Copyright © 20 June 2025 by Peter G. Raeth

Scalable Point-to-Point LoRa P2P Sensor Network Via Flood-Messaging

Peter G. Raeth
Independent Researcher and Educator

Proof of Concept / Disclaimer

This document and its associated source code describe a proof of concept only. It has not been verified in an industrial environment. While the author has applied good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaims all responsibility for errors or omissions, including, without limitation, responsibility for damages from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code or other technology this work contains or describes is subject to open source licenses or intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Abstract

This project is a proof-of-concept that builds a peer-to-peer network of LoRa transceivers to support remote sensing and data transport. It uses flood-messaging to eliminate the need for more-costly LoRaWAN and third-party services. We begin with basics and gradually build the network. After having a working network, we add options to demonstrate the network's capabilities. Each section of this document takes a specific step forward and specifies the required hardware, associated documentation, and reliable sources of purchase. Project material is posted at the GitHub Repository. This present document is meant to be active in the sense that links are provided where necessary for additional details. While this pdf form of the document can be read without an internet connection, getting full value requires one. Let's dig in and peel back the onion.

Copyright © 20 June 2025 by Peter G. Raeth

Table of Contents

Copyright © 20 June 2025 by Peter G. Raeth

Copyright © 20 June 2025 by Peter G. Raeth

Table of Figures

Copyright © 20 June 2025 by Peter G. Raeth

Project Contributors

A paper by these three professors inspired the approach used by this project:

- Professor Philip Branch, School of Science, Computing and Engineering Technologies, Swinburne University of Technology, Melbourne, Australia
- Professor Binghao Li, Faculty of Engineering, University of New South Wales, Sydney, Australia
- Professor Kai Zhao, Faculty of Engineering, University of New South Wales, Sydney, Australia

Their well-written paper discussed the design, implementation, and testing of their LoRa flood-messaging network. The citation and link to their paper is: Branch, P., Li, B., Zhao, K. (2020) A LoRa-Based Linear Sensor Network for Location Data in Underground Mining. MDPI, Telecom, 1(2), 68-79, https://www.mdpi.com/2673-4001/1/2/6.

Introduction

Flood-Messaging, in its simplest form, is like Ethernet's UDP networking in that delivery of messages is highly likely but not absolutely guaranteed. Also, there is no effort to maintain message order. A source broadcasts messages. Those messages are received and rebroadcast by relay nodes. Destination (Basestation) nodes make use of message contents. Over time, messages age out of the system. Figure 1 Illustrates the basic idea and the three types of network nodes involved.

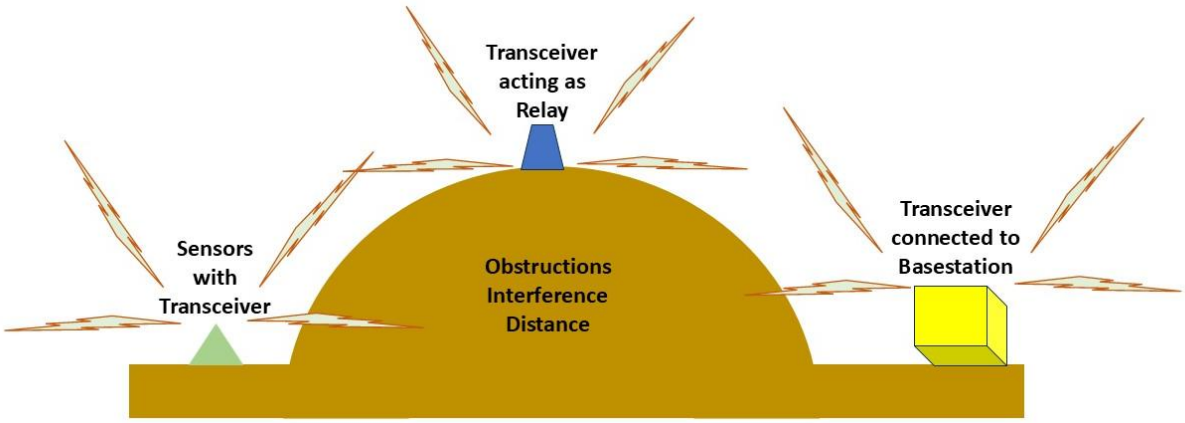


Figure 1. Fundamental idea behind LoRaP2P flood-messaging.

To dig deeper into the concepts and practical implementation of LoRaP2P flood-messaging, start by reading the paper by Branch, Li, and Zhao. Then read about its evolution in the paper by Raeth and Branch. In this present document, we get down into the weeds of realization via software and hardware. To begin, we put together modules that provide basic functionality. Then we build an inexpensive grassroots network. We continue by adding options that illustrate what can be done

Copyright © 20 June 2025 by Peter G. Raeth

with the network. The Appendix contains a spreadsheet that cites each piece of hardware needed and reliable sources. Part-specific documentation is on each product page or linked from there.

Lets begin by laying the groundwork.

Laying the Groundwork

We start by examining the learning background one needs to proceed beyond plug-and-play. Then we put components together to build a simple functional transceiver that broadcasts sensor data. Be sure to read the entire section of interest before proceeding with a given step. Nothing beats understanding before doing. You will gain insights that help you as you proceed.

What You Need to Know

New projects require us to learn more than we may already know. They also lead to hardware purchases as well as writing and installing software. We talk now about those topics.

Education

Although we start with simple steps and little hardware, and hope for plug-and-play, understanding the guts of the software and modifying it is not for beginners. Where appropriate, reference is made to courses at Saylor Academy. Course completion earns an online certificate.

> *Saylor Academy is a nonprofit initiative working since 2008 to offer free and open online courses to all who want to learn. We offer nearly 100 full-length courses at the college and professional levels, each built by subject matter experts. All courses are available to complete — at your pace, on your schedule, and free of cost.*

One needs background in the following:

- Python programming language (CS105: Introduction to Python)
- C++ programming language (CS107: C++ Programming)
- C++ as implemented via Arduino IDE (Arduino Language Reference)
- LoRa itself (Introduction to LoRa Technology, LoRa and LoRaWAN Introduction)
- Arduino hardware (general link, see the appendix for specific devices)

Copyright © 20 June 2025 by Peter G. Raeth

Integrated Development Environments

There are many IDEs one may use. These two are the ones I have found to be most useful. Both tools are well-documented for installation and employment.

- PyCharm Community is an excellent open-access tool for working with Python. (See the bottom of the linked page.) Versions exist for Windows and Linux.

- Arduino IDE is an excellent open-access tool for working with Arduino C++ and associated microcontroller boards. Versions exist for Windows and Linux. (We use the latest 1.x.)

Hardware

Suitable hardware for this project has not been obvious. It is important to look beyond advertising and hype to find reliable hardware and suppliers.

The present thrust relies on the Arduino product line and their MKR WAN 1310 microcontroller board with its embedded LoRa transceiver. Many other options were rejected for various reasons:

* requires rats-nest of jumper wires to implement even simple applications
* poor or disorganized documentation
* lack of versatility
* limited to the infrastructure and components of one company
* unresponsive tech support and inactive user community
* dysfunctional website or out-of-stock on most products
* unable to get basic examples to function correctly
* limited to LoRaWAN, unable to work with LoRaP2P
* shipping increases overall cost by more than 50%
* products are fragile or otherwise of low quality

All things considered, I decided to go with Arduino's product line. Their distribution is global and they have many distributors. They have warehouses in many countries such that devices ordered directly from Arduino ship from the nearest warehouse. Their robust products are relatively expensive, when compared to other options. But they work well, manufacturing is consistent, and they survive many scenarios of physical and electrical application. Documentation is extensive and their user community is active. There are any number of less-expensive clones but my personal experience with clones has not been good. Using original Arduino devices, it is possible to build a grassroots network for around $US165.

It is not necessary to purchase all the hardware at once. It can be purchased as this document proceeds since we build a grassroots LoRaP2P network piece-by-piece and then add options. An appendix contains a table of all hardware components used in this project, prices (at time of publication), and reliable purchase sources. Item-specific documentation is either on the product page or linked from there.

Copyright © 20 June 2025 by Peter G. Raeth

An Elementary Microcontroller

Arduino's product line is highly modular. It begins with a baseboard and then adds plugin modules. Wire-In points are provided if third-party components are desired. There are several options for baseboards. These vary in price and capability. For this project, we work with the MKR WAN 1310 since it has an embedded LoRa transceiver and so offers the best chance to build a useful network and connect to various sensors.

Assembling Hardware

We start with one MKR WAN 1310, its antenna, and its USB cable. From the devices' documentation, you will see that the MKR WAN 1310 is somewhat small in physical size. Some people will need a magnifying light so they can use both hands while working with small components. It all depends on your eyesight.

Read the product documentation carefully. It explains the MKR WAN 1310 in detail. For this first task, we use Pins 5v, GND, and A1. We also use LED_BUILTIN. These are illustrated in Figure 2. The asterixed "pins" accommodate male jumper wires. (Jumper wires come in male/male, male/female, and female/female, depending on your needs.)

Copyright © 20 June 2025 by Peter G. Raeth

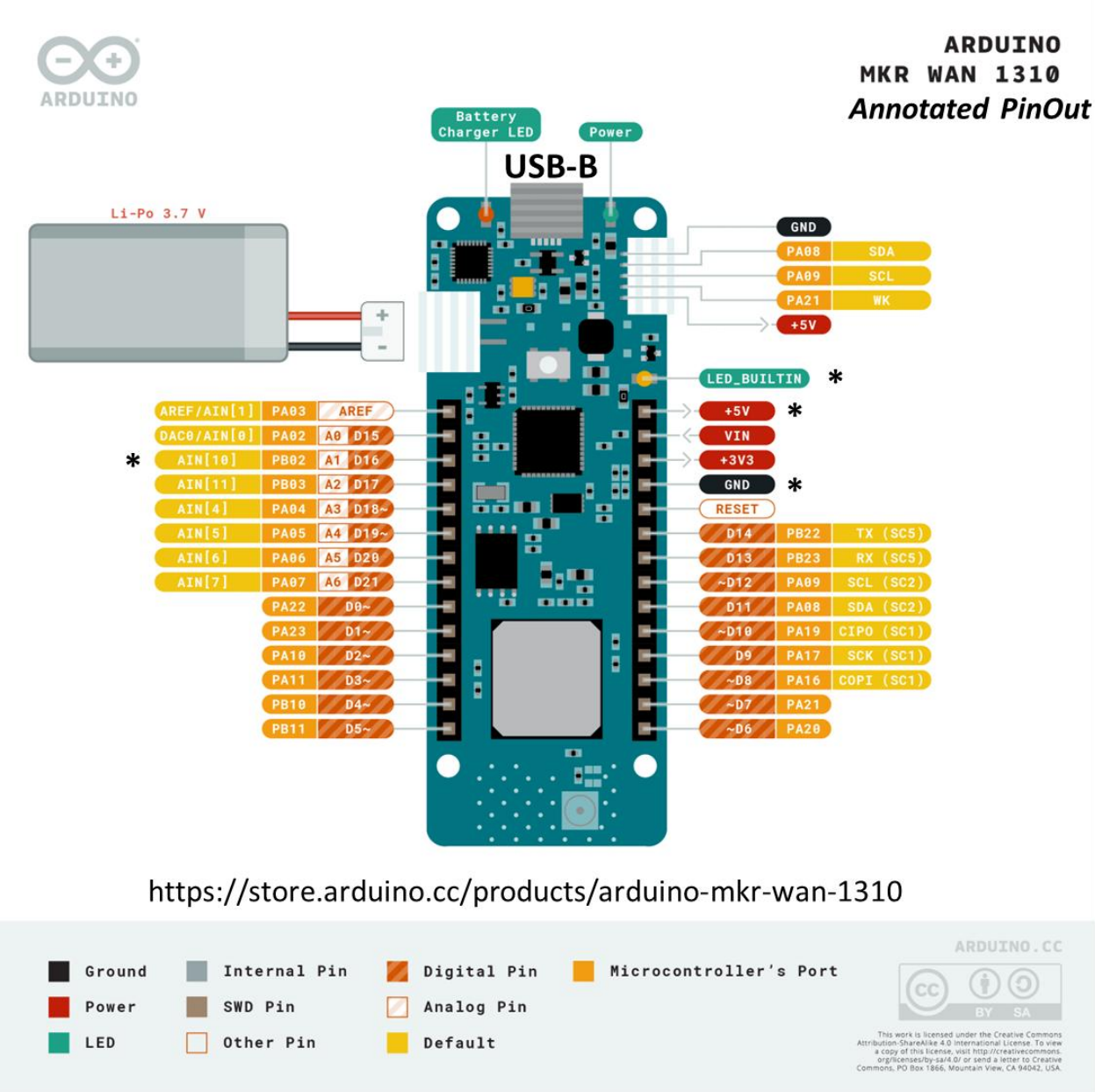


Figure 2. Annotated MKR WAN 1310 pinout diagram.

