Lecture 8: C Programming III

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers

Lesson Objectives

- Become familiar with scope rules in C and be able to place variables in appropriate scope
- Understand use of static, const, and volatile variable qualifiers
- Understand and be able to use macros
- Understand and be able to use structs
- Understand and be able to use bitwise operations and bit masks

Scope Rules in C

- We have already learned about <u>global</u> variables and <u>local</u> variables
  - Global variables: declared outside of any function and can be used anywhere
  - Local variables: declared inside a function, only valid inside that function

```c
// Global variable
double gVoltage = 0.0 ;

void main(){

    gVoltage = ADC_res*3.3/16834;

    return ;
}
```

```c
void main(){

    // Local variable
    double gVoltage = 0.0 ;

    gVoltage = ADC_res*3.3/16834;

    return ;
}
```

Scope Rules in C

- Actually, variables can be declared not just at file scope (global) or within functions (local), but also within <u>blocks</u>
  - Blocks are any part of code surrounded by { }
  - Examples: for loops, while loops, if statements, etc.

```
void main(){

    int counter = 0 ;
    ...
    if(counter > 10){
       int inner_count = 0 ;
       ...
    }
    // inner_count not valid here
    ...
    return ;
}
```

```
void main(){
    int i ;

    for(i = 0; i<4; i++){
        short position = 200 ;
        // position valid here
        ...
    }
    // position not valid here
    ...
    return ;
}
```

Scope Rules in C

- Memory considerations:
  - When variables lose scope (e.g., local variables in a function that returns) their memory is freed
  - Same is true for variables declared only in a block
  - Variables that are at global scope *hog memory for the lifetime of your program*
  - Since memory space on MCU’s can be very small, important to minimize number of global variables
  - Best practice is to declare vast majority of variables as local within a function
    - *Local within a block is not used very often.*

Pass by Value vs Pass by Reference

- What happens when you need to pass variables from function to function?
  - By default, function arguments are “passed by value” – a copy of the data is made within the new function

```
void readPosition(){
    int position = read_from_sensor() ;

    controlFunction(position) ;
}

void controlFunction(int position){

// Changing position here does not change position variable in readPosition
// function.  position here is considered local variable
    position = 2 ;

    return ;
}
```

Pass by Value vs Pass by Reference

- To change a local variable in one function from another function, you must pass data by reference instead
  - To achieve pass by value, you can pass a pointer to data
  - Example: Function that swaps two variables

```
void swap(int* p, int* q){

    int temp ;

    // place p in temp variable
    temp = *p ;
    // assign p to q
    *p = *q ;
    // assign q to temp
    *q = temp ;
}
```

```
void main(){
    int p = 5 ;
    int q = 3 ;

    // pass in addresses of p and q
    swap(&p, &q)

    return ;
}
```

Static Variables

- static keyword is used to *allow a variable to retain its value when a block is re-entered*
  – Normally, when a block exits the variable is completely destroyed
  – Not the case with static variables

```
void f(void){

    static int  count = 0 ;

    count++ ;

    if (count % 2 == 0){
        ...
    }else{
        ...

    return ;
}
```

- When function is entered for the first time, count initialized to 0
- Thereafter, whenever function is called, count retains its previous value from last time function was called (count preserved in memory)
- Initialization only happens first time function is called

Static Functions

- A function can be declared as static as well, which has a totally different meaning
- Usually, functions in one .c file can be called by functions in another .c file
  - This means you can easily develop your code in multiple files, which assists in modular programming
- Functions that are declared with static keyword are *only visible to the file which contains them*
  - Cannot be called by functions which live in other .c files

Static Function Example

sourceFile1.c

```
static void functionOne(int calcValue){

    int q = calcvalue ;
    // can call functionTwo from here

    ...
    return ;
}
```

sourceFile2.c

```
void functionTwo(void){

    // cannot call functionOne from here

    ...
    return ;
}
```

const Keyword

- Constant variables (declared with the const keyword) are initialized once, but can never be changed thereafter
  - Trying to change a const variable will result in a compiler error

```
void f(void){

    const float gravity = 9.81 ;

    float weight = mass*gravity ;

    gravity = 0.0 ;        Will result in compiler error!

    return ;
}
```

volatile Keyword

- volatile keyword used rarely in typical computer programs, but used often in microcontroller programming
- volatile is used to denote that *a variable may be changed by hardware* even though it appears to not change within program
- In MCU programming, often we assign a variable to a certain memory location
  - This memory location may be tied to a hardware device – say an analog to digital converter

volatile Keyword

- In microcontroller programming, most often used when a variable is changed in an interrupt routine
  - Interrupt service routine is called when something (which you specify) happens in hardware
    - *i.e., a digital input pin goes high*
  - This causes execution to stop, interrupt service routine is executed, and then normal execution routine resumes

volatile Keyword

- Example: Count the number of falling edges on a pin, store in global variable

```
volatile int fallingEdges = 0 ;

void PORT1_IRQHandler(void){

  fallingEdges = fallingEdges + 1;

}

void main(){

  initialize_EdgeCounter() ;

  while(1){
     printf("falling edges = %d\n",fallingEdges);
     WaitForInterrupt() ;
  }
}
```

