ENGD2103 Embedded Systems Fundamentals C programming language self-primer

J. Dimitrov and Dr M. A. Oliver

jordan@dmu.ac.uk and michael.oliver@dmu.ac.uk

De Montfort University
Leicester, UK.

Introduction to C programming language

Exercises

You are expected to try out all code in these notes and then go ahead and work on the exercises as well in your own time.

These notes will provide you with a hands-on introduction to C programming on an Arduino platform. These notes contain several code samples. You are highly recommended to type these character-by-character into your Arduino IDE as this effort will drastically facilitate your learning.

Program structure

Any C program is a collection of declarations and functions. There is only one rule stating that there must be a single function called `main()`. The classic example is the "hello world" program.

```
#include <stdio.h>
main() {
  printf("hello, world\n");
}
```

If you want to compile and execute this program you will need an ANSI C compatible compiler like `gcc` and a suitable operating environment, e.g. `Windows`, `MacOS`, `Linux`, etc.

Arduino programs are C programs called sketches. As we saw earlier, the Arduino IDE has set up the `main()` function for us and this has 2 implications

1. `main()` is not allowed in an Arduino sketch
2. instead of a `main()` we will use `setup()` and `loop()`.

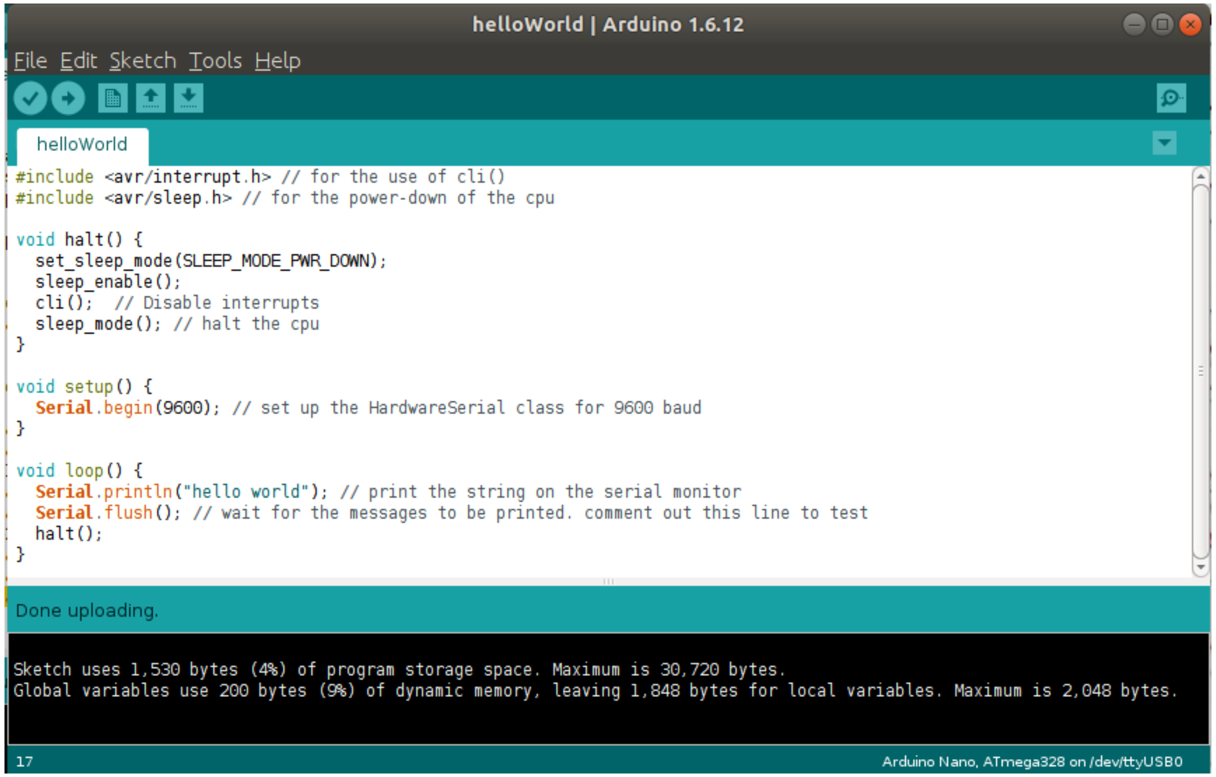
Embedded systems run as long as they are powered and this continuous execution makes `halt()` empty of meaning. We will interpret this as a deep sleep mode.