Installing Software

Once the antenna has been mounted, connect a USB-A/USB-B cable to the device. This cable has to provide power and serial communication. Then plug the other end of the cable into a USB-A port on your computer. You will notice the activated charging and power LEDs near the USB-B port.

Follow these directions to install and configure the Arduino IDE. Notice that we are installing the latest v1.x, not the latest version, 2.x. The reason is that v2.x is still in beta.

Install the latest version of “Arduino SAMD Boards”. Within the IDE, use Tools/Board/Boards Manager. Search on “MKR”. Select the library and press Install. Once the library is installed, ensure

Copyright © 20 June 2025 by Peter G. Raeth

you have the correct port selected, Tools/Port/Arduino (MKR WAN 1310). Once you reach that point, try a blank sketch and press the “Verify” button. Compilation should succeed if everything has been done correctly.

Now load a relatively simple example, one that just blinks the built-in LED, File/Examples/Basics/Blink. After pressing Verify, the build should complete successfully.

Sketch uses 11964 bytes (4%) of program storage space. Maximum is 262144 bytes.

Global variables use 2988 bytes (9%) of dynamic memory, leaving 29780 bytes for local variables. Maximum is 32768 bytes.

Now press the Load button. The program will recompile and install itself in the device. Once that process completes, you will notice LED-BUILTIN is blinking. This step seems trivial but it is important to verify IDE installation and cable connection. Some USB cables provide only power and do not facilitate data communication. Others provide only data communication. The one specified in the appendix’ parts list provides both. We want to be sure everything is working as expected.

As we end this section, we have demonstrated that the device is installed and configured correctly. We have programmed a basic operation and so are ready for something more.

Adding a Simple Sensor

Sensors do nothing more than provide voltage that has to be interpreted according to the sensor type. For our simple sensor, we supply zero voltage and verify that we are reading it correctly. Later, we will add various sensor types. For each, we will show how to read and interpret their voltage. According to the board’s documentation, no more than 3.3 volts should be connected to the board, lest the board be ruined.

To read voltages, the device uses an integrated Analog-to-Digital Converter (ADC). ADC’s do not give a direct voltage reading. Rather, they give an integer value according to the specified resolution. See this article by Arrow Electronics for more detail.

For this example, we put no voltage at all into the device. To that end, a jumper wire is used to connect the A1 pin to the GND (ground) pin. After connecting the device to a USB port, we can load the SimpleSensor code. On the serial monitor, you will see:

13:50:37.454 -> ==================================
13:50:37.454 -> Arduino MKR 1310 read-voltage test
13:50:37.454 -> ==================================
13:50:37.454 -> 0.0
13:50:38.461 -> 0.0
13:50:39.471 -> 0.0
13:50:40.481 -> 0.0
13:50:41.444 -> 0.0
13:50:42.449 -> 0.0
13:50:43.455 -> 0.0
13:50:44.464 -> 0.0
. . .

Copyright © 20 June 2025 by Peter G. Raeth

That is exactly what we would expect to see. If you want to experiment with the idea, turn off the Serial Monitor, comment all println in setup, and turn on the Serial Plotter. Let the program run for a bit. Then unplug the GND side of the jumper wire. Follow up by plugging the jumper wire back into GND. You will see a plot similar to Figure 3.

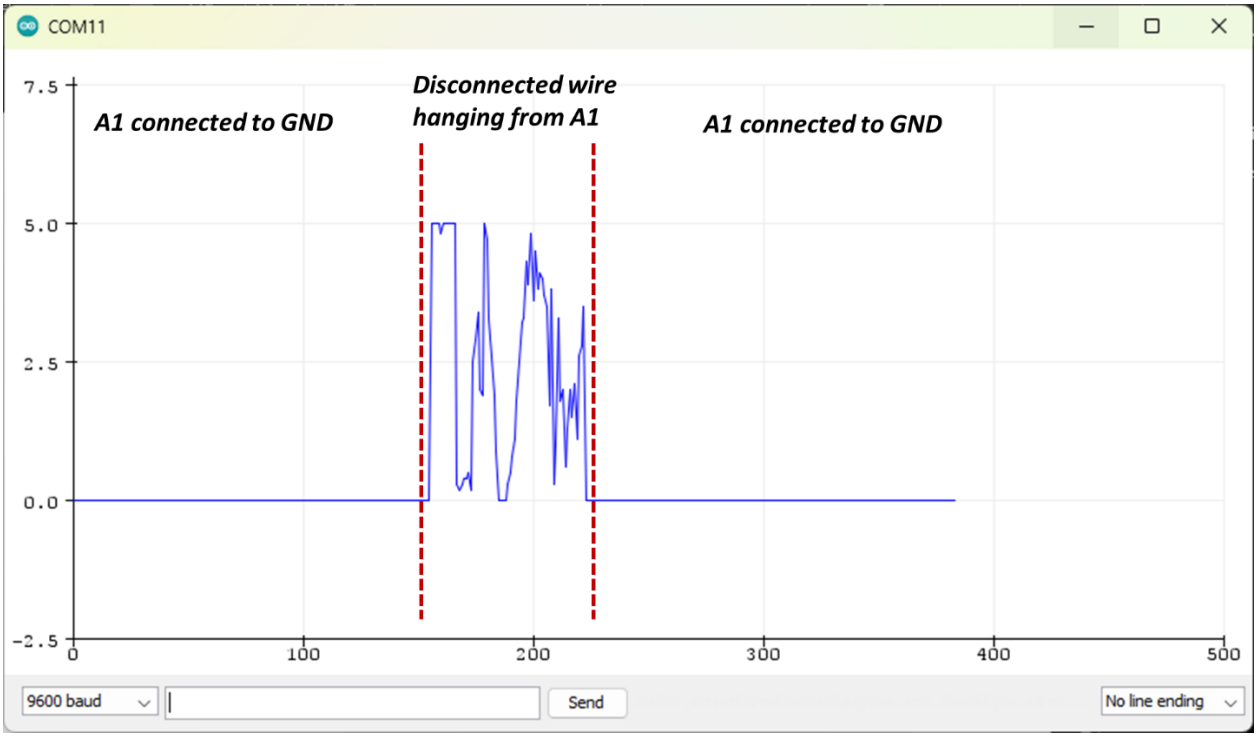


Figure 3. Plot of voltage reading at A1 pin (annotated).

A First Cut at LoRaP2P

We are now able to sense voltage inputs. Let's learn how to send and receive that data. We begin by composing and sending simple LoRa messages that contain readings from sensors connected to the device. A second node receives, parses, and displays the contents of those messages. You need two MKR WAN 1310, their antennas, and their associated USB cables.

Which LoRa library to use is an important consideration. This project uses a modified version of the baseline LoRa library maintained by Logsnath Logu and Sandeep Mistry. That library is still in development but is a good foundation.

Why does this project supply a modified version of the baseline LoRa library? Attempts to use its CAD callback did not yield a reliable process as the project grew. The project reached a point where the callback was never called. (The baseline has no alternative to the CAD callback.) The project works very well, over long periods of time, without the CAD callback. So there is unlikely to be anything in the code causing memory corruption. There is a recommendation for an alternative to the baseline CAD function.

Copyright © 20 June 2025 by Peter G. Raeth

Another reason for modification is that the baseline sometimes causes two "receptions" to be made of the same packet. There was another recommendation on how to solve that. It worked well to solve the double-reception problem this project experienced. Both recommendations were applied to create the modified baseline LoRa library.

Also at issue is that the alternative library, MKRWAN, recommended in Arduino's documentation seems to require a gateway and to not work in pure P2P mode. No update has been made to the baseline library code in the past two years, as of the time this document was written. But its basic capability has worked well for this project, when the two modifications are added.

After the present documentation was written, the author became aware of the RadioLib library. This may well be something to consider in the long term. See Centelles' superb dissertation for an example application. This library follows a different philosophy than the present library. Its examples do compile for the MKR WAN 1310.

Sender

This first cut is based on an article by Karl Soderby. Merged into Soderby's code is our program for reading sensor voltage. Connect the GND and A1 pins. Load SenderNode_Basic. You will see the following on the Serial Monitor:

10:41:50.161 -> ==========
10:41:50.161 -> Arduino MKR 1310 basic LoRa sender test
10:41:50.161 -> ==========
10:41:50.161 ->
10:41:50.161 -> Sending packet: 1
10:41:50.208 -> Packet 1 : Value 0.0
10:41:51.219 ->
10:41:51.219 -> Sending packet: 2
10:41:51.267 -> Packet 2 : Value 0.0
10:41:52.275 ->
10:41:52.275 -> Sending packet: 3
10:41:52.323 -> Packet 3 : Value 0.0
. . .

Receiver

If we take a second MKR WAN 1310 and load ReceiverNode_Basic, we see that the LoRa packets are being transmitted accurately.

10:50:19.146 -> ==========
10:50:19.146 -> Arduino MKR 1310 basic LoRa receiver test
10:50:19.146 -> ==========
10:50:19.379 -> Received packet 'Packet 481 : Value 0.0' with RSSI -26
10:50:20.428 -> Received packet 'Packet 482 : Value 0.0' with RSSI -26
10:50:21.495 -> Received packet 'Packet 483 : Value 0.0' with RSSI -26
10:50:22.562 -> Received packet 'Packet 484 : Value 0.0' with RSSI -26
. . .

Copyright © 20 June 2025 by Peter G. Raeth

Ensuring Proper Use

Now that we have a basic sender/receiver example, we should now consider proper use. While this author is not able to speak to exact legal matters, all nations have their requirements for proper use of LoRa radio frequencies. If you are communicating with some third-party service, they may have deeper requirements. For example: Which radio frequency to use is set by each nation. This table shows frequency by nation. Notice that some nations do not allow unlicensed radio broadcasts.

Messaging duty cycle is also set by each nation (limitations on how often messages can be sent). For instance, the USA has no duty-cycle limitations but transmissions cannot last longer than 400 milliseconds. Europe has a duty cycle of 1%. Considerable technical detail is given in this document by Semtech. A good parameter calculator is posted by Arjan.

For my own case in the USA, from the calculator and assuming a 13-byte overhead:

> Frequency: 915mhz; Spreading Factor: 7; Bandwidth: 125khz

Other factors shown in the linked calculator's table are in regard to The Things Network.

We also have to be concerned about Channel Activity Detection (CAD) since we do not want a network node to try broadcasting at the same time as another node. This topic is discussed in Semtech's Introduction to Channel Activity Detection. Their application note speaks to different CAD uses. This author has been unable to confirm that the LoRa baseliine library we are using has a fully-functional CAD capability. However, something we can do is check for an existing signal, per Fujiura Toyonori's modification. His CAD method was integrated into the baseline library.

Configuring Country-Oriented Settings

Adding country-oriented settings to sender nodes is not difficult. These are indicated for the author's case in SenderNode_USA. Use the tools mentioned above to discover the settings for your country and change the configuration accordingly. The serial monitor will show the same result except that setup() displays the configuration settings. ReceiverNode_Basic is sufficient to receive and display the messages broadcast by SenderNode_USA.

Building a Grassroots Network

Having established send/receive, we now build a grassroots network. Recall from Figure 1 that there are three types of nodes in a flood-messaging network: sensor, relay, basestation. Although any node could be made to send and receive as the ability of the network and its nodes grows, for now we continue with sensor nodes that only broadcast their data on some schedule. Relays only receive and rebroadcast. Basestations only receive. Figure 4 offers a view of our grassroots network and its components.

