

Module Introduction

Lecturer: Professor Ronan Reilly
Materials: Dr Rudi Villig
Maynooth University

Online at <http://moodle.maynoothuniversity.ie>

EE108 – Computing for Engineers

1

1

Module descriptor summary

2

□ Module Aims

- to learn C programming in an embedded systems environment

□ Module learning outcomes

- Design, code, and debug small C programmes
- Decompose a problem into sub-problems and implement a solution using the most appropriate code features
- Use and manipulate C pointers and arrays
- Read and then explain the operation of a C programme
- Explain key C programming concepts

2

2

Syllabus / topics covered in this module

3

□ Part 1

- Introduction to C and fundamental elements of all programmes
 - expressions, statements, statement blocks and function blocks
- Primitive data types and expressions in C
 - Differentiating constants and variables, local and global scope
- Selection (conditional) statements in C
 - If, nested if, cascaded if, switch, ternary operator
- Iteration statements in C
 - While, do-while, for
- Functions in C
 - Declaration, implementation (definition), function call and return
- Programme design
 - Decomposing a problem

27 September 2020

3

Syllabus / topics covered in this module

4

□ Part 2

- Arrays in C
- Scalar pointers
 - How primitive types are stored in memory, address-of operator, dereference operator
- Array pointers
 - Relationship between array subscripting and pointer arithmetic
- Strings in C (string pointers and null terminated character arrays)
- The const keyword as it applies to pointers
- More advanced (but essential) types
 - Enum, struct, typedef, union

27 September 2020

4

Assessment

5

□ Module elements and assessment

- 22 x 1 hr lectures
- 2 x 1 hr class tests (32% of module marks)
- 4 x 10 hr assignments (20% of module marks)
- 8 x 3 hr labs (12% of module marks)
- 2 x 3 hr lab tests (36% of module marks)

□ This is a 100% continually assessed module

- **no exam but also no chance to repeat in Autumn**

*Definitive module descriptor available at
<https://www.maynoothuniversity.ie/electronic-engineering/current-students/module-descriptors>*

27 September 2020

5

To succeed...

6

□ **All** material we discuss is part of the course

- All examples given in lectures, labs, or online are relevant (e.g. to tests)
- Extra explanations given in class/online but not in the notes are also relevant



You should take notes or mark up your notes in every class

(learning to do this well is a key to success in university and elsewhere)

□ If you don't understand something, ask (during class or in online forum)

□ **Beware!**

- Penalties for late submission of assignments
- There is generally no facility for catch up on missed assessments (unless absence is certified, e.g. medical cert)
- As normal, continual assessments can not be repeated

27 September 2020

6

1.00 Introduction to C programming (and embedded systems)

EE108 – Computing for Engineers

7

Overview

8

□ Aims

- Briefly introduce embedded systems and key features of the C programming language

□ Learning outcomes – you should be able to...

- Explain the concept of an embedded system
- Give simple examples of an embedded system
- Recognise the outline/template for Arduino C programmes
- Recognise expressions, statements, statement blocks and function blocks
- Use a standard coding style and recognise deviations from the coding style

27 September 2020

8

About the C language

9

□ Origin

- General purpose language developed by Kernighan and Ritchie between 1969 and 1973 along side Unix operating system
- Generally replaces low level assembly language

□ Most common descendants

- C++, Java, C#, JavaScript among others
- The expression and statement syntax of C is widely used

□ General purpose language, but it's main application areas today are

- System level computing, OS and device drivers (e.g. Unix, Linux, Windows, etc.)
- **Embedded systems**

27 September 2020

9

What is an embedded system?