Interrupt service routine

volatile Keyword

- Why do you even need to declare something volatile?
- Why can't this just be treated as a global variable, and if it gets changed by hardware (or in interrupt routine), who cares?

volatile Keyword

- Why do you even need to declare something volatile?
- Why can’t this just be treated as a global variable, and if it gets changed by hardware (or in interrupt routine), who cares?
- Compiler performs optimization steps
  - It analyzes program workflow, and optimizes execution
  - By default, it assumes that the only time a variable gets changed is when the program changes it
  - Optimization is performed under this assumption
  - volatile tells not to make this assumption – otherwise compiled code may yield incorrect execution!

Structures (struct)

- We already know about variable types and arrays
- In C, you can define your own container for data consisting of a heterogenous group (array) of variables
- Usually done in header files or at global scope

In header file:

```
struct carData{

    int make ;
    int model ;
    int color ;
    int license ;
} ;
```

In source code:

```
void main(){

  // declare variable c of type carData
  struct carData c ;

  // assign data
  c.make = 3 ;
  c.model = 14 ;

}
```

Structures (struct)

- Advantage to using structs is that you can pass whole groups of data around by sending a single pointer
- Example from air vehicle control software:

In guidance.h:

```
struct StateVectorStruct{

      double x ;                        // ft
      double y ;                        // ft
      double z ;                        // ft
      double xdot ;                     // ft/s
      double ydot ;                     // ft/s
      double zdot ;                     // ft/s
      double heading ;                  // rad
      double headingrate ;              // rad/s
      double wx ;                       // ft/s
      double wy ;                       // ft/s
};
```

In guidance.c:

```
void main(){

  struct StateVectorStruct StateVector ;


  StateVector.x = read_GPS(0) ;
  StateVector.y = read_GPS(1) ;
  StateVector.z = read_GPS(2) ;
  StateVector.xdot = read_GPS(3) ;
  ...

  //Send entire state vector to be output (pass by ref)
  output_data(&StateVector) ;


}
```

typedef

- typedef keyword allows you to explicitly associate a variable type with an identifier
  - Then provides mechanism for more intuitive variable declarations

In source code:

In header file:

```
typedef  char  uppercase ;
typedef  int   INCHES;
typedef unsigned long size_t;
```

```
void main(){

  // declare variable U of type uppercase
  uppercase  U ;

  // declare variable length and width to
  // be of type INCHES
  INCHES  length ;
  INCHES  width ;

}
```

Macros

- Recall our discussion from first C lecture on #define statements (macro definitions)
  – Given macro definition:

```
#define    arg1    arg2
```

  – During preprocessing, arg1 is replaced everywhere in code by arg2
- In previous discussion, we had used this functionality to set constants, etc

```
#define    VectorSize    400
```

Macros

- You can also use macros to implement some type of function/calculation
  - General definition:

```
#define   identifier( identifier, ..., identifier)   expression
```

  - Example:

```
#define   SQ(x)     ((x) * (x))
```

In rest of code: `SQ(7+w)` expands to `((7+w) * (7+w))`

Macros

- Why would you want to do this?
- Why not just encapsulate everything in function calls?

Macros

- Why would you want to do this?
- Why not just encapsulate everything in function calls?
- Every time a function is called, some overhead is incurred
  - Time and memory used to push arguments on stack, etc.
- Macros allow us to replace function calls with inline code, which is more efficient
  - However, they allow inline code to be encapsulated by something that "looks like" a function

Macros

- Macros are used often in microcontroller programming particularly for memory locations/registers
  - Registers are fixed locations in memory control GPIO pin functionality, Analog to Digital conversion, etc.
  - Rather than remember all the memory locations (hex values), we use macros to relate a readable name to that memory location
    - *When we want to write to memory location in code, we call it by name rather than hex memory location*

Macros

- Example from mspm0g350x.h

In mspm0g350x.h:

```
#define GPIOB_BASE  (0x400A2000U)      /*!< Base address of module GPIOB */
...
static GPIO_Regs     const GPIOA       = ((GPIO_Regs *) GPIOA_BASE);
```

