|  |
| --- |
| Ignition sense using LuvitRED |
|  |





**Franco Arboleda**

**24-Oct-16**

Table of Contents

1 Introduction 3

2 Setting up the GPIO for Ignition detection. 5

2.1 Buildng the ModGate skeleton 6

2.2 Mapping the Modbus registers and Firewalling 9

2.3 Changing the slave ID and register number 11

2.4 Adding the requesters 13

3 Wake up the CloudGate on ignition sense. 15

4 Timed Wake up 19

5 Adding local values (32 bits) 23

# Introduction

This document covers the configuration of ignition sense on the CloudGate using LuvitRED.

Ignition sense is a CloudGate feature currently only available on the CloudGate LTE (CG0114) using firmware version 2.x.x. This feature allows the user to power on the CloudGate by pulling high the ignition sense input line on the power connector.

The pinout of the power connector, seen from the terminal side, is as follows:

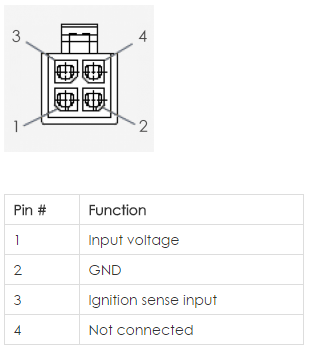
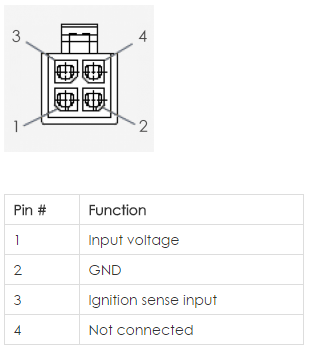


Figure : Power connector Pinout specification.

The electrical specification for the Ignition sense input is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Min** | **Max** | **Unit** |
| Input range | 0.0 | 33 | V |
| Input Low | 0.0 | 2.7 | V |
| Input High | 4.1 | 33 | V |

Table : Ignition sense Pin Electrical specifications.

* The input signal will be read as Low (zero) when the level is lower than 2.7VDC.
* The input signal will be read as High (one) when the level is higher than 4.1VDC.

For this configuration we will to use the "Advanced Editor" of LuvitRED.

**Notes**:

1. This document assumes that the reader is familiar with the LuvitRED and the terms explained on the document called Basics\_of\_LuvitRED\_vXXX.pdf.
2. We are using CloudGate firmware version and LuvitRED version for this configuration.

# Setting up the GPIO for Ignition detection.

For this application we are going to use the GPIO in node from LuvitRED.

The "gpio" in node (See Figure 3) is the one that will be in charge of providing us with the Ignition Sense PIN status.



Figure 2: GPIO in node.

Let's first configure our "gpio" in node:

1. Drag and Drop one "gpio" in node into the LuvitRED editor and double click on it (See Figure 7).

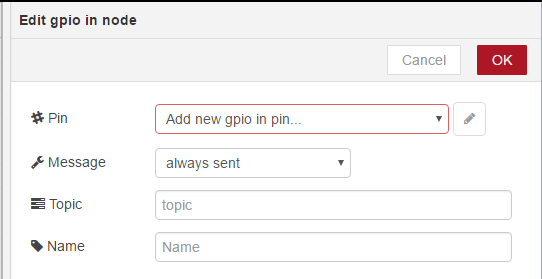


Figure : New gpio in node configuration.

1. The first item that needs to be changed is the “#Pin” configuration. For this, click on the pencil icon to “Add new gpio in pin”:

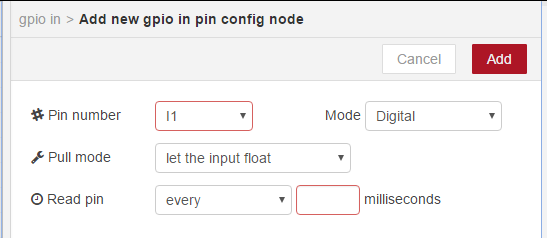


Figure : New gpio in pin default configuration.

We are going to change the standard configuration of the pin as follows:

* #Pin number: IGN
* Mode: Imx

Notice that as soon as the Mode is changed from Digital to Imx, the other options are automatically removed from the configuration page:

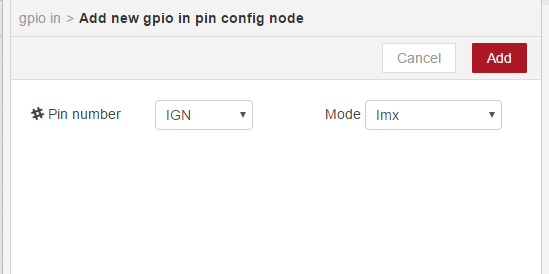


Figure : New gpio in pin configuration for ignition sense.