10

Embedded system:

(a) a computing system designed to perform a dedicated task

(b) The computing system is embedded within a system or device which is not itself a computer



BMW 745i
About 70 embedded systems

Washing machine
1 embedded system



Aldebaran NAO
About 19 embedded systems

27 September 2020

10

Arduino based embedded systems

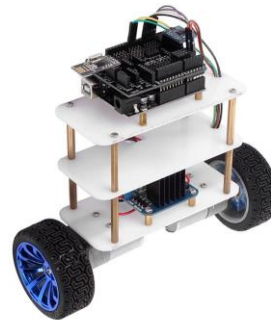
11



Arduino + EE108 daughterboard



Arduino powered garden control, reproduced from *makezine.com*



Arduino self balancing robot, reproduced from *Sain Smart*

27 September 2020

11

About the development environment for EE108

12

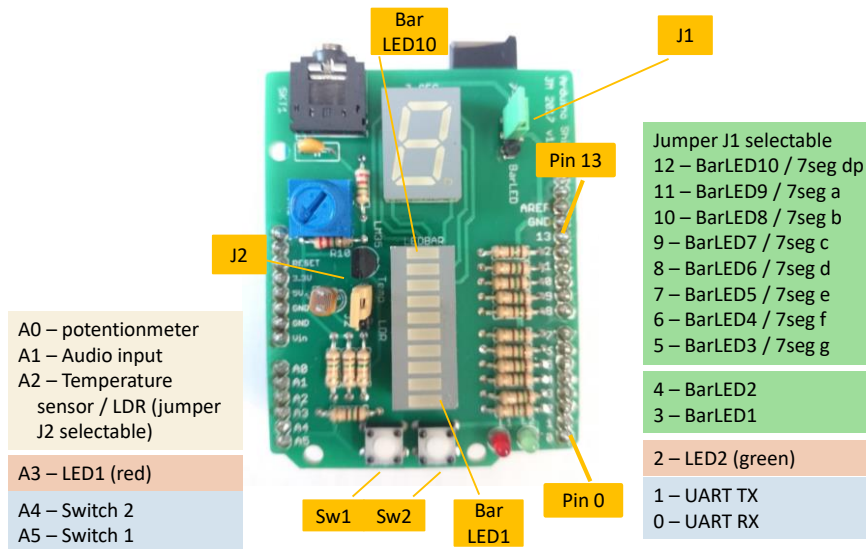
- We will use the Arduino Uno + EE108 Daughterboard for labs and assignments
 - The daughterboard provides some basic inputs and outputs (primarily buttons and LEDs) used in labs and assignments
- The Arduino Uno is a board based around the ATmega328P **microcontroller**
 - 8 bit core (vs. 32 bit for smartphones, 64 bit for desktops)
 - Unlike a general CPU, other useful hardware is integrated onto the chip package with the core
 - Includes flash memory, RAM, general purpose input output ports, an analogue to digital converter (ADC), a communications interface, timers, etc.
 - Makes it easier to build intelligent devices and systems (particularly **sense-think-act** systems)
 - It has limited memory so programmes must not be wasteful
 - Just 2KiB bytes of RAM (for variables), Just 32KiB for programmes (compiled code)
 - This is why C is the language of choice

27 September 2020

12

EE108 daughterboard pin assignments

13



27 September 2020

13

Arduino capabilities used in EE108

14

□ Digital input/output

- Arduino Uno has 14 general purpose digital input/output pins available (11 used in EE108 for LEDs and switches)

□ Analog input

- Arduino Uno has 6 “analog input” pins available (3 used as analog inputs – potentiometer, temperature sensor/LDR, audio jack; 3 used for digital I/O in EE108)

□ Serial communications

- Hardware UART available for communication with console on PC

□ Additional Hardware on daughterboard

- 2 active-low momentary switches
- 2 general purpose LEDs, jumper selectable BarLED or 7 segment display (only one at a time)
- Potentiometer, audio input, jumper selectable temperature sensor or light sensor (LDR)

27 September 2020

14

More about the C language

15

□ The C language is standardized by ANSI and ISO

- Most recent C standard is C11 (from 2011)
- **The version of C used in EE108 is C89 (from 1989)** and is the most widely supported version across many systems

(Strictly, Arduino C is actually a simplified variant of C/C++ based on Wiring [see wiring.org.co]. It includes some features beyond C89, but we'll try to restrict ourselves to C89 where possible for maximum portability)

□ C is a permissive low level language

- Powerful, efficient access to all features of the hardware
- Assumes that you know what you are doing
- Can be difficult to use and error prone if you don't !

