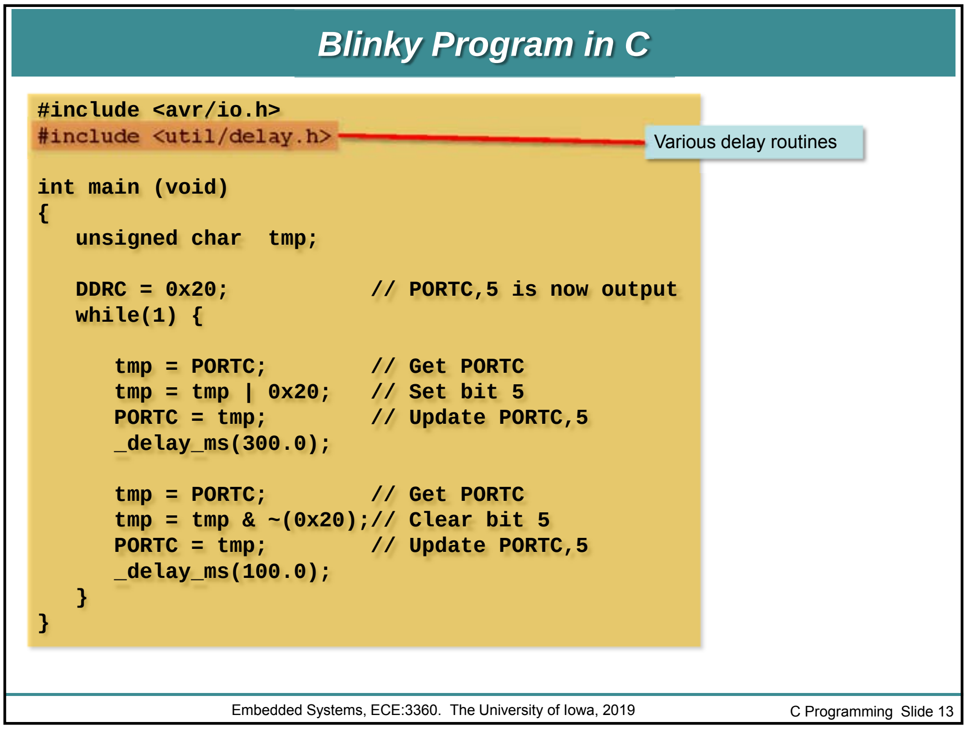
Hardware clock speed. Most slides do NOT show this, but you should **always** have this in your code



Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;             // PORTC,5 is now output
   while(1) {

      tmp = PORTC;          // Get PORTC
      tmp = tmp | 0x20;     // Set bit 5
      PORTC = tmp;          // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;          // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;          // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

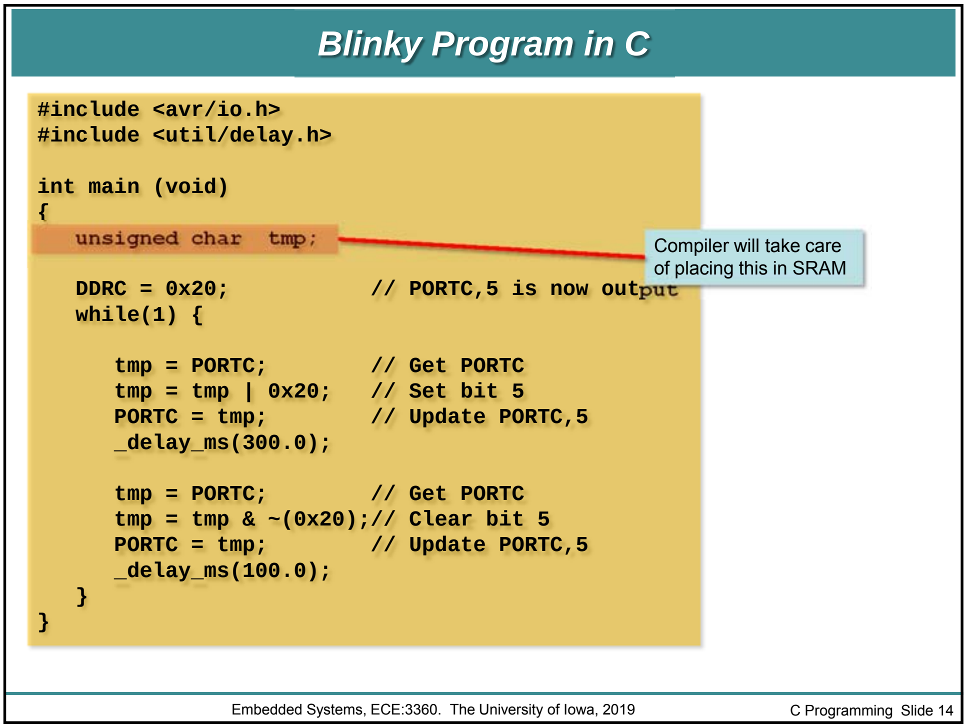
Define **PORTC**, **DDRC**, **PINC**, …



Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;           // PORTC,5 is now output
   while(1) {

      tmp = PORTC;        // Get PORTC
      tmp = tmp | 0x20;   // Set bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;        // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

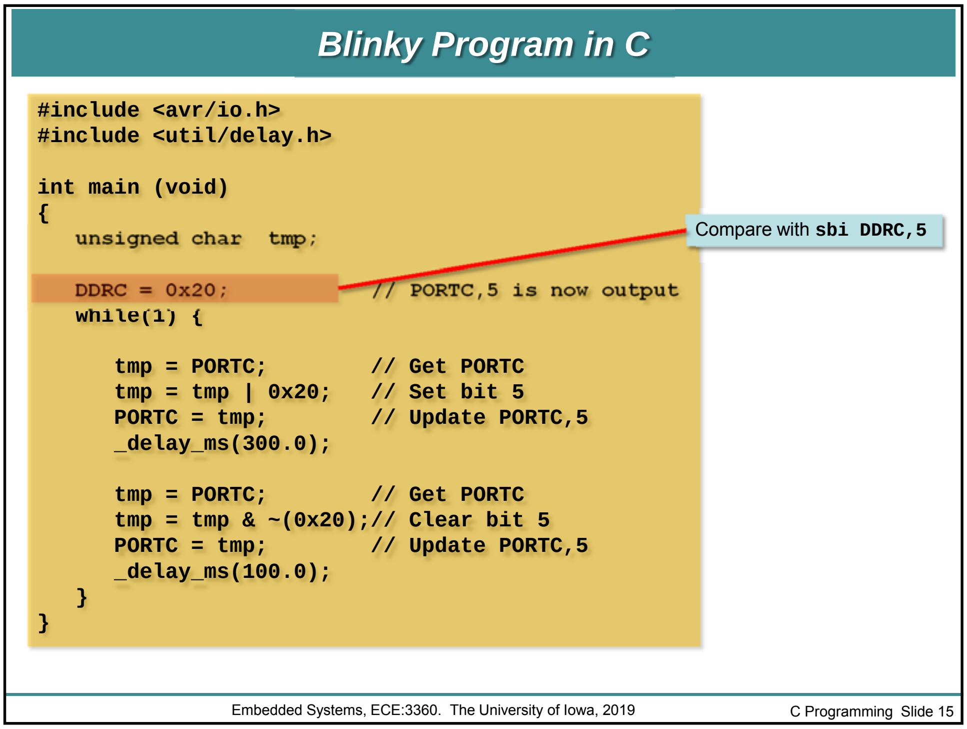
Various delay routines



Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;           // PORTC,5 is now output
   while(1) {

      tmp = PORTC;        // Get PORTC
      tmp = tmp | 0x20;   // Set bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;        // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

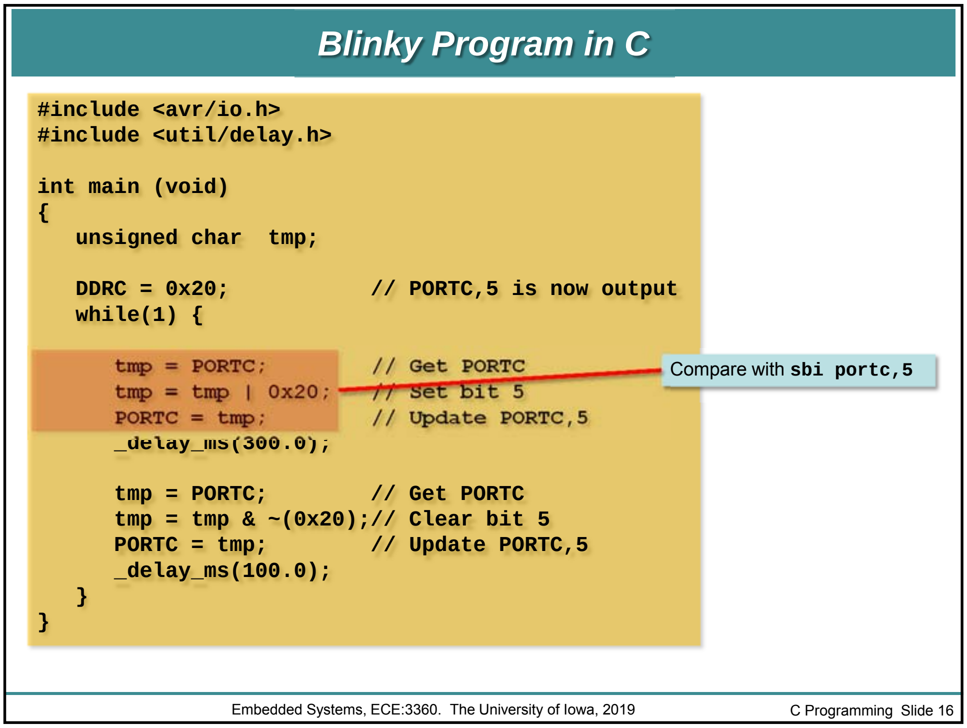
Compiler will take care of placing this in SRAM



Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;             // PORTC,5 is now output
   while(1) {

      tmp = PORTC;         // Get PORTC
      tmp = tmp | 0x20;    // Set bit 5
      PORTC = tmp;         // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;         // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;         // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

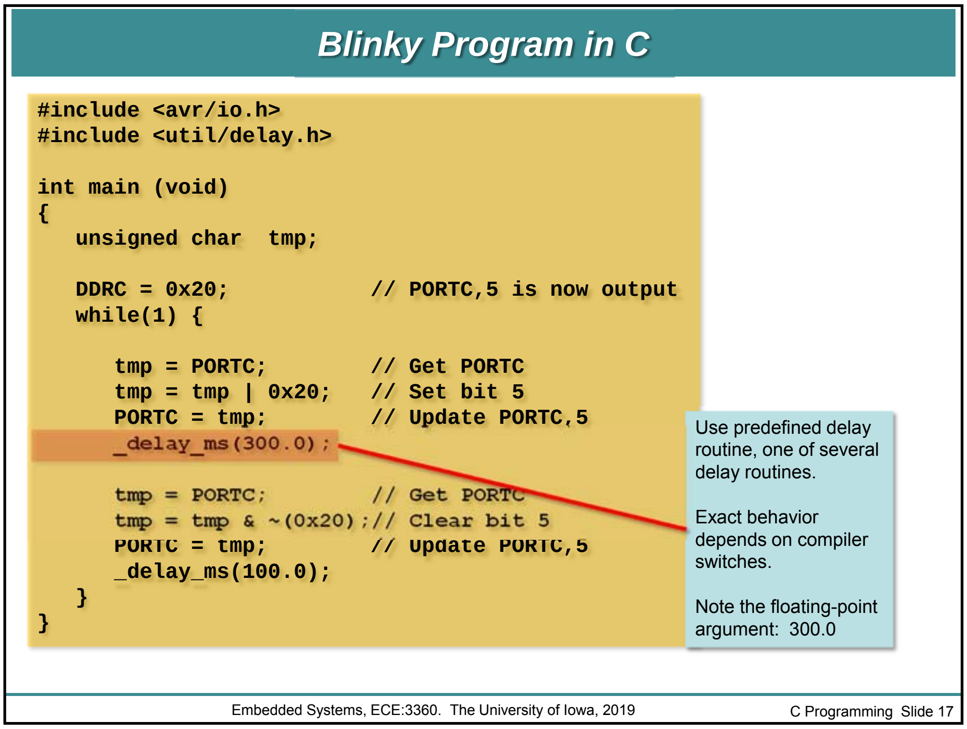
Compare with **`sbi DDRC,5`**



Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;             // PORTC,5 is now output
   while(1) {

      tmp = PORTC;         // Get PORTC
      tmp = tmp | 0x20;    // Set bit 5
      PORTC = tmp;         // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;         // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;         // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

