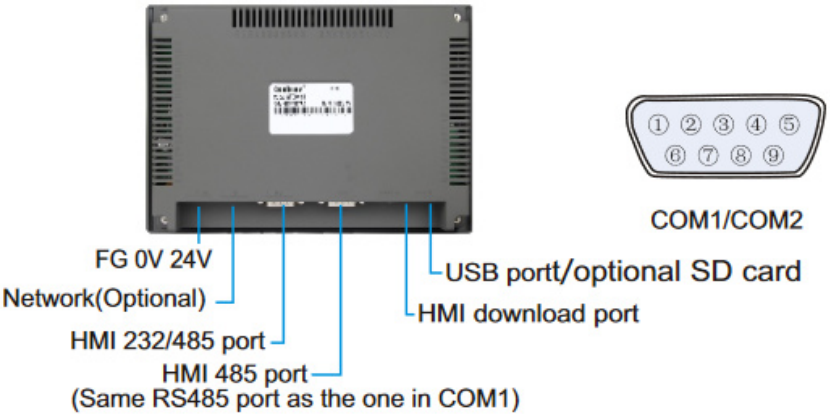
COOLMAY HMI MT6070HA / ARDUINO CLONE / ARDUINO UNO and MODBUS RTU

0-Connecting the devices:

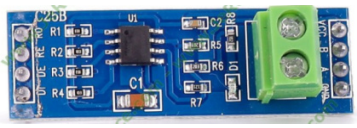
You need:

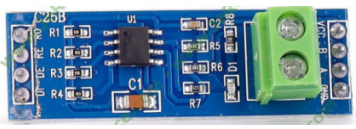
- -a COOLMAY MT6070H HMI: the master
- -2xMAX485 shields
- -an arduino uno board: the slave 3
- -an arduino clone board: the slave 1
- -an USBasp cable and 2xUSB cables

MT6070H



Pin No	Signal	Description
Definition of COM1 RS232 port		
2	RXD	Receive
3	TXD	Send
5	GND	Ground
Definition of COM1/COM2 RS485 port		
1	A	485+
6	B	485-



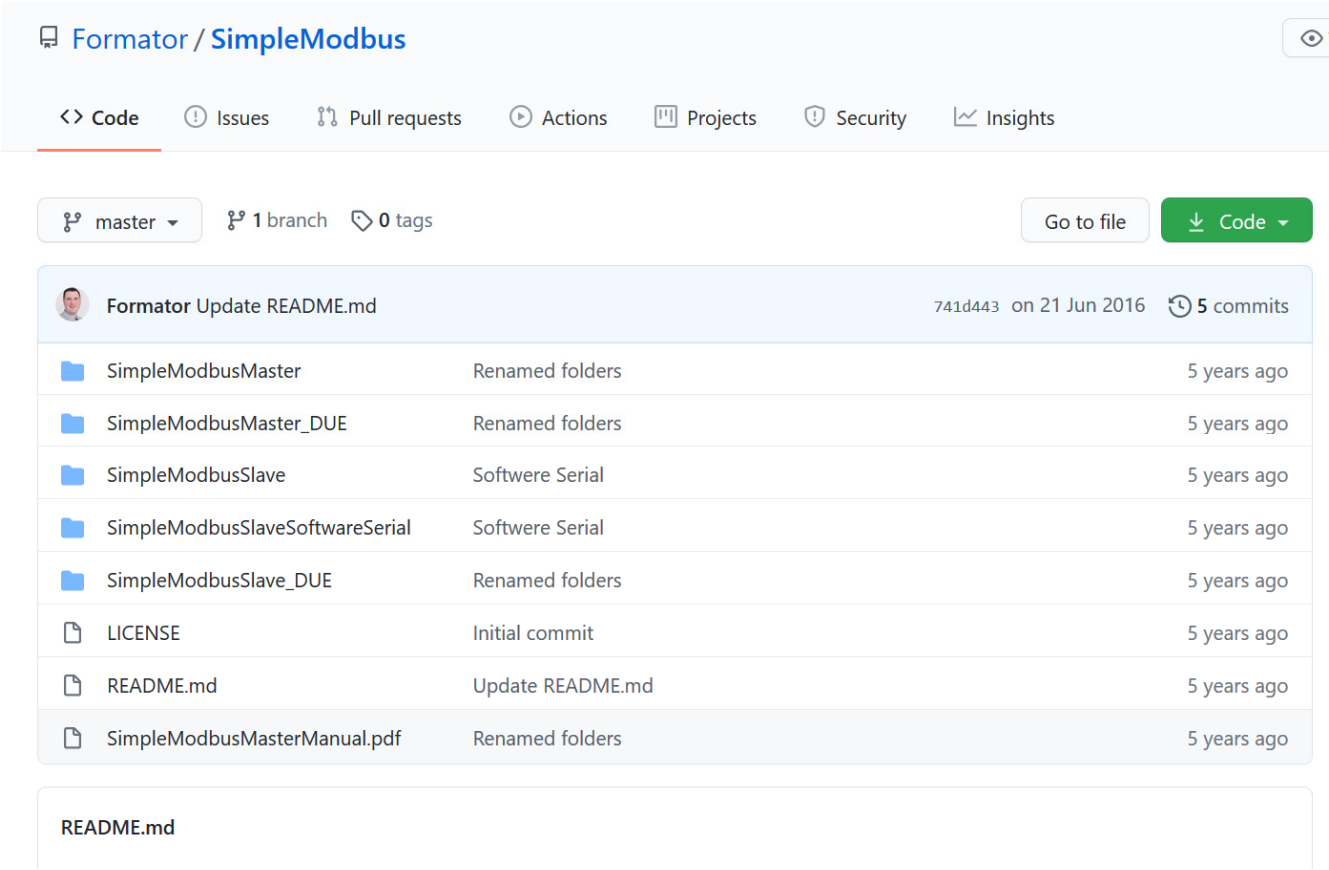


PART 1: HMI the MASTER, Clone the SLAVE1, UNO the SLAVE3

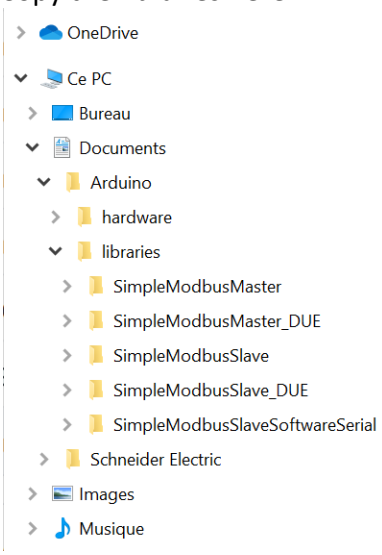
1-Arduino and softwares needed:

1-1- Install Arduino 1.8.13

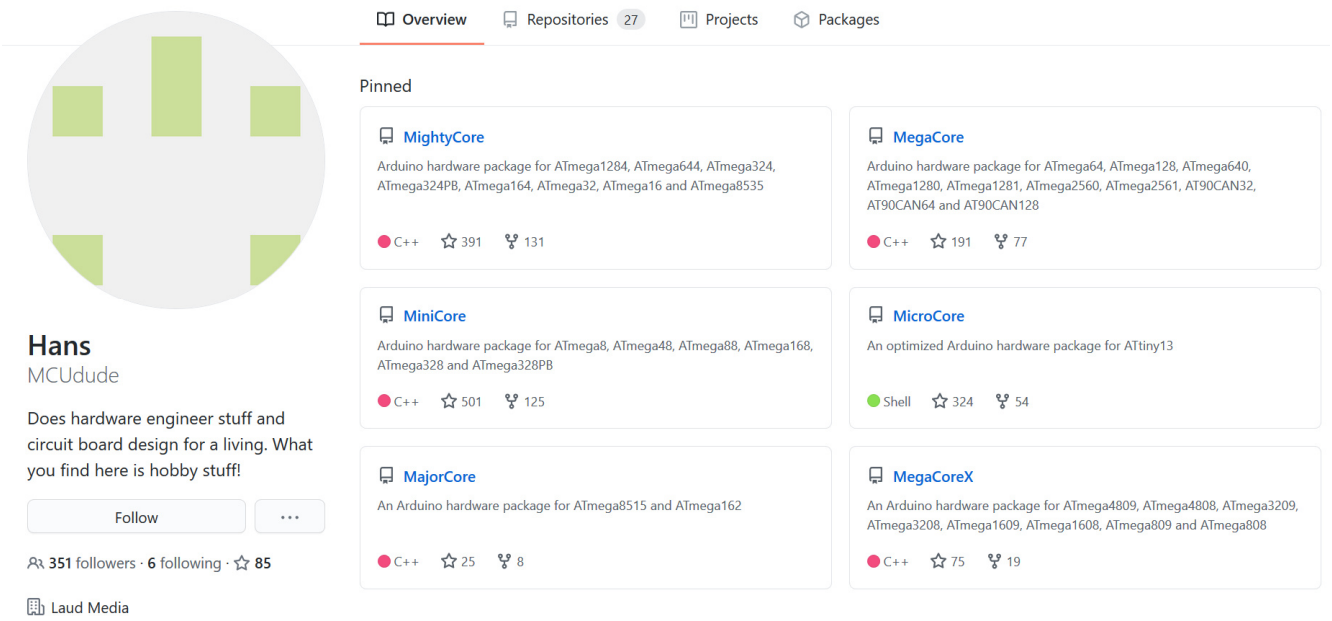
1-2- Go to https://github.com/Formator/SimpleModbus for the modbus rtu lib downloading:



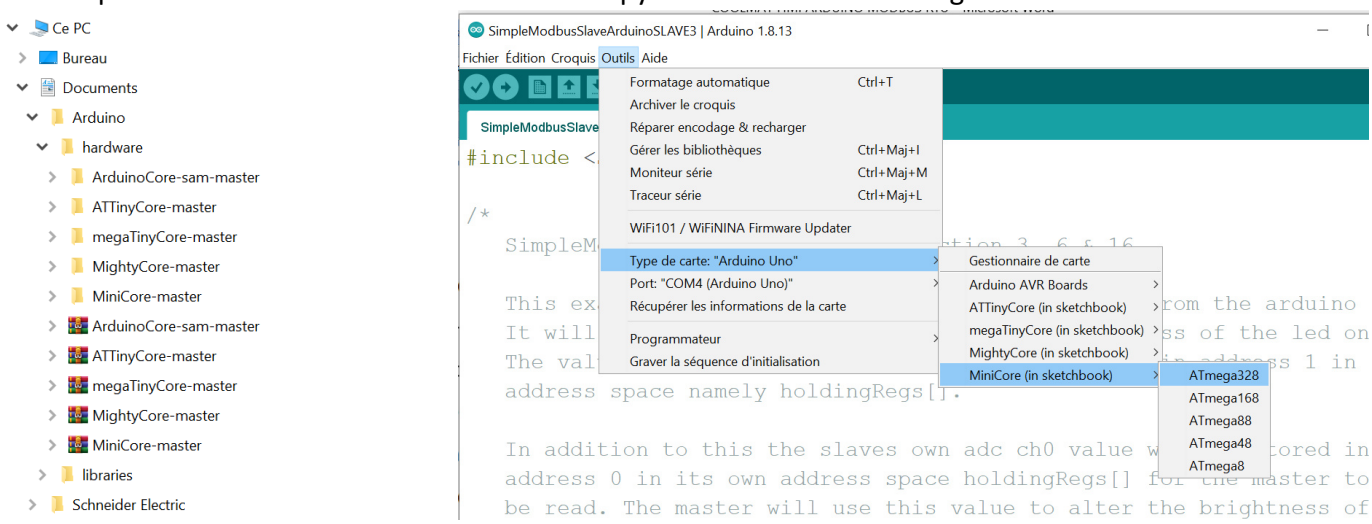
Copy the libraries here:



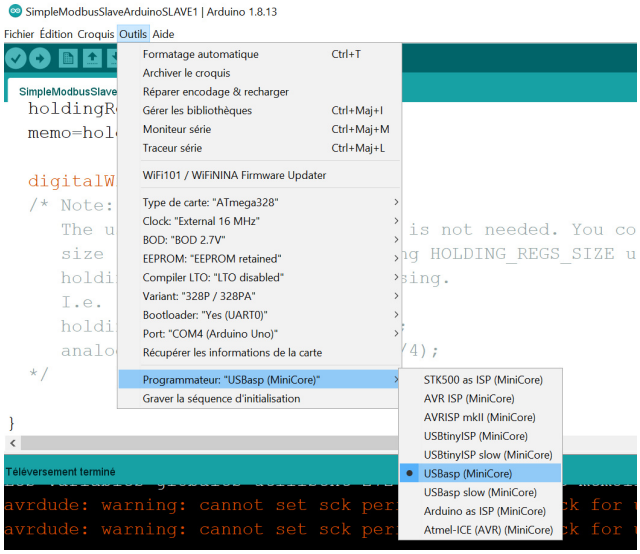
1-3- Go to https://github.com/MCUdude and download the 5 packs of boards: you need here the Minicore because I use an arduino clone board made of an Atmega328P

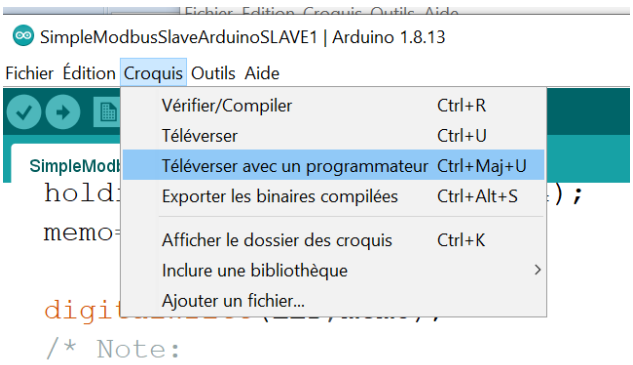


Put the pack in a folder called “hardware” and copy it here and check if it’s good:



And the good progammer here and use it like this:

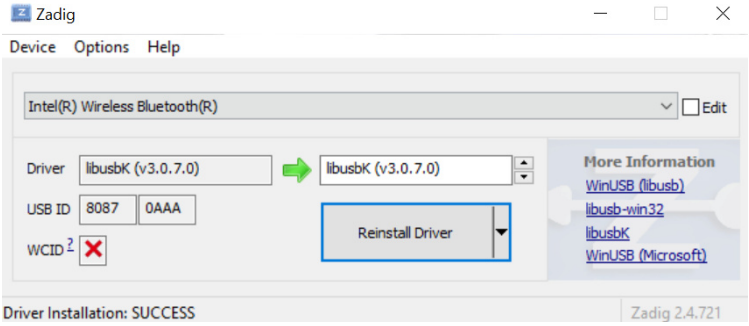




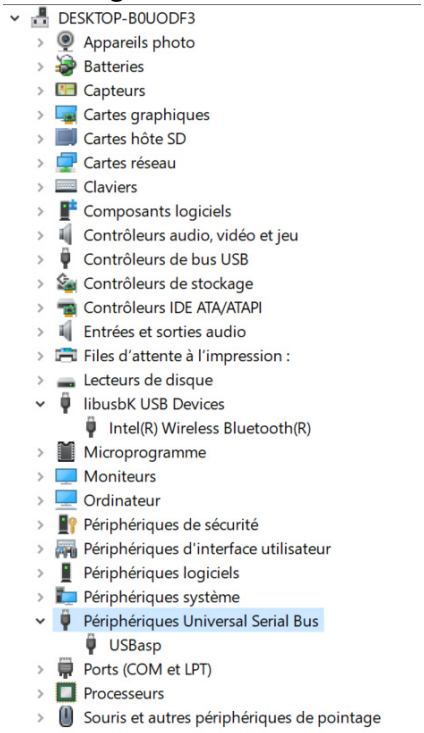
1-4- USBasp needed:

When you connect for the first time the driver is not correctly installed

So you need Zadig to reinstall the driver

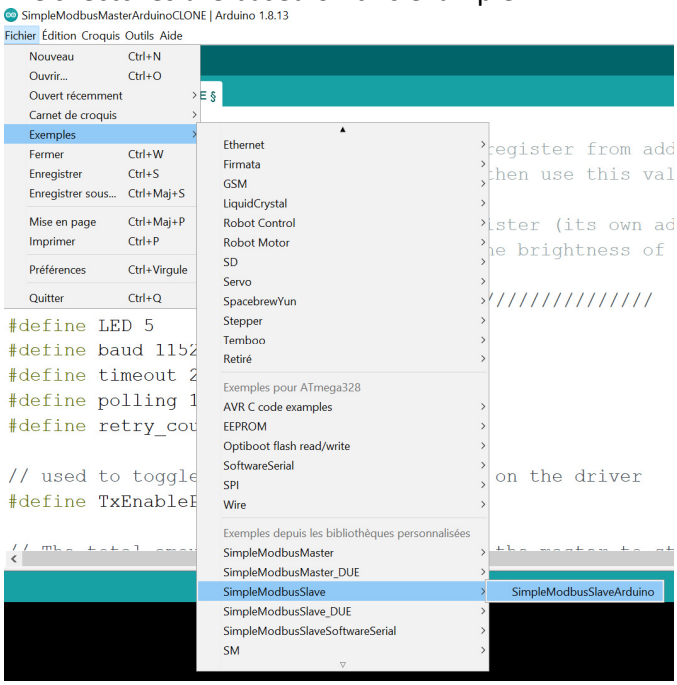


Now it's good:



1-5- The programs for the both slaves:

The skectches are based on this example:



The arduino boards are slaves and there will be only one master: the HMI. This HMI will control leds on the both slave whenever you want.
I used the serial pin 0 and pin 1 (RX TX) and the pin 2 to control the link.
Serial settings: SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits and 115200 bauds according to those of the HMI.
The HMI lights ON/OFF the pin 5 of the slave 1 via holdingRegs[1] while the slave1 send the value 32765 via holdingRegs[0] to the HMI. A button on A0 of the arduino clone board light ON/OFF a lamp of the HMI via holdingRegs[2].
At the same time the HMI lights ON/OFF the pin 5 of the slave 3 via holdingRegs[1] while the slave 3 send the value 6666 via holdingRegs[0] to the HMI. A button on A0 of the arduino UNO board light ON/OFF a lamp of the HMI via holdingRegs[2].