Arduino's "hello world" — 1

The Arduino equivalent for the "hello world" program is

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu

void halt() {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  cli();  // Disable interrupts
  sleep_mode(); // halt the cpu
}

void setup() {
  Serial.begin(9600); // set up the HardwareSerial class for 9600 baud
}

void loop() {
  Serial.println("hello world"); // print the string on the serial monitor
  Serial.flush(); // wait for the message to be printed. comment out this line to test
  halt();
}
```

Arduino's "hello world" — 2

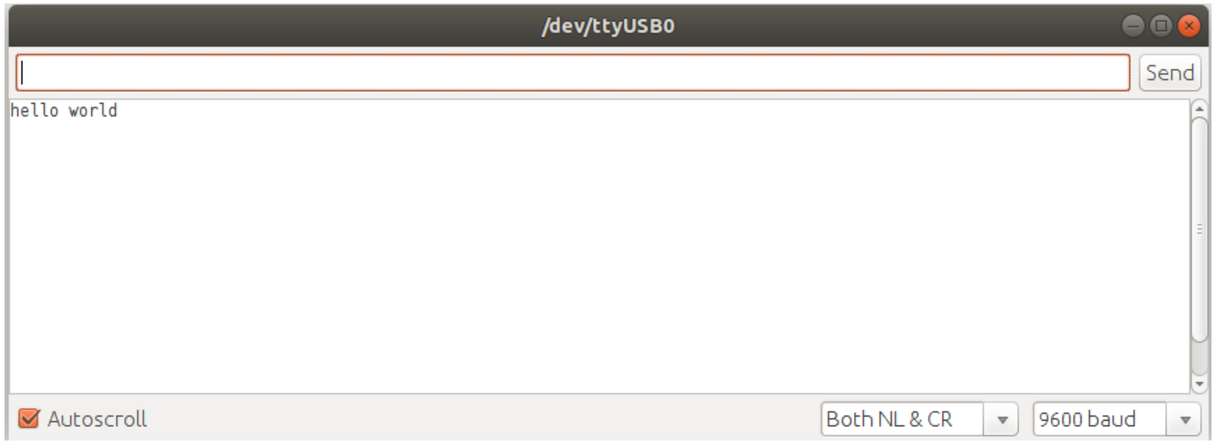


```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu

void halt() {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  cli();  // Disable interrupts
  sleep_mode(); // halt the cpu
}

void setup() {
  Serial.begin(9600); // set up the HardwareSerial class for 9600 baud
}

void loop() {
  Serial.println("hello world"); // print the string on the serial monitor
  Serial.flush(); // wait for the messages to be printed. comment out this line to test
  halt();
}
```

Done uploading.

```
Sketch uses 1,530 bytes (4%) of program storage space. Maximum is 30,720 bytes.
Global variables use 200 bytes (9%) of dynamic memory, leaving 1,848 bytes for local variables. Maximum is 2,048 bytes.
```

17 Arduino Nano, ATmega328 on /dev/ttyUSB0



```
hello world
```

Autoscroll: ✓ — Both NL & CR — 9600 baud

Variables

Variables need to be declared before use in C programs. A variable declaration consists of

```
type var-name;
```

Types in C can be **`void`** (when a type-less declaration is needed), Integer (**`char, int, long`**), Floating point (**`float, double`**) and many more.
Var-name is an identifier, i.e. a name, and must be a word made of letters and numbers starting with a letter or underscore (_).
Variables are used to store values of their (or compatible) type. The assignment operator is used to assign a value to a variable.

Example variables, assignment

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
int x; // global integer variable declaration
void halt() {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() { Serial.begin(9600); }
void loop() {
  x=0; // assign 0 to x
  Serial.print("x = "); // print the string on the serial monitor
  Serial.println(x, DEC); // print the value as a decimal integer
  x=x-1; // decrement x
  Serial.print("x = "); Serial.println(x, DEC);
  x=x+2; // add 2 to x
  Serial.print("x = "); Serial.println(x, DEC);
  Serial.flush(); // wait for the messages to be printed
  halt();
}
```

