Embedded Systems 2011
Embedded Programming, Arduino and the NICTA ed1

Introduction to the Embedded Challenge

Welcome

Welcome to the Embedded Stream of the 2011 NCSS Challenge. If you have not yet done so please install the Arduino software on your machine. You will need the software to work through these notes. The "Getting Started" guide will run you through the software installation process.

This first week of notes is longer than the notes for each of the next four weeks as you need to learn a few basics to get started in programming. Please, if possible, work through the notes with your software installed and your board connected trying out the examples as we go.

What are Embedded Systems?

The term "embedded system" is used to describe a physically small computer built into an everyday object like a mobile phone, a car or a washing machine. It is often a special purpose device designed to perform one task usually with some real-time constraint. The term "system" refers to the fact that the computer and the surrounding hardware work together as a single unit. For example the embedded system in your washing machine is closely linked with both with the washing machine control panel and the mechanical components of the machine. Programming the embedded system is literally programming the function of the washing machine itself. Once installed the embedded system may simply run the same program over and over until the end of the life of the device. This is in contrast to your desktop computer which is re-programmable for many different tasks.

What makes embedded systems interesting is the fact that they themselves are becoming quite complex devices. Writing software for embedded systems is a major task for many companies and the ever increasing demands on embedded systems is keeping research organisations, like NICTA, busy. Also interesting aspect of embedded systems is that the hardware and software is cheap and readily available to hobbyists and others to use in building their own programmable devices.

What is Arduino?

The Embedded Challenge makes use of the Arduino programming language and the Arduino development environment. Arduino is an open source environment intended for use by artists, hobbyists and others to use in making interactive devices. The Arduino hardware platform is based on a *microcontroller* which is essentially a small computer on a single integrated circuit. The microcontroller contains a processing unit, random access memory (RAM), flash memory (non-volatile storage) and circuits to interface with external hardware.

There is a comprehensive web site dedicated to Arduino:

http://arduino.cc/

The great thing about the Arduino software is that a group of people have already done the background work to make it all relatively easy to use. We use it for the Challenge as it allows us to concentrate directly on programming and interfacing with interesting external components. If you want a feel for the underlying complexity of the microcontroller take a peek at the manufacturers (448 page, 13.3MB) data sheet at the URL below and then you can appreciate the simplicity of the interface offered by the Arduino software:

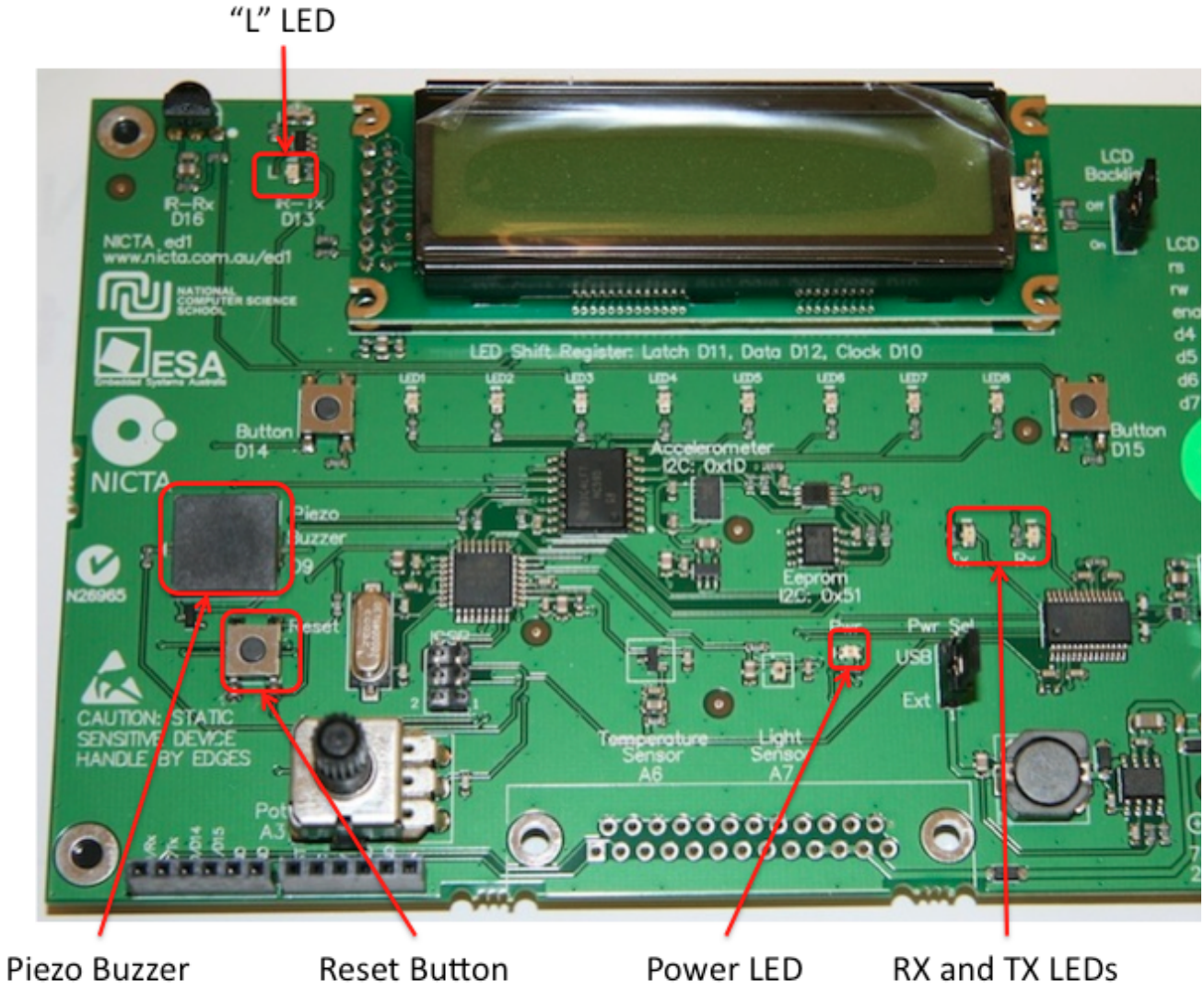
http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf

An Arduino board, like the ones described on the Arduino web site, consist of a microcontroller and a very small number of components required to both make the microcontroller work and interface with a host computer. At the edge of the board are rows of pins to which external components can be connected. This Challenge uses a different board, the NICTA ed1. Instead of the rows of pins waiting for connection to external components the NICTA ed1 has a number of components built in and permanently connected to the microcontroller.

The NICTA ed1 board has been designed by NICTA specifically for this Challenge. It is not an Arduino board but is completely compatible with Arduino software.

A picture of the NICTA ed1 with some of the features labeled is shown below. If you have a NICTA ed1 board from 2009 you will notice that the Reset button has been moved and the potentiometer looks a little different but

functionally the boards are identical.



Where to Find Help

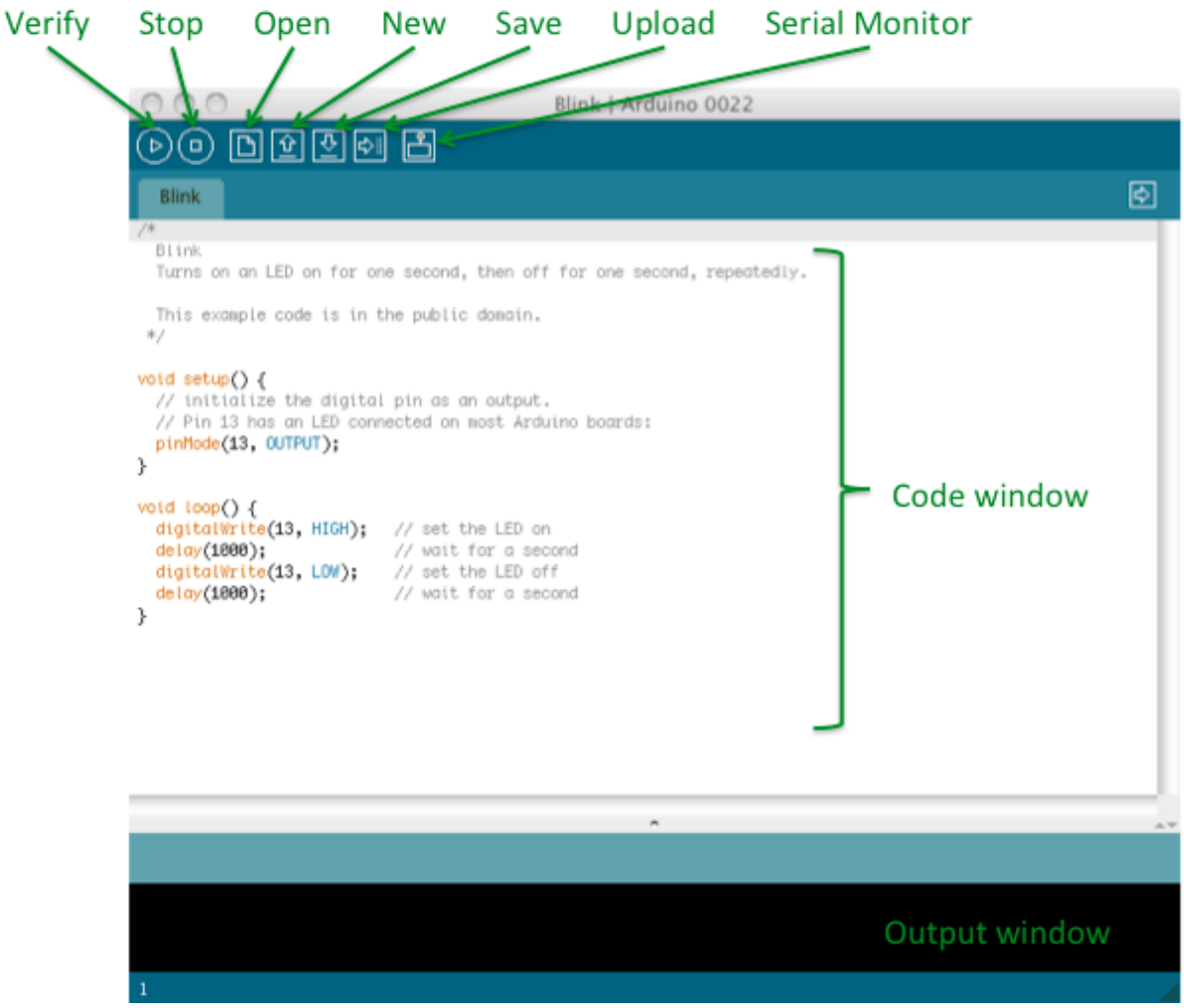
The notes will have most of the information you need to complete the Challenge questions but you may need to read the Arduino reference as well. Look for programming reference material, particularly for Arduino library calls, on the Arduino Language Reference page. Another reference that may be useful is the freely available Arduino Notebook which you might be able to print out.

First Run of the Arduino Software

By the time you get to here you should have the Arduino software downloaded and installed on your computer.

The installation pages on the Arduino web site should have introduced you to running the IDE (stands for "Integrated Development Environment") and the menus for choosing your serial interface and board type.

Lets have another look at the IDE.



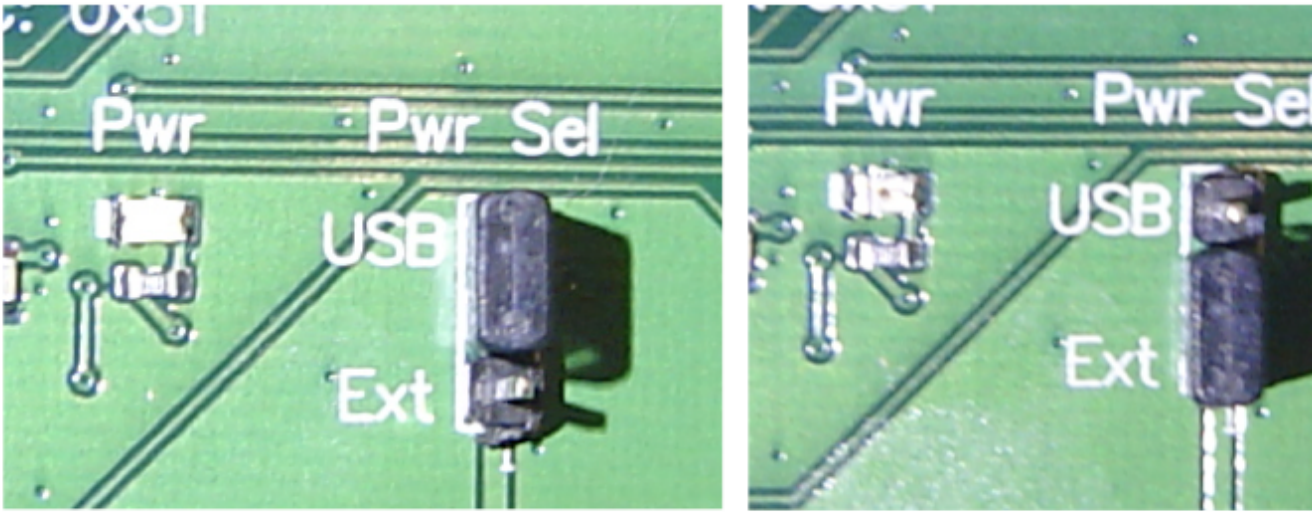
The figure above shows the IDE open with the Blink program code loaded. Start the Ardiuno software on your computer and if the Blink program does not come up by default, load the Blink program from the menu **`File->Examples->1.Basics->Blink`**.

On the top row of the IDE interface there are a small number of buttons.

Press the **`Verify`** button.

This compiles the program currently being displayed in the text window, in this case the Blink program. In the Arduino environment the term "sketch" is used to describe a program. In this course the terms "sketch" and "program" are used and mean the same thing.

Make sure your ed1 board is connected via the USB cable to the computer. Check that the power indicator LED on the board is on. If the power LED does not light up then disconnect the USB cable and look at the jumper marked "Pwr Sel" which is adjacent to the power LED. This jumper should cover the top two pins, if it covers the lower pins instead then move it by pulling it straight up off the board and pushing it down over the top two pins. Then reconnect the USB cable. The diagram below shows how the jumper must be set for the ed1 to be powered from the USB cable.



Right Wrong

On the **Tools->Board** menu make sure **Arduino Duemilanove w/ ATmega328** is selected with a tick.

On the **Tools->Serial Port** menu make sure your USB serial connection is selected.

Press the **Upload** button.

This sends the compiled program to the ed1 board over the USB connection. You can tell if it is working okay as the LEDs on the ed1 board labelled "RX" and "TX" will flash as data is being transferred. Once transferred the code will start running, you should see the LED labelled "L" flashing on and off. Congratulations you have compiled and downloaded your first program!

Editing and Saving Code

Edit the program to vary the rate of flashing. Look for the statements that say:

```
delay(1000);
```

This are calls to an Arduino library function to pause program execution for a number of milliseconds, in this case for 1000ms (1 second). You can make the LED flash at a faster rate by changing the number from **1000** to **500** in both statements. Try it.

