Remote

Solar Powered

(with battery backup)

Soil Moisture Sensor

Theory of Operation

# Description

This device is a soil moisture sensor that determines a relative number between 0 and 100 indicating completely dry to entirely wet. The number is sent to a controller for further analysis and/or action. It is battery powered and the battery is recharged using a small solar panel. The device is required to be calibrated and the calibration routine is part of the software.

The components are: a soil moisture sensor, an Arduino Nano with an nRF24 radio, an 18650 Li-Ion battery, a charge controller board and a solar panel. The housing is made up of 3D printed parts.

This device is part of a system that also includes a gateway and a controller.

# [The Arduino](https://www.arduino.cc/)

This project uses the [Arduino Nano](https://store.arduino.cc/products/arduino-nano) with an [nRF24 radio](https://www.aliexpress.us/item/3256806748199858.html) (this could be an [RF Nano](https://www.aliexpress.us/item/3256807353403141.html)) which is programmed with the [Arduino IDE](https://www.arduino.cc/en/software/). Communication requires the [Arduino MySensors library](https://docs.arduino.cc/libraries/mysensors/) to be installed into the Arduino IDE.

Parenthetical numbers are the code line number.

# The Soil Sensor (542)

The soil moisture sensor used in this project is a simple capacitive measurement. Reading the sensor is one line of code:

analogRead(0);

This could be adequate for simple purposes, but this *read* doesn’t tell the whole story. *analogRead* returns a number between 0 and 1023. But the number returned from the sensor is a smaller range than that. If the sensor is entirely immersed in water a number near 300 is returned and if completely dry, a number near 900. But these numbers are dependent on several factors. Manufacturing tolerances are the biggest culprit, but not just of the sensor itself, but also of the computational device used (eg. Arduino).

Suffice to say, in order to get a reasonable, predictable number that relates to how moist the soil is, the combination of the sensor and the Arduino requires calibration. To wit: what is the number when absolutely dry and when fully wet? When known, assume a linear relationship.

# Calibration (384)

So, before the device is deployed, each sensor/Arduino must be calibrated. This is a matter of notifying the program when the sensor is immersed in water and when it is completely dry.

A separate program could be loaded for each condition. The program would be run with the assumption that the sensor is completely wet, read the sensor and store the value in non-volatile memory (EEPROM, Electrically Erasable Programmable Read Only Memory).

There is a small challenge in that the memory is only *byte addressable* (only one byte, 8-bits, can be read or written at a time) and the value is 10 bits. So the upper 8 bits are extracted (even though only two are used) and stored in one location and the lower 8 bits are stored in an adjacent location.

A better way is to send a message to the Arduino that indicates that the Arduino is to store the current moisture value in the EEPROM associated with the condition. [Communication](#vlq0uyrnk1oa) gets ahead of ourselves and will be explained later.

If it were only that easy. Because this is a [battery](#tj51o8gakwkc) powered device, the battery voltage fluctuates which affects the moisture reading. Thus, the condition is not only wet or dry, but high or low battery voltage. Before sending the message, the voltage level that powers the Arduino must be set. This means that there are four calibration constants, Wet/Battery Minimum (**CnW**), Wet/Battery Maximum (**CxW**), Dry/Battery Minimum (**CnD**), Dry/Battery Maximum (**CxD**).

Great! We have calibration constants. How does this get a more repeatable number that represents soil moisture? It is assumed that the number read from the sensor is on a linear scale between the dry reading and the wet reading. This is a pretty good approximation. This is where the [map() function](https://docs.arduino.cc/language-reference/en/functions/math/map/) is employed. (It is assumed that the reader will go to the hyperlink to understand the map() function.) Using the map function the dry and wet numbers for the current voltage are determined. These two numbers are used as the range for determining the percentage moist from the value read from the sensor. (If it takes you some time to figure this out, you’re in good company.)

One note. At low battery voltages, the relationship is not linear. If the battery is consistently in this range, it needs to be replaced. Compensating for this nonlinearity would complicate this measurement immensely for little improvement in accuracy. After all, we’re not sending a probe to the moon, we’re only going out into the back yard.

# Communication

The [MySensors](https://www.mysensors.org/) environment is used to communicate with a controller, which, in this case, is [Home Assistant](https://www.home-assistant.io/).

## [MySensors](https://www.mysensors.org/)

MySensors is adequately documented and the reader is directed to that website. It is no longer being developed and has almost non-existent support.

Devices will communicate through a [gateway](https://www.mysensors.org/build/select_gateway) to a controller, such as Home Assistant.

There are four code sections that control how the device communicates.

### Compiler Variables (285)

These will set up the paths and variables which the program will use to utilize the MySensors environment. This is how the various sensors (CHILD) of the device are distinguished.

### Before() (479)

Code executed at the beginning of the MySensors environment initialization.

### Presentation() (516)

Sends the description of the program and information about each sensor

### Receive() (635)

This is an quasi interrupt service routine (ISR) that receives data from the controller (Home Assistant). An ISR should have a bare minimum of code. Variables used in an ISR must be of type *volatile*, meaning that their value will change at any time within the body of the program. Boolean flags (of type *volatile*) are used to indicate that data is received so that they can be acknowledged at a time convenient to the body of the program.

Sensors are created to receive data from Home Assistant. They must be defined as type *V\_TEXT* and be presented as type *S\_INFO*.

The nodeID can be set before loading the operational code onto the Arduino. The program, **addressAsUint8\_t\_1\_00.ino**, is used for this purpose. If not set, it will be arbitrarily set by the gateway, which can lead to multiple devices having the same nodeID.

In order to better use data in Home Assistant, the variable names (created in presentation) should have the nodeID as part of the name. (see *varName*)

## [Home Assistant](https://www.home-assistant.io/)

Home Assistant is a control program that performs high level operations with input from and output to multiple, disparate sources and destinations. It is quirky, obtuse, arcane and is supported by independent, non-affiliated programmers.

MySensors is integrated into Home Assistant through the MySensors *Integration*. Devices are almost automatically configured. That is, the MySensors device must send an initial value for all sensors to be used.

Also, and importantly, sent data can be overrun. Therefore, after sending data from the Arduino device, there must be a wait time which should be determined through trial-and-error. (For example, a serial gateway, 2S for presentation and 200mS for data; for an MQTT gateway 4S and 500ms.)

An *Input\_number* “helper” is used to send the calibration messages and sampling period from the controller to the Arduino.

# The Battery (555)

It turns out that reading the battery voltage is a [not-so-secret encantation](https://forum.arduino.cc/t/arduino-secret-voltmeter-explanation/447711). Actually, it is reading the voltage of 5V, which is the power input used in this project.