Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.

Electrical and Computer Engineering

Kettering University

Announcement

Lecture 19: C Programming for Embedded Systems

Today’s Topics

By the end of this class you should be able to:

- Describe the basic C programming process
- Declare variables and constants in C
- Use pointers to access specific memory addresses

Using a High Level Language

High-level languages

- Generally, more human-readable
- A high-level language is more processor independent, meaning that the program can sometimes run on several types of hardware, usually with small modifications.
- Less code to do the same amount of work

C programming language

- Developed in 1972 by Dennis Ritchie at the Bell Laboratories.
- Named “C” because it is derived from an earlier language “B.”
- Closely related to the development of Unix OS.
  - Unix was originally written in assembly language on a PDP-7
  - Needed to port PDP-11. It led to the development of an early version of C
  - The original PDP-11 version of the Unix system was developed in assembly language. Later, most of the Unix kernel was rewritten in C.
- Well suited for embedded systems.

C for an Embedded System

- We won’t explicitly discuss C syntax.

- We will focus on C for embedded systems.

- Topics that we will discuss on the next three lecture
    - Definition of variables and constants
    - Calling assembly program from C
    - Using multiple files
    - Parameter passing in C
    - Interrupt handling in C

Constant Declaration

#define

- C has a method of defining constants much like defining constants in assembly.
- Like assembly, declaring constants in this manner does ***not*** use any memory, and the values assigned to the labels are used only during the compile process.

```
fav_num   EQU       $27
```

is equivalent to

```
#define   fav_num   0x27;
```

Basic C Data Types

- Declaring variables in C is similar to assembly in that we must supply a ***label*** and a ***size***.

- In C, however, we must also state whether the variable is ***signed*** or ***unsigned*** so that the complier can pick the *correct version of comparison statements* (i.e. BHI vs. BGT).
  - In *assembly*, this was not necessary because it was the *responsibility of the programmer* to choose the correct version.

Basic C Data Types …

- The common data types are listed below.

Data type	size
unsigned char	8 bits
signed char	8 bits
unsigned short, ***unsigned int***	16 bits
signed short, ***signed int***	16 bits
unsigned long, ***unsigned long int***	32 bits
signed long, ***signed long int***	32 bits

Important Note:

- The above lengths are common, but they are not universal. Different combinations of processors/compilers may use different values. This is one of the first items to verify when using a new device/programming tool.

Basic C Data Types …

Example: Convert the following C variable declarations into assembly variable declarations.

Notes:

- The assembly declarations for signed and unsigned values are the same – no distinction is made.
- Arrays in C use square brackets.
- Without specifying signed or unsigned, signed is used by default (at least by Codewarrior)

C	Assembly		
unsigned char count; uint8_t count;	count	DS.B	1
signed char count; int8_t count:	count	DS.B	1
unsigned int rti_ints; uint16_t rti_ints;	rti_ints	DS.W	1
signed long profit; int32_t profit;	profit	DS.W	2
unsigned char mylist[4]; uint8_t mylist[4];	mylist	DS.B	4

Variable Types Modifiers

- To complicate variable definitions a little more, C compilers require clarification as to ***how the variables behave***. This allows and/or prevents certain *optimizations*.

Static

- Stored in a known RAM address.
- The **contents** of the variable in memory **will only be changed by the program.**
- Optimizations are allowed that assume the value will not change between loads and stores.
- Most variable used in class have been static.
- **Default behavior**

Volatile

- Stored in a known RAM address.
- The ***contents*** of a volatile variable in memory ***may be changed by hardware at any time***.
- ***Input ports are volatile.***

Getting to Specific Addresses

- In the previous variable definitions, we created a variable that the complier assigned to some, *random address*, and any assignments to that variable name change the *memory contents*. So if we said

unsigned char DDRB;

… and then later said…

DDRB = 0xFF ;

… then the 8-bits at address DDRB is changed, as we'd expect.

- However, in the HCS12, DDRB refers to a control register at address 0x0003. ***Thus, the C language needs a way to link a variable to a specific location. How can we do this? Answer! Use pointers!!***

Pointers in C

- A more advanced method to access address is by defining pointers.

- Pointers are addresses of items.

- In C, unlike assembly, a label (a variable) can represent either the **value of a variable** or the **address of a variable**.