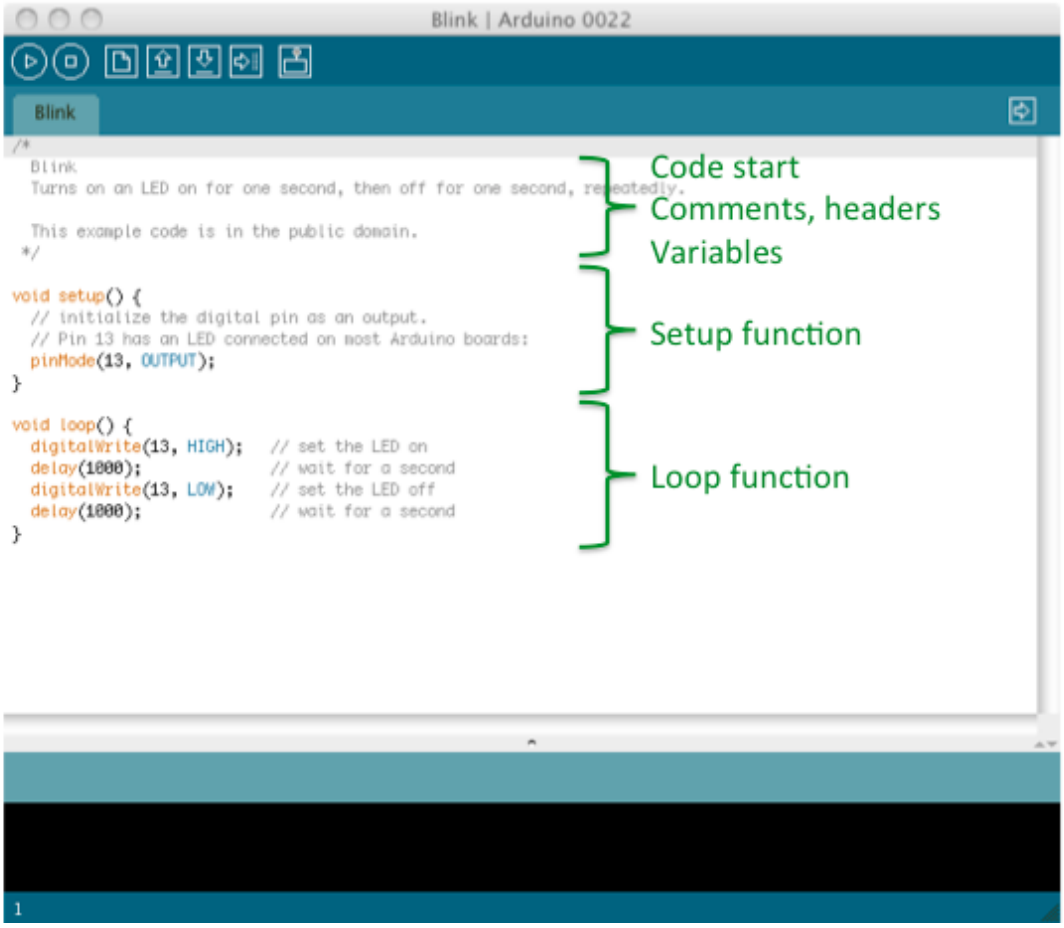
After making the edits press the **Verify** and **Upload** buttons again to compile and upload the new program. You should now notice the LED flashing at twice the rate that it was before.

If you press the **Save** button, to keep a copy of this version of the file, a text window will pop up stating that you cannot save to this location and will ask for a new program name. This is the same for all of the example programs, if you edit them you must save them in an alternate location. Press **Save** and select a new name for your program, by default the program is saved in a directory named "Arduino". The location of this directory depends on your computer operating system. For Mac OSX the location is under the "Documents" directory in your home account.

When browsing for the Blink program you may have noticed other example programs listed in the menu tree. Not all of these programs will work with the NICTA ed1 as many depend on certain configurations of hardware. Please resist the temptation to load other example programs until you know more as some programs may cause damage to the board.

The Structure of a Sketch

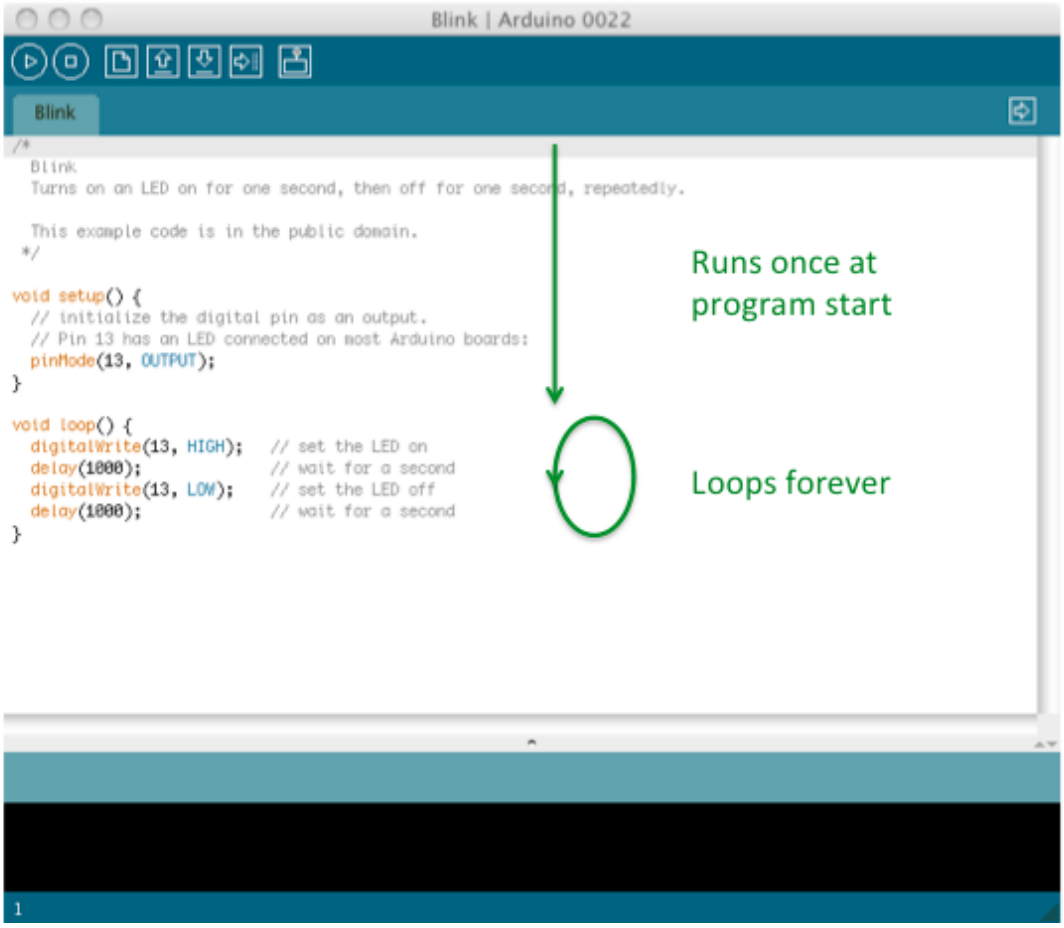
An Arduino program has the following structure:



There are three main sections:

1. a section at the top which can be used for declaring variables, constants, comments and including header files for external libraries
2. a "**`setup()`**" function for performing commands that happen only once at the start of the program
3. a "**`loop()`**" function for repeatedly performing a set of commands over and over

A running program follows the execution path shown in the picture below. The **`setup()`** function runs once at the start of the program, the **`loop()`** function runs to completion and then runs again over and over until either replaced or the power is turned off:



Walk Through of the The Blink Program

Here is the source code for the Blink program you ran as part of the exercise to set up the NICTA ed1 board:

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

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Let's step through this code and see what is happening. The first part of the code is a comment that extends over

multiple lines:

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

  This example code is in the public domain.
 */
```

Comments can start with the characters **/*** and end with ***/**. Anything in between (except for an ***/** of course) is regarded as a comment and not part of the code. As shown in the Blink program comments can run for multiple lines. The * character at the start of each line is just to make it look nice and is a common style in C programming.

The next part of the Blink code is the **setup()** function which starts with the line:

```
void setup()
```

The word **void** refers to the "type" of the value returned by the function. Neither the **setup()** and **loop()** functions return anything and this is what the type **void** means. The next part of the line is the name of the function **setup** and is followed by an argument list for the data being passed to the function. Here it is **()** which is an empty set of brackets as this function does not accept arguments. Despite the use of empty brackets in the Arduino example sketches it is actually better practice to formally indicate a lack of arguments with the **void** keyword, e.g.

```
void setup(void)
```

All Arduino programs must contain the functions **setup()** and **loop()** even if they are empty.

The next line is a opening curly brace **{** and delimits the start of the code contained in the function. In general the syntax for writing a function is:

```
return_type function_name(argument declarations)
{
   declarations
   statements
}
```

The first few lines of the **setup()** function are more comments telling us (the human readers of the program) what is going on. This is an alternative form of writing a comment, everything on a line after the **//** characters is a comment and the end of line signifies the end of the comment.

```
// initialize the digital pin as an output.
// Pin 13 has an LED connected on most Arduino boards:
```

After the comment lines we have the statement that will be executed when the **setup()** function runs:

```
pinMode(13, OUTPUT);      // sets the digital pin as output
```

Here we are calling the Arduino library function **pinMode()** with two arguments: a constant value **13** and the built in Arduino constant named **OUTPUT**. The statement is concluded with a semicolon (as all statements must be). As the previous comment lines tell us this is an Arduino library function to set Arduino pin 13 as an output.

The Arduino library function **pinMode()** is just one of the functions that we will be learning about. It sets a particular digital Arduino pin to act electrically as either an input or an output. In this case the function sets pin 13 to be an output. On the NICTA ed1 board pin 13 is connected to the "L" LED on the upper left hand corner of the board. The second argument **OUTPUT** is a special variable known as a constant. In this case it is defined by the Arduino library and is used specially with this function.

All Arduino functions are described in the language reference on the Arduino web site and the web site is the definitive source of information for the Arduino language. For example the web page for the **pinMode()** function is at:

http://arduino.cc/en/Reference/PinMode

The next line of the Blink code is a curly brace indicating the end of the statements that make up the **setup()** function.

The next function is **loop()**:

```
void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Like the **setup()** function, **loop()** has a **void** return type to indicate that it returns no value and the **()** characters indicate that it accepts no arguments.

The purpose of the statements that make up the function may be obvious to you: the first lights up the LED, the second pauses, the third turns off the LED and the fourth pauses again. The function automatically runs over and over repeating the same sequence. The result is a flashing LED.

All of the statements in the **loop()** function are calls to Arduino library functions. **digitalWrite()** sets an electrical output on a pin either **HIGH** (5 volts on this board) or **LOW** (0 volts). **HIGH** and **LOW** are constants defined by the Arduino library to indicate the output level.

The **delay()** function results in program execution suspending for a period of time. The argument to delay is an integral number of milliseconds.

Writing a "Hello World" Program

Time for you to write a new program from scratch. A typical first program with any computer language is one called "Hello World" which simply prints that message to the screen (see http://en.wikipedia.org/wiki/Hello_world_program). In this case we will send the "Hello World" text back across the serial link (over USB) to your PC.

In the Arduino IDE start a new program with the **New** button or with the menu command:

File->New
This opens a new sketch file with name formatted as *sketch_mmmdda* where the latter portion of the name is a date code (month day) followed by 'a', 'b', 'c' depending how many files you have started that day. The menu command:

File->Save As...
Allows you to enter a more descriptive name for the sketch you are creating. Use the menu and save your (as yet empty) new file as "hello_world".

Copy and paste the following code into your sketch and save.

```
/*
 * using the serial port to print "Hello World"
 */

void setup(void)
{
  Serial.begin(9600);

  Serial.println("Hello World");
}
```

Press the **Verify** button to compile. In the bottom panel of the IDE you should see an error message that looks something like this:

It can take some deciphering but the message is simply telling us we have forgotten the mandatory **loop()** function. When reading (and acting) on compilation error messages like this just deal with the top one or two errors at a time as it is quite common for a single error to create a cascading series of events that, really, you can

just ignore. In this case the error message above has two parts. To paraphrase what they mean:

1. **`core.a(main.cpp.o): In function `main':`** "There has been a compilation error"
2. **`/Applications/Arduino.app/Contents/Resources/Java/hardware/arduino/cores/arduino/main.cpp:10: undefined reference to `loop'`** "The **`loop()`** function is missing"

The error message will most likely change depending on the version of the Arduino IDE you are using. Don't worry if the message you see on your IDE is different. The overall message is that a problem has occurred during compile and you need to read it to find out some hints as to why.

Let's add the missing function and try again, it is okay to have an empty **`loop()`** function.

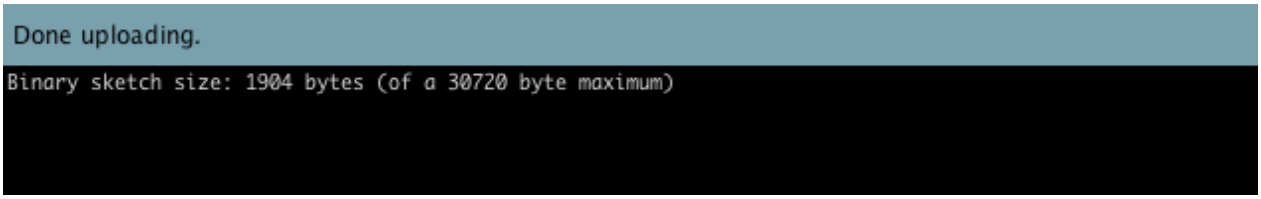
Add the missing lines so your code now looks like:

```
/*
 * using the serial port
 */

void setup(void)
{
  Serial.begin(9600);

  Serial.println("Hello World");
}

void loop(void)
{
}
```

Save and try to verify again. This time you should see the message:

Now in comforting white text (rather than alarmist red) this message tells us our sketch has compiled okay and a binary of 1904 bytes in size has been created. Do not worry if your binary sketch has a different size, the main point is that all is okay.

Upload to your Arduino board with the **`Upload`** button. You should see the bottom of the IDE now has the following message:



Press upload again and but this time watch the "TX" and "RX" LEDs on the Arduino board. (Press it again if you miss it).

After a small delay there is a flurry of activity as the code is sent over the USB cable and into the microcontroller. First the "L" LED lights a couple of times and then the "RX" and "TX" flash as the data is transmitted. The program has been saved into non-volatile memory called flash memory.

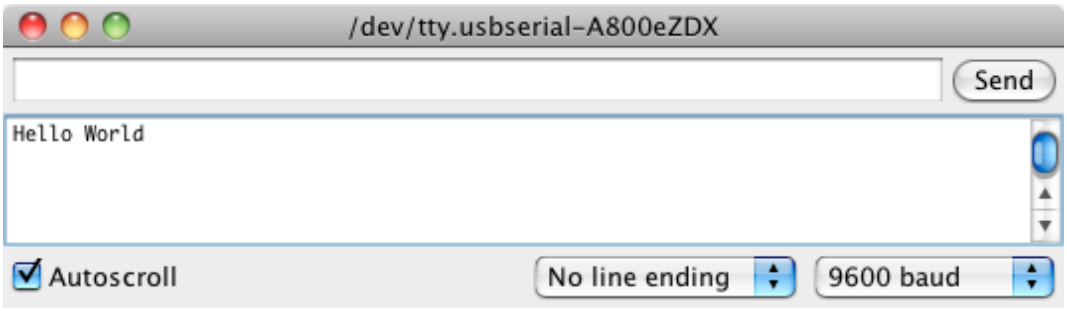
After download the program automatically starts to run on the board. The small program we have written calls two Arduino library functions once each during the **`setup()`** function:

```
Serial.begin(9600);
Serial.println("Hello World");
```

The first function configures the serial port on the microcontroller to operate at a rate of 9600 bits per second (baud). The second sends the text "Hello World" out of the serial port. Serial ports operate with a device on each end; to see the output we have to configure the other end of the connection, the one that terminates on your

computer at the other end of the USB cable.