Click on the “Add” button.

1. After adding the new gpio in pin, change the “Message” from “always sent” to “only sent on change” so that the node only sends a value when the pin changes its state. Change the “Topic” and “Name” of the node to “Ignition” as shown on Figure 9:

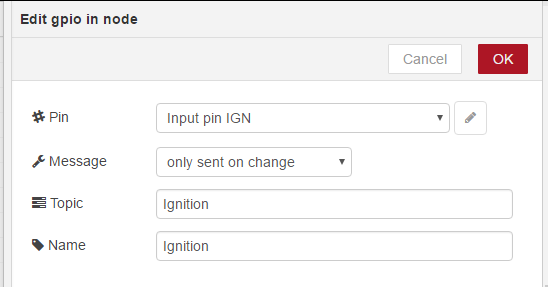


Figure : gpio in node configured.

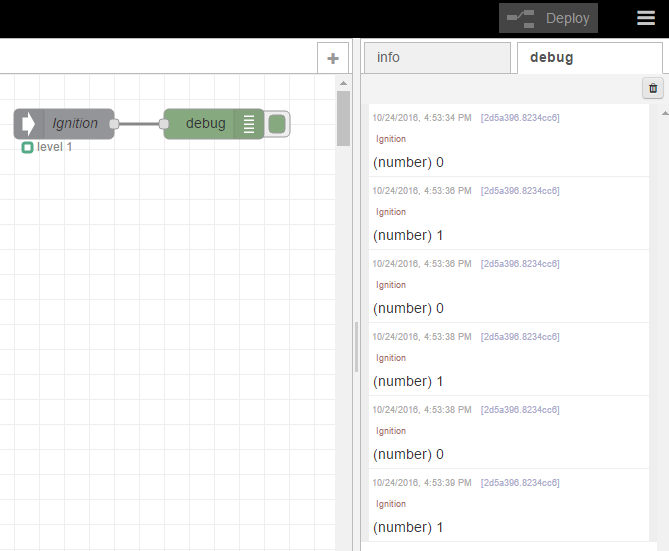
1. Click on "OK" to close the node configuration.
2. Drag and Drop a “debug” node into the LuvitRED editor and connect the two nodes as shown on Figure 8:



Figure : Nodes connected.

1. Click on “Deploy”.

The configured flow will display the status of the ignition pin as per Table 1



# Wake up the CloudGate on ignition sense.

We are currently aggregating data from two different Modbus RTU serial devices, but what happens if we want to contact an extra Modbus device, but this time a Modbus TCP device?

Let say that we just added a third Modbus device to the equation and that our scenario looks like this now:



Figure : ModGate application - Modbus TCP added.

This new device is SlaveID 3 and the register we want to read is register 1 (holding register). Our mapping table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ModGate ID** | **ModGate Register** | **Slave ID** | **Holding Register** | **Contained Value** |
| 1 | 1 | 1 | 1 | 10 |
| 1 | 2 | 1 | 2 | 20 |
| 1 | 3 | 2 | 1 | 2000 |
| 1 | 4 | 3 | 1 | 555 |

Table : New register mapping.

These are the step to follow:

1. Add a new entry on the "ModGate\_IN" node according to the values shown on the last line of the above table (This change will add an extra output on the node):

