SnapModbus Reference manual

Summary

Introduction

SnapModbus is an Open-Source multi-platform suite (library and tools) to manage Modbus communication in strict adherence to the modbus.org specifications, both master and slave side. It supports all the functions described in the documents:

- MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3
- MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b
- MODBUS over Serial Line - Specification and Implementation Guide V1.02

through the standard transport protocols and the most widespread unofficial protocols.

It’s released under lglp v3, so it’s free also for commercial use.

Why SnapModbus ?

The Modbus protocol has been consolidated for many years of use, there are millions of devices in the world that implement it and, given that it is a protocol with public and well documented specifications, there are dozens of excellent libraries that implement it, both free and commercial.

So, the question is: did we need any more libraries? How are they different?

The main purpose of this suite is to **greatly simplify** the commissioning of a Modbus system, Generally, the communication libraries focus only on the way of transferring data to and from the peripheral, limiting themselves to the syntax and showing an "educational" aspect where all the devices are homogeneous.

The reality is quite different, we often find ourselves faced with a "field" made up of various peripherals of different technology (Ethernet or serial) which have different response times and whose data need to be updated with different methods and times. The challenging work **is not** how to exchange data with one device, but how to exchange data with **all of them** efficiently.

SnapMB's goal (which I hope it has achieved) is to manage a complex field as easily and efficiently as possible. This through the abstraction of the transfer protocol (a TCP or RTU client are the same object and its behavior can be changed on the fly) and the availability of two architecture models:

- Protocol-independent half-duplex
- Fully parallel multithreaded management.

Many wrappers (library interfaces) and examples for the main high-level programming languages are provided, **only a basic knowledge of your own programming language is required**.

For an "instant use", in the distribution, are already present the compiled tools and the binary libraries for the OS which don't have c++ compiler by default.

> Info
> In accordance with the latest trend of fairness and inclusion, some key terms have been changed. **Master** and **Slave** become **Controller** (or **Client**) and **Device**; **Blacklist** and **Whitelist** become **Blocklist** and **Allowlist**. You will find this terminology both in the documentation and in the source code. It's a small mental effort against a great cultural advantage, so thanks for your patience.

Highlights

- Complete adherence to Modbus specifications and complete implementation of functions
- Can read up to 65535 registers/bit with a single call.
  The Client automatically splits calls when the number of elements is greater than allowed by the protocol.
- Fully thread-safe
  Allows the creation of multiple Devices (with different IP/Port) working with same shared areas. See **here**
- Optimized procedures for serial management
- Smart Connect
  During the TCP connection, a ping is performed first to avoid the TCP timeout.
- Intelligent timeout management
  Since in Modbus protocol, a "non answer" is itself an answer, it's important to set a correct timeout. SnapMB allows self-learning of the response times of each device and can automatically set the timeout to double the average times.
- Data consistency is always guaranteed both at the device and application level (see **here**).
- Advanced debugging and raw transfer functions that allow the experimentation and implementation of particular software using few lines of code (gateways or protocol converters)
- 20 parameters allow fine tuning.

Specifications

Environment	
Architecture	Native Intel/ARM - 32/64 bit
Supported OS	• Windows • Linux • FreeBSD • macOS

Environment	
Datalink	• Ethernet • Serial (RS232/442/485)
Transport protocols	• TCP • UDP • RTU Over TCP • RTU Over UDP • RTU • ASCII
Modbus Functions	All listed into **MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3**

> Note
> **UDP**, **RTU Over TCP** and **RTU Over UDP** are not officially supported by Modbus specifications

Test platforms used

OS/Version	Toolchain used	OS Model	App Tested
Windows 7 Pro	Visual Studio/MinGW (*)	Win64	32/64 bit
Windows 10 Pro	Visual Studio/MinGW (*)	Win64	32/64 bit
Windows 11 Pro	Visual Studio/MinGW (*)	Win64	32/64 bit
Linux (Intel) Mint 21.00 (5.15.0-43)	g++ 11.3.0	x86_64	64 bit
Linux (Intel) Ubuntu 21.10	g++ 11.2.0	x86_64	64 bit
Linux (ARM) Raspbian 32 bit	g++ 10.2.1	arm32	32 bit
Linux (ARM) Raspberry OS	g++ 10.2.1	aarch64	64 bit
FreeBSD (Intel) 13.1	g++ 11.3.0	amd64	64 bit
macOS (Intel) Catalina	g++ (clang 14.0.0)	x86_64	64 bit
macOS (Intel) Monterey	g++ (clang 14.0.0)	x86_64	64 bit
macOS (Intel) Ventura	g++ (clang 14.0.0)	x86_64	64 bit
macOS (Apple M2) Monterey	g++ (clang 14.0.0)	arm64	64 bit
macOS (Apple M2) Ventura	g++ (clang 14.0.0)	arm64	64 bit

(*) *Visual Studio 2022 Community Edition / TDM GCC 64 (g++ 10.3.0)*

Architecture

In SnapModbus there are three Objects

- The **Field Controller** and the **Client**, aka Masters, which are the active actors, they manage the communication querying the peripherals.
  - The Field Controller manages the whole Field in Half-Duplex way.
  - The clients are peer-to-peer, each of them can manage only one device but they do it independently, that is, in a multithreaded architecture they can work in parallel.
- The **Device**, aka Slave, which is the passive actor, it serves the queries that came from the active actor.

> Note
> The **Field** is a virtual Bus, i.e., a set of heterogeneous peripherals: RS232/485 with different speeds and formats (RTU or ASCII) and Ethernet, which are managed by the same Controller. The physical bus, on the other hand, provides that all the peripherals are homogeneous, i.e., this happens with Modbus RTU/ASCII.

Client/Controller

Using the Field Controller

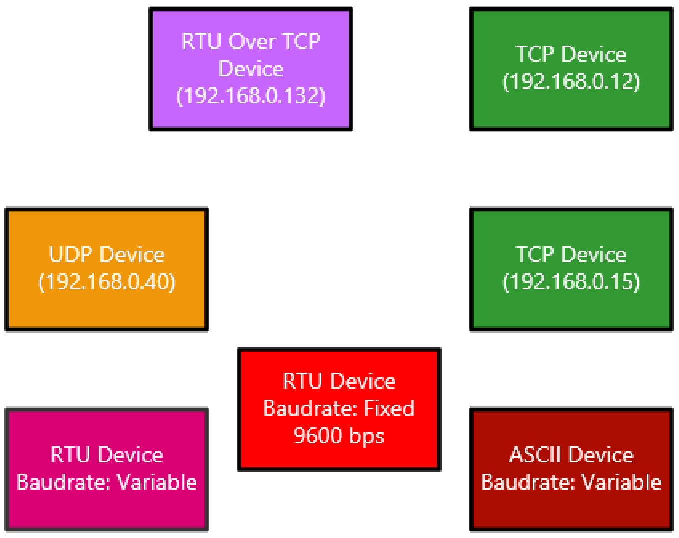
The Field Controller works in Half-Duplex mode, i.e. it exchanges data with one Device at a time in a sequential manner. Send the request, wait for the response and, only then, can move on to the next Device. Half-duplex management is not the most efficient, but the great advantage of the Field Controller is that of managing all the Devices present, of whatever type they are.

To use the Field Controller three steps are needed:

1. List of Devices and definition of hardware resources.
2. Standardization of addresses (Device ID)
3. Creation of the controller and device descriptors.

Let's look at a practical case study.

Suppose we have a set of Devices to manage as in the figure.


