Program Flow

A program is essentially a list of instructions for a computer to follow. Unlike a person, the computer will follow the instructions exactly and in the order specified.

When it gets to the last instruction, it will stop — and there's really no way to tell it to go on from there, only to reset it manually, and have it start back at the beginning.

That's typically not what one expects of a modern system - to have to power cycle it to run through a set of instructions. Instead we generally expect a system to do something, then wait for some signal to either do the same thing again, or to do something else. This implies that the computer is waiting for something. But computers don't wait. They execute one instruction after another until there are no more, then they stop.

Fortunately, one of the instructions they can follow is called a branch. Normally the computer would execute the next instruction in line after the current one is complete. The branch instruction tells the computer to jump to a different part of the list and start executing instructions from there. Using that, we can now have the computer 'loop' though a set of instructions, returning back to the top of the list instead of ending.

Working at a higher level, the C++ language used in our Arduino IDE still mimics the basic operation of the underlying hardware, with programs beginning at a certain point and ending on the last instruction.

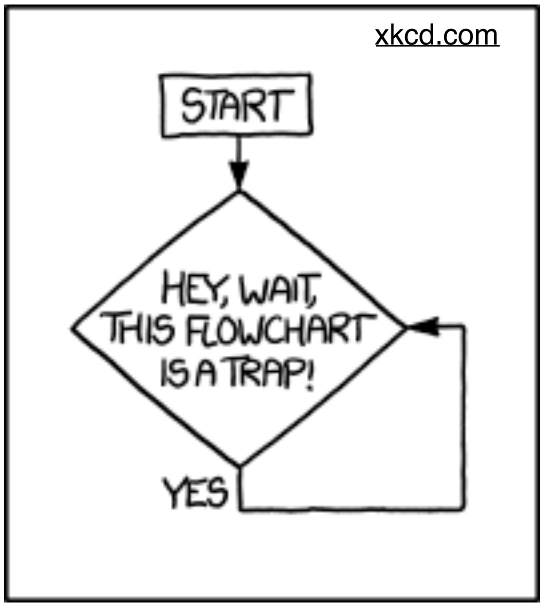
The Arduino IDE tries to simplify programming for microcontrollers by providing two default sections for instructions. Those placed inside the `setup()` area will be executed first and only once, while those placed in the `loop()`area will be executed next and when the last one is reached, execution will start over at the first instruction in the `loop()` section. This allows you to program some initial setup and configuration and then have a built-in loop for the rest of your work.

Some terminology and syntax needs to be addressed in this experiment, we'll just cover the basics now and get into some of the details in the next few experiments.

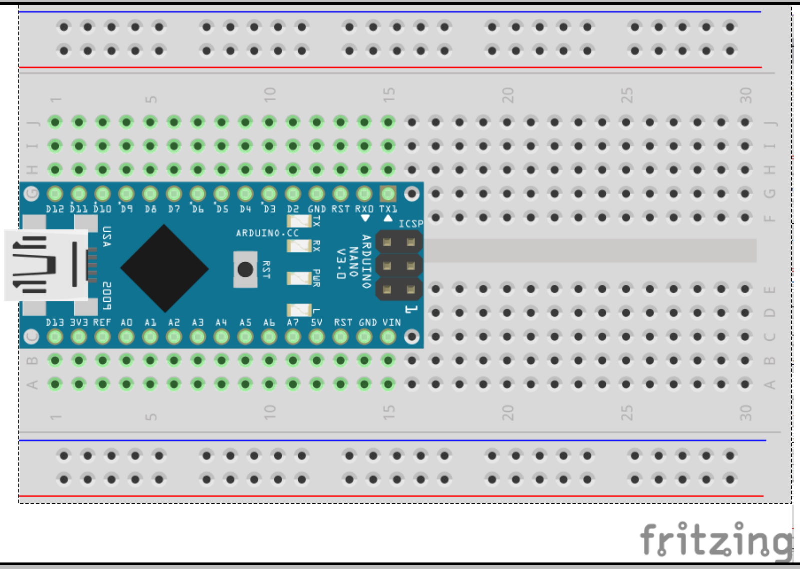
The instructions that are to be performed in the `setup()` and `loop()` areas are identified by being enclosed in curly brackets. So after setup() there is an open curly bracket (`{`). After that you can put your various instructions. When you are done, a closing curly bracket, or brace, (`}`) lets the compiler know you are done defining what needs to be done in `setup()`. The same syntax is used in `loop()`.

Although the computer can only continually execute instruction after instruction until it runs out — or forever if it's told to branch back to the start. There are times where you as a programmer need the computer to just wait for a bit. The Arduino environment has you covered with several special instructions to do just that. In the program for this experiment you will see that something called `delay()` is used. It does just what it says, it delays, or waits for a while before letting the program flow continue on to the next instruction. How long? That's up to you. Whatever number you put inside the parenthesis determines how long it will wait. The scale is in milliseconds — there are 1000 milliseconds in a second — so `delay(500)` will wait 1/2 of a second. Of course, the processor must be executing some instructions, so delay() is actually just shorthand for many individual instructions that result in a delay of the specified amount of time.

You may have noticed the semicolon hanging on the end of all the example assignments and definitions. In our environment, the end of a single instruction must be terminated with a semicolon. This lets the compiler easily separate commands from one another as it tries to convert this higher-level language of instructions directly into the binary machine language that the particular microprocessor we are using will actually execute. It is easy to overlook or forget to add these when you are beginning programming, so whenever your code does not build, check your semicolons first!



Experiment: Quick, it’s time to delay().

Components	**Wiring Diagram**
✓Computer ✓Arduino IDE	

Connection Instructions
No connections required, we will be using the internal LED for this example.

Sketch(es)	holdUp.ino

Analysis Questions
Does your code seem a bit redundant once you have completed the task? How did you determine what the appropriate delay times would be?

Programming Tasks
After uploading the initial program and observing its function, modify the program to blink out a distress call, SOS, in morse code. Morse code consists of long ‘dashes’ and short ‘dots’. In this case it will be long and short blinks of an LED. In Morse code, and S is represented by 3 short dashes, and an O by 3 long dashes, although the origin of SOS as a distress signal may not be tied to the letters for any particular reason. (see wikipedia) Still … - - - … with a short delay in-between sends is still generally recognized as the ‘SOS’ distress call. You should be able to do this by cutting and pasting sections of the existing code.