* [Serial Connection](https://andresrcs.rbind.io/2020/02/09/arduino_microcontroller_r/" \l "serial-connection" \t "_blank)
* [Writing to a SQL server](https://andresrcs.rbind.io/2020/02/09/arduino_microcontroller_r/#writing-to-a-sql-server)
* [Using the MQTT protocol](https://andresrcs.rbind.io/2020/02/09/arduino_microcontroller_r/#using-the-mqtt-protocol)
* [Disclaimer](https://andresrcs.rbind.io/2020/02/09/arduino_microcontroller_r/#disclaimer)

Have you ever wanted to get your own real-world data using sensors? With the advent of the [Open-Source Hardware](https://en.wikipedia.org/wiki/Open-source_hardware) movement, this has become more accessible for the general public, microcontrollers and sensors have become cheaper and easier to program with human-friendly languages like Arduino (C++) and MicroPython (Python), so now anybody can produce its own real-world sensorial data, but, there is still a problem, How do we get this data into R so we can analyze it?

Depending on your application’s requirements you could choose among a plethora of options. For example, if you only need data on batches and you can have easy physical access to your device, you could simply add an SD card module to your project and log the data to a .csv file on an SD card but, that wouldn’t be fun, isn’t it? A much cooler solution would be to retrieve data remotely and preferably wirelessly, don’t you think?. Let’s explore some options.

**Serial Connection**

If you have never thought about getting data from microcontrollers before, I bet the first thought that comes to your mind, is to somehow directly connect the microcontroller with R to get the data, but this is currently the most technically challenging option, it can be done using a serial connection between the microcontroller and your computer.

I can’t test this approach myself since I don’t have a physical Linux or macOS machine to connect a microcontroller to, so for this case, I’m just going to link to a [blog post](https://zhuhao.org/post/connect-arduino-chips-with-r/) by [@haozhu233](https://twitter.com/haozhu233) (the package author) showcasing its functionality.

If your application requires real-time data acquisition or very low latency this might be your only option and you may want to dive deeper into this.

Streaming arduino sensor signals right into [@rstudio](https://twitter.com/rstudio?ref_src=twsrc%5Etfw)! I wonder if there is really anything [#rstats](https://twitter.com/hashtag/rstats?src=hash&ref_src=twsrc%5Etfw) can’t do [pic.twitter.com/nT6q6TT6xc](https://t.co/nT6q6TT6xc)

**Writing to a SQL server**

There are two ways of doing this, one is to directly connect the microcontroller to the database and the other is to use another service to pull data from the microcontroller and write it into the database.

The first option is harder to accomplish, as far as I know, there is only one [Arduino library](https://www.arduinolibraries.info/libraries/my-sql-connector-arduino) that provides a SQL connector for MySQL servers but, there are no already-made solutions for other SQL servers and although you could write your own connector for other Open Source SQL servers (like PostgreSQL), that would require a considerable amount of technical skills.

The second option is easier to implement, and my personal favorite.

The first step is to broadcast the data from the microcontroller, one way of doing this is to implement a simple web server on the microcontroller itself and serve the data on a suitable format like HTML, JSON or CSV.

This is a simple example of an Arduino Sketch for an ESP8266 board (Wemos D1 mini) and DS18B20 temperature probe sensors, that serves HTML and JSON outputs over a web server.

The second step is to pull the data out of the microcontroller and into a SQL server, there are a lot of approaches and languages you can use to implement this part but obviously we are going to use R. You can write an R script that connects to the microcontroller, retrieves the data and writes it into a SQL server, and schedule the script with a cron job.

#! /usr/bin/env Rscript

# DATA ADQUISITION##############################################################

raw\_data <- rjson::fromJSON(file="<http://192.168.0.102/reading.json>")

current\_time <- as.character(Sys.time())

# PARAMETERS ###################################################################

current\_location = "some\_place"

point\_names = c("probe0" = "measurement\_point\_0",

"probe1" = "measurement\_point\_1",

"probe2" = "measurement\_point\_2")

# LIBRARIES ####################################################################

library(dplyr)

# DATA CLEANING ################################################################

tidy\_data <- as\_tibble(raw\_data) %>%

tidyr::gather(measurement\_point, value) %>%

mutate(time = current\_time,

location = current\_location,

variable = stringr::str\_extract(measurement\_point, "^.+(?=-)"),

measurement\_point = stringr::str\_extract(measurement\_point, "(?<=-).+$"),

measurement\_point = stringr::str\_replace\_all(measurement\_point, point\_names)

) %>%

select(time, location, variable, measurement\_point, value)