Expressions

We will consider three different kinds of expressions for now

1. arithmetic — the result is a number type value. `5+x`, `5-x`, `5*x`, `5/x`, `x/5.0`. The last example is a floating point expression. The others depend on the type of `x`, i.e. if the type of `x` is integer, then the type of the expression will be integer. If the type of `x` is float, then the type of the expression is float. Normal priorities of the operations apply and parentheses are used to enforce a change in them.
2. assignment is an operation, i.e. `x = expression` has a value it is the value of `expression`. This is to facilitate `y = (x=0) + 1;` which means assign 0 to x and 1 to y.
3. bool — boolean type is defined in Arduino as containing the two values `true` and `false`. Logical comparison operations < (less than), > (more than), <= (less or equal), >= (more or equal), == (equal), != (not equal), && (and), || (or), ! (not)

Example expressions

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
int x, y; // global integer variable declaration
bool flag; // global boolean flag
void halt() {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {
  Serial.begin(9600); // set up the HardwareSerial class for 9600 baud
}
void loop() {
  y = (x=0)+1; // assign 0 to x and 1 to y
  flag = x>y;
  Serial.print("x = "); Serial.print(x, DEC); Serial.print(", y = "); Serial.print(y,
        DEC); Serial.print(", flag = "); Serial.println(flag);
  flag = ( !( (x=x-1) >= (y=y+2) ) ) && (x<0) && !(y>=10);
  Serial.print("x = "); Serial.print(x, DEC); Serial.print(", y = "); Serial.print(y,
        DEC); Serial.print(", flag = "); Serial.println(flag);
  Serial.flush(); // wait for the messages to be printed
  halt();
}
```

If-statement

The **`if-else`** statement is used to take decisions. Formally the syntax is

```
if (expression) statement1
else statement2
```

where the **`else`** part is optional. The `expression` is evaluated; if it is `true` (that is, if `expression` has a non-zero value), `statement1` is executed. If it is `false` (`expression` is zero) and if there is an **`else`** part, `statement2` is executed instead.

The example

```
if (a > b) z = a;
else z = b;
```

is an implementation of `z=max(a,b)`

While-loop

The **`while`** statement is used to iterate statements. Formally the syntax is

```
while (expression) statement
```

where the `expression` is evaluated; if it is `true`, the possibly compound `statement` is executed. If it is `false`, the **`while`** completes its execution and the following statement is executed.

What will be the values printed on the serial monitor?

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
int x, y; // global integer variable declaration
void halt() { set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli();
      sleep_mode();}
void setup() { Serial.begin(9600); }
void loop() {
  y=x=0; // assign 0 to x and y
  while (x<10) { x = x+1; y = y+x; }
  Serial.print("x = "); Serial.print(x, DEC); Serial.print(", y = ");
        Serial.println(y, DEC);
  Serial.flush(); // wait for the messages to be printed
  halt();
}
```

The Blink Program — 1

One of the simplest programs that can be run on the Arduino is the “Blink” program. This program simply blinks the LED on and off over a 1 second period.

- Open the Arduino IDE
- Type in the following code:

```
// The Arduino Nano has a built-in LED on Digital I/O pin 13
#define LED     13
void setup() {
    // Setting-up the LED pin as an output
    pinMode(LED, OUTPUT);
}
void loop() {
    // Turn the LED on for 500 ms
    digitalWrite(LED, HIGH);
    delay(500);
    // Turn the LED off for 500 ms
    digitalWrite(LED, LOW);
    delay(500);
}
```

The Blink Program — 2

One of the simplest programs that can be run on the Arduino is the “Blink” program. This program simply blinks the LED on and off over a 1 second period.

- In “Tools” ensure that the selected Board is “Arduino Nano”. For the laboratory PCs, the value for Port should typically be chosen to be the highest available COM port.
- Click the Upload button. If this is the first compile, you will be asked for a filename: type “blink” at the prompt. The code should compile and be uploaded into your Arduino Nano. If it does not compile, locate and correct any errors, then try again.
- Once uploaded the LED should flash with a period of 1 second (500ms + 500ms).
- Read through the code (including the comments) and try to understand what is happening.
- Modify the program such that the LED stays on for 250ms and off for 250ms.