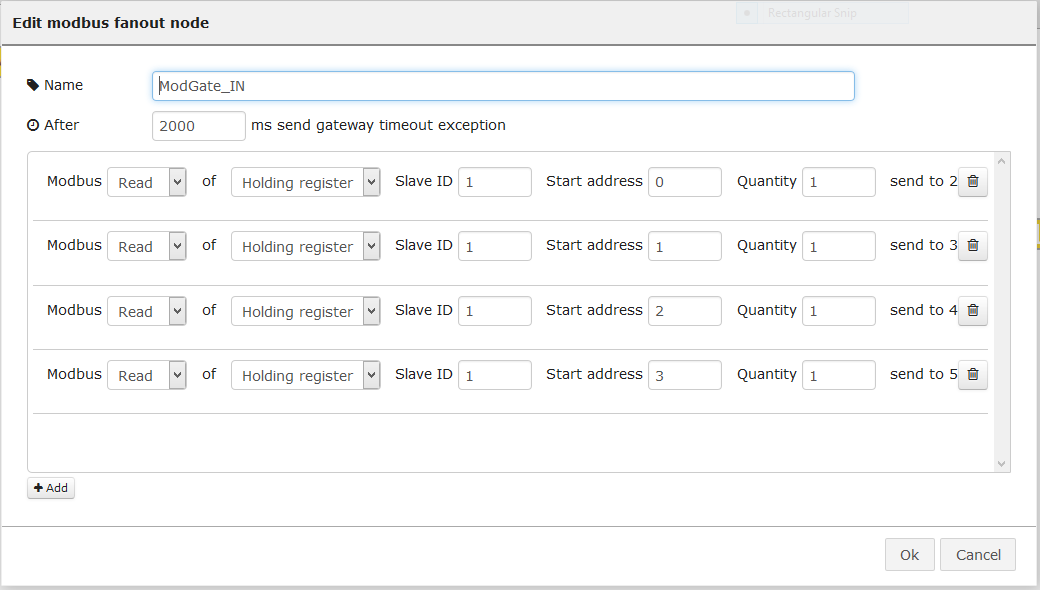


Figure : New entry on ModGate\_IN node.

1. Add a new function and adapt the name and code inside to talk to Slave ID 3 and register 1:

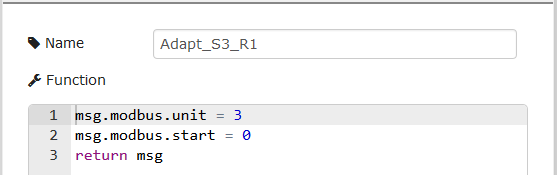


Figure : New function node.

1. Add a new requester and name it, but this time add a **new** requester and configure it for Modbus TCP. You will need to know the IP address and port (502 by default) of the Slave device. In our case the IP is 192.168.1.14:

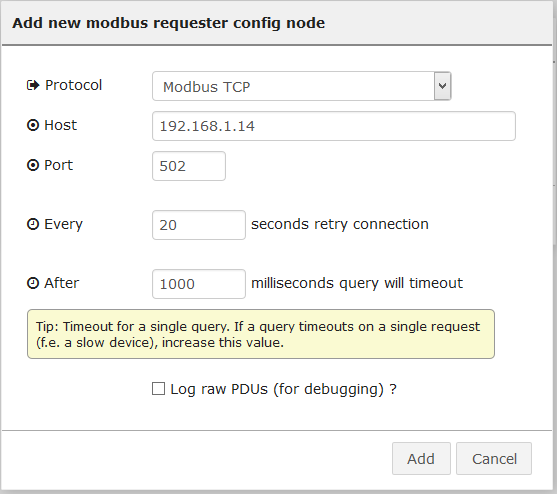


Figure : Modbus TCP requester.

1. Connect fifth output of the ModGate\_IN node to the function node, then function node to the new requester node and the requester node to ModGate\_OUT node and click on "Deploy":

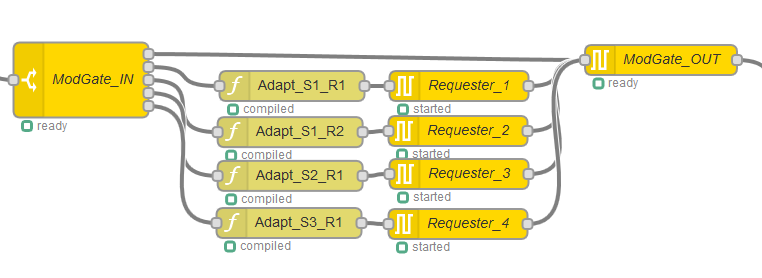


Figure : New nodes connected.

To test this setup we are going to use modpoll.exe again. The command used is:

***modpoll.exe -m tcp -a 1 -r 1 -c 4 -t 4 -l 5000 192.168.1.1***

We are now reading 4 registers instead of 3:

* **-m tcp**: use Modbus TCP
* **-a 1**: Connect to Slave ID 1
* **-r 1**: Start at register 1
* **-c 4**: Read **4 registers**
* **-t 4**: interpret the data as 16 bit holding register values
* **-l 5000**: Poll rate to 5 seconds
* **192.168.1.1**: Local IP of the CloudGate/ModGate

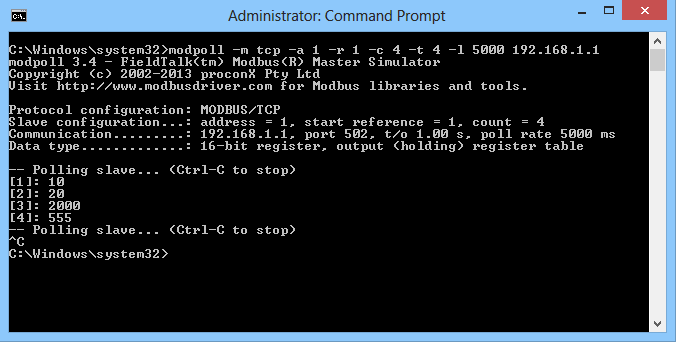


Figure : Modpoll Modbus master.

