Australian National University

Week 7
A review of C programming for embedded applications
Video Lecture 10

ENGN4213/6213

Digital Systems and Microprocessors

What is in this lecture

- We review some C essentials
  - Variables and constants
  - Simple I/O
  - Common C operations and constructs
  - Control instructions
  - Functions
  - Pointers
  - Arrays

Resources

- Your C primer and associated resources (there is a list of them on Wattle)
- https://www.learn-c.org/en/Welcome
- Topic quizzes in Wattle
- C programming for beginners
  - https://www.youtube.com/watch?v=KJgsSFOSQv0
  - Slides from ENGN2219 (if you have done the course)

Let's quickly learn and practice C with me

- Open the link
- https://www.learn-c.org/en/Welcome

  – In this lecture, we will concentrate to practice some basics C

  – In the Labs, you will be using C within the context of programming a micro-controller.

Hello world

- The best known basic program:

```
#include <stdio.h>

void main()
{
   printf("Hello World");
   while(1);
}
```

← A preprocessor directive

← A main function (all programs have one)

← Some basic I/O (default peripheral)

← A looping statement ?!?

- Even in such a simple program a first difference with programming as you have known it so far is evident.
- Embedded programs never "end". Rather, they usually ***loop indefinitely*** (often doing something useful, unlike the above). Once a processor executes a STOP instruction it won't do anything else. A loop keeps it active whilst idling.

The very basics

- `;` semicolons go at the end of most expressions
- `{ }` braces mark the beginning and end of a function or subsection (same as `begin/end` in Verilog)
- // or /* … */ are acceptable syntax for inline and block comments, respectively

- names of variables/functions:
  - Are case sensitive
  - Must not start with a number
  - Should not be more than 32 characters long (depends on the compiler)
  - Must not be a reserved keyword

- C is a “free-form” language: blank spaces are ignored by the compiler unless delineated by quotes.

Variable types

- Variables in C are associated with a variety of possible types
  - The type affects the quantity of memory allocated to contain the information
  - Variables are declared using a type identifier and a name, e.g. **`int i;`**

Type	Size (bits)	Range
`bit`	1	0,1
`char`	8	-128 to 127
`unsigned char`	8	0 to 255
`int`	16	-32768 to 32767
`unsigned int`	16	0 to 65535
`long int`	32	-2147483648 to 2147483647
`unsigned long int`	32	0 to 4294967295
`float`	32	±1.175e-38 to ±3.402e38
`double`	32	±1.175e-38 to ±3.402e38

Variable scope

- Local variables
  - Exist only locally with respect to a function. Are not accessible to other functions (limited scope). So for example one can declare a variable `int a;` inside multiple functions of the same program with no risk of conflicts.
- Global variables
  - A variable allocated in a memory space which can be accessed by all of the program functions (unlimited scope). A global variable is declared before the *main* function. It can be modified by any function and will retain its new value for other functions to use.

Variable scope – example

- Example:

```
unsigned char glob_var; //my global variable

void  myfunction(void) {
   int mylocalvar;              //a local variable
   mylocalvar=123;     //this assignment is ok: local variable
   glob_var=101;       // this assignment is ok: global variable
}

void  main () {
   mylocalvar=123;     //ERROR: mylocalvar is not local to main
   glob_var=101;       //this assignment is ok: global variable
   while(1) … ;        //loop forever
}
```

Constants

- Declared in the same way as variables, their value does not change throughout the program.
  - Sometimes the keyword const can be used, e.g., `const float pi=3.14159;`
  - A constant can be stored in the program memory (ROM). Variables, on the other hand, are saved in volatile memory (e.g., RAM.)
- Numeric values for constant can be expressed in different systems:
  - Decimal (number without prefix, e.g. `int a=32;`)
  - Hexadecimal (number with prefix **`0x`**, e.g. `int h=0xff;`)
  - Binary (number with prefix **`0b`**, e.g. `int b=0b11101100;`)
- Character constants use quotation marks and have an alternative hexadecimal representation, e.g.,