The clone board: slave ID 1

```
#include <SimpleModbusSlave.h>

/*
   SimpleModbusSlaveV10 supports function 3, 6 & 16.

   This example code will receive the adc ch0 value from the arduino master.
   It will then use this value to adjust the brightness of the led on pin 9.
   The value received from the master will be stored in address 1 in its own
   address space namely holdingRegs[].

   In addition to this the slaves own adc ch0 value will be stored in
   address 0 in its own address space holdingRegs[] for the master to
   be read. The master will use this value to alter the brightness of its
   own led connected to pin 9.

  The modbus_update() method updates the holdingRegs register array and checks
  communication.

  Note:
  The Arduino serial ring buffer is 64 bytes or 32 registers.
  Most of the time you will connect the arduino to a master via serial
  using a MAX485 or similar.

  In a function 3 request the master will attempt to read from your
  slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
  and two BYTES CRC the master can only request 58 bytes or 29 registers.

  In a function 16 request the master will attempt to write to your
  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
```

```
   NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
   54 bytes or 27 registers.

   Using a USB to Serial converter the maximum bytes you can send is
   limited to its internal buffer which differs between manufactures.
*/

#define  LED 5

// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  ADC_VAL,  //adress 0
  PWM_VAL,  //ADDRESS 1
  BUTTON, // address 2
  HOLDING_REGS_SIZE // leave this one
  // total number of registers for function 3 and 16 share the same register array
  // i.e. the same address space
};
unsigned int holdingRegs[HOLDING_REGS_SIZE]; // function 3 and 16 register array
////////////////////////////////////////////////////////////
int memo;
void setup()
{
  /* parameters(HardwareSerial* SerialPort,
                long baudrate,
    unsigned char byteFormat,
                unsigned char ID,
                unsigned char transmit enable pin,
                unsigned int holding registers size,
                unsigned int* holding register array)
  */
 /* Valid modbus byte formats are:
    SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits
    SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit
    SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit

    You can obviously use SERIAL_8N1 but this does not adhere to the
    Modbus specifications. That said, I have tested the SERIAL_8N1 option
    on various commercial masters and slaves that were suppose to adhere
    to this specification and was always able to communicate... Go figure.

    These byte formats are already defined in the Arduino global name space.
 */
```

```
  modbus_configure(&Serial, 115200, SERIAL_8N2, 1, 2, HOLDING_REGS_SIZE, holdingRegs);

  // modbus_update_comms(baud, byteFormat, id) is not needed but allows for easy update
  //of the port variables and slave id dynamically in any function.
  modbus_update_comms(115200, SERIAL_8N2, 1);

  pinMode(LED, OUTPUT);
   pinMode(14, INPUT);
}

void loop()
{
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.

  modbus_update();

  //holdingRegs[ADC_VAL] = analogRead(A0); // update data to be read by the master
                                            //to adjust the PWM
  holdingRegs[0] = 32565;
  //analogWrite(LED, holdingRegs[PWM_VAL]>>2); // constrain adc value from the arduino
                                                //master to 255
  holdingRegs[2]=digitalRead(14);
  memo=holdingRegs[1];

  digitalWrite(LED,memo);
  /* Note:
     The use of the enum instruction is not needed. You could set a maximum allowable
     size for holdinRegs[] by defining HOLDING_REGS_SIZE using a constant and then access
     holdingRegs[] by "Index" addressing.
     I.e.
     holdingRegs[0] = analogRead(A0);
     analogWrite(LED, holdingRegs[1]/4);
  */

}
```

The arduino board: slave ID 3

```
#include <SimpleModbusSlave.h>

/*
   SimpleModbusSlaveV10 supports function 3, 6 & 16.

   This example code will receive the adc ch0 value from the arduino master.
   It will then use this value to adjust the brightness of the led on pin 9.
   The value received from the master will be stored in address 1 in its own
   address space namely holdingRegs[].

   In addition to this the slaves own adc ch0 value will be stored in
   address 0 in its own address space holdingRegs[] for the master to
   be read. The master will use this value to alter the brightness of its
   own led connected to pin 9.
```

```
  The modbus_update() method updates the holdingRegs register array and checks
  communication.

  Note:
  The Arduino serial ring buffer is 64 bytes or 32 registers.
  Most of the time you will connect the arduino to a master via serial
  using a MAX485 or similar.

  In a function 3 request the master will attempt to read from your
  slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
  and two BYTES CRC the master can only request 58 bytes or 29 registers.

  In a function 16 request the master will attempt to write to your
  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
  54 bytes or 27 registers.

  Using a USB to Serial converter the maximum bytes you can send is
  limited to its internal buffer which differs between manufactures.
*/

#define  LED 13

// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  ADC_VAL,  //adress 0
  PWM_VAL,  //ADDRESS 1
  BUTTON, // address 2
  HOLDING_REGS_SIZE // leave this one
  // total number of registers for function 3 and 16 share the same register array
  // i.e. the same address space
};

unsigned int holdingRegs[HOLDING_REGS_SIZE]; // function 3 and 16 register array
////////////////////////////////////////////////////////////
int memo;
void setup()
{
  /* parameters(HardwareSerial* SerialPort,
                long baudrate,
    unsigned char byteFormat,
                unsigned char ID,
                unsigned char transmit enable pin,
                unsigned int holding registers size,
                unsigned int* holding register array)
  */
```

```
  /* Valid modbus byte formats are:
     SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits
     SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit
     SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit

     You can obviously use SERIAL_8N1 but this does not adhere to the
     Modbus specifications. That said, I have tested the SERIAL_8N1 option
     on various commercial masters and slaves that were suppose to adhere
     to this specification and was always able to communicate... Go figure.

     These byte formats are already defined in the Arduino global name space.
  */
  modbus_configure(&Serial, 115200, SERIAL_8N2, 3, 2, HOLDING_REGS_SIZE, holdingRegs);

  // modbus_update_comms(baud, byteFormat, id) is not needed but allows for easy update
  //of the
  // port variables and slave id dynamically in any function.
  modbus_update_comms(115200, SERIAL_8N2, 3);

  pinMode(LED, OUTPUT);
   pinMode(14, INPUT);
}
void loop()
{
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.

  modbus_update();

  //holdingRegs[ADC_VAL] = analogRead(A0); // update data to be read by the master to
                                            //adjust the PWM
  holdingRegs[0] = 6666;
  //analogWrite(LED, holdingRegs[PWM_VAL]>>2); // constrain adc value from the arduino
  //master to 255
  holdingRegs[2]=digitalRead(14);
  memo=holdingRegs[1];
  digitalWrite(LED,memo);
  /* Note:
     The use of the enum instruction is not needed. You could set a maximum allowable
     size for holdinRegs[] by defining HOLDING_REGS_SIZE using a constant and then access
     holdingRegs[] by "Index" addressing.
     I.e.
     holdingRegs[0] = analogRead(A0);
     analogWrite(LED, holdingRegs[1]/4);
  */

}
```

2-COOLMAY HMI and softwares needed:

Download here http://www.coolmay.com/Download-159-36-41.html

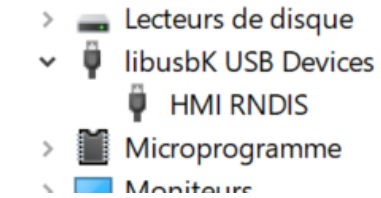
CoolMayHMI V5.906EN	2019-12-31	Down
HW-40B Editer setup	2018-04-08	Down

2-1-Install COOLMAY HMI soft and the very special driver :

Install COOLMAY HMI and link the diplay like this :

- Special USB wire to the USB port of your PC
- The ethernet wire directly on the ethernet shield of your Arduino.
- Power on the display

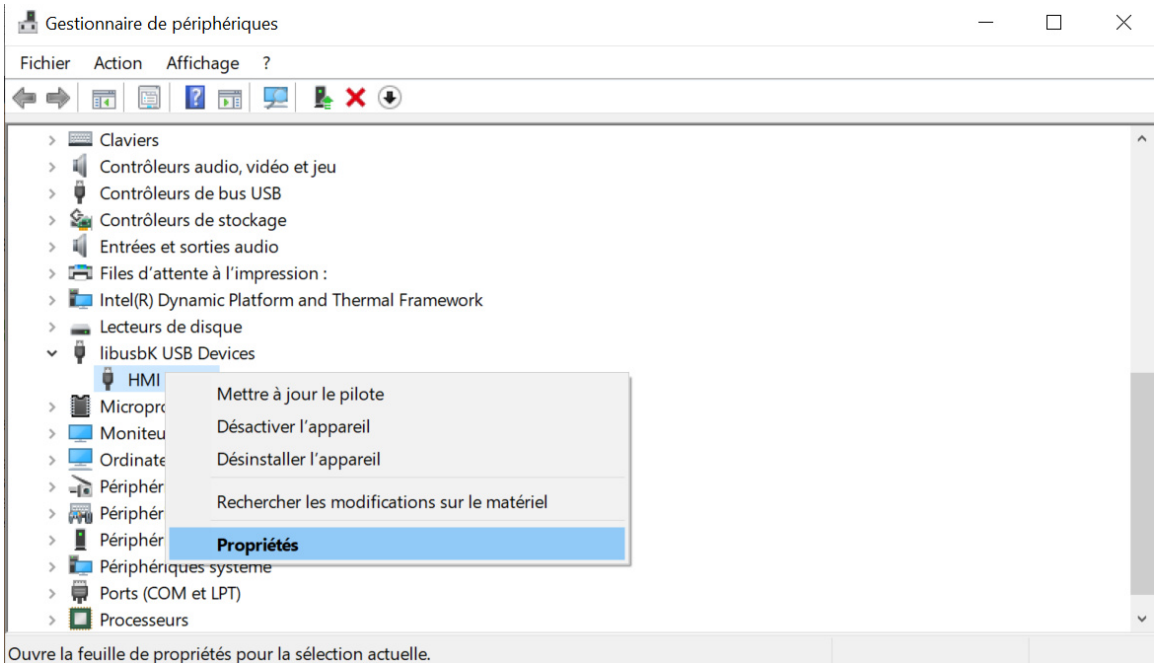
Have a look on the peripheral devices on your PC :



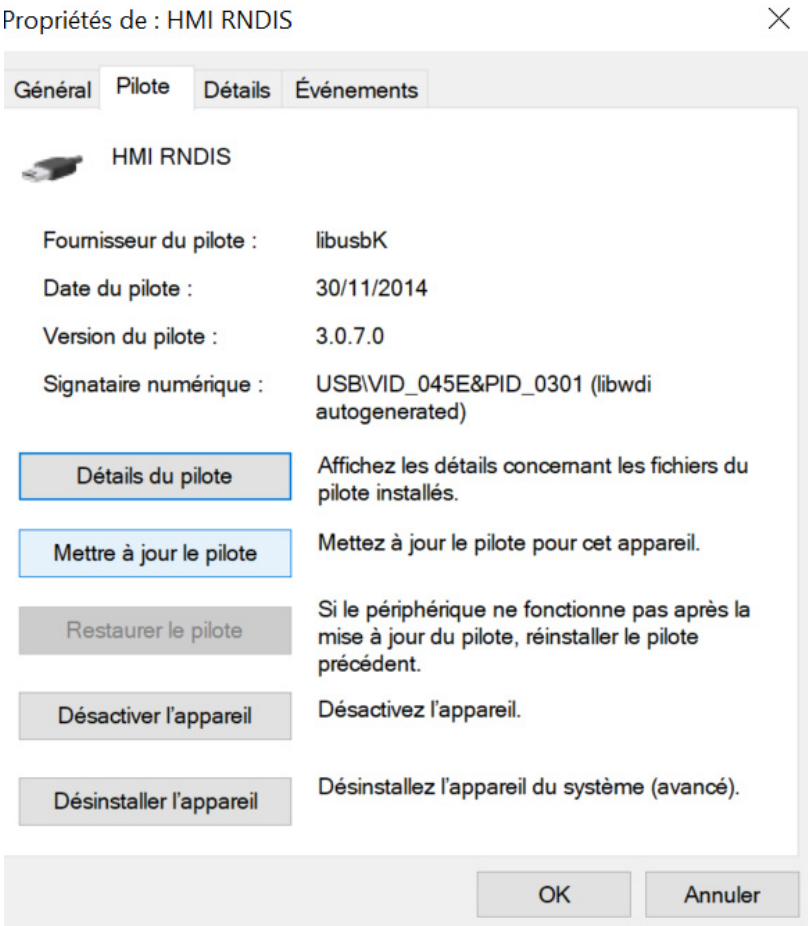
The device RNDIS appears as USB device, **if not libusk use Zadig after restart in non signature driver mode.** In the network list, there is only other networks.

It appears as HMI RNDIS, the wifi embedded functions are disabled. You need to install the driver in order to activate the wifi embeded on your display.

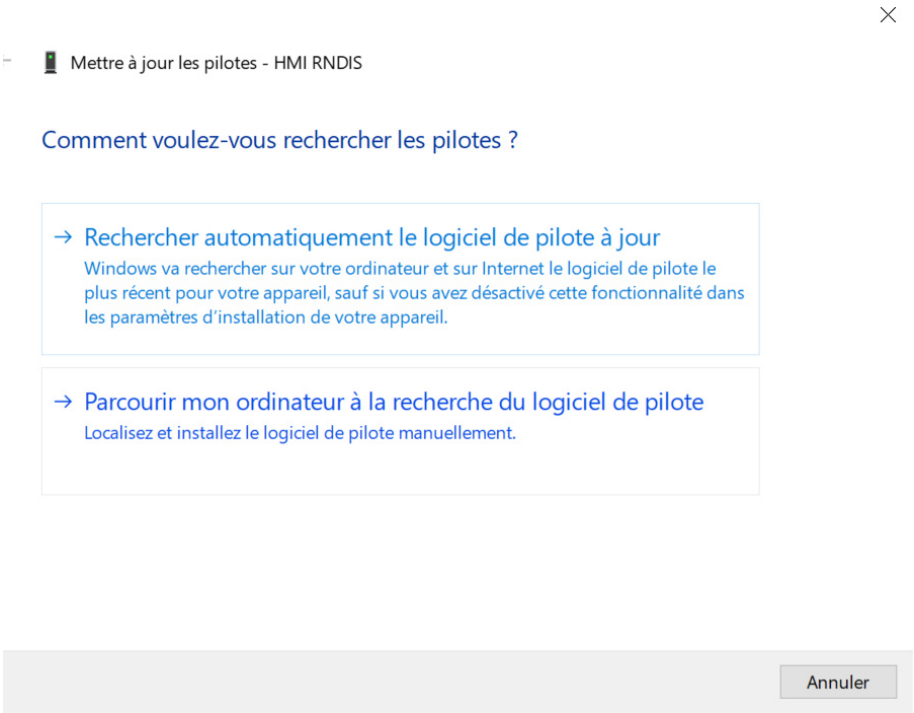
Right click, Properties :



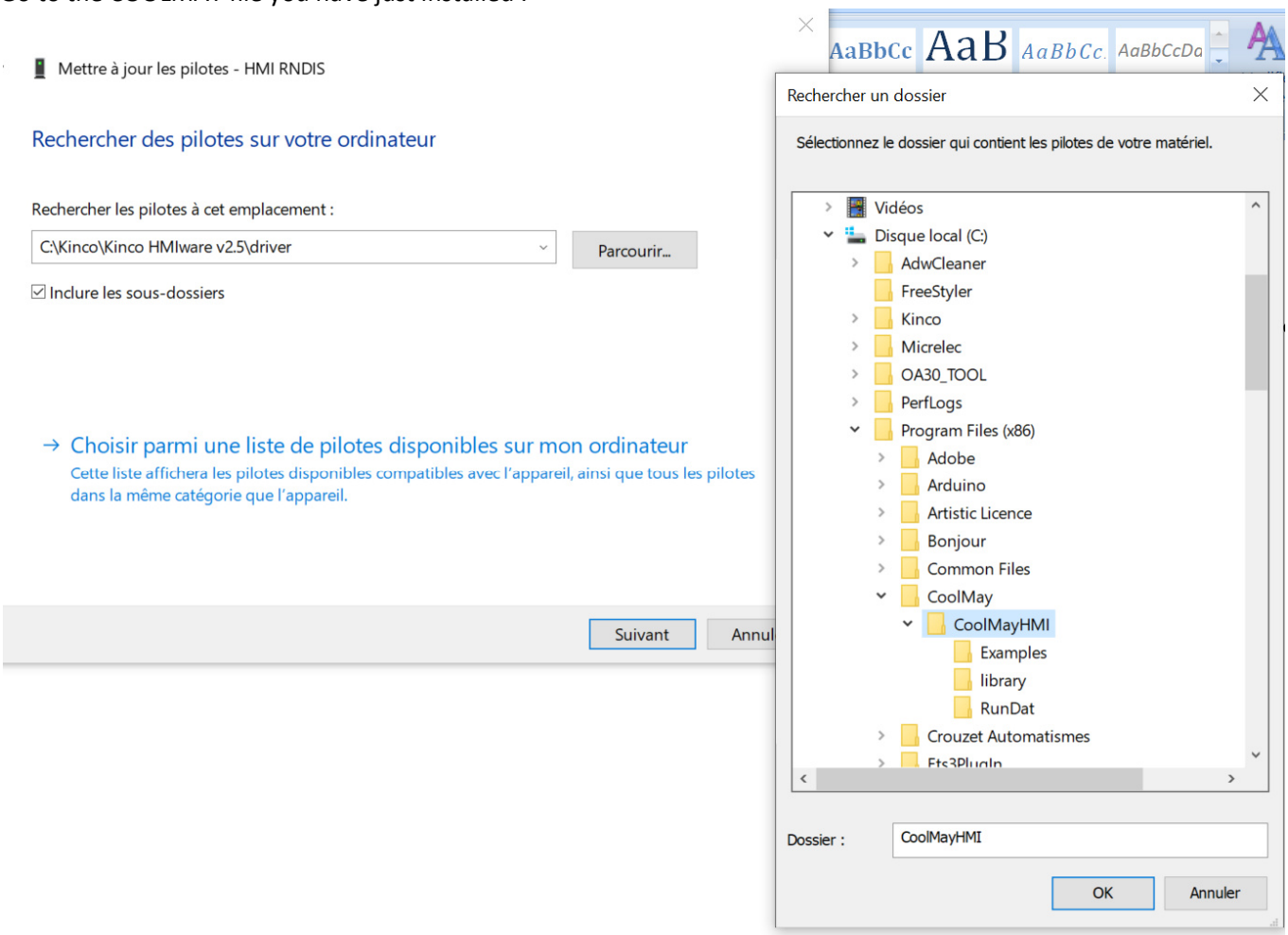
Update the driver :

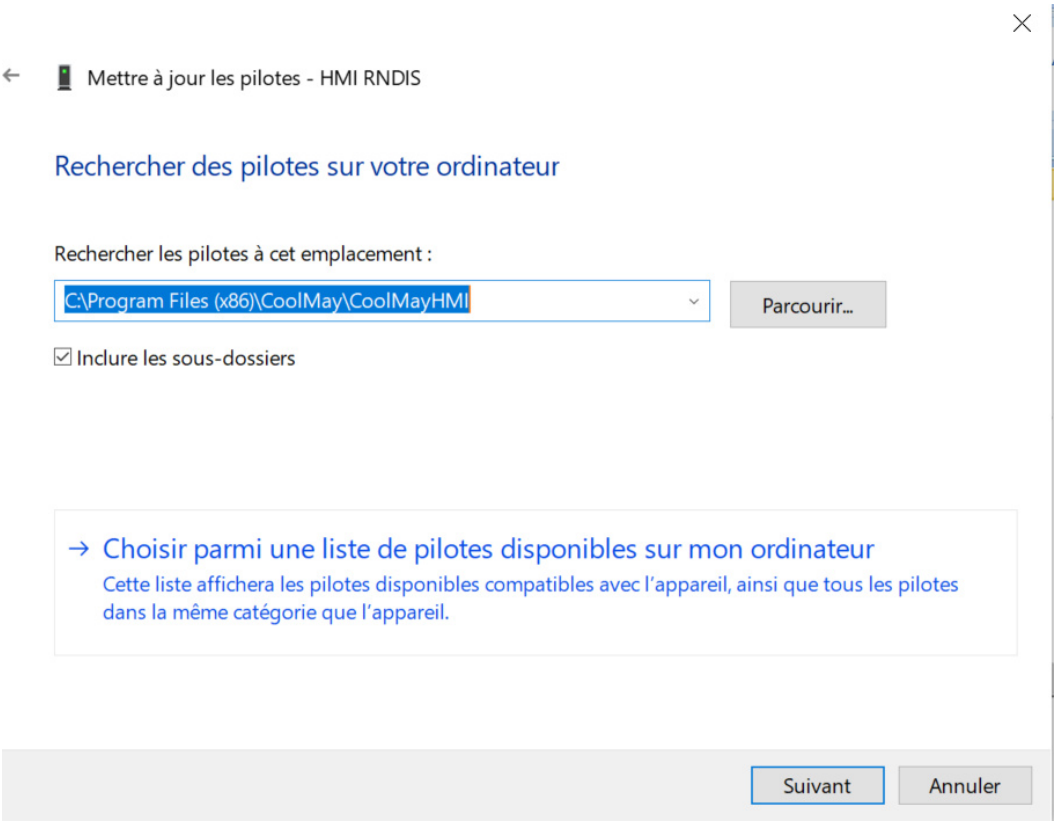


Look for the driver in the PC :