# Timed Wake up

So far we are communicating with three different Modbus devices using either Modbus RTU or Modbus TCP, but what happens if we want to add a value that is local to the CloudGate, for example the signal strength of the cellular interface?

The process is very similar as the one described above with few differences:

* No need to use a modbus request node since we are not requesting values from an external device.
* We need to use a node to obtain the signal strength: "siglevel" node



Figure : Siglevel node.

* We need to encode the response value properly before sending it to the "modbus fanin" node: "modbus encode" node



Figure : Modbus encode node.

First we need to take a look at the actual value we are going to add into our configuration:

* The signal strength is a negative value.
* The signal strength is a value between 0 and -113 (approximately).

Taking into account the above two items, we should be able to encode this negative value in a 16 bit register, so we can use a Modbus Holding register for this.

Our mapping table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ModGate ID** | **ModGate Register** | **Slave ID** | **Holding Register** | **Contained Value** |
| 1 | 1 | 1 | 1 | 10 |
| 1 | 2 | 1 | 2 | 20 |
| 1 | 3 | 2 | 1 | 2000 |
| 1 | 4 | 3 | 1 | 555 |
| 1 | 5 | - | - | Signal Strength |

Table : New register mapping.

These are the step to follow:

1. Add a new entry on the "ModGate\_IN" node according to the values shown on the last line of the above table (This change will add an extra output on the node):

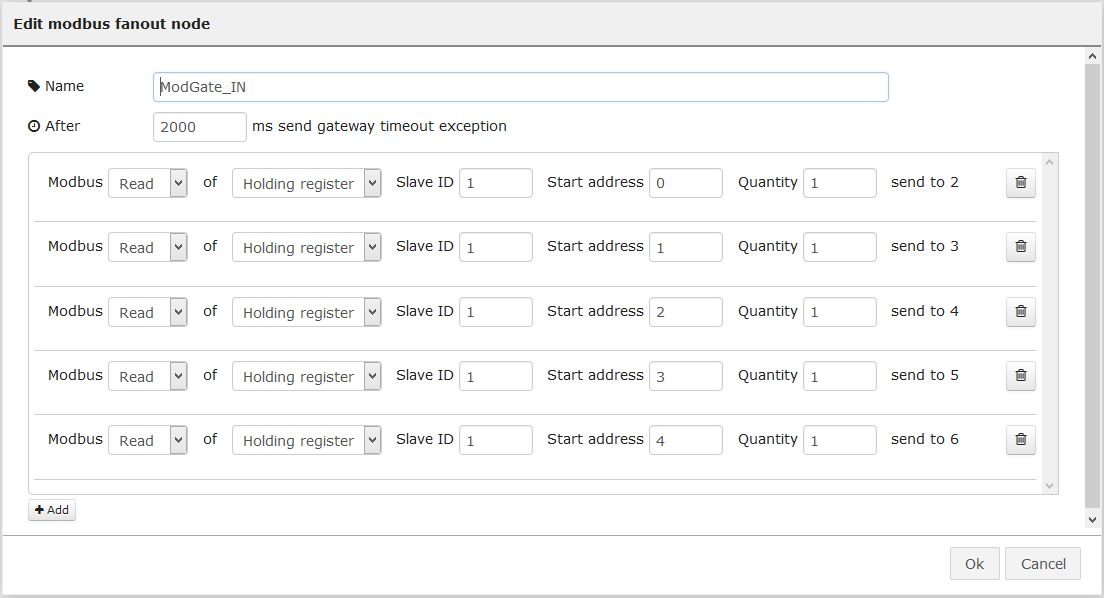


Figure : New entry on ModGate\_IN node.

1. Add a "siglevel" node and change its name:

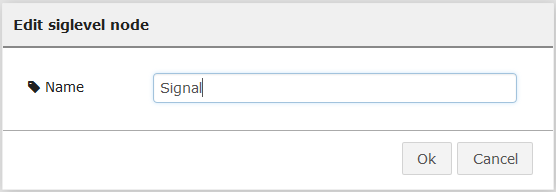


Figure : Siglevel node configuration.