  `'t'` is equivalent to `'\x74'`

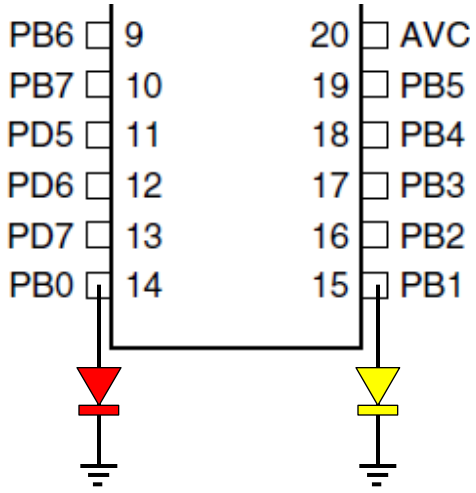
Enumerations and definitions

- Enumerations are used to create a set of constants which contain integers in sequence. This can be useful for readability of the code.
  - Example: **`enum`**`{numbernil, number1, number2, number3};` is the same as writing `numbernil=0; number1=1; …`
  - Another example: **`enum`**`{first=10, second, last};` means that *second* will equal 11 and *last* will equal 12.
- Definitions rely on the preprocessor directive `#define`.
  - Syntax: **`#define <termA> <termB>`**
    (followed by *no notes or semicolon*)
  - They inform the preprocessor that all instances of <termA> sould be treated as instances of <termB>.
  - This can be particularly useful when certain identifiers in the program code can be assigned names which can be more easily identified and/or remembered.

Enumerations and definitions – example

- Note: PORTB is an identifier from AVR libraries which identifies the I/O register for portB on the microcontroller chip. Note that, usually, capitalised names are used for identifiers provided by libraries. Setting values of 0 and 1 to the various digis of this register means forcing a certain output from the µC.

```
…
enum {red_led=1, yellow_led, both_leds};
#define leds PORTB
…
…
PORTB=0x1; //turn red LED on
leds = red_led;
   //completely equivalent statement
…
```



Storage classes

- Automatic
  - An automatic variable (syntax **`auto int v;`** or simply **`int v;`**) is the standard type.
    - Allocated as uninitialised when called
    - Memory space released once the program exits the called function
- Static
  - A static local variable is a local variable but it is allocated in the memory space. This means that when a function is called again, it can retrieve a value saved in the global memory during a previous call.
  - Example of syntax: **`static int statvar;`**
- Register
  - Similar to a standard auto variable but the memory is reserved in the faster working registers, if they are available. This can be forced to improve speed in critical application. Use sparingly as registers are few.
  - Example: **`register char value_3;`**

Type casting

- Type casting means forcing the program to treat a variable as if it were of a different type for the purpose of a certain operation.
  - Syntax: `(<cast_type>)var_name`
- Type casting is particularly important in arithmetic between variables of different type, as the accuracy is determined by the variable of the smallest size. So, for example, an operation between an integer and a char may not give a correct result for numbers greater than 127 unless type casting is used.
  - Example:

```
char y=78; int z;
z=y*10;
```

The result for this operation is 12 (check the overflows for yourself), not 780. Using typecasting `z=(int)y*10;` allows for the computation of the correct result.

Operators and expressions

- ***Arithmetic operators*** (C follows standard precedence rules):

Multiplication	`*`
Division	`/`
Modulo	`%`
Addition	`+`
Subtraction	`-`

- ***Bitwise operators***. Imagine these as combinatorial operations between words of n-bit length. The result is also n-bit long.

Negation (complement)	`~`	`~(1101) = 0010`
Left shift	`<<`	`1101 << 2 = 0100`
Right shift	`>>`	`1101 >> 2 = 0011`
AND	`&`	`1100 & 0101 = 0100`
OR, XOR	`|, ^`	`1100	0101 = 1101`

Operators and expressions (2)

- ***Logical operators.*** Note that these are different from bitwise ones.

AND	&&
OR	||

These operators deal with the operands by interpreting them as TRUE (equivaent logic 1) if the operand is non-zero, FALSE (equivalent logic 0) means that the operand is zero.

- ***Relational operators***. Used in comparisons, they return a 1 if TRUE or a 0 if FALSE.