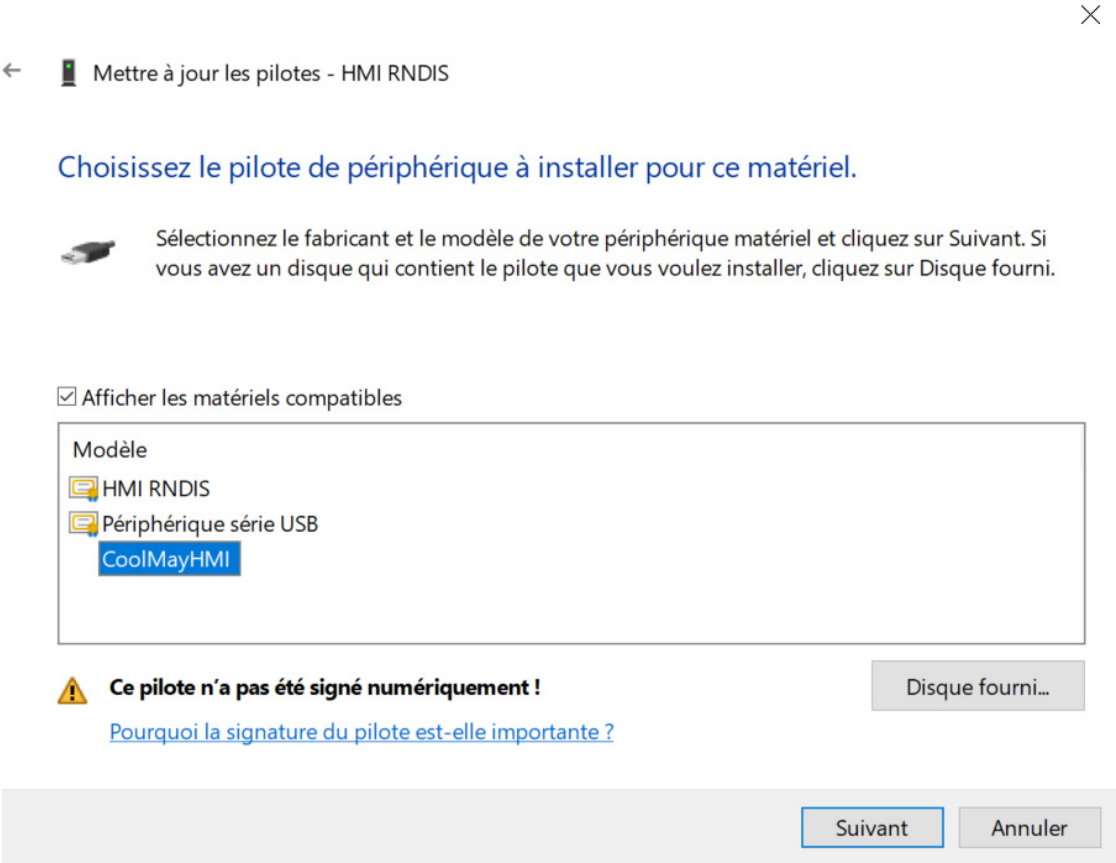


Go to the COOLMAY file you have just installed :

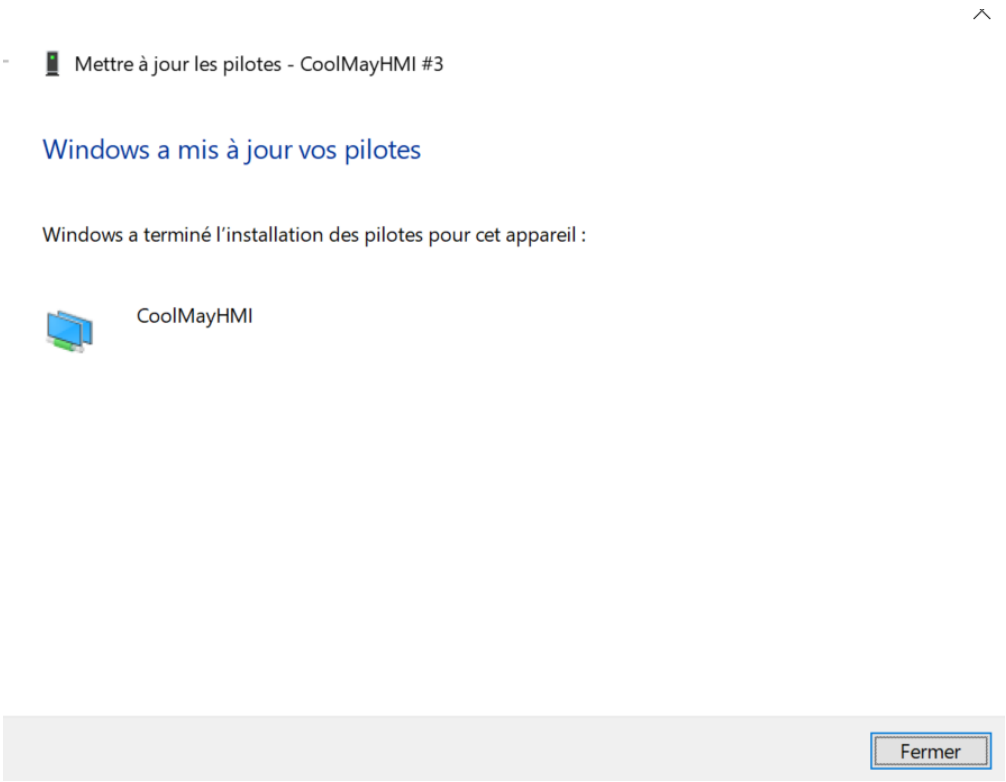


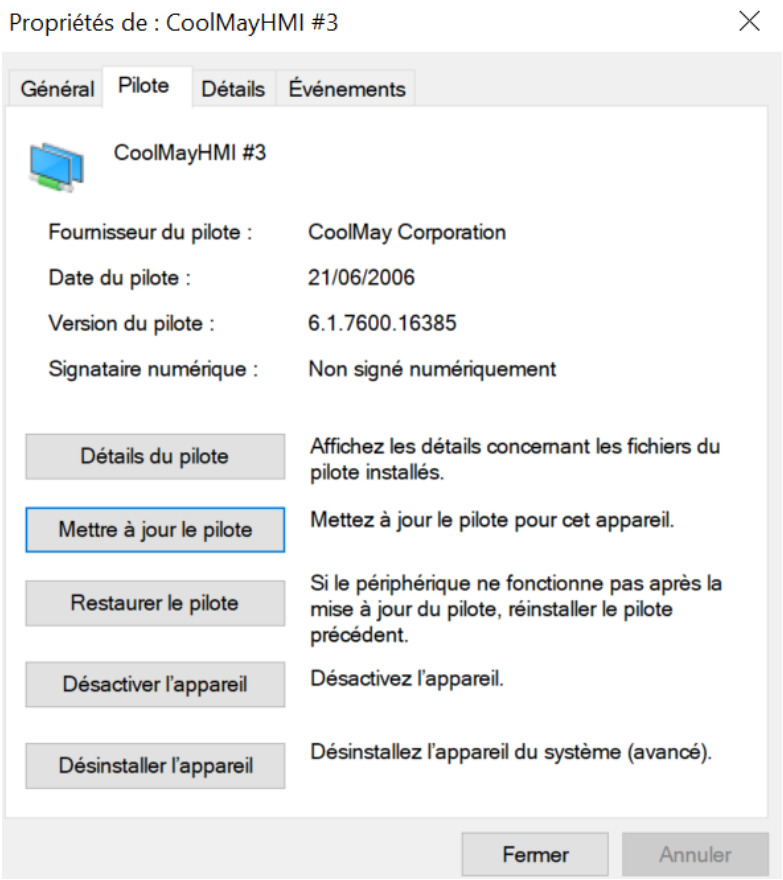


Click on choose in a list : CoolMayHMI

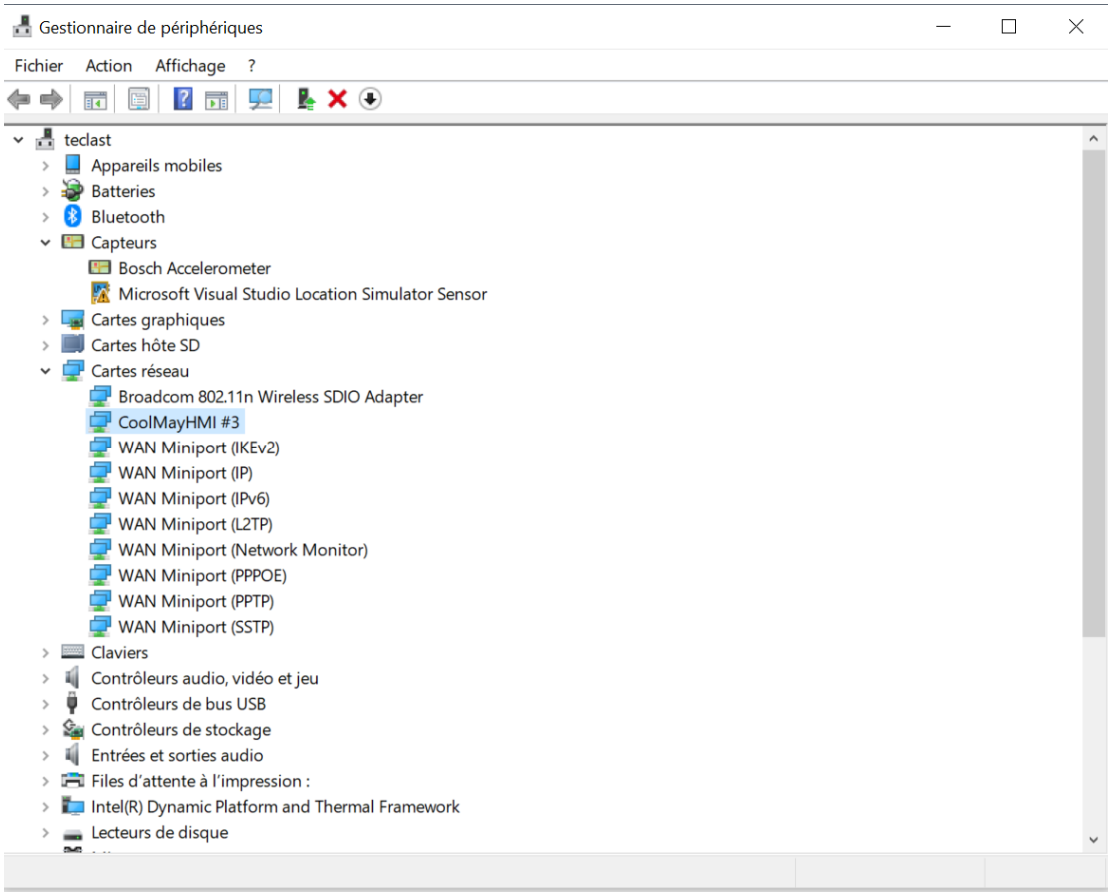


Done :

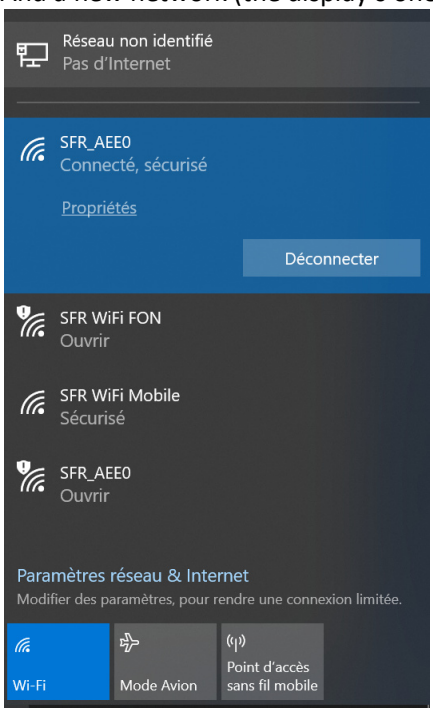




It has been transform as a network connector :



And a new network (the display's one) appears in the list as undentified network with no internet communication.



Now you're able to download the sketch you'll done with CoolmayHMI software.

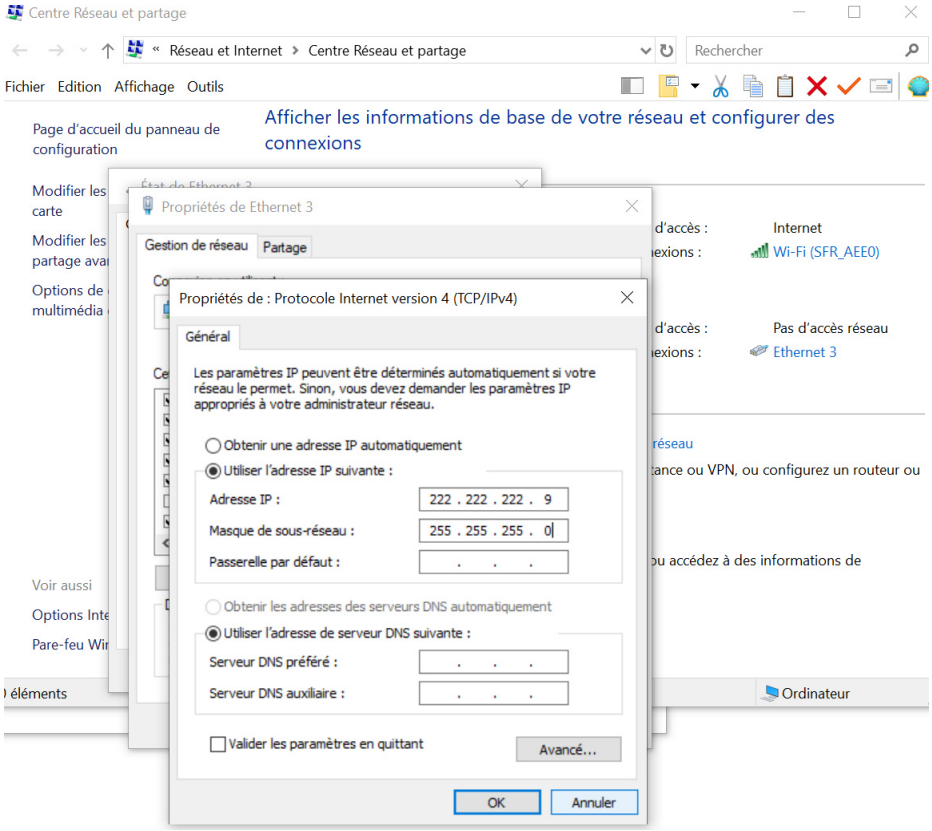
2-2-IP adresses settings and MODBUS RTU communication:

You will have to work with 2 different addresses on your display :

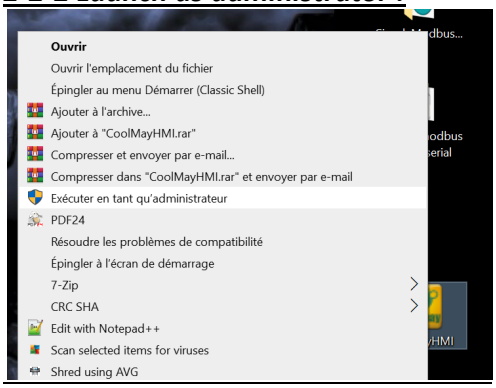
- One is used to download the program
- The other is made for communicate with the ethernet peripheral device (Arduino , PLC….)

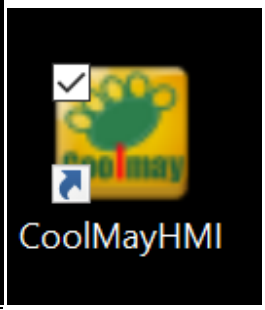
2-2-1 IP address for download :

You must use an IP address like 222.222.222.9 because the basic address of the display is 222.222.222.222

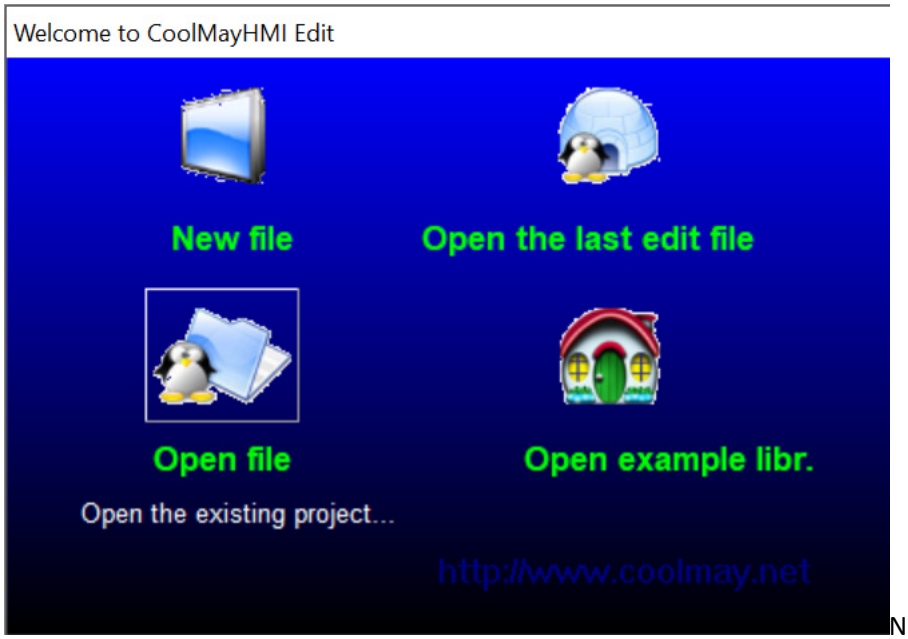


2-2-2 Launch as administrator :



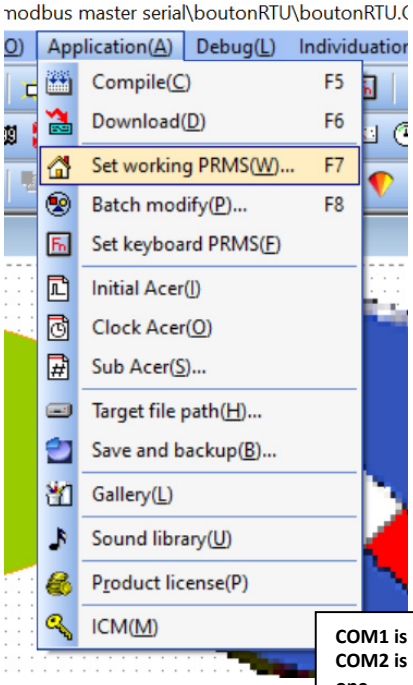


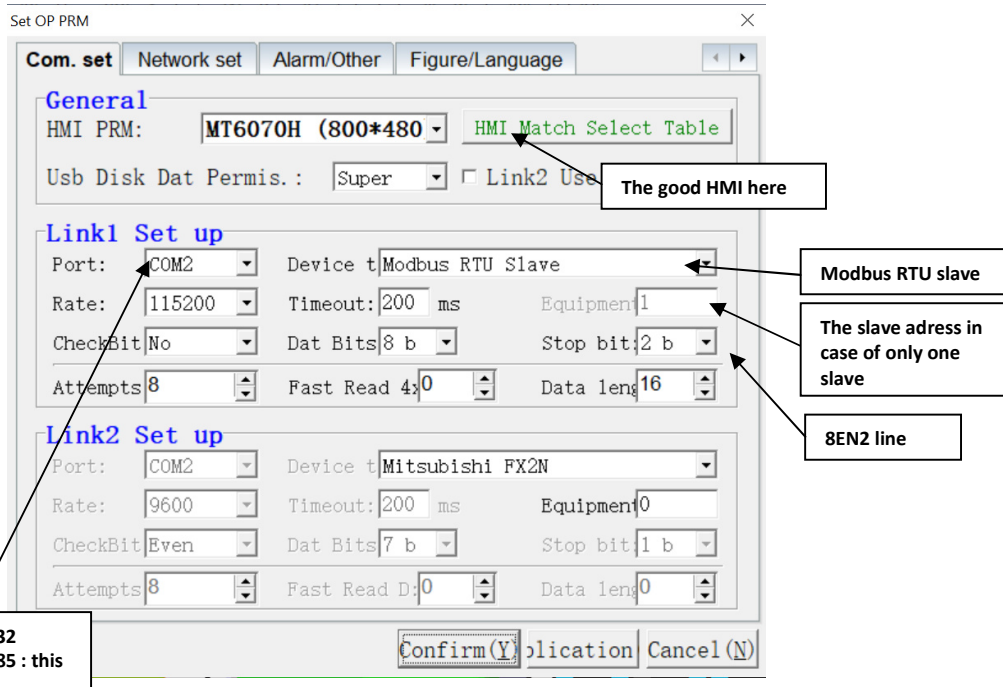
And open the home made sketch IP search :