RTU Over TCP Device (192.168.0.132)
TCP Device (192.168.0.12)
UDP Device (192.168.0.40)
TCP Device (192.168.0.15)
RTU Device Baudrate: Fixed 9600 bps
RTU Device Baudrate: Variable
ASCII Device Baudrate: Variable

As we can see, they are very different from each other.

Ethernet Devices are of little concern to us because, even if they have different transport protocols, from a hardware point of view they will be connected to the same LAN segment.

Among the serial Devices, however, there is one that has a non-variable speed, and therefore we will have to make some considerations.

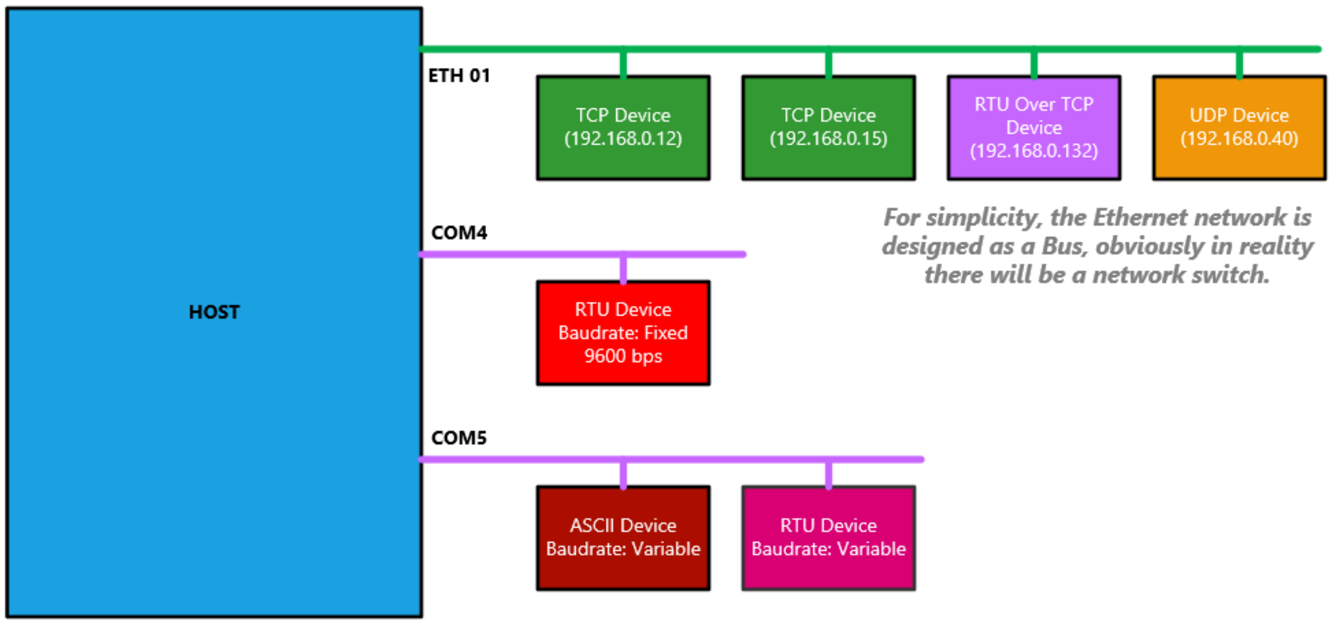
Hardware resources definition

Now we have to make a decision. To save hardware we can use only one serial port, but in this case, we have to **degrade** the speed of the other Devices to 9600 bps, or, we want to benefit from a higher speed and use a second serial adapter.

Suppose we choose the second option and therefore have two independent serial ports in our host, the first at 9600 bps to manage **RTU Device slow** and the second at 115200 bps to manage **RTU Device fast** and **ASCII Device fast**. The serial ports can be physical, which is very unusual today, or constituted by USB/RS485 adapters.

So, in conclusion, to manage our field, there will be 3 communication channels, one Ethernet and two RS485.

This will be our hardware configuration.


HOST
ETH 01
TCP Device (192.168.0.12)
TCP Device (192.168.0.15)
RTU Over TCP Device (192.168.0.132)
UDP Device (192.168.0.40)
For simplicity, the Ethernet network is designed as a Bus, obviously in reality there will be a network switch.
COM4
RTU Device Baudrate: Fixed 9600 bps
COM5
ASCII Device Baudrate: Variable
RTU Device Baudrate: Variable

Addressess standardization

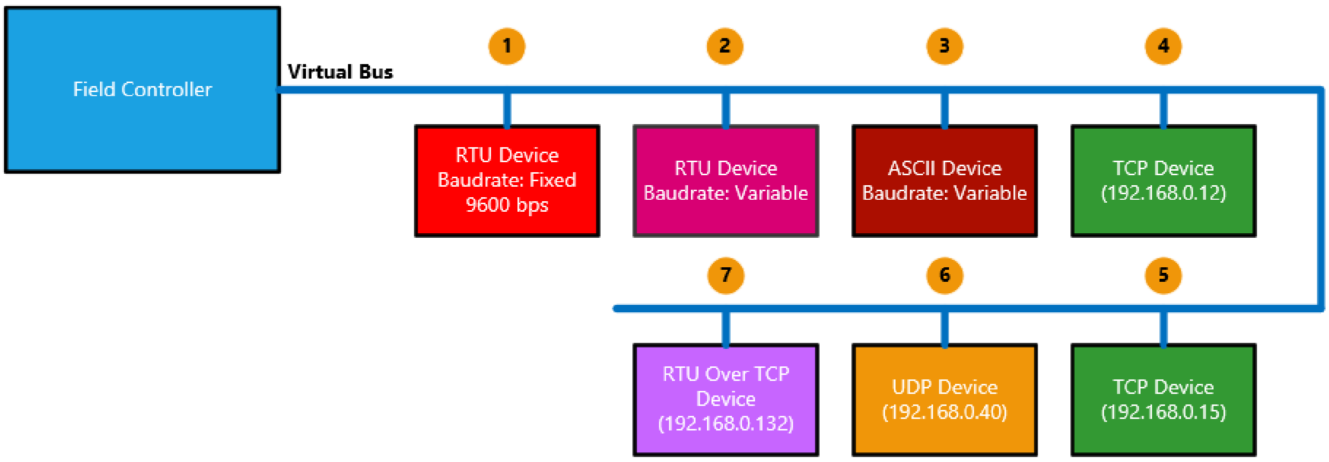
In order to work, the Field Controller, needs that the Devices Addresses (ID) must be unique, this because it holds a virtual bus. So, we need to assign an ID also to the network Devices (into our program of course, not physically).

Device	ID	Physical port	Parameters	Proto/Format
RTU Device slow	1	COM4	9600, E, 8, 1	RTU
RTU Device fast	2	COM5	115200, E, 8, 1	RTU
ASCII Device	3	COM5	115200, E, 8, 1	ASCII
TCP Device 1	4	ETH 01	192.168.0.12:502	TCP
TCP Device 2	5	ETH 01	192.168.0.15:502	TCP

Device	ID	Physical port	Parameters	Proto/Format
UDP Device	6	ETH 01	192.168.0.40:502	UDP
RTU Over TCP Device	7	ETH 01	192.168.0.132:502	RTU Over TCP

> ✎ Tip
> It’s more convenient assign low addresses to serial devices, since they must be set explicitly via hardware (or software configuration) and because some old Devices have a limited addressing range. For network devices, the ID is only a logical Index.

