Microcontrollers Systems and Interfacing

week 2 experimental lab

1 Communication using the serial monitor

One advantage of using a *tethered* environment (where the microcontroller is connected to a real computer during development) is that we can interact with the microcontroller using the USB connection.

1.1 Serial monitor 'hello world'

The first program most people write on any new platform just prints 'hello world'. This is very easy to do while the microcontroller is tethered to a host computer.

► Send 'hello world' from the microcontroller back to the host computer.

In your `setup()` function, enable serial communication using `Serial.begin(9600)`. (`Serial` is the name of the serial communications device connected to the host computer. The argument `9600` sets the communication speed, in bits per second, between the microcontoller and the host computer.)

In your `loop()` function, print the value of a `counter` variable on the `Serial` device. To make this interesting, increment `counter` each time the `loop()` function is executed. (Declare the `counter` variable to be `static` so that its value is preserved between successive calls to the `loop()` function.) Avoid overloading the communication channel by pausing for 250 ms at the end of `loop()`.

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

void loop(void) {
  static int counter = 0;
  Serial.println(counter);
  ++counter;
  delay(250);
}
```

Don't forget: to see the output, you have to open the serial monitor from the 'Tools' menu in the IDE!

► Increase the communication speed.

In some situations, 9600 bits per second is too slow. Increase the communication speed to 19,200 bits per second, or even to 115,200 bits per second. (Note: you will have to modify this number in *two* places for communication to be successful. If you have any trouble at all, ask for help!)

► Modify your program to print the counter in binary.

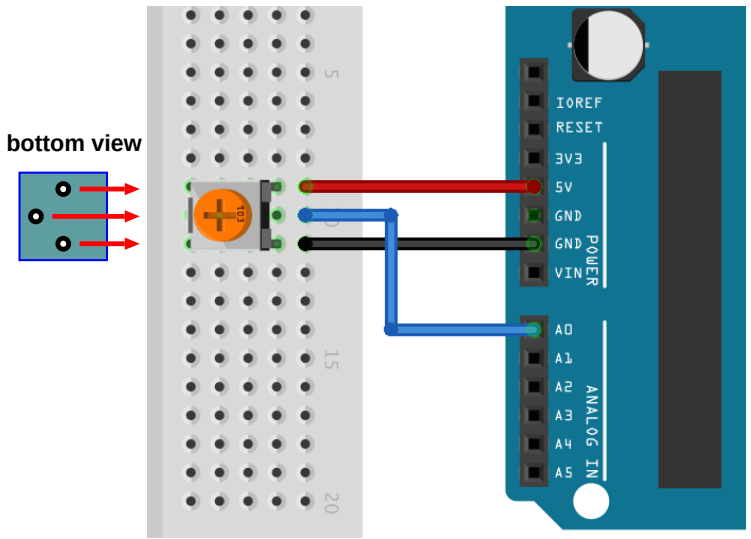
See the reference material in the appendix for information about how to do this.

2 Analogue input

Instead of printing a counter, let's read a voltage from an external device and report its level back to the host computer.

► Create a variable voltage using a potentiometer.

Disconnect your microcontroller from the host computer. Then use a breadboard to connect a potentiometer as shown. (The two outer pins are connected to 0 V and 5 V. The volate center pin will vary between these two values as the control knob is turned.) Do not reconnect the microcontroller yet.



IMPORTANT: The potentiometer **must** be connected exactly as shown. Incorrect orientation could damage it permanently.

Each of the three pins on the potentiometer must be connected to a different row of the breadboard. The centre pin of the potentiometer must be connected to the analogue input of the microcontroller. The outer two pins of the potentiometer must be connected to the 5 V and GND connections of the microcontroller.

The potemtiometer pins are fragile. Never force it into the breadboard or you will break the pins. Place the potentiomenter on the breadboard and then gently rock it back and forth to encourage the pins into the holes.

▶ Read the value of the voltage and report it to the host computer.

Connect the middle pin of the potentiometer to an analogue input on the microcontroller. (You do not need to configure the analogue input pins in `setup()`. The analogue pins are configured as inputs by default.)

Use `analogRead()` to read a number representing the voltage on the analogue input pin. (Note that the analogue input pins are called `A0` through `A5`.) The number you receive will vary between 0 (when the input pin is at 0 V) and 1023 (when the input pin is at 5 V). Instead of printing the counter to the `Serial` device, print the value you read from the analogue pin.