Press the **`Serial Monitor`** button on the IDE. After a short delay a new window will pop up that should look like this:



If you see something different, like garbage characters, it may be that your IDE has the wrong baud rate selected. When two devices do not agree on a baud rate then communication is garbled. The baud rate set in the **`Serial.begin()`** function needs to match the baud rate set on the Arduino IDE. If the two rates do not match then you might have something that looks like this:

Select "9600 baud" from the pop up menu (the menu is where it says "9600 baud" on the first serial output diagram above) and press the reset button on the Arduino board. If you had the correct bit rate selected from the start try and select a different one and see what happens (it does not hurt anything). Pressing reset on the Arduino board after you have selected a new baud rate is good practice as you see the program execution from the beginning. (Enabling the serial monitor and selecting a different baud rate usually resets the board for you but it is better practice to do it yourself).

Serial Output

Serial output lets a program running on the NICTA ed1 send a text message back to the pop up serial window part of the IDE. It is a powerful tool that you can use to help debug your programs later on in the Challenge.

The Arduino **`Serial`** library provides access to the serial port. Like all other Arduino libraries mentioned in this course the definitive reference is the Arduino web site.

There are two variants to the function to print to the serial port:

```
Serial.print(arguments);
Serial.println(arguments);
```

The difference is that the latter prints a carriage return character (**`'\r'`**) followed by a newline character (**`'\n'`**) and the first one does not. Otherwise they are the same and just print out a single numeric value or a string (collection of characters) depending on the passed argument.

For example, consider the following lines of code (paste them into your **`setup()`** function):

```
Serial.println("Here is a number:");
Serial.println(42);
```

this will print out:

```
Here is a number:
42
```

If you are familiar with other programming languages these print functions will be primitive. To print out a more

complex line may require multiple print statements. For example

```
Serial.print("Is ");
Serial.print(42);
Serial.println(" really the meaning of life?");
```

will result in:

```
Is 42 really the meaning of life?
```

When the argument to either the `Serial.print()` or `Serial.println()` functions is an integer then an optional second argument flags the how the output is displayed.

Modifier	Prints out
DEC	decimal
HEX	hexadecimal number
OCT	octal number
BIN	binary number
BYTE	ASCII character

The default is to print the number out in decimal.

Variables

Now lets look at a more complex program that introduces the usefulness of variables. Cut and paste the following code into a new program window:

```
int ledPin = 13;   // use the "L" led
int pause = 1000;  // ms to pause between transitions

void setup(void)
{
  pinMode(ledPin, OUTPUT);
}

void loop(void)
{
  digitalWrite(ledPin, HIGH);   // set the LED on
  delay(pause);                 // wait
  digitalWrite(ledPin, LOW);    // set the LED off
  delay(pause);                 // wait
}
```

You may recognise this as structurally the same as the Blink program we looked at earlier. The main difference (apart from comments) is the first two lines:

```
int ledPin = 13;   // use the "L" led
int pause = 1000;  // ms to pause between transitions
```

These lines are variable declarations. All variables must be declared before use. In this case the variables named **`ledPin`** and **`pause`** are being declared as having a type of integer and assigned an initial value.

Declaring a variable means that we are putting aside a small piece of memory that we will use for saving some data. You can think of it as being like a mail box. The declaration allows us to name the variable and the name allows us to access and change the data stored by the variable. The type refers to both how much memory (how large a mailbox) we need to store the data and what sort of data it is. More on types and memory use in coming weeks.

For this program using variables makes sense. Say we want to change how long the program pauses between turning the LED on and off. Because we have a variable we only need to change a single value at the top of the program, we do not have to look for every occurrence of the number **`1000`** and consider if we need to change it to another number. Likewise if the LED was connected to a different pin then we only have one location to change

the pin number rather than three in the previous program. Using well named variables also makes your program easier to read. The text **ledPin** says much more about what the program is intended to achieve than the number **13**.

Variable declarations follow the format:

```
[modifiers] type_name variable_name [= optional_initial_value];
```

where:

- **modifiers** are optional type modifiers (we will discuss "unsigned" soon). No modifiers are required for these variables.
- **type_name** indicates what sort of a variable this is. An **int** means integer variable and on the Arduino board this is a 16 bit value that can store an integer in the range of **-32,768** and **32,767**.
- **variable_name** is the label given the the variable in this case **ledPin** and **pause**.
- **optional_initial_value** is an optional constant value to initialise the variable

The semicolon delineates the end of the statements.

The position of the variable declaration at the top of the file outside the **setup()** and **loop()** functions signifies the variables as having "global scope". This means that the variable is accessible from code anywhere in this file after it is declared, in any function.

Another Integer Type

The **ledPin** and **pause** variables in the program source code shown above are typed as **int** meaning that they can be an integer into the range of **-32,768** to **32,767**. This is fine for the variable containing the pin number of the LED connection however it is easy to imagine a program that may need to pause for a longer period of time (i.e. longer than 32.8 seconds). The type for the argument passed to the Arduino library **delay()** function accommodates this by being an **unsigned long integer**. This type can hold values from 0 to 4,294,967,295 (2 to the power of 32 minus 1) which is potentially a much longer time to wait (49.7 days!). If we pass a compatible type to the function, such as an **int**, it is automatically changed to an **unsigned long** for us and we do not have to worry.

The **unsigned** keyword is an optional type modifier in the declaration statement. It signifies that the variable will be used for positive integers only. For **long** type the difference is:

```
long a;          // a ranges from between -2,147,483,648 to 2,147,483,647
unsigned long a; // a ranges from 0 to 4,294,967,295
```

Math Operations

You can modify variables with mathematical operations. There are a set of mathematical operators that you can use on numeric variables:

Operator	Use	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo	a % b

The assignment operator = allows us to assign a value, such as the result of a calculation, in a variable. For example to store the result of the multiplication of variables **a** and **b** in a variable named **c** we would write:

```
c = a * b;
```

We can even use the variable **c** on the right hand side, e.g.

```
c = 3;
```

```
c = c + 1; // after this statement c now contains the value 4
```

Another example, this time a complete program, to print out a sequence of numbers 5,10,15 etc to the serial port. Cut and paste the following and try it yourself:

```
// define a variable and initialise to 0
int n = 0;

void setup(void)
{
  Serial.begin(9600);
}

void loop(void)
{
  n = n + 5;
  Serial.println(n);
  delay(500);
}
```

The variable **n** is is initialised to 0 and every time the loop function is called it's value goes up by 5. The **delay()** function call makes the loop run just twice per second, slow enough not to overload the serial port (which has a limited baud rate) and slow enough so we can see the output.

Variable Names, Scope and Lifetime

One last thing to know before getting into the Challenge questions for this week is a bit more information on declaring variables. So far you have seen that variables have a type (such as **int** and **long**), a name and an optional initialisation value.

A variable can have any name you like as long as it is not a keyword already in use by the Arduino programming environment. If the text color of the word changes as you type it then it is a keyword and you should not use it as the name for a variable. If you have a program bug that looks a bit strange make sure you have not used a keyword as a variable name. Easy ones to accidentally use are "true", "false", "new", "min" and "max".

Two other aspects of variables that you need to be aware of are that they have a scope and a lifetime.

The term "scoping" refers to where in the code a variable name can be used. The basic rule is that a variable is accessible from where it is declared to the end of the nearest wrapping code block which is defined by the nearest **{** character before the declaration and whatever is the matching **}** signifying the end of that code block.

For example consider:

```
int a;    // global scope, visible from here to the end of the file

void setup(void)
{
  int b;  // local scope, visible only inside function setup
}

void loop(void)
{
  int c;  // local scope, visible only inside function loop
}
```

The variable **b** can be used only inside the setup function and the variable **c** can be used only within the loop function. Both variables **b** and **c** are known as "local" variables as they are only visible inside the block they are declared in. In contrast variable **a** is a "global" variable and is visible throughout all code in the file.

Variables also have a lifetime depending on where they are declared. If a variable is declared outside any function, like the variable **a** above, then it is has a "static" lifetime. Memory space for the variable is allocated once at the start of the program and is maintained until the program ends. In contrast, variables declared inside a function have an automatic lifetime, they are created automatically once the function is called and freed up when the function ends. If the function is called again they are created again. The keyword "static" is another type modifier that can be used in a variable declaration to indicate a static lifetime.

For example try the following code:

```
int g=1;       // declared outside any function, global scope, static lifet

void setup(void)
{
  Serial.begin(9600);
}

void loop(void)
{
  int a=10;          // local scope, automatic lifetime
  static int s=100; // local scope, static lifetime

  Serial.println(g);
  Serial.println(a);
  Serial.println(s);

  g = g + 1;
  a = a + 1;
  s = s + 1;

  delay(500);      // slow down the output
}
```

The output from this code looks like

```
1
10
100
2
10
101
3
10
102
4
10
103
```

and so on

The program execution can be stopped by selecting the "stop" icon in the IDE menu bar.

The global and local variables **g** and **s** are both static and they retain their values between calls to function **loop**. However variable **a** is redeclared every time the function is called and re-initialised to the same value. The statement adding 1 to the variable is meaningless since at the end of the function the memory space allocated to store the data for the variable named **a** is reclaimed.

Time to tackle the Challenge questions for this week.

If You Want More

Each week this last section of the notes will contain some additional information of interest about embedded programming, Arduino or the board. The section is entirely optional. There is nothing here that you need to know to complete this weeks (or any weeks) questions.

Arduino vs C/C++

If you have prior experience with C or C++ you should note that:

- The libraries available are only a subset of the standard C/C++ libraries. In particular, there is no standard I/O library as this library is often omitted when developing for an embedded platform.

- The Arduino programming language is not an independent language like python. It is C++ linked in with the Arduino Integrated Development Environment and a number of support libraries. The language that you will be learning is really a large chunk of the C portion of C++. A more accurate description of the language used in this Challenge is "C programming in the context of the Arduino programming environment".
- The C++ aspect touches us as the Arduino libraries use objects the *Class.member()* syntax to access object methods.

Cross Compiling

The NICTA ed1 board connects to your computer via a USB cable. In embedded programming terms your computer is known as the *development system* and the ed1 board itself is known as the *target*. The development system is where the software is compiled and the target is where the software is run. The thing to realise here is that your computer has a completely different processor architecture to the ed1 board. The CPU inside your computer is most likely either an Intel or AMD device and has an architecture that is commonly known as x86 (for 32 bit devices) or x86-64 (for 64 bit devices). This is completely different from the 8 bit "Modified Harvard Architecture" of the ATmega328 processor on the ed1 board. Differing CPU architectures between development and target systems is quite common in embedded programming. Software development is usually much faster, with nicer tools, on a PC than on the target system (on which it may not be possible at all). The software that really makes it all work is a "cross compiler", that is a compiler that runs on one type of CPU but is compiling software that will run on a different type of CPU. This all happens transparently when you use the Arduino IDE, the software you compile will not actually run on your PC at all but will only run on the ed1 board or a similar device.

Embedded Systems 2011
Integers, Branching and Digital Logic on the ed1

Control Structures

The programs you have written so far are constrained by the Arduino sketch structure of the **`setup()`** and **`loop()`** functions. The statements within **`setup()`** run once from top to bottom followed by the statements inside the **`loop()`** function which also run through from top to bottom but then repeat indefinitely.

Control structures allow your code to make decisions about whether or not to run chunks of code or repeat them.

Branching with "if"

The **`if`** statement allows your code to branch with the flow of execution directed according to the evaluation of an expression. The syntax of the **`if`** statement is shown in the code example below. The **`else`** is optional. In this example, if the **`expression`** evaluates to be non-zero, **`statement1`** will be invoked; otherwise the else clause will be executed, and **`statement2`** will be invoked.

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

The evaluated **`expression`** is tested as **`true`** if the expression evaluates to a non-zero amount, if the expression evaluates to zero then the result is **`false`**. The easiest way to think about this is that zero means **`false`** and anything else means **`true`**.

Expressions commonly evaluated by the **`if`** statement use comparison operators. These are shown in the table below, all evaluate to **`1`** for **`true`**, and **`0`** for **`false`**.

Operator	Meaning
==	Equal
>	Greater
>=	Greater or equal
<	Less than
<=	Less than or equal
!=	Not equal

Be careful not to use a single **=** when you mean **==** as a single **=** character means assignment not equality even in an **`if`** statement

If you want to execute more than one statement in either branch, then you must enclose the sequence of statements with curly brackets, for example:

```
if (expression) {
    statement1;
    statement2;
 } else {
    statement3;
    statement4;
 }
```

Logical Operations

You can combine expressions using logical operators to perform multiple tests. Use brackets to ensure that the operations are carried out in the order in which you expect (which is called *precedence*).

Logical Or

Operator is **||**

For example **a** || **b**, which is true if **a** or **b** are true, otherwise false

Logical And

Operator is **&&**

For example **a** && **b**, which is true if both **a** and **b** are true, otherwise false

Logical Not

Operator is **!**

For example **!a**, which is true if **a** is false, false if **a** is true

Notes on Use of Logical Operators

- Variables **a** and **b** could be expressions (expressions would usually be wrapped in brackets to ensure precedence).
- The logical operations short-cut: as soon as the result is known, the rest of the expression is not evaluated. This is important if any of the expressions have side effects.
- C++ has a set of complex precedence rules; it is usually easiest, and good programming practice, to use brackets to make your meaning clear to the compiler if you are combining logical operators. In general, **!** binds more tightly than **&&** which binds more tightly than **||**; and all logical operators bind less tightly than the other non-assignment operators encountered in this course.

Examples of Use of the if Statement

Expressions, Brackets and Code Blocks

The expression evaluated by the **if** statement needs to be wrapped in brackets. Single line code blocks do not need to be delimited by curly brackets. For example:

```
if (a < 5)
  Serial.println("a is less than 5");
else
  Serial.println("a is greater than or equal to 5");
```

More complex expressions can be easier to read with extra brackets to ensure precedence but still require an outer most set of brackets. The **else** clause is optional. Multi-line code blocks need wrapping in curly brackets:

```
if ( (a < 5) && (a > 0) ) {
  Serial.print("a is less than 5");
  Serial.println(" and greater than 0");
}
```

You can nest **if** statements. Careful indentation should help you work out later what was your intention when debugging.

> Indentation is not required by the C++ compiler but if you do not indent your program then it may be practically impossible to read.