According to the previous table, this will be our field configuration.


Field Controller
Virtual Bus
1
2
3
4
RTU Device Baudrate: Fixed 9600 bps
RTU Device Baudrate: Variable
ASCII Device Baudrate: Variable
TCP Device (192.168.0.12)
7
6
5
RTU Over TCP Device (192.168.0.132)
UDP Device (192.168.0.40)
TCP Device (192.168.0.15)

Implementation and use

Now we are ready to implement our Field Controller in a very simply way, note, the “Broker”, inside the library, is the ancestor class of the Field Controller and the Client.

> ✎ Tip
> If we have many devices, a good practice could be to assign a mnemonic name to the Device ID using an integer constant, in order to avoid confusion when we write the data exchange code.

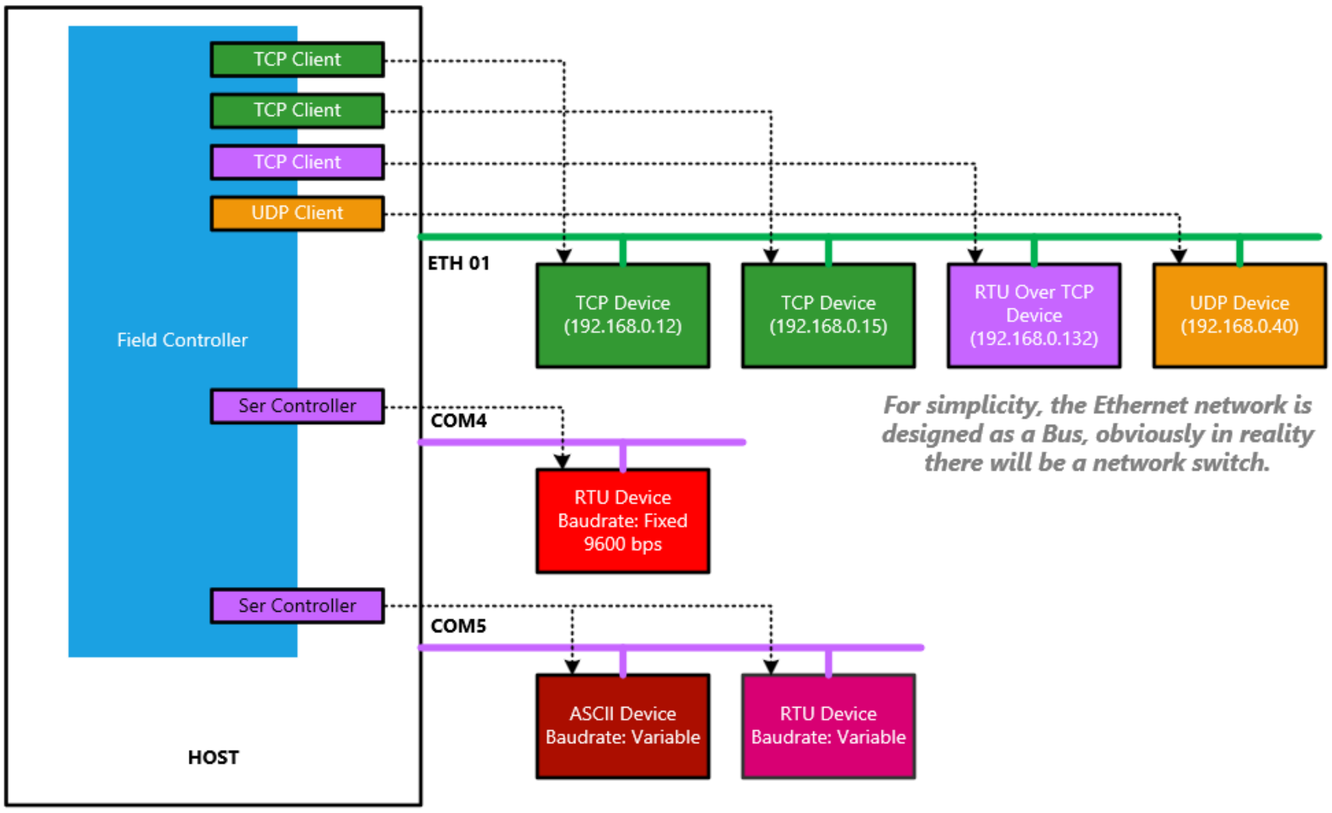
C# example

```
static SnapMBBroker Controller;

// Creates the FieldController
Controller = new SnapMBBroker();
// Setup
Controller.AddDevice(MBConsts.FormatRTU, 1, "COM4", 9600, 'N', 8, 1, MBConsts.FlowNONE);
Controller.AddDevice(MBConsts.FormatRTU, 2, "COM5", 115200, 'E', 8, 1,
MBConsts.FlowNONE);
Controller.AddDevice(MBConsts.FormatASC, 3, "COM5", 115200, 'E', 8, 1,
MBConsts.FlowNONE);
Controller.AddDevice(MBConsts.ProtoTCP, 4, "192.168.0.12", 502);
Controller.AddDevice(MBConsts.ProtoTCP, 5, "192.168.0.15", 502);
Controller.AddDevice(MBConsts.ProtoUDP, 6, "192.168.0.40", 502);
Controller.AddDevice(MBConsts.ProtoRTUOverTCP, 7, "192.168.0.132", 502);
// Read Data
Controller.ReadHoldingRegisters(1, 12, Amount, Device_1_Regs); // Read From RTU Device
(1)
Controller.ReadHoldingRegisters(5, 1, Amount, Device_5_Regs);  // Read From TCP Device
(5)
```

Technical insight

The Field Controller is a pass-through class container, it allocates a Network Client for each Network Device and allocates a Serial Controller (an internal class) for each "group" of serial devices which have the same Comport Name (however, a verification on the remaining parameters such as Baud rate, Data Bits etc. is made) Since it inherits from the "Broker", it has all methods mapped onto the Modbus function, so, when a request is made (which contains the Device ID), it will use the ID to index the internal array containing the right Client (or Controller) and it will pass the request to it.