1. Add a "change" node and configure it as shown on Figure 37:

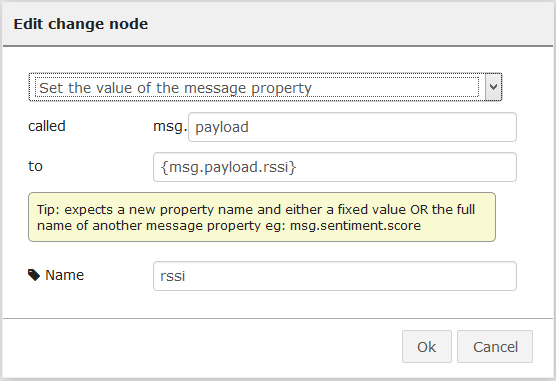


Figure : Change node configuration.

* Since the siglevel node includes not only the rssi, but the ecio, we need to change the payload to only contain the rssi.
* We are setting the ***msg.payload*** to ***msg.payload.rssi***
* We are changing the name of the node.

1. Add a "modbus encode" node and configure it as shown on Figure 38:

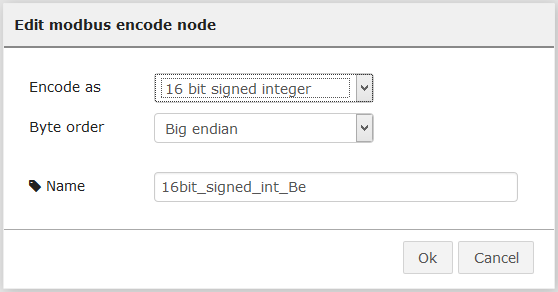


Figure : Modbus encode node configuration.

* We are Encoding the value as a 16 bit **signed** integer according our evaluation made earlier over the value.
* We are keeping the Big endian Byte order.
* We are changing the name of the node.

1. Connect the nodes to the ModGate\_IN and ModGate\_OUT nodes as shown below and then click on "Deploy":

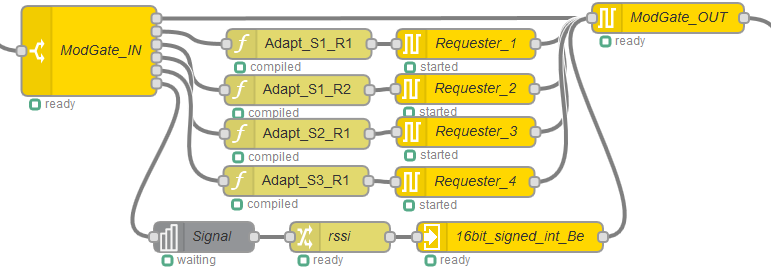


Figure : New nodes connected.

To test this setup we are going to use modpoll.exe again. The command used is:

***modpoll.exe -m tcp -a 1 -r 1 -c 5 -t 4 -l 5000 192.168.1.1***

We are now reading 5 registers instead of 4:

* **-m tcp**: use Modbus TCP
* **-a 1**: Connect to Slave ID 1
* **-r 1**: Start at register 1
* **-c 5**: Read **5 registers**
* **-t 4**: interpret the data as 16 bit holding register values
* **-l 5000**: Poll rate to 5 seconds
* **192.168.1.1**: Local IP of the CloudGate/ModGate

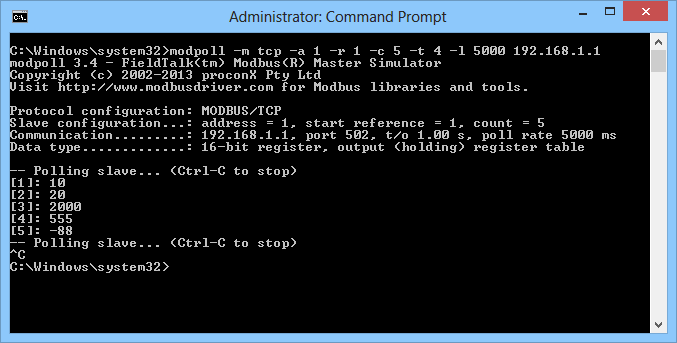


Figure : Modpoll Modbus master.

# Adding local values (32 bits)

On section 4 we explained how to add a 16 bit value, more specifically a 16 bit signed value. In this case we want to add a value that is a signed floating point value such as the latitude, longitude and altitude.

The process is almost the same as the one explained on the previous section with few differences:

* Instead of a siglevel node, we need a "GPS" node:



Figure : GPS node.