Copyright © 20 June 2025 by Peter G. Raeth

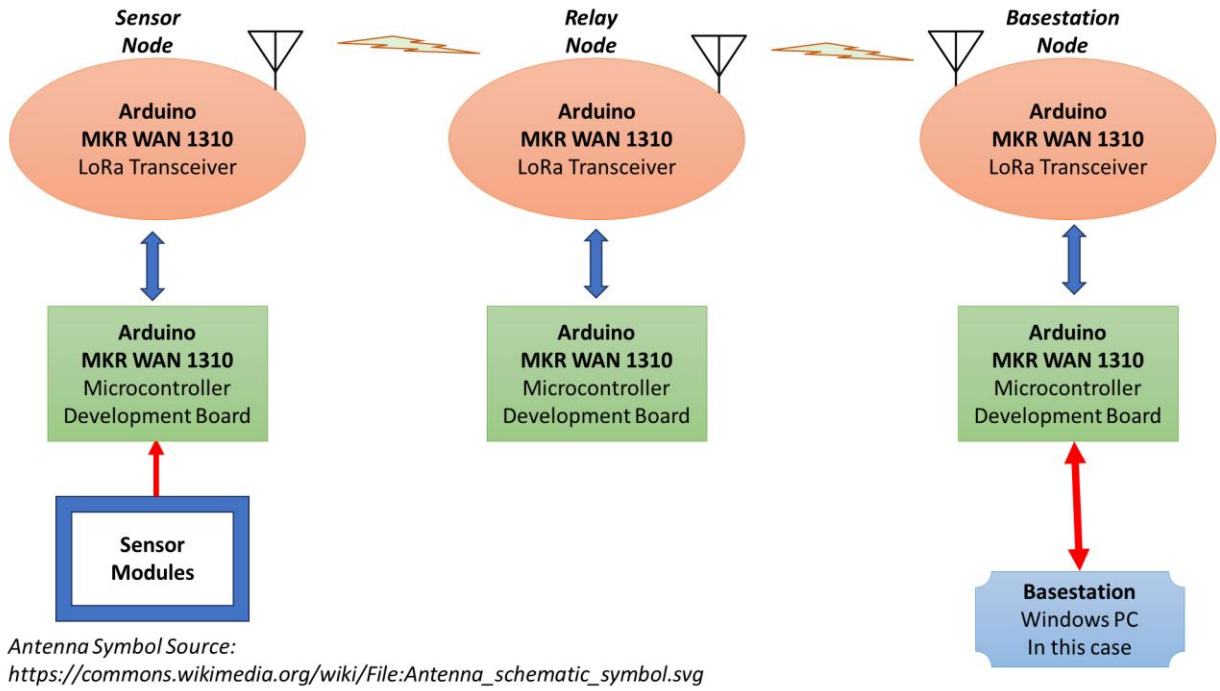


Figure 4. Component relationships in grassroots network.

Recall that we are talking about radio broadcast. There is no reason that any node could not receive a message sent any other node. For instance, the basestation could directly receive a message from a sensor node, as well as that same message rebroadcast by a relay node. It is necessary to account for that possibility without limiting the network to only one basestation.

Forming Messages for Broadcasting

From a network perspective, it is necessary for messages to contain certain basic information. We apply the following header:

Byte	Meaning
0	Message Length
1	System ID
2	Source Node ID
3	Destination Node ID
4	Message ID - High Byte
5	Message ID - Low Byte
6	Message Type
7	Sensor ID
8	Number of Rebroadcasts Remaining
9..max	Message Contents

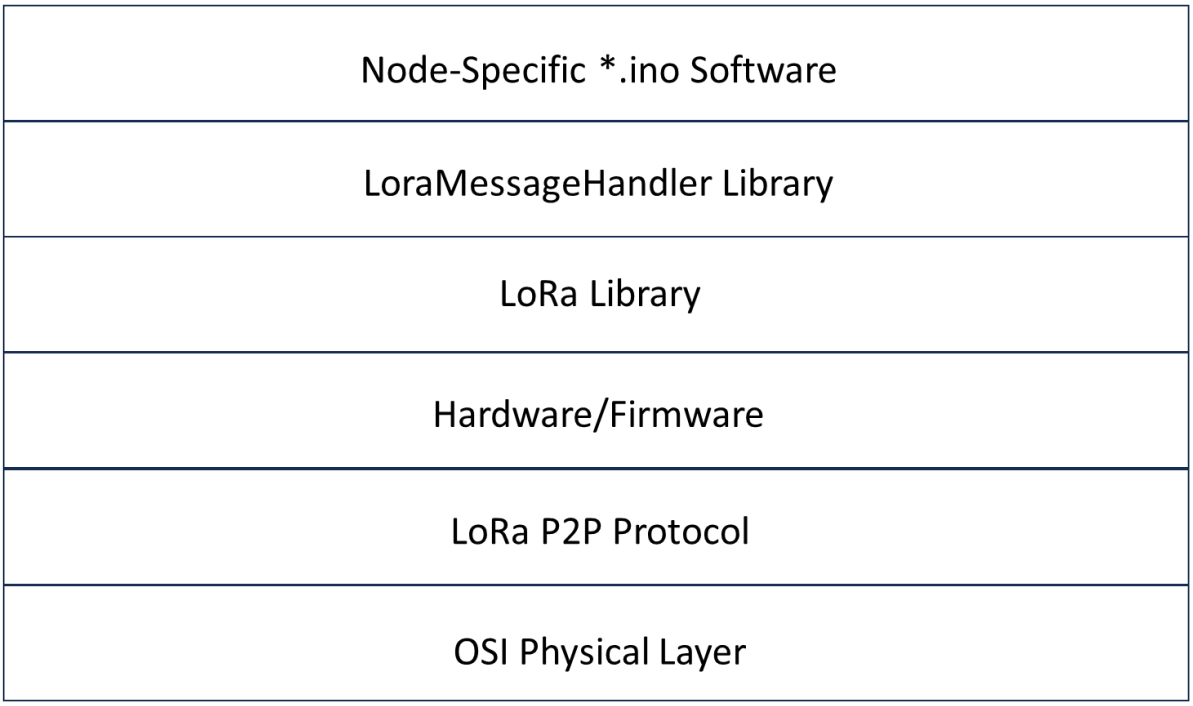
Notice that we are dealing with bytes, although one can still decide to send messages containing only ascii text. Bytes are unsigned eight-bit integer values in the range 0..255. Text allows for values in the range 0..127. As options are added to the network, we will see why we really do need to think in terms of bytes instead of just text.

Copyright © 20 June 2025 by Peter G. Raeth

We employ a standard byte-level messaging library that is evolving with this project, and which accesses the modified LoRa library. This approach maintains commonality between nodes, except for what appears in the ino file specific to each node type. Take the code in the LoRaMessageHandler directory and put that into your Arduino libraries directory.

LoRaMessageHandler.cpp contains a DEBUG variable at the very top. If you want to view the library's operational updates on the serial monitor, uncomment that variable. Each node's ino file has a comparable DEBUG variable so that it is not necessary to see LoRaMessageHandler,s text as the network operates. During standard operation, both variables should be commented. This is especially true for basestation nodes since those communicate via the serial port with a PC.

LoRaMessageHandler's constructor sets LoRa parameters. As distributed, those settings are for this project in the USA. Use the links given earlier to determine appropriate values for your application and your nation. Ultimately, the project could evolve this approach into a parameter-entry file. The relationship between software components is summarized in .



Copyright © 20 June 2025 by Peter G. Raeth

Sensor Node

Recall what we did when we built a simple sender/receiver. We expand now on that idea by using the supplied standard messaging library. After loading Grassroots_SensorNode, on the serial monitor you should see something similar to:

11:49:20.323 -> Microprocessor is active
11:49:20.790 -> ==
11:49:20.790 -> Arduino MKR 1310 Grassroots LoRa flood-messaging sensor node test
11:49:20.790 -> ==
11:49:20.790 ->
11:49:21.261 -> Trying to send this message: Volts: 0.0
11:49:21.308 -> Sent this message: Volts: 0.0 (Text Length: 12)
11:49:21.308 ->
11:49:24.052 -> Trying to send this message: Volts: 0.0
11:49:24.099 -> Sent this message: Volts: 0.0 (Text Length: 12)
11:49:24.099 ->
. . .

DEBUG is uncommented in the ino file but commented in LoRaMessageHandler.cpp. You can see more detail by uncommenting DEBUG in that file. Notice how our earlier work is carried forward. Messages use the same LoRa access methods applied earlier.

As you form text messages, be sure to follow a format that ensures accurate parsing of Sensor Type, Units, and Value. Multiple sensor values can be put that way into the same message. In this case, we are using text. But that is not essential. One could always just send a number as bytes and then specify the sensor type. Then, a table could be used to decode the value sent. There are a lot of messaging options as the need arises.

Relay Node

Relays use another MKR WAN 1310. They rebroadcast messages as they are received. After a given number of rebroadcasts, a message ages out and is no longer rebroadcast.

Load the code for Grassroots_RelayNode. On the receiver's Serial Monitor you should see something similar to:

11:45:22.341 -> Microprocessor is active
11:45:22.820 -> ==
11:45:22.820 -> Arduino MKR 1310 Grassroots LoRa flood-messaging relay node test
11:45:22.820 -> ==
11:45:22.820 ->
11:45:27.162 -> Rebroadcasting... Success
11:45:27.260 ->
11:45:29.979 -> Rebroadcasting... Success
11:45:30.027 ->
11:45:30.460 -> Rebroadcasting... Success
11:45:30.508 ->
. . .

Copyright © 20 June 2025 by Peter G. Raeth

Basestation Node

There are two components to the basestation, an MKR WAN 1310 and a personal computer running a python program (Linux or Windows, Mac may work too). The MKR component plugs into a USB serial port on the PC. The PC talks to the microprocessor, processes the messages, and displays results.

MKR Component

The MKR associated with the basestation receives messages and passes them to the PC via the USB serial port. It ignores messages not meant for the particular basestation to which it is attached.

Load the code for Grassroots_BasestationNode_MKR. Uncomment DEBUG. You should see something like this on the serial monitor:

12:40:44.332 -> Node is active
12:40:44.809 -> ==
12:40:44.809 -> Arduino MKR 1310 Grassroots LoRa flood-messaging basetation node test
12:40:44.809 -> ==
12:40:44.809 ->
12:40:48.911 -> Forwarding: Volts: 0.0
12:40:48.911 ->
12:40:48.959 -> Old message: 1208 1208
12:40:48.959 ->
12:40:50.400 -> Forwarding: Volts: 0.0
12:40:50.400 ->
12:40:50.449 -> Old message: 1209 1209
12:40:50.449 ->
12:40:52.420 -> Forwarding: Volts: 0.0
12:40:52.420 ->
. . .

"*Volts: 0.0"* is the message the PC component of the basestation would receive. Notice how duplicate messages are rejected as "Old".

Personal Computer Component

Your knowledge of Python will come in handy when engaging the PC component of the basestation. That code is under Grassroots_BasestationNode_PC. For this grassroots demonstration, the personal computer simply repeats received messages. Later, we can add graphical displays as desired. We can also interact with external systems as devices are controlled or data is communicated.

DEBUG on the MAK component has to be commented because the PC becomes the "serial monitor". Be sure to turn the IDE's serial monitor off. It is important to comment DEBUG in the LoRaMessageHandler library and the ino file.

Copyright © 20 June 2025 by Peter G. Raeth

You can run the Python program from the console or the Python IDE. Something similar to the following appears on the PC during this example:

Serial ports:
Detected 4 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM16	*COM16*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM16)*
COM11	*COM11*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM11)*
COM3	*COM3*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM3)*

Connecting to LoRaP2P microcontroller/transceiver (COM16)
Connected. Awaiting Messages...

1750352600.042058: Volts: 0.0
1750352600.042058: Volts: 0.0
1750352600.710033: Volts: 0.0
1750352603.5677621: Volts: 0.0
1750352605.265617: Volts: 0.0
. . .

As described earlier, messages can be formed, interpreted, and acted upon according to the needs of the application.

Some Thoughts

We see clearly now the basics of Arduino's MKR WAN 1310, how to query sensors, and how to send and receive messages that contain sensor data. Messages should be composed so that they are easily parsed, so that their essential contents can be extracted.

We may be curious about options that make a network more useful. Let's look into some possibilities in the next sections.

Data Security

Here we briefly address the issue of data security. We take the opportunity to implement an AES encryption/decryption process.

Many people want to have secure communication. Although that can never be perfect, we have the Advanced Encryption Standard (AES) published by the US National Institute of Standards and Technology (NIST). AES is the same symmetric block cipher the US government uses to protect classified information. You can read more about AES in this posting by TechTarget: https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard.

There are many implementations of AES posted on the internet. While I do not argue about their correctness, there was only one posting I could find that demonstrated how to use the underlying AES implementation to encrypt and decrypt variable-length messages. This library was written by Sergey Bel. We began with his AES repository but had to make a small modification. Put that library

Copyright © 20 June 2025 by Peter G. Raeth

into your Arduino libraries directory. We use this library to provide an example of how to implement AES encryption/decryption for LoRaP2P message broadcasts. There are many options for use that are discussed in the repository.

Cryptology and Cyber-Security are vast fields into which this project does not delve. You may find this open-access cryptology book useful, https://cryptobook.nakov.com. You can read a discussion about the issues at hand in the well-written master's thesis by Priyansha Gupta. Finally, AES is key based. There are encryption/decryption techniques that do not require pre-established keys.

Demonstrating Message Encryption/Decryption

After reworking the example in the selected AES library, the example's code, AES_Baseline, shows the basic approach. It simply creates a message, encrypts it, then decrypts, and displays the result.