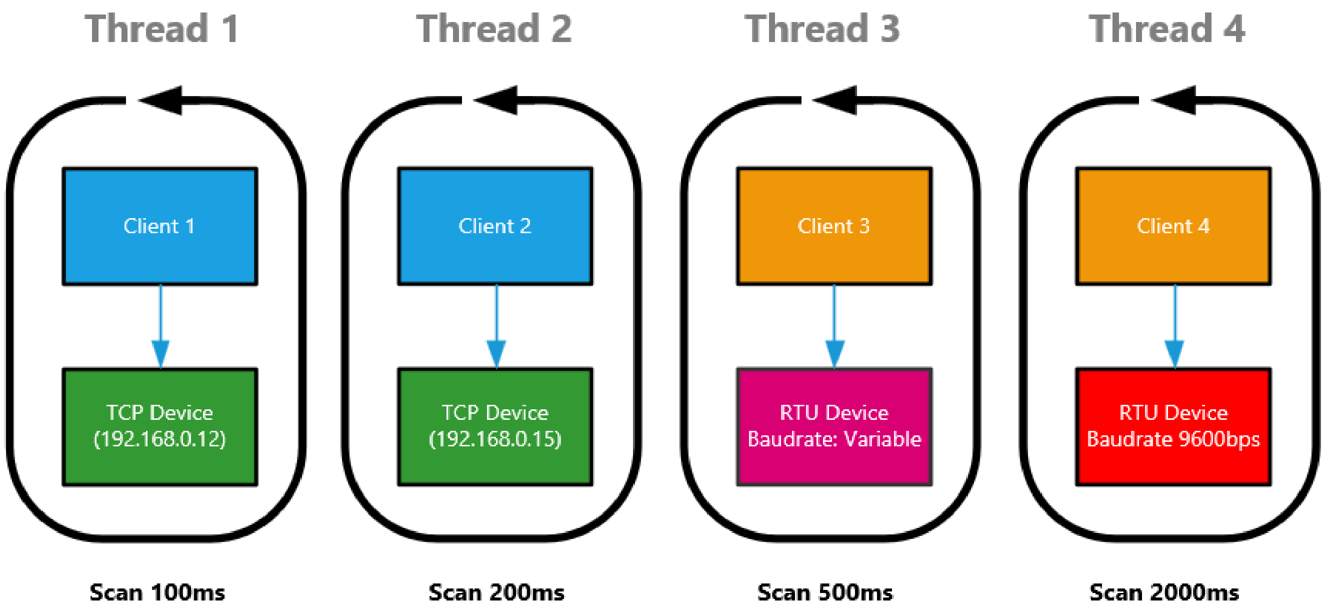

TCP Client
TCP Client
TCP Client
UDP Client
Field Controller
Ser Controller
Ser Controller
HOST
ETH 01
TCP Device (192.168.0.12)
TCP Device (192.168.0.15)
RTU Over TCP Device (192.168.0.132)
UDP Device (192.168.0.40)
For simplicity, the Ethernet network is designed as a Bus, obviously in reality there will be a network switch.
COM4
RTU Device Baudrate: Fixed 9600 bps
COM5
ASCII Device Baudrate: Variable
RTU Device Baudrate: Variable

Using the Clients

Unlike the controller, a clients establish a peer-to-peer connection with the devices. Each Client communicates with only one Device but does so completely independently from the others. This allows us to create a multithreaded architecture

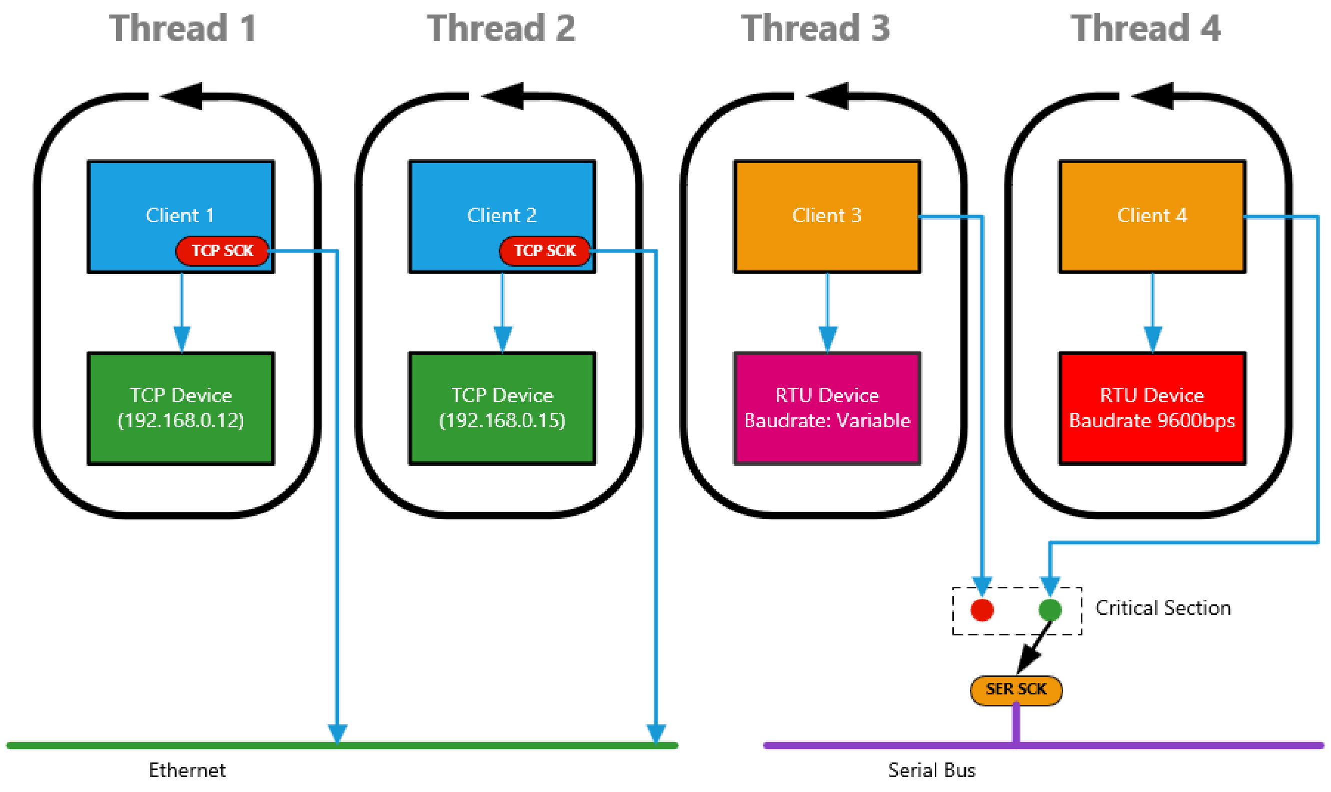
> ✎ Info
> All SnapModbus Objects are thread-safe

This type of architecture is very useful when the Devices need different scanning times, for example a Device that acquires the average air consumption can be interrogated every 2000ms, while the temperature of a thermoregulator could be collected every 500ms and, perhaps, the digital I/O of a palletizer need to be refreshed as soon as possible. As shown in the figure, we will create a thread that will manage its Client at regular intervals, independently from the others.


Thread 1
Thread 2
Thread 3
Thread 4
Client 1
Client 2
Client 3
Client 4
TCP Device (192.168.0.12)
TCP Device (192.168.0.15)
RTU Device Baudrate: Variable
RTU Device Baudrate 9600bps
Scan 100ms
Scan 200ms
Scan 500ms
Scan 2000ms

Using a sequential approach, we would have to have a refresh clock as high as the fastest Device and create some internal dividers to refresh the slower Devices, and this is quite annoying from a programming point of view. It is also not very efficient because, for example, Ethernet Devices, which can work simultaneously, would be refreshed one by one. Now, while we are used to using TCP Clients in multithreaded environments (each Client has its own socket), **it is strange to think of independent serial clients when the communication channel is unique and cannot be shared.**

The client management mechanism is shown in the next figure.



Each TCP (or UDP) client has its own socket that accesses the network independently from the others. Instead, when we create a serial client, it requests to the ChannelsManager (an internal class) a serial socket with the required characteristics (com port, speed, etc.). The ChannelsManager, if such socket does not exist, creates it, and returns its instance to the client, otherwise it returns the instance of an already created socket.

Each Client, when it must carry out a transaction (send request - wait for response), locks the serial socket, which will put any other clients "on hold" who want to access the Bus. At the end of the transaction the bus will be assigned to another client that was waiting.

This methodology allows us to unify the management of clients by greatly simplifying our code.

Conversely, during destruction, the serial manager manages a reference count for each serial socket.

Finally, if we realize that the serial bus is congested because we have many devices, we can install a second serial adapter and change **only and exclusively** the identifier of the com port in our clients.

Obviously, multithreading is a "feature", this means that it is possible to use the clients even refreshing them sequentially in the same thread without problems.