Compare with **`sbi portc,5`**



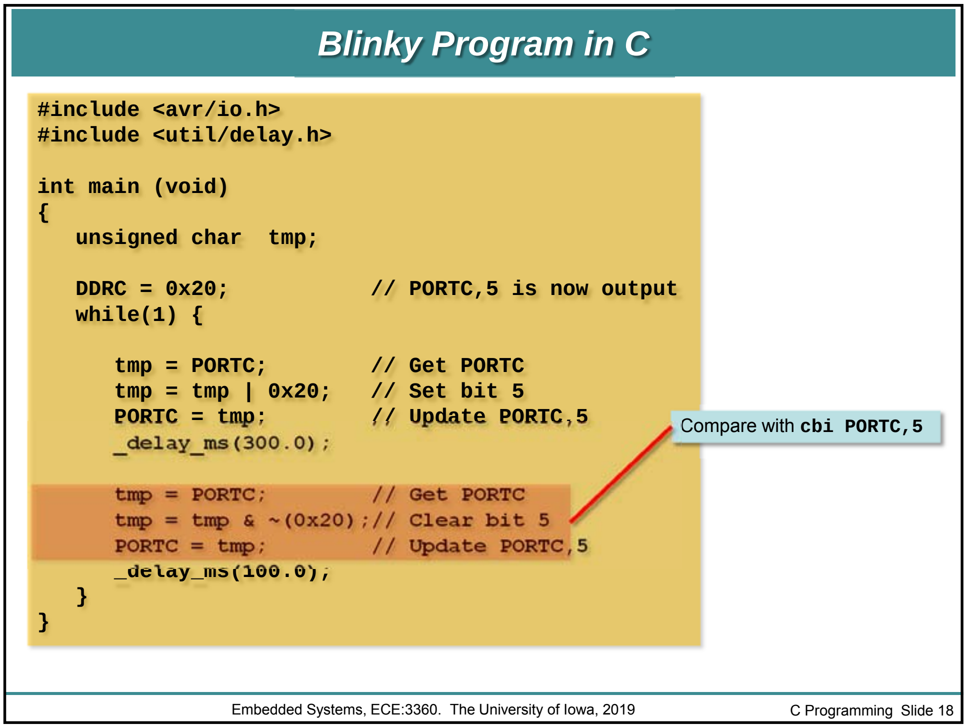
Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;           // PORTC,5 is now output
   while(1) {

      tmp = PORTC;        // Get PORTC
      tmp = tmp | 0x20;   // Set bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;        // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

Use predefined delay routine, one of several delay routines.

Exact behavior depends on compiler switches.

Note the floating-point argument: 300.0

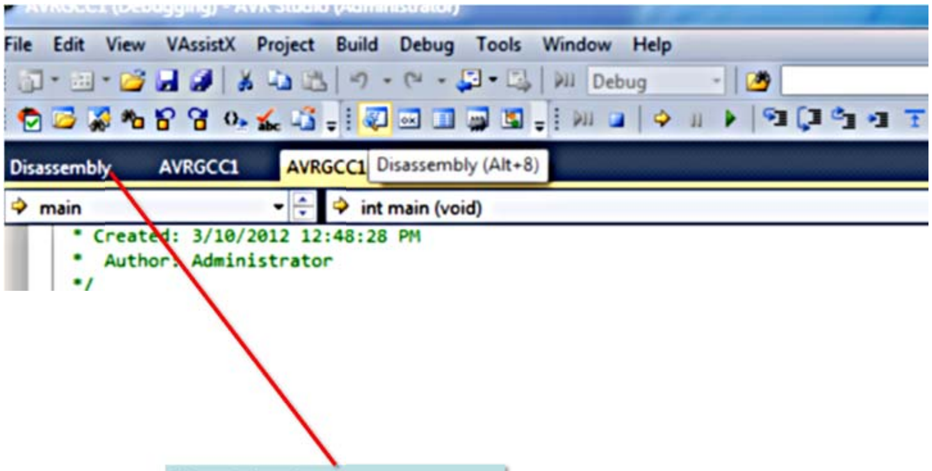


Blinky Program in C

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;           // PORTC,5 is now output
   while(1) {

      tmp = PORTC;        // Get PORTC
      tmp = tmp | 0x20;   // Set bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;        // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

Compare with `cbi PORTC,5`

Disassembly


AVRGCC1 (Debugging) - AVR Studio (Administrator)
File Edit View VAssistX Project Build Debug Tools Window Help
Debug
Disassembly
AVRGCC1
AVRGCC1
Disassembly (Alt+8)
main
int main (void)

```
 * Created: 3/10/2012 12:48:28 PM
 *  Author: Administrator
 */
```

When debugging, click here to see the assembly language statements that are generated for the C statements

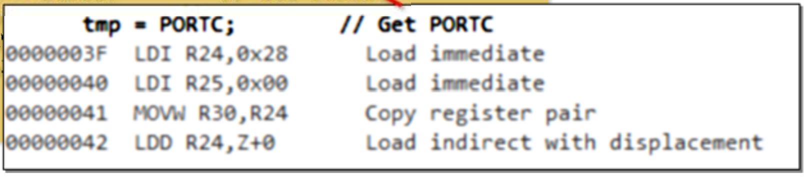
Disassembly - Example

```
--- No source file -------------------------------------------------------------
00000000  RJMP PC+0x001A                                      Relative jump
00000001  RJMP PC+0x0021                                      Relative jump
00000002  RJMP PC+0x0020                                      Relative jump
00000003  RJMP PC+0x001F                                      Relative jump
00000004  RJMP PC+0x001E                                      Relative jump
00000005  RJMP PC+0x001D                                      Relative jump
00000006  RJMP PC+0x001C                                      Relative jump
00000007  RJMP PC+0x001B                                      Relative jump
00000008  RJMP PC+0x001A                                      Relative jump
00000009  RJMP PC+0x0019                                      Relative jump
0000000A  RJMP PC+0x0018                                      Relative jump
0000000B  RJMP PC+0x0017                                      Relative jump
0000000C  RJMP PC+0x0016                                      Relative jump
0000000D  RJMP PC+0x0015                                      Relative jump
0000000E  RJMP PC+0x0014                                      Relative jump
0000000F  RJMP PC+0x0013                                      Relative jump
00000010  RJMP PC+0x0012                                      Relative jump
00000011  RJMP PC+0x0011                                      Relative jump
00000012  RJMP PC+0x0010                                      Relative jump
00000013  RJMP PC+0x000F                                      Relative jump
00000014  RJMP PC+0x000E                                      Relative jump
00000015  RJMP PC+0x000D                                      Relative jump
00000016  RJMP PC+0x000C                                      Relative jump
00000017  RJMP PC+0x000B                                      Relative jump
00000018  RJMP PC+0x000A                                      Relative jump
00000019  RJMP PC+0x0009                                      Relative jump
--- ../../../../crt1/gcrt1.S ---------------------------------------------------
0000001A  CLR R1                                              Clear Register
0000001B  OUT 0x3F,R1                                         Out to I/O location
0000001C  SER R28                                             Set Register
0000001D  LDI R29,0x04                                        Load immediate
--- ../../../../crt1/gcrt1.S ---------------------------------------------------
0000001E  OUT 0x3E,R29                                        Out to I/O location
0000001F  OUT 0x3D,R28                                        Out to I/O location
00000020  RCALL PC+0x0003                                     Relative call subroutine
00000021  RJMP PC+0x0031                                      Relative jump
00000022  RJMP PC-0x0022                                      Relative jump
--- Z:\es_s17\labs\rb_gcc1\rb_gcc1\Debug/.././main.c ---------------------------
{
00000023  PUSH R28                                            Push register on stack
00000024  PUSH R29                                            Push register on stack
00000025  RCALL PC+0x0001                                     Relative call subroutine
00000026  PUSH R1                                             Push register on stack
00000027  IN R28,0x3D                                         In from I/O location
00000028  IN R29,0x3E                                         In from I/O location
                        int i = 0;
00000029  STD Y+2,R1                                          Store indirect with displacement
0000002A  STD Y+1,R1                                          Store indirect with displacement
                        DDRC = (1<<5);
0000002B  LDI R24,0x27                                        Load immediate
0000002C  LDI R25,0x00                                        Load immediate
0000002D  LDI R18,0x20                                        Load immediate
0000002E  MOVW R30,R24                                        Copy register pair
0000002F  STD Z+0,R18                                         Store indirect with displacement
                        PORTC = 0x20;
00000030  LDI R24,0x28                                        Load immediate
00000031  LDI R25,0x00                                        Load immediate
00000032  LDI R18,0x20                                        Load immediate
00000033  MOVW R30,R24                                        Copy register pair
00000034  STD Z+0,R18                                         Store indirect with displacement
                        for(i=1;i<=10;i=i+1);
00000035  LDI R24,0x01                                        Load immediate
00000036  LDI R25,0x00                                        Load immediate
00000037  STD Y+2,R25                                         Store indirect with displacement
00000038  STD Y+1,R24                                         Store indirect with displacement
00000039  RJMP PC+0x0006                                      Relative jump
--- No source file -------------------------------------------------------------
0000003A  LDD R24,Y+1                                         Load indirect with displacement
0000003B  LDD R25,Y+2                                         Load indirect with displacement
--- No source file -------------------------------------------------------------
0000003C  ADIW R24,0x01                                       Add immediate to word
0000003D  STD Y+2,R25                                         Store indirect with displacement
0000003E  STD Y+1,R24                                         Store indirect with displacement
0000003F  LDD R24,Y+1                                         Load indirect with displacement
00000040  LDD R25,Y+2                                         Load indirect with displacement
00000041  SBIW R24,0x0B                                       Subtract immediate from word
00000042  BRLT PC-0x08                                        Branch if less than, signed
--- Z:\es_s17\labs\rb_gcc1\rb_gcc1\Debug/.././main.c ---------------------------
                        PORTC = 0x00;
00000043  LDI R24,0x28                                        Load immediate
00000044  LDI R25,0x00                                        Load immediate
00000045  MOVW R30,R24                                        Copy register pair
00000046  STD Z+0,R1                                          Store indirect with displacement
00000047  LDI R24,0x0A                                        Load immediate
00000048  STD Y+3,R24                                         Store indirect with displacement
--- c:\program files (x86)\atmel\studio\7.0\toolchain\avr8\avr8-gnu-toolchain\avr\include\util/delay_basic.h
                        __asm__ volatile (
00000049  LDD R24,Y+3                                         Load indirect with displacement
0000004A  DEC R24                                             Decrement
0000004B  BRNE PC-0x01                                        Branch if not equal
0000004C  STD Y+3,R24                                         Store indirect with displacement
--- Z:\es_s17\labs\rb_gcc1\rb_gcc1\Debug/.././main.c ---------------------------
                        PORTC = 0x00;
0000004D  LDI R24,0x28                                        Load immediate
0000004E  LDI R25,0x00                                        Load immediate
0000004F  MOVW R30,R24                                        Copy register pair
00000050  STD Z+0,R1                                          Store indirect with displacement
  }
00000051  RJMP PC-0x0021                                      Relative jump
--- No source file -------------------------------------------------------------
00000052  CLI                                                 Global Interrupt Disable
00000053  RJMP PC-0x0000                                      Relative jump
00000054  NOP                                                 Undefined
```

Disassembly – C program compiled with “-O0” Option

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;            // PORTC,5 is now output
   while(1) {

      tmp = PORTC;         // Get PORTC
      tmp = tmp | 0x20;    // Set bit 5
      PORTC = tmp;         // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;         // Get PORTC
      tmp
      POR
      _de
   }
}
```

AVR Studio allows one to the corresponding assembly-langue instructions in the so-called **disassembly window.**



```
          tmp = PORTC;           // Get PORTC
0000003F  LDI R24,0x28             Load immediate
00000040  LDI R25,0x00             Load immediate
00000041  MOVW R30,R24             Copy register pair
00000042  LDD R24,Z+0              Load indirect with displacement
```

Disassembly – C program compiled with “-O0” Option