To illustrate encrypt/decrypt within LoRa P2P, SenderNode_Encrypt and ReceiverNode_Decrypt were created based on what we saw in the basic send/receive demonstration. The sensor node encrypts and broadcasts messages. The receiver decrypts and displays the messages.

Encryption/decryption has not yet been applied within the grassroots network. Notice that it is necessary that message lengths be divisible by 16. Thus it may be necessary extend the length of messages beyond their original intent. That is normal to AES but has to be accounted for in the message to be transmitted by sensor nodes and interpreted by relay and basestation nodes. Note also that decryption takes time. If messages are sent too fast then some will be missed.

Possible Extension

The employed AES implementation uses a static key but the key need not be static as long as both sender and receiver use the same key. Varying keys over time may enhance security. A message could be used to trigger a change in key without sending the key itself. As this project is not a study in communication security, the author has not attempted to go beyond what is demonstrated.

Error Detection

It is sometimes the case that data becomes corrupted when moving from one point in a network to another. One way to catch such corruption is via Cyclic Redundancy Checks (CRC). CRC is an error-detection process that catches data changes during transmission or storage. It helps ensure data integrity by adding a checksum to the data and verifying it at the receiving end. More information is offered by TanzoLab. Further detail is accessible via the National Institute of Standards and Technology.

Demonstration

We used TanzoLab's process to generate a CRC subroutine. Add the CRC library to your Arduino libraries folder. You can see it applied within the library's CRC_Baseline and CRC_Testing. These illustrate the underlying process. That process was integrated into our basic send/receive codes within ErrorDetection. Load SenderNode_CRC and ReceiverNode_CRC into different MKR WAN 1310. As they run, on the serial monitor you will see messages passing and CRC being calculated

Copyright © 20 June 2025 by Peter G. Raeth

and verified. For the purposes of this project, the chip's CRC capability was activated. This capability rejects corrupted messages without feedback. This present exposition is one way of checking message integrity via CRC and providing feedback.

Soil-Moisture Sensor

All sensors only deliver a voltage. (In the case of MKR WAN 1310, that voltage must be limited to 0 .. 5vdc.) Voltages have to be interpreted in terms of the type of sensor and the application to which it is applied. In our case, we convert voltages delivered by soil-moisture sensors to Volumetric Water content (VWC). "VWC is the ratio of the volume of water to the unit volume of soil. Volumetric Water Content can be expressed as a ratio, percentage, or depth of water per depth of soil (assuming a unit surface area), such as inches of water per foot of soil." (Ref: Oklahoma State University Agricultural Extension)

Capacitive soil-moisture sensors are far more reliable and accurate than resistives. Resistives deteriorate rapidly. In synergy, their accuracy and reliability declines as well. For brief explorations, they may well be fine but I avoid them because of my other work. Voltage readings for all types of soil-moisture sensors have to be calibrated for the soil in which they will be used. Many companies that sell these sensors provide a table that shows calibration according to soil type. They have already carried out a laboratory-grade calibration procedure. You can find many such procedures on the internet. A general approach is described by Daniel Fernandez in his video. A similar process is described by Vegetronix. These processes result in a VWC vs Voltage curve, an example of which is shown for Miracle Grow Potting Soil. The data used to generate such plots can be used to produce an equation that relates voltage output by the sensor to VWC. An implementation of code that employs a piecewise-linear equation for Arduino is offered by Michael Ewald.

In the next sub-sections, we add a soil-moisture sensor to our MKR WAN 1310. Different soil-moisture sensors can certainly be used but their required voltages and calibrations have to be studied and implemented. Those will not be exactly the same as described here. As we go forward, we mount and test the basic arrangement. Then we calibrate for known input voltages. Finally, we add a Vegetronix VH400-2M capacitive soil-moisture sensor.

Setting Up

The first effort injected various input voltages for analog input A1, just as we did for our SimpleSensor, although that demonstration simply connected A1 to 5v. In this present case (SimpleSensor_VariableVoltage), we used a DC power supply and connected its negative terminal to its own ground and to the MKR's GND pin. The power supply's positive terminal was connected to the MKR's A1 pin. The voltage was then varied over the 0 .. 5vdc range. Figure 5 shows what appeared on the serial plotter.

Copyright © 20 June 2025 by Peter G. Raeth

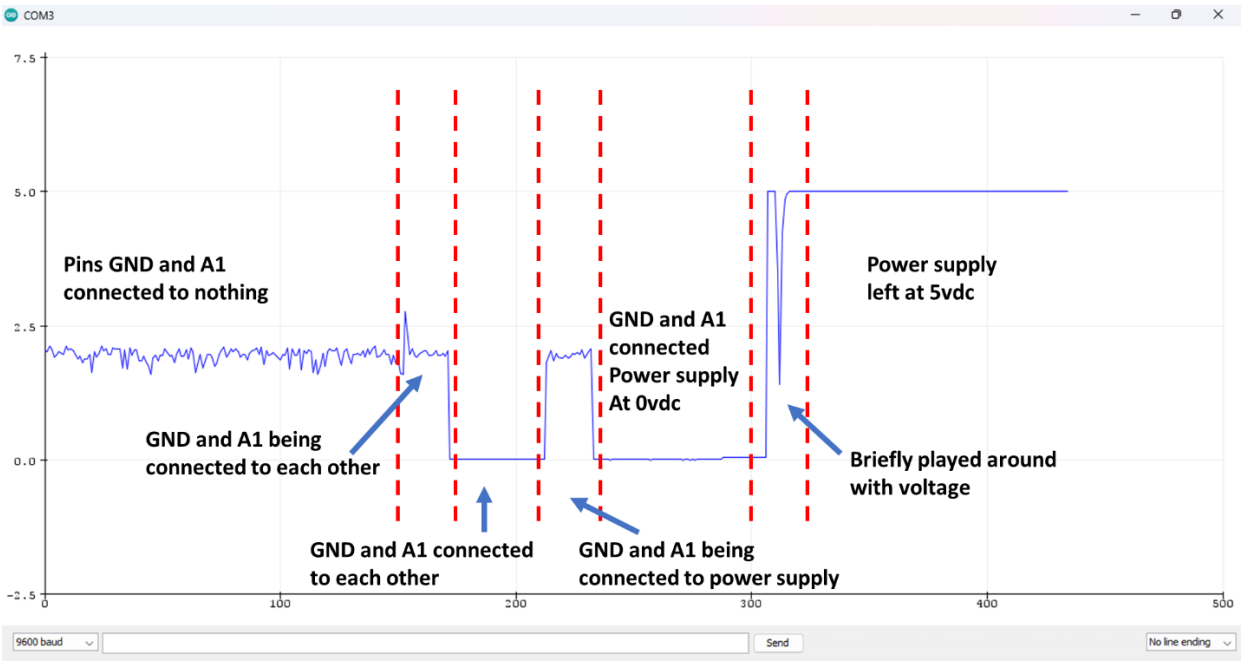


Figure 5. Simple sensor application using DC power suppl (annotated)y.

Employing a Soil-Moisture Sensor

Now we add a Vegetronix VH400-2M capacitive soil-moisture sensor (SimpleSensor_SoilMoisture). This is an industrial-grade sensor. If you are just experimenting, something less expensive can be used. *Figure 6* shows how to wire this sensor into the MKR.

Vegetronix VH400-2M Soil Moisture Sensor Wiring Table

Bare	Ground: MKR GND
Red	POWER: MKR 5v
Black	OUT: MKR A1

Figure 6. Vegetronix wiring table.

Copyright © 20 June 2025 by Peter G. Raeth

Soil-moisture sensors need a calibration equation to convert sensor voltage to volumetric water content (VWC) for the particular soil in question. There are simple and complex approaches to arriving at this equation. (Vegetronix documentation discusses that topic.)

For one particular soil, Vegetronix offers the volts-vs-VWC chart shown in *Figure 7*.

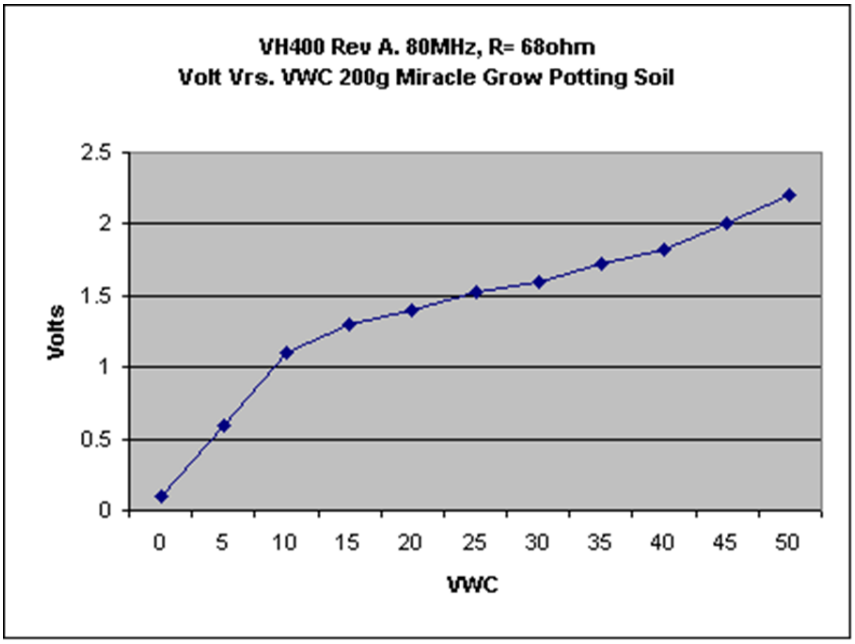


Figure 7. Vegetronix plot of volts-vs-VWC for a given soil.

They do not offer their data table but if we eyeball the chart and reverse the axes, we can arrive at an equation such as $\mathbf{y = -17.741x^4 + 77.926x^3 - 95.824x^2 + 47.161x - 3.6598}$, a fourth-order polynomial for converting x (voltage) to y (VWC). Or, we could use a set of piecewise linear equations. At issue is that their calibration does not cover the full range of their sensor's voltage. For all sensors, sensor voltage calibration is needed to convert the sensor's output voltage to units meaningful to the application. See Figure 8 for the plot of voltage as the sensor is put completely but gradually into water and then removed. You can see how the voltage changes as the available moisture changes.

Copyright © 20 June 2025 by Peter G. Raeth

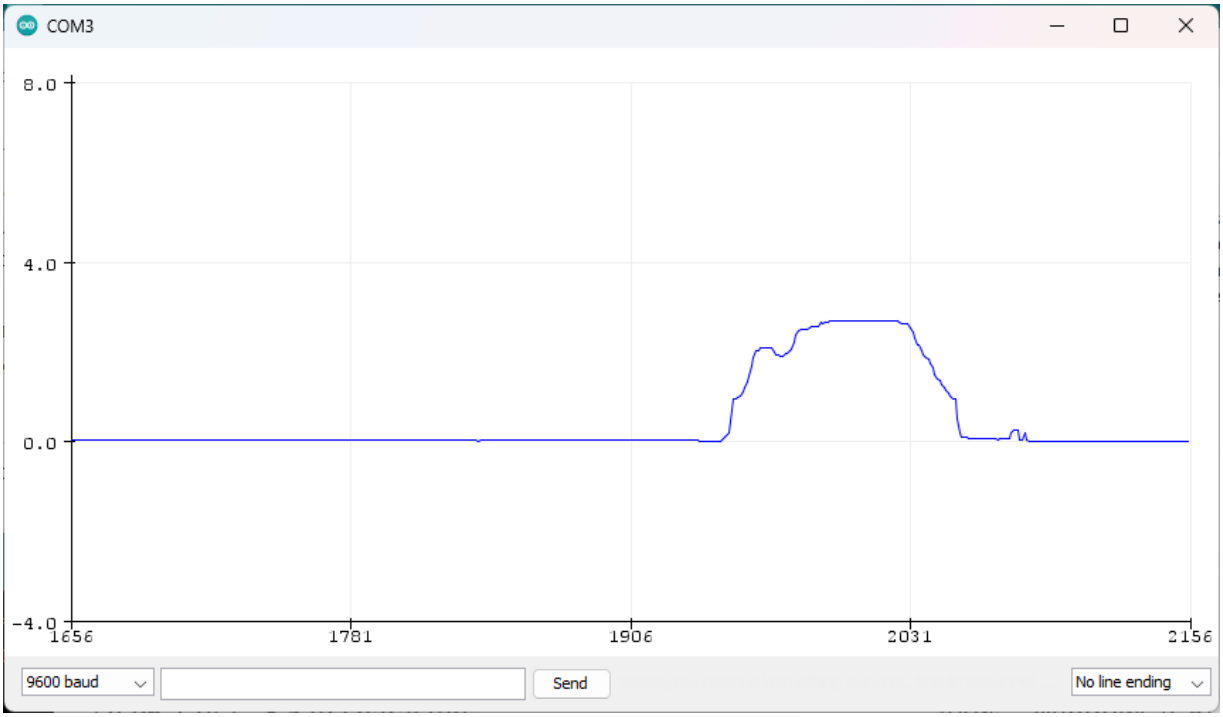


Figure 8. Voltage plot for soil-moisture sensor.