* We will need to store the GPS data in a global variable in order to retrieve it upon request. There are few ways of doing this, but we are going to use a function node.
* To retrieve the data we need to use a "global retrieve" node:



Figure : Global retrieve node.

First we need to take a look at the actual values we are going to add into our configuration:

* The values might be negative depending on the geographical location.
* The values are floating point numbers.

Taking into account the above two items, we should be able to encode this negative value as a 32 bit value using IEEE 754 encoding, so we are going to need two Modbus Holding registers per value.

Our mapping table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ModGate ID** | **ModGate Registers** | **Slave ID** | **Holding Register** | **Contained Value** |
| 1 | 1 | 1 | 1 | 10 |
| 1 | 2 | 1 | 2 | 20 |
| 1 | 3 | 2 | 1 | 2000 |
| 1 | 4 | 3 | 1 | 555 |
| 1 | 5 | - | - | Signal Strength |
| 1 | 6 & 7 | - | - | Latitude |
| 1 | 8 & 9 | - | - | Longitude |
| 1 | 10 & 11 | - | - | Altitude |

Table : New register mapping.

These are the step to follow:

1. First we need to store the GPS data in a global variable in order to make it available upon request:
   1. Add a new GPS node and configure it as follows:

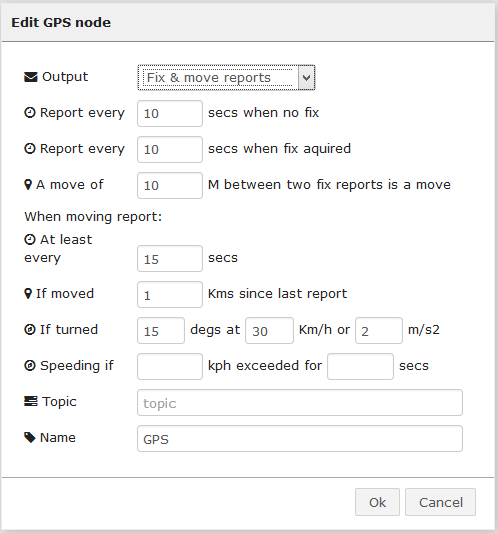


Figure : GPS node configuration.

* 1. Add a new function node and configure it as follows:

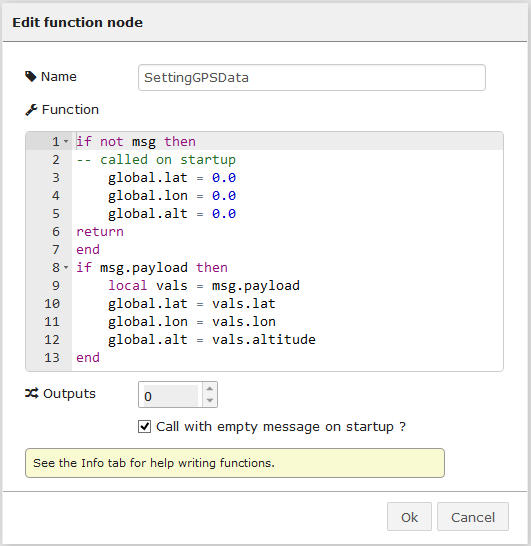


Figure : Function node configuration.

**NOTE1:** The global variables are called lat, lon and alt (set to ZERO at startup)

**NOTE2:** Check the Call with empty message on startup checkbox!

* 1. Connect both nodes together, leave them separated from the main ModGate skelethon and click on "Deploy":



Figure : GPS data extraction.

1. Add three new entries on the "ModGate\_IN" node according to the values shown on the last three lines of Table 7 (This change will add three extra outputs on the node):