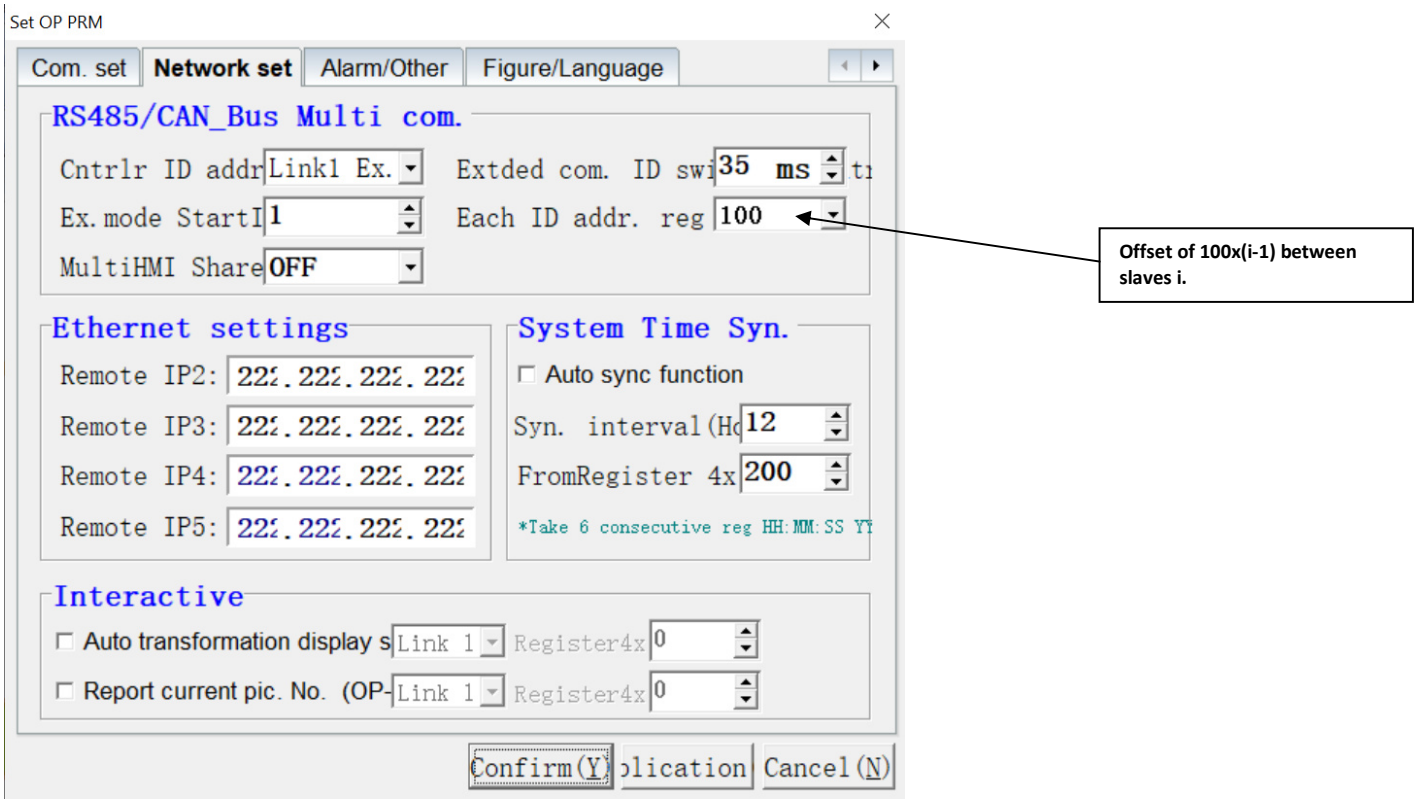
New file

Set working:





COM1 is RS232
COM2 is RS485 : this one
115200 bauds like arduinos



Have a look here for multi slave communication:

1.2.2 Multi-machine Communication Settings

1) Open "Application --- Set Working Parameters --- Network Settings" .

Controller ID Address Mode: Select Extended Mode.

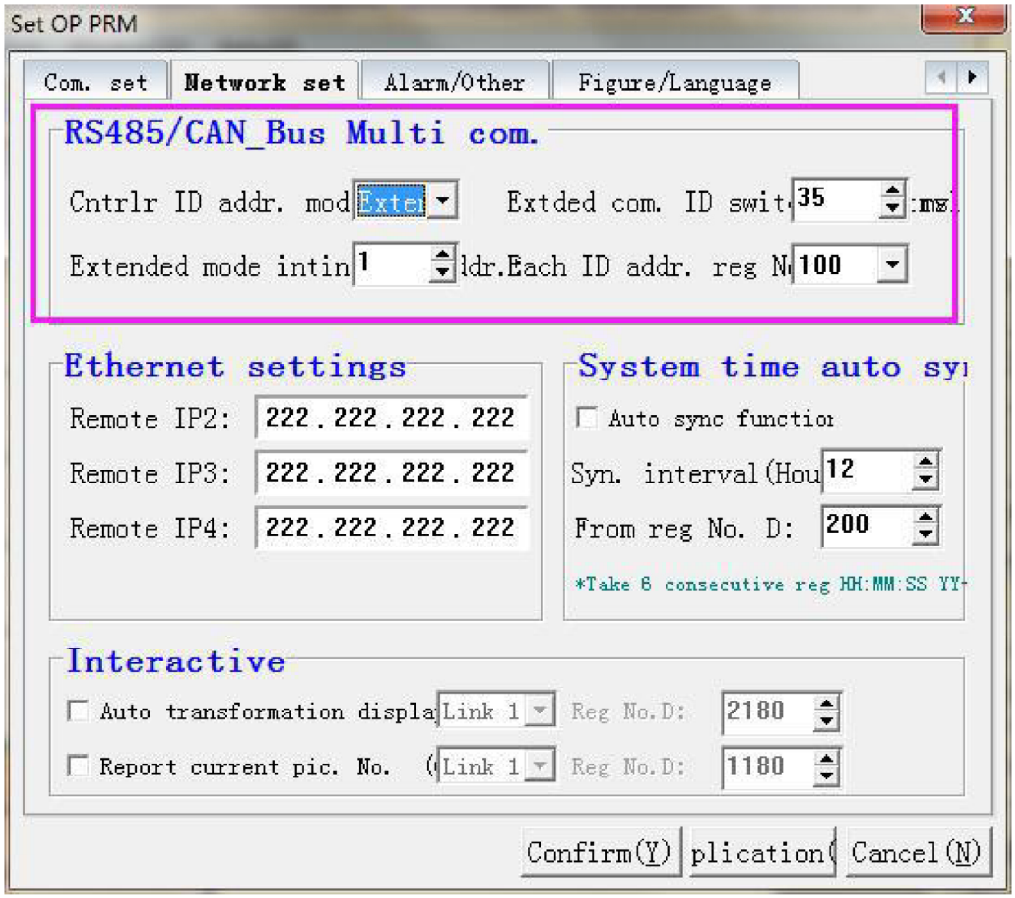
Extended communication ID switching interval: The default is 35ms, which can be adjusted according to actual communication.

Extended Mode Start ID: The default is 1, which is the first slave station number of the connected slave.

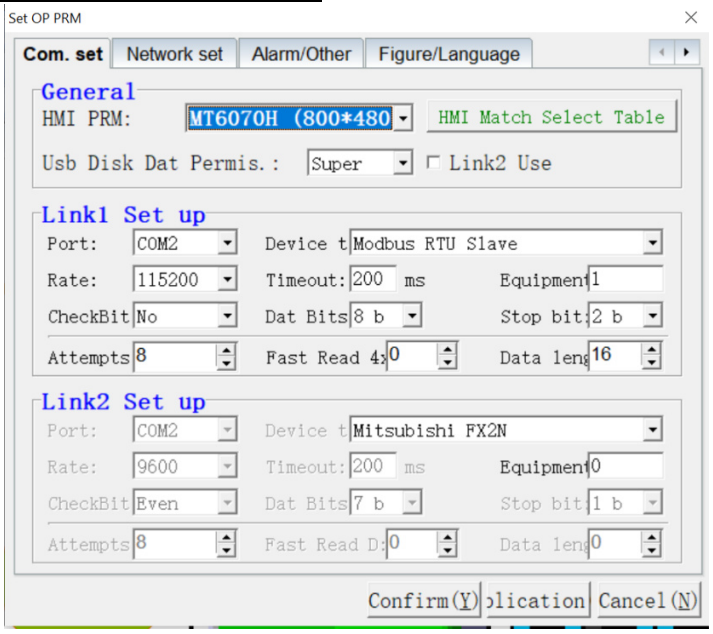
Each ID address register number: 100-30000 range can be set according to the actual register range setting of each slave.

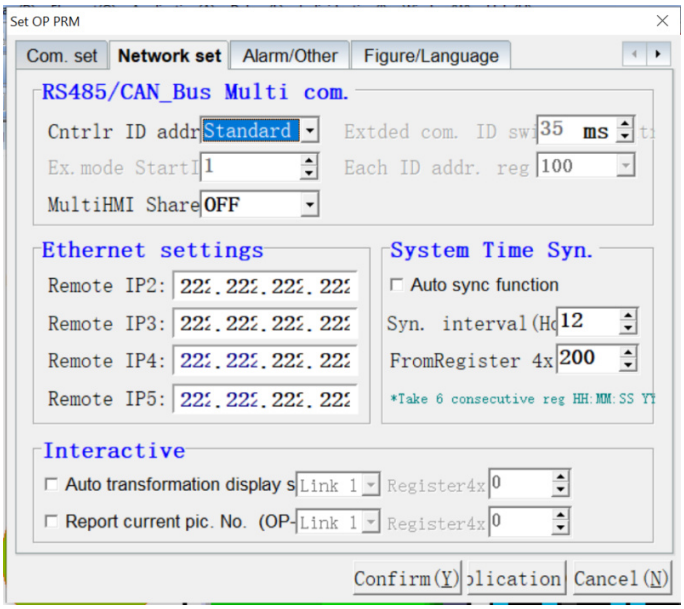
The following figure shows: the HMI is connected with multiple slaves, the first slave station number is from 1. Number of each ID address register set 1000

When 4x0-4x999 indicates the address register of slave 0-999, 4x1000-4x1999 indicates 0-999 of slave 2.The register address, 4x2000-4x2999, represents register address 0-999 of slave 3... and so on.



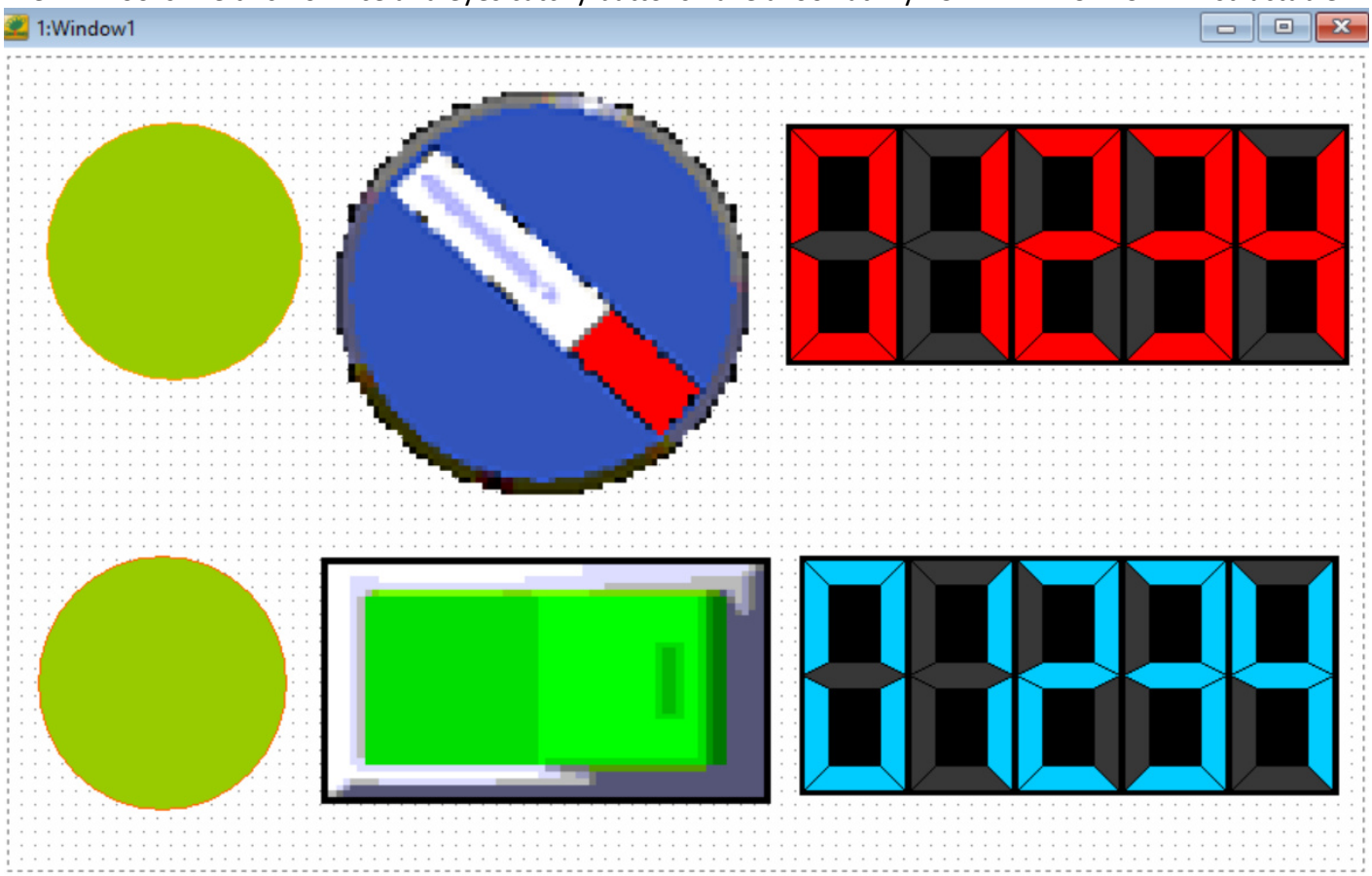
In case of only one slave:





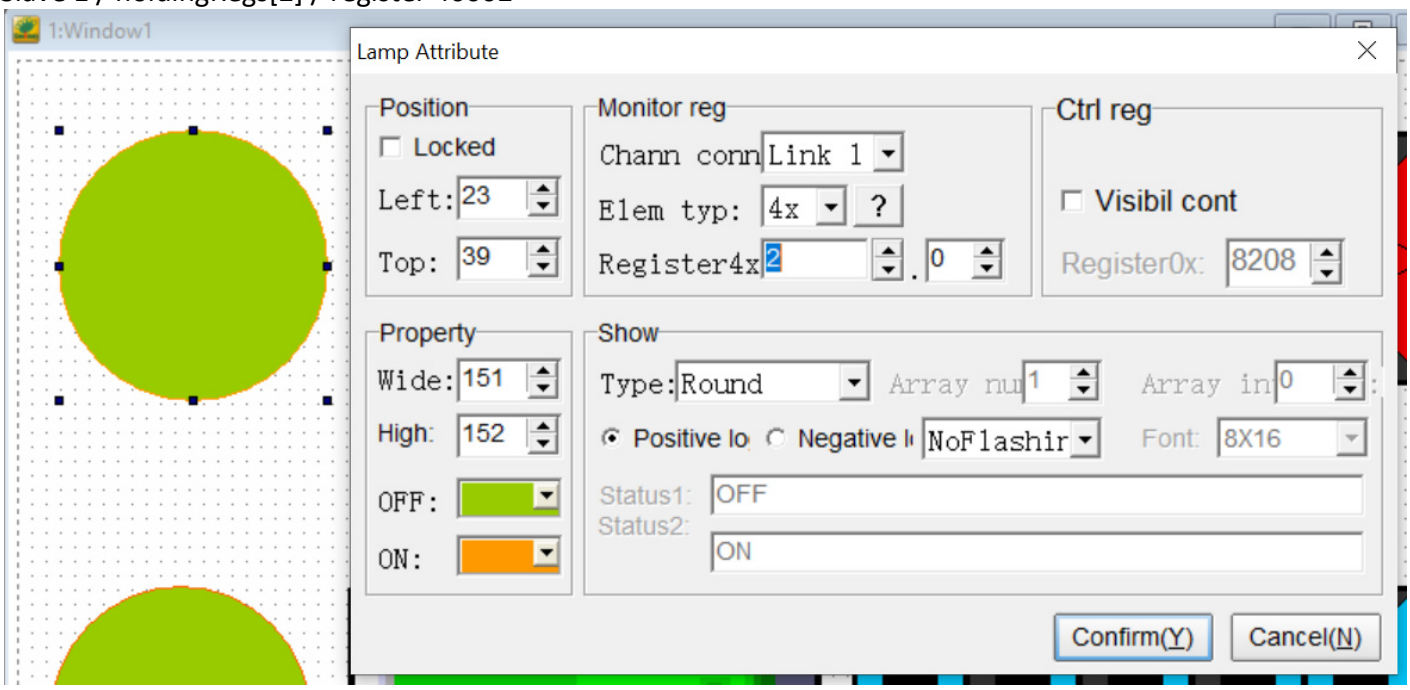
2-3-An example of HMI:

The HMI looks like this: for nice and eyes catchy buttons have a look at my TUNNEL DE CHAUFFE instructable