```
#i
#i

in
{
   while(1) {

                           // Get PORTC
      tmp = tmp | 0x20;    // Set bit 5
      PORTC = tmp;         // Update PORT
      _delay_ms(300.0);

      tmp = PORTC;         // Get PORTC
      tmp
      POR
      _de
   }
}
```

When hand-coding, it can be confusing when to **IN/OUT**, when to **LDS/STS**, etc. This is because, even though the registers, ports, etc. are all in SRAM, some instructions can’t reach all of SRAM.

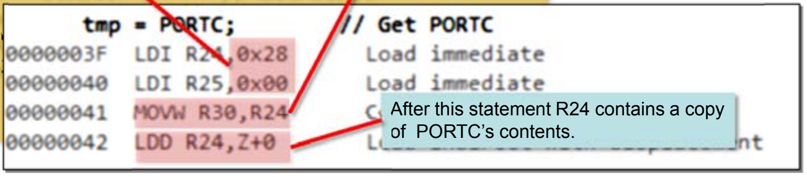
However, one can access any SRAM location with **LDS, STS** and the index registers. **PORTC** on the ATmega88, for example, has an equivalent SRAM address 0x28. This is the approach the GCC compiler takes. Rather than using **IN PORTC** instruction, it access **PORTC** via the SRAM.

SRAM address of **PORTC**

This statement has the effect of

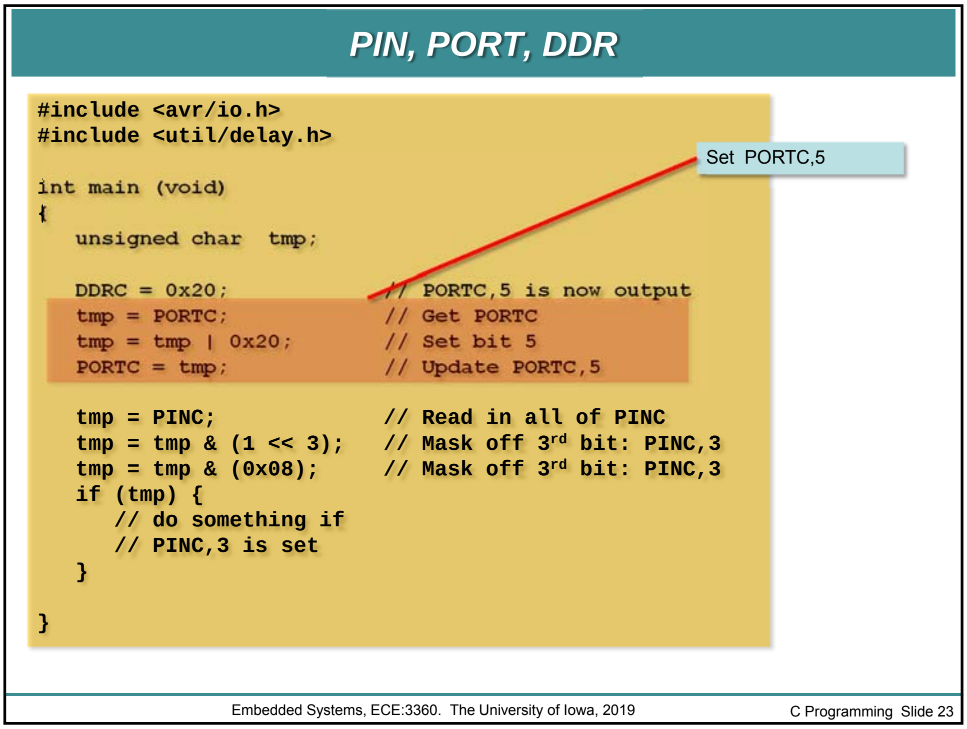
R30:R31 ← R24:R25

Note **R30:R31** is the **Z** register, so that after this statement the **Z** register points to **PORTC**’s SRAM address.



```
          tmp = PORTC;           // Get PORTC
0000003F  LDI R24,0x28             Load immediate
00000040  LDI R25,0x00             Load immediate
00000041  MOVW R30,R24             C
00000042  LDD R24,Z+0              L                           nt
```

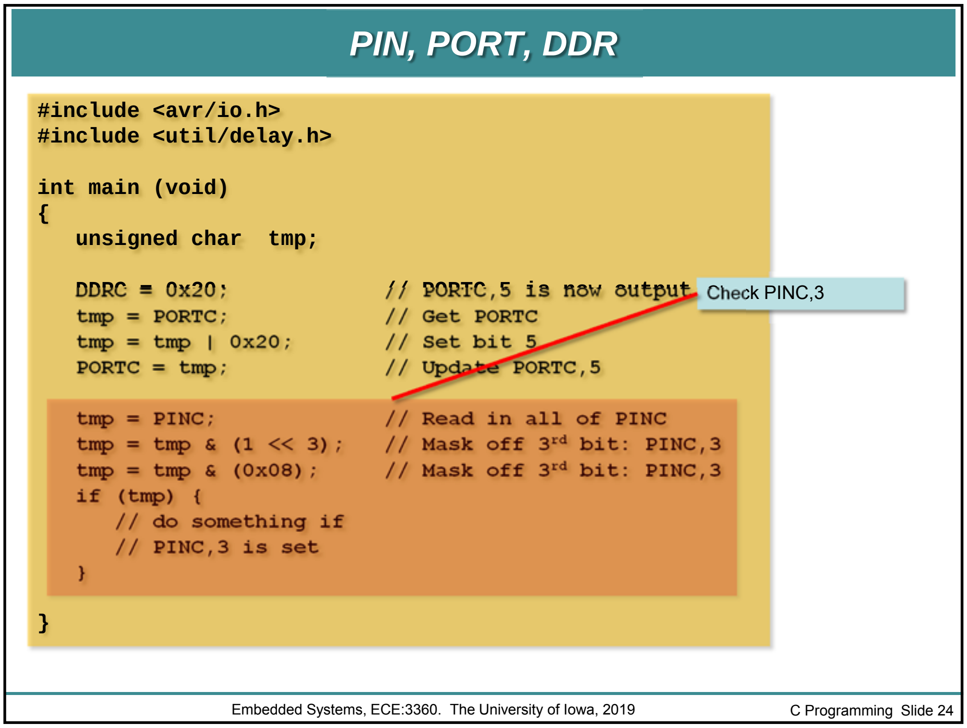
After this statement R24 contains a copy of PORTC’s contents.



PIN, PORT, DDR

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;                // PORTC,5 is now output
   tmp = PORTC;                // Get PORTC
   tmp = tmp | 0x20;           // Set bit 5
   PORTC = tmp;                // Update PORTC,5

   tmp = PINC;                 // Read in all of PINC
   tmp = tmp & (1 << 3);       // Mask off 3rd bit: PINC,3
   tmp = tmp & (0x08);         // Mask off 3rd bit: PINC,3
   if (tmp) {
      // do something if
      // PINC,3 is set
   }

}
```

Set PORTC,5

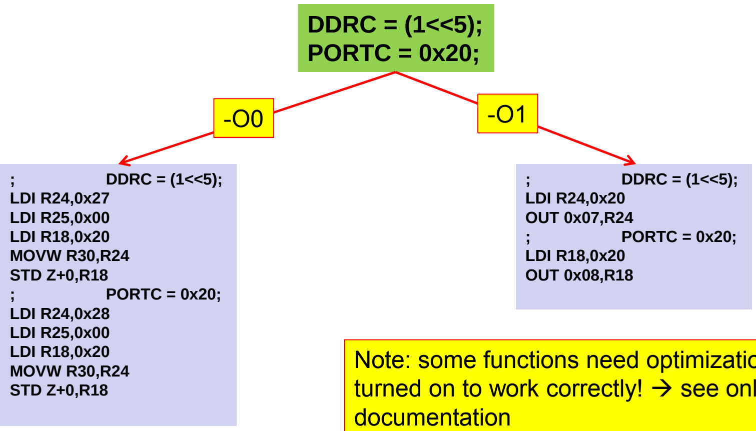


PIN, PORT, DDR

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

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;                // PORTC,5 is now output
   tmp = PORTC;                // Get PORTC
   tmp = tmp | 0x20;           // Set bit 5
   PORTC = tmp;                // Update PORTC,5

   tmp = PINC;                 // Read in all of PINC
   tmp = tmp & (1 << 3);       // Mask off 3rd bit: PINC,3
   tmp = tmp & (0x08);         // Mask off 3rd bit: PINC,3
   if (tmp) {
      // do something if
      // PINC,3 is set
   }

}
```

Check PINC,3

Impact of C Compiler Optimization



```
DDRC = (1<<5);
PORTC = 0x20;
```

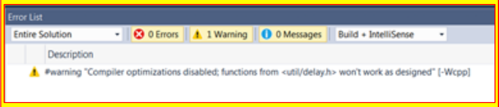
-O0

```
;                DDRC = (1<<5);
LDI R24,0x27
LDI R25,0x00
LDI R18,0x20
MOVW R30,R24
STD Z+0,R18
;                PORTC = 0x20;
LDI R24,0x28
LDI R25,0x00
LDI R18,0x20
MOVW R30,R24
STD Z+0,R18
```

-O1

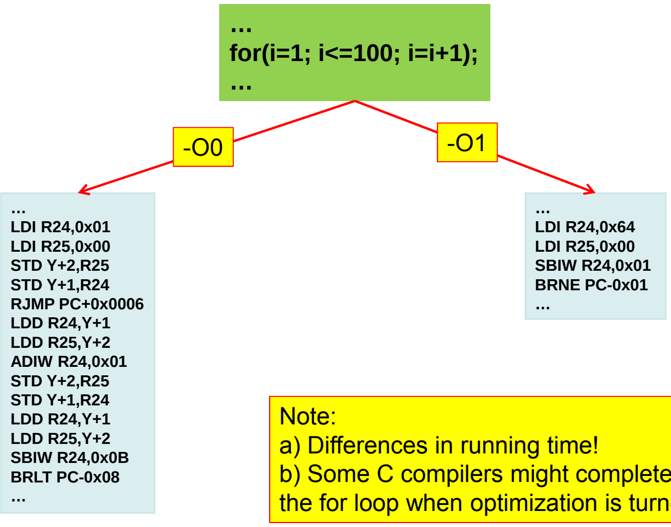
```
;                DDRC = (1<<5);
LDI R24,0x20
OUT 0x07,R24
;                PORTC = 0x20;
LDI R18,0x20
OUT 0x08,R18
```

Note: some functions need optimization turned on to work correctly! → see online documentation



```
#warning "Compiler optimizations disabled; functions from <util/delay.h> won't work as designed" [-Wcpp]
```

Impact of C Compiler Optimization



```
…
for(i=1; i<=100; i=i+1);
…
```

-O0

```
…
LDI R24,0x01
LDI R25,0x00
STD Y+2,R25
STD Y+1,R24
RJMP PC+0x0006
LDD R24,Y+1
LDD R25,Y+2
ADIW R24,0x01
STD Y+2,R25
STD Y+1,R24
LDD R24,Y+1
LDD R25,Y+2
SBIW R24,0x0B
BRLT PC-0x08
…
```

-O1