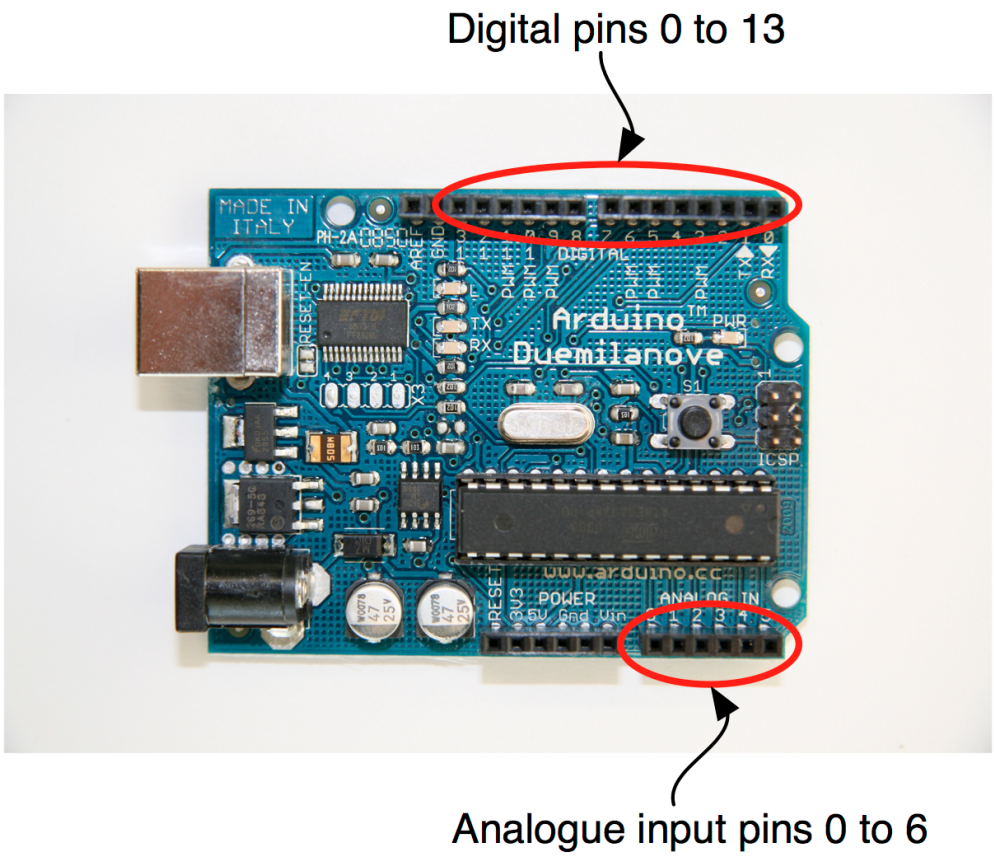
```
if ( (a < 5) || (b < 5) ) {
  Serial.println("at least one of a of b is less than 5");
  if ( (a < 5) && (b < 5) ) {
    Serial.println("in fact both a and b are less than 5");
}
```

Long conditional expressions can wrap over lines but use indenting carefully to make it easy to read. For example this **if** statement extends over two lines:

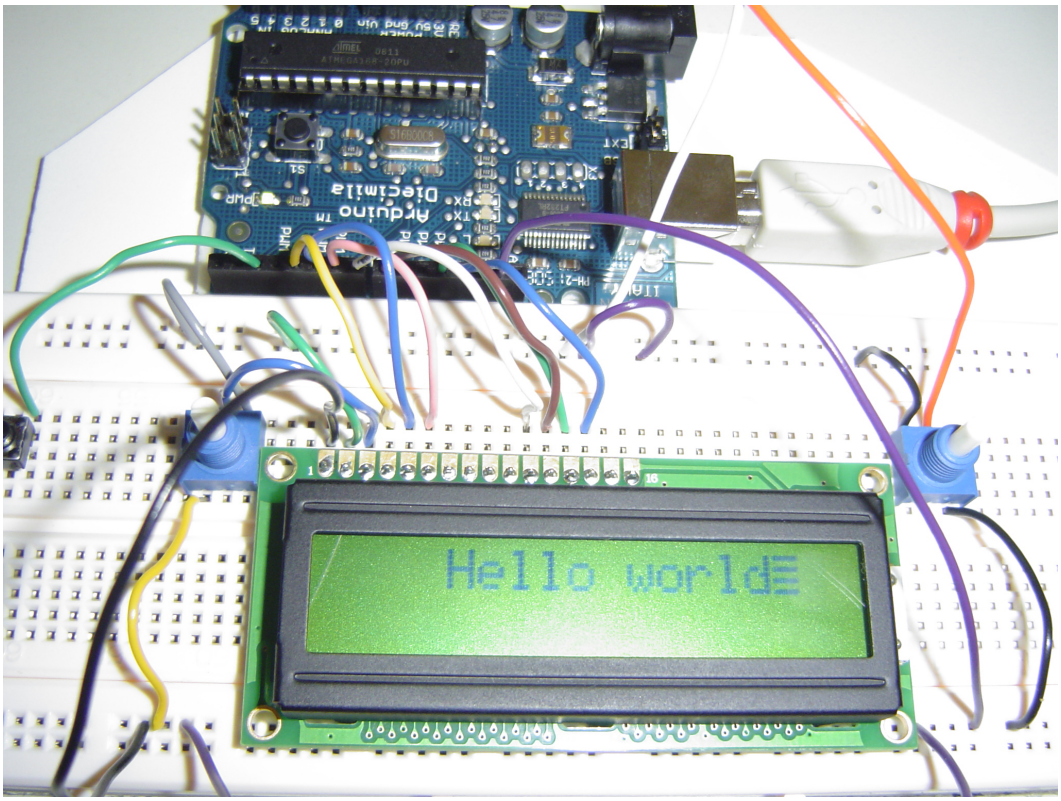
```
if ( (room_temperature > 30) &&
     (heater_status == HIGH) ) {
  turn_heater_off();
}
```

Peripherals on the NICTA ed1

In Arduino pins are either classified as "analogue input pins" or "digital pins". Each type of pin has a separate numbering scheme. For the microcontroller used on the ed1 the analogue pins range from 0 to 8 and the digital pins range from 0 to 21. These pin numbers correspond to physical connection points on a traditional Arduino board. For example, below is a picture of an Arduino Duemilanove board, the digital pins are the row of black connection points labelled "DIGITAL" running on the top of the board and the analogue input pins are the black connection points labelled "ANALOG IN" on the bottom of the board. (The other connection points are for power, ground and other electrically useful things to connect to. Not all pins brought out on all boards)



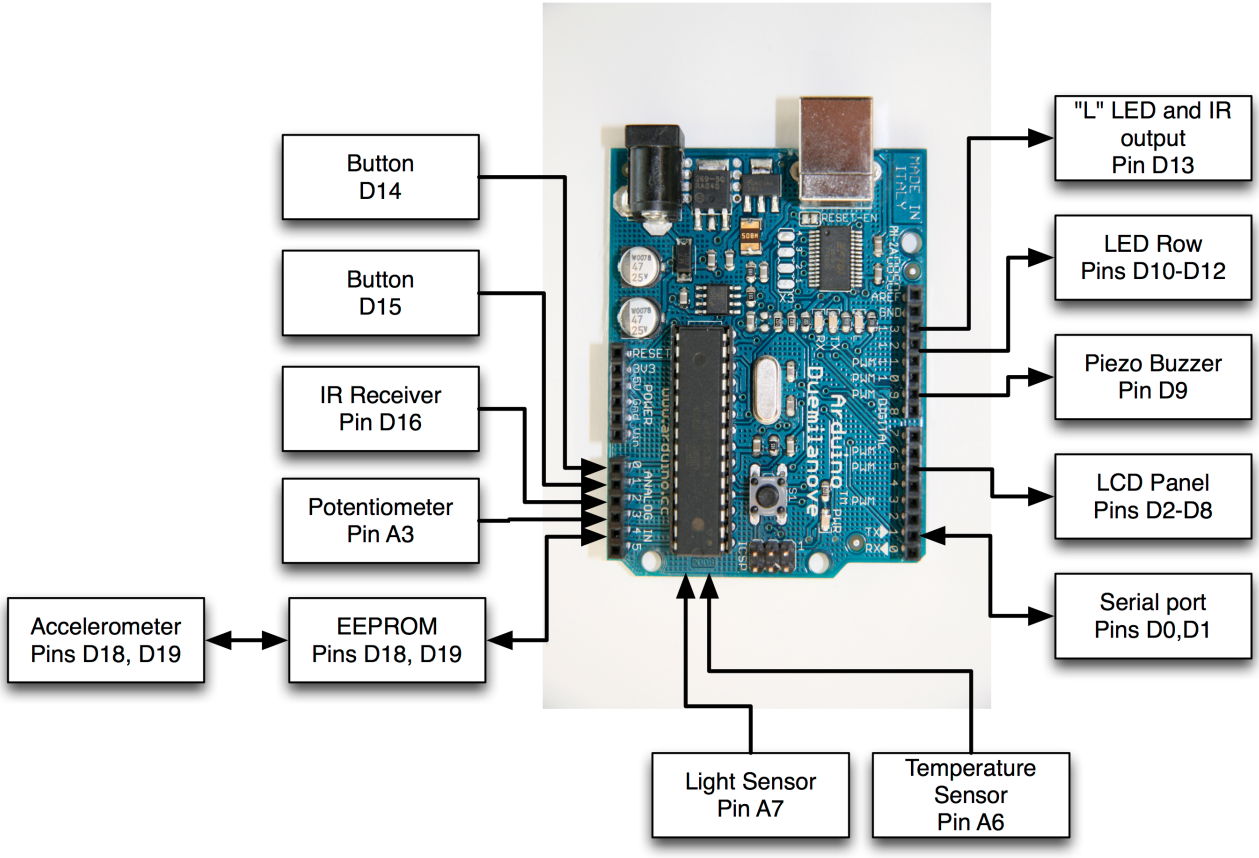
Typically, when using one of these boards, you would first wire up some external components and then write your software referring to the appropriate pin numbers connected to the external parts. For example here is a picture of an Arduino board connected to potentiometers and an LCD panel using a breadboard and wires.



The code running on the Arduino board shown above knows (via constants in the program) which pins have been wired to the LCD panel and potentiometers.

On the NICTA ed1 board all these pins are permanently connected (*hard wired*) to devices on the board. The Arduino equivalent pin number is labeled on the board next to the component to make programming easier. To avoid confusion the text printed on the NICTA ed1 board uses A*n* to signify analogue input pin *n* and D*n* to signify digital pin *n*.

Diagrammatically the NICTA ed1 board is the equivalent of this:



The use of each Arduino pin equivalent on the NICTA ed1 is listed in the table below:

Digital Pin	Use
0	(RX) Serial input
1	(TX) Serial output
2	LCD panel pin 5
3	LCD panel pin 6
4	LCD panel pin 7
5	LCD panel pin 1
6	LCD panel pin 2
7	LCD panel pin 3
8	LCD panel pin 4
9	Piezo buzzer
10	Data clock for shift register (LEDs)
11	Latch for shift register (LEDs)
12	Data for shift register (LEDs)
13	"L" LED and IR output LED
14	Pushbutton 1
15	Pushbutton 2
16	IR Receiver
17	not used*
18	EEPROM and Accelerometer
19	EEPROM and Accelerometer
20	not used*
21	not used*

Analogue Pin	Use
0	not used*
1	not used*
2	not used*
3	Potentiometer
4	not used*
5	not used*
6	Temperature sensor
7	Light Sensor

* not used for this pin type. The analogue and digital pin ranges actually overlap. If you want to know more read the optional notes for this week

Using Arduino Digital Pins

You saw commands to use digital pins last week when controlling the "L" LED. The `pinMode()` function sets a digital pin to be an input or output; `digitalRead()` returns the value (`LOW` or `HIGH`) of a pin; and `digitalWrite()` sets the value.

This week you will use the Piezo buzzer, LCD panel and Accelerometer. You will use the digital inputs and analogue input pins in coming weeks.

Piezo Buzzer

The Piezo buzzer is an electronic device that vibrates to make a noise. "Piezo" stand for piezoelectric. The device contains a plate that changes shape when a voltage is applied. By rapidly turning the voltage on and off at the right frequency the device will generate a tone.

The Piezo buzzer is connected to digital pin 9. By configuring pin 9 as a digital output and rapidly switching it from **HIGH** to **LOW** and back again you can make the piezo generate sound. The following code feeds a "square wave" to the Piezo buzzer at the frequency of around 250Hz.

```
/*
 * Vibrate the Piezo buzzer
 */
int PIEZO = 9;
int pause = 2000; // in microseconds will give us a 250Hz tone

// 250Hz is 250 cycles per second. The duration of a cycle is called the
// period. 1 period = 1/cycles, hence 250Hz has a period of 4ms.
// Each period has an ON half and and OFF half so the delay for each half
// period is 2ms or 2000 microseconds.

void setup()
{
  pinMode(PIEZO, OUTPUT);
}

void loop(void)
{
  digitalWrite(PIEZO, HIGH);
  delayMicroseconds(pause);
  digitalWrite(PIEZO, LOW);
  delayMicroseconds(pause);
}
```

Try out the code on your board. Have a play with the **pause** variable to vary the tone frequency. You may notice that setting **pause** to a large number may not have the desired effect. The **delayMicroseconds()** function is similar to the **delay()** function except that it delays execution for the given number of microseconds not milliseconds (where a microsecond is a one millionth of a second). However the **delayMicroseconds()** function only works reliably for values up to 16383 microseconds, any longer and you should use the **delay()** function instead.

The LCD Panel

The LCD panel is an alphanumeric display with 2 rows of 16 characters. When you first turned the board on it may have displayed the text **NICTA ed1**. Using the LCD panel from within your code requires the use of the Arduino **LiquidCrystal** library.

Libraries and the "#include" Statement

Arduino comes with a number of useful libraries that supply extra functions for making your coding life easier. One of those libraries is called the **LiquidCrystal** library and to make use of it we need to tell the compiler about the functions provided so the code will compile. You do this with the **#include** macro and to use the **LiquidCrystal** library you need to put the following code near the top of your program:

```
#include <LiquidCrystal.h>
```

The ".h" at the end of the library name signifies it as a header file, containing definitions and declarations for a library. To save typing the Arduino IDE menu command **Sketch-> Import Library** will insert the appropriate **#include** for you. This is not a program statement but a compiler directive and there is no semicolon on the end of the line.

Using the LiquidCrystal Library

Information on functions provided by the **LiquidCrystal** library can be found in the Arduino reference. You can access a local version of the reference via your IDE using the **Help->Reference** menu.

The library must be initialised with a declaration:

```
LiquidCrystal lcd(rs, rw, enable, d4, d5, d6, d7);
```

which declares a variable named **lcd** to be of type **LiquidCrystal**. The parameters, **rs** and so on, are the labels for the different electrical connections on the LCD panel that are connected to Arduino pins. For the NICTA ed1 these parameters are printed on the board adjacent to the LCD display so do you don't have far to look to find them. Make sure you use the correct values in your code. Any example code you see on the Internet will most likely use different values that correspond to the physical wiring of their (different) board.