27 September 2020

15

Contd.

16

□ C is a small language

- Just 32 keywords (but many operators which can complicate things)
- Original reference book was just 49 pages

□ C is a procedural language

- It is not object oriented and there are no objects in C
(Arduino does use some objects but we'll mostly ignore this)
- C can simulate some useful traits of object oriented if necessary

□ C compilers are fussy!

- Uppercase/lowercase matters
- Terminating a line of code with a semicolon matters
- Terminating comments correctly (and not nesting them) matters

□ C compilers do not always seem very helpful

- Error messages can be difficult to figure out, particularly at first

27 September 2020

16

Programme building blocks

17

27 September 2020

17

Getting started with programming

18

□ Programming is...

- simply telling a computer what to do
- If you can explain to someone in detail how to do some task, then you have at least some idea of how to programme a computer

□ Computers are stupid

- A computer will do exactly what you tell it – no more and no less
- It cannot fill in the blanks or do what you intended rather than what you actually said
- One of the challenges of programming is realising just how little a computer “knows” and therefore how to “explain the task” in terms the computer can understand

27 September 2020

18

Programming and abstraction

19

- **Computers are complex (despite being stupid)**
 - Substantial computer programmes may be thousands or millions of lines of code (e.g. flight software for mars probes is about 3 million lines)
- **People depend on abstraction to work with complex systems like this**

See also <http://www.eskimo.com/~scs/cclass/progintro/sx1.html>

 - People can only keep a few things in their head at once
 - The solution is to compartmentalize the system (divide and conquer) so that we can think about small pieces in isolation, e.g.
 - You want the computer system to light a LED when a button is pressed
 - How do you check for a button press?
 - How do you light a LED?
- **You need to pay attention to detail and assumptions**
 - In previous example: Which button do you mean? Which LED do you mean? Should the LED stay on while the button is pressed or just flash briefly? Etc.

27 September 2020

19

To programme successfully

20

- **Divide and conquer**
 - Break big problems into smaller ones
 - Use **encapsulation** – hide the details of low level solutions from the higher level problem wherever possible
- **Be specific and complete** – remember the computer is stupid
 - Tell the computer everything it must do and exactly what it must do
- **Pay attention to detail** and beware of your assumptions
- **Write code that is as readable as possible**
 - People (including yourself) need to read and understand the programmes you write
 - Consider code to be write once (you), read many times (you, and maybe others), so write it for readability



27 September 2020

20

Programme building blocks

21

- Programmes in C (and many other languages) are composed of the following building blocks
 - Expressions
 - Statements
 - Statement blocks
 - Function blocks

27 September 2020

21

Expressions

22

```
// EXAMPLES of expressions: the following are all expressions (and not valid
// C statements on their own)

a + b           // suppose a is 1 and b is 1, the expression value is 2
a + b * 2 + c * 5 - d
a - 3000
big < small     // if small is 1 and big is 2, the expression value is 0 (false)
c && d
```

An expression is sequence of operators and operands that specifies a computation yielding a value

- An **operator** is a mathematical or logical action (e.g. add, multiply, and)
- An **operand** is a bit of data acted on by an operator (e.g. in the expression $a+b$, both a and b are operands)
- **Note: An expression is not complete on it's own – it always appears as part of a statement**

27 September 2020

22

Expressions: Operator precedence

23

- The order of calculations in a complex expression (with multiple operators) is determined by **operator precedence rules**
 - Arithmetic operator precedence is as you would expect
 - However, you will have to learn and remember (or constantly look up) the precedence for relational, logical, and other operators