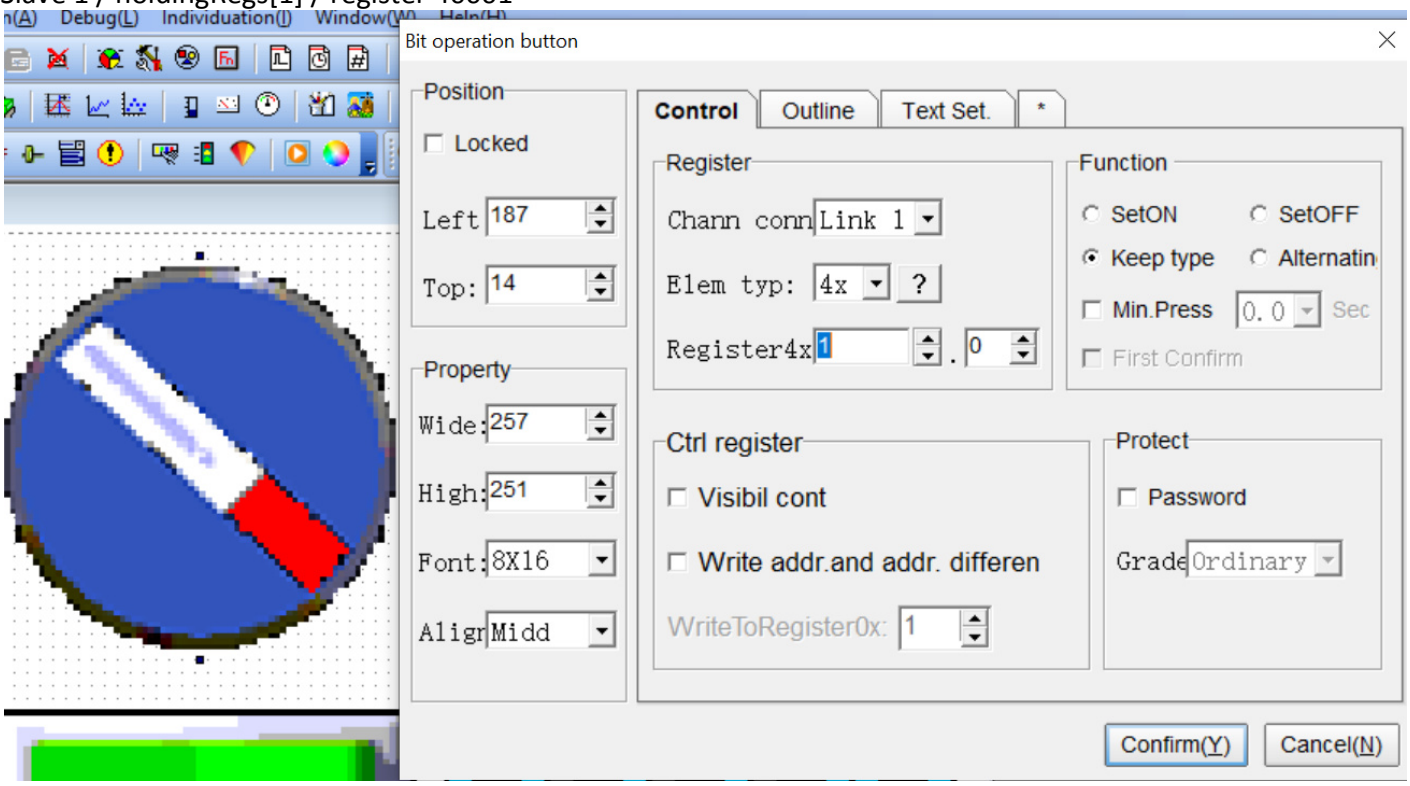


Controls for slave 1:

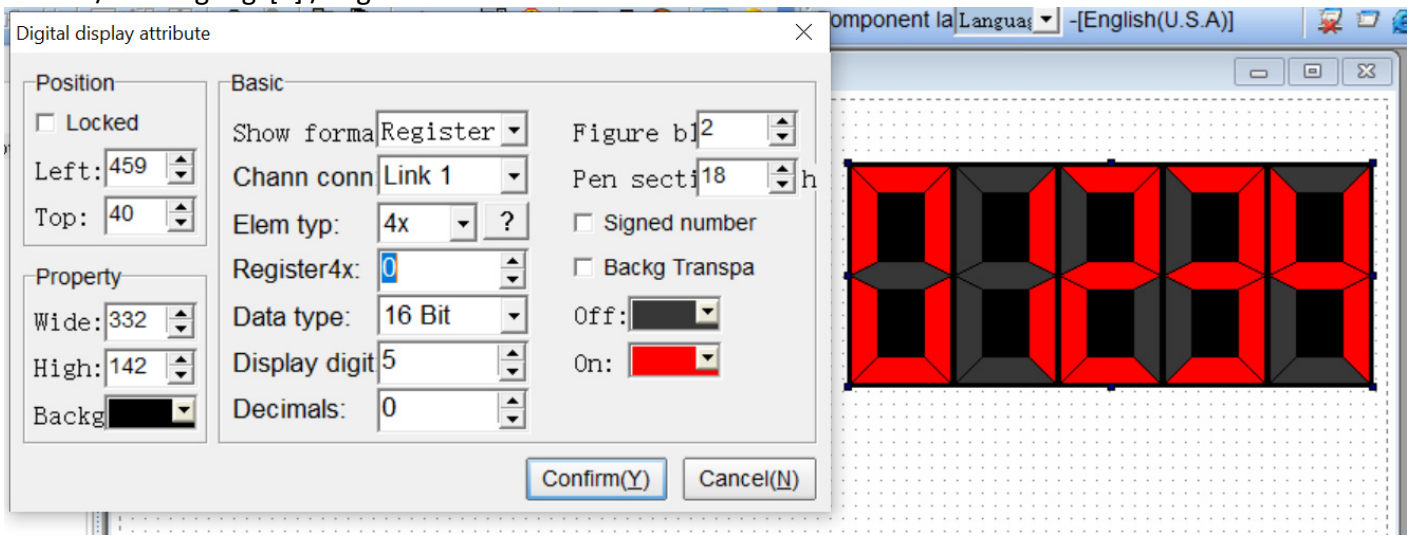
Slave 1 / holdingRegs[2] / register 40002



Slave 1 / holdingRegs[1] / register 40001

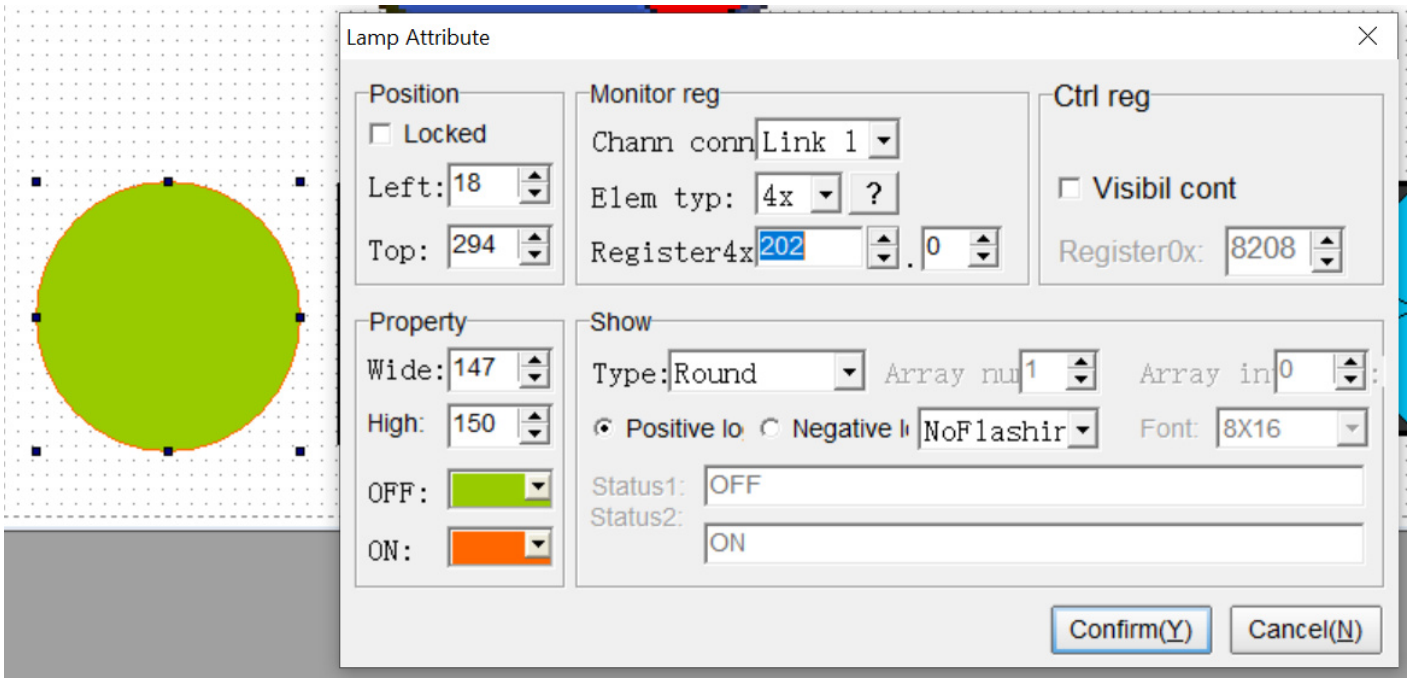


Slave 1 / holdingRegs[0] / register 40000

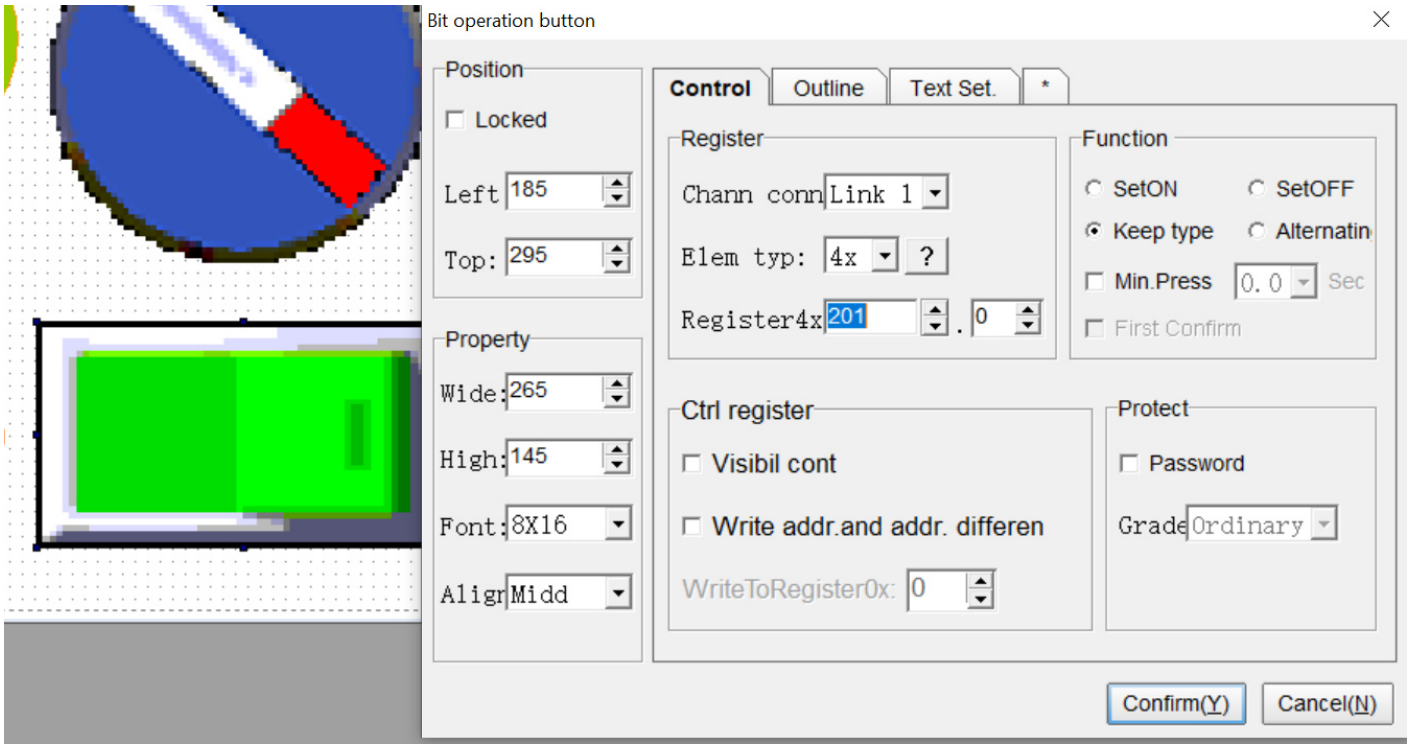


Controls for slave 3:

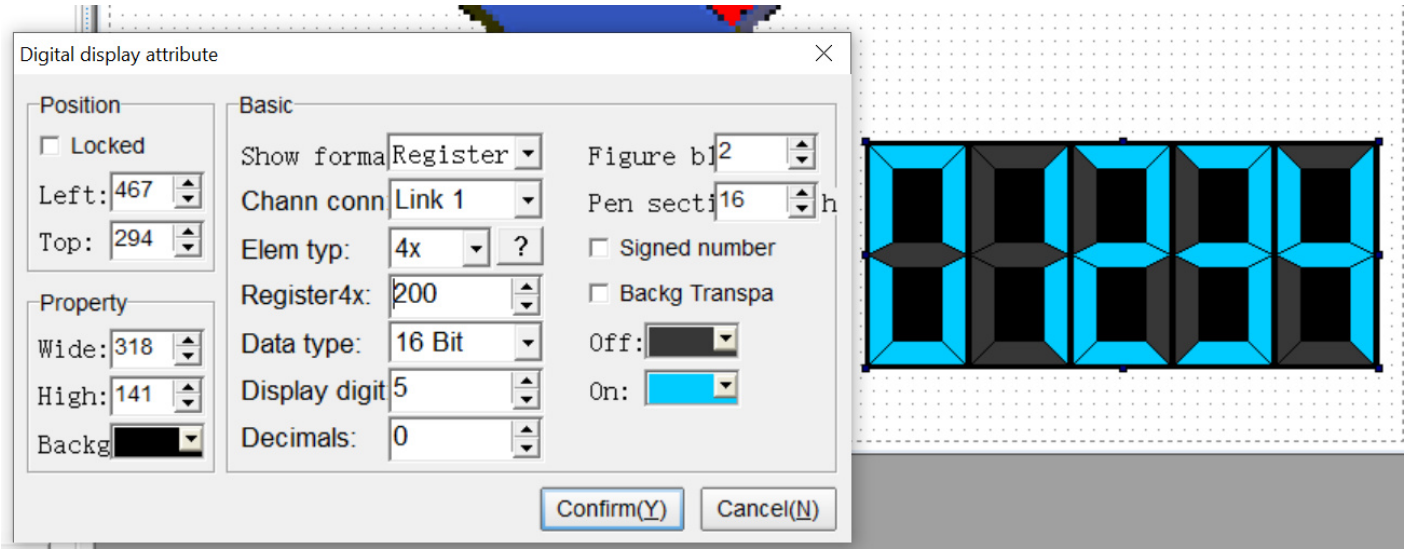
Slave 3 / holdingRegs[2] / register 40202



Slave 3 / holdingRegs[1] / register 40201

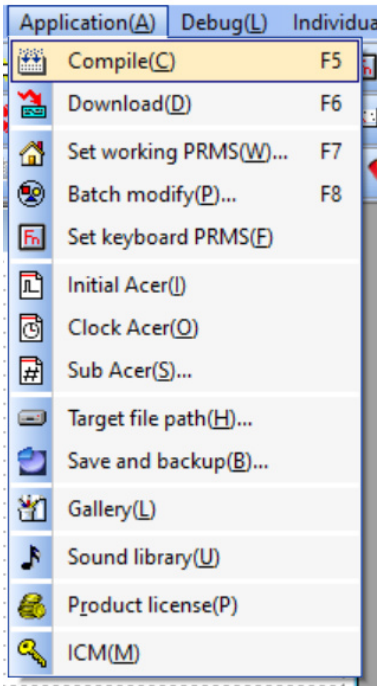


Slave 3 / holdingRegs[0] / register 40200

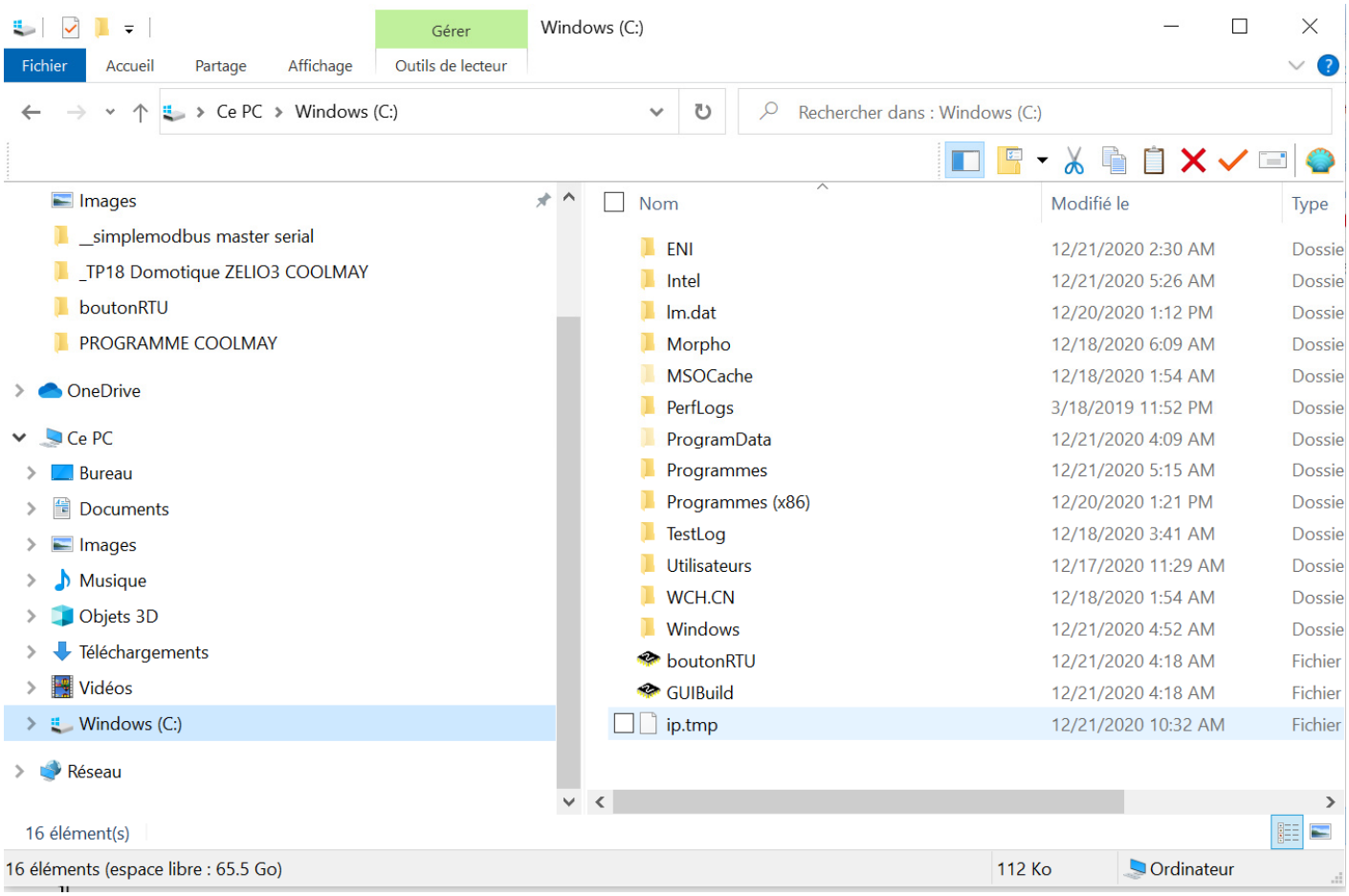


2-4-Upload the sketch:

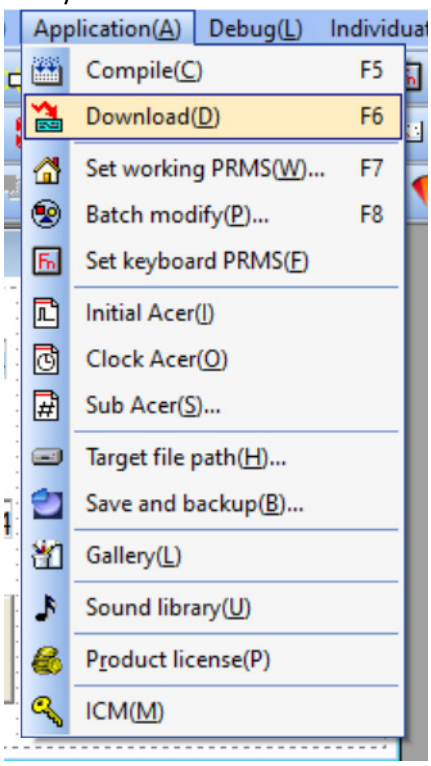
Now, Compile :