Once you have a variable declared correctly the next step is to initialise it with the dimensions of the LCD panel. This is done with a call to the **begin()** function which is accessed via the variable name and usually called from within the **setup()** function. The LCD panel on the NICTA ed1 board has 16 columns and 2 rows so, using the **lcd** variable declared above, the call to **begin()** looks like:

```
lcd.begin(16, 2);
```

Once initialised you can write values to the LCD panel in much the same way as the serial port. The **lcd.print()** command accepts strings and numeric values with an optional base modifier. After using the **lcd.print()** function the "cursor" is automatically placed at the next character location. For example:

```
lcd.print(10);
lcd.print(20);
```

shows

```
1020
```

on the LCD panel.

The function **lcd.clear()** clears the panel and resets the cursor to the home position (top left hand corner). The **lcd.setCursor()** function moves the cursor to a specified location relative to home (top left hand corner, 0, 0).

Experiment with your own code to become familiar with the LCD panel. A sample program showing use of the LCD panel is shown below:

```
/*
 * LCD panel example, show hex numbers
 */

#include <LiquidCrystal.h>

// declare LCD interface
LiquidCrystal lcd(6,7,8,2,3,4,5);

void setup(void)
{
  // initialise with size of ed1 LCD panel
  lcd.begin(16, 2);
}

void loop(void)
{
  // loop variable
  static int i;

  // print HEX value at that position in the row
  lcd.clear();
  lcd.setCursor(i, 0);
  lcd.print(i, HEX);

  // increment and reset if needed
  i = i + 1;
  if (i == 16)
    i = 0;

  // if we call lcd.clear(); lcd.print(); too fast the panel will not be readal
  delay(500);
}
```

Accelerometer on the NICTA ed1

The 3-axis accelerometer on the ed1 measures "proper acceleration" (acceleration relative to freefall) along each of the 3 perpendicular axes. You can think of it in terms of "g force" as used to describe the force felt by pilots in an aircraft. Standing still on the ground you experience 1g of force from the ground holding you up. In freefall you would experience 0g. The 3 readings provided by the accelerometer depend on how it is oriented. We will assume the board is standing flat on a horizontal surface, in which case the Z axis is vertical (positive upwards), the Y axis (positive) is towards the USB port, and the X axis (positive) is towards the LCD display. At rest, and after calibration, the X, Y, Z readings should be approximately 0, 0, 64 which corresponds to 0g, 0g, 1g. If the board were in free fall, all the measurements will be zero. If you accelerate the board in the Z direction (i.e. up) the Z output will increase, or in the negative Z direction (ie. Down) the output will decrease. Since the measurements are relative to free fall the easiest way to vary the readings is simply to tilt the board in any direction.

The accelerometer is a complex device that can be set to operate in different modes. The mode we will use has outputs for each of its X, Y and Z axes as a single byte of data; interpreted as a type of **`char`** ranging from **`-128`** to **`127`**. Each increment corresponds to a value of g/64. You can simply read the output of the accelerometer read functions into a type of **`int`**.

Calibration is required every time the accelerometer has been powered on.

Installing the Library for NICTA ed1

The EEPROM and the accelerometer built in the NICTA ed1 have a library interface.

This library is available for download from /static/files/embedded/ed1_1August2011.zip.

To install you must first locate the directory in which the Arduino software stores user written programs on your machine. Under MAC OS X the default location in the file system is under the directory **Documents/Arduino/** in your home account, under Windows it is **My Documents\Arduino**. Within that directory create a new directory named **`libraries`** (if it does not already exist). Inside the libraries directly unzip the downloaded NICTA ed1 library. You should end up with the library in the directory:

`Arduino/libraries/ed1/`

To test, quit and restart your Arduino IDE. If the library is installed in the right place you will see it listed as an available library for importing under the menu **`Sketch->Import Library....`** You will also see some new example programs listed under **`File->Examples->ed1`**. You can test by trying to run each of the examples. The accelerometer example configures, calibrates the accelerometer chip and displays X,Y,Z readings on the LCD panel. The EEPROM library example will attempt to write and read a single floating point value from the EEPROM. More on the EEPROM functions in later weeks.

Using the Library

To use the NICTA ed1 library you need the following **`#include`** directives at the top of your program:

```
#include "Wire.h"
#include "ed1.h"
```

Using the NICTA ed1 Accelerometer Library

Load and run the example program **`File->Examples->ed1->Read_XYZ`** which prints out the readings from the 3 axis on the accelerometer. When the program is running, lift the board up (by the edges) rotate it in different ways and observe how the outputs vary. Note that even when the board is sitting completely at rest that there are small random variations to the readings (called noise).

The available functions are:

`int Accel.begin(int)`	initialise part
`char Accel.readX(void)`	read X axis acceleration
`char Accel.readY(void)`	read Y axis acceleration
`char Accel.readZ(void)`	read Z axis acceleration
`int Accel.calibrate(void)`	calibrate accelerometer

The **`int Accel.begin(int)`** initialises the part. It must be called with the appropriate mode parameter which is **`ACCEL_MEASURE_2G_MODE`**. The function returns a value of type **`int`** which is set to **`0`** (zero) if the part is responding

or to **-1** if the part is not responding. An example of how you could call it is:

```
if (Accel.begin(ACCEL_MEASURE_2G_MODE) != 0)
  exit(0) // error part is not responding
```

The functions **Accel.readX()**, **Accel.readY()** and **Accel.readZ()** all return signed 8 bit values of the measurement of acceleration in the X, Y and Z axis respectively. For the 2g mode these will be in the range **-128** to **127** where **64** is 1g and **0** is 0g. You can simply assign the output of these functions to an **int**.

The function **Accel.calibrate()** performs a calibration of the accelerometer. Calibration is required every time the power is cycled on the part. It is quick and you would normally call it just after **Accel.begin()** during the **setup()** function. Make sure the board is placed on a level surface when the function runs. After calibration the X and Y axis will return values around 0 (no G) and the Z axis will return 64 (1g). Before calibration the X,Y and Z axis acceleration values could vary from those amounts quite a bit. The **Accel.calibrate()** function returns a value of type **int** of zero and need not be checked.

Suggested Structure for Your Sketches and Common Errors

Last thing for this week are comments about writing source code.

As Arduino is essentially C++ it is remarkably easy to write code that is syntactically correct, runs as intended but is virtually impossible to read. It is strongly suggested that you attempt to lay out your code to make it easy to read. This will help you when debugging and make it easier for a tutor to make suggestions at where you may have gone wrong.

Use sensible comments, indent code blocks consistently and use brackets to ensure precedence. There is a menu function **Tools->Auto Format** that can help out.

An example of a suggested layout for your code is shown below.

```
/*
 * LCD panel example, show hex numbers
 */

#include <LiquidCrystal.h>

// declare LCD interface
LiquidCrystal lcd(6,7,8,2,3,4,5);

// loop variable
int i;

void setup(void)
{
  // initialise with size of ed1 LCD panel
  lcd.begin(16, 2);
}

void loop(void)
{
  // print HEX value at that position in the row
  lcd.clear();
  lcd.setCursor(i, 0);
  lcd.print(i, HEX);

  // increment and reset if needed
  i = i + 1;
  if (i == 16)
    i = 0;

  // if we call lcd.clear(); lcd.print(); too fast the panel will not be readal
  delay(500);
}
```

Compiler directives should be first, in the order of **#include** lines followed by **#define** lines (you may not have used **#define** yet and can ignore it for now). Remember that these lines are not terminated by semicolons. Compiler directives are followed by global variable declarations, these are either variables you have created for your code or variables you need to declare for the libraries you are using. Functions come last, start with your own and end with the

mandatory **`setup()`** and **`loop()`** functions. Within a function start with declarations for any local variables and remember to wrap any code blocks in curly brackets.

When trying to understand messages from the compiler remember that the messages can be cryptic and to try and concentrate on just the first one or two as the rest may be all related. When in doubt look in your code for:

- Missing semicolons at the end of statements
- Missing curly brackets at the beginning or end of code blocks
- Missing brackets in comparison or Boolean expressions
- Variables that have not been declared or are mistyped
- Function calls with missing parameters
- Using = when you really meant ==

Use indentation, it is not required in C/C++ for the program to compile but it does make it much easier to read and helps when catching bugs like missing curly brackets.

If You Want More

Like last week the following notes are completely optional. You do not need any of the information in this section to complete the Challenge questions.

Arduino Libraries

The programs you have seen in these notes so far have appeared to be self-contained but this is not the case. They are modified transparently by the Arduino IDE before they are compiled to turn them into valid C++ code. These modifications add header files that define the signatures of the **`setup()`** and **`loop()`** functions, declare the Arduino library functions (such as **`digitalWrite()`** and **`Serial.print()`**), add the so-called built-in predefined constants, and so on. The compiler then knows about these functions and can check types and usage.

Declarations for the built in libraries are included by default but extensions, like the LiquidCrystal library are not. If you want to use one of these libraries then you need to include the definitions for the library explicitly with the appropriate **`#include`** directive.

Arduino Pin Numbers

The analogue and digital pins actually overlap as the analogue input pins can be re-purposed as digital pins. In building the NICTA ed1 we actually used some of the analogue pins for digital IO:

Analogue Input Pin	Analogue Input Use	Digital Pin Number	Digital IO Use
0	None	14	Pushbutton 1
1	None	15	Pushbutton 2
2	None	16	IR Receiver
3	Potentiometer	17	None
4	None	18	Two wire interface
5	None	19	Two wire interface
6	Temperature sensor	20	None
7	Light Sensor	21	None

Four of the pins are brought out to a header at the bottom right of the ed1 board so they can be used for external connections to the board. Unfortunately we did not think of a header for the I^2C interface and we have since been asked for it many times!

A board schematic is available on the NICTA web site from a link at the bottom of this page:

http://www.nicta.com.au/ed1

Accelerometer

Much more formal information on acceleration and accelerometers on wikipedia:

http://en.wikipedia.org/wiki/Accelerometerhttp://en.wikipedia.org/wiki/Proper_acceleration

The complete datasheet for the accelerometer is available here:

http://www.freescale.com/files/sensors/doc/data_sheet/MMA7455L.pdf

Much of the data sheet is taken up with the interface and with a discussion on modes not available through the ed1

library. These modes, such as freefall detection, can be accessed but you will have to do the work of either modifying the library or writing your own code to access the device.

Sponsors

Embedded Systems 2011
Looping, Numbers, Bits, Bytes, Functions and LEDs

Looping with For and While

The **if** statement gives us a mechanism for selectively executing a block of code once based on the outcome of a conditional expression. What happens if you want to execute that block multiple times? The **while** and **for** statements are control structures that let you build loops into your code to repeatedly execute sections of code.

The syntax of the **while** statement is:

```
while (expression)
   statement;
```

The expression is evaluated. If it is non-zero then the statement(s) in the body of the loop are executed once and the expression is evaluated again. The cycle continues until the expression evaluates as zero. As with the **if** statement, the statement can be a code block of multiple statements delimited with curly brackets.

Here is an example:

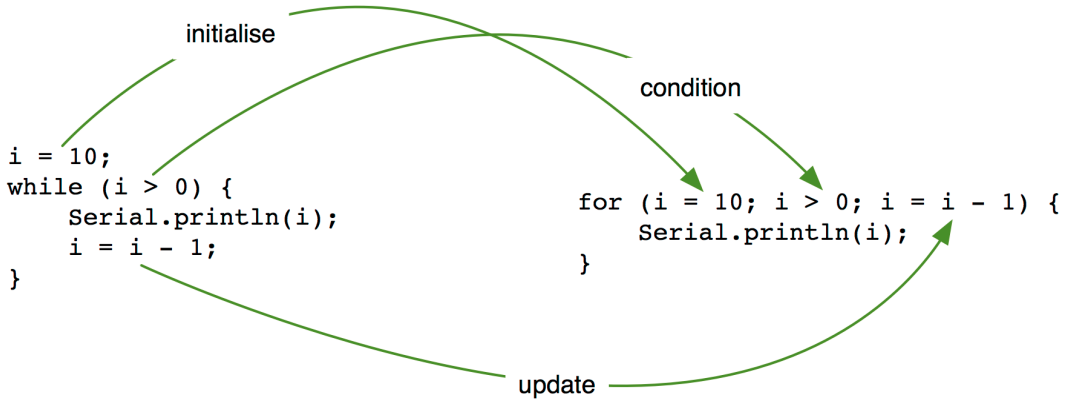
```
void setup(void)
{
    int i;

    Serial.begin(9600);

    i = 10;
    while (i > 0) {
        Serial.println(i);
        i = i - 1;
    }
}

void loop(void) {}
```

In this code the **while** loop evaluates the expression **(i > 0)** and if the result is non-zero executes the statements to print out the value of the variable **i** and then decrease the value of **i** by one. The expression **(i > 0)** is evaluated again and the loop continues until the expression evaluates to zero.

Copy the program into the Arduino IDE, compile and run. You should get the following output (remember to press the "Serial Monitor" button on the IDE):

```
10
9
8
7
6
5
4
3
2
1
```

The initialisation/test/update sequence for the variable **i** in the example above is so common that C has a special statement for it: the **for** loop.



Here is the same program as before, but using a **for** loop.

```
void setup(void)
{
    int i;

    Serial.begin(9600);

    for (i = 10; i > 0; i = i - 1)
        Serial.println(i);
}

void loop(void) {}
```

Any of the three parts, *initialise*, *condition*, or *update* can be omitted. Thus simple infinite loops are:

```
for (;;)
  ; /* loop forever */ ;

// or for the while statement
while (1)
  ; /* loop forever */
```

Some Examples

The **while** statement is a useful mechanism to use to wait for an event. Say, for example, you want to wait for the X axis of the accelerometer to return a value of less than 5. You could use a **while** loop to repeatedly read the X axis value from the accelerometer, code execution will not move past this loop until the conditional expression evaluates to **True**:

```
while (Accel.readX() >= 5)
  ;  // do nothing but wait
```

The **for** statement excels as a mechanism for creating sequences. For example to flash the LED "L" three times in a row:

```
int i;
int ledpin = 13;

for( i = 0; i < 3; i = i + 1 ) {
  digitalWrite(ledpin, HIGH);
  delay(200);
  digitalWrite(ledpin, LOW);
  delay(200);
}
```

Loop Checklist

When debugging you loops try to consider these four points and when determining if your code will do as you intend:

- **starting** - Are there any variables we need to setup before we start a loop? This is often called *loop initialisation*.
- **stopping** - How do we know when to stop? This is called the *stopping* or *termination condition*.
- **doing** - What are the instructions we want to repeat. This will become the body of the loop control structure.
- **changing** - Do the instructions change in any way between each repetition. For example, we might need to keep track of how many times we have repeated the body of the loop, that is, the number of iterations.

The break and continue Statements

When using loop control structures such as `for` or `while` you have access to statements to finish the loop processing prematurely.

The two statements are:

- `break` causes control to pass to the first statement *after* the loop body.
- `continue` causes the loop condition to be retested, and if it is still OK, the loop body will be started again.

For example:

```
while (expression) {
  if (a)
    continue;  // abandon this iteration,
               // jump back to re-testing loop condition expression

  if (b)
    break;     // leave loop,
               // jump to the first statement after the loop
}
```

These statements can sometimes save you from horribly nested `if else` combinations within loop bodies.

Number Representation

As the Arduino Programming Language is based on C++ the variable types are closely related to the physical aspects of storing information in a computer system. The smallest unit of storage is a **bit** (short for binary digit), which is a binary number and represents the smallest part of physical memory that can either be **on** or **off**. Eight bits grouped together are a byte and computers typically handle data in groups of bytes.