Network Inclusion

Sensor data is just like any other data that can be formatted and included in a LoRa packet. We illustrate this with our Duplex_Test software (Duplex_Sender_SoilMoisture and Duplex_Receiver_SoilMoisture). In this case, the sender transmits. The receiver receives and parses. This was achieved by deleting the transmit code within the receiver and the receive code within the sender. (The receiver does not have a sensor installed.) Load and run the codes on different MKRs. The serial monitors show what is happening.

Programmable USB Hub

There are cases when data needs to be sent through our network but the data comes from a sensor that cannot be directly connected to the MRK WAN 1310. If the sensor can be connected to a device that communicates via an accessible and controllable USB data stream, one solution is to use a programmable USB HUB to link the MKR WAN 1310 with the device to which the sensor is connected. In this section we build and demonstrate a basic USB Hub.

Copyright © 20 June 2025 by Peter G. Raeth

The following hardware is required:

- Arduino Uno R3 and its associated but separate USB cable.
  (It is possible that Arduino Uno R4 and Arduino Mega will also work.)
- RaspberryPI v4 and its associated but separate Ethernet cable. Creating a programmable USB HUB does not necessarily require such a meaty SBC. It all depends on what you want the hub to do with incoming data before the data is passed on. It also depends on how many ports of which version you need. (This particular board has two v2 and two v3 USB ports. Both work equally well.) For a specific implementation, choose the hardware, OS, and software most suited to that purpose. The device purchased can be used as part of numerous evolving proofs-of-concept. Note too that Raspberry PI has plugin modules for various applications.
- Arduino MKR WAN 1310 and its associated but separate USB cable.

Setting up the Hardware

Here is what the project did to get the hardware ready to go. The RaspberryPi-v4 is programmed in Python-v3.10. That software runs just as well on a PC so testing is easy.

RaspberryPi-v4

Here are the steps followed to get this device ready to use. This device operates just like any headless Linux device. There are also similarities with remotely-accessed Linux computers.

1. **Configured for headless operation (no desktop or keyboard).**
   Follow this documentation. To burn the OS, I employed a Windows PC and a USB SD Card adaptor. A 16gb micro-SD card proved sufficient. Used the imager's advanced-settings options to avoid having to modify the OS' files. Notice that there are many OS one can install. These depend on the purpose at hand. Selected OS/Other/Lite-64 since this device will take 64 bits and a desktop or keyboard was not needed. Using the imager's advanced-settings options greatly simplifies the configuration process.

   - Left hostname at raspberrypi.local to maintain synergy with the Pi device's documentation.
   - Enabled SSH and set password authentication plus username/password.
   - Did not set wireless LAN since that was not needed.
   - Set locale options per my geographical location.
   - Turned off all persistent settings.

   Burn took a few minutes. Process went smoothly. Dismounted the SD card from the PC and mounted it on the device. (Card faces outward from its socket, downward relative to the board, you see the face of the card as it is being inserted.)

2. **Connected device to ethernet network and plugged in the power source.**
   Which socket to use on the board is clearly marked. It does not appear possible to plug the power supply into the wrong socket. Plug the power supply's adaptor firmly into the socket. If you do it too lightly, it will not connect. However, as in all such things, if you have to force it, you

Copyright © 20 June 2025 by Peter G. Raeth

are doing something wrong. The power supply's switch is not marked for on/off. For those with good eyes in good lighting, there appears to be a nub on the switch that marks the ON position. If you press the switch down on that nub, in the direction of the device, the power is on. Once the device is booted, a red LED will show. The green blinking LED will have stopped.

3. **Checked for presence on ethernet network.**
   Router's device-list showed two devices without names. That was the correct number. Brought up the PC's command console. Entered the command "ping raspberrypi". Ping confirmed the device's presence. Used Putty and Port 22 to connect to raspberrypi. Login console appeared. Entered username/password. Linux prompt appeared. "python --version" showed that v3.9.2 is installed. That is sufficient for our purposes.

4. **Once the device's operation was verified, the next step installs the device into its case.**
   Turned off the device and then disconnected the power supply and ethernet cable. Removed the SD Card from the device. Otherwise, it is likely the card will break. Followed the instructions that come with the case. All parts and the screwdriver are provided in the kit. Note: Pins on the device are not numbered but Pin-10 is marked. Look carefully at the diagram in the instructions. Count pins carefully when connecting the fan. After Step 2 in the instructions, notice the two heat sinks provided in the kit. There is no instruction regarding the heat sinks but have a look at this video. This video shows how to install the provided heat sinks. These appear meant for the processor and memory chips. Heat sinks for the USB and ethernet controllers are not included in the kit. Before Step 3 in the instructions, installed the two provided heat sinks. After Step 5 in the instructions, the two halves of the case were not tight together. There was still looseness. Tried using a better screwdriver but found that more tightness risked stripping the screws' bodies and heads. Once Step 5 is completed, notice the back of the case, the opposite side from the fan. There are two wall mounts. Be careful of how long the mounting screws are, and the size of the heads. It is possible to damage the board or short it out. The case is not designed to accommodate pin-mounted plugins. Notice too the four round indentations on the back of the case. These are for the feet provided in the same bag with the screws. A scissor serves to cut these to size for mounting. The material surrounding the round feet is designed to peal away. Installing the feet and laying the case down on the feet improves air circulation and thus cooling for both wall and table mounting. Finally, notice the carry-case that comes with the kit. This is a nice travel and storage case.

5. **Verified installation and operation.**
   Reinserted the SD Card containing the 64-bit Lite operating system. This is a Debian Linux OS extended for Raspberry Pi. Reattached the power supply and ethernet cable. Turned the power switch to ON. The fan came on. The red and ethernet LEDs came on. Used Putty to connect to the device. Logged in. "python --version" showed that v3.9.2 is already installed. This completed implementation and verification of the device. A concern is the loose case but perhaps a plastic tie strap would fix that.

6. **Moved files from PC to RaspberryPi**
   It is necessary to move Python software files from the PC to the RaspberryPi. An excellent open-access tool for that is FileZilla. After installation, go to File/SiteManager. Use SFTP as the

Copyright © 20 June 2025 by Peter G. Raeth

protocol and raspberrypi as the host. Port remains blank. Connect and login. File transfers are drag-and-drop.

Arduino Uno R3

To learn how to set up this particular microcontroller development board, see this getting-started guidance. You will also want these details as you dig deeper. The board comes with various examples.

Top-Level Architecture

Figure 9 gives a block diagram of how the various hardware components operate together. Either Arduino component or the hub can do whatever is necessary with the data it contains.

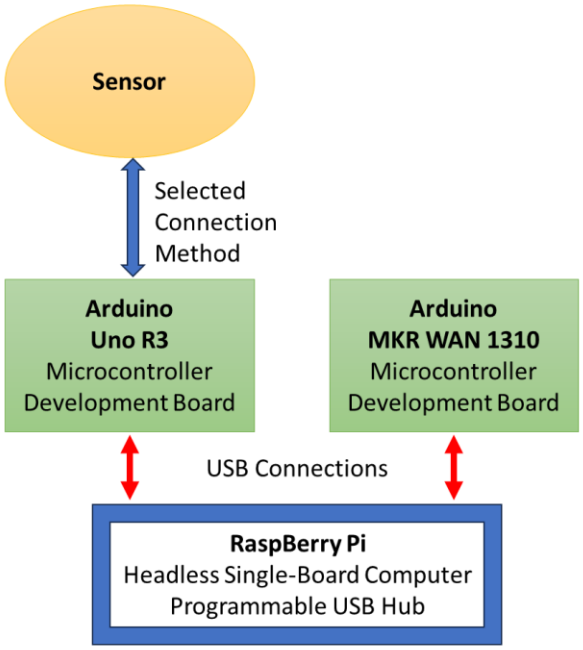


Figure 9. Illustration of programmable USB Hub.

In general, we can connect disparate devices via their USB serial ports, facilitated by a programmable USB HUB. In the following demonstrations, each connected device begins by exchanging a single-byte handshake with each device connected to the hub.

Within the hub's code, notice there are two variables: PORT_A_NAME and PORT_B_NAME near the top of the program. These have to be set to the ports being used by the two devices. When the hub starts to run, it will list all connected devices. You can see from there which port names are appropriate. If the hub does not connect to both devices, it will abend. Set the port names as appropriate and try again. (We are working with just two ports but the idea is not limited to just two.)

Copyright © 20 June 2025 by Peter G. Raeth

Simple Demonstration

As a first demonstration, both the MKR and the Uno were programmed to send a short message at random intervals. As messages arrived at each device, they were read and ignored. The USB Hub displayed the various messages as they passed through. The codes for each are under the subdirectory SimpleDataExchange. Load the USB_HUB, MKR, and Uno codes into the appropriate device. Then start the USB_HUB running.

On the Hub's display you will see something similar to:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Uno (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746460602.36 > PORT_A message length: 20; Message from MKR: 0
1746460603.37 > PORT_A message length: 20; Message from Uno: 0
1746460606.04 > PORT_A message length: 20; Message from MKR: 1
1746460606.31 > PORT_A message length: 20; Message from Uno: 1
1746460608.84 > PORT_A message length: 20; Message from MKR: 2
1746460611.03 > PORT_A message length: 20; Message from Uno: 2
1746460613.41 > PORT_A message length: 20; Message from MKR: 3
1746460613.57 > PORT_A message length: 20; Message from Uno: 3
1746460616.59 > PORT_A message length: 20; Message from Uno: 4
1746460617.30 > PORT_A message length: 20; Message from MKR: 4
. . .

As the code continues to run, you will notice a divergence between the values being reported. You will also notice that a device may generate two messages where the other device generates only one. Both are caused by the interval between messages being random and each device starting with a different random seed. This was done to demonstrate the independence of the two devices.

Copyright © 20 June 2025 by Peter G. Raeth

Basic Ping-Pong

The USB HUB can be programmed to fulfill whatever data processing/transfer needs are required. For the basic ping-pong application, the MKR WAN 1310 sends an instigator message to the other device to get the process going. This message can be of various lengths up to 63 bytes but the byte that counts is the second byte in the message, that byte is a counter. Each device increments the counter and returns the message. The hub's code is the same as earlier except right at the bottom where the message displayed by the hub is different. That part of the code is clearly marked.

Within the BasicPingPong directory three code blocks: Uno, MKR, and USB_HUB. Each should be loaded into its appropriate device.

Once the codes are all loaded and all the connections to the hub are made, start the hub running. On the hub's display will be something like:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Uno (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746471931.71 > PORT_A message length: 63. Value of Interest: 1
1746471931.93 > PORT_B message length: 63. Value of Interest: 2
1746471934.13 > PORT_A message length: 63. Value of Interest: 3
1746471934.36 > PORT_B message length: 63. Value of Interest: 4
1746471936.56 > PORT_A message length: 63. Value of Interest: 5
1746471936.78 > PORT_B message length: 63. Value of Interest: 6
1746471939.09 > PORT_A message length: 63. Value of Interest: 7
1746471939.31 > PORT_B message length: 63. Value of Interest: 8
1746471941.51 > PORT_A message length: 63. Value of Interest: 9
1746471941.74 > PORT_B message length: 63. Value of Interest: 10
1746471944.04 > PORT_A message length: 63. Value of Interest: 11
1746471944.27 > PORT_B message length: 63. Value of Interest: 12
. . .

Copyright © 20 June 2025 by Peter G. Raeth

Camera Access Example

Within Arduino's product line, there are several cameras. However, none of these appear to be hats for the MKR WAN 1310. Seems like they would have to be wired in. A camera available for the Uno R3 is the PixyCam v2 camera. It plugs directly into the Uno's SPI port. Of the many third-party cameras that operate through USB connections, I could find none with an accessible and interpretable datastream. So, for this demonstration, I stuck with the Pixy2 to demonstrate a realistic means of connecting third-party sensors to a LoRa network without soldered wiring or jumpers.

Very good documentation is provided for the Pixy2. Follow it carefully. Depower the Uno and plug the Pixy into that board's SPI connector. There are lots of good examples as you go along.

Top-Level Architecture

Figure 10*Figure 9* shows a block diagram of how the various hardware components operate together. This approach expands on what we did with earlier applications of the Programmable USB Hub illustrated in Figure 9. Notice how the camera communicates directly with the Uno via its SPI plug. (Such a plug does not exist on the MKR.) The Uno queries the camera and forwards data as appropriate in a generalization of a sensor query/response/messaging process.