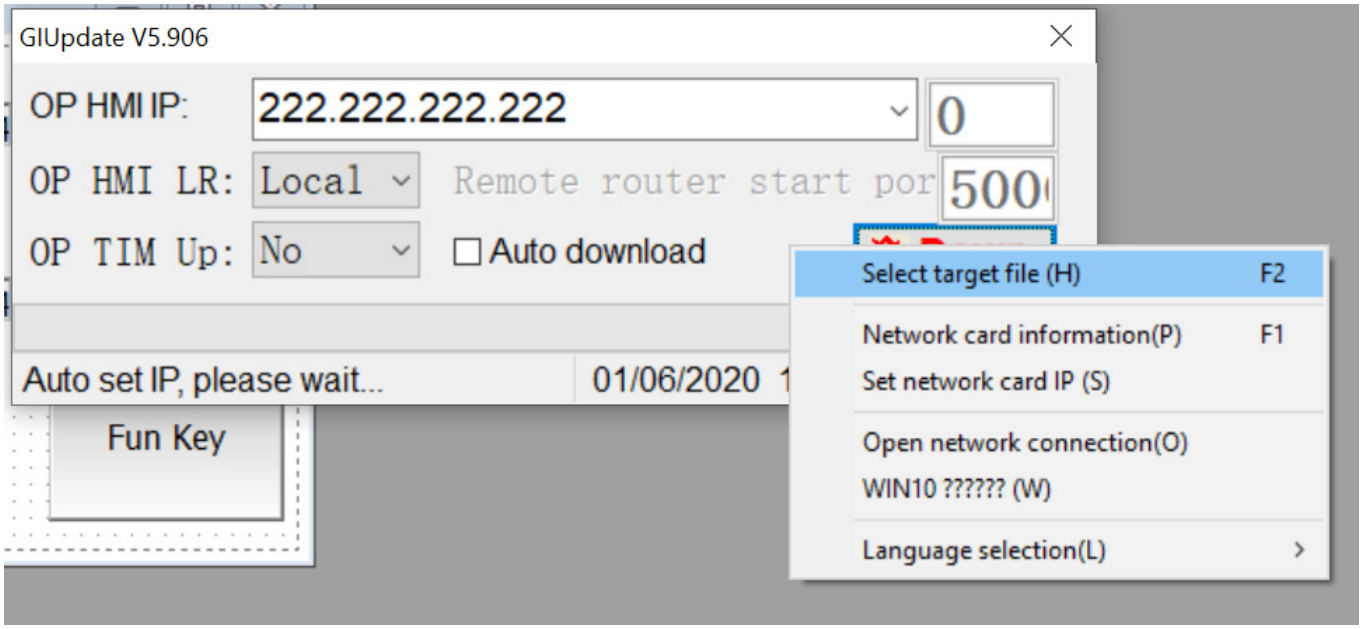
Done and generated in xxx.hw6 under c:/



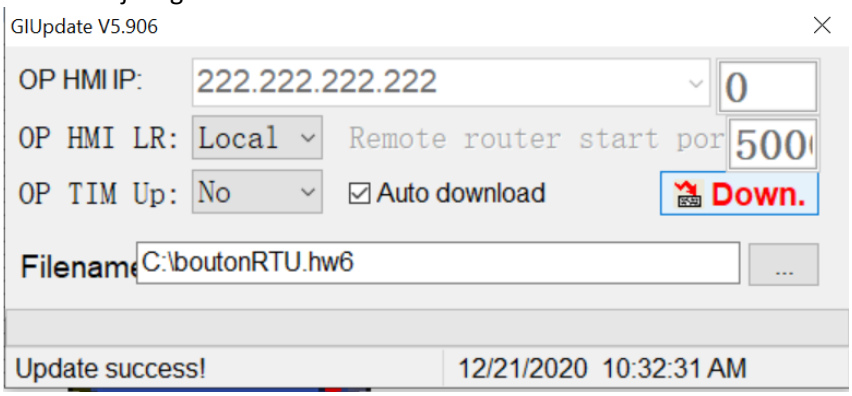
Now you are able to download the sketch : **if Download is not launch, restart CoolmayHMI Build as an Administrator.**



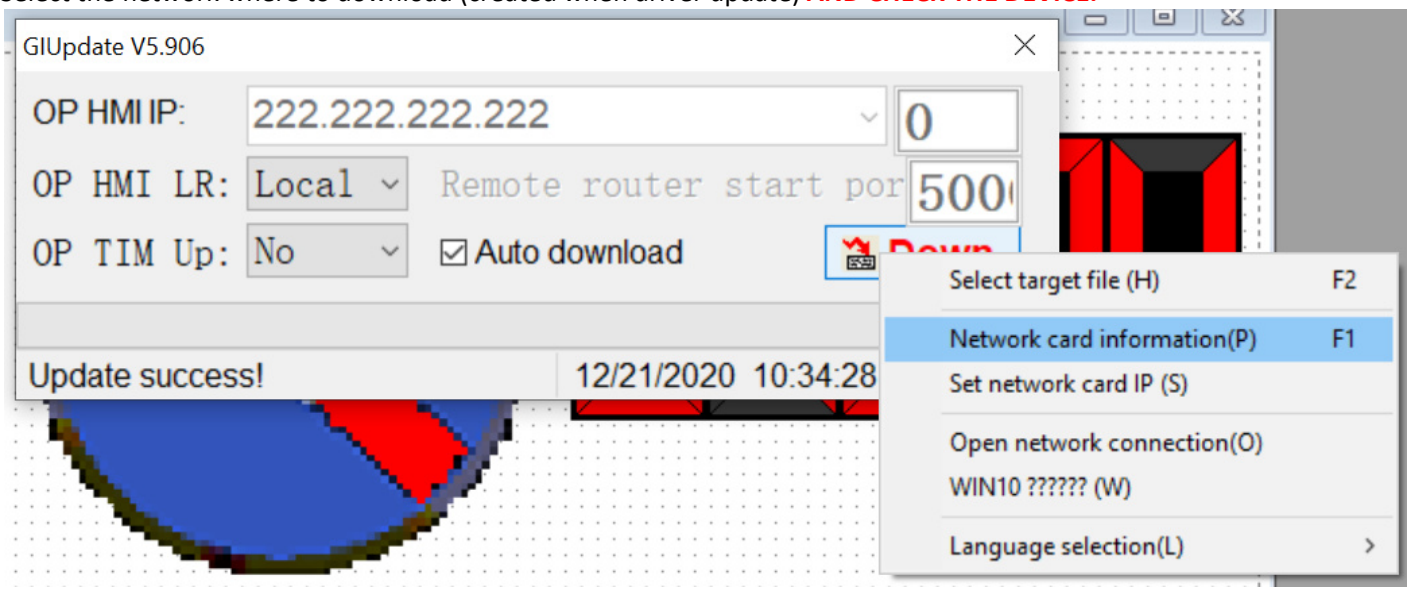
Right click :



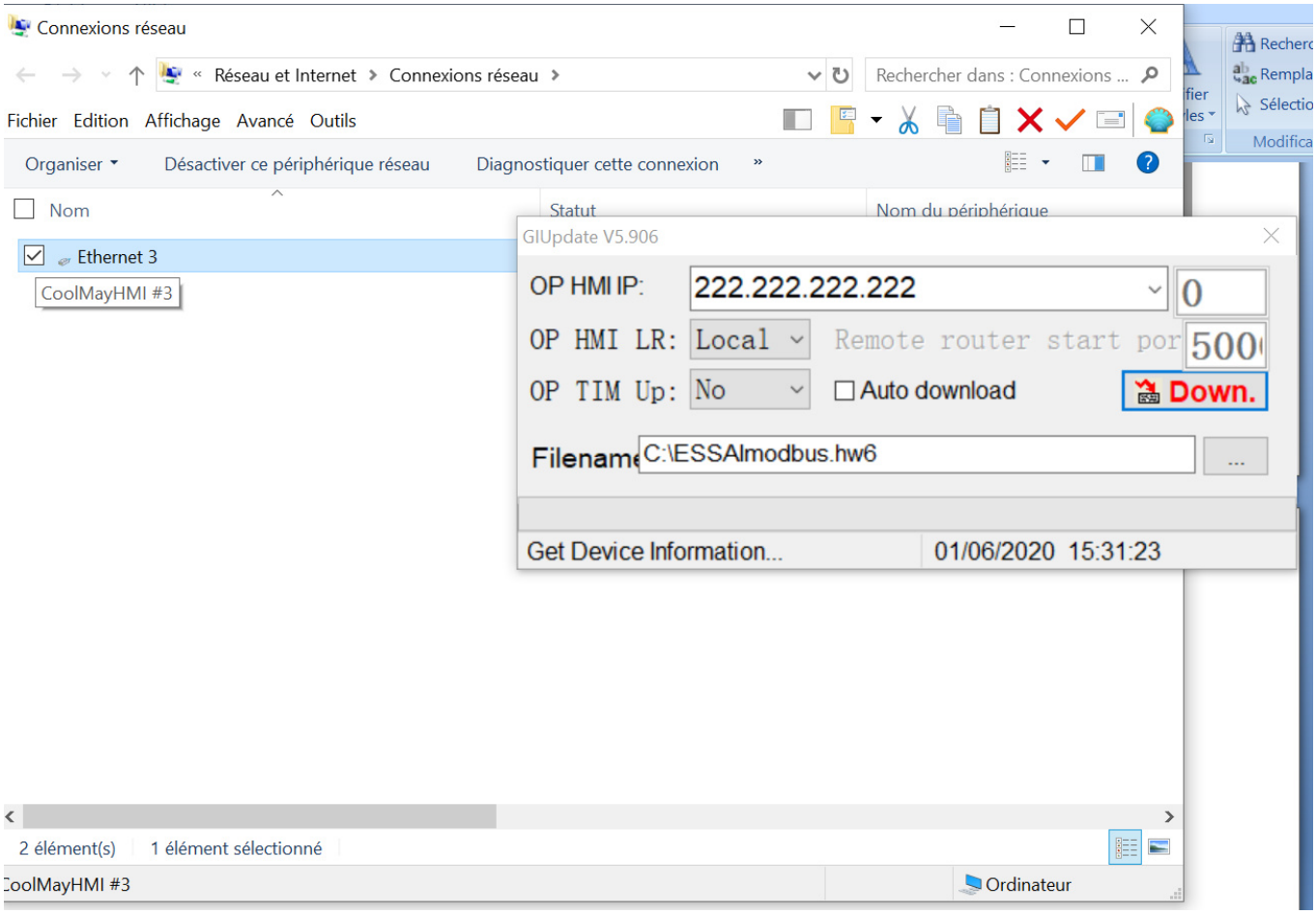
Select the just generated file :



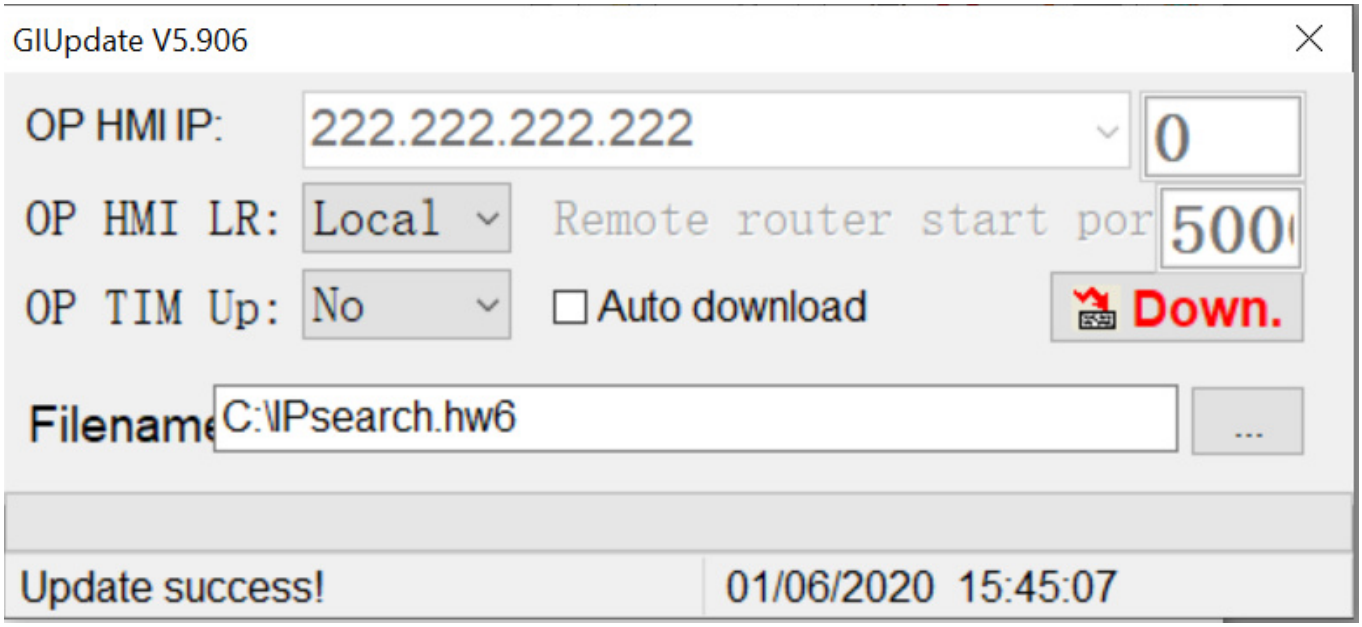
Select the network where to download (created when driver update) **AND CHECK THE DEVICE.**



Select the wlan of the display :



Down :



Success.

If the network doesn't run, stop and restart the master.

PART 2: HMI the SLAVE1, Clone the MASTER, UNO the SLAVE3

The schematic and the links between the devices remain the same. The Clone is now the master because you embed a program in state machine programming to control systems: you can't do it with an HMI as master because it doesn't respect safety machine rules.

The matrix of registers used for the communication:

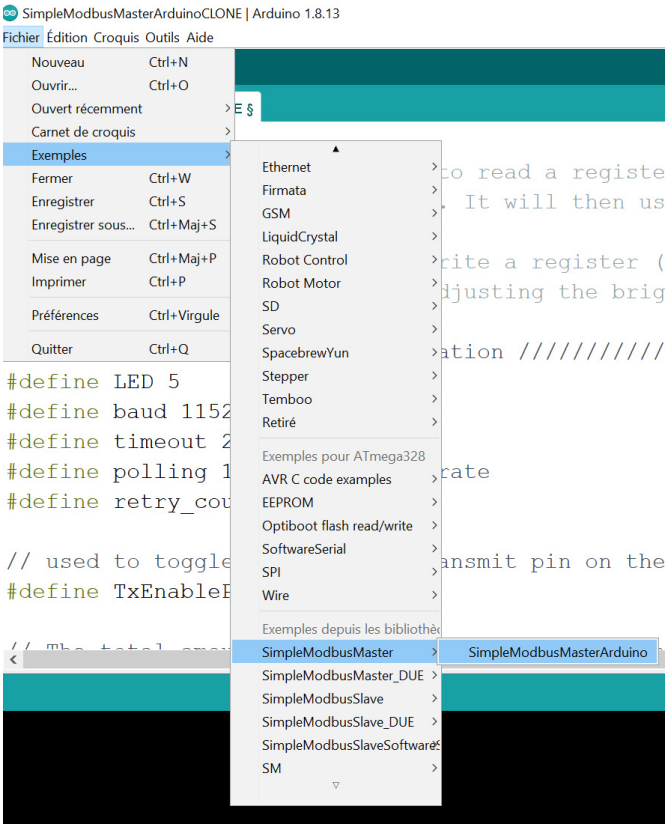
Names of Packets received/sent in the MASTER (enum array)	Addresses in enum array	Holding registers addresses in an array defined by TOTAL_NO_OF_REGISTERS (15 word max for com)	Names
READ_SLAVE3_adr0_6666	0	10	holdingRegs[10] holdingRegs[_40011]
WRITE_SLAVE3_adr1_LED13	1	1	holdingRegs[1] holdingRegs[WRITE_SLAVE3_adr1_LED13]
READ_SLAVE3_adr2_BUTTON	2	14	holdingRegs[14]
_40004	3	3	holdingRegs[3] holdingRegs[_40004]
_40005	4	4	holdingRegs[4] holdingRegs[_40005]
_40006	5	5	holdingRegs[5] holdingRegs[_40006]
_40007	6	6	holdingRegs[6] holdingRegs[_40007]
_40008	7	7	holdingRegs[7] holdingRegs[_40008]
_40009	8	8	holdingRegs[8] holdingRegs[_40009]
_40010	9	Not used	
_40011	10	10	holdingRegs[10] holdingRegs[_40011]
_40012	11	Not used	
_40013	12	Not used	
TOTAL_NO_OF_PACKETS	No address and leave this one		
		13	
		14	holdingRegs[14]
		15	

TOTAL_NO_OF_REGISTERS (here 15) must be equal or greater than TOTAL_NO_OF_PACKETS (here 12).

When you read a value from a slave you must do it via another register BECAUSE YOU REACH DIRECTLY the READ VALUES you want to proceed. For example, here I read a value ‘6666’ from the slave3 and I want to display it on the HMI (slave1) on an LCD display with th address 40009. First: I store the value in holdingRegs[10] or holdingRegs[_40011] (but not holdingRegs[READ_SLAVE3_adr0_6666], not working). Second: I store holdingRegs[10] or holdingRegs[_40011] in holdingRegs[8] or holdingRegs[_40009].

2-1 The Clone as MASTER:

The skectch is based on this example:



The communication seems to be very slow. In order to get more speed:
-I use a state machine for a multitasking running
-I change some communication settings like this:
SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit
115200 bauds

modbus_construct is a function which gives addresses and makes relationships between the registers of enum (coming from the slaves) and the local registers of the master:

```
//modbus_construct(&packets[THE REGISTER IN THE LIST], 1, READ_HOLDING_REGISTERS, register @,
//slave id, 0);
modbus_construct(&packets[READ_SLAVE3_adr0_6666], 3, READ_HOLDING_REGISTERS, 0, 1, 10);
modbus_construct(&packets[READ_SLAVE3_adr2_BUTTON], 3, READ_HOLDING_REGISTERS, 2, 1, 14);
modbus_construct(&packets[WRITE_SLAVE3_adr1_LED13], 3, PRESET_MULTIPLE_REGISTERS, 1, 1, 1);

modbus_construct(&packets[_40005], 1, READ_HOLDING_REGISTERS, 0x40005, 1, 4);
modbus_construct(&packets[_40004], 1, PRESET_MULTIPLE_REGISTERS, 0x40004, 1, 3);
modbus_construct(&packets[_40006], 1, PRESET_MULTIPLE_REGISTERS, 0x40006, 1, 5);

modbus_construct(&packets[_40008], 1, READ_HOLDING_REGISTERS, 0x40008, 1, 7);
modbus_construct(&packets[_40007], 1, PRESET_MULTIPLE_REGISTERS, 0x40007, 1, 6);
modbus_construct(&packets[_40009], 1, PRESET_MULTIPLE_REGISTERS, 0x40009, 1, 8);
```

Here the master operates and controls inputs and outputs registers:

```
//holdingRegs[ADC_VAL] = analogRead(A0); // update data to be read by the master
//to adjust the PWM
holdingRegs[_40006] = 32565;
holdingRegs[_40004] = digitalRead(14);
memo = holdingRegs[_40005];
digitalWrite(LED, memo);

//holdingRegs[_40009] = holdingRegs[10];
//holdingRegs[_40009] = holdingRegs[0]; NOT WORKING
holdingRegs[_40009] = holdingRegs[_40011];
//holdingRegs[_40009] = holdingRegs[READ_SLAVE3_adr0_6666];NOT WORKING
holdingRegs[WRITE_SLAVE3_adr1_LED13] = holdingRegs[_40008];
//holdingRegs[_40007] = holdingRegs[READ_SLAVE3_adr2_BUTTON]; NOT WORKING
holdingRegs[_40007] = holdingRegs[14];
```

If the network doesn't run, reset the master.