Writing numbers in binary is tedious as each single byte requires eight characters to write it down. An easier representation is to use hexadecimal, which is base 16, and essentially gives you a shorthand method of representing a byte with just two characters (One hexadecimal number for each 4 bit half byte or *nibble*). The compiler treats any number that begins with **`0x`** (zero followed by an x) as hexadecimal. For hexadecimal the digits above **9** are written as **`a`**, **`b`**, **`c`**, **`d`**, **`e`** and **`f`** and the case does not matter. Binary numbers are representing with a leading **`B`** character but only up to a single byte (i.e. a **`B`** followed by eight **`1`** or **`0`** characters) can be represented.

For example:

Decimal	Hexadecimal	Binary
0	0	B00000000
1	0x1	B00000001
2	0x2	B00000010
3	0x3	B00000011
4	0x4	B00000100
5	0x5	B00000101

6	0x6	B00000110
7	0x7	B00000111
8	0x8	B00001000
9	0x9	B00001001
10	0xa	B00001010
11	0xb	B00001011
12	0xc	B00001100
13	0xd	B00001101
14	0xe	B00001110
15	0xf	B00001111
16	0x10	B00010000

Shown below is a table of the Arduino language standard byte groupings for integers and the consequent number of bits used for storing data in those types. If you have programmed in C or C++ on another platform you may be surprised that an integer in Arduino is only 2 bytes instead of the 4 used on many other platforms.

Type name	Alternative name	Size (bytes)	Size (bits)	Range
signed char		1	8	-128 to 127
unsigned char	byte	1	8	0 to 255
signed int		2	16	-32768 to 32767
unsigned int	word	2	16	0 to 65535
signed long int		4	32	-2 147 483 648 to 2 147 483 647
unsigned long int		4	32	0 to 4 294 967 295

Floating Point

The numbers you have been dealing with so far have been integers but C also provides mechanisms for floating point numbers. Internally a floating point number consists of a sign bit, a coefficient (or significand) and an exponent. The `float` type uses 24 bits for the coefficient, seven bits for the exponent and a sign bit. In C there is also a double precision floating type, which uses 53 bits for the coefficient but this is not supported by the compiler for the microcontrollers used by Arduino. You can declare a variable to be of type `double` but the variable will actually be a `float`.

Implementation details for floating point vary between different manufacturer's hardware. The ATmega328 used on the NICTA ed1 board has no built-in floating point unit (hardware that supports direct operations on floating point numbers) and the compiler provides a library of code for floating point operations.

For the Challenge there are a two things you need to know about using floating point; using floating point is not precise and operations using them are slower than using integers.

Precision is an issue because of the internal representation of a coefficient and an exponent. Quite often a number can not be represented exactly, but most are represented by an approximation to the true value. As a result you **cannot** compare floating point numbers for equality with any reliability and you have to allow for a little slack. If you need to test a floating point number, such as in an `if` statement, then use operators such as greater than or less than rather than equality.

The lack of a floating point unit means that mathematical operations involving floats are broken up into a potentially large number of integral operations. This is done automatically by the compiler and built in libraries. The end result is code that takes more room to store in the non-volatile program memory and runs slower than operations involving integers. Even so, in the code that you write, unless it is a particular situation where speed is important, most of the time you will not notice the slow down.

If you want to know more about floating point there is a good article in Wikipedia, http://en.wikipedia.org/wiki/Floating_point.

Casts

In C you can force an expression to be converted to a different type with a *cast*. A cast looks like this:

```
(type) expression;
```

You can think of a cast as a conversion from one type to another.

Some things to note:

- If you cast a **float** or a **double** to an **int**, any fractional part is thrown away and the number is rounded down to the nearest integer
- If you cast a **long** to a **float** then some precision may be lost. The **long** has 32 bits of precision and the **float** has 24, the least significant part will be discarded

In some situations casting is done automatically for you. If an arithmetic expression involves both floating point and integers, the integers will be cast to floating point before use. Rather than second guess that the compiler will automatically do what you want it is better to cast explicitly and put the cast in yourself. This also has the advantage of making your code more readable by others. For example,

```
float a;
int b;

a = (b * 2000) / 3.456;         // will the b * 2000 overflow?
a = ((float)b * 2000) / 3.456; // explicit casting avoids potential overf.
```

Bitwise Operators

There are a set of operators that manipulate the individual bits of data that are storing a data type. Most often (and certainly in the Challenge) you will only manipulate bits of a integer data type. Remember each bit is simply an on or off, **1** or **0**, value. Each integer is a row of these bits and these operations compare and move these individual bits around and, hence, change the value being stored.

If you have trouble with this section of the notes please ask for help in the forums. For programmers new to C++ based languages bitwise operators can be the hardest part of the language to understand.

The operations are:

- Left shift (**<< n**): Shift the number to the left by **n** number of binary bit positions, putting zeros in at the right (each bit position is equivalent to a multiply by two)
- Right shift (**>> n**): Shift to the right by **n** number of binary bit positions (each bit position is equivalent to a divide by two). If the left hand argument is of signed type, the value of the most significant bit is shifted in at the left; if it is unsigned, then zeros are shifted in
- Bitwise and (**&**): **a & b** yields a number that has one bit where both **a** and **b** have one bit
- Bitwise xor (^): **a ^ b** yields a number that has one bit wherever **a** or **b** have one bit but not both
- Bitwise or (|): **a | b** yields a number that has one bit wherever **a** or **b** have one bit or both
- Bitwise not (~): **~a** flips all **1** bits in **a** to **0**, and all **0** bits to **1**.

The Row of LEDs

In weeks 1 and 2 you wrote code to control the LED "L" which is directly connected to pin 13. There are eight other LEDs on the ed1, marked **LED1**, **LED2** through to **LED8**. These LEDs are controlled through a digital logic device known as a shift register and are not directly connected to output pins on the microcontroller. To turn these LEDs on and off requires writing a byte of data to the shift register where each bit in the byte corresponds to whether or not an LED should be on or off.

The shift register is controlled by three digital outputs:

Digital Pin	Register Line
11	Latch
12	Data

10	Clock

The **Latch** and **Clock** lines are for control of the device with the data to be drive the LEDs on and off going out the **Data** line.

You will not be surprised to find out that software control of the shift register is made easy by a number of Arduino built in functions. For example, a program that switches LED8 on and the rest of the LEDs off is shown below:

```
/*
 * Turn on LED8
 */

int LATCH = 11;
int DATA = 12;
int CLOCK = 10;

byte a = 1;

void setup()
{
  // you need to set the mode of the
  // pins controlling the LEDs
  pinMode(LATCH, OUTPUT);
  pinMode(DATA, OUTPUT);
  pinMode(CLOCK, OUTPUT);

  // each call to write a value to the LEDs
  // requires these three commands
  digitalWrite(LATCH, LOW);
  shiftOut(DATA, CLOCK, LSBFIRST, a);
  digitalWrite(LATCH, HIGH);
}

void loop(void)
{
}
```

The work is being done by the **shiftOut()** function. The first two arguments to **shiftOut()** are the pin numbers for data and clock connections to the shift register. The third and fourth arguments are a significant bit flag and the byte of data itself. Each bit value in the data byte controls one LED light. If a bit is set then the corresponding LED is turned on and if the bit is not set then the LED is OFF. The third parameter can have two values **LSBFIRST** or **MSBFIRST** which indicate which bit of the data byte is written to the shift register first. The effect is to control which way around the data byte is displayed on the LEDs. For the example code above, with a byte of **0x01** written out to the shift register, LED8 is turned on. If the parameter **MSBFIRST** is passed to the **shiftOut()** function in the above code then LED1 would have been turned on and not LED8. Use either **MSBFIRST** or **LSBFIRST** as appropriate in your code.

The **shiftOut()** function will only be effective if the **Latch** control line is turned LOW first and then returned to HIGH afterwards. You need all three commands as a set to control the LEDs. Since this code must always be grouped together you should consider wrapping it as a function.

Functions

A function is an independent, named piece of code that performs a specific operation. It can be run by referring to it by name and providing any required information as arguments. You have seen the Arduino functions **delay()** etc.

If you have some code that is used over and over then it can be udeful to wrap it in a separate function. As you saw in week 1 with **setup()** and **loop()** a function can either return a value of a particular (declared) type or nothing. A function can be passed one or more *arguments* or none at all, it all depends on the function declaration. If the function has been written to expect three arguments of a particular type then they must be supplied or the compiler will output an error message.

An example of a function is shown below. This function is passed two integers, of type **int**, and returns the sum of those numbers also as an **int**:

```
int sum(int a, int b)
{
  int c;

  c = a + b;
  return c;
}
```

Remembering what you know about integer data types you should consider whether or not this function would be more useful if all the data types were **long** instead.

The return statement

The **return** statement terminates the execution of a function and returns control back to the calling function. If the function was declared as returning a data type then the **return** statement should include an appropriately typed value to send back to the calling function, if the function does not have a return type (i.e. returns type **void**) then simply call **return** with no arguments.

A **return** can be called at any time and control is immediately passed back to the calling code, no further execution of code below the **return** statement takes place. Using **return** in this fashion can be handy for avoiding nested **if else** statements. For example the following function uses **return** in the body of **if** statements to avoid nesting and indentation:

```
void foo()
{
  if (expression 1)
    return;

  if (expression 2)
    return;

  // other statements
  return;
}
```

Stopping Program Execution

When power is applied to the ed1 board then the **loop()** function is run over and over. What happens if you want to halt execution of your program? You have two choices; you can put your program into an infinite loop using one the examples with **for** or **while** that you saw before or you can call **exit(0)** which then causes you program to exit (and it will not run again utill the board is reset).

If You Want More

Like last week the following notes are completely optional. You do not need any of the information in this section to complete the Challenge questions.

Octal

A less common but valid way of representing numbers is in octal which is base 8. The table shown in the notes this week could have an additional column showing the equivalent representation of numbers in octal. In practice this is used much less that hexadecimal.

For example:

Decimal	Octal	Hexadecimal	Binary

0	0	0	00000000
1	01	0x1	00000001
2	02	0x2	00000010
3	03	0x3	00000011
7	07	0x7	00000111
8	010	0x8	00001000
14	016	0xe	00001110
15	017	0xf	00001111
16	020	0x10	00001000
-1	-01	-0x1	11111111

Two's Complement

Two's complement is a method for representing negative integers. The right most (most significant bit) is used to give the most significant bit negative (instead of positive weighting).

So if you label bits from the least significant bit (zero) to the most significant bit (7 for a **char**, 15 for an **int**, 31 for a **long** - but we'll use **char** in the example) then the value of a number is

$$b_0 2^0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 + b_5 2^5 + b_6 2^6 - b_7 2^7$$

Thus binary **B10000011** is:

$$1 + 2 - 2^7$$

or **-125**.

Minus 1 (**-1**) in two's complement binary notation is **B11111111**.

The most significant bit doubles as a sign bit, when it is set the number is negative and when it is zero the number is positive or has value zero.

Because recording sign effectively uses up a bit of storage the largest positive number represented by a signed type in C++ is only about half that possible with the unsigned type.

See the http://en.wikipedia.org/wiki/Ten's_complement on the Method of Complements if you want more of the background mathematics.

Shift Register

See the ShiftOut tutorial http://www.arduino.cc/en/Tutorial/ShiftOut on the Arduino web site if you want to know more about the underlying electronics of shift registers.

The Tone Function

There is an Arduino library function for moving a pin from **HIGH** to **LOW** and back again over and over automatically. This is useful if you want to generate a particular tone from the Piezo buzzer and want your program to be doing something else at the same time.

The **tone()** command moves a nominated pin **HIGH** and LOW at a specified frequency. The time the pin is **HIGH** is equal to the time it is **LOW** effectively producing a square wave.

Only one pin can be controlled via the **tone()** function at a time.

There are two ways to call the function:

```
byte pin;               // pin on which tone is generated
unsigned int frequency; // frequency of the tone in hertz
unsigned long duration; // play tone for this many milliseconds

tone(pin, frequency);           // variant 1: play tone until notone()
```

```
tone(pin, frequency, duration); // variant 2: play tone for duration
```

If a duration is not specified then `noTone(pin)` must be called to stop the tone on the that pin.

The command is aimed at producing audible frequencies roughly 80Hz to 15kHz. However it can be used for a wider frequency range down to around 31Hz and up to around 65535Hz (maximum unsigned int).

The tone is generated for you by an Arduino library function that sets up an internal hardware timer inside the microcontroller. Every time the timer ticks over the library code moves the up up or down. Rounding errors with the maths to do with setting up the timer mean that you may not quite get the frequency you are after.

Embedded Systems 2011
Arrays, Floating Point and Analogue Inputs

Arrays

So far the variables we have used have all been named individually and explicitly. As with many other languages you have access to arrays for when you want to have a set of variables that are to be treated uniformly.

You can think of an array in one of two ways:

- As a group of variables that can be operated on as a unit, indexed by an integer value, or
- As a mapping from an integer value to some other value.

A common example of an array is a string which is stored as an array of characters terminated by a **NULL** character (a character with a value of zero). Characters in strings are of type **char** which is an eight bit value (as opposed to the 16 bit integer type **int** that you may most of the time).

Square brackets are used both in declaring an array (to give it a dimension), and when using an element from an array (in dereferencing).

```
int a[10];
```

declares an array of ten integers, called **a**.

If the dimension is known from the context it can be omitted. For example:

```
char s[] = "A string";
```

declares an array of 9 characters with the following values:

```
s[0] = 'A'
s[1] = ' '
s[2] = 's'
s[3] = 't'
s[4] = 'r'
s[5] = 'i'
s[6] = 'n'
s[7] = 'g'
s[8] = 0
```

Array indices start at 0 so for the 9 character array in the example above the index of the last value in the array is **s[8]**.

C++ does not provide multi-dimensional arrays you can get the same effect using arrays of arrays.

The #define Macro

When configuring arrays, and other parts of your program, it is sometimes convenient to use a **#define** macro statement to create a symbolic name for a constant value. For example:

```
#define LENGTH (10)

int a[LENGTH];
```

to set the variable **a** to be an array of 10 integers. The compiler's pre-processor reads the **#define** macro then does a text replacement in the rest of its input. In the example above all the **LENGTH** values would be replaced by **(10)** It is important to note that a **#define** macro does not have a semicolon at the end of the line like a C++ statement. Sometimes the Arduino IDE environment will hide this from you and may automatically strip any trailing semicolons. Do not get into the habit of relying on this as the marking system (and indeed any other C++ compiler) does not do this and you will consume one precious marking attempt finding out.

The parentheses around the value **10** are good programming practice but optional in this case. A case where you will get into trouble if you do not have the parentheses is the following:

```
#define FOO 10 + 5
```

```
int a = FOO * 4;
```

Here, the pre-processor expands the definition of **a** to be **a = 10 + 5 * 4** since it does a straight text replacement. Due to order of operators, the **5 * 4** gets computed before the **+ 10** resulting in the expression **a = 30;** when you almost definitely meant **a = 60;** (4 times 15). Adding the parentheses around the definition of **FOO** fixes this problem from possibly occurring, which is why always adding parentheses around the value of your **#define** statements is always a good idea.

```
#define FOO (10 + 5)

int a = FOO * 4;
```

The sizeof Operator.