> ✎ Tip
> If we install multiple serial adapters, it is good practice to balance the workload, i.e., assign clients to adapters so that the percentage of bus usage times is equally distributed.

Concurrency

As mentioned, all SnapMB objects are thread-safe, this means you can use the same client in two different threads.
This is not a good practice, however, if you need to write "on demand" to another thread and don't want to allocate a new client, you can use this technique.

Generally the best strategy is to have one client per thread, but sometimes this is not possible because some TCP Devices, unlike SnapMB devices, cannot handle multiple connections.

Broadcast

Broadcast is the ability to send a function to all devices simultaneously. There are some key points:

1. This only makes sense for Modbus RTU where a Controller addresses its Devices at the protocol level (using the Device ID), in Modbus/TCP and its derivatives (UDP, RTU Over TCP and RTU Over UDP) the communication is peer- to-peer, so the devices are addressed using the underlying network protocol (IP address).

2. It can be used **only** for write functions.

3. Since the protocol is half-duplex, Devices receiving a Broadcast function will not respond.

> ⚠ Warning
> The broadcast is **unsafe** because:
>
> 1. Since there is no response from the Devices, we don't know if something was wrong.
> 2. Devices can be very different from each other, so we are not sure about their resource limits (max registers/coil indexes).

The FieldController broadcasts all write functions with DeviceID = 0 to all serial devices, regardless of their physical adapter.

Device

Architecture

Similarly to the objects already seen, Client and Controller, even the Devices are polymorphic, i.e. there is only one Device which can be Ethernet or Serial and work with all the supported transport protocols.

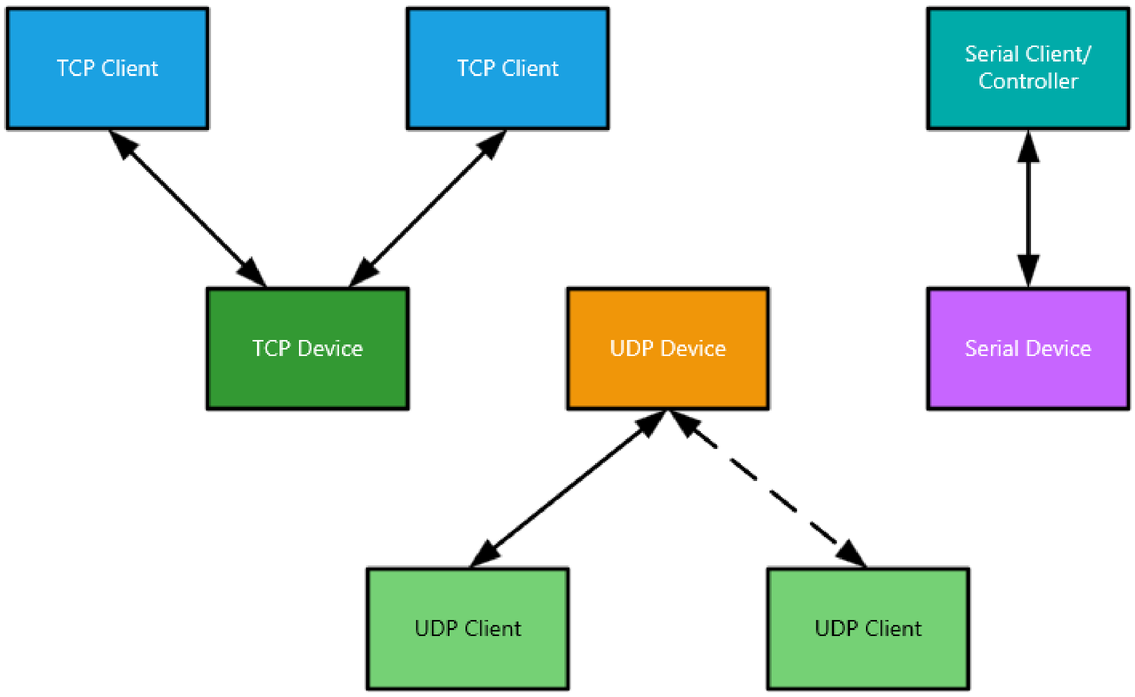
However, due to different technologies, their behavior is different.

The TCP Device (which handles TCP and RTU Over TCP transports) is fully multithreaded and can serve multiple clients simultaneously. It is a full-fledged TCP server that creates a socket associated with a thread for each connection, which independently manages the requests.

The UDP Device (which manages UDP transports and RTU Over UDP) has a single thread but can service and arbitrate the requests of multiple UDP clients.

Both TCP and UDP Devices have a PeerList, which can be **AllowList** or **BlockList**, in which it is possible to store the IP addresses that can access or those whose consent is denied.

Finally, the Serial Device (which handles RTU and ASCII formats) has a single thread and handles requests from a single Client or Controller, since there is no concept of master address on the serial line.



User program Interface

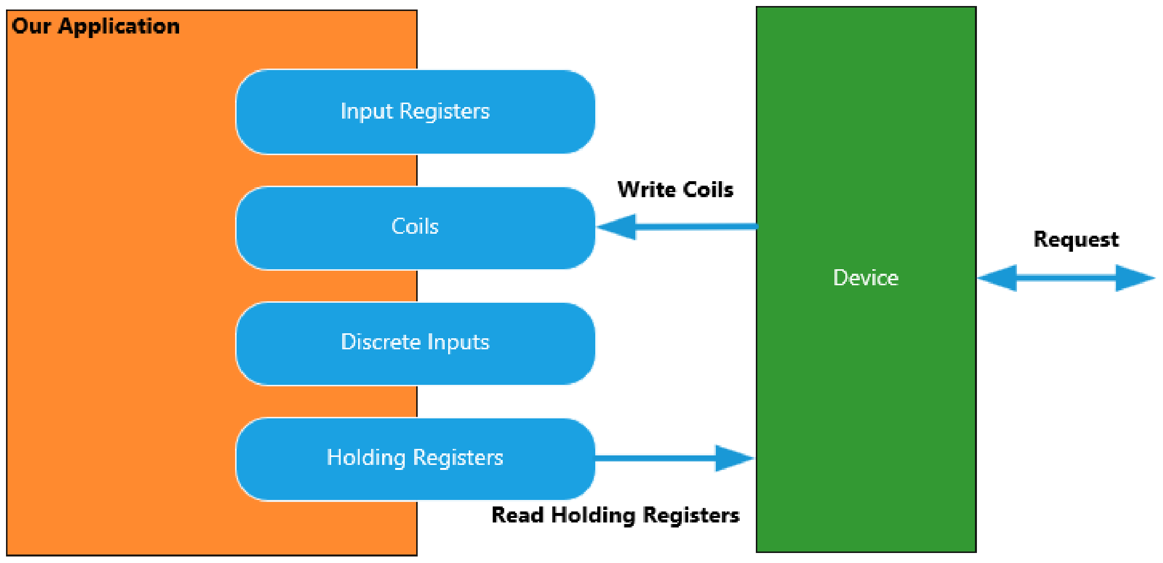
The Device is a "request handler" i.e., an object that replies to the Client/Controller request. All this cannot happen in a completely autonomous way, it is necessary that our application is aware of this data exchange and that it can modify its behavior accordingly.

The Device, once created, communicates with our application through **shared resources** and/or **callbacks**.

Shared resources

This is the most intuitive method, we allocate four memory areas in our application (struct or Array) and tell the Device: these are your **Holding registers**, **Coils**, **Input registers** and **Discrete inputs**; when the Controller requests to read or write something, you must use these areas.