```
…
LDI R24,0x64
LDI R25,0x00
SBIW R24,0x01
BRNE PC-0x01
…
```

Note:
a) Differences in running time!
b) Some C compilers might completely ignore the for loop when optimization is turned on!

Delays

There are several methods for creating delays with using WinAVR/gcc

1) Ad-hoc cycle wasting. Write your own delay loop in C.

2) Simple delay loops that perform a busy-waiting. Does not use interrupts, should be used for short delays only.

`void _delay_loop_1(uint8_t count)`	3 CPU cycles per count (0-255) excluding overhead
`void _delay_loop_2 (uint16_t count)`	4 CPU cycles per count (0-65535) excluding overhead

3) Wrappers around `_delay_loop_1()` and `_delay_loop_2()`. Needs the clock frequency, set by for example: `#define F_CPU 8000000UL` or the `-DF_CPU=8000000UL` compiler switch. Note: compiler switches affect behavior!

`void _delay_ms(double ms)`	Delays ms, uses `_delay_loop_2()`
`void _delay_us(double us)`	Delays µs, uses `_delay_loop_1()`

4) Hardware timers. Program the hardware timers as you would in assembly language

Delays

Question. Estimate the approximate delay; the following statements will affect with WinAVR, ATmega88PA, 8 MHz clock, no clock division using CLKDIV fuse, and `-DF_CPU=8000000UL` compiler switch:

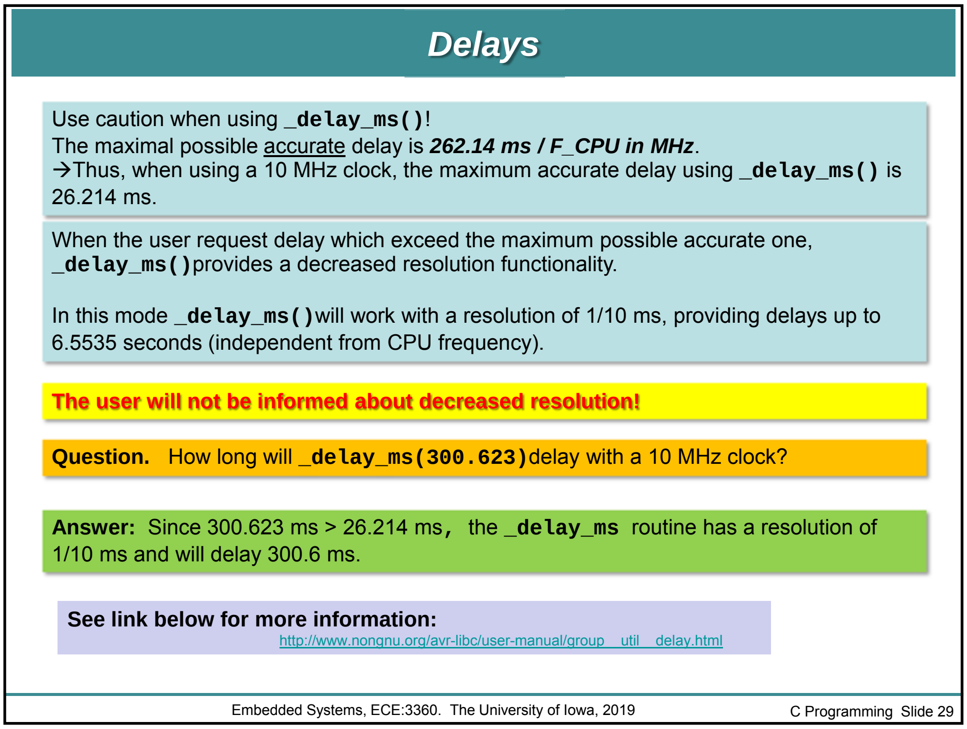
```
#include <util/delay.h>
main(){

   unsigned char counter;

    counter = 0;
    while (counter !=10) {
       _delay_loop_2(30000);
       counter++;
    }
}
```

Answer. The function `_delay_loop_2` takes 4 cycles,

so that `_delay_loop(30000)` takes 120,000 cycles.

The system clock is 8 MHz, so 120,000 cycles will take 15 ms.

The **`while`** loop executes 0,1,..9, => total of 10 times, so the total delay is **150 ms**. The other statements (branch, increment, …) make the actual delay slightly longer.



Delays

Use caution when using `_delay_ms()`!
The maximal possible accurate delay is ***262.14 ms / F_CPU in MHz***.
→Thus, when using a 10 MHz clock, the maximum accurate delay using `_delay_ms()` is 26.214 ms.

When the user request delay which exceed the maximum possible accurate one, `_delay_ms()`provides a decreased resolution functionality.

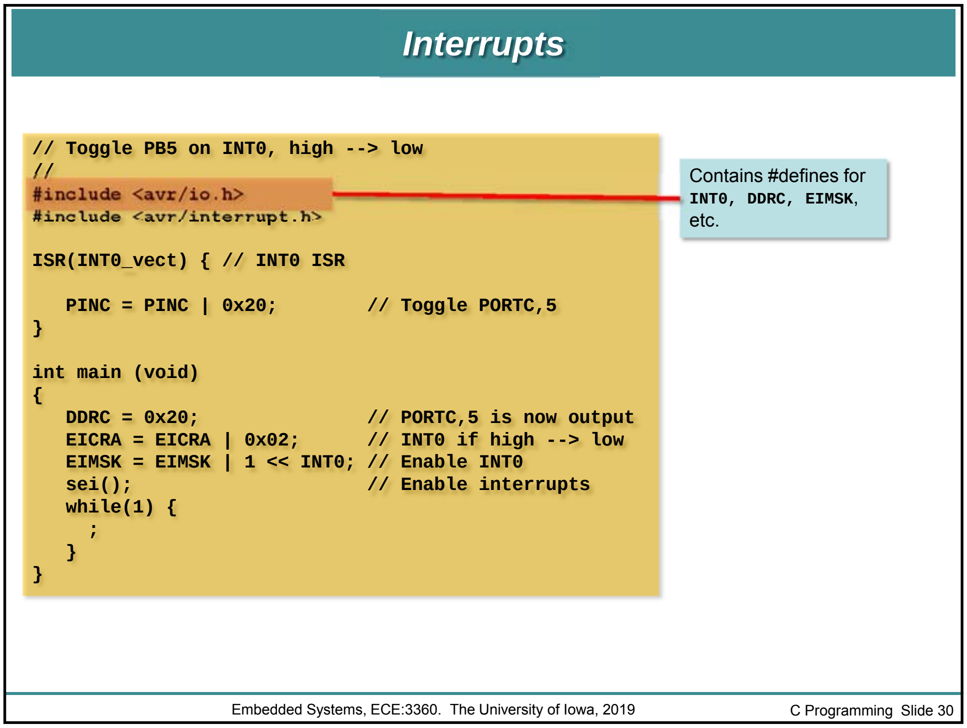
In this mode `_delay_ms()`will work with a resolution of 1/10 ms, providing delays up to 6.5535 seconds (independent from CPU frequency).

The user will not be informed about decreased resolution!

Question. How long will `_delay_ms(300.623)`delay with a 10 MHz clock?

Answer: Since 300.623 ms > 26.214 ms, the `_delay_ms` routine has a resolution of 1/10 ms and will delay 300.6 ms.

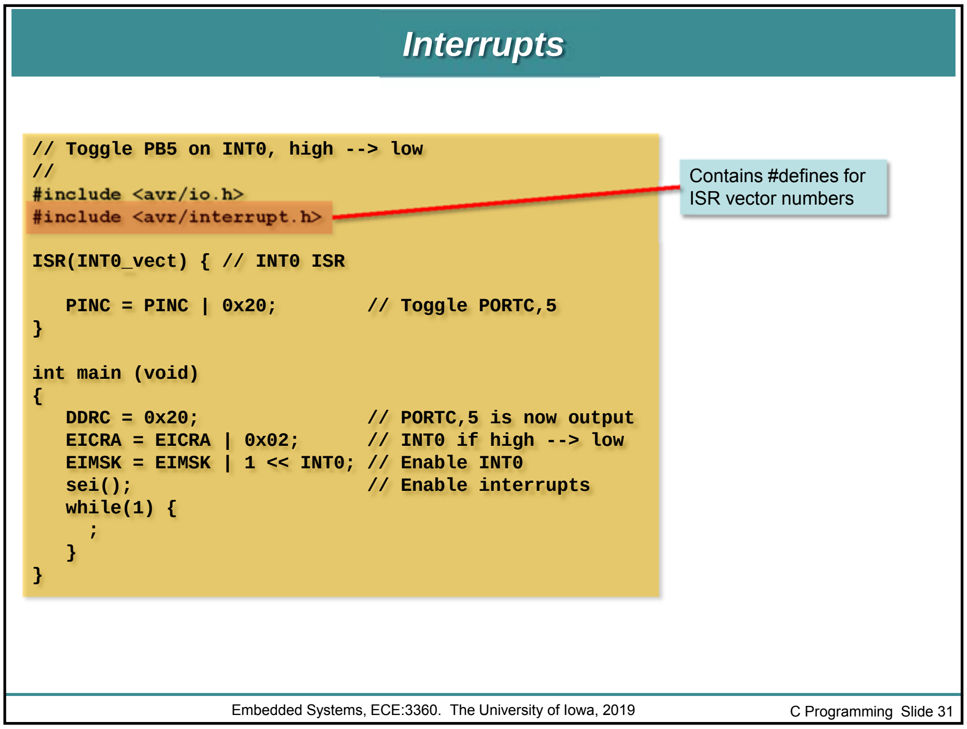
See link below for more information:
http://www.nongnu.org/avr-libc/user-manual/group__util__delay.html



Interrupts

```
// Toggle PB5 on INT0, high --> low
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) { // INT0 ISR

   PINC = PINC | 0x20;        // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;               // PORTC,5 is now output
   EICRA = EICRA | 0x02;      // INT0 if high --> low
   EIMSK = EIMSK | 1 << INT0; // Enable INT0
   sei();                     // Enable interrupts
   while(1) {
     ;
   }
}
```

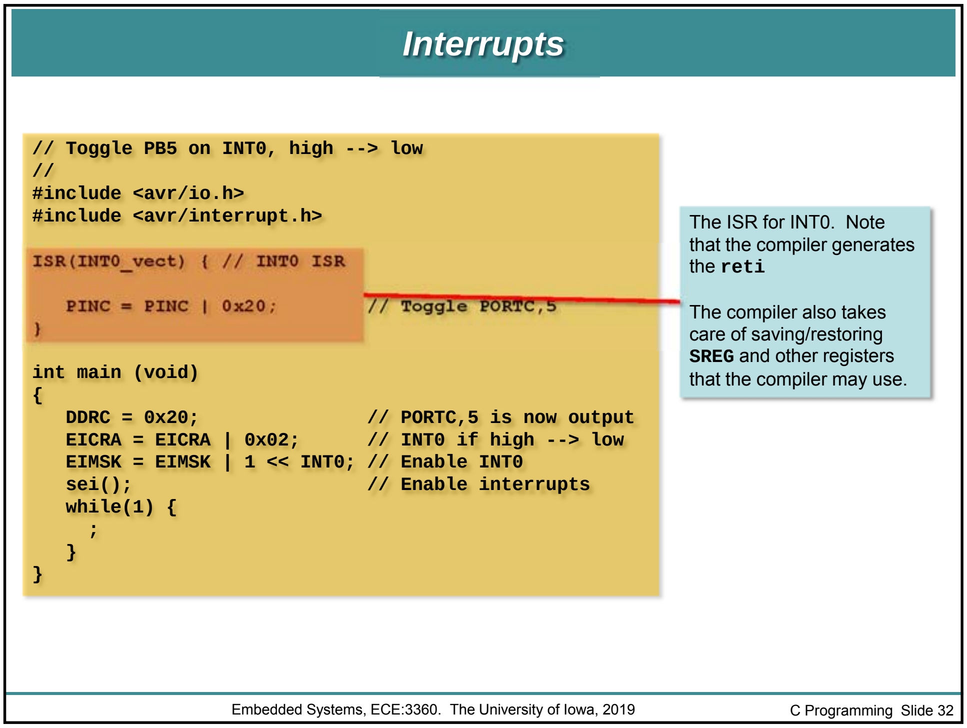
Contains #defines for `INT0, DDRC, EIMSK,` etc.



Interrupts

```
// Toggle PB5 on INT0, high --> low
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) { // INT0 ISR

   PINC = PINC | 0x20;        // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;               // PORTC,5 is now output
   EICRA = EICRA | 0x02;      // INT0 if high --> low
   EIMSK = EIMSK | 1 << INT0; // Enable INT0
   sei();                     // Enable interrupts
   while(1) {
      ;
   }
}
```

Contains #defines for ISR vector numbers

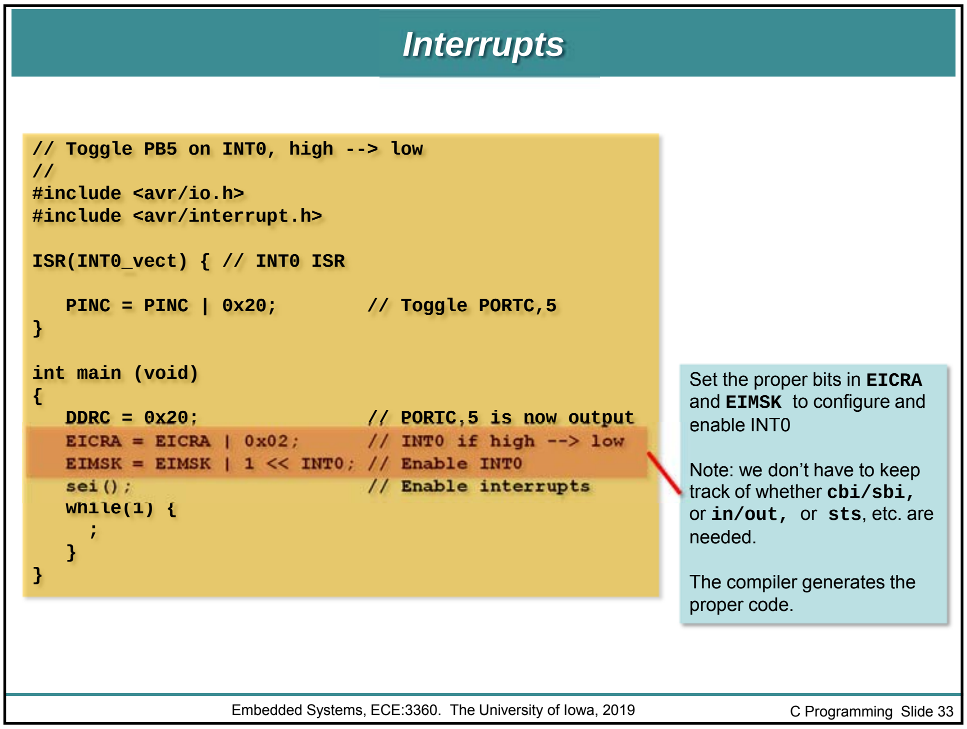


Interrupts