***Good practice:** don't rely on the implicit precedence rules—instead add parentheses to make the order you want explicit (the expression inside the innermost parentheses is always evaluated first)*

```
// EXAMPLE: suppose a is 2 and b is 5
// is the value of the next expression 17 (i.e. 2 + 15) or 21 (i.e. 7 * 3) ?
a + b * 3      // you must know the precedence rules to work out the value
a + (b * 3)    // parentheses make the precedence explicit and clear
```

27 September 2020

23

24

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Associativity determines the order in which the equal precedence operators in each row are applied

We'll see operator precedence again later

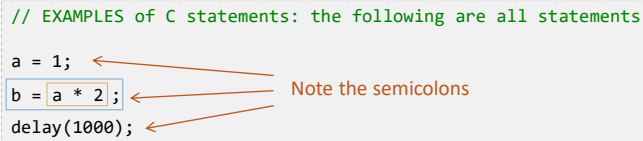
27 September 2020

24

Statements

25

```
// EXAMPLES of C statements: the following are all statements  
  
a = 1;  
b = a * 2;  
delay(1000);
```



Note the semicolons

A statement is the smallest complete instruction in C—it specifies some action to be carried out

- A programme is nothing more than a sequence of statements
- Statements are used to assign values to variables, call functions, select from different courses of action based on conditions, etc.
- In C, all simple statements must end with a semicolon—leaving it out results in something that is not a statement which either confuses the computer or doesn't work at all

27 September 2020

25

Statement blocks / compound statements

26

A statement block is one or more statements grouped together by curly braces in C

- Statements between the opening and closing brace form the **body of the statement block**
- A statement block can be used anywhere that a simple statement can be used—this is most commonly useful in conditional (if, switch) and loop (for, while, do-while) constructs
- Statement blocks have other more advanced properties to be examined in later notes

```
// EXAMPLE: a statement block (as part of an if-statement)  
  
if (buttonState == HIGH) {      // block begins with opening brace  
    digitalWrite(RED_LED, HIGH); // first statement in statement body  
    digitalWrite(GREEN_LED, HIGH);  
}                               // block ends with closing brace
```

27 September 2020

26

A function block is a reusable statement block that has been given a name and is designed to solve a particular problem

- ❑ Functions are the primary feature of procedural languages such as C used to implement abstraction and the divide and conquer strategy for writing programmes
- ❑ A good function is a “black box” which specifies the kind of inputs you must supply and the kind of outputs and side effects you can expect in return. Internal details of how it produces the outputs or side effects should be encapsulated (hidden).
- ❑ Library functions are already written by someone else so that you can simply call them, e.g. some Arduino library functions include: `delay`, `digitalRead`, `digitalWrite`, `analogRead`, etc.
- ❑ You can (and should) also write your own functions that you can then call as many times as you like to avoid unnecessary code repetition and help with abstraction and encapsulation

27 September 2020

27

❑ Function implementation/definition

- ❑ The function definition specifies the name of the function, any parameters (mostly inputs) and the return type (what the function will evaluate to)
- ❑ The body of the function definition, (AKA the function body - between the opening and closing brace) is the code that will be executed when the function is called/invoked

```
// EXAMPLE: function definition
// This function takes no parameters (indicated by the empty parentheses
// after the function name) and returns nothing (indicated by the "void"
// before the function name).

void flashLed() {                // opening brace = start of function body
    digitalWrite(RED_LED, HIGH); // switch on the red LED
    delay(100);                  // delay for 100 ms
    digitalWrite(RED_LED, LOW);  // switch the red LED off again
}                                // closing brace = end of function body
```

27 September 2020

28

□ Function call

- A function call statement causes the code of the named function to be executed. It can be called wherever needed, facilitating code reuse.

```
// EXAMPLE: function call  
  
// the following statements cause the code in the flashLed function body  
// to be executed twice  
flashLed();  
flashLed();
```

- Note that to call the function you do not need to know the details of the function body (i.e. how exactly the function does its job)
- You only need to know the function name, the parameters that it needs and type of value that it will return (if any).