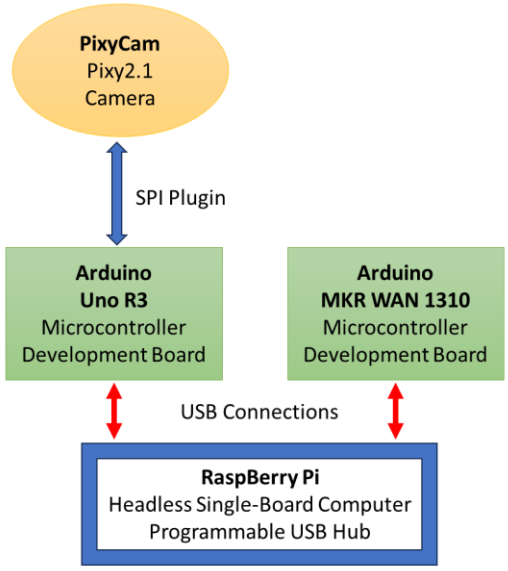


Figure 10. PixyCam connected to Uno R3 within Programmable USB Hub.

This is one way we can connect two disparate devices via their USB serial ports, facilitated by a programmable USB Hub. (It is true that some sensors will connect directly to the RPi.) In this trial (CameraAccessExample), the MKR asks the Uno to send the pixels of an entire image. Uno responds by sending messages that contain each pixel's data, one pixel at a time. The Hub enables and reports the message interchange. We use the very same USB Hub as in the BasicPingPong example except that the hub will only display the length of messages being passed. Nothing else changes in the hub, compared to earlier examples.

Copyright © 20 June 2025 by Peter G. Raeth

As the system runs, you will see something like this on the Hub's display:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Uno (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746478403.24 > PORT_A message length: 15
1746478403.28 > PORT_B message length: 8
1746478404.28 > PORT_B message length: 8
1746478405.28 > PORT_B message length: 8
1746478406.28 > PORT_B message length: 8
1746478407.28 > PORT_B message length: 8
1746478408.29 > PORT_B message length: 8
1746478409.29 > PORT_B message length: 8
1746478410.29 > PORT_B message length: 8
1746478411.29 > PORT_B message length: 8
1746478412.29 > PORT_B message length: 8
1746478413.29 > PORT_B message length: 8
1746478414.30 > PORT_B message length: 8
...

You can imagine how long this will take to send an entire image. A better message formation is needed, one that goes beyond pixel-by-pixel.

Camera Access Improved

At issue with single-pixel messages is the large number of messages required. One wishes more than one pixel could be included in a message. That is certainly possible.

Recall that Arduino I/O buffers are only 63 bytes long. Lets assume for now that is the maximum message size we can work with. Certainly, all the pixels in a typical image are not going to fit into one message. It is necessary to partition the image somehow. A convenient partitioning is to go row by row through the image and partition the columns according to the number of layers that describe

Copyright © 20 June 2025 by Peter G. Raeth

each pixel. For example, grayscale images have only one layer. So, only one data value is needed. That value covers the range 0..255. RGB images from a typical camera require three data values. NirRGB images require four values. Some images require thousands of values per pixel but that is another story. The general idea behind image partitioning, as used here, is summarized in Figure 11.

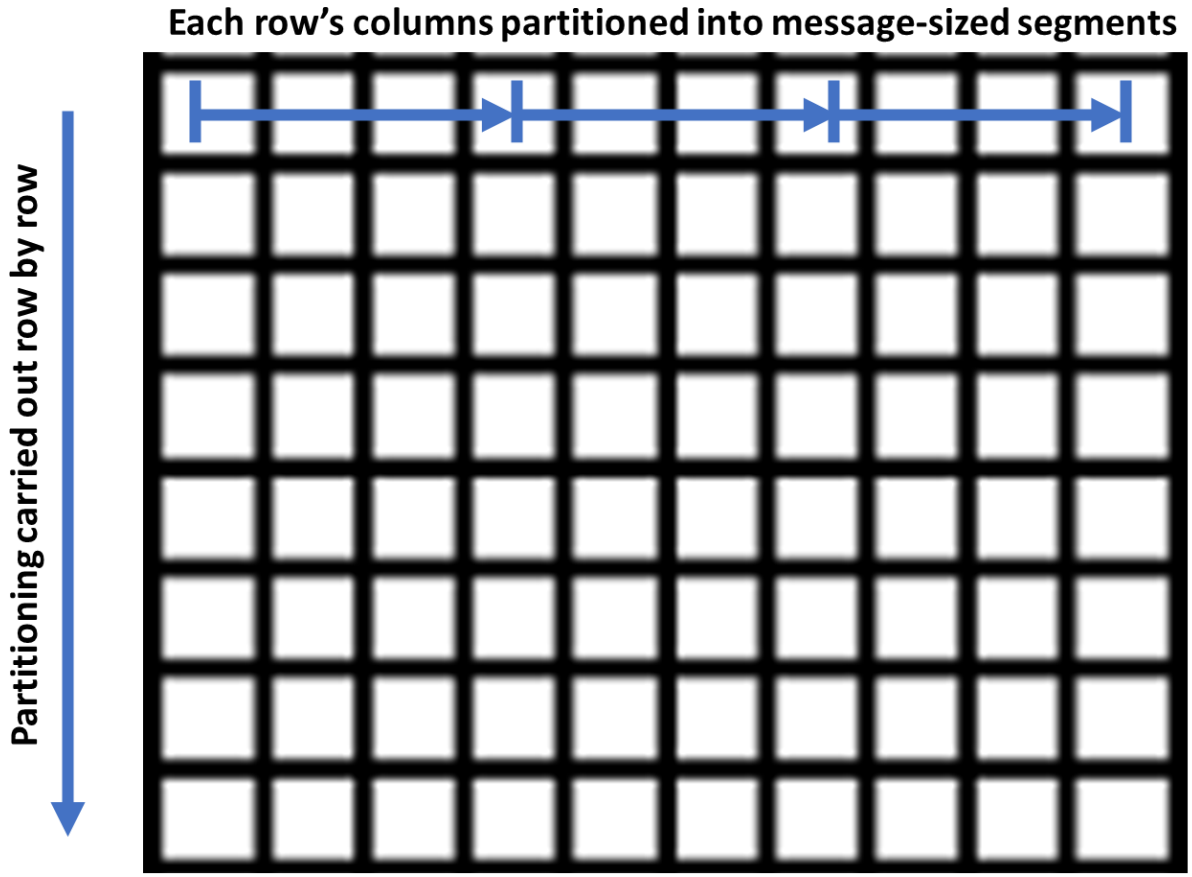


Figure 11. Summary of image partitioning.

In the subdirectory ImprovedCameraAccess, code to perform this partitioning has been added to the Uno's code. Notice that the Uno's code has been rearranged so that this new approach to messaging is better accommodated. It also assumes the request is for an entire image.

Copyright © 20 June 2025 by Peter G. Raeth

When the various codes are uploaded to the appropriate devices, something like this will be seen on the hub's display:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Uno (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746484692.68 > PORT_A message length: 15
1746484692.75 > PORT_B message length: 55
1746484694.82 > PORT_B message length: 55
1746484696.87 > PORT_B message length: 55
1746484698.94 > PORT_B message length: 55
1746484701.00 > PORT_B message length: 55
1746484703.06 > PORT_B message length: 55
1746484705.13 > PORT_B message length: 55
1746484707.22 > PORT_B message length: 55
1746484709.34 > PORT_B message length: 55
. . .

This continues until all image segments are sent through the hub.

At issue with all but the demonstrations but PingPong is that data keeps flowing at the whim of the microprocessors. This is fine as long as there is sufficient time between messages for whatever has to be done with each message. The hub could be programmed with threads so that there is a thread is assigned to each port to do nothing but capture and store messages. Then the opposite port can retrieve the next message on demand. That would work. A simpler approach for this particular project is to expand the PingPong idea so that the microcontroller controlling the campera only sends its next segment when the MKR requests it. Then there is no need for estimating a wait time. This is especially the case when large messages are involved. Some kind of data flow control is needed. Lets see what can be done about all that.

Copyright © 20 June 2025 by Peter G. Raeth

Expanded Ping-Pong

At issue with Arduino's Uno R3 is its limited memory. The Arduino Mega 2560 overcomes that. (So does the Arduino Uno R4 but that has not been tried in this project.) All the previous demonstrations work well on the Uno R3 but the ImprovedCameraAccess demo stretched its memory so much that nothing more can be added. For using a longer message so that fewer image-segment messages need be sent takes us beyond the Uno R3 memory limit. So we now switch to the Mega 2560. There is some increase in cost but not significantly.

This new approach in ExpandedPingPong uses longer messages (222 bytes due to LoRa limits in the USA). It also uses different means of reading and writing messages. This derives from a lot of experimentation and study on data flow control. The new approach sends messages in such a way that the small I/O buffers are not overrun. A new message is not sent until it is requested.

You will notice in the Mega code a reference to Serial1. That connection was used to help debug the code running on that device. All such references can be removed. If you want to see how those references were used in debugging, see this GitHub project. It is a good use for Arduino Uno R3.

Once you have the various codes loaded into the appropriate devices, you should see something like this on the hub's display:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Mega 2560 (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746487446.8780675 > PORT_A message length: 223. Value of Interest: 1
1746487447.4229777 > PORT_B message length: 223. Value of Interest: 2
1746487447.447337 > PORT_A message length: 223. Value of Interest: 3
1746487447.9916184 > PORT_B message length: 223. Value of Interest: 4
1746487448.0157685 > PORT_A message length: 223. Value of Interest: 5
1746487448.5616803 > PORT_B message length: 223. Value of Interest: 6
1746487448.5853834 > PORT_A message length: 223. Value of Interest: 7
. . .

Copyright © 20 June 2025 by Peter G. Raeth

Expanded Camera Access

Now we apply ExpandedPingPong to the problem of getting an entire image off the camera. Look in the ExpandedCameraAccess directory. The same hub is used, except the displayed text is different. After depowering the Mega (or Uno, if you want to use shorter messages), mount the camera, and repower.

For the USA, and this type of project, the maximum content of a LoRa packet is 222 bytes. If we use that number, it took 1020 messages over 4.54 minutes to pass an entire image from the camera at hand, through the hub, to the MKR. If we limit packet content to 63 bytes, the size of Arduino I/O buffers, then the runtime was 10.88 minutes. 4080 messages were sent. It is important to note that nothing was done with the messages arriving at the MKR. Ultimately, these will become LoRa transmission packets. That will add additional time: waiting for a clear channel, forming the packet, and transmitting the packet. (USA has no restrictions on numbers of packets, just on time for packet transmission. The goal is very short bursts but to make best use of clear channels.)

After loading the code into the appropriate devices, something like the following will appear on the hub's display for the 63-byte case:

Serial ports:
Detected 3 total ports

Name	*Device*	*Manufacturer*	*Description*
COM1	*COM1*	*(Standard port types)*	*Communications Port (COM1)*
COM6	*COM6*	*Arduino AG (www.arduino.cc)*	*Arduino MKR WAN 1310 (COM6)*
COM15	*COM15*	*Arduino LLC (www.arduino.cc)*	*Arduino Mega 2560 (COM15)*

Configuring serial ports...
Connected to PORT_A.
Port Name: COM6
Port Baudrate: 9600
Connected to PORT_B.
Port Name: COM15
Port Baudrate: 9600

Looking for PORT_A handshake: Success. Sending handshake to PORT_B.
Looking for PORT_B handshake: Success. Sending handshake to PORT_A.
Handshakes exchanged

1746556342.1618145 > From PORT_A: Message of length 2
1746556342.1618145 > From PORT_A: Message of length 2
(1) 0.0 min > From PORT_B: Message length 55; 2 / 2
1746556342.237089 > From PORT_A: Message of length 2
(2) 0.0 min > From PORT_B: Message length 55; 2 / 18
. . .
1746556994.722074 > From PORT_A: Message of length 2
(4080) 10.88 min > From PORT_B: Message length 31; 205 / 306
1746556994.8551452 > From PORT_A: Message of length 2
(4081) 10.88 min > From PORT_B: Message length 5; 17519 / 28261

Copyright © 20 June 2025 by Peter G. Raeth

Some Thoughts

The project spent a lot of time exploring programmable USB Hubs and resolved several issues. The demonstrations show various applications. Of course, there are other ways of connecting devices. One can try direct wiring, for example, or devices that convert USB to other port types. Programmable USB Hubs are good for cases where direct wiring is not tractable and when one wants to avoid USB adaptors. Yes, the hub itself has to exist but that is a general-purpose programmable device. Assumed too is that the devices to be connected via a programmable USB Hub both have cooperating data streams. The hub has to be programmed to accommodate the data streams in question. In our case, we were able to program all devices to suit our application.