Sometimes you want to know how big a data item or type is. **sizeof** returns the size in bytes of its argument which must be either a data item, or a cast. On the Arduino, which uses two-byte ints, these statements hold:

```
sizeof 0 == 2
sizeof (char) == 1
sizeof (int) == 2
sizeof (long) == 4
```

One common idiom for finding the number of items in an array is to take the size of the array and divide by the size of an element in the array. The **#define** in the example below means that the label **N** can be used to represent the length of array **a[]**

```
int a[] = {0, 1, 2, 3, 4 }
#define N (sizeof a/sizeof a[0])
```

Not Much RAM

Arrays are a powerful tool and can easily consume memory. On a typical PC, with anything from 512MB to multiple Gigabytes of RAM, this is not a problem. However in an embedded system you need to be aware of RAM usage. The NICTA ed1 board has just 2kB of RAM, just enough space for 1024 integers. To make it worse your program and whatever libraries you are using need some of this space so you actually have less than 2kB of space to use when declaring variables for your program. A declaration like:

```
long a[512];
```

is simply not possible (you can do the math).

When your program has consumed all the RAM on the board it will cease to function as expected. The LEDs may light and the LCD panel may start flashing. If you have an error where your program appears to function okay for a period of time then weird stuff happens take a look at how much memory space you are using.

Using Constants.

All numbers in C++ are have a type. Constants are also typed: the compiler gives them the smallest type that will hold the value. To force a particular type, you can either use a cast, thus:

```
(unsigned)3;
```

or you can use a suffix. There are two suffices: **U** for *unsigned* and **L** for *long*. You can combine them, so **OUL** is a long unsigned zero, taking four bytes on the Arduino.

There is also a **long long** type, that is guaranteed to be at least 64 bits (8 bytes); it is indicated with suffix **LL**

Short Cut Operators

There are a number of shortcut operators to simplify some common expressions.

Compound assignment operators

Operator	Example	Effect
+=	`a += 2;`	`a = a + 2;`
-=	`a -= b+4;`	`a = a - (b + 4);`
*=	`a *= sin(c);`	`a = a * sin(c);`
/=	`a /= d;`	`a = a / d;`

The value of the expression in each case is the value *after* the operation. Thus, things like: **`c = 2 + (a*=3);`** are legal, and mean multiply **`a`** by three, assign the result to **`a`** add two, and assign the result to **`c`**

In general, if you are using an assignment operator inside an expression, it is best to use parenthesis to ensure the precedence you mean.

Increment and Decrement operators

Incrementing or decrementing a variable is a very common operation. There is shorthand for adding or subtracting one from an integer. The value of an increment or decrement operator is the value of the operand before the operation (for a post- operator) or after the operation (for a pre- operator).

Thus **`++x`** means, 'increment x, then yield its value', whereas **`x++`** means 'the value of x, and increment x after using it'. The table should make things clear.

Operation	Equivalent operation
`y = x++;`	`y = x; x = x + 1;`
`y = ++x;`	`x = x + 1; y = x`
`y = x--;`	`y = x; x = x - 1;`
`y = --x;`	`x = x - 1; y = x;`

Bitwise shifts can also be used as compound operators. Like other compound operations use brackets in complex statements both to ensure precedence for the operation that you want and make your code more readable by others.

Operator	Example	Effect
<<=	`a <<= 4;`	`a = a << 4;`
>>=	`a >>= c;`	`a = a >> c;`

Conditional Assignment

The following sequence is also common in programming.

```
if (expression)
  var = foo;
else
  var = bar;
```

An equivalent, but more concise way, to code the above is:

```
var = (expression) ? foo : bar;
```

where the variable **`var`** is assigned to be **`foo`** if **`(expression)`** evaluates as true and assigned to be **`bar`** if the evaluation is false.

Analogue Inputs

The NICTA ed1 board has a number of pins that Arduino software classifies as either "Digital" pins or "Analogue Input" pins. On the NICTA ed1 board analogue input pins are connected to the light sensor, the potentiometer and the temperature sensor. The pin numbers used are printed on the board next to the part.

In Arduino software the analogue input pins are numbered separately to the digital pins. Analogue input pin 3 **IS NOT** the same as digital pin 3. For analogue inputs you do not need to call the `pinMode()` function to setup the pin.

Reading Analogue Inputs

An analogue input is used when there is a range of possible values to be measured. This is different to Digital IO where a value is either on or off. The analogue inputs measure voltage and on the ed1 this is over the range of 0 to 5 volts. Within the micro-controller reading analogue inputs is performed by a piece of hardware called an analogue to digital converter (ADC).

Any device in the real world has limited precision. For example, an actual voltage may be "1.2345678" volts but the measurement device may only provide 3 digits of precision and provide a reading of "1.23" volts. On the Arduino the precision of the ADC is 10 bits which means that the 0 to 5 volt input is represented by a number between 0 and 1023 with the actual voltage present on the input pin rounded to the nearest 10 bit value. A reading of 1 means an input of around 1 / 1024 of 5 volts (0.00488 volts). A reading of 2 means an input of around 0.00977 volts and so on. The maximum possible reading is 1023/1024 of 5 volts (4.99511 volts).

An analogue input is obtained using the `analogRead()` function. The operation of the ADC and associated hardware takes time and each call to `analogRead()` takes around 100 microseconds.

The code shown below is an example of how to read an analogue input. The input from the light sensor is printed to the serial port every second. You can see how the value changes as you either shine a torch or place your finger over the sensor.

```
#define LIGHT (7)    // analogue input pin 7

void setup(void)
{
  Serial.begin(9600);
}

void loop(void)
{
  int a;

  // read analogue input every second and print to serial port
  a = analogRead(LIGHT);
  Serial.println(a);
  delay(1000);
}
```

Useful Functions `map()` and `constrain()`

Quite often you will want to "re-map" the reading from the analogue input pin to another scale. A linear transformation function is provided by the Arduino software as the `map()` function. The function only works for integers. The function is declared as:

```
long map(input, input_low, input_high, output_low, output_high)
```

where all parameters are of type `long` The input is assumed to lie within the region bounded by **`input_low`** and **`input_high`** and returns a value inside the region **`output_low`** and **`output_high`**

Say, for example, that you wanted to read the potentiometer in 50 steps, you could have a program that looked like this:

```
#include <LiquidCrystal.h>

#define POT (3)

// setup LCD panel
LiquidCrystal lcd(6,7,8,2,3,4,5);

void setup(void)
{
}

void loop(void)
{
  int raw;
  int mapped;

  // read raw 0-1023 from potentiometer
  raw = analogRead(POT);

  // convert to 0-50 value we want
  mapped = map(raw, 0, 1023, 1, 50);

  // display
  lcd.clear();
  lcd.print(raw, DEC);
  lcd.setCursor(0, 1);
  lcd.print(mapped, DEC);

  delay(100);
}
```

You will notice that the steps are evenly spaced with the execution of the last jump from 49 to 50. For most purposes you can ignore that feature. If you are writing software that requires a different spacing you will need to write your own function.

A couple of aspects to be aware of using the **`map()`** function:

1. if the input value is outside the input region bounded by **`input_low`** and **`input_high`** then the output of the function may not be what you expect. The **`constrain()`** function (see the Arduino reference) can be useful to process the input prior to calling **`map()`**
2. the output region bounding variables, **`output_low`** and **`output_high`** do not have to be positive and **`output_high`** can be less than **`output_low`** so you can map high inputs to low values and low inputs to higher values. For example, swap the **`1`** and **`50`** output region parameters in the example above and see what happens.

The Potentiometer on the NICTA ed1

A potentiometer, or variable resistor, is connected on the ed1. A read of the analogue input pin connected to it will return a value between 0 and 1023 depending on the position of the knob. The potentiometer the is the black knob near the lower left hand corner of the board and is connect to analogue input pin 3.

The Light Sensor on the NICTA ed1

The light sensor is based on a photodiode and varies it's output voltage dependent on the intensity of light received. The more intense the light the greater the output voltage. The light sensor is near the center of the board towards the bottom. The actual part itself has a small white square drawn around and it looks similar to the LEDs. The sensor is connected to analogue input pin 7.

In practice a read of the light sensor will not return a value of more than 800-900 even in the brightest light.

The Temperature Sensor on the NICTA ed1

The temperature sensor outputs a voltage that is dependent on the temperature of the device itself. At zero degrees Celsius the part has an approximate output voltage of 0.4 volts. For every one degree Celsius rise in temperature the output voltage will rise by approximately 19.5 millivolts. Without calibration the sensed temperature is only accurate to around plus or minus two degrees Celsius. A code fragment containing a formula for converting from voltage back to the sensed temperature is:

```
a = analogRead(6);
voltage = (float)a * (5.0 / 1024.0);
temperature = (voltage - 0.4) / 0.0195;
```

If You Want More

Like last week the following notes are completely optional. You do not need any of the information in this section to complete the Challenge questions.

The "switch" statement.

C has a way of branching to a label based on the value of an expression. The syntax should be obvious from the example:

```
switch (c) {
case 1:
 /* do something */
 break;
case 2:
 /* do something else */
 break;
case 3:
 /* FALL THROUGH */
case 4:
 /*do more stuff, for c = 3 or 4 */
  break;
default:
 /* report an error: c should have been four or less */
}
```

In this case `break` says 'break out of the switch statement'. Without a `break` directive, control *falls through* from one case to the next — as happens if **c** is 3 in the example. `default` matches any value of the controlling expression that is not matched by an explicit label.

Serial Input on the Arduino

The commands `Serial.print` and `Serial.println` output text from the Nicta ed1 board to the serial monitor window on the IDE. There are corresponding commands for serial input where text entered into the input window on the IDE is received (read) by software running on the Arduino board.

Reading from the Serial Port

The microcontroller on the ed1 board has hardware built into it for hosting a serial port. The serial output commands, like `Serial.print()` send characters to this hardware device where they are converted to electrical signals and sent back to the host computer (or other device connected to the other end of the port). Serial is a two way path, characters can be sent from an external device to the ed1 board. The hardware serial port handles the electrical connection and reads in characters that are sent to it. The microcontroller has a buffer (memory space) for holding up to 128 characters that have been received. The Arduino library functions for reading serial input interact with the hardware controlling this buffer space.

The two most common functions are:

Function	Use
`Serial.available()`	Returns the number of characters in the buffer available for reading
`Serial.read()`	Read a single value from the buffer

The simplest way to read input from the serial port is to poll for a character. "Polling" means actively wait for input (halting other program execution) until input is available to read.

You can use the **`getCharacter()`** function shown below to read a single character from the serial input. Calling the function will poll for input and then return one character. Subsequent characters can be obtained by calling the function again. The **`while`** statement is being used to halt the program by executing a loop that does nothing until there is input available to read at the serial port. Once the loop exits a value is read from the serial port and returned to the calling program.

The **`getCharacter()`** function and an example of use are shown below:

```
/*
 * Example of the use of getCharacter() to read serial input
 *
 * Program waits for serial input and echos back to the serial
 * port one character at a time
 */

char getCharacter(void)
{

  /* loop to wait for input */
  while( !Serial.available() )
    ;

  /* read input and return as a char */
  return( (char)Serial.read() );
}

void setup(void)
{
  Serial.begin(9600);
}

void loop(void)
{
  char c;

  c = getCharacter();
  Serial.println(c);
}
```

Copy the code into your IDE and run on the ed1. You can type values into the serial input text box and press the "Send" button to forward to the ed1 board. Hitting the return button on the keyboard is the same as pressing the "Send" button on the IDE.

Because the **`getCharacter()`** function is returning one character at a time you will notice that entering more than one character into the text box before sending will still produce one character per line of output. For example the input:

```
Hello
```

produces the output:

```
H
e
l
l
```

-

You can type, or paste, lines of text into the text input window on the Arduino IDE. No data is sent to the board until the "Send" button in the IDE or the "Return" key on your keyboard is pressed. At that point the characters are sent over the serial port to the ed1 board. The Arduino IDE strips out special characters like "newline" which are normally used to delimit the end of a line of text data.

Character Encoding

An interesting aspect of the example code in the last two sections is that input and output into a running program on the ed1 board was in the form of characters not numbers. But we know that data is essentially stored as numbers so what is going on? The answer is that both the IDE application running on your computer and the program running on the Arduino board share a common table encoding numbers to characters.

There are many possible standard codings for character sets including, for example, ASCII, EBCDIC, and UTF8. In this case we using the ASCII character set which is the default for C. The character type `char` in C is just an eight bit number, and it is this eight bit number that is passed over the serial port. The standard ASCII character set uses seven bits (i.e. half of the available eight bit numbers) to encode characters. A table is shown on the Arduino web site at:

http://www.arduino.cc/en/Reference/ASCIIchart

For example, from the chart linked above, the digit '**0** has ASCII code of 48; lower case '**a** has code of 97; upper case '**A** is 65. ASCII is nice in that characters in general sort the same as their codes, and the characters '0' to '9', 'a' to 'z' and 'A' to 'Z' form three contiguous ranges.

The format of using single quotes is an easy way to enter literal characters into C source code instead of entering the numeric value.

```
// The following statements are identical
char a = 97;
char a = 'a';

// These three initialise identical 9 character arrays
char s[] = "A string"; // NULL character added automatically
char s[] = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0'}; // NULL charac
char s[] = {65, 32, 115, 116, 114, 105, 110, 103, 0};      // NULL charac
```

The character `'\0'` in the example above is a short cut for the special character NULL, some common special characters are:

C	Character	ASCII value in hexadecimal
`'\0'`	NULL	0x0
`'\n'`	Newline, nl	0x0a
`'\t'`	Tab	0x09

Overloading

The example code for the `getCharacter()` function contains another interesting Arduino feature. The behaviour of the `Serial.print()` and `Serial.println()` commands vary according to the data type of the argument passed to it. This behaviour is known as overloading. The defaults are:

Type of Argument	Default Behaviour for the `serial.print()` and `Serial.println()` functions
`int` or `long int`	print as decimal value
`char`	print as a single character
string (array of `char`	print as a string of characters

For example:

```
char ch = 'a';
int in = 97;
```

```
long int lin = 97;
char str[] = "a";

Serial.println(ch);         // Outputs "a"
Serial.println(ch, BYTE);   // Outputs "a"
Serial.println(ch, DEC);    // Outputs "97"

Serial.println(in);         // Outputs "97"
Serial.println(in, BYTE);   // Outputs "a"
Serial.println(in, DEC);    // Outputs "97"

Serial.println(lin);        // Outputs "97"
Serial.println(lin, BYTE);  // Outputs "a"
Serial.println(lin, DEC);   // Outputs "97"

Serial.println(str);        // Outputs "a"
Serial.println(str, BYTE);   // Compilation error, cannot convert string
Serial.println(str, DEC);   // Compilation error, cannot convert string t
```

Embedded Systems 2011
Random Numbers, Custom Characters, Buttons and EEPROM

Generating Random Numbers

Some programs need a source of random numbers. It is tempting to creatively think up your own source of random numbers but you need to be aware that there is a great deal of mathematics behind random number generators and you are almost always better off using a supplied function.

In Arduino there is a function named **`random()`** which can supply random numbers in the range 0 up to **`RANDOM_MAX`** which is **`0x7fffffff`**.