27 September 2020

29

First programme overview

- For embedded systems, the typical outline is:
 - Initialise the system (hardware devices, initial variables, etc.)
 - Then repeat the some set of actions forever (until powered down)
- Arduino-C strongly guides you towards this outline by defining two empty functions that you must fill in
 - **setup** — this function runs once each time the arduino board boots up and you should use to initialise the system and any hardware you need
 - **loop** — this function is called by the arduino framework repeatedly forever. The code you write in this function should be designed to run repeatedly. Each time the function returns, it is immediately called again.

27 September 2020

30

First programme overview - internals

31

- In reality C programmes generally start with an entry point function called main. The Arduino framework hides this detail and defines this main function so that it calls the setup and loop functions that you write.

In arduino-C, you don't explicitly write the following code (but it is there)

```
// Arduino framework code
// (hidden from you and you
// don't modify this)

int main(void) {
    ...

    setup(); // call your setup function

    for (;;) { // loop forever
        loop(); // call your loop function
    }
    ...
}
```

.../Arduino/hardware/arduino/avr/cores/arduino/main.cpp

Instead you just write the code for the setup and loop functions

```
void setup() {
    // put your setup code here.
    // It runs once when the
    // board boots up
}

void loop() {
    // put your main code here.
    // This function is called
    // repeatedly forever.
}
```

27 September 2020

31

First programme

32

- Since Kernighan and Richie (who designed C) most books start with the “Hello World” programme
- However our embedded system environment differs from the desktop in important ways
 - No operating system – our programmes will run on the “bare metal”
 - No keyboard, display, file system, standard input or output, etc.
- For our first programme, we'll blink a LED...

27 September 2020

32


```

/** Blink
 * Turns on an LED on for one second, then off for one second, repeatedly.
 */

const int LED1 = A3; // make it easier to remember LED pin assignment

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin connected to LED1 as an output.
  pinMode(LED1, OUTPUT);
}

// the loop function is called by the Arduino framework repeatedly, forever
void loop() {
  digitalWrite(LED1, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for 1000 ms (1 second)
  digitalWrite(LED1, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);              // wait for 1000 ms
}

```



Embedded programmes are generally designed to run forever (until loss of power or failure) running the same code repeatedly. This approach is known as the **superloop architecture**.

The Arduino environment hides some details from you (encapsulation again) and deals with calling the **loop** function repeatedly so that the body of this function is repeated forever

27 September 2020

33

Writing clear readable code

1. Standard programme template / file layout
2. Indentation and whitespace
3. Braces
4. Comments
5. Naming of identifiers

27 September 2020

34

1. Standard template for Arduino-C programmes

35

```
includes...

definitions and declarations...

void setup() {
  // put your setup code here, to run once (after system resets)
  statements...
}

void loop() {
  // put your main superloop code here
  // this loop function will be called repeatedly by the Arduino framework
  // (until power down)
  statements...
}

other functions that you define...
```

27 September 2020

35

2. Indentation and white space

36

- ❑ C rarely requires any white space between elements – people, however, do!
- ❑ Put a single space after every comma or semicolon
- ❑ Put a couple of spaces between a statement and a comment on same line
- ❑ Avoid multiple statements on a single line
- ❑ Indent in multiples of 4 (or maybe 2) spaces
 - ❑ **Don't use tabs** as they don't display consistently on different machines
 - ❑ **4 space indentation** is the most common indentation used by programmers today (although 8 was more used in the past)
 - ❑ In the notes I'll indent by 2, but only because it helps fit code on the slides. (Arduino also indents by 2)
 - ❑ Indent consistently (all indented blocks must line up!)

27 September 2020

36

3. Braces and indents

37

Recommended style (K&R / 1TBS): Opening brace on the same line as controlling statement (if any), nested statements indented, closing brace outdented

```
if (speed < THRESHOLD) { // opening brace on same line as controlling statement
    a statement indented 4 spaces...
    another statement indented to same level
} // closing brace outdented to same level as initial controlling statement
```

Acceptable style (Allman): Opening brace on following line to controlling statement (if any) at same indent, nested statements indented, closing brace outdented

```
if (speed < THRESHOLD)
{ // opening brace on new line, same indent as controlling statement
    a statement indented 4 spaces...
    another statement indented to same level
} // closing brace outdented to same level as opening brace
```

Many students prefer to use the Allman style and that's fine.