<u>The sketch of the master:</u>

SimpleModbusMasterArduinoCLONE

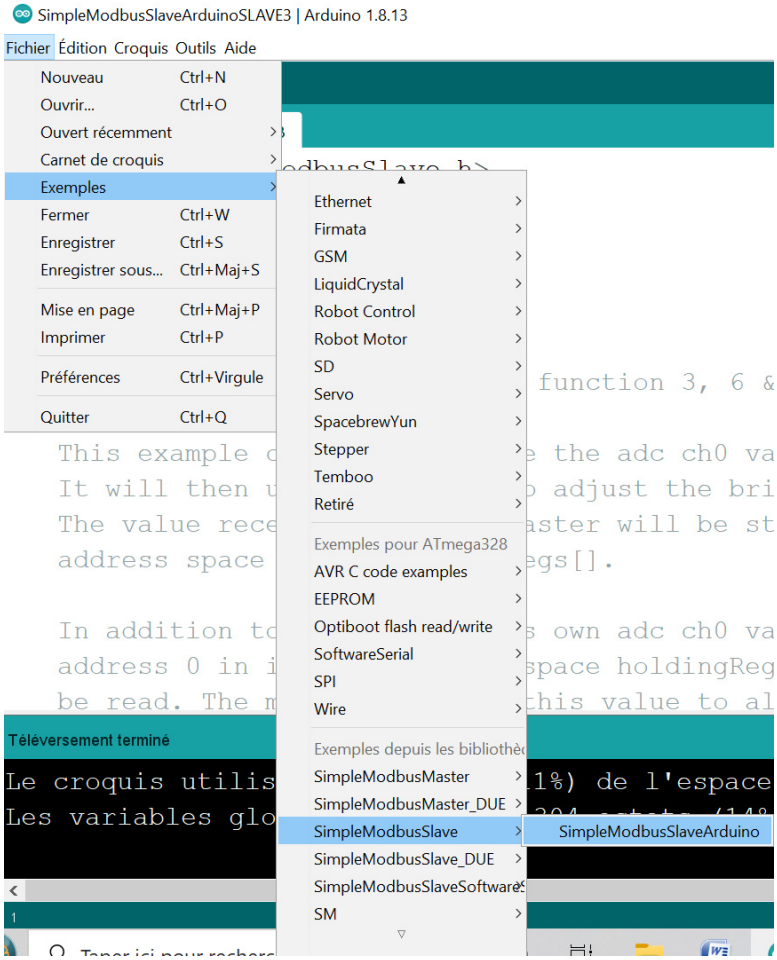
```
#include <SimpleModbusMaster.h>
#include <SM.h>

SM machine(&etape0);
/*
   The example will use packet1 to read a register from address 0 (the adc ch0 value)
   from the arduino slave (id=1). It will then use this value to adjust the brightness
   of an led on pin 9 using PWM.
   It will then use packet2 to write a register (its own adc ch0 value) to address 1
   on the arduino slave (id=1) adjusting the brightness of an led on pin 9 using PWM.
*/
//////////////////// Port information ///////////////////
#define LED 5
#define baud 115200
#define timeout 250
#define polling 100 // the scan rate
#define retry_count 5 // 10
// used to toggle the receive/transmit pin on the driver
#define TxEnablePin 2

// The total amount of available memory on the master to store data OF ENUM ARRAY
#define TOTAL_NO_OF_REGISTERS 15

// This is the easiest way to create new packets
// Add as many as you want.
// TOTAL_NO_OF_PACKETS is automatically updated.
```

```
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  READ_SLAVE3_adr0_6666,  //address 0 6666
  WRITE_SLAVE3_adr1_LED13,  //address 1 LED 13
  READ_SLAVE3_adr2_BUTTON, // address 2  BUTTON AO
  _40004, // address 3
  _40005, // address 4
  _40006, // address 5
  _40007, // address 6
  _40008, // address 7
  _40009, // address 8
  _40010, // address 9
  _40011, // address 10
  _40012, // address 11
  _40013, // address 12
  TOTAL_NO_OF_PACKETS // leave this one
  // total number of registers for function 3 and 16 share the same register array
  // i.e. the same address space
};

// Create an array of Packets to be configured
Packet packets[TOTAL_NO_OF_PACKETS];

// Masters register array
unsigned int holdingRegs[TOTAL_NO_OF_REGISTERS]; // function 3 and 16 register array

int memo;
int memo2;
void setup()
{
  /* parameters(HardwareSerial* SerialPort,
                long baudrate,
    unsigned char byteFormat,
                unsigned char ID,
                unsigned char transmit enable pin,
                unsigned int holding registers size,
                unsigned int* holding register array)
  */
  /* Valid modbus byte formats are:
     SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits
     SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit
     SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit
     You can obviously use SERIAL_8N1 but this does not adhere to the
     Modbus specifications. That said, I have tested the SERIAL_8N1 option
     on various commercial masters and slaves that were suppose to adhere
```

```
    to this specification and was always able to communicate... Go figure.
    These byte formats are already defined in the Arduino global name space.
  */
  // Initialize each packet
  //modbus_construct(&packets[THE REGISTER IN THE LIST], 1, READ_HOLDING_REGISTERS, register @,
  //slave id, 0);
  modbus_construct(&packets[READ_SLAVE3_adr0_6666], 3, READ_HOLDING_REGISTERS, 0, 1, 10);
  modbus_construct(&packets[READ_SLAVE3_adr2_BUTTON], 3, READ_HOLDING_REGISTERS, 2, 1, 14);
  modbus_construct(&packets[WRITE_SLAVE3_adr1_LED13], 3, PRESET_MULTIPLE_REGISTERS, 1, 1, 1);

  modbus_construct(&packets[_40005], 1, READ_HOLDING_REGISTERS, 0x40005, 1, 4);
  modbus_construct(&packets[_40004], 1, PRESET_MULTIPLE_REGISTERS, 0x40004, 1, 3);
  modbus_construct(&packets[_40006], 1, PRESET_MULTIPLE_REGISTERS, 0x40006, 1, 5);

  modbus_construct(&packets[_40008], 1, READ_HOLDING_REGISTERS, 0x40008, 1, 7);
  modbus_construct(&packets[_40007], 1, PRESET_MULTIPLE_REGISTERS, 0x40007, 1, 6);
  modbus_construct(&packets[_40009], 1, PRESET_MULTIPLE_REGISTERS, 0x40009, 1, 8);
  //modbus_construct(&packets[holdingRegs[0]], 1, PRESET_MULTIPLE_REGISTERS, 1, 1, 0);

  // Initialize the Modbus Finite State Machine
  modbus_configure(&Serial, baud, SERIAL_8O1, timeout, polling, retry_count, TxEnablePin, packets,
                                                                    TOTAL_NO_OF_PACKETS, holdingRegs);
  pinMode(LED, OUTPUT);
  pinMode(14, INPUT); //AO
}

void loop()
{
  EXEC(machine);
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.

  modbus_update();


}
State etape0() {
  //holdingRegs[ADC_VAL] = analogRead(A0); // update data to be read by the master
  //to adjust the PWM
  holdingRegs[_40006] = 32565;
  holdingRegs[_40004] = digitalRead(14);
  memo = holdingRegs[_40005];
  digitalWrite(LED, memo);

  //holdingRegs[_40009] = holdingRegs[10];
  //holdingRegs[_40009] = holdingRegs[0]; NOT WORKING
  holdingRegs[_40009] = holdingRegs[_40011];
  //holdingRegs[_40009] = holdingRegs[READ_SLAVE3_adr0_6666];NOT WORKING
  holdingRegs[WRITE_SLAVE3_adr1_LED13] = holdingRegs[_40008];
  //holdingRegs[_40007] = holdingRegs[READ_SLAVE3_adr2_BUTTON]; NOT WORKING
  holdingRegs[_40007] = holdingRegs[14];
  /* Note:
     The use of the enum instruction is not needed. You could set a maximum allowable
     size for holdinRegs[] by defining HOLDING_REGS_SIZE using a constant and then access
     holdingRegs[] by "Index" addressing.
     I.e.
     holdingRegs[0] = analogRead(A0);
     analogWrite(LED, holdingRegs[1]/4);
  */

}
```

2-2 The UNO as SLAVE3:

The skectch is based on this example:



The communication seems to be very slow. In order to get more speed:

-I use a state machine for a multitasking running

-I change some communication settings like this:

SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit

115200 bauds

SimpleModbusSlaveArduinoSLAVE3

```
#include <SimpleModbusSlave.h>
#include <SM.h>

SM VALEUR(&etape0);
SM MACHINE(&etape10);
/*
   SimpleModbusSlaveV10 supports function 3, 6 & 16.

   This example code will receive the adc ch0 value from the arduino master.
   It will then use this value to adjust the brightness of the led on pin 9.
   The value received from the master will be stored in address 1 in its own
   address space namely holdingRegs[].

   In addition to this the slaves own adc ch0 value will be stored in
   address 0 in its own address space holdingRegs[] for the master to
```

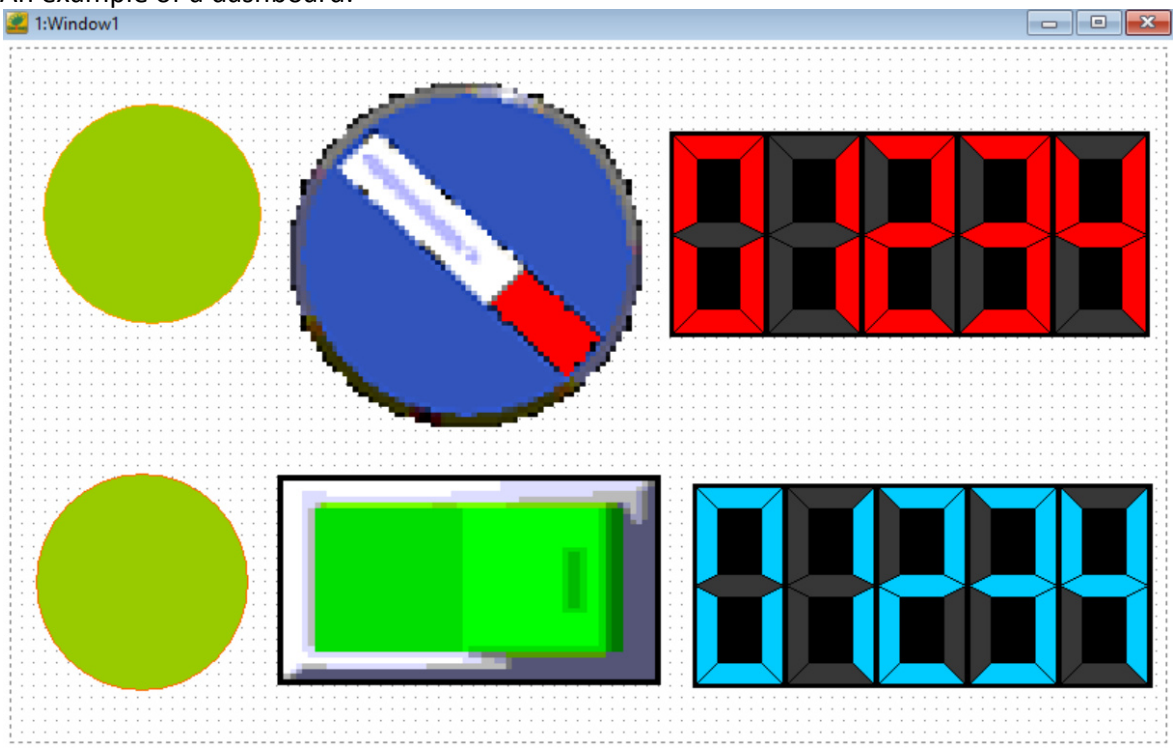
```
  be read. The master will use this value to alter the brightness of its
  own led connected to pin 9.

  The modbus_update() method updates the holdingRegs register array and checks
  communication.

  Note:
  The Arduino serial ring buffer is 64 bytes or 32 registers.
  Most of the time you will connect the arduino to a master via serial
  using a MAX485 or similar.

  In a function 3 request the master will attempt to read from your
  slave and since 5 bytes is already used for ID, FUNCTION, NO OF BYTES
  and two BYTES CRC the master can only request 58 bytes or 29 registers.

  In a function 16 request the master will attempt to write to your
  slave and since a 9 bytes is already used for ID, FUNCTION, ADDRESS,
  NO OF REGISTERS, NO OF BYTES and two BYTES CRC the master can only write
  54 bytes or 27 registers.

  Using a USB to Serial converter the maximum bytes you can send is
  limited to its internal buffer which differs between manufactures.
*/

#define  LED 13

// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////

enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  ADC_VAL,  //adress 0
  PWM_VAL,  //ADDRESS 1
  BUTTON, // address 2
  HOLDING_REGS_SIZE // leave this one
  // total number of registers for function 3 and 16 share the same register array
  // i.e. the same address space
};
```

```
unsigned int holdingRegs[HOLDING_REGS_SIZE]; // function 3 and 16 register array
////////////////////////////////////////////////////////////
int memo;
int valeur;

void setup()
{
  /* parameters(HardwareSerial* SerialPort,
                long baudrate,
    unsigned char byteFormat,
                unsigned char ID,
                unsigned char transmit enable pin,
                unsigned int holding registers size,
                unsigned int* holding register array)
  */

  /* Valid modbus byte formats are:
     SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits
     SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit
     SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit

     You can obviously use SERIAL_8N1 but this does not adhere to the
     Modbus specifications. That said, I have tested the SERIAL_8N1 option
     on various commercial masters and slaves that were suppose to adhere
     to this specification and was always able to communicate... Go figure.

     These byte formats are already defined in the Arduino global name space.
  */


  modbus_configure(&Serial, 115200, SERIAL_8O1, 3, 2, HOLDING_REGS_SIZE, holdingRegs);

  // modbus_update_comms(baud, byteFormat, id) is not needed but allows for easy update
  //of the
  // port variables and slave id dynamically in any function.
  modbus_update_comms(115200, SERIAL_8O1, 3);

  pinMode(LED, OUTPUT);
  pinMode(14, INPUT);

}

void loop()
{
  EXEC(VALEUR);
  EXEC(MACHINE);
  // modbus_update() is the only method used in loop(). It returns the total error
  // count since the slave started. You don't have to use it but it's useful
  // for fault finding by the modbus master.
  modbus_update();
}
```

```
State etape10() {
 //holdingRegs[ADC_VAL] = analogRead(A0); // update data to be read by the master to
                                          //adjust the PWM
  holdingRegs[0] = valeur;
  //analogWrite(LED, holdingRegs[PWM_VAL]>>2); // constrain adc value from the arduino
  //master to 255
  holdingRegs[2]=digitalRead(14);
  memo=holdingRegs[1];

  digitalWrite(LED,memo);
  /* Note:
     The use of the enum instruction is not needed. You could set a maximum allowable
     size for holdinRegs[] by defining HOLDING_REGS_SIZE using a constant and then access
     holdingRegs[] by "Index" addressing.
     I.e.
     i.e.
     holdingRegs[0] = analogRead(A0);
     analogWrite(LED, holdingRegs[1]/4);
  */

}

State etape0() {
  valeur=6666;
  if (VALEUR.Timeout(200)) VALEUR.Set(etape1);
}
State etape1() {
  valeur=2222;
  if (VALEUR.Timeout(200)) VALEUR.Set(etape0);
}
```

2-3 The HMI as SLAVE1:

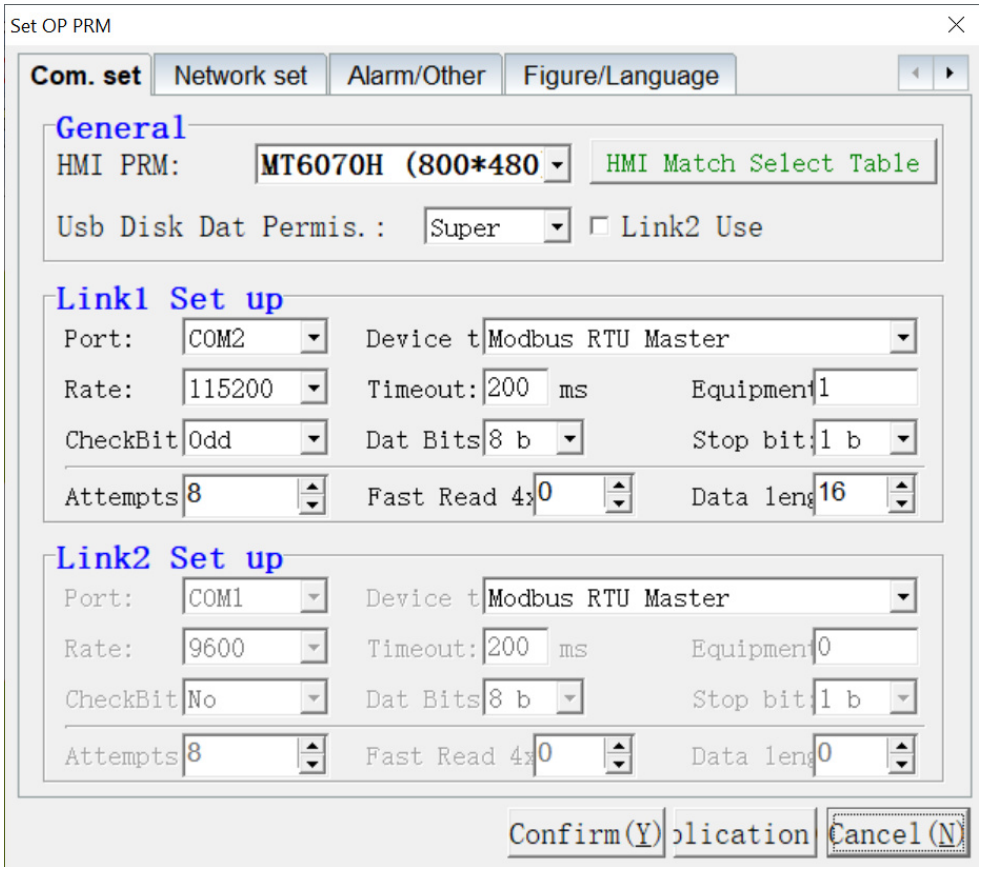
An example of a dashboard:

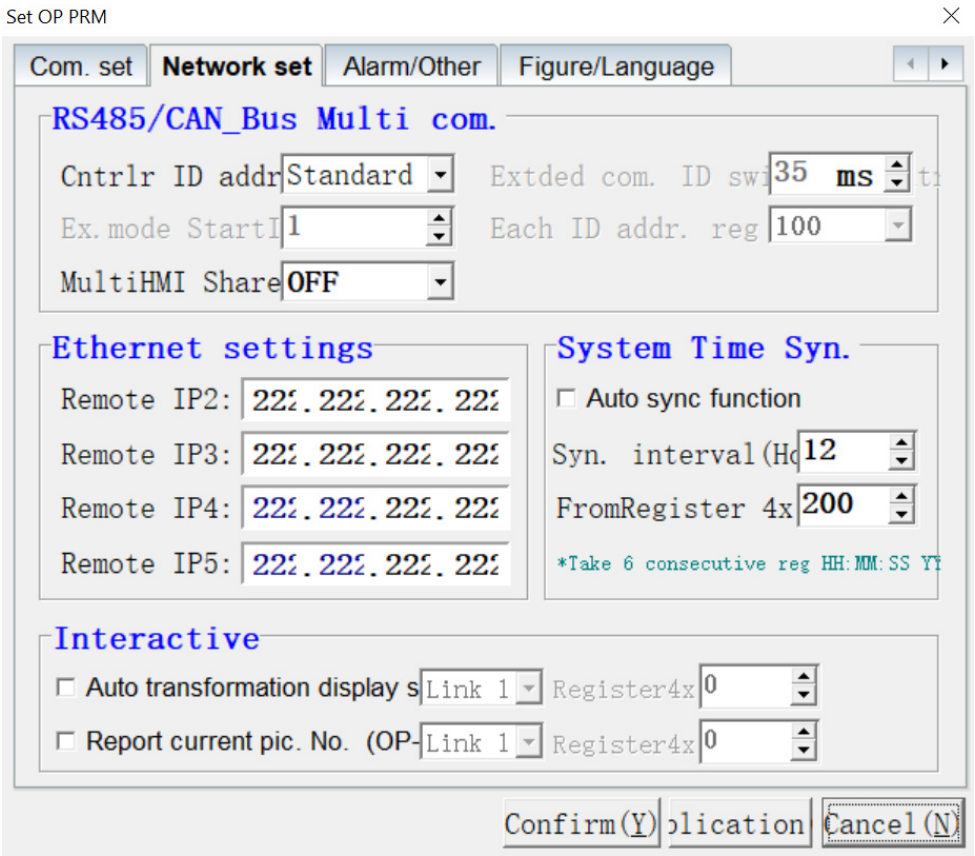


Supervision for the MASTER

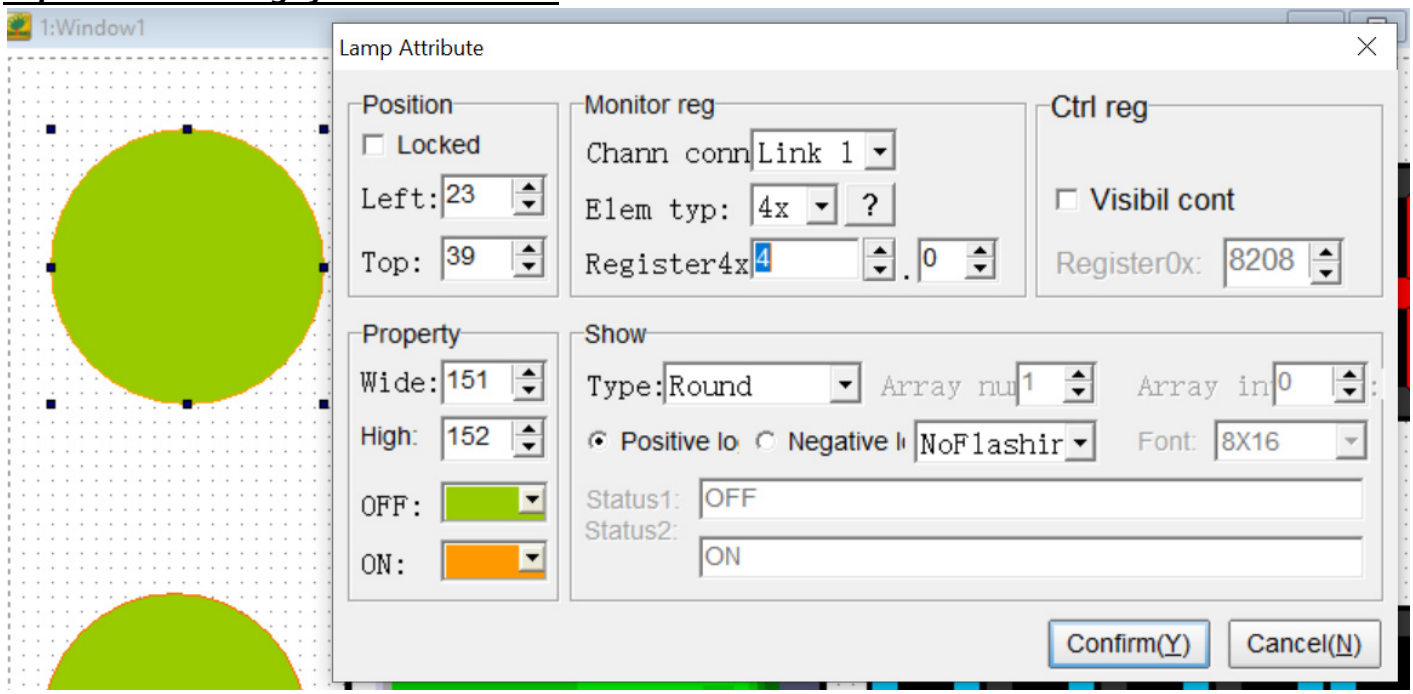
Supervision for the SLAVE3

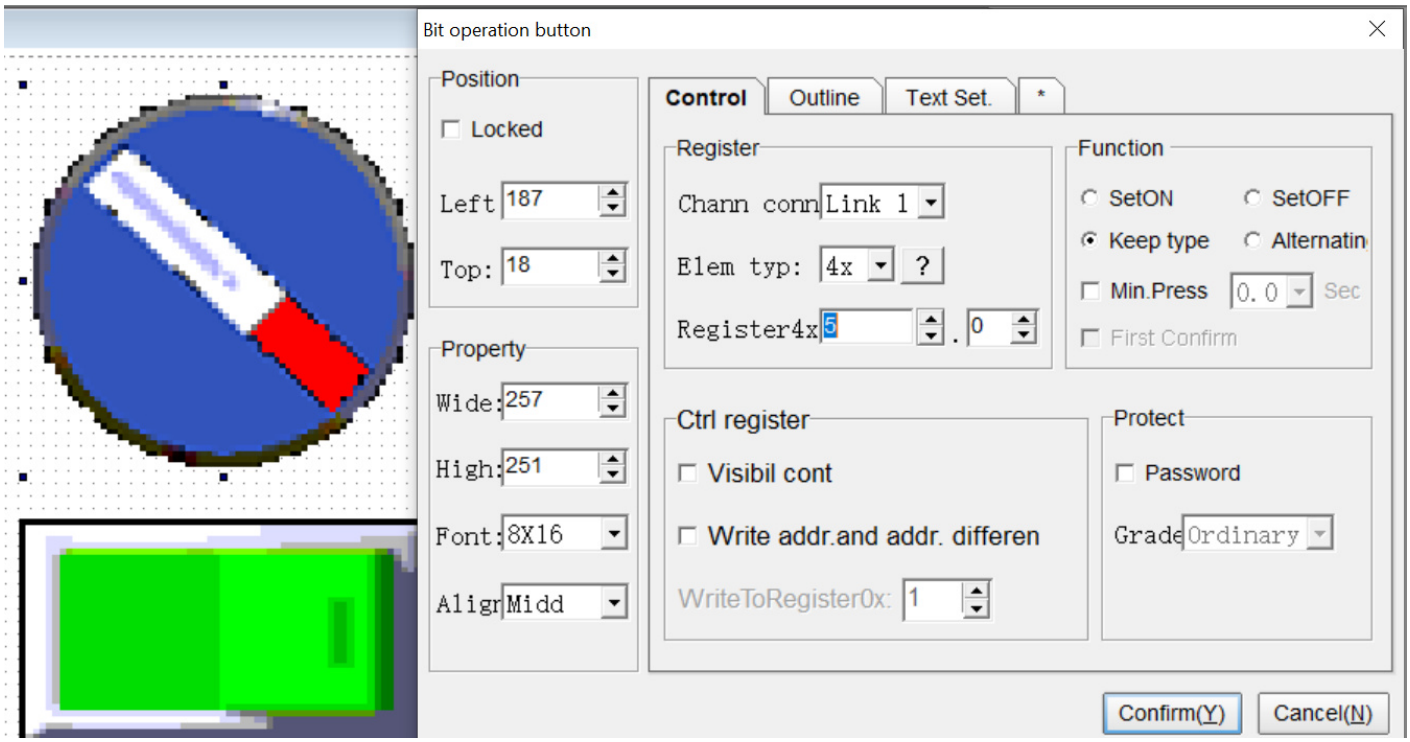
Communication settings:

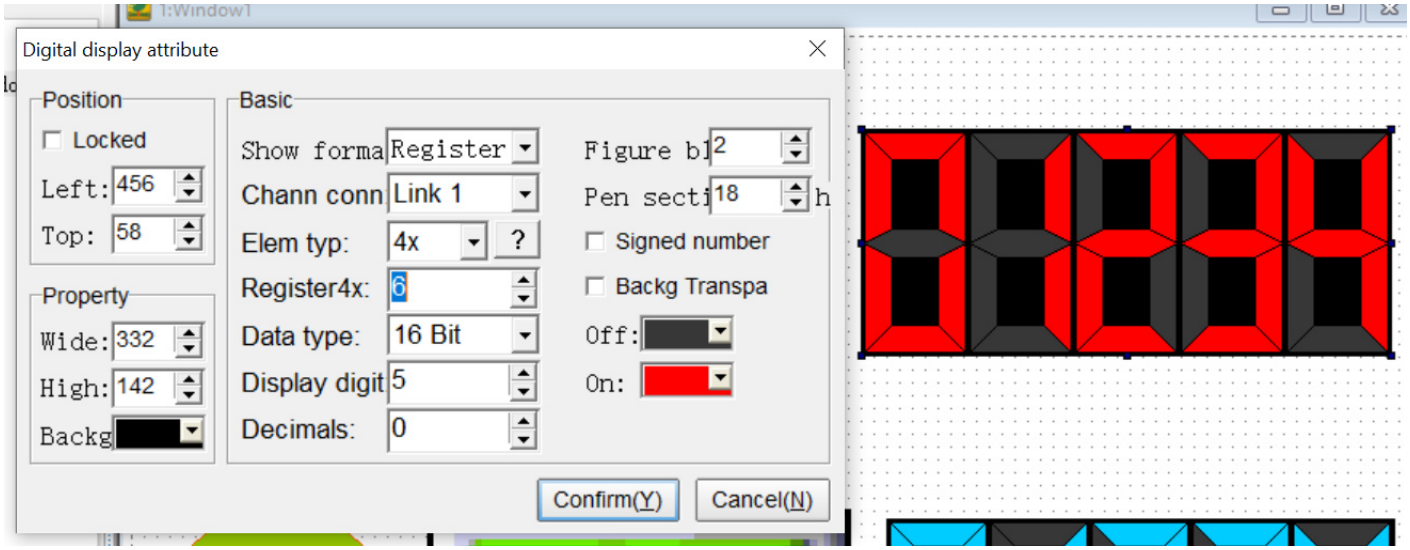




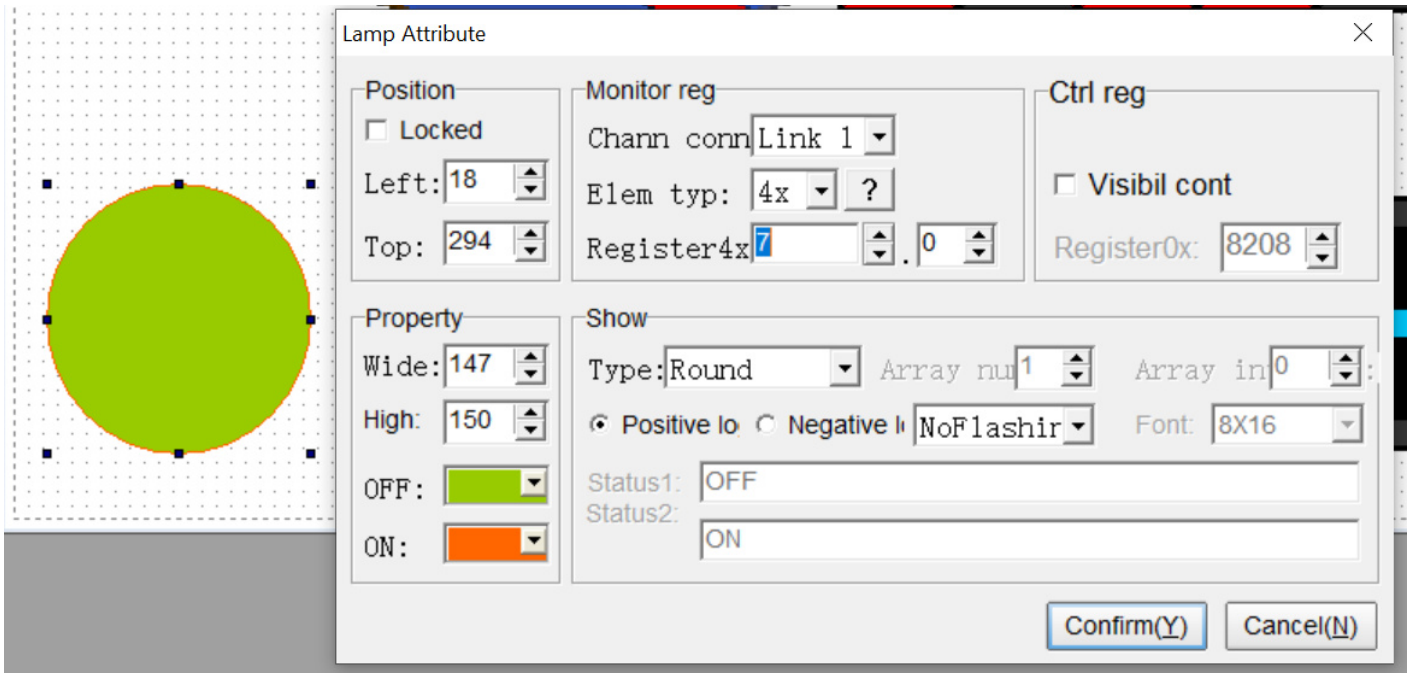
Supervision settings for the MASTER:

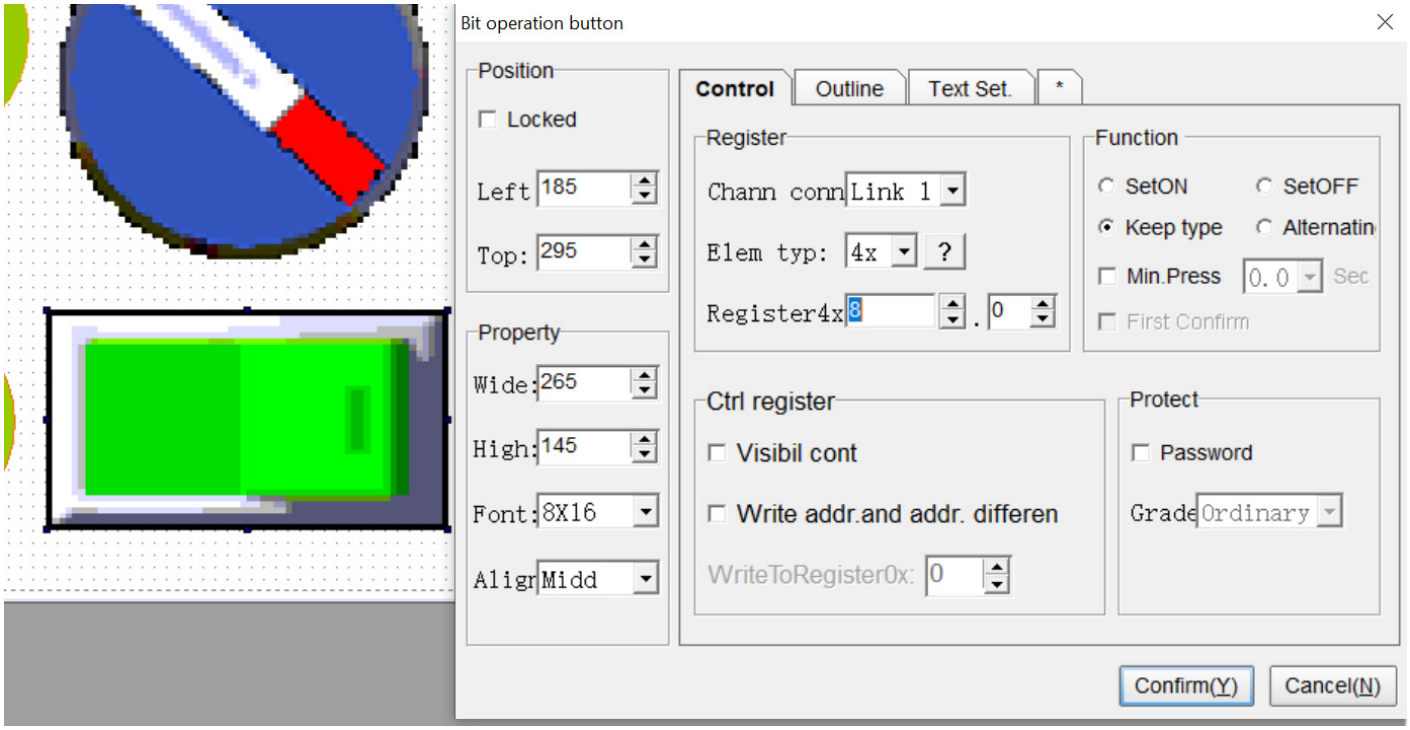


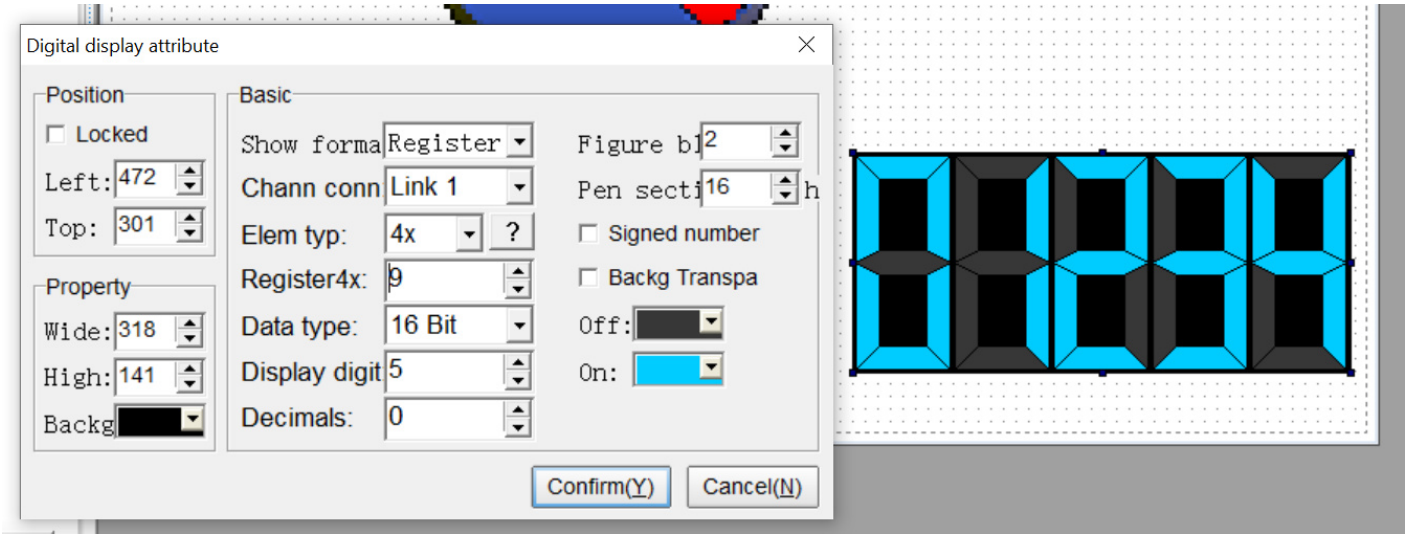




Supervision settings for the SLAVE3:




Bit operation button
Position
Locked
Left 185
Top: 295
Property
Wide: 265
High: 145
Font: 8X16
Align Midd
Control
Outline
Text Set.
Register
Chann conn Link 1
Elem typ: 4x
?
Register4x: 8
0
Function
SetON
SetOFF
Keep type
Alternatin
Min.Press
0.0
Sec
First Confirm
Ctrl register
Visibil cont
Write addr.and addr. differen
WriteToRegister0x: 0
Protect
Password
Grade Ordinary
Confirm(Y)
Cancel(N)


Digital display attribute
Position
Locked
Left: 472
Top: 301
Property
Wide: 318
High: 141
Backg
Basic
Show forma Register
Chann conn Link 1
Elem typ: 4x
?
Register4x: 9
Data type: 16 Bit
Display digit 5
Decimals: 0
Figure b 2
Pen secti 16
h
Signed number
Backg Transpa
Off:
On:
Confirm(Y)
Cancel(N)