The Blink Program — 3

One of the simplest programs that can be run on the Arduino is the “Blink” program. This program simply blinks the LED on and off over a 1 second period.

- Modify the program such that the LED stays on for 100ms and off for 900ms.
- Extend the program to repeatedly run though the following sequence:

```
LED ON  500ms
LED OFF 500ms
LED ON  100ms
LED OFF 100ms
LED ON  100ms
LED OFF 100ms
LED ON  100ms
LED OFF 2000ms
```

“Hello World” Program — 1

The “Hello World” program is usually the first C program that one writes when learning C. Traditionally, this displays the message “Hello World” on the screen. However, there is no screen on the Arduino Nano; the message is written to the serial port instead.

- Start with a new Arduino script. One way of achieving this is to click “File” then “New” from your existing Arduino script.
- Type in the code listed below.

```
void setup() {
// Set-up the serial port for 115200 Baud.
// (115200 and 9600 are commonly used Baud rates).
Serial.begin(115200);
Serial.print("Hello world");
}

void loop() {
    // For this exercise, nothing happens here
}
```

“Hello World” Program — 2

- Compile and upload the code (this time using the filename hello). The code should compile then start running. At this point you should see nothing.
- Open a Serial Monitor (by clicking Tools then Serial Monitor). On opening the Serial Monitor, the Arduino Nano is reset and the code will run again. You should see a message in the serial port.
- If garbage appears, the most likely cause is a mismatch in Baud rate. If this is the case, change the Baud rate (towards the bottom-right of the Serial Monitor) to match that set in the code. At this point, the Arduino will reset and you should see a correct message appear.
- Read through the code (including the comments) and try to understand what is happening.
- Add an extra `Serial.print()` call to add an extra message string of your choice after the “Hello World” message.
- Modify the code from Exercise 2.1; this time displaying the “Hello World” message using `Serial.println()` instead of `Serial.print()`. What is the difference between `Serial.print()` and `Serial.println()`.
- Modify the code from Exercise 2.2 by adding the newline character `\n` in the middle of one of your strings. What happens when this is added?

Fahrenheit To Celsius

The listing below is a slightly more involved program that introduces integer (**`int`**) variables, integer arithmetic, comparisons and repetition (**`while`**).

```
void setup() {
  int fahr, celsius, lower, upper, delta, step;
  Serial.begin(115200);// Set-up the serial port for 115200 Baud.
  lower = 0;// Lower limit of temperature table
  upper = 300;  // Upper limit
  delta = 20;   // Step size ('step' is considered a reserved word)
  fahr = lower;// Output the table
  while (fahr <= upper)
    {
    celsius = 5 * (fahr - 32) / 9;
    Serial.print(fahr);
    Serial.print("\t");   // Print a tab
    Serial.println(celsius);
    fahr = fahr + delta;
    }
}
void loop() {
    // For this exercise, nothing happens here
}
```

Fahrenheit To Celsius

- Compile and upload the code (this time, naming the program source FahrToCelsius).
- A table of Fahrenheit values alongside their Celsius values, is sent to the Serial monitor.
- Read through the code and try to understand what is happening. Does this make sense? What do you think the character `\t` stands for?
- Modify your program to add a heading across the top of the table.
- Create an equivalent program that sends the corresponding Celsius to Fahrenheit table to the serial port.
- Up to now, you have been working with integer values. Modify the code of the previous exercise such that all the variables are now floating point. Hint: the keyword for the floating-point type is **`float`**. Observe the difference in output between using integer values and floating-point values.
- Referring back to the previous exercise, the Fahrenheit value can be presented to 4 decimal places using `Serial.print(fahr, 4);` as opposed to `Serial.print(fahr);` Try this out. Once satisfied that this behaves, modify your code to present the Fahrenheit value to 0 decimal places, and the Celsius values to 1 decimal place.
- Modify your program to present the table in reverse order, i.e. from 300 degrees to 0. Try extending this to present the table from 300 degrees to -100.

{...}

The compound operator in C has dual purpose.

1. Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement.

