Microcontrollers Systems and Interfacing

week 1 experimental lab

1 Experimental lab 1

Our goal is to have fun with (and learn a little about) microcontrollers, and the devices that they use to interact with the real world.

1.1 Organisation

You should attempt to complete the lab experiments for a class before that class takes place. You can ask for help with anything you do not understand during the actual class periods, which will be interactive Q&A sessions.

Throughout these experimental lab sessions you will see two indicators in the left margins.

▶ This one (a red triangle) indicates a task you are supposed to complete. It is usually followed by the information you need to complete the task.

☺ 1 Ask the instructor to check your work.

This one (a smiley and a number) means you should have your progress checked by the instructor or the teaching assistant. It is important you do this, since your progress will be used to calculate part of your final grade.

1.2 Equipment

We will use a very common microcontroller, the Atmel ATmega328P. Programming it requires an *integrated development environment*, or 'IDE', and we will use Arduino (because it has many convenient features). Microcontrollers usually have no operating system, and cannot run normal applications, and so we will run the IDE on a laptop computer and upload our programs to the Arduino microcontroller over a USB connection.

1.3 Setting up the development environment

Download and install the Arduino IDE from here: `http://www.arduino.cc/en/Main/Software`

1.4 Running the IDE

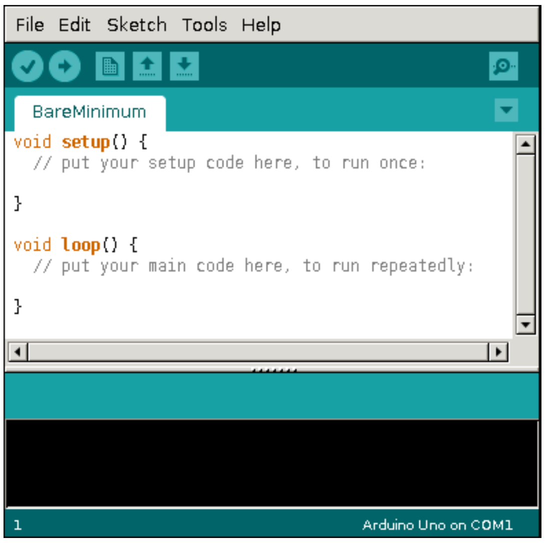
▶ Launch the development environment and configure it.

Make sure the Arduino board is connected to your computer using the USB cable.

Double-click on the '∞' icon (at the top of the screen on the lab laptop), and wait a few seconds for the IDE to launch. A window will open that looks like the one shown on the right.

In the **`Tools > Board`** menu, make sure '**`Arduino Uno`**' is selected.

In the **`Tools > Port`** menu, make sure the correct serial port is selected. (This is probably '**`COM1`**' for Windows, '**`/dev/cu.usbmodemFB0001`**' for MacOS, or '**`/dev/tty/ACM0`**' for Linux.)

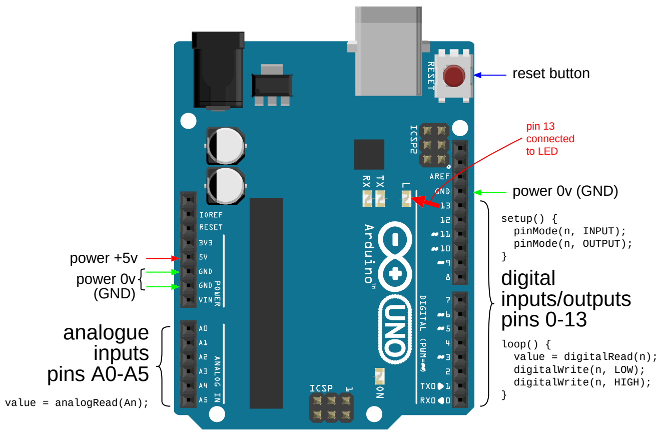


```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

2 Microcontroller hardware

The microconroller has a number of *general-purpose input/output* (GPIO) pins. These pins can be used to communicate digital or analogue information between the microcontroller and the physical world. On the Arduino board there are two 'headers' (rows of electrical contacts into which wires can be inserted). These headers are connected to the GPIO pins of the microcontroller, and to other useful voltages such as 'ground' (GND) and power (5 V).

There are twenty GPIO pins, numbered 0 to 19. The first 14 of them (on the right side of the board in the diagram) are used mainly for digital input/output; they are identified simply by their number (0 to 13). The remaining six pins (on the left side) are used mainly for analogue input; they are usually identified by symbolic name (A0 to A5).