> ✎ Tip
> It is not necessary to allocate all resources, we may only want to work with Coils and Holding Registers, for example.



Concurrency

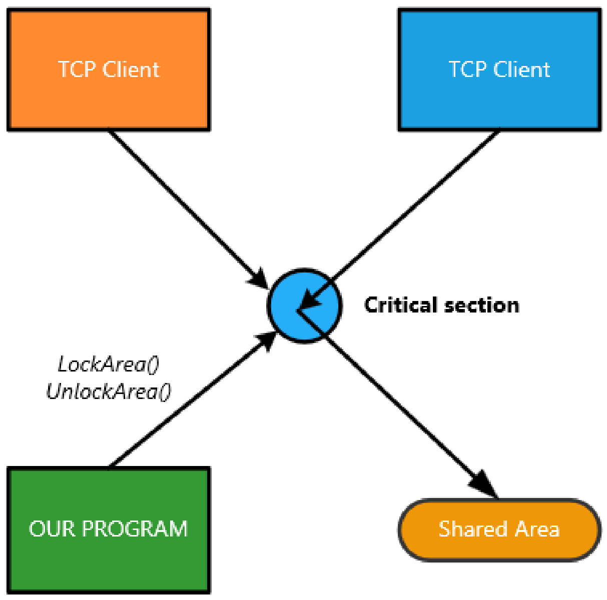
The memory areas exist in our application, so we can access them at any time, however, since we do not know when the Device reads or writes to them, it is necessary to synchronize access to avoid reading "partial data".

To do this, the Device provides **Lock/Unlock** and **Safe Copy Area** methods that allow you to lock the memory area before accessing it.

> Note
> The lock/unlock mechanism for shared areas is **at the application level**, i.e. if you create two devices associated with two different addresses/ports, which work on the same shared areas, data consistency is valid for both the devices simultaneously.

If the logic of our program is like that of a PLC, that is cyclical sequential, it could be useful to work with a double buffer, i.e., mirror memory areas in which we can transfer the data to work on.

Concurrent access of multiple socket threads is also governed by a critical section, and this happens automatically.



Finally, if we need to know when a device has written/read data in a certain area, we can use the Event Callback (see below).

After the creation, to share a resource, you need of only one line of code:

```
Device.RegisterArea(<Area type>, <Area reference>, <Area Size>);
```

Example

```
// C#
Device.RegisterArea(MBConsts.mbaHoldingRegisters, ref HoldingRegisters, regs_amount);
```

```
// C++
Device->RegisterArea(mbaHoldingRegisters, &HoldingRegisters, regs_amount);
```

```
/* C */
device_RegisterArea(Device, mbaHoldingRegisters, &HoldingRegisters, regs_amount);
```

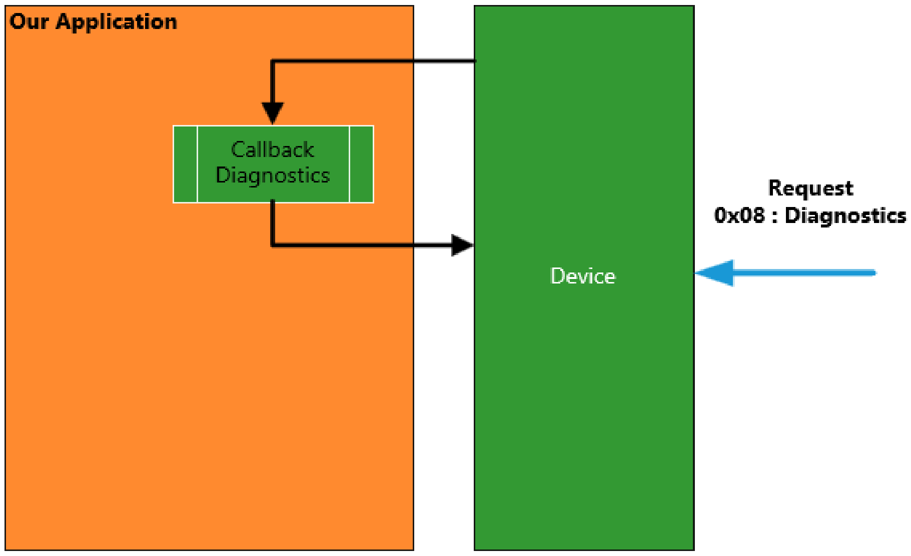
```
// Object Pascal
Device.RegisterArea(mbaHoldingRegisters, @HoldingRegisters, regs_amount);
```

> Note
> In C# to prevent the garbage collector action, which is not aware that a memory area is accessed from unmanaged code, the area is automatically "pinned" via a GCHandle. Have a look at **RegisterArea()** into SnapMB.net.cs

Callbacks

Callbacks are functions (implemented as Delegates in C#) in our code which are called by the Device when something happens. And usually, they are associated to a Modbus functions.

In other words, we can say to the Device: *when you receive a request "0x08: Diagnostics" you must call this function.*



Within the callback we have access to the information the client wants to write, or we can respond with the information it has asked to read.

There are 3 callback not associated to Modbus functions:

- **Events** (see below)
- **PacketLog**, which makes received and sent telegrams available in binary form.
- **Passthrough**, which can be used to write a gateway or a protocol converter.

> Warning
> the callback is executed in the device thread and its execution time determines the response time of the device towards the client, so:
>
> - Pay attention to the resources you access; they should be thread-safe.
> - Perform quick operations. For example, loading information to be sent from disk could cause the client to time out.

Also using the callbacks involve only one line of code:

```
Device.RegisterCallback(<Callback type>, <Function address>, usrPtr);
```

usrPtr is a value that the Device returns to us into the callback, it can be used to store an Object reference and can be set to NULL (nil or IntPtr.Zero) if we don't need of it.

Shared resources vs Callbacks

Here some concepts:

- To handle Registers/Bit we can use Shared resources or Callbacks
- To handle all other functions, we can **only** use Callbacks
- If a callback was not set, the Device automatically issues the answer **0x01 exception error** (that means the resource is not available / the function is not implemented)
- Register/Bit Management: if we have both Shared Resource and Callback, if the request is to write, the Shared Resources are used first, then the Callback is activated, the opposite if the operation is to read.

Log and Events

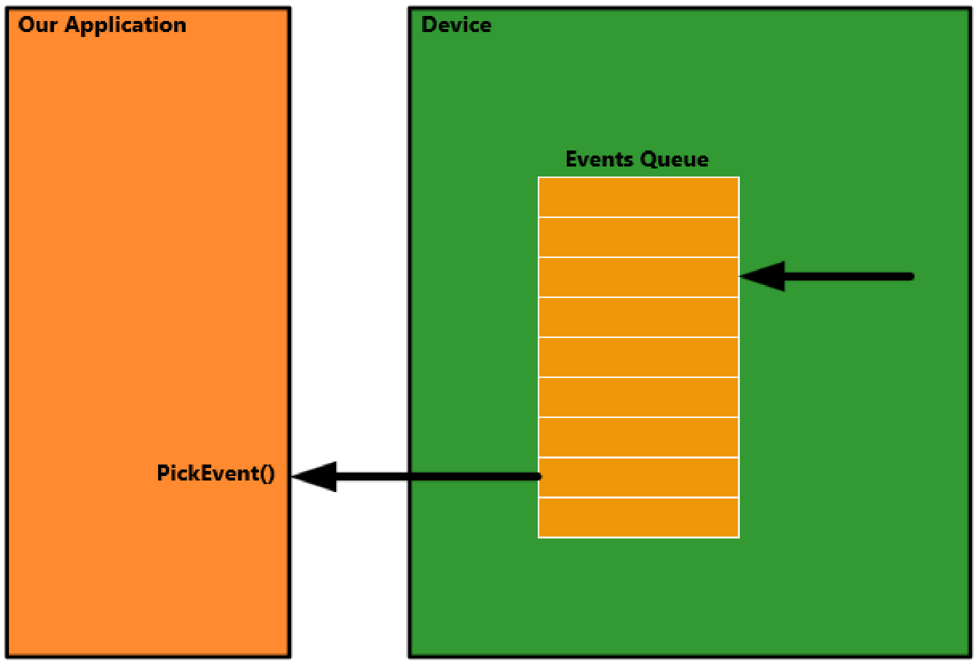
When something happens the Device create an Event (a struct containing some information), every event is inserted into an internal a circular queue and, if we set the DeviceEvent callback, it is called passing the event as parameter.