In hw_gpio.h:

```
typedef struct {
...
 volatile uint32_t DOUTSET31_0;                        /* !< (@ 0x00001290) Data output set 31 to 0 */
} GPIO_Regs;

#define GPIO_DOUTSET31_0_DIO26_MASK  ((uint32_t)0x04000000U)  /* !< sets DIO26 bit in DOUT31_0 register */
```

In MSPM0 source code:

```
#include "mspm0g350x.h"

int main(){
...
    GPIOA->DOUTSET31_0 = GPIO_DOUTSET31_0_DIO26_MASK  ;          // Set pin PA26   ;
...
```

Macros

- For instance, look at memory map at line 254 of mspm0g350x.h

Bitwise Operations

- Binary bitwise operations are used a lot in microcontroller programming
  - Usually to set options
- Bitwise operators perform logical operations on each corresponding bit of two values
  - Bitwise AND: &
  - Bitwise XOR: ^
  - Bitwise OR: |

a	b	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Bitwise Operations

- In microcontroller programming, registers are used to control functionality
  - By setting certain bits in registers, we can modify functionality as desired
- For instance, GPIOA digital pin outputs are enabled by setting bits at memory location GPIOA->DOE31_0
  - This is memory location 0x400A12C0 (can be determined by looking at macros and offsets in Reference Manual)

Bitwise Operations

- Suppose we want to enable outputs on pins PA04 and PA07

9.3.43 DOESET31_0 (Offset = 12D0h) [Reset = 00000000h]

DOESET31_0 is shown in Figure 9-46 and described in Table 9-46.

Return to the Summary Table.

Writing 1 to a bit position in this register sets the corresponding bit in the DOE31_0 register.

Figure 9-46. DOESET31_0

31	30	29	28	27	26	25	24
DIO31	DIO30	DIO29	DIO28	DIO27	DIO26	DIO25	DIO24
W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h
23	22	21	20	19	18	17	16
DIO23	DIO22	DIO21	DIO20	DIO19	DIO18	DIO17	DIO16
W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h
15	14	13	12	11	10	9	8
DIO15	DIO14	DIO13	DIO12	DIO11	DIO10	DIO9	DIO8
W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h
7	6	5	4	3	2	1	0
DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h

Table 9-46. DOESET31_0 Field Descriptions

Bit	Field	Type	Reset	Description
31	DIO31	W	0h	Writing 1 to this bit sets the DIO31 bit in the DOE31_0 register. Writing 0 has no effect. 0h = No effect 1h = Sets DIO31 in DOE31_0
30	DIO30	W	0h	Writing 1 to this bit sets the DIO30 bit in the DOE31_0 register. Writing 0 has no effect. 0h = No effect 1h = Sets DIO30 in DOE31_0

Will need to set bits 4 and 7 of this register to 1

Bitwise Operations

- Steps to do this

```
pin4_mask = 0x00000010 ;  // 1 in bit 4, zeros everywhere else

pin7_mask = 0x00000080 ;          // 1 in bit 7, zeros everywhere else

bit_mask = pin4_mask | pin7_mask ;
```

```
#include “msp.h”

int main(){

   GPIOA->DOESET31_0 = bit_mask ;       // Set bits in DOESET31_0 register
...
```

Bitwise Operations

- Steps to do this

```
pin4_mask = 0x00000010 ;  // 1 in bit 4, zeros everywhere else

pin7_mask = 0x00000080 ;          // 1 in bit 7, zeros everywhere else

bit_mask = pin4_mask | pin7_mask ;
```

bitwise OR operation

```
#include “msp.h”

int main(){

   GPIOA->DOESET31_0 = bit_mask ;       // Set bits in DOESET31_0 register
...
```

Macro expands to *(0x40000000)

Offset of 0x12D0 from base GPIOA register addresses

Bitwise Operation

- Performing this bitwise operation yields (only lower 16 bits shown):

`0x00000010`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
`0x00000080`	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Bitwise OR																
`0x00000090`	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0

Gets written to DOESET31_0 register

Bitwise OR operations used often to combine different options when writing to hardware registers.

Bitwise Operations

- Another example: Setting outputs on GPIO pins
  - Setting bits in DOUT31_0 register controls whether output pins are high or low (if output already enabled)

9.3.38 DOUT31_0 (Offset = 1280h) [Reset = 00000000h]

DOUT31_0 is shown in Figure 9-41 and described in Table 9-41.

Return to the Summary Table.

Data output for pins configured as DIO31 to DIO0.