One interesting debugging tool is python's trace ability. At one point, we started the hub using "python -m trace --trace main.py". This did give helpful information. See this link for more on python trace. Details are given in python documentation. PyCharm's debugger was very helpful as was this probe.

It is important to remember that USB ports are not COM ports. Nor are they RS-232 ports. More discussion is available. It is possible to add USB/<port type> adaptors to Arduino devices but that is an additional cost and requires appropriate programming.

An Example Sensor Network

We now put various pieces together into an example network that goes beyond the grassroots example demonstrated earlier. This new example includes a single-value sensor, camera, and GUI display. This new example is illustrated in Figure 12.

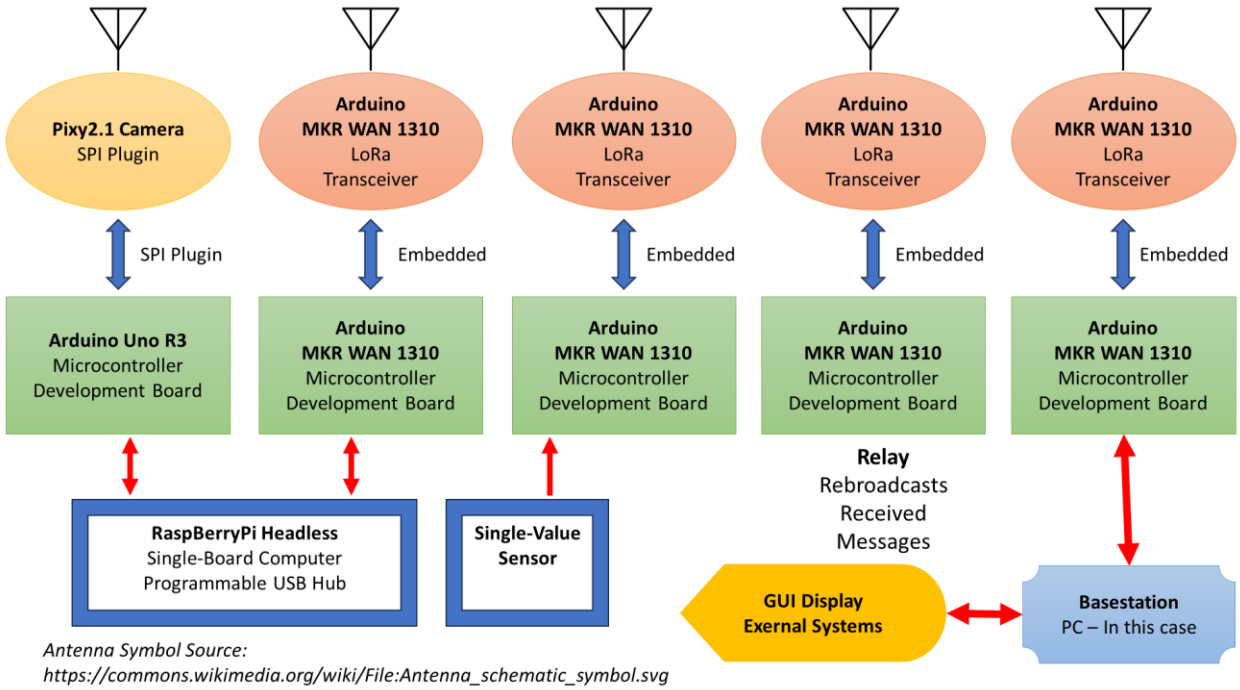


Figure 12. LoRa network with sensor, camera, and GUI display.

Copyright © 20 June 2025 by Peter G. Raeth

A Caution about Overheating

While developing this example, the MKR device used to transmit the camera's messages stopped working and could not be recovered. There was no physical damage and great care was used in its handling and placement while operating. While I do not have the skills nor the equipment to determine exactly why, previous reading and experience make me suspect overheating. Numerous trials were conducted during development. These caused thousands of messages to be transmitted over a relatively short period of time. (All messages met US legal standards, as this layman understands them.)

In an effort to compensate for the assumed overheating, a search for device-specific heatsinks was conducted. None were found. Oversized heatsinks were purchased and cut down to size. (Be very careful and safe when doing this.) The crude result was attached to the MKR's processor and LoRa chip using the adhesive already part of the heatsink. Since then, the new MKR device has survived all trials. Still, multiple trials over a short period of time are no longer necessary. There is an appendix that offers more details on this issue.

Use of CRC for Error Detection

One must assume that some messages will be corrupted during their transit from transmitter to receiver. To demonstrate this, three transceivers were placed indoors several feet apart. 1020 camera byte messages were sent at the same time as 1020 ascii text messages (2080 messages total). At the destination, 8 text messages were found to be corrupted. If one assumes the same number of camera message corruptions, the corruption ratio is 16 / 2080, 0.77%.

From reading the Lora library documentation, bug reports, and pull requests, the baseline library does contain a functional error-detection (CRC) capability. This capability was activated. It is a detection capability on the LoRa chip that rejects failed messages without notice. Rerunning the same test, there were zero corruptions detected at the receiving node since corrupted messages were rejected by the transceiver before being made available to the processor.

While a 0.77% error rate may not seem like much, it still allows erroneous data to be accepted. Depending on the application, that can be serious. It is true that vetting data should always be done anyway. But it is better to reject messages with known errors.

Also, imagine a network with several relays. It is possible for the message ID to be corrupted. If it is made much larger than the current ID then all subsequent messages will be rejected until the incrementing ID catches up with the corrupted ID. Many messages could be lost thereby. One can readily understand the network malfunction that would result.

It is possible, as we have seen, for the message itself to contain CRC data. That requires message space, memory, and execution time to carry and process the CRC data. The advantage would be feedback when corruption is found. In the case of in-message CRC, on-chip CRC should be disabled.

Given signal and message corruption, we should expect to miss some data that is circulating in the network. That has to be recognized, as needed, in any process that receives and processes data.

Copyright © 20 June 2025 by Peter G. Raeth

For example: A message could be uncorrupted but contain faulty data due to sensor malfunction. Or, if calculating an average, one cannot not assume a given number of data points arrived.

Software

Look in the directory NetworkApplication. There are codes for the various modules. They are loaded in the same way as the grassroots network and USB_HUB examples. The camera's transceiver now broadcasts the messages. You will notice the handshaking going on between the camera and the transceiver. This was found to be necessary to keep the two in snyc.

The basestation has a transceiver module and a PC module. The transceiver acts much like a relay but does not rebroadcast any messages. Instead, it transfers messages intended for the basestation to the PC. The basestation itself runs two threads. One thread only takes in messages and puts them into a first-in/first-out queue. The other thread takes messages from the queue and processes them.

GUI

As messages are received, there is any number of things that could be done with them. Each message contains a message-type, a source-id, and a sensor-number. For text messages, these can use keywords to indicate what to do with the text. Data messages are plotted and images are displayed, in this example, but there is no end to the possibilities, including equipment control.

When the PC side of the basestation starts, you will see the GUI shown in Figure 13.

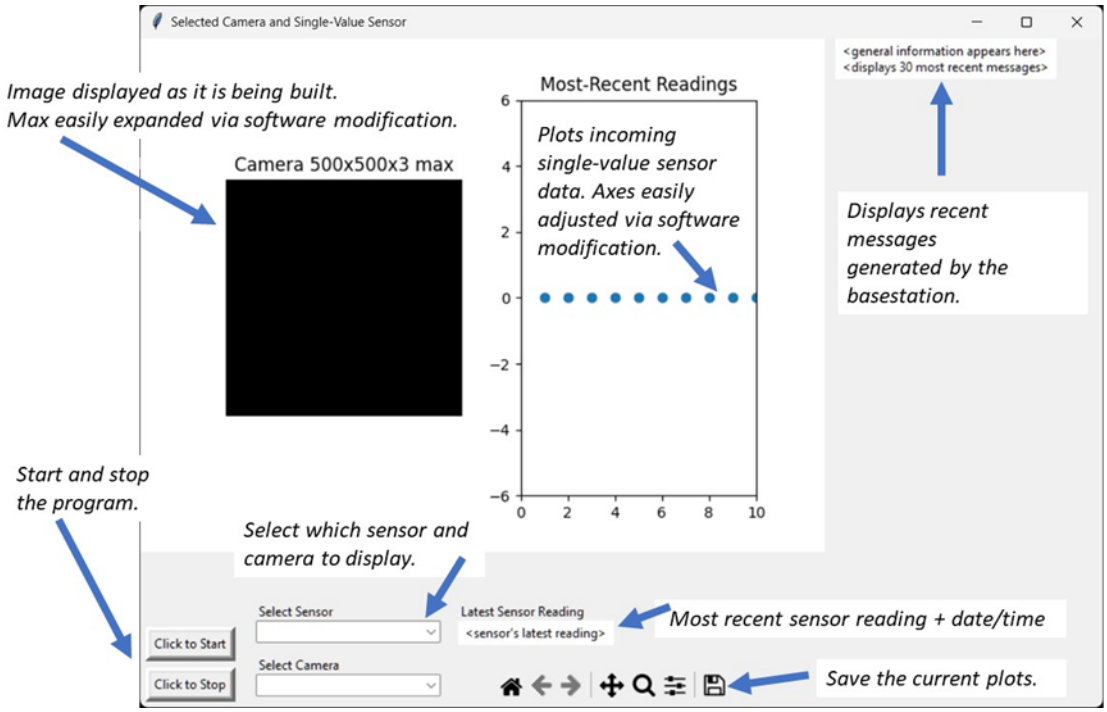


Figure 13. Bastation GUI (annotated).

Figure 13 is annotated to explain the GUI's components. Originally, the GUI would automatically set the axes for the plot of the single-value sensor. It would also adjust the display area for the picture.

Copyright © 20 June 2025 by Peter G. Raeth

However, this required constantly redrawing the GUI and took far too long. The underlying graphics libraries, tkinter and matplotlib, support an animation capability but that requires assumptions on image size and plot ranges. I chose animation so that a reasonable response time could be achieved. There may well be a better approach. That is something that could be explored.

Once the GUI is displayed, nothing else will happen until the START button is pressed. Having done that, you may see something like the following on the command console:

11:33:03: USB_Serial_Connection: Starting
11:33:03: Serial Ports:
11:33:03: Detected 1 total ports
11:33:03: COM1 : COM1 : (Standard port types)
11:33:03: Configuring serial port COM12
11:33:03: could not open port 'COM12': FileNotFoundError(2, 'The system cannot find the file specified.')
11:33:03: Is the COM12 device connected and active?
11:33:03: Has COM12 name and baudrate been correctly specified?
11:33:03: Is the IDE Serial Monitor of the device connected to COM12 been deactivated?
11:33:03: LoRa transceiver is not connected.
11:33:03: Exiting thread. (USB_Serial_Connection)

That is what happens when no devices are plugged into one of the basestation's USB/Serial ports. Once the module is plugged in and the correct port name is given at the top of the basestation PC's main.py, one can either restart the program or just press START again. Something like the following will appear:

11:51:05: USB_Serial_Connection: Starting
11:51:05: Serial Ports:
11:51:05: Detected 2 total ports
11:51:05: COM1 : COM1 : (Standard port types)
11:51:05: COM12 : COM12 : Arduino LLC (www.arduino.cc)
11:51:05: Configuring serial port COM12
11:51:10: Connected to port COM12
11:51:10: Awaiting Messages...

The basestation will now respond to all messages forwarded by the transceiver component of the basestation. Figure 14 shows the display after receiving messages from only a single-value sensor, in this case, the node's battery voltage.