So, we can get events in two way, the first, synchronous, setting a Callback and consuming the event as soon as it is created.

The second, asynchronous, extracting in a polling cycle the event from the queue in another thread.

The first method is suggested when we want an immediate reaction to the event, knowing, for example, if a certain memory area was written.

The second for event logging, using **PickEvent(Event)** or directly **PickEventAsText(string)** which returns the textual representation of the event.



Into the examples supplied you will find both methods.

```
// Pascal WinForm example of a timer tick event which
// appends the device events into a Memo object
procedure TMainForm.timLogTimer(Sender: TObject);
var
    Message : string;
begin
    // Returns true if an event was picked, false if the queue is empty
    while Device.PickEventAsText(Message) do
        Log.Lines.Add(Message);
end;
```

Error handling

The Device autonomously manages the Modbus protocol, including its exceptions, this means that:

- If we have not shared a memory area for the Input Registers and we have not even allocated a callback to manage them, if a request to read these registers arrives, the Device, according to the specification, will respond with the exception error **0x01: Illegal function**
- If we have allocated 256 registers and an access beyond this limit is requested, the device will respond with the exception error **0x02: Illegal data address**

Gateway (Protocol Converter)

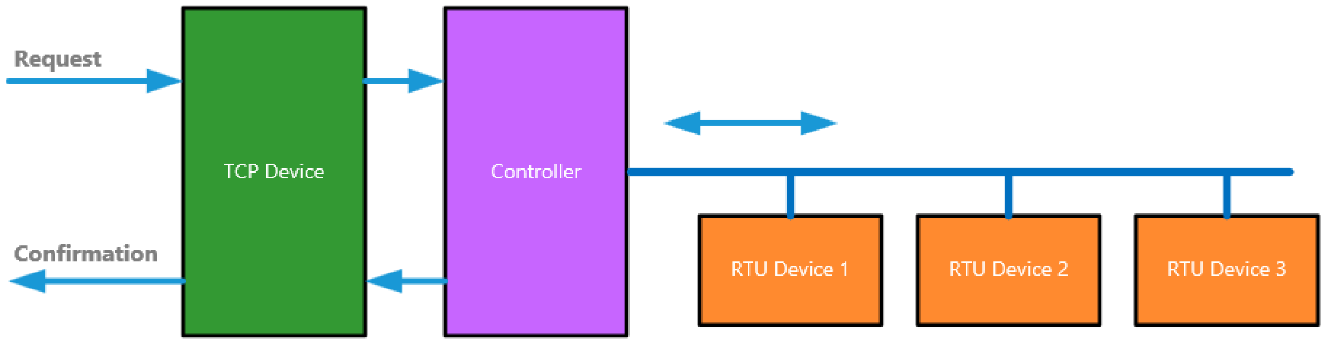
The Gateway is not a native SnapMB object, but it's very simple to implement using two special characteristics of the Device and the Controller:

- The Passthrough Event (Device)
- The RawRequest (Controller)

The gateway is a protocol converter, it contains a Device set to the source protocol and a Controller set to the target protocol.

When a request arrives, the Device relays it to the Controller which executes the request, then, the answer received by the Controller is relayed back to the Device.

The figure shows the principle diagram of a TCP/RTU Gateway, but, since our Controller manages a virtual bus, it is also possible to implement an RTU/TCP Gateway or others.



Into the examples you will also find the Gateway, however this is where the magic happens:

```
int SNAP_API PasstroughHandler(void* usrPtr, byte DeviceID, void* RxPDU, word RxPDUSize,
void* TxPDU, word& TxPDUSize)
{
    // Relay the request through the client
    int rawResult = InnerClient->RawRequest(DeviceID, RxPDU, RxPDUSize, TxPDU,
TxPDUSize, 0);

    // Converts Timeout and Hardware errors to be fully Modbus compliant
    if (rawResult != mbNoError)
    {
        // Timeout
        if ((rawResult & 0x0FFFFFFF) == 0x00050007)
            return errGatewayTargetFailed; // Target failed to respond
        // Other error (not Modbus protocol error)
        if (TxPDUSize == 0)
            return  errSlaveDeviceFailure; // Generic failure
    }
    return 0;
}
```

Serial Sniffer

It is possible to make a serial sniffer very simply in two different ways.

1. Using a passthrough event like we did for the gateway
2. Using the PacketLog event.

Both methods allow you to have access to all the telegrams that are intercepted by the Device on an RS485 line.

Api reference

SnapModbus functions are exported following **“C” calling convention**, and, since they internally are object oriented, there is always the object reference as parameter.

The Object reference is a struct defined as below:

```
typedef struct {
 uintptr_t Object;
 uintptr_t Selector;
}XOBJECT;
```

it consists of two "native integer", i.e. two integer that can be 32 or 64 bit wide, depending of way the build. **Never change it**, it must be just passed as parameter to the working functions.

Into the high-level wrappers, there are classes that incapsulate these functions, so, for example, you will not directly use the function:

```
Broker_ReadHoldingRegisters(XOBJECT &Broker, <Fun 0x03 parameters>)
```

Instead, you will call:

```
Client.ReadHoldingRegisters(<Fun 0x03 Parameters>); // C#, Object Pascal
Client->ReadHoldingRegisters(<Fun 0x03 Parameters>); // C++
```

More

Binary libraries cannot export overloaded functions, so, to create an Ethernet Client, in plain-c you must call the function:

```
void broker_CreateEthernetClient(XOBJECT& Broker, const char* Address, int Port, int
Proto);
```

High-level object-oriented wrappers only have **one class** with overloaded constructors, so the type of the object will be determined by the parameter list.

> Note
> Here, except in special cases, the functions will be listed as exported by the library.
> For high level syntax, please, refer to the wrappers

Client/Controller

Creation/Destruction

CreateFieldController()

```
void broker_CreateFieldController(XOBJECT& Broker);
```

Creates a Field Controller.