Figure 9-41. DOUT31_0

31	30	29	28	27	26	25	24
DIO31	DIO30	DIO29	DIO28	DIO27	DIO26	DIO25	DIO24
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h
23	22	21	20	19	18	17	16
DIO23	DIO22	DIO21	DIO20	DIO19	DIO18	DIO17	DIO16
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h
15	14	13	12	11	10	9	8
DIO15	DIO14	DIO13	DIO12	DIO11	DIO10	DIO9	DIO8
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h

Set bit 24 to make output high on pin 24

Clear bit 9 to make output low on pin 9

Bitwise Operations

- Example function which sets pins PA09 and PA24 high, all other GPIOA pins low

```
#include “msp.h”

#define PIN09MASK   0x00000200       // will set bit 9 of register to 1
#define PIN24MASK   0x01000000       // will set bit 24 of register to 1

void Port_Init(void){

    // Set all GPIOA pins low
    GPIOA->DOUT31_0 = 0x00000000 ;   // Clears all pins

    // Now set pins 9 and 24 high
    GPIOA->DOUT31_0 = PIN09MASK | PIN24MASK ;  // = 0x01000200 = pins 9, 24 high,
                                               //  all others low
...
```

Bitwise Operations

- Suppose you want to leave all register values intact and just set a single bit
  - May happen when you are not sure of what other register bit states are

```
#include "msp.h"

#define PIN09MASK   0x00000200   // will set bit 9 of register to 1
#define PIN24MASK   0x01000000   // will set bit 24 of register to 1

void Port_Init(void){

    // Set all GPIOA pins low
    GPIOA->DOUT31_0 = 0x00000000 ;          // Clears all pins

    // Now set pins 9 and 24 high
    GPIOA->DOUT31_0 = PIN09MASK | PIN24MASK ;  // = 0x01000200 = pins 9, 24 high,
                                                //  all others low

    GPIOA->DOUT31_0 |= 0x00000001 ;  // Now set bit 0 high as well without changing anything else
```

Bitwise Operations

- Bitwise OR operation works only if you want to set certain bits high
- Instead, use bitwise & and bitwise negation (~) operation to set certain bits low
- Example: Consider an 8 bit register which has current value of 0x4F
  - You want to set bit 3 low

Example Register:

0 1 0 0 1 1 1 1

(0x4F)

Since you want to set bit 3 low, start with bit "mask" of 00001000

~00001000 = 11110111

New Register Value:

	0 1 0 0 1 1 1 1
&	1 1 1 1 0 1 1 1
	0 1 0 0 0 1 1 1

Bitwise Operations

- Example: Set pin PA05 low, keep all other GPIOA pins high

```
#include “msp.h”

#define PIN05MASK   0x00000020       // will be used to clear bit 5

void Port_Init(void){

    // Set all GPIOA pins high
    GPIOA->DOUT31_0 = 0xFFFFFFFF ;   // Sets all pins

    // Clears pin PA05
    GPIOA->DOUT31_0 &= ~PIN05MASK ;   // Clears bit 5 but leaves other bits intact
...
}
```

Bitwise Operations

- Let’s visualize how DOOUT31_0 register value is changing (last 8 bits only shown below)

Initial state:		1 1 1 1 1 1 1 1	0xFF
`GPIOA->DOUT31_0 &= ~PIN05MASK`	&	1 1 1 1 1 1 1 1	
`PIN05MASK = 0x00000020`		1 1 0 1 1 1 1 1	
		1 1 0 1 1 1 1 1	0xDF

Bitwise Operations

- Sometimes you just want to toggle a bit in a register (without knowing whether it is 1 or 0 beforehand)
  - For instance, toggling an LED
- Use exclusive OR for this (^)
- Example: Consider an 8 bit register which has current value of 0x9E. You want to toggle bit 4.

Example Register:

1 0 0 1 1 1 1 0

(0x9E)

Since you want to set toggle bit 4, start with bit "mask" of 00010000 ⟶

New Register Value:

	1 0 0 1 1 1 1 0
^	0 0 0 1 0 0 0 0
	1 0 0 0 1 1 1 0

Bitwise Operations

- Example program: toggle red and blue LED’s (pins PA0, PB22, and PB26)

Default Initialization

- In general, if variables are declared but not initialized to a value, they contain some random "garbage" number
  - This is because you are just specifying that a certain CPU register will be referred to by a variable name
  - Example:

```
void main(){

  int newVariable ;        // declared by not initialized

  int otherVariable = 4 ;  // declared and initialized

  int result = newVariable + otherVariable ;   // will be indeterminant
```

Default Initialization

- It is best practice to initialize variables as you declare them (always...even if you plan to change value later)

```
void main(){

  float val1 = 0.0 ;
  float val2 = 0.0 ;
  float val3 = 0.0 ;
  float val4 = 0.0 ;
  float val5 = 0.0 ;
  float val6 = 0.0 ;
  ...

}
```

Note: Some special types of variables (arrays, structures, and static variables) are automatically initialized by compiler.