Copyright © 20 June 2025 by Peter G. Raeth

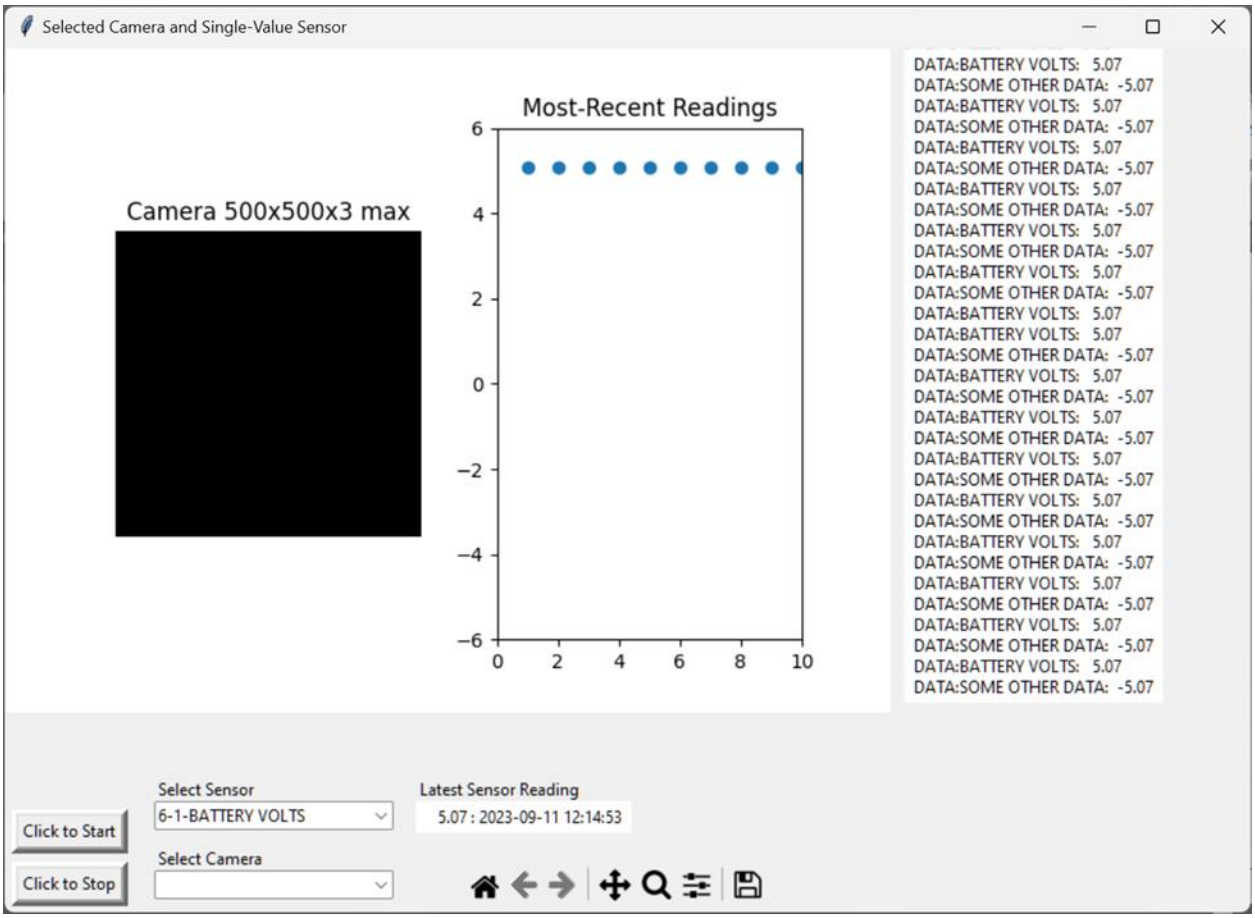


Figure 14. Basestation GUI after receiving single-value sensor data.

Overlaying the Camera Image

At the same time as the basestation is responding to single-value sensor data, it can also respond to multi-value sensor data. Cameras are an example of such sensors. As camera data is received, it is displayed on the left side of the GUI.

Ultimately, we want the camera to await a request to send another image, or send images on a schedule. At the present time, sensor nodes in the flood-messaging system do not receive messages. Rather, they send their readings on a periodic basis. In the case of this example, the camera sends an image upon connecting with the transceiver. It then waits for the transceiver/camera/hub to be reset. That situation will improve as the project proceeds.

Part of the reason for the length of time to send one image (about 20 minutes) is that the Pixi2.1 camera cannot take a static image and hold it in memory. It is necessary to query each pixel separately, rather than pointing to a new location in the camera's memory. Notice too that 1021 messages are needed to send the entire segmented image if the maximum message length is used. Otherwise, more messages are needed. It all depends on the memory of the device hosting the camera. Another reason is the amount of diagnostic information generated per message by the hub. This was necessary for testing but can be eliminated prior to installation. One has also to decide if echo-testing of messages is still needed.

Copyright © 20 June 2025 by Peter G. Raeth

See Figure 15 for an example of what the GUI will display regarding camera images.

Figure 15. Example display of camera data.

You will notice that the image itself is in the upper-left corner of the camera display. That is because the camera is of smaller dimension than the maximum display dimension.

The GUI presented here is rather fundamental and simple. There may be better development approaches. For instance, Google's Flutter may be worth investigating.

Copyright © 20 June 2025 by Peter G. Raeth

Appendix – Hardware Component Sources and Documentation

Here is a table of all hardware components used in this project, prices (at time of publication), and reliable purchase sources. Shipping, import fees, and taxes are not included since those vary according to your source, location, and volume purchased. Some source links are affiliate. Purchases through those links provides a bit of income to this project. Where a component is mentioned in the document, refer to this table for source of purchase. Complete documentation on the component in question is found directly on the product page or is linked from the product page.

Description (See product pages for documentation)	**Price (USD)**	**Total Number Required**	**Source** (2)
A suitable Linux or Windows computer is assumed to exist and to have at least five independent, available, and powered USB ports. (4)			
Basic Essentials - Grassroots Network			
Arduino MKR WAN 1310 Baseboard , with antenna	46.00	3	Arduino Official Store
Micro USB Cable (USB A / USB B)	6.90	3	Arduino Official Store
Male-Male jumper wires	4.80	1	Arduino Official Store
(1) Total Cost	163.50		
Optional Accessories			
Stand-alone magnifying light	40.00	1	Amazon
(3) Soil-Moisture Sensor	45.00	1	Vegetronix
(5) Heatsinks that can be cut to size	7.49	1	Amazon
Total if all purchased:	92.49		
For Exploring Image Transmission			
Arduino Uno R3	28.00	1	Amazon
USB 2.0 Cable	7.00	1	Amazon
RaspberryPi single-board computer	129.00	1	Amazon
Ethernet Cable	7.00	1	Amazon
Micro SD Card	8.00	1	Amazon
USB Micro SD Card Adaptor	9.00	1	Amazon
PixyCam + Connector Cable	70.00	1	Amazon
Total if all purchased:	258.00		

Copyright © 20 June 2025 by Peter G. Raeth

(1) Prices are at time of writing. They do not include shipping or taxes.
"Cost" refers to the sum of individual prices times the number required.

(2) "Arduino Official Store" refers to Arduino's global network.
See the bottom-right of the page to select your region.

(3) This is an industrial-grade sensor that happened to be already
on hand. For brief indoor experiments, cheaper sensors will do. For instance:
https://amzn.to/40DeuvC.
One has to study and understand the new sensor, and correctly integrate it
relative to the host device and the soil in question.

(4) My personal experience is that a USB port extender, even powered,
is insufficient to meet this criterial. Rather, the ports need to be internal to
the PC and independent of each other. To be sure, this adds cost to the PC.

(5) Take great care with good safety when cutting these.
The result is crude but appears to work well.
If you find an alternative, please post an issue.

Copyright © 20 June 2025 by Peter G. Raeth

Appendix – LoRa Network Types

This project employs LoRa communications technology. LoRa (Long Range) employs unlicensed radio frequencies whose use and configuration are determined country-by-country. An important aspect of this means of radio communication is that LoRa broadcasts are very resistant to electromagnetic interference and physical obstacles (within reasonable limits). The result is improved signal penetration and propagation. This capability is due to the digital modulation employed, which is based on spread-spectrum technology. (Links to LoRa details are given earlier in this document.)

There are two major domains within LoRa application, LoRaWAN (LoRa for wide-area networks) and LoRaP2P (LoRa for direct peer-to-peer networks).

LoRaP2P operates without the need for a gateway. LoRaWAN employs a full network protocol and employs gateways as its means of providing network management and security. Since LoRaP2P does not employ a gateway, it is simpler and less expensive to implement for localized applications. LoRaWAN is meant for large-scale networks with a large number of devices. Like Ethernet UDP LoRaP2P messages have a high chance of delivery but delivery is not guaranteed. Nor is message order ensured.

Within LoRaP2P, there are two approaches to building a network when direct sender/receiver connections cannot be ensured, flood and mesh networks. Both create networks where data can hop between multiple devices through various routes. This enhances reliability and range within challenging environments by allowing data to be sent and received in spite of obstacles so that data has a better chance of reaching its destination.

Flood networks use very simple networking and messaging protocols, whereas mesh networks use more complex network management and message routing approaches. The mesh networks observed by this author to date depend on real-time operating systems and specific computational hardware. On the other hand, the flood network employed in this project has much less dependency on specific hardware and does not use a real-time operating system. Both network types have to be configured for the transceiver to be employed. The implementation of flood networking used in this project is easily transportable from one computational device to another.

Flood and mesh networks operate by connecting multiple devices (nodes) together, where each node can act as a message relay. This allows data to be transmitted across the network by "hopping" from one node to another until it reaches its destination, effectively creating a self-healing network that can reroute data if one node fails. Essentially, every device on the network can send and receive data directly with each other, not just through a central point or gateway, creating a robust and flexible communication system.

Copyright © 20 June 2025 by Peter G. Raeth

Both flood and mesh approaches have the following benefits:

- Ability to connect numerous devices, each acting as a network node that can both send and receive data.

- Transmitted data is forwarded so that it reaches the target node.

- Network congestion and node drop-outs are overcome.

- Multiple connection paths increases the chance of message delivery. If a node fails, data can still be transmitted through alternative routes, ensuring network resilience.

- There is nothing that prevents a network node from connecting to external systems.

This project employs flood networking. A good example of mesh networks is Meshtastic. But this author's experience is that it is very hardware dependent. Still, international networks have been built using this approach. As this project is focused on local low-cost networks that are flexible regarding hardware, we continue to develop flood networks. The author has built rudimentary networks (one each: sensor, relay, basestation nodes) for under $175US. LoRaWAN begins with a gateway at a cost of around $300US.

Copyright © 20 June 2025 by Peter G. Raeth

Appendix – Potential for Overheating

This appendix summarizes material found on these two websites:

- https://www.hackster.io/savedabees-co/connected-hive-using-mkrwan-1310-3770b1
- https://docs.arduino.cc/tutorials/mkr-wan-1310/lorawan-regional-parameters

While the Arduino MKR WAN 1310 is generally not prone to overheating, there are situations where it might. It can also be the case that other issues, like wiring errors or excessive current draw from pins, are the cause of malfunctions and "burnouts". Overheating is typically associated with prolonged heavy data transmission or running on batteries with a high discharge rate.

Factors that could lead to overheating:

- High data transmission: Continuously transmitting large amounts of data can generate heat, especially if the board is in a poorly ventilated environment. (The environment of this project is well ventilated.)

- Battery usage: If the MKR WAN 1310 is powered by a battery, particularly one that's not compatible or not being managed properly, the MKR could overheat.

- Poor ventilation: If the MKR WAN 1310 is placed in a confined or poorly ventilated space, the heat generated can build up and potentially lead to overheating.

Signs of potential overheating:

- Board malfunction: If the board stops working, especially during or shortly after a period of heavy data transmission, it could be a sign of overheating.

- Difficulty transmitting messages: If the board is having trouble transmitting even simple text messages, it could indicate that it's overheating.

What to do if you suspect overheating:

- Check the environment: Ensure that the board is in a well-ventilated area.

- Monitor battery health: If using batteries, make sure they are compatible with the MKR WAN 1310 and that the power is being managed properly.

Copyright © 20 June 2025 by Peter G. Raeth

- Reduce data transmission: If possible, reduce the amount of data being transmitted at a time to see if it helps.

- Use a heatsink: While not typically needed, a small heatsink could be considered for areas where the board is exposed to prolonged heat.

Heatsinks are generally not needed for the Arduino MKR WAN 1310 under normal operating conditions. The device is designed to be robust and can typically handle its own heat dissipation without needing additional cooling measures.

• LoRa functionality: The MKR WAN 1310 is designed for long-range communication, which typically involves lower power consumption than other types of radio communication.

• Optimized design: Arduino boards are engineered to be compact and energy-efficient, with the MKR WAN 1310 being no exception.

• Low power use: The LoRa protocol itself is designed to be energy-efficient, further reducing the need for heat dissipation.

While not typically needed, there might be specific scenarios where a heatsink could be beneficial:

• High ambient temperatures: If the MKR WAN 1310 is used in an environment with consistently high temperatures, a heatsink might be helpful.

• Overclocking or high-power operations: If the device is overclocked or used in a way that significantly increases its power draw, a heatsink could be considered.

• Very long communication intervals: If the device is transmitting for extended periods with long communication intervals, the heat generated could be a factor.

For normal use, a heatsink is unlikely to be necessary for the Arduino MKR WAN 1310. However, if there are specific reasons to suspect that the device might overheat, a heatsink can be a way to ensure it remains within its intended operating temperatures.