Parameter	Values
Broker	Object descriptor returned

CreateEthernetClient()

```
void broker_CreateEthernetClient(XOBJECT& Broker, int Proto, const char* Address, int
Port);
```

Creates an Ethernet Client working with a given transport protocol

Parameter	Values
Broker	Object descriptor returned
Proto	Transport Protocol
Address	Device IP Address e.g. "192.168.1.15"
Port	Device Port, usually 502

Transport protocol

Value	Protocol
0	TCP
1	UDP
2	RTU Over TCP
3	RTU Over UDP

CreateSerialClient()

```
void broker_CreateSerialClient(XOBJECT& Broker, int Format, const char* PortName, int
BaudRate, char Parity, int DataBits, int Stops, int Flow);
```

Creates a **Serial Client** working with a given Data Format

Parameter	Values
Broker	Object descriptor returned
Format	Data Format
PortName	Serial Channel Name as known to the host OS (1)
Baudrate	Speed (bps) e.g. 9600, 19200, 115200 ...
Parity	Frame parity
DataBits	Frame data bits : 7 or 8
StopBits	Frame stop bits : 1 or 2
Flow	Control Flow

(1) In Windows it will be "COM1", "COM2" and so on.. In Linux/BSD/macOS, please refer to the device list into `/dev`

Format

Value	Format
0	RTU
1	ASCII

Parity

Value	Parity
E or **e**	Even Parity
O or **o**	Odd Parity
N or **n**	No Parity

Flow

Value	Flow

Value	Flow
0	No control flow
1	Hardware control flow (RTS/CTS)

Destroy()

```
void broker_Destroy(XOBJECT& Broker);
```

Destroys the Object

Parameter	Values
Broker	Object descriptor which came from the creation function

Behaviour change

Sometime we don't know in advance the type of Client that we need, or we need to change its behavior at runtime.

While it's very easy to destroy and then recreate a client, sometimes using managed languages like C# it's better to let the garbage collector destroy our object. So there are three ChangeTo() methods which internally destroy the Client and then recreate it.

These helper methods are present **only** into the high-level wrappers, since low-level programming (like plain-c) doesn't need of them.

They are overloaded methods and have the same syntax of the constructors.

let's see the methods of C# since its syntax is halfway between C++ and Pascal.

ChangeTo()

```
// Changes the current Object to a Field Controller
public void ChangeTo()
// Changes the current Object to an Ethernet Client
public void ChangeTo(int Proto, string Address, int Port)
// Changes the current Object to a Serial Client
public void ChangeTo(int Format, string PortName, int BaudRate, char Parity, int
DataBits, int Stops, int Flow)
```

> ⚠ Warning
> Since the internal Object is destroyed, all it resources will be freed, so, if the current Object is a Field Controller, the devices list will be emptied, also if the new Object is a Field Controller itself.

Object control functions

Connect()

```
int broker_Connect(XOBJECT& Broker);
```

Connects the Broker.

Object Type	Behavior
FieldController	Calls the Connection method for all inner objects (see below). Returns **errSomeConnectionsError** if some of them failed.
TCP Client	Connects to the Device. Returns an error if the connection failed due to refusion or invalid address
UDP Client	Simply creates the UDP Socket.
Serial Client	Opens the Serial Port. If the Serial Socket is shared among other Clients and it's already opened, nothing will happen.

Note:

1. RTU Over TCP is a Transport protocol of TCP Client
2. RTU Over UDP is a Transport protocol of UDP Client

Parameter	Values
Broker	Object descriptor which came from the creation function

Return	Meaning
0	Success
Other	See the error tables

Disconnect()

```
int broker_Disconnect(XOBJECT& Broker);
```

Disconnects the Broker.

Object Type	Behavior
FieldController	Calls the Disconnection method for all inner objects (see below)
TCP Client	Performs a TCP/IP disconnection to the Device and destroys the TCP socket.
UDP Client	Simply destroys the UDP Socket.
Serial Client	Closes the Serial Port. If the Serial Socket is shared among other Clients and it's already closed, nothing will happen.

Note:

1. RTU Over TCP is a Transport protocol of TCP Client
2. RTU Over UDP is a Transport protocol of UDP Client

Parameter	Values
Broker	Object descriptor which came from the creation function

This function returns always 0 (success)

AddControllerXXXDevice()

```
int broker_AddControllerNetDevice(XOBJECT& Broker, int Proto, byte DeviceID, const char*
Address, int Port);
int broker_AddControllerSerDevice(XOBJECT& Broker, int Format, byte DeviceID, const
char* PortName, int BaudRate, char Parity, int DataBits, int Stops, int Flow);
```

Allocates a new target device into a FieldController as explained **here**
Since this involves a Client creation, the parameters are the same of its constructor.

The high-level wrappers expose this function as a single overloaded method named `AddDevice()`
The list of parameters will determinate what function has to be called.

Example

```
// This will call broker_AddControllerNetDevice()
Controller.AddDevice(MBConsts.ProtoTCP, 1, "192.168.1.10", 502);
// This will call broker_AddControllerSerDevice()
Controller.AddDevice(MBConsts.FormatRTU, 1, "COM5", 19200, 'E', 8, 1,
MBConsts.FlowNONE);
```

Return	Meaning
0	Success
Other	See the error tables

SetXXXParam()

```
int broker_SetLocalParam(XOBJECT& Broker, byte LocalID, int ParamIndex, int Value);
int broker_SetRemoteDeviceParam(XOBJECT& Broker, byte DeviceID, int ParamIndex, int
Value);
```

The **Local param** refers to a Broker's parameter which is global, i.e., it modifies the Broker behavior against all Devices managed. The **Remote Device param** is specific of **that** Device managed.

E.g.

The Send Timeout is a local param because it belongs to the Broker and is the same for the transactions with all Devices.
The Auto Timeout flag is a remote param, every Device has its own.

These are **Multiplex** functions, e.g.

`SetRemoteDeviceParam(1, par_FixedTimeout, 1500);`
Will set the Receive timeout of the Device 1 to 1500 ms.
Where par_FixedTimeout is an integer constant = 9;

`SetLocalParam(3, par_SendTimeout, 100);`
If the Broker is a FieldController, will set to 100 ms the Send Timeout of the **inner client** who manages the Device N.3

Local Param

Parameter	Values
Broker	Object descriptor which came from the creation function
LocalID	Inner client Index
ParamIndex	Parameter Index
Value	Value to set

> Note
> If the Broker is a Client (not a FieldController) the **LocalID** param is ignored

Remote Param

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Remote device Address (ID)
ParamIndex	Parameter Index
Value	Value to set

Refer **here** for the parameter list.

GetIoBufferXXX()

```
int broker_GetIOBufferPtr(XOBJECT& Broker, byte DeviceID, int BufferKind, pbyte &Data);
int broker_GetIOBuffer(XOBJECT& Broker, byte DeviceID, int BufferKind, pbyte Data);
```

After a transaction, it's possible to get the TX/RX buffer from the Broker. This is very useful for debug purpose.

`broker_GetIOBufferPtr()` Returns the pointer to the buffer choosed.
`broker_GetIOBuffer()` Directly copies the data into a given buffer.

The first is more fast because we can pass the pointer directly to a dump/storage function.
The second can be used in a managed environment (C#) where the use of pointers is tricky.

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
BufferKind	The buffer choosed
&Data	Pointer to the buffer containing the data
Data	Area pointer where the data must to be copied

BufferKind

Value	BufferKind

Value	BufferKind
0 (bkSnd)	Data sent
1 (bkRcv)	Data received

Return	Meaning
int Value	Buffer size (byte)

GetDeviceStatus()

```
int broker_GetDeviceStatus(XOBJECT& Broker, byte DeviceID, TDeviceStatus& DeviceStatus);
```

Reads the status of a given device.

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Remote device Address (ID), ignored if the Broker is a Client
DeviceStatus	Reference (Pointer to) of a TDeviceStatus struct

TDeviceStatus struct

```
typedef struct {
    int32_