```
// Toggle PB5 on INT0, high --> low
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) { // INT0 ISR

   PINC = PINC | 0x20;        // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;               // PORTC,5 is now output
   EICRA = EICRA | 0x02;      // INT0 if high --> low
   EIMSK = EIMSK | 1 << INT0; // Enable INT0
   sei();                     // Enable interrupts
   while(1) {
      ;
   }
}
```

The ISR for INT0. Note that the compiler generates the `reti`

The compiler also takes care of saving/restoring **SREG** and other registers that the compiler may use.



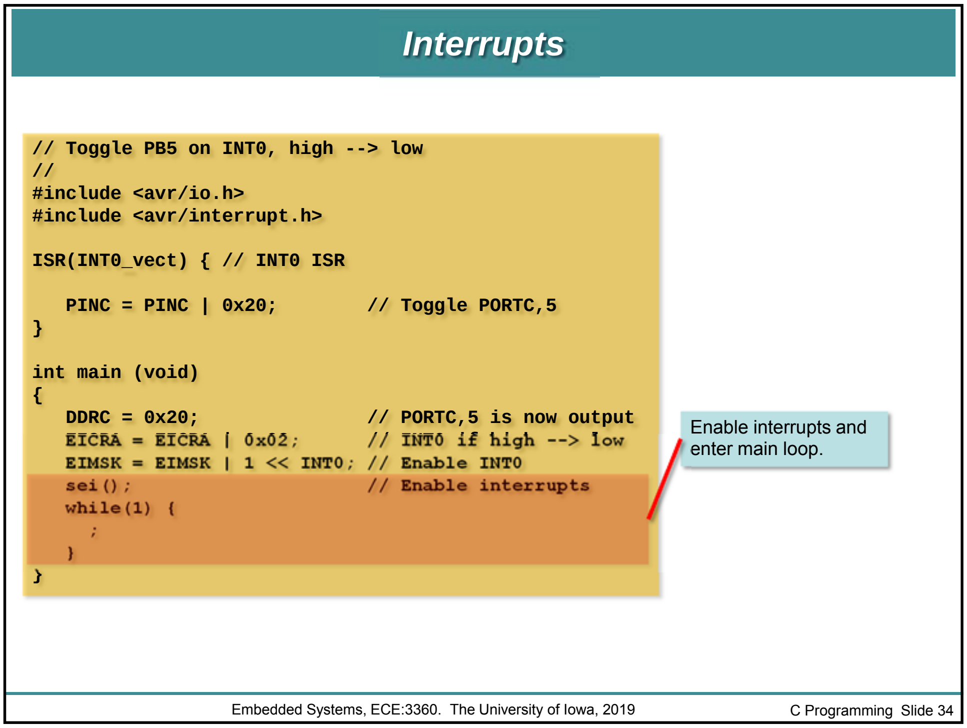
Interrupts

```
// Toggle PB5 on INT0, high --> low
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) { // INT0 ISR

   PINC = PINC | 0x20;         // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                // PORTC,5 is now output
   EICRA = EICRA | 0x02;       // INT0 if high --> low
   EIMSK = EIMSK | 1 << INT0;  // Enable INT0
   sei();                      // Enable interrupts
   while(1) {
     ;
   }
}
```

Set the proper bits in **EICRA** and **EIMSK** to configure and enable INT0

Note: we don't have to keep track of whether **cbi/sbi,** or **in/out,** or **sts**, etc. are needed.

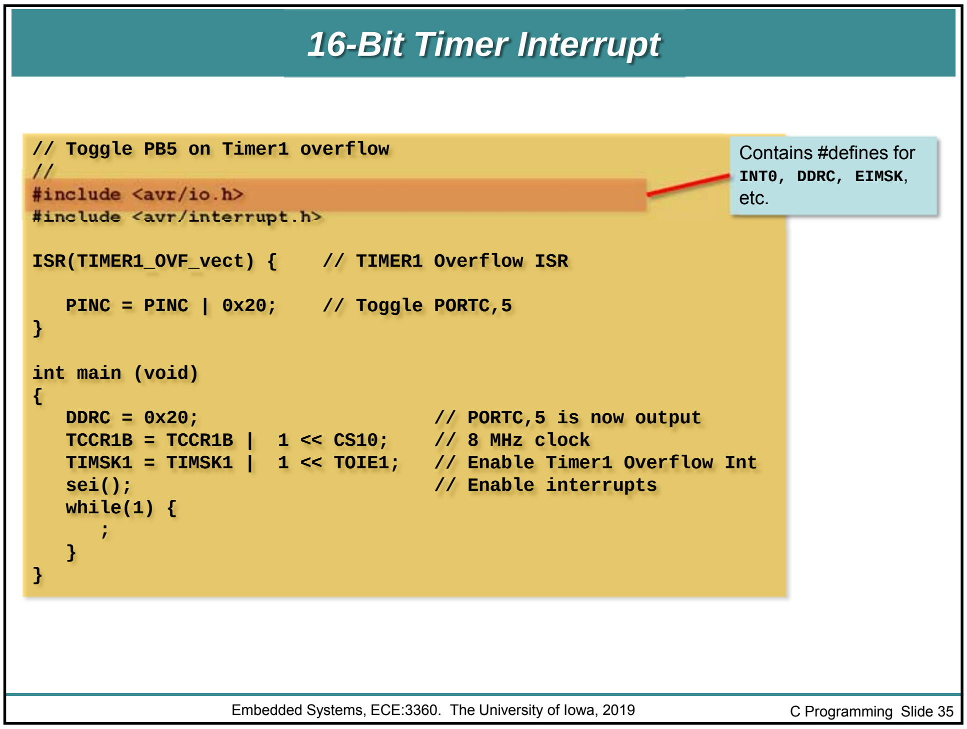
The compiler generates the proper code.



Interrupts

```
// Toggle PB5 on INT0, high --> low
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) { // INT0 ISR

   PINC = PINC | 0x20;         // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                // PORTC,5 is now output
   EICRA = EICRA | 0x02;       // INT0 if high --> low
   EIMSK = EIMSK | 1 << INT0;  // Enable INT0
   sei();                      // Enable interrupts
   while(1) {
     ;
   }
}
```

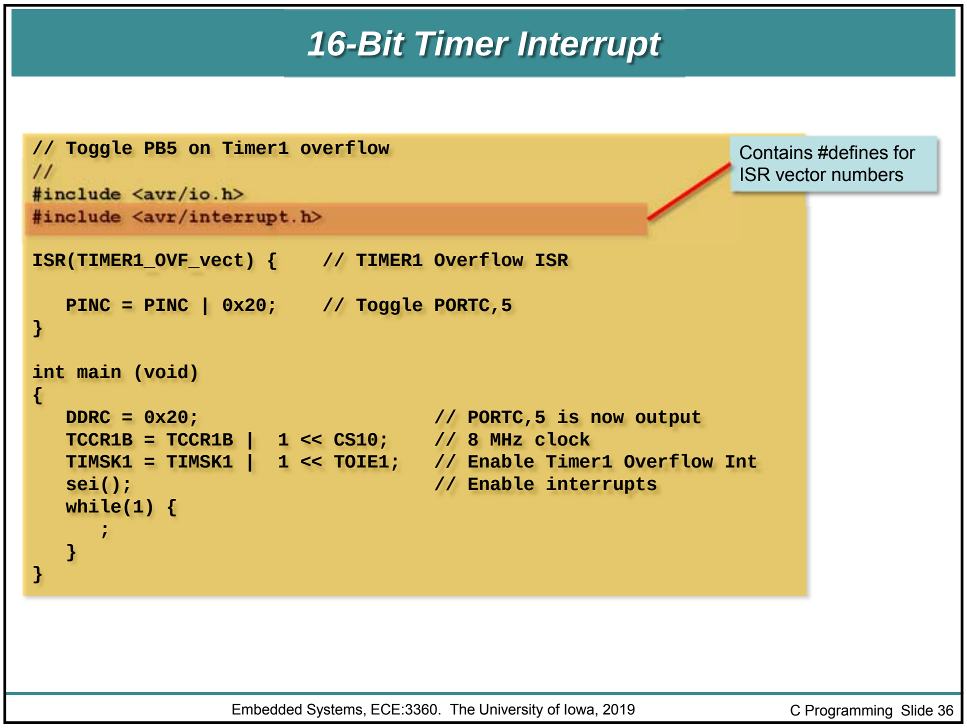
Enable interrupts and enter main loop.



16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {    // TIMER1 Overflow ISR

   PINC = PINC | 0x20;    // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                     // PORTC,5 is now output
   TCCR1B = TCCR1B |  1 << CS10;    // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;   // Enable Timer1 Overflow Int
   sei();                           // Enable interrupts
   while(1) {
      ;
   }
}
```

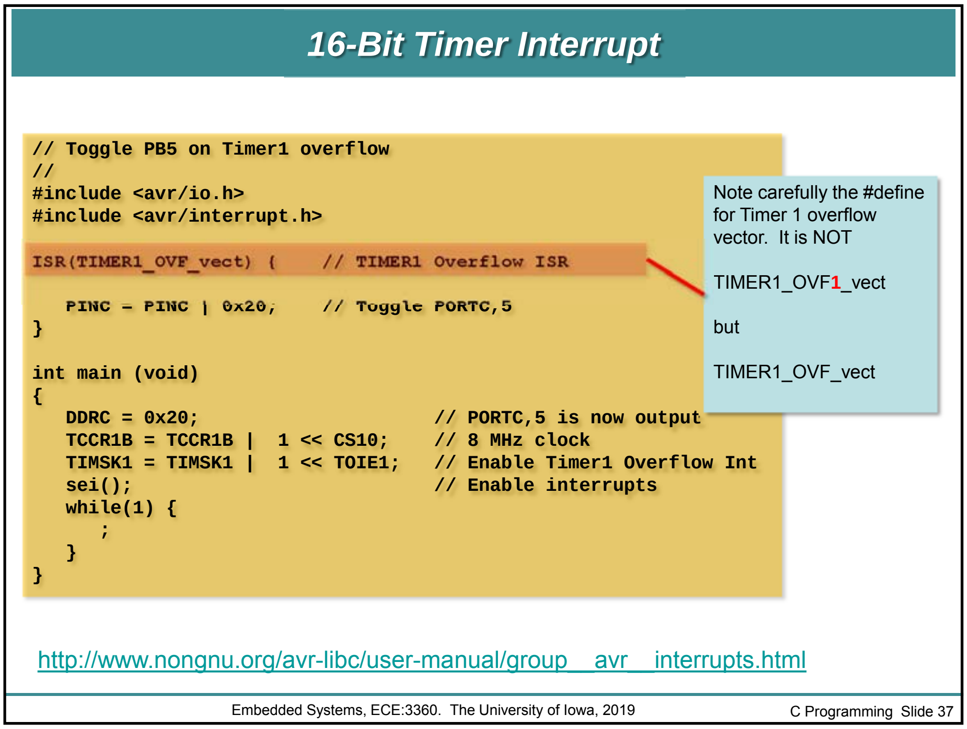
Contains #defines for **INT0, DDRC, EIMSK,** etc.