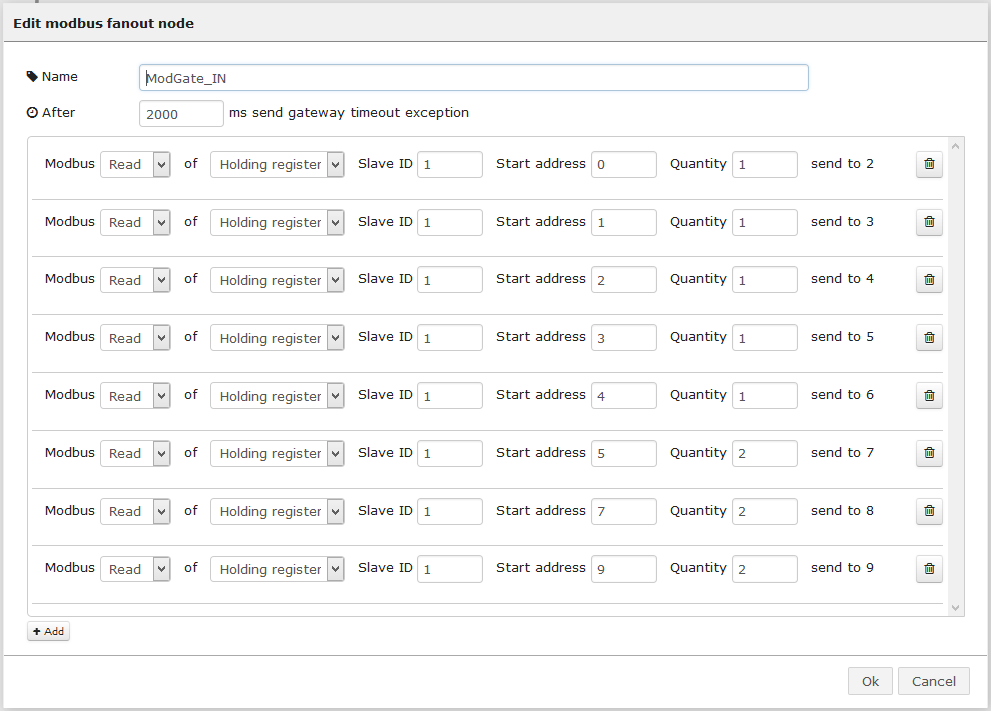


Figure : New entries on ModGate\_IN node.

**NOTE:** Be aware that the three new entries need to have a Quantity of 2 registers and therefore the Start address needs to reflect this jump. The start addresses are 5, 7, and 9 in this case each with a Quantity of 2 instead of 1.

1. Add three "global retrieve" nodes and configure them as shown on Figure 47, Figure 48 and Figure 49:

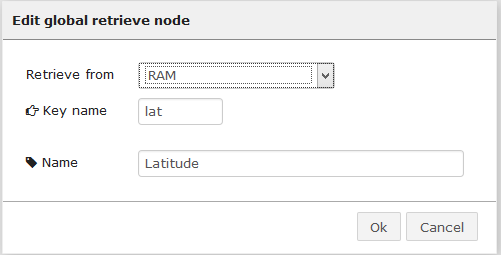


Figure : Global retrieve node configuration (latitude).

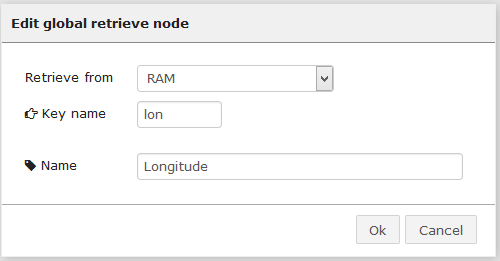


Figure : Global retrieve node configuration (longitude).

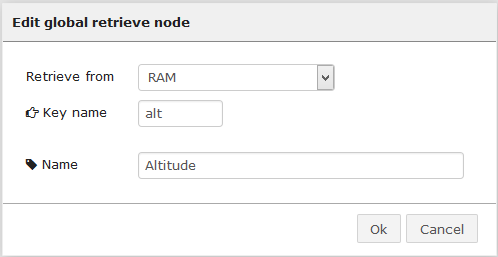


Figure : Global retrieve node configuration (altitude).

1. Add a "modbus encode" node and configure it as shown on Figure 50:

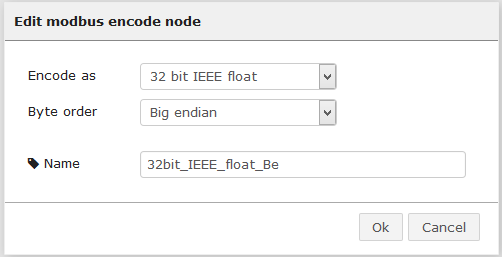


Figure : Modbus encode node configuration.

* We are Encoding the value as a 32 bit **IEEE float** according our evaluation made earlier over the value.
* We are keeping the Big endian Byte order.
* We are changing the name of the node.