- Pointers in C
  - A * symbol denotes a pointer.
  - A & symbol denotes the **address of a variable**.
  - The size of a pointer is the size of an address in the processor, in this case two bytes.
  - A pointer must be specified with the type of item it points to.

Pointers in C…

- The syntax for declaring a pointer type is:

type_name *pointer_name

For Example:

- a) int *ax;
  - Declares that the variable ***ax*** is a pointer to an integer.
- b) char *cp;
  - Declares that the variable ***cp*** is a pointer to a character.

Pointing to Known addresses

- There are different ways to point to specific memory locations in a microcontroller.
- We'll use a common method that gives us a label, like DDRB, without using the pointer symbol * in our code.
- This method works on more compilers than others, making it widely applicable.

The Sequence Explained:

1) volatile unsigned char
   - DDRB should refer to this. "Volatile" tells the compiler the value might change due to hardware.
2) *(volatile unsigned char):
   - Creates a pointer to the desired item.
3) *(volatile unsigned char) 0x0003:
   - Specifies the memory address (0x0003) the pointer points to.
4) **((volatile unsigned char) 0x0003):
   - Gets the value at the memory address 0x0003.
   - This is like saying there's a volatile unsigned char at address 0x0003.
5) **#define DDRB (*(volatile unsigned char *)0x0003)**
   - Assigns the label DDRB to the memory address 0x0003.
   - Now, you can use DDRB in your code, simplifying tasks like DDRB = 0xFF.

```
#define DDRB (*(volatile unsigned char *)0x0003)
```

Breaking it down step by step:

1. Data Type Declaration:

   volatile unsigned char

- volatile informs the compiler that the value might change unexpectedly (e.g., due to hardware).
- unsigned char specifies that we are working with an 8-bit unsigned character.

2. Pointer Declaration:

   (volatile unsigned char *)

- This indicates that we are declaring a pointer to a volatile unsigned char.

3. Memory Address:

   0x0003

- Specifies the memory address we want to access (in this case, the address associated with DDRB).

4. Dereferencing the Pointer:

   (*(volatile unsigned char *)0x0003)

- Retrieves the value stored at the memory address. In this context, it's a volatile unsigned char at address 0x0003.

5. Assigning a Label:

   #define DDRB (*(volatile unsigned char *)0x0003)

- Creates a label (DDRB) for the memory address. Now we can use DDRB in our code instead of the raw memory address.

Examples

```
#define PORTB (*(volatile unsigned char *) 0x0001)
#define DDRB   (*(volatile unsigned char *) 0x0003)

#define PTP      (*(volatile unsigned char *) 0x0258)
#define DDRP   (*(volatile unsigned char *) 0x025A)

#define PTH      (*(volatile unsigned char *) 0x0260)
#define DDRH   (*(volatile unsigned char *) 0x0262)
#define PIEH     (*(volatile unsigned char *) 0x0266)
#define PIFH     (*(volatile unsigned char *) 0x0267)
```

Homework Examples

Write C statements to do each of the following:

1) Set Ports B and P to all outputs:

- DDRB = 0xFF;
- DDRP = 0xFF;

2) Wait for bit 0 of Port H to be 1.

while(PTH & 0x01 == 0x00){}

3) Enable the left most 7-segment display (bit 3 of Port P to 0) and disable the other three digits without affecting other bits.

- PTP = PTP | 0x07;
- PTP &= 0xF7;

Variable Scope

- One issue that arises in C that didn’t in assembly is variable scope.
- Any variable declared in assembly is valid at any point in the file.
- However, in C, the variable is only valid in the ***code block*** in which it is defined and *only after the definition*.
- The overall file is considered a code block.
- Sections of code defined by { } are also considered code blocks.

Variable Scope

```
//  constant declarations
#define inc_value 5;   // valid hereafter, not a variable

//  variable declarations
static int sum;          // valid hereafter, is a variable

void main(void){     // main program block
        int temp = 3;        // local to the function "main"
        sum = 0;              // local to the function "main", hides the variable above
        for(int count=0; count<10; count++)
        {
                sum = sum + temp;
        }
        // count no longer exists
}

// temp no longer exists, but sum still does
void incsum(void){
        sum++;       // modifies the global version, not the one in main
}
```

<u>Notes</u>:

- Most compilers place global variables (at the top) in known addresses.
- Automatic variables (in {} blocks) are placed on the stack and destroyed once the block is left.