There are two variants to the **`random()`** function:

`random(high)` - return a **`long`** in the range **`0`** <= number < **`high`**
`random(low, high)` - return a **`long`** in the range **`low`** <= number < **`high`**

As with other similar functions in other languages the numbers returned are from a sequence of pseudo-random integers generated from an initial "seed" state. Each call to **`random()`** returns the next number in the sequence. This can be used to your advantage, given the same seed state each sequence of random numbers will be identical which can be useful for debugging or trying to duplicate a previous result.

Try the following program on your board and look at the output in the serial terminal window as you press the reset button:

```
void setup(void)
{
  Serial.begin(57600);

  for(int i= 0; i < 10; i++) {
    Serial.print(random(0x100), HEX);
    Serial.print(" ");
  }

  Serial.println();
}

void loop (void) {}
```

You will see the same set of 10 numbers printed over and over. The numbers within the sequence are random but the sequence itself is repeatable. Of course if you are generating random numbers you may want the randomness to extend to repeated running of the software. Really it would be preferable if the program you ran above was generating a new set of 10 numbers each time the program ran. The way to achieve this is to provide a "seed" to the random number generator and it is here you can unleash your creativity.

The random number generator is seeded by passing an **`unsigned int`** to the function **`randomSeed()`**. If you have not called **`randomSeed()`** then a seed value of **`1`** is assumed. The problem of course is one of chicken and egg, you want a unique sequence of random numbers and to get a one you need to supply a random number but where to find one?

A convenient source of randomness is an unconnected (floating) analogue input and there are two on the NICTA ed1. The digital pins connected to the buttons are actually re-purposed analogue input pins numbered **`0`** (digital pin **`14`**) and **`1`** (digital pin **`15`**). You can seed the random number generator with the input from one of those like so:

```
randomSeed(analogRead(0));
```

Edit the code you used above to generate sequences of 10 random numbers and put the above call to **`randomSeed()`** at the top of the **`setup()`** function. What happens?

This time you should see a different sequence of 10 random numbers being printed out each time the program runs, or do you? Three things may affect the randomness of the seed.

The first is if you press the corresponding button. If you are reading analogue input **`0`** and you press button **`D14`** then the input is not floating anymore and the number returned from **`analogRead()`** will be more or less constant (try it, run the program and press the button).

Second, if you have configured the button for use in your program then you have probably set in place the "pullup" resistor. This connects the pin, via a resistor, to 5V and again any numbers returned from **analogRead()** will be more or less constant.

Third, even if the input is floating then in practice it maybe returning a limited set of the possible 10 bit range output of the ADC. Run the above program multiple times and probably within ten attempts or so you will see the same sequence.

Seeding the random number generator is where you can unleash your creativity. Find other sources of randomness in your particular circumstance and exploit them. On the NICTA ed1 you have a variety of sensors, perhaps your board is in motion, read in values from the accelerometer and use them as a seed. Perhaps the light sensor is returning variable enough results to use as a seed, perhaps the temperature, perhaps a combination of all three?

One suggestion is to make use of both the floating analogue inputs (before you configure the pins for use as buttons) and multiply them together and use the result as the seed. Work something out that suits your environment.

Creating Custom Characters for the LCD

There are functions for creating and using your own characters on the LCD panel.

If you look closely you may see that each character position on the LCD panel is a 8x5 array of pixels (8 tall and 5 wide). The **LiquidCrystal** library lets you define up to eight characters that you can display together on the LCD panel at any one time.

The function **createChar()** takes an 8 byte array as an argument and uses the least significant 5 bits from each byte to define which pixels are on (bit **1**) and off (bit **0**) on each row. The use of the function is:

```
createChar(int n, byte glyph[8])
```

where **n** is the number of the character from **0** to **7** and **glyph** is the array of bytes. The function **write(n)** prints the character number **n** to the LCD screen at the current cursor position.

An example is:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(6,7,8,2,3,4,5);

byte glyph[8] = {
  B10001,
  B01010,
  B01010,
  B00100,
  B00100,
  B01010,
  B01010,
  B10001,
};

void setup() {
  lcd.createChar(0, glyph);
  lcd.begin(16, 2);
  lcd.write(0);
}

void loop() {}
```

You can call **createChar()** as often as you like within a program but you can only have up to eight user defined characters at any one time. A side effect of calling **createChar()** is that the current cursor position is forgotten. You must call a method thats set the cursor position before a call to a **print()** or **write()** method will work. The methods **begin()**, **home()** and **clear()** will set the cursor to the home position or call **setCursor()** to set the position to a specific position.

Digital Input

The NICTA ed1 board has a number of pins that Arduino software classifies as either "Digital" pins or "Analogue Input" pins. On Arduino boards these pins are usually all available for connection to user devices. On the NICTA ed1 board all the pins are connected (hard wired) to devices already on the board.

The pins connected to digital input devices on the NICTA ed1 are listed in the table below:

Digital Pin	Use
14	Pushbutton 1
15	Pushbutton 2
16	IR Receiver
18	Used for the two wire interface
19	Used for the two wire interface

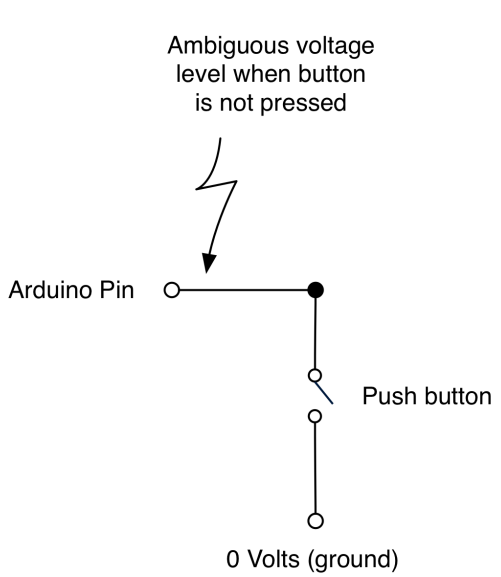
Use the Arduino `pinMode()` function to set a pin to be an input and use the `digitalRead()` to get the current value (LOW or HIGH) of a pin.

Use of a "Pullup Resistor" for a Digital Input

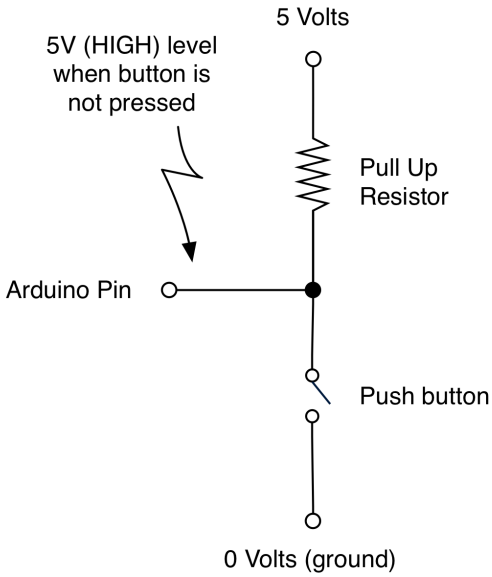
If a digital pin is configured as an input then you may also have to call the `digitalWrite()` function to configure an internal pull-up resistor for that pin. On the NICTA ed1 board this is required for both of the push buttons connected to pins 14 and 15.

Sometimes you need to know a little about electronics. After all, you are writing code that is interacting with electronic devices.

The term "pull up resistor" has to do with the of how the pin is electrically wired. On the NICTA ed1 the digital pins 14 and 15 are connected to buttons. A pullup resistor is configured to make the circuit look like the diagram below. When a button is pressed then an electrical connection is made to 0 volts (or ground) on the board, when a button is not pressed the pin is connected to 5V via the pullup resistor. Without the pullup resistor the input would be "floating" when the button was not pressed and return an ambiguous result.



No Pull Up Resistor



Pull Up Resistor in Place

The resistor is internal to the microcontroller and has a high enough value so that when the button is pressed only a small current flows.

Using Button Inputs in Code

The buttons connected to digital pins 14 and 15 on the NICTA ed1 are wired up such that when a button is pressed a call to `digitalRead()` will return LOW. One of the challenges for your software is to detect the actual button press, a call must be made to `digitalRead()` **while** the button is being pressed otherwise your software will miss it. There is no memory or other circuitry in the microcontroller to remember the button press for you.

You have a choice of waiting indefinitely, polling, for a press of the button like so:

```
while (digitalRead(14) == HIGH)
  ;
```

You also have another other choice of periodically checking on the button state while your software is working on some other task at the same time. The idea is that your code would be looping quickly and checking on the button state frequently enough that a press would not be missed.

Here is an example that of using button D14 to control the LEDs. Load it onto your board and see what it does. This is an example of the second choice, the code continues to run and reads the state of the button often enough to respond quickly.

```
/*
 * Example use of NICTA button D14
 */

#define LATCH (11)
#define DATA (12)
#define CLOCK (10)

#define D14 (14)

void setup(void)
{
  // LED control
  pinMode(LATCH, OUTPUT);
  pinMode(DATA, OUTPUT);
  pinMode(CLOCK, OUTPUT);

  // push button pins and configure pullup resistors
  pinMode(D14, INPUT);
  digitalWrite(D14, HIGH);
}

void ledWrite(byte a)
{
  digitalWrite(LATCH, LOW);
  shiftOut(DATA, CLOCK, LSBFIRST, a);
  digitalWrite(LATCH, HIGH);
}

void loop(void)
{
  byte leds;

  if (digitalRead(D14) == LOW)
    leds = 0xf0;
  else
    leds = 0x0f;

  ledWrite(leds);
}
```

EEPROM on the NICTA ed1

Electrically Erasable Programmable Read-Only Memory (EEPROM) is non-volatile memory, that is, memory that retains information even when turned off. You can store data into and retrieve data from an EEPROM and it does not lose the data when power is removed. Conceptually, EEPROM performs a similar function to a disk drive, but is slower and has far less capacity. However it is much more robust which is usually a real advantage for embedded systems. Non-volatile memory enables one program to write data to the EEPROM and a second program to access the data produced by the first, whether or not the board was powered off in between. It should be pretty obvious that the second program needs to "know" what data has been written, where in EEPROM memory, what type of data (byte, integer, long, float, etc.), and array sizes if arrays are involved. It follows that the more complex the arrangement of data you are planning to save in EEPROM, the more careful you should be in planning and documenting how you will do it.

It is extremely important to remember EEPROM's basic characteristic is that it only provides a limited number of read/write cycles (of the order of 1,000,000) in each area of memory. This means that writing to the same place (EEPROM memory address) in a tight program loop should be avoided at all times.

The part on the NICTA ed1 contains 128kbit of storage arranged as 16k of 8 bit bytes. Hence you can read and write to 16384 (128 * 1024 / 8) different byte locations. The numbers looks simpler expressed in hex, 0x4000 bytes of storage with the first address being **`0x0`** and the last address **`0x3fff`**.

Accessing the EEPROM with the ed1 Library

The available functions are:

`void EEPROM_ed1.begin(void)`	initialise part
`void EEPROM_ed1.sendB(int address, byte val)`	write `byte` val to address
`byte EEPROM_ed1.readB(int address)`	read `byte` from address
`void EEPROM_ed1.sendI(int address, int val)`	write `int` val to address
`int EEPROM_ed1.readI(int address)`	read `int` from address
`void EEPROM_ed1.sendL(int address, long val)`	write `long` val to address
`long EEPROM_ed1.readL(int address)`	read `long` from address
`void EEPROM_ed1.sendF(int address, float val)`	write `float` val to address
`float EEPROM_ed1.readF(int address)`	read `float` from address

The function **`EEPROM_ed1.begin(void)`** must be called at the start of your program to initialise the EEPROM library. No parameters are required. The other functions read or write basic types to and from the EEPROM.

The **`address`** to which the value is written or read from is an **`int`** in the range **`0x0`** to **`0x3fff`**. Recall that each address points to a single **`byte`** of storage space. Writing a **`long`** or other type that takes more than a byte of storage space to address **`0x3fff`** will not work as you might intend as the first byte will be written to address **`0x3fff`** and the subsequent bytes will "wrap around" to be written to addresses **`0x0`**, **`0x1`** and so on up until the size of the type. When writing or reading multiple values make sure you increment, or decrement, the address parameter by the appropriate amount. For example to write two successive values of type **`float`**:

```
float a, b;
int address;

EEPROM_ed1.sendF(address, a);
address =+ 4;   //or better address += sizeof(a);
EEPROM_ed1.sendF(address, b);
```

For another example of using the EEPROM have a look at the example program that came with the ed1 library:

`Files -> Examples -> ed1 -> EEPROM_test`

If You Want More

Like last week the following notes are completely optional. You do not need any of the information in this section to complete the Challenge questions.

Two Wire Interface and Digital Communications

You have used the serial port for communicating with the NICTA ed1 from a computer. A serial port is just one of many different methods by which electronic devices can send information to each other. These methods may be over a relatively long distance, such as various network technologies like Ethernet, or over shorter distances, like SATA used to connect hard drives. At the integrated circuit level there are many standardised interconnect methods and one of these is the "Two Wire Interface" (TWI) or "Inter-Integrated Circuit" (I2C) which is supported by the microcontroller on the NICTA ed1.

The Two Wire serial Interface (TWI) bus was developed to allow simple, robust and cost effective communication between integrated circuits in electronics. The strengths of the TWI bus includes the capability of addressing up to 128 devices on the same bus, arbitration, and the possibility to have multiple masters on the bus. TWI is ideally suited for typical microcontroller applications. The TWI protocol allows for device interconnection using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

The TWI bus is a multi-master bus where one or more devices, capable of taking control of the bus, can be connected. Only Master devices can drive both the SCL and SDA lines while a Slave device is only allowed to issue data on the SDA line. Data transfer is always initiated by a Bus Master device.

The EEPROM and accelerometer are intelligent TWI devices mounted on the NICTA ed1. They are external to the board's microcontroller and can exchange information with it using the TWI protocol. If you are interested here are links to the datasheets:

Accelerometer datasheet

EEPROM datasheet

If you are interested in more information about EEPROM see the EEPROM Wikipedia entry

Arduino Wire Libary Software Support for TWI

The Arduino standard Wire library implements the TWI protocol to allow you to communicate with I2C / TWI devices and hides the low-level complexities of TWI communication. On the Arduino, SDA (data line) is on analog input pin 4, and SCL (clock line) is on analog input pin 5.

However the Wire library does not hide the complexity of individual TWI devices themselves. If you look at the manufacturers data sheets for either of these two devices (above) you will see they contain quite a bit of information on how to access and control these devices. Communication with either device is in the form of sending and receiving byes in a device specific protocol, a little like sending and receiving bytes over the serial port.

You are strongly recommended to use the provided ed1 library to access the EEPROM part on the NICTA ed1 board. If for some reason you choose after the Challenge to use the Wire library then there are some things you should know. First the Wire library supports page reads and writes but fails silently if you attempt to read or write more than 30 bytes at a time. Secondly, the EEPROM part only supports page writes into bounded 64byte pages and if you write over a page boundary then the write will wrap around to the start of the page. Reads however will not. The provided ed1 library does reads and writes by byte which is slightly slower but has no complications with page boundaries.