16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {    // TIMER1 Overflow ISR

   PINC = PINC | 0x20;    // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                     // PORTC,5 is now output
   TCCR1B = TCCR1B |  1 << CS10;    // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;   // Enable Timer1 Overflow Int
   sei();                           // Enable interrupts
   while(1) {
      ;
   }
}
```

Contains #defines for ISR vector numbers



16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {     // TIMER1 Overflow ISR

   PINC = PINC | 0x20;     // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                      // PORTC,5 is now output
   TCCR1B = TCCR1B |  1 << CS10;     // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;    // Enable Timer1 Overflow Int
   sei();                            // Enable interrupts
   while(1) {
      ;
   }
}
```

Note carefully the #define for Timer 1 overflow vector. It is NOT

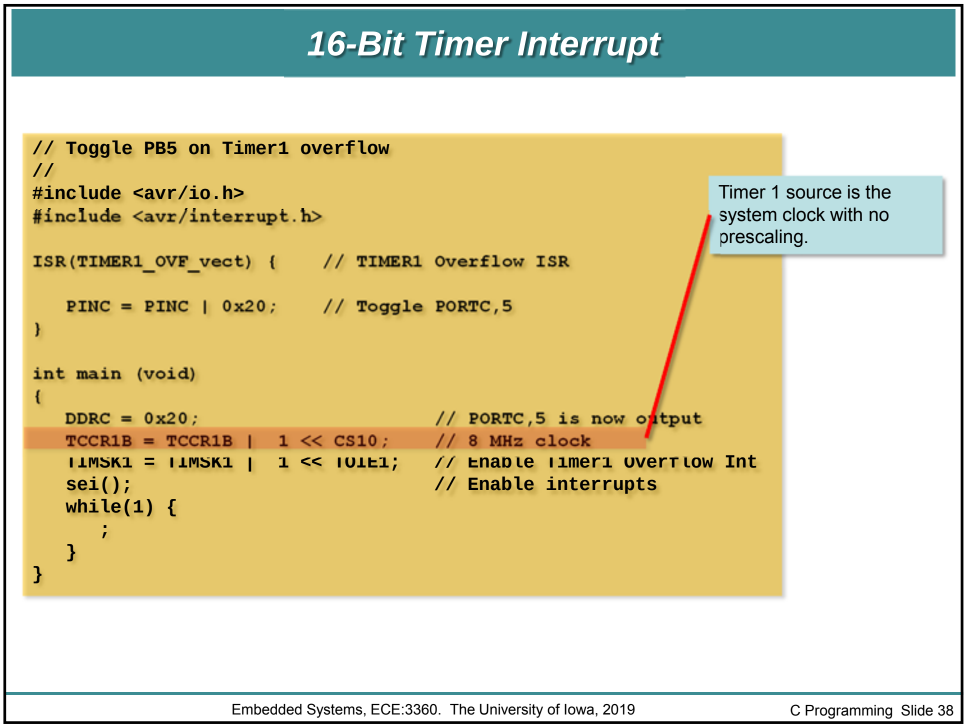
TIMER1_OVF1_vect

but

TIMER1_OVF_vect

http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Embedded Systems, ECE:3360. The University of Iowa, 2019 C Programming Slide 37

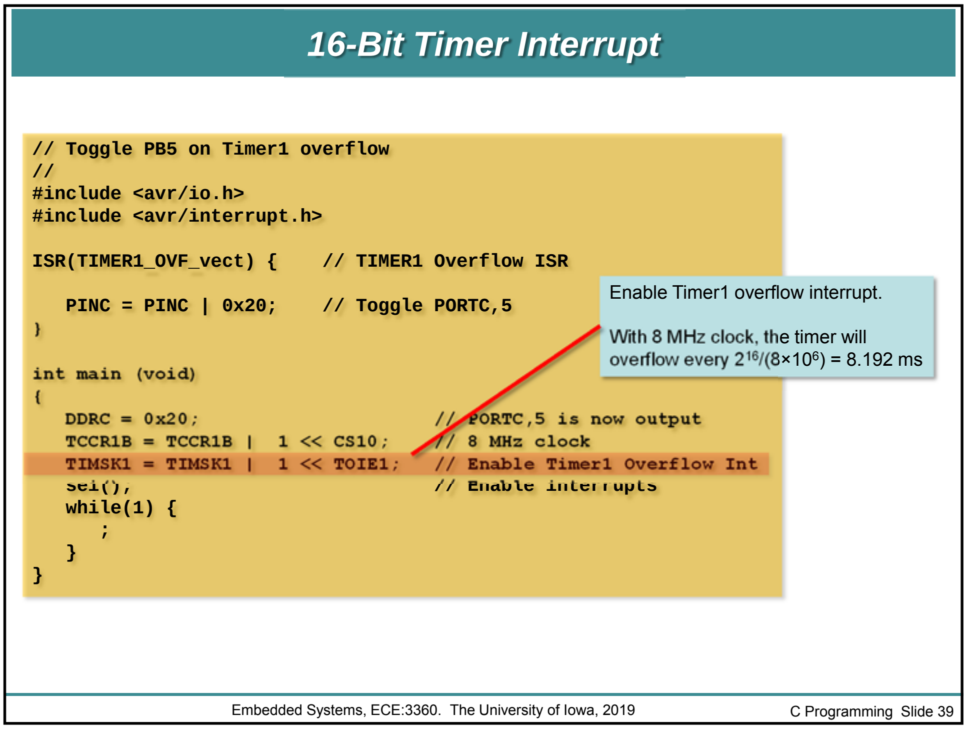


16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {     // TIMER1 Overflow ISR

   PINC = PINC | 0x20;     // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                      // PORTC,5 is now output
   TCCR1B = TCCR1B |  1 << CS10;     // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;    // Enable Timer1 Overflow Int
   sei();                            // Enable interrupts
   while(1) {
      ;
   }
}
```

Timer 1 source is the system clock with no prescaling.

Embedded Systems, ECE:3360. The University of Iowa, 2019 C Programming Slide 38

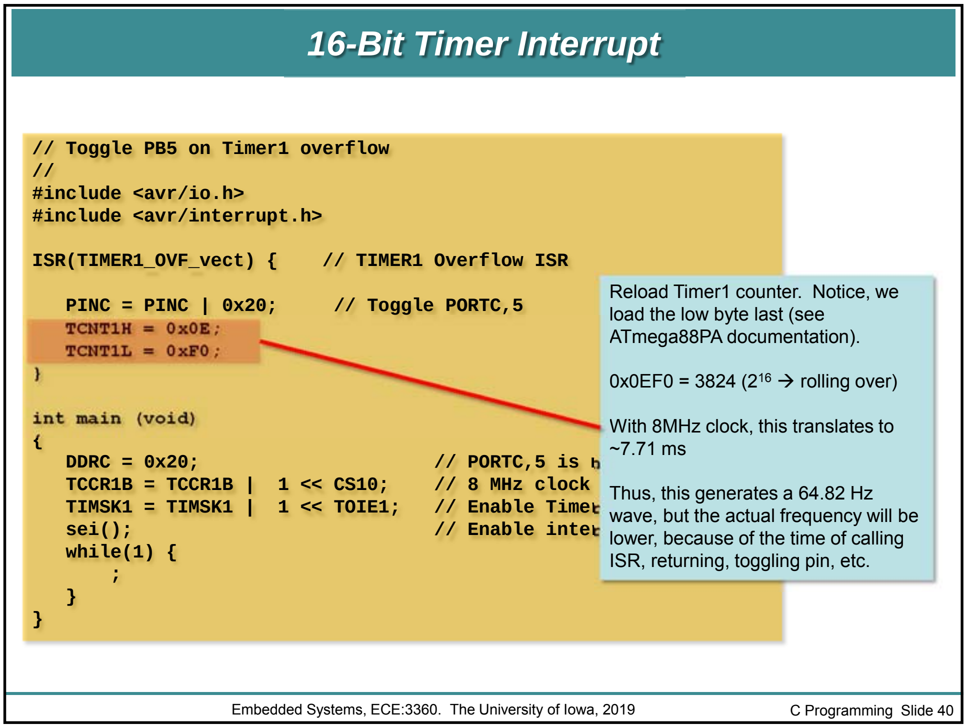


16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {    // TIMER1 Overflow ISR

   PINC = PINC | 0x20;    // Toggle PORTC,5
}

int main (void)
{
   DDRC = 0x20;                        // PORTC,5 is now output
   TCCR1B = TCCR1B |  1 << CS10;       // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;      // Enable Timer1 Overflow Int
   sei();                              // Enable interrupts
   while(1) {
      ;
   }
}
```

Enable Timer1 overflow interrupt.

With 8 MHz clock, the timer will overflow every $2^{16}/(8\times10^6)$ = 8.192 ms



16-Bit Timer Interrupt

```
// Toggle PB5 on Timer1 overflow
//
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_OVF_vect) {    // TIMER1 Overflow ISR

   PINC = PINC | 0x20;     // Toggle PORTC,5
   TCNT1H = 0x0E;
   TCNT1L = 0xF0;
}

int main (void)
{
   DDRC = 0x20;                        // PORTC,5 is n
   TCCR1B = TCCR1B |  1 << CS10;       // 8 MHz clock
   TIMSK1 = TIMSK1 |  1 << TOIE1;      // Enable Timer
   sei();                              // Enable inter
   while(1) {
       ;
   }
}
```

Reload Timer1 counter. Notice, we load the low byte last (see ATmega88PA documentation).

0x0EF0 = 3824 (2^{16} → rolling over)

With 8MHz clock, this translates to ~7.71 ms

Thus, this generates a 64.82 Hz wave, but the actual frequency will be lower, because of the time of calling ISR, returning, toggling pin, etc.

Inline Assembly Language Code

One can embed assembly language instructions in C code by using the **asm** key word.

```
asm volatile(“cli”::);

asm volatile( “sei” ”\n\t”
      ::);
```

```
asm volatile( “nop ; This is a comment” ”\n\t”
              “nop ; This ASM inline includes 2 nops” ”\n\t”
               ::);
```

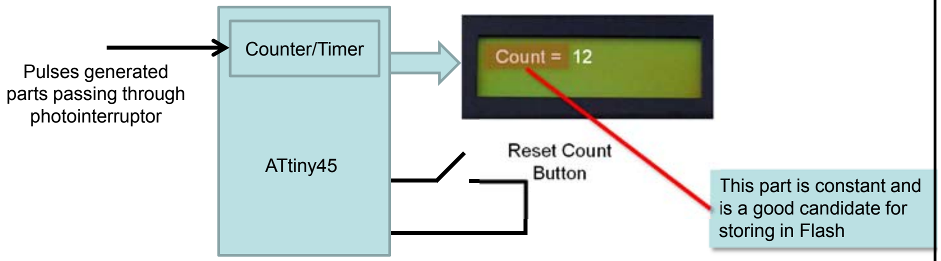
For more details see: www.nongnu.org/avr-libc/user-manual/inline_asm.html

Constant Strings and Tables

SRAM is limited and one would normally not use it for constant strings and tables.

Constant means: does not change at run-time

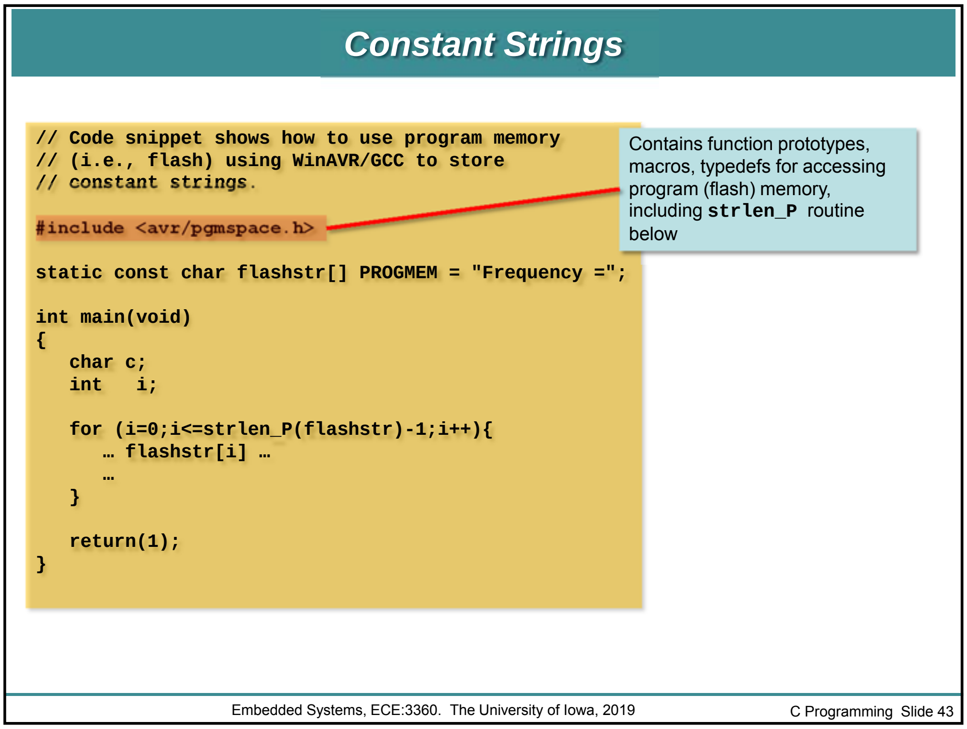
Program memory (Flash) on the other hand, is much more plentiful, and while one cannot write to it during run-time, one can read from it at run-time.