2. Braces are used to scope variable names.

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
int x; // global integer variable declaration
void halt() {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {Serial.begin(9600);}
void loop() {
  x=0; Serial.print("global x = "); Serial.println(x, DEC);
  { int x; x=1; Serial.print("local x = "); Serial.println(x, DEC); }
  Serial.print("x = "); Serial.println(x, DEC);
  Serial.flush(); // wait for the messages to be printed
  halt();
}
```

`for` statement — 1

`for` is the main iteration statement. The syntax of the statement is

```
for ( expression1 ; expression2 ; expression3 ) statement;
```

Statement can be an empty or a compound statement. The expressions can also be empty as well. Semantically the `for` loop is equivalent to

```
expression1;
while ( expression2 ) {
  statement;
  expression3;
}
```

What will be the behaviour of the following code?

```
for ( ; ; ) Serial.print("hello world\n");
```

Another typical example is

```
for ( i=0; i<10; i++) {
  Serial.print("hello world "); Serial.println (i, DEC);
}
```

for statement — 2

What will the behaviour of the following code be?

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli();
     sleep_mode();}
void setup() {Serial.begin(9600);}
void loop() {
  float fahr;
  for (fahr = 0; fahr < 300; fahr = fahr + 20) {
    Serial.print(fahr, 2);
    Serial.print('\t');
    Serial.println((5.0/9.0) * (fahr-32.0), 2);
  }
  Serial.flush(); // wait for the messages to be printed
  halt();
}
```

Nested `if`

The `if` statement allows us to make a binary choice. More often then not, we will need to make a choice between more than just 2 options. An example would be our timetable:

```
if (day == Monday) {
  if (hour == 9) Serial.println("Maths");
  else if (hour == 10) Serial.println("Embedded");
  else ...
}
else if (day == Tuesday) {
  ...
}
else ...
```

`switch` statement

The semantics of programs written with nested **`if`** statements is very difficult to comprehend. A better code structure is achieved using the **`switch`** statement which is a multi-way decision that tests whether the `expression` matches one of a number of the constant integer values `constExpr1`, `constExpr2`, etc., and branches accordingly.

```
switch (expression) {
  case constExpr1: statements
  case constExpr2: statements
     ...
  default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

`switch` statement

The timetable example would look much better

```
switch (day) {
  case Monday:
     switch (hour) {
        case 9: Serial.println("Maths"); break;
        case 10: Serial.println("Embedded");
        case 11: Serial.println("... with Embedded lab as well"); break;
        ...
     }
  case Tuesday: {
        ...
     }
  ...
}
```

Note the **`break`** statements! If `hour==10` on the `Monday` case, then both `Serial.println("Embedded");` and `Serial.println("Embedded"); Serial.println("... with Embedded lab as well");` will be executed. If `hour==11`, then only `Serial.println("... with Embedded lab as well");` will run.

`break` and `continue`

- In many cases it is necessary to exit from a loop other than by testing at the top. The **`break`** statement provides an early exit from **`for`** and **`while`**, just as from **`switch`**. A **`break`** causes the innermost enclosing loop or switch to be exited immediately.
- The **`continue`** statement is related to **`break`**. It causes the next iteration of the enclosing **`for`** or **`while`** loop to begin.

`++` and `--`

The increment operator `++` adds 1 to its operand, while the decrement operator `--` subtracts 1 as in `++n1;`

The unusual aspect is that `++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix operators (after the variable: `n++`). In both cases, the effect is to increment `n` . But the expression `++n` increments `n` before its value is used, while `n++` increments `n` after its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. Example