Equal to	==
Not equal to	!=
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=

Incrementing, decrementing and compound assignments

- C has some unique syntax options. These not only are quick to write (compound assignments) but are also directly related to the underlying Assembly instructions for the processor (incr/decr)
- ***Increment/decrement operators***
  - Allow an identifier to be modified in a pre-increment or post-increment manner. For example, **`x=x+1;`** can be written as **`x++;`** or **`++x;`**. What's the difference?
  - The former is a *post-increment* and the latter is a *pre-increment.* The following example is explanatory:

```
i = 1;
k = 2 * i++; //after the operation is complete, k=2 and i=2
i = 1;
k = 2 * ++i; //after the operation is complete, k=4 and i=2
```

  - A similar reasoning can be conducted with pre- and post decrements operators `x--;` and `--x;`

Incrementing, decrementing and compound assignments (2)

- ***Compound assignment operators***
  - By combining the assignment operator `(=)` with an arithmetic or logical operator, C provides a syntax option which saves time.
    Some examples are shown below

```
b += 3;  //same as b = b + 3;
q *= 4;  //same as q = q * 4;
x /= 1;  //same as x = x / 1;

y &= 3;  //same as y = y & 3;
s ^= 4;  //same as a bit-by-bit XOR between s and 4;
```

Operator precedence

- For your reference (if ever in doubt use parentheses!)

Name	*Level*	*Operators*
Primary	1 (High)	() . [] ->
Unary	2	! ~ - (type) * & ++ -- sizeof
Binary	3	* / %
Arithmetic	4	+ -
Shift	5	<< >>
Relational	6	< <= > >=
Equality	7	== !=
Bitwise	8	&
Bitwise	9	^
Bitwise	10	|
Logical	11	&&
Logical	12	||
Conditional	13	? :
Assignment	14 (low)	= += -= /= *= %= <<= >>= &= ^= |=

Control statements

- As the name suggests, they control the flow of the program
- Most students should be already comfortable with these from previous programming work, including Verilog.
- We will touch on them very quickly as a cursory look should be sufficient.

- `while`
- `do… while`
- `for`
- `break / continue`
- `if / else`
- `switch / case`

while

- Executes the content of a loop for as long as the condition inside the argument remains true.
- Syntax:

```
while (condition)
{
    expressions;
    …
}
```

- Example:

```
while (PINB & 0x02)      // hangs for as long as the
          ;              // second bit of port B remains
                         // high
```

do… while

- Essentially the same as `while`, except the loop is executed at least once even if the looping condition is not met.
- Syntax:

```
do
{
    expression1;
    expression2;
    …
} while (condition);
```

for

- A construct generally used to execute one or more statements a required number of times. A **for** loop comprises an initialisation (*expr1*), a test (*expr2*) and a looping action which eventually leads to the test condition being met (*expr3*)

- Syntax:

```
for (expr1;expr2;expr3)
{
    expressions;
    …
}
```

- Example

```
for (i=1;i<=10;i++)    //outputs the value of i
    printf("%d",i);    //throughout the 10 iterations
```

if / else

- A basic branching operation which offers two alternative execution pathways depending on whether a test condition is met or not.

- Syntax:

```
if (condition) {
    expressions;
} else {
    other_expressions;
 }
```

- Example

```
if (thief_sensor) alarm=1;
else alarm=0;
```

An embedded example

```
#include <AVR.h>
#define test_port PORTD
void main(void)
{
    unsigned char count, bitmask; //variables
    bitmask=1;

    for (count=0;count<=7;count++) //for loop to test 8 bits
    {
         if (test_port & bitmask)
              printf("Bit %d is high\n", (int)count);
         else
              printf("Bit %d is low\n", (int)count);

         bitmask <<= 1; //shift bit for testing
    }
    while(1); //this just sits here and does nothing
}
```

switch / case

- A construct used to create a multiple branching based on an expression which can return multiple values *constX* (not just true/false as in `if/else`).
- Syntax:

```
switch(expression)
{
        case const1:
                statement1;
                statement2;
        case const2:
                statement3;
                …
                statement4;
        …
        default:
                statementx;
                …
}
```

switch / case (2)

- Example:

```
unsigned char sel;
sel = PINC & 0x0f;//selector based on the lower nibble of PORTC
switch(sel)
{
     case'0':
     case'1':    //I believe this is called "fall-through": a
     case'2':    // number of cases lead to the same outcome
     case'3': printf("temperature OK");
              break;
     case'4':
     case'5': printf("temperature high");
              break;
     default: printf("temperature critical");
}
```

- In a **`switch`** statement, *all statements after the matching constant are executed*. Note the use of **`break`** statements. They are required in order to confine execution to the code associated with the verified state alone.

