Lecture 22:
C Programming 4 Embedded Systems

Today’s Goals

- Basic C programming process
- Variables and constants in C
- Pointers to access addresses

Using a High Level Language

- High-level languages
  - More human readable
  - Less dependent on processors
  - Less source code, generally
- 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 lectures
  - 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 we define constants in assembly.
- Declaring constants does not use any memory just like in assembly.
- The values defined are used during compiling source code.

is equivalent to

Basic Data Types

- Variables in C can be defined as either ‘signed’ or ‘unsigned.’
- In assembly, programmers have responsibilities to choose right version of instruction when using comparison instructions.
- In C, the compiler chooses the proper comparison according to the variable types (signed or unsigned).
- Data Types
  - char (character) – 1 byte or 8 bits
  - int (integer) – 2 bytes or 16 bits (note: it depends on the processor. We have 16 bit integer since we are using 16 bit processor.)
  - long – 4 bytes or 32 bits
  - ** You can use ‘unsigned’ before the data type if you want to explicitly use ‘unsigned’ data type.

Examples

- Convert the following C variable definition into assembly code.

C	Assembly
unsigned char count;	
char count;	
unsigned int rti_ints;	
long profit;	
unsigned char mylist[4];	

- Note:
  - The assembly declarations for signed and unsigned values are the same – no distinction is made.
  - Arrays in C use square brackets.
  - In assembly, a label represents the address of the value.
  - In C, a variable represents a value. But an array name (mylist) is the address of the first item in the array.

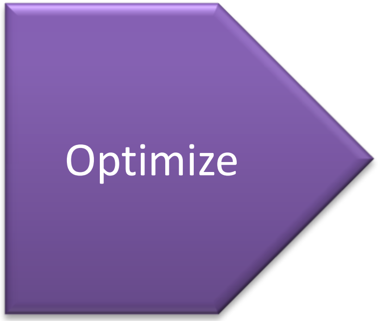
Volatile

Variables for input ports should be defined as volatile

- A variables declared to be volatile will not be optimized by the compiler because the compiler must assume that the value can be changed at any time.

```
int foo = 0;

void bar (void) {
    while (!foo)
        ;
}
```




```
int foo = 0
void bar (void) {
    while (1)
        ;
}
```

What if I have an interrupt service routine something like this?

```
interrupt void isr_foo (void) {
    if(button_pressed)
        foo = 1;
}
```

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;
…
DDRB = 0xFF;
```

- The one byte at address DDRB is changed. But.. Well.. This is not we want.

- In assembly, DDRB is used to refer to a control register at address 0x0003.

- How can we do this? Answer! Use pointers!!

Pointers in C

- In C, unlike assembly, a label (a variable) can represent either the value of a variable or the addresses of a variable.
- Pointers in C
  -
  -
  -
  -

Examples

- The following lines of code demonstrate how pointers function.
- Examples
  - `int* var1;`
  - `var1 = 0x1000;`
  - `*var1 = 0x1234;`

Definitions for I/O Ports

- In your C program, you want to use the same labels used in assembly program.
  - DDRB, PORTB, …
- Here is a way.
  -
  -
- Examples
  -
  -
- Anatomy of the definitions and usages
  -
  - `0xFF will be set to address 0x0003 as the content of it.`
  -
  -

Examples

```
#define PORTB    (*(char *) 0x0001)
#define DDRB     (*(char *) 0x0003)
#define PORTP    (*(char *) 0x0258)
#define DDRP     (*(char *) 0x025A)
#define PORTH    (*(char *) 0x0260)
#define DDRH     (*(char *) 0x0262)
#define PIEH     (*(char *) 0x0266)
#define PIFH     (*(char *) 0x0267)
```

- Set PORTB and P to all output:
  -
  -
- Wait for bit 0 of PORTH to be 1:
  -
- Enable the left most 7 segment display (bit 3 of PORTP to 0) and disable the other three digits without affecting other bits.
  -
  -
- Clear the flag bit for PORTP bit 7
  -

Variable Scope

```
int c; /* Global variable */

void foo()
{
   int c = 0;   /* Declared in outer block */
   do
   {
     int c = 0; /* This is another variable called c */
     ++c ;       /* this applies to inner c   */
     printf("\n c  = %d ", c );
   }
   while( ++c  <= 3 );     /* This works with outer c */

   /* Inner c  is dead, this is outer */
   printf("\n c = %d\n", c );
}
```

- All constants and variables have scope
  - In other words, the values they hold are accessible in some parts of the program, where as in other parts, they don't appear to exist.

- There are 4 types of scope:
  - block, function, file and program scope. Each one has its own level of scope.

Variable Scope

One more example

```
int sum; /* global variable */

/* temp is not available inside incsum */
void incsum(void){
    sum++;
}

Void foo(void)
{
    int temp = 3;
    sum = 0;

    for(int count=0; count < 10; count++)
    {
        sum = sum + temp;
    }
    // count no longer exists in some compilers
}
```

Questions?

Wrap-up

What we've learned

- C programming in embedded systems

What to Come

- More about C programming in embedded systems