```
setup() {
  pinMode(n, INPUT);
  pinMode(n, OUTPUT);
}
```

```
loop() {
  value = digitalRead(n);
  digitalWrite(n, LOW);
  digitalWrite(n, HIGH);
}
```

```
value = analogRead(An);
```

The digital pins must be *configured* before they are used. In the microcontroller program, digital pins are configured using the `pinMode()` function (see below) as either input or output. After configuration, input pins can be read using `digitalRead()` and output pins can be written using `digitalWrite()`.

Pin 13 is conveniently connected to a small *light-emitting diode* (LED). If pin 13 is configured as an output then whenever it is set to '1' (or 'high') the LED will be illuminated.

Embedded programming can be more difficult than desktop (or even mobile) application programming, in part because embedded programs are difficult to debug. In some situations, the LED connected to pin 13 is the only source of debugging 'feedback' that is available to you. (In a few situations, even this LED will not be available.)

3 Programming the microcontroller

Microcontroller programs are called 'sketches' in the Arduino system. (The language is a dialect of C++, which is used a lot for programming embedded systems.) Our first sketch is as simple as possible, just to test the connection. It will make a small light-emitting diode (LED) blink on the microcontroller board.

Every sketch has two main parts: the *setup* part and the *loop* part.

The `setup()` function is called *once* to configure the microcontroller, the sketch first starts running, If you think of the microcontroller as emulating some hardware, the `setup()` function describes what the hardware *is*.

The `loop()` function is called *repeatedly* to perform *one complete cycle* of useful work, for as long as the microcontroller is powered-up and running. If you think of the microcontroller as emulating some hardware, the `loop()` function describes what the hardware *does*.

```
void setup() {
  // configure the hardware here,
  // using pinMode(), etc.
}

void loop() {
  // perform one complete cycle
  // of useful work here, using
  // digitalWrite(), etc.
}
```

Both of these functions (`setup()` and `loop()`) are provided by the programmer. (In other words, *you* have to write them.) They are the Arduino equivalent of the `main()` function of a C program. Since they return no result, they should both be declared '`void`'.

3.1 Configuring and using digital pins

The 14 digital input/output pins are numbered 0 to 13. Each digital pin that you want to use must be *configured* in the `setup()` function either as an INPUT or as an OUTPUT. Configuration is done using the `pinMode()` function, which takes two arguments: the *number* of the pin being configured and the *direction* of the pin (INPUT or OUTPUT).

```
pinMode(n, OUTPUT); // can now use digitalWrite() to set the value of pin n
```

In the `loop()` function, digital output pins control external devices by generating a voltage which can be controlled using `digitalWrite()`.

```
digitalWrite(n, LOW);  // set pin n to 0 (0 volts)
digitalWrite(n, HIGH); // set pin n to 1 (5 volts)
```

Setting an output pin to LOW makes it represent logical 'false', or turns *off* the device (such as an LED) connected to it. Setting the output pin to HIGH makes it represent 'true', or turns *on* the device (such as an LED) connected to it.

4 Making the LED blink

Your first 'challenge' is to...

► Make the LED connected to pin 13 blink once per second.

Most of what you need for this is explained in the previous pages. For convenience, here is a summary of the relevant information:

- Your sketch needs two functions
  - `setup()` configures the microcontroller, and
  - `loop()` performs one complete cycle of work (blinking the LED once during one second)
- In `setup()` use `pinMode()` to configure pin 13 as an OUTPUT.
- In `loop()` use `digitalWrite()` to turn the output (and hence the LED) on and off
  - setting pin 13 to HIGH turns it 'on', and
  - setting pin 13 to LOW turns it 'off'

You will also have to ensure that the `loop()` function takes exactly one second to complete. The easiest way to do this is using the `delay()` function:

```
delay(n);  // pause execution for n milliseconds
```

☺ 2 Ask the instructor to check your work.

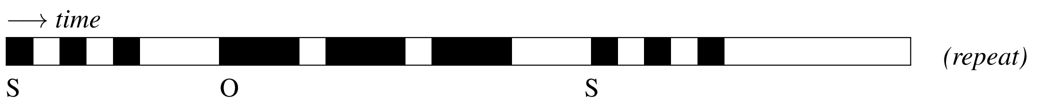
Ask the instuctor (or Teaching Assistant) to check your work.

4.1 Additional challenges

If you finish early, try these additional challenges. (The final two require some additional programming knowledge.)

4.1.1 Make an 'emergency beacon'

Modify your program so that the LED sends a message using Morse Code. For example, the international emergency signal is the letters 'S', 'O' and 'S' sent using Morse Code. (The message stands for: 'Save Our Souls'.) The pattern is three short flashes, then a pause, then three long flashes, then a pause, then three short flashes again, followed by a longer pause before repeating. In the following diagram, black represents the LED being illuminated:



To make your Morse Code precise, you could use the following timing:

- a dot (short flash) should be one unit of time
- a dash (long flash) should be three units of time
- the space between dots and dashes within a letter should be one unit of time (same as a dot)
- the space between letters should be three units of time (same as a dash)
- the space between words (at the end of the 'SOS' message) should be seven units of time

4.1.2 Generalise your program (hard)

Your program probably uses `digitalWrite()` and `delay()`s to send S-O-S. Modify it so that the signals to be sent are encoded in a string and sent by a function called `sendMorse()`. For example, to send S-O-S your program would use:

```
sendMorse("... --- ..."); // sends the SOS signal in Morse code
```

4.1.3 Make your program's interface more user friendly (harder)

Instead of passing dots and dashes to `sendMorse()`, pass a string of normal characters and make `sendMorse()` encode them into dots and dashes automatically:

```
sendMorse("SOS"); // sends the SOS signal in Morse code
```

(The table on the following page might be helpful.)

Microcontrollers Systems and Interfacing

week 1 reference material

A Morse code

Letters

A	· –	N	– ·
B	– · · ·	O	– – –
C	– · – ·	P	· – – ·
D	– · ·	Q	– – · –
E	·	R	· – ·
F	· · – ·	S	· · ·
G	– – ·	T	–
H	· · · ·	U	· · –
I	· ·	V	· · · –
J	· – – –	W	· – –
K	– · –	X	– · · –
L	· – · ·	Y	– · – –
M	– –	Z	– – · ·

Numbers

1	· – – – –	6	– · · · ·
2	· · – – –	7	– – · · ·
3	· · · – –	8	– – – · ·
4	· · · · –	9	– – – – ·
5	· · · · ·	0	– – – – –

Punctuation

Comma	,	– – · · – –
Full stop	.	· – · – · –
Question mark	?	· · – – · ·
Semicolon	;	– · – · – ·
Colon or division	: ÷	– – – · · ·
Slash	/	– · · – ·
Dash	–	– · · · · –
Apostrophe	’	· – – – – ·
Inverted commas	”	· – · · – ·
Left parenthesis	(	– · – – ·
Right parenthesis	)	– · – – · –
Double hyphen	=	– · · · –
Cross	+	· – · – ·
Multiplication sign	×	– · · –
Commercial at	@	· – – · – ·