```
n=5;
x = n++;
// here x==5 and n==6
n=5;
x = ++n;
// here x==6 and n==6
```

`#include` and `#define`

C programming language provides certain facilities by means of a preprocessor, which is conceptually a separate first step in compilation. **`#include`** and **`#define`** are used to include a source or to replace a token by an arbitrary sequence of characters within the current file.

Any line in the current file of the form **`#include`** `"filename"` or **`#include`** `<filename>` is replaced by the contents of the file `filename`. The difference is where the file `filename` is taken from. `"filename"` searches in the current folder and `<filename>` asks the operating system to search in its *include* folders.

`#define` `name replacement` is macro substitution, i.e. any subsequent occurrences of the token `name` will be replaced with the `replacement`.

Both **`#include`** and **`#define`** are processed *before* the compilation begins, i.e. any calculations that may be required due to the substitutions will be performed by the compiler and will not take time during the execution of the program.

Arrays

An array is a finite sequence of variables of the same type. The declaration `int a[10];` creates 10 individual variables of type `int`, i.e. `a[0]`, `a[1]`, `a[2]`, `...`, `a[9]` are all individual integer variables. The indexes of the variables always start from `0`. Arrays can also be indexed by expressions whose values are calculated at run-time, i.e.

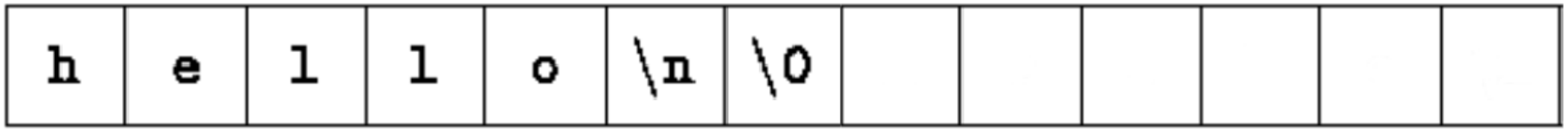
```
int i, a[10];
for(i=0; i<10; i++) a[i]=0;
for(i=0; i<10; a[i++]=0); // equivalently
```

zeroes all variables of the array `a[]`. What does the following example do?

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() { set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli();
      sleep_mode();}
int i, a[10], b;
void setup() {Serial.begin(9600);}
void loop() {
  for(i=0; i<10; a[i]=++i);
  for(i=0; i<10; i++) Serial.println(a[i]);
  for(b=a[i=0]; i<9; i++) a[i]=a[i+1];
  a[9]=b;
  for(i=0; i<10; i++) Serial.println(a[i]);
  Serial.println("\nhalted");Serial.flush();halt();
}
```

Strings

Strings are simply null terminated, character arrays.



Therefore the code

```
char s[13] = "hello\n";
Serial.print(s);
```

creates and prints a character array of 13 individual char variables `s[0]`, `s[1]`, `...`, `s[12]`. The first 7 have been assigned `s[0] = ’h’`, `s[1] = ’e’`, `...`, `s[4] = ’o’`, `s[5] = ’\n’` and `s[6] = ’\0’`. Note that `s[7]` to `s[12]` have been left undefined and can hold any value. You can change the values of the string by changing the values of its individual variables. What will this do?

```
char s[13] = "hello\n";
Serial.print(s);
s[5] = ’ ’; s[6] = ’w’; s[7] = ’o’; s[8] = ’r’; s[9] = ’l’; s[10] = ’d’; s[11] =
     ’\n’; s[12] = ’\0’;
Serial.print(s);
s[5] = ’\0’;Serial.print(s);
Serial.println("\nhalted");Serial.flush();halt();
```

Challenge

The challenge is to create a program that takes an input from the `Serial.read()` stream, interprets its input as a `hh:mm` formatted time in “digital clock”, i.e. 2 digits for the hour, colon, 2 digits for the minutes and outputs its equivalent in the “past” and “to”, i.e. “analogue clock”. In other words, if the input is “10:40”, the output should be “20 to 11” and if the input is “23:15”, the output should be “15 past 11”.

You will need to know

- `Serial.available()` — this returns the number of characters in the `Serial` input stream.
- `Serial.read()` — this returns the first available character from the input stream, removing it from the input stream, if there are characters to be returned, and it returns -1, if there are no characters in the input stream at all.