`break` / `continue`

- Two keywords to stop the execution of a loop before completion of the current iteration. Both can be used in combination with any looping construct (`while`, `for`, …).
- **`break`** forces the program to "step out" of the loop. If you are working with nested statements, only the current level will be exited (any higher-level loops will continue to run).
- **`continue`** forces the program to jump immediately to the beginning of the next loop iteration, skipping the remainder of the current one.

break / continue (2)

- Example (`break`)

```
int c;
while(1)
{
   while(1)
   {
        if (c>100)
           break;
        ++c;
        printf("%d\n",c);
   }
   c = 0;
}
```

- Example (`continue`)

```
int c=0;
while(1)
{
   while(1)
   {
        if (c>100)
           continue;
        ++c;
        printf("%d\n",c);
   }
}
```

Functions

- Sections of code which share a common task and can be separated from the *main* for
  - Improving ease of debugging
  - Reusing code multiple times

- An important concept in relation to functions is that of *libraries*, i.e., collections of functions which relate to the same application which can be shared amongst programmers
  - Improves software reliability (functions are tested over and over)
  - Reduces development time

Functions (2)

- The standard format of a C function is:

```
type function_name (type param1,
   type param2, …)
{
   statement1;
   statement2;
   statement3;
        …
}
```

An example:

```
char btn_in(void)
{
   while((PIND&0x1)==0)
        ;
   return 1;
}
```

- A function (and its parameters for that matter) can be of any variable type (e.g., `int`, `char`, etc.). The function type indicates what type of value the function will return to its caller. If the function does not take any parameters or does not return a value, the type `void` should be used. The default type for a function is `int`.

Function prototyping

- Just as is required for variables, in order for functions to be used they need to be declared first. There are two ways to do so:
  - Writing all functions ahead of the *main*, in order of call. However, if the program is complex, this might be impossible.
  - Using *function prototypes.* A function prototype is a one-line description of the function (its "header"). This informs the compiler of the existence of each function and the type and number of inputs/outputs it is associated with. The actual code for the function can then go at the bottom of the file, even past the *main.*
- The next slide shows the typical code structure for a program which uses function prototyping. Take notice of the syntax.

Function prototyping (2)

```
int function1(int, int); //prototype for function1

void function2(void); //prototype for function2

int var1, var2, var3;  //declaration of global variables

void main (void)      //main
{
        var3=function1(var1,var2);
        function2();
}
/*the function declarations can now go at the bottom
since the prototypes have been used above*/

int function1(int x, int y)
{/*function content here*/}

void function2(void)
{/*function content here*/}
```

The `return` instruction

- Stops the execution of the called function and returns control to the caller. For this reason, it is generally placed *at the end* of a function/subroutine.
- In a function of type `void`, it is not required for `return` to be expressed explicitly (the return event occurs after the function has completed its execution)
- In functions which return a value, `return` is required.
- Functions which return a value can be used as part of an expression, for example:

```
float cube(float v)
{
   return (v*v*v);
}

//and further down in the main…
vol_sphere=4/3*3.14159*cube(r);
```

Recursion

- A very powerful feature of C which allows to write very efficient programs with only a few lines of code. Recursion means writing a function that can call itself.
- This technique must be used carefully as it can easily cause the computer to run out of memory.
- Here's a typical example: ***computation of factorials***.
- Hopefully you remember factorials from Mathematics: the factorial of a positive integer number is the product of all of the positive integers which are ≤ than the argument of the factorial operation.
  For example

$$4! = 4 * 3 * 2 * 1 = 24$$

Recursion (2)

