Jonathan W. Valvano February 28, 2001, 11:00am-11:50am

**(35) Question 1.** Consider the following simple C program. Assume this code is located in file.c.

```
short A1; int A2;
short B1; short static B2;
short C1; short volatile C2;
short D1=5; short const D2=5;
short F1(short n){ return n+1;}
short static f2(short n){ return n+1;}
short FreezingPoint1=32; short FreezingPoint2=0x20; // degrees F
void program(void){
  short e1; short static e2;
}
```

**(5) Part a)** What is the difference between `A1` and `A2`?

`short` is always 16 bits, while `int` is compiler-dependent. On the ICC12, they are both 16 bits.

**(5) Part b)** What is the difference between `B1` and `B2`?

`B1` is public, and can be accessed from anywhere in the software system. In particular, any file can define B1 as

```
extern short B1;
```

and access this variable. The linker will resolve the address uncertainty. `B2` is private, and can be only be accessed from software in this particular file.

**(5) Part c)** What is the difference between `C1` and `C2`?

`C2` will not be optimized by the compiler (`C1` can be optimized). In particular, it will not keep a copy of `C2` in a register, rather it will reload a new value each time it is needed. It assumes `C2` can be changed by operations other than direct software action. Two good applications of `volatile` are I/O ports and global variables shared by two or more threads. If SC0SR1 were not `volatile` then the compiler could take this C code

```
unsigned char InChar(void){
   while ((SC0SR1 & RDRF) == 0){};
   return(SC0DRL);}
```

and create the following "optimized" assembly

```
InChar:: ldaa SC0SR1   ; get a copy of SC0SR1
loop:    bita #$80     ; check for RDRF
         beq  loop
         ldab SC0DRL
         clra
         rts
```

In the next example, assume `count` is incremented by a background interrupt and `wait` is called from the foreground. If `count` were not `volatile` then the compiler could take this C code

```
unsigned char count;
void wait(void){
   while (count<100){};
}
```

and create the following "optimized" assembly

```
wait::  ldaa count   ; get a copy of count
loop:   cmpa #100     ; check for RDRF
        blo  loop
        rts
```

**(5) Part d)** What is the difference between `D1` and `D2`?

Formally, `const` means can't be changed. On an embedded system, it means D2 is stored in ROM, and D1 is stored in RAM. With ICC12, there will be two copies of D1. The initial value is stored in ROM as "idata" and the actual working copy of D1 is stored in RAM.

**(5) Part e)** What is the difference between `e1` and `e2`?

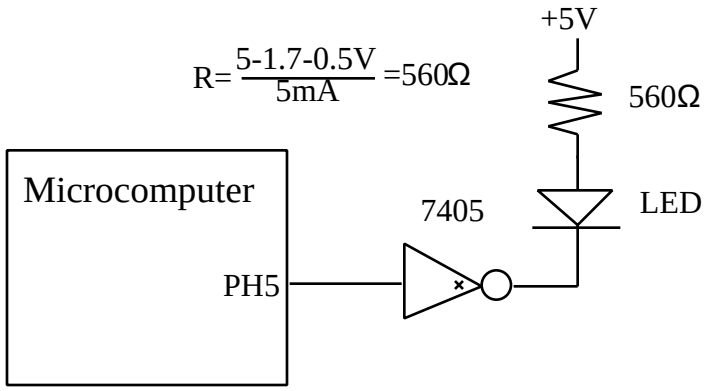
When used inside a function `static` has a different meaning than when used outside a function. In this situation, it means `e2` is statically allocated in permanent RAM (data section of ICC12) and the values persist from one function call to the next. `e1` is dynamically allocated on the stack, used inside the function, and deallocated at the end of the function. The values of `e1` do not persist from one function call to the next.

**(5) Part f)** What is the difference between `F1` and `f2`?

In this context, `static` is used in a similar manner to the way `static` is used for a variable outside a function. `F1` is public, and can be called from anywhere in the software system. In particular, any file can define F1 as
`extern short F1(short n);`
and call this function. The linker will resolve the address uncertainty. `f2` is private, and can be only be called from software in this particular file.

**(5) Part g)** What is the difference between `FreezingPoint1` and `FreezingPoint2`?
The only difference is style. `32` is much better style (easy to read) than `0x20` in this context.

**(15) Question 2.** Interface an LED. The LED has a desired operating point of 1.7 V and 5 mA.




**(50) Question 3.** Show the implementations for these three functions.

```
// Initialize analog interface
void Analog_Init(void){
  COPCTL = 0;     // diable COP
  TSCR   = 0x80;  // enable timer
  TMSK2  = 0x33;  // 1us TCNT
  DDRJ   = 0x00;  // PORTJ inputs
  DDRH   = 0xBF;
  PORTH  = 0;     // initially zero
}
// Output data to DAC channel
void Analog_Output(unsigned char channel, unsigned char data){
  short endT;
  DDRJ  =  0xFF;    // output data
  PORTJ =  data;    // send data
  PORTH =  channel; // specify channel
  PORTH |= 0x20;    // start DAC
  endT  =  TCNT+1000;
  while((endT-(short)TCNT)>0){};  // wait 1 ms
  PORTH &= ~0x20;   // stop
  DDRJ  =  0x00;    // input data
}
// Input data from ADC channel
unsigned char Analog_Input(unsigned char channel);
  unsigned char data;
  PORTH =  channel; // specify channel
  PORTH |= 0x80;    // start ADC
  while((PORTH&0x40)==0){};  // wait for ADC
  data = PORTJ;     // read ADC result
  PORTH &= ~0x80;   // stop
  while((PORTH&0x40)!=0){};  // wait for ADC
  return data;
}
```