1. Connect the nodes to the ModGate\_IN and ModGate\_OUT nodes as shown below and then click on "Deploy":

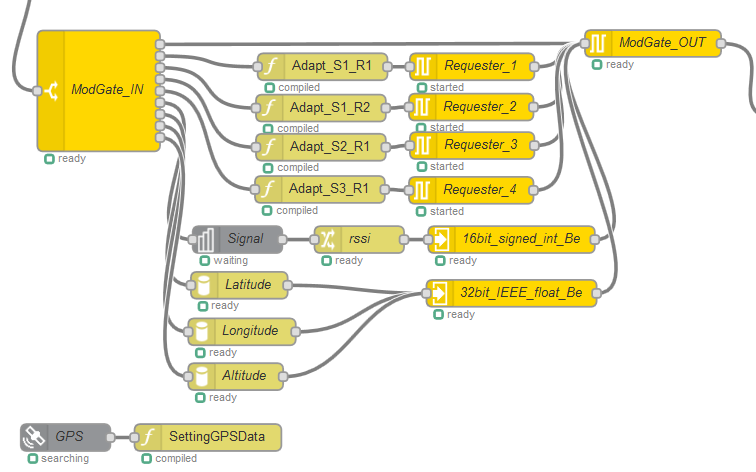


Figure : New nodes connected.

To test this setup we are going to use modpoll.exe again. The command used is:

***modpoll.exe -m tcp -a 1 -r 6 -c 3 -t 4:float -f -l 5000 192.168.1.1***

We are now reading only the new registers instead of all. This is because the modpoll.exe tool cannot retrieve values of different types (16 bit integers vs 32 bit floats) at the same time:

* **-m tcp**: use Modbus TCP
* **-a 1**: Connect to Slave ID 1
* **-r 6**: Start at register 6
* **-c 3**: Read **3 registers**
* **-t 4:float**: interpret the data as 32 bit float data type in a holding register table
* **-f:** Slave operates on big-endian 32-bit floats
* **-l 5000**: Poll rate to 5 seconds
* **192.168.1.1**: Local IP of the CloudGate/ModGate

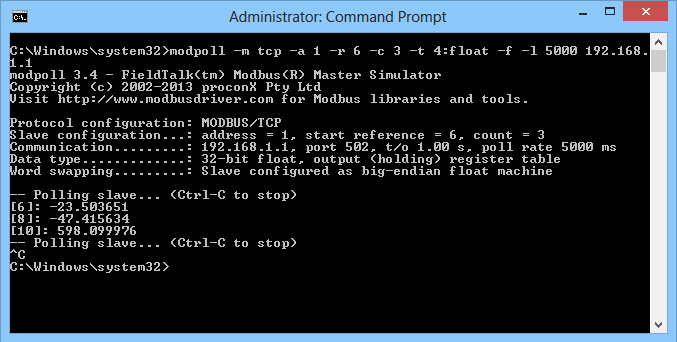


Figure : Modpoll Modbus master (32 bit float).

We should still be able to get the values we configured on the previous sections by using the previous modpoll.exe command:

***modpoll.exe -m tcp -a 1 -r 1 -c 5 -t 4 -l 5000 192.168.1.1***

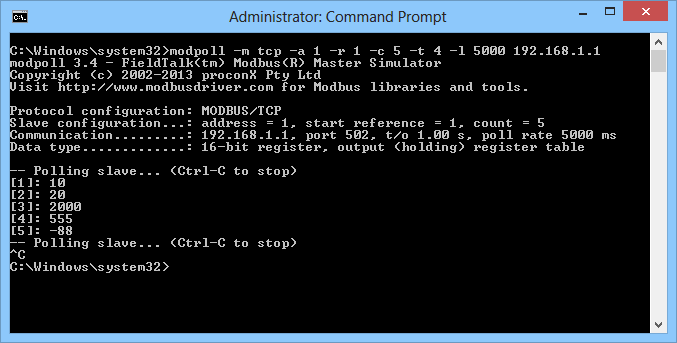


Figure : Modpoll Modbus master (16 bit integer).

If we execute the above command increasing the amount of registers to read from 5 to 11, we should get all values, but the last six values will be the IEEE encoded values:

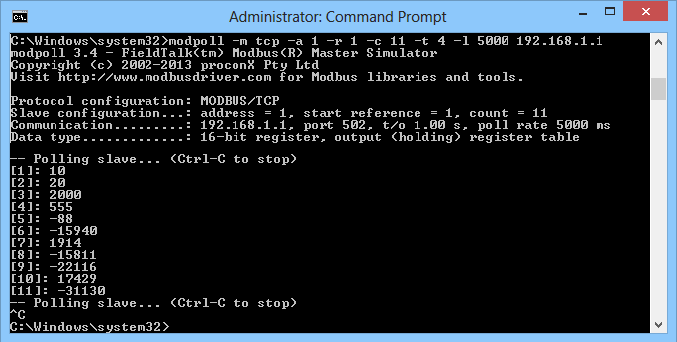


Figure : Modpoll Modbus master (all values as 16 bit integers).