# LOAD DATA ####################################################################

connection\_string <- glue::glue("Driver={{PostgreSQL ANSI}};\\

Uid={Sys.getenv('MY\_UID')};\\

Pwd={Sys.getenv('MY\_PWD')};\\

Server=localhost;\\

Port=5432;\\

Database=sensors;")

con <- odbc::dbConnect(odbc::odbc(), .connection\_string = connection\_string, encoding = "utf8")

DBI::dbAppendTable(conn = con,

name = 'sensors\_data',

value = tidy\_data)

odbc::dbDisconnect(con)

unlink(x = "\*.log", force = TRUE)

Now that you have the data into a SQL server, you are at one SQL query away of getting it into R

library(odbc)

library(glue)

connection\_string <- glue::glue(

"Driver={{PostgreSQL ANSI}};\\

Uid={Sys.getenv('MY\_UID')};\\

Pwd={Sys.getenv('MY\_PWD')};\\

Server={Sys.getenv('MY\_REMOTE')};\\

Port=5432;\\

Database=sensors;"

)

con <- dbConnect(

drv = odbc::odbc(),

.connection\_string = connection\_string,

encoding = "utf8"

)

query <- "

SELECT \*

FROM public.sensors\_data

"

raw\_data <- dbGetQuery(

conn = con,

statement = query

)

dbDisconnect(con)

What makes this approach my personal favorite is that it is fairly easy to implement, very flexible and allows you to implement almost any security measure you might need.

**Using the MQTT protocol**

For more resource-constrained applications you can use a similar approach to the previous one, but with a different technology, the MQTT protocol is ideal for low power devices over slow and unreliable connections, moreover, this is also a way to get data from LoRa networks like the publicly available [TTN (The Things Network)](https://www.thethingsnetwork.org/) which provides an API over MQTT, giving you access to a [low-power wide-area network](https://en.wikipedia.org/wiki/LPWAN) for remote sensor deployment over greater distances.

This approach also allows you to write data to a SQL server like in the previous example but, a more interesting application for this would be to get instant status updates to show on dashboards or shiny apps. Consider this simple shiny app that receives a message from a simulated remote sensor and updates the color of a circle accordingly, you can update the temperature value of the simulated sensor by sending an MQTT message to the test topic from a system terminal on any computer with this command mosquitto\_pub -h [test.mosquitto.org](http://test.mosquitto.org) -t simulated\_sensor -q 1 -m 25.9 -r and see how the color changes.

In order for this to work you need to have the mosquitto client installed in your system.

sudo apt install mosquitto-clients

library(shiny)

library(ggplot2)

library(rmqtt) # This package is just a simple wrapper for an MQTT client using

# system() calls, you need to have the mosquitto client installed

# in your system for this to work.

# Initialize Simulated Sensor

mqtt\_topic\_publish(

topic = "simulated\_sensor",

message\_to\_send = "24.9",

host = "[test.mosquitto.org](http://test.mosquitto.org)",

port = 1883,

qos = 1,

retain\_message = TRUE

)

ui <- fluidPage(

titlePanel("Dashboard"),

mainPanel(

h2("Temperature Indicator"),

plotOutput("status\_indicator")

)

)

server <- function(input, output, session) {

get\_color <- function() {

message <- mqtt\_topic\_subscribe(

topic = "simulated\_sensor",

host = "[test.mosquitto.org](http://test.mosquitto.org)",

port = 1883,

qos = 1,

intern = TRUE,

num.messages = 1

)

if (as.numeric(message) > 25) {

color = 'red'

} else {

color = 'green'

}

return(color)

}

pollData <- reactivePoll(100, session,

checkFunc = get\_color,

valueFunc = get\_color

)

output$status\_indicator <- renderPlot({

ggplot(data = data.frame(x = 1, y = 1), aes(x, y)) +

geom\_point(size = 50, color = pollData()) +

scale\_x\_discrete() +

scale\_y\_discrete() +

theme\_void() +

theme(axis.title = element\_blank())

})

}

shinyApp(ui = ui, server = server)

If you are interested in testing this, here is a GitHub Gist with the Arduino Sketch for an ESP8266 board whit a DS18B20 temperature probe sensor that can update the values in the example app.

I know this doesn’t look very interesting right now but consider the implications. You could be getting instant updates in your dashboard from sensors deployed at kilometers downrange and you could even control remote actuators from your shiny app by sending an MQTT message from it with the click of a button!

The downside of this method is that you need to set an MQTT broker (you should not abuse the publicly available ones that are intended for testing), but it can be done for no or little money on a Raspberry Pi or an EC2 AWS instance.

**Disclaimer**

I want to make clear that this is by no means an exhaustive list, there are other alternatives and countless variations of the ones presented here so if none of these fitts your particular project’s needs, just keep looking, very likely someone has come up with a more suitable method.