```
void loop(void) {
  int value = analogRead(A0);
  Serial.println(value);
  delay(250);
}
```

3 Use the input to control an output

Let's use the analogue input voltage that we are reading to control the frequency of blinking of the LED connected to pin 13. For this you will need to combine your sketches from last week and this week.

▶ Make the LED blink each time you read the analogue voltage.

Copy the `setup()` code from your LED blinking sketch into your analogue voltage reading sketch. Then copy the `loop()` code from your LED blinking sketch into your analogue voltage reading sketch. Set the blink period to a couple of hundred milliseconds. Verify that the LED blinks each time a voltage is reported on the serial monitor.

▶ Make the LED blink at a rate proportional to the input voltage.

You have delays for the LED 'on' and 'off' periods (currently using fixed values). You also have a source of variable numbers between 0 and 1023 controlled by the potentiometer. Combine the two so that your delay times are variable, controlled by the potentiometer.

Note that for the smoothest operation you should disable any printing code in your `loop()` function.

4 Challenges

If you want to go further, try some of the following challenges...

4.1 Convert the analogue reading into some other unit

The analogue reading varies between 0 and 1023. These values represent a voltage range of 0 V to 5 V. Write a sketch that prints the input *voltage* instead of the raw analogue value. Hint: use a `float` variable (instead of an `int`) to hold the value.

The potentiometer physically turns through 270 degrees. Write a sketch that prints the *angle* of the potentiometer instead of the raw analogue value.

4.2 Monitor two analogue inputs at the same time

You have a second potentiometer in your experimental kit that you can connect to `A1`. Modify your serial monitor program to print both input voltage values on the same line, separated by a space. Hint: to print a space, use: '`Serial.print(" ");`'. When this works in the serial monitor, see what happens if you open the serial plotter instead.

4.3 Use two analogue inputs to control a graphical application

If you have completed the previous challenge, and you are comfortable with installing software and writing programs on your host laptop computer, try using the Arduino analogue inputs to control a program running on your laptop. Begin by downloading and installing Processing from here: `https://processing.org/download/`

Open Processing and click on the little blue box top-right of the main window that says "Java". Select "Add mode…". A window will open. Click on "Python Mode for Processing 3" and then on the "Install" button below. Wait a few minutes while the software is installed. When a green tick appears next to "Python for Processing" you can close the pop-up window. Click again on the little blue box top-right of the main window that says "Java" and change the mode to "Python".

Visit the course web site at: `https://kuas.org/~piumarta/id2` and find `sketch_etch.pyde` in the week 2 materials section. Open the link, copy the program, and paste it into the Processing main window. (Make sure the indentation looks *exactly* the same as in the file on the web site, or *nothing* will work.) Modify the definition of `port` on the third line to match the serial port that your Arduino board is connected to. Run the program and have fun with the two potentiometers.

Try changing the `delay()` value in the Arduino sketch to higher or lower numbers. What effect does it have on your ability to control the graphical program?

[DIFFICULT] Modify `sketch_etch.pyde` to print connected *lines* instead of single points.

▶ Make some interesting art using the program, take a screenshot, and share it with the rest of the class in the Teams channel.

Microcontrollers Systems and Interfacing

week 2 reference material

A Serial monitor functions

The following methods can all be used with the `Serial` object to control two-way communication with the host computer.

`begin(`*`speed`*`, `*`config`*`)` initialises the serial line to operate at the given *speed* (*baud* rate, or *bits per second*). The *config* parameter sets the number of data bits, parity type, and the number of stop bits. The default *config* is `SERIAL_8N1` (1 start bit, 8 data bits, no parity, one stop bit). Digital pins 0 and 1 are used for serial communication, and become unavailable for input/output after calling this function.

`end()` disables serial communication. Digital pins 0 and 1 are available for input/output after calling this function.

`available()` returns the number of bytes available to be read.

`read()` reads and returns the next byte of seral data. Returns -1 if no data is available.

`peek()` is similar to `read()` except the data is not removed from the input buffer.

`flush()` delays until all outgoing data has been written.

`print(`*`value`*`, `*`format`*`)` prints the *value* (a string or a number) on the serial lines. If *value* is an integer then the optional *format* argument tells the function what base to use: `BIN`, `OCT`, `DEC`, or `HEX`. If *value* is a floating point number then the optional *format* is an integer that controls how many digits are printed after the decimal point. Returns the number of bytes written to the serial line.

`println(`*`value`*`, `*`format`*`)` is the same as `print()` except that a newline character is also printed.

`write()` writes one or ore bytes to the serial line. When called with one integer argument, the corresponding byte is written to the serial line. When called with a single string argument, the bytes of the string are written. When called with two arguments, the first must be a pointer to an array of bytes to be written and the second specifies the length of the array.

Be careful not to send data faster than the configured speed can transmit. Remember that One character is ≈10 bits (1 start bit, 8 data bits, 0 parity bits, 1 stop bit) and use that to calculate how much information you can send every second:

- 9600 baud (the default) is approximately 960 characters per second;
- 115200 baud (the fastest) is approximately 11520 characters per second.

A small terminal screen (80 columns × 24 lines) contains 1920 characters. The serial buffer (holding data waiting to be sent) is only 64 bytes long. When it is full, your program will *block* (stop running) until space becomes available. Use `delay()`, and/or `Serial.flush()`, and/or a counter to limit how often you send data, to ensure that the buffer *never* becomes full.