Language reference link at `https://www.arduino.cc/en/Reference/HomePage`.

for loop

Consider the following Arduino sketch that uses a for loop to count from 0 to 9 and output the results via the serial interface.

```
void setup() {
    int i;
    Serial.begin(115200);
   for (i = 0; i < 10; i++)
    {
        Serial.println(i);
    }
}
void loop() {
    // put your main code here, to run repeatedly:
}
```

Type-in this code and satisfy yourself that it behaves as expected.

for loop

Now consider the following Arduino sketch that also uses a for loop to count from 0 to 9 and output the results via the serial interface.

```
void setup() {
    int i;
    Serial.begin(115200);
   for (i = 0; i <= 9; i++)
    {
        Serial.println(i);
    }
}
void loop() {
    // put your main code here, to run repeatedly:
}
```

Type-in this code and satisfy yourself that it behaves as expected.

for loop

- Modify the above code such that the system counts from -10 to +10.
- Modify the previous exercise such that the system counts from -10 to +10 in steps of +2.
- Modify the previous exercise such that the system counts down from +10 to -10.
- Modify the previous exercise such that the system counts down from +10 to -10 in steps of -2.
- Repeat the Fahrenheit-to-Celsius conversion exercises above, this time using a for loop instead of while.

Bit Shifting

The » operator is used to right-shift a value. The following code is used to shift a value i 1 place to the right `i = i >> 1;` or `i >>= 1;`

The « operator is used to left-shift a value. The following code is used to shift a value i 2 places to the left `i = i << 2;` or `i <<= 2;`

Consider the following piece of code that sets the most significant bit of a byte value to 1 (giving a value of $2^7 = 128$). This is repeatedly right-shifted 1 place to the right until the resultant value is 0.

Bit Shifting

```
void setup() {
    byte i;
    Serial.begin(115200);
    Serial.print("Decimal");Serial.print("\t");Serial.print("Hex");
    Serial.print("\t");Serial.println("Binary (without leading zeros)");
    Serial.println("----------------------------------------");
    for (i = 128; i; i >>= 1) {
        Serial.print(i); Serial.print('\t'); Serial.print(i, HEX);
        Serial.print('\t'); Serial.println(i, BIN);
    }
}
void loop() {
    // put your main code here, to run repeatedly:
}
```

Repeat the above but in reverse. In other words, start with i as $2^0 = 1$, run through the loop each time i is non-zero, and left-shift i by 1 place. When you run the program, you will see the value of i increase by a factor of 2 for each iteration. Can you explain why i becomes zero?

Functions

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and making code changes easier.

Each function definition has the form

```
return-type function-name(argument declarations)
{
   declarations and statements
}
```

Various parts may be absent; a minimal function is

```
 void dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is `viod`, then the function returns nothing.

Functions

```
void setup() {
  /* code run once before anything else */
}
void loop() {
  /* code run forever after setup() */
}

void main() {
  setup(); for(;;) loop();
}
```

We can recall that the compiler uses the sketch we write to create the `main.cpp` file above. Having that in mind, please . . .

static

...consider the following two sketches

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {Serial.begin(9600);}

void loop() {
  int i=0;
  if (i<3) Serial.println(i);
  else halt();
  i=i+1;
}
```

and ...

static

…can you see the difference?

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {Serial.begin(9600);}

void loop() {
  static int i=0;
  if (i<3) Serial.println(i);
  else halt();
  i=i+1;
}
```

const

While the following

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {Serial.begin(9600);}

void loop() {
  const int i=0;
  if (i<3) Serial.println(i);
  else halt();
  i=i+1;
}
```

demonstrates the use of constant variables which may not be changed by the program we are writing but may be changed by an external process. Except that it should diagnose explicit attempts to change `const` variables, a compiler may ignore this qualifier.

Type casting

Type casting is used in C to force the type of any expression with a unary operator called cast. The syntax is

```
 (type name) expression
```

and this simply means that the result of the `expression` will be interpreted as of type `type name`. For example, in

```
x=(int) z;
```

the *value* of the variable `z` is interpreted as integer and assigned to `x`. The actual variable `z` is left unchanged.