- Here is the code

```
int fact(int n)
{
   if (n==0)  return 1;          //have a think about how this works
   else  return (n*fact(n-1));
}

void main (void)
{
int n;
n=fact(4);                       //computes 4! recursively
printf("4 ! = %d",n);            //prints the result
while(1);                        //finished: idle
}
```

Arrays

- Arrays are special variables which can hold more than one value under the same variable name, organised with an index.

```
/* defines an array of 10 integers */
 int numbers[5];

/* populate the array */
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

Notice that arrays in C are zero-based, which means that if we defined an array of size 5, then the array cells 0 through 4 (inclusive) are defined.

https://www.learn-c.org/en/Arrays

Arrays can only have one type of variable, because they are implemented as a sequence of values in the computer's memory.

Pointers

- They allow programs to perform operations in a more generalised way and often more efficiently.
- Pointers are a bit obscure at first, but they can be understood in simple terms. You must stop thinking of variables as "values" but rather as "memory locations containing values". Pointers are *an explicit way of working with indirect memory addressing*.
- Introducing **two new operators**:
  - The ***indirection or dereferencing operator:*** ***
  - The ***address operator:*** **&**
- https://www.learn-c.org/en/Pointers
- https://www.youtube.com/watch?v=KJgsSFOSQv0 (explain pointers well – easy to understand)

Pointers (2)

- Declaration:

```
char *p;    //p is the pointer to a character
int *ip;    //ip is the pointer to an integer
```

- A pointer is *a variable that contains the address of a memory location*. In the simplest case, this location contains a data variable. Note that although variables can be different bitsizes, all pointers have the same size.

- Once the pointer is declared, it must be assigned a value. After that, it can be used to refer *indirectly* to that data location.

```
char a, b, c;
p=&a;    //p is now pointing to a
c=*p+b; //achieves the same result as c=a+b!
*p=b;    //achieves the same result as a=b;
```

Common errors with pointers

- Whenever you see the symbol * you should get used to thinking "*the memory location pointed to by* …" and whenever you see & you should think *"the memory address of…".* Terrible mistakes can be made by using pointers. E.g.,

```
char a=0, b=0;
char *p;
p = a; //p is assigned the value of a, points to mem loc 0x0000!
b = *p; //this will not achieve b = a!
*p = a + b + 1; //this will likely corrupt the memory!

----

char a=0, b=0;
char *p;
p = &a; //this is now correct
b = p; //this will not achieve b = a!
```

- The "wrong" assignments above *are not illegal* in C. So the compiler will not complain. But they might cause your program to badly malfunction!

Pointer arithmetic

- Why do pointers need to have different types if they are all the same bitsizes?
- A pointer points to the memory address where the variable begins, but the *length of the data variable* is defined by the pointer type
- This is useful in ***pointer arithmetic***. Pointers can be the subject of basic arithmetic operations. The result of the operation will see the address pointed at moved by a quantity correspondent to the type. For example:

```
int *ptr;
long *ptr2;

ptr = ptr + 1; //moves pointer to next integer (2 bytes forward)
ptr2 = ptr2 - 1; //moves pointer to previous long (4 bytes back)

ptr++;       //these syntax options can also be used
--ptr2;
```

Pointer arithmetic

- Why do pointers need to have different types if they are all the same bitsizes?
- A pointer points to the memory address where the variable begins, but the *length of the data variable* is defined by the pointer type
- This is useful in ***pointer arithmetic***. Pointers can be the subject of basic arithmetic operations. The result of the operation will see the address pointed at moved by a quantity correspondent to the type. For example:

```
int *ptr;
long *ptr2;

ptr = ptr + 1; //moves pointer to next integer (2 bytes forward)
ptr2 = ptr2 - 1; //moves pointer to previous long (4 bytes back)

ptr++;       //these syntax options can also be used
--ptr2;
```

Summing up

- We have reviewed some C essentials
  - Variables and constants
    - Types and casting, scope, enumerations, definitions, classes
  - Common C operations and constructs
    - Operators, C-style compound assignments
  - Control instructions
    - Loops and branching
  - Functions
    - Call/return, prototyping, recursion
  - Pointers