This part is constant and is a good candidate for storing in Flash

We used the AVR **`lpm`** instruction previously when writing software to display constant strings on LCDs

The WinAVR/GCC compiler has built-in keywords/modifiers, macros, and functions for working with constant strings.



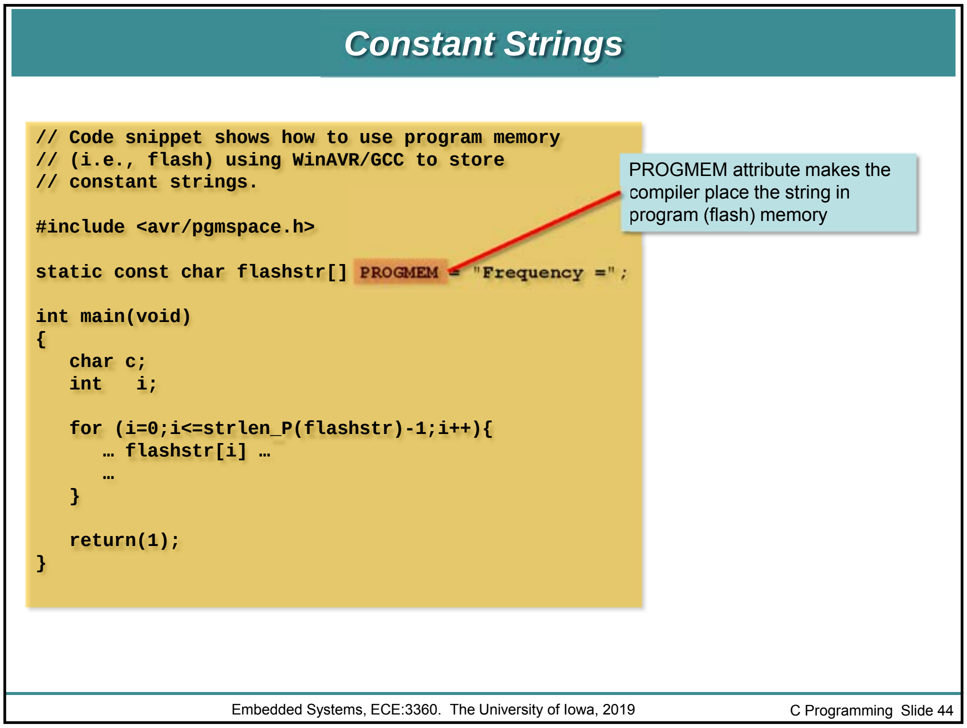
Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency =";

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      … flashstr[i] …
      …
   }

   return(1);
}
```

Contains function prototypes, macros, typedefs for accessing program (flash) memory, including **`strlen_P`** routine below

Embedded Systems, ECE:3360. The University of Iowa, 2019

C Programming Slide 43



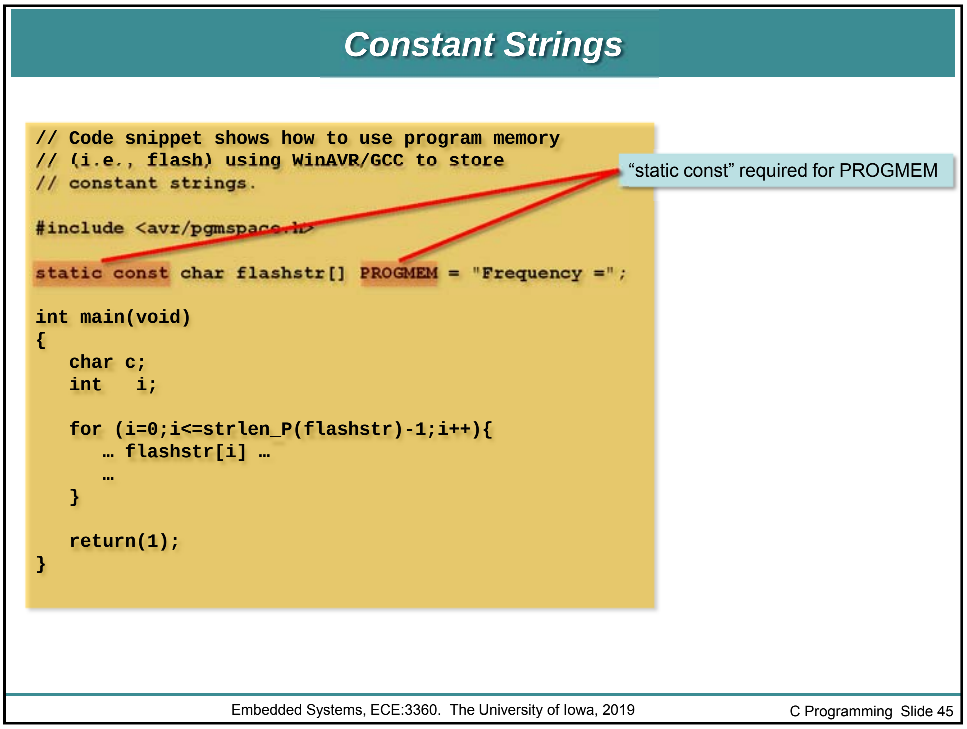
Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency =";

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      … flashstr[i] …
      …
   }

   return(1);
}
```

PROGMEM attribute makes the compiler place the string in program (flash) memory

Embedded Systems, ECE:3360. The University of Iowa, 2019

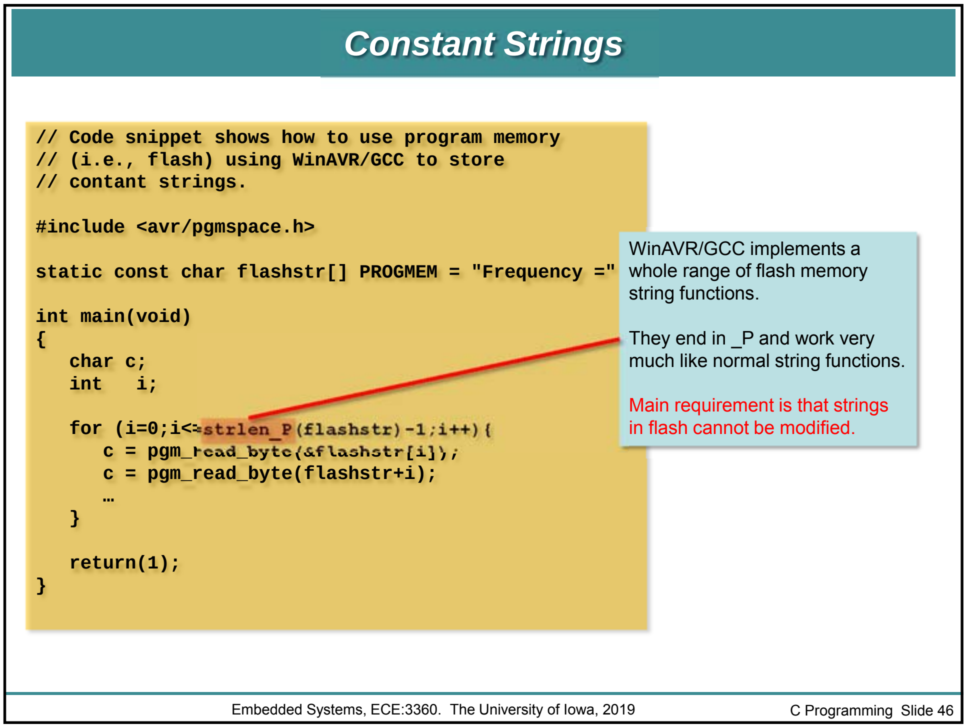
C Programming Slide 44



Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency =";

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      … flashstr[i] …
      …
   }

   return(1);
}
```

“static const” required for PROGMEM



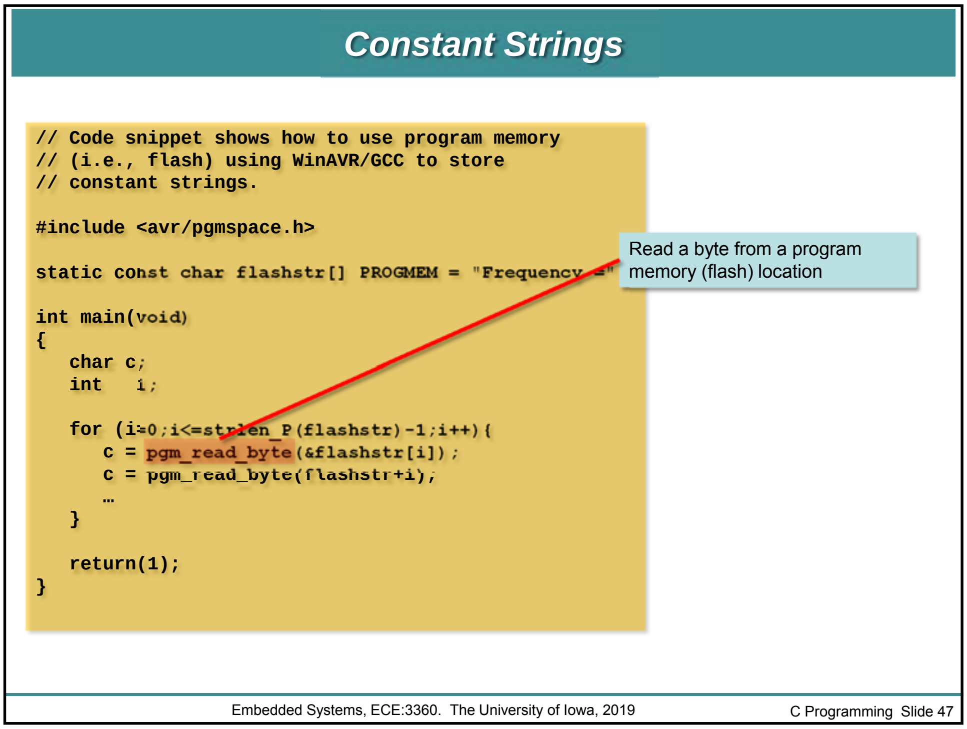
Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// contant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency ="

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      c = pgm_read_byte(&flashstr[i]);
      c = pgm_read_byte(flashstr+i);
      …
   }

   return(1);
}
```

WinAVR/GCC implements a whole range of flash memory string functions.

They end in _P and work very much like normal string functions.

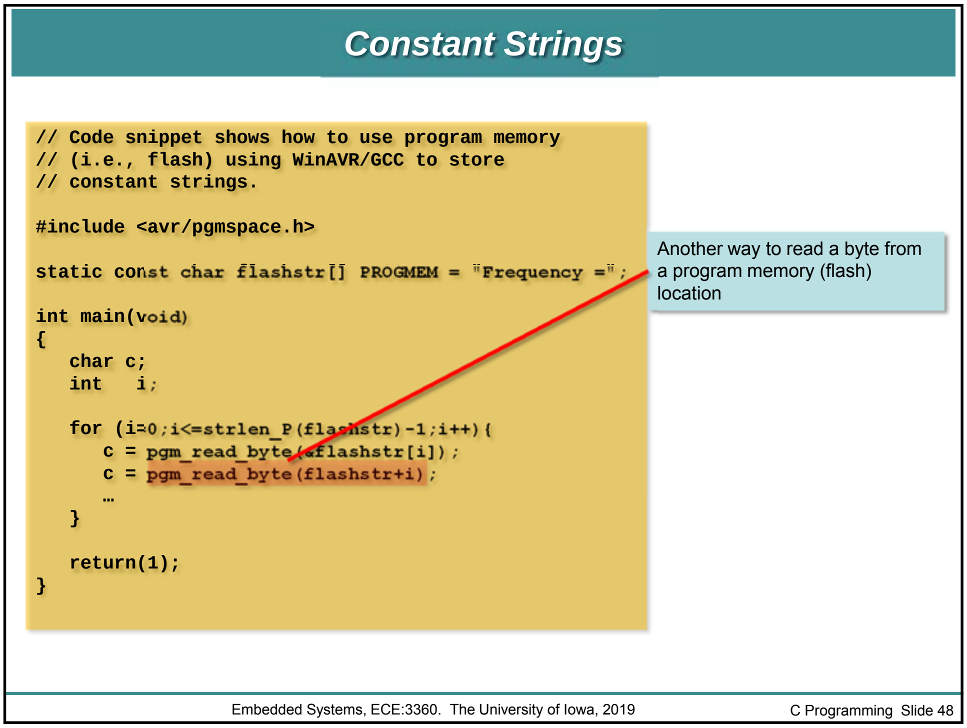
Main requirement is that strings in flash cannot be modified.



Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency ="

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      c = pgm_read_byte(&flashstr[i]);
      c = pgm_read_byte(flashstr+i);
      …
   }

   return(1);
}
```

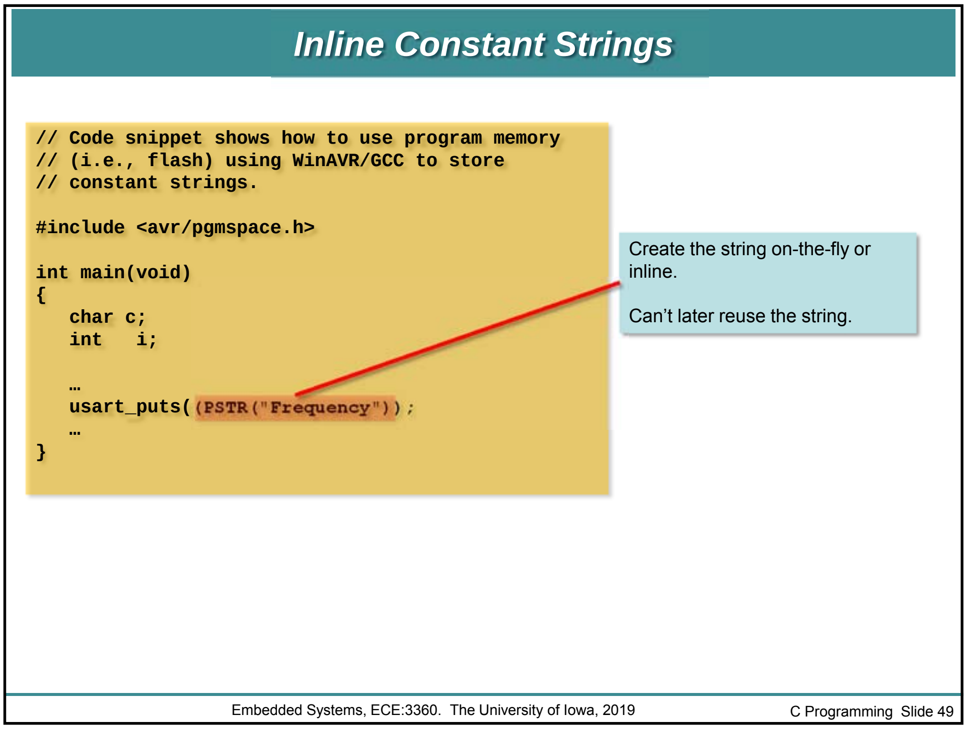
Read a byte from a program memory (flash) location



Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

static const char flashstr[] PROGMEM = "Frequency =";

int main(void)
{
   char c;
   int   i;

   for (i=0;i<=strlen_P(flashstr)-1;i++){
      c = pgm_read_byte(&flashstr[i]);
      c = pgm_read_byte(flashstr+i);
      …
   }

   return(1);
}
```

Another way to read a byte from a program memory (flash) location

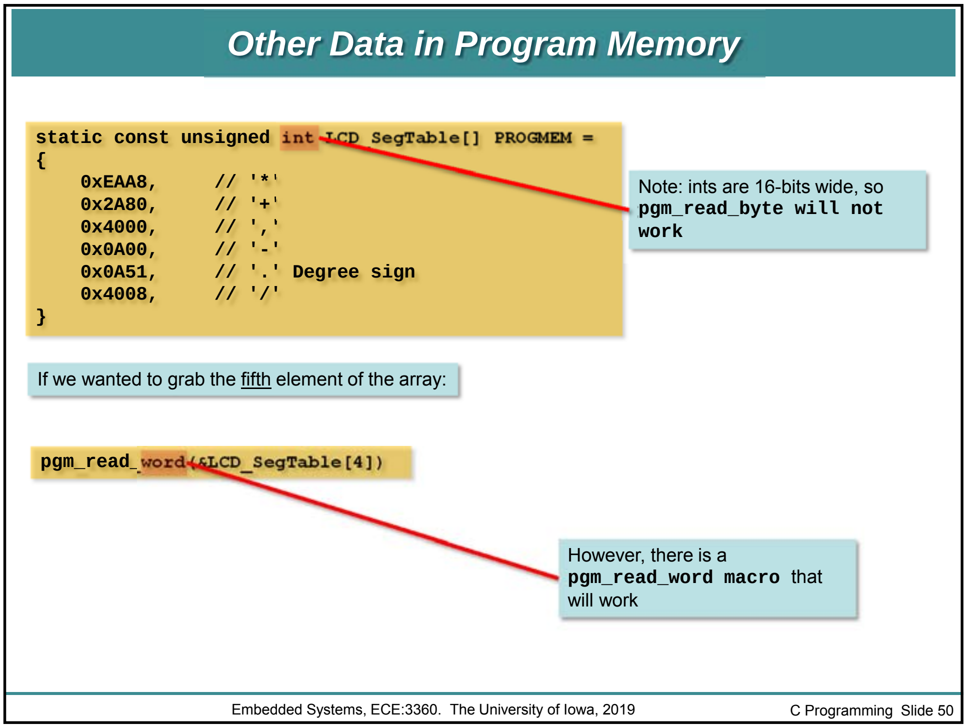


Inline Constant Strings

```
// Code snippet shows how to use program memory
// (i.e., flash) using WinAVR/GCC to store
// constant strings.

#include <avr/pgmspace.h>

int main(void)
{
   char c;
   int   i;

   …
   usart_puts((PSTR("Frequency"));
   …
}
```

Create the string on-the-fly or inline.

Can’t later reuse the string.



Other Data in Program Memory

```
static const unsigned int LCD_SegTable[] PROGMEM =
{
    0xEAA8,     // '*'
    0x2A80,     // '+'
    0x4000,     // ','
    0x0A00,     // '-'
    0x0A51,     // '.' Degree sign
    0x4008,     // '/'
}
```

Note: ints are 16-bits wide, so **pgm_read_byte will not work**

If we wanted to grab the <u>fifth</u> element of the array:

```
pgm_read_word(&LCD_SegTable[4])
```

However, there is a **pgm_read_word macro** that will work

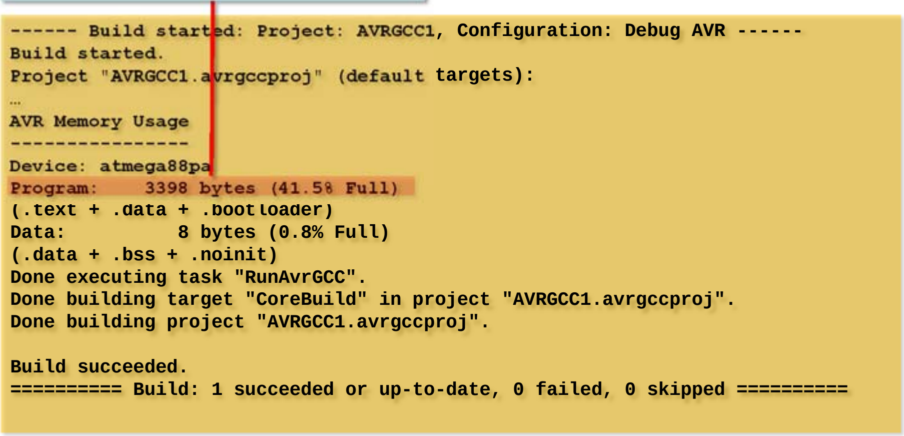
Caution

One has to be careful, with C programming. Consider the following short program (BLINKY.C). If one compiles this and check the program memory (Flash) usage, it shows that it uses more than 40% of the Flash memory!

```
#ifndef F_CPU
#define F_CPU 8000000UL    // 8 MHz clock speed
#endif
#include <avr/io.h>
#include <util/delay.h>

int main (void)
{
   unsigned char  tmp;

   DDRC = 0x20;           // PORTC,5 is now output
   while(1) {

      tmp = PINC5;        // Get PORTC
      tmp = PINC;
      tmp = tmp | 0x20;   // Set bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(300.0);

      tmp = PORTC;        // Get PORTC
      tmp = tmp & ~(0x20);// Clear bit 5
      PORTC = tmp;        // Update PORTC,5
      _delay_ms(100.0);
   }
}
```

Caution

This says that 41% of the FLASH has been used



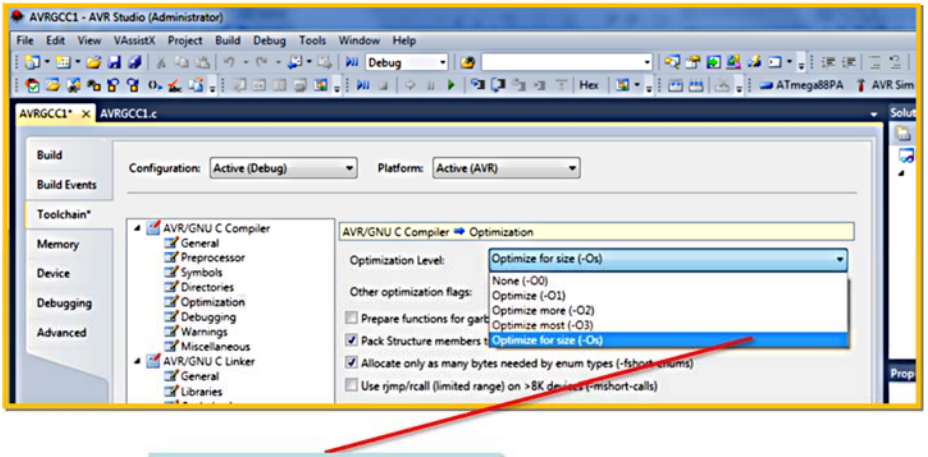
```
------ Build started: Project: AVRGCC1, Configuration: Debug AVR ------
Build started.
Project "AVRGCC1.avrgccproj" (default targets):
…
AVR Memory Usage
----------------
Device: atmega88pa
Program:     3398 bytes (41.5% Full)
(.text + .data + .bootloader)
Data:           8 bytes (0.8% Full)
(.data + .bss + .noinit)
Done executing task "RunAvrGCC".
Done building target "CoreBuild" in project "AVRGCC1.avrgccproj".
Done building project "AVRGCC1.avrgccproj".

Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

How can this be? The culprit turns out to be the `_delay_ms()` routine

The `_delay_ms()` and `_delay_us()` routines uses a significant program memory (they call floating-point math routines to convert arguments to timer ticks) . Either use the lower level `_delay_loop_1()` and `_delay_loop_2()` routines or use the –Os options.

Optimize for Size

Using can **`_delay_ms()`** and **`_delay_us()`** routines us a significant program memory (they call floating-point math routines to convert arguments to times ticks) . Either use the lower level **`_delay_loop_1()`** and **`_delay_loop_2()`** routines or use the –Os options.



Configuration: Active (Debug)

Platform: Active (AVR)

Optimization Level: Optimize for size (-Os)

Other optimization flags:

☐ Prepare functions for garb

☑ Pack Structure members t

☑ Allocate only as many bytes needed by enum types (-fshort-enums)

☐ Use rjmp/rcall (limited range) on >8K devices (-mshort-calls)

Select –Os to optimize for size

Optimization – Issues (…from avr-libc documentation)

- **Q: My program doesn't recognize a variable updated within an interrupt routine when using the optimizer:**

```
uint8_t flag;
...
ISR(SOME_vect)  {
flag = 1;
}
...
int main(void)  {
…
  while (flag == 0) { ...
  }
…
}
```

- **The compiler will typically access flag only once, and optimize further accesses completely away, since its code path analysis shows that nothing inside the loop could change the value of flag anyway.**
- **To tell the compiler that this variable could be changed outside the scope of its code path analysis (e. g. from within an interrupt routine), the variable needs to be declared like: volatile uint8_t flag;**

C & LCD

Option: port existing assembly language LCD routines

Find LCD routines on Web. For example:

http://winavr.scienceprog.com/example-avr-projects/avr-4-bit-lcd-interface-library.html

has a collection or routines for 4-bit interfacing on an ATmega88. We can't guaranteed it works, but it is worth a try.

Here is another candidate:

http://homepage.hispeed.ch/peterfleury/doxygen/avr-gcc-libraries/group__pfleury__lcd.html

Again, we can't guaranteed it works, but it is worth a try. In the past, students had very good results using these routines.

... EOL