Important: (1) **Do not mix and match styles.** (2) The opening/closing braces must never be indented relative to the controlling statement

See also https://en.wikipedia.org/wiki/Indentation_style

27 September 2020

37

Brace/indent examples

38

K&R style

```
void someFunction(...) {
    if (expr...) {
        statements...
    } else {
        statements...
    }

    for (expr...) {
        statements...
    }

    do {
        statements
    } while (expr...);
}
```

Allman style

```
void someFunction(...)
{
    if (expr...)
    {
        statements...
    }
    else
    {
        statements...
    }

    for (expr...)
    {
        statements...
    }

    do
    {
        statements
    } while (expr...);
}
```

Many students prefer to use the Allman style.

27 September 2020

38

Brace/indent anti-patterns – do not do this

39

```
if (expr...) {  
    statements...  
}  
else {  
    statements...  
}
```

Attempting to use K&R style, but added extra newlines – **do not do!**

```
void someFunction(...)  
{  
    if (expr...) {  
        statements...  
    }  
}
```

Attempting to use Allman style, but indented opening brace and then did not indent the nested if-statement – **do not do!**

Indented the opening brace – **do not do!**

The nested statements are indented correctly

See also https://en.wikipedia.org/wiki/Indentation_style

27 September 2020

39

4. Comments

40

- Use comments as needed to explain what your code is trying to do
 - C programmes are often difficult to understand without comments
- C89 comments (and one more modern variant)

```
/* This is a multi-line comment. Everything between the start of comment  
   marker and end of comment marker is ignored by the compiler  
*/  
  
/* This is the single line comment style supported by C89 */  
  
// This is a single line comment in C++, C99, Arduino-C, but not C89.  
  
// In double slash comment everything up to the newline is ignored  
  
// Nesting comments...  
  
/* /* this is an illegal nested comment - why??? */ */  
/* // but this is a legal nested comment - why??? */  
// // this is also legal  
// /* and this is also legal */
```

27 September 2020

40

5. Identifiers

41

Identifiers are the names given to constants, variables, and functions

C has no particular requirements except that the spelling must be absolutely consistent and case matters

Recommended naming style (required for EE108!):

variable and function names use **mixed** case

(start with a lowercase letter, combine multiple words using internal capitalization and avoid underscores). E.g.

someLongVariableName, someLongFunctionName

Constants use all capitals separated by underscores

THIS_IS_A_CONSTANT

Alternative style: constants as above; variables and functions may be named using all lowercase separated by underscores, e.g.

some_long_variable_name, some_long_function_name

Don't mix and match styles and don't use this style in EE108

27 September 2020

41

Contd.

42

□ Try to use self explanatory names

- it can also save the need to add comments

□ Don't make names unnecessarily short

- Names with 8-20 characters are best

Gorla, N., A.C. Benander, and B.A. Benander. 1990. "Debugging effort estimation using software metrics." IEEE Transactions on Software Engineering SE-16, no. 2 (February): 223-31.

- Remember code is "**write once, read many times**"

```
c = c + 1; // increment the counter    what counter? What does this even mean?

// in the next version the name make it obvious that it is a counter at least,
// but still no idea what it is for
count = count + 1;

// finally, the self explanatory version    it doesn't need any comment!
numLoopsSinceStart = numLoopsSinceStart + 1;
```

27 September 2020

42

Self test question

43

Q. Write out the following code with correct brace style, indentation etc. Fix the naming scheme for all variable and function identifiers. (Note: You don't need to understand what the code does to be able to answer this.)

```
const char MyConstant = 21; int i;  
void some_Functionname()  
{ for(i=3;i>0; i =i+1) { digitalWrite(MyConstant+i ); }  
}
```

27 September 2020