Type casting

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void setup() {Serial.begin(9600);}
void loop() {
  int x;
  float z;

  for (z = 0; z<5; z+=0.1) {
    x=(int) z;
    Serial.print(z);Serial.print('\t'); Serial.println(x);
  }
  halt();
}
```

Did you find anything unusual about the result of this code when you ran it in your lab session?

Bitwise operations

C provides six operations for bit selection and manipulation; these may only be applied to integer type operands, that is, `char`, `short`, `int`, `long`, etc.

- `&` — bitwise AND
- `|` — bitwise inclusive OR
- `^` — bitwise exclusive OR
- `<<` — left shift
- `>>` — right shift
- `~` — one’s complement (unary)

You can use and change the code on the next slide accordingly, to see the result of each bitwise operation given above, in your lab session.

Bitwise operations

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
void binaryPrint(unsigned char x) {
  unsigned char mask;
  for (mask=1<<7;mask;mask>>=1) Serial.print((x&mask)?1:0);
}
void binaryPrintln(unsigned char x) { binaryPrint(x);Serial.print('\n'); }
void setup() {Serial.begin(9600);}
void loop() {
  int x,y;
  for (x = 0; x<256; x++)
    for (y = 0; y<256; y++) {
    binaryPrint((unsigned char) x);Serial.print('\t');
    binaryPrint((unsigned char) y);Serial.print('\t');
    binaryPrintln(((unsigned char) x)&((unsigned char) y));
  }
  halt();
}
```

Conditional Expressions

The statements

```
if (a > b)
z = a;
else
z = b;
```

compute in `z` the maximum of `a` and `b`. The conditional expression, written with the ternary operator "`?:`", provides an alternative way to write this and similar constructions. In the expression `expr1 ? expr2 : expr3` the expression `expr1` is evaluated first. If it is non-zero, i.e. `true`, then the expression `expr2` is evaluated, and that is the value of the conditional expression. Otherwise `expr3` is evaluated, and that is the value. Only one of `expr2` and `expr3` is evaluated.

Boolean type and operations

Standard C has type `boolean` but you will need to `#include <stdbool.h>` and the type is actually called `_Bool`, with **`bool`** being a macro synonym for `_Bool`. C++ has got the `boolean` type in its primary syntax. Therefore, you can use **`bool`** or `boolean` interchangeably and freely since Arduino C is C++. The three boolean operations you can use are

- `&&` — logic AND
- `||` — logic OR
- `!` — logic not (unary)

You can use and change the code on the next slide accordingly, to see the result of each logic, aka boolean, operation given above, in your lab session.

Boolean type and operations

```
#include <avr/interrupt.h> // for the use of cli()
#include <avr/sleep.h> // for the power-down of the cpu
void halt() {
  Serial.print("\nhalted\n");Serial.flush(); // wait for the messages to be printed
  set_sleep_mode(SLEEP_MODE_PWR_DOWN); sleep_enable(); cli(); sleep_mode();
}
boolean boolPrint(unsigned char x) {
  unsigned char mask;
  for (mask=1<<7;mask;mask>>=1) Serial.print((bool) (x&mask));
  Serial.print((x)?"\ttrue\t":"\tfalse\t");
  return((x)?true:false);
}
void boolPrintln(unsigned char x) { Serial.print(boolPrint(x));Serial.print('\n'); }
void setup() {Serial.begin(9600);}
void loop() {
  int x,y;
  for (x = -5; x<6; x++) for (y = -5; y<6; y++) {
      Serial.print(x);Serial.print('\t'); Serial.print(y);Serial.print('\t');
      boolPrintln((unsigned char)(x && y));
  }
  halt();
}
```

Challenge

Your task is to create a program that takes an input from the `Serial.read()` stream, interprets its input as a single binary operation on natural numbers in the format `n o n`, i.e. a natural number for the first operand, followed by a single character for the operation, followed by a natural number for the second operand. In other words, if the input is “`124 - 120`”, the output should be “`4`” and if the input is “`1 & 2`”, the output should be “`0`”.

The available operations are “`+, -, *, /, &, |`” as defined above.

Exercises

- Write a function with the following prototype **`void`** `writeBinary(byte value);` This takes a parameter called value. The function will write an 8-bit binary representation of value to the serial port.
- Write a function with the following prototype `byte readHex();` This reads a hexadecimal value from the keyboard in the form 0xHH and will return the converted value as a byte (once valid input has been parsed).
- Write a program that uses two calls to `readHex()` to read in two variables a and b.
  - Perform a bitwise OR on the two variables
  - Perform a bitwise AND on the two variables
  - Perform a bitwise XOR on the two variables
  - Perform a bitwise inversion on a
  - Perform a bitwise inversion on b

  In each case use `writeBinary()` to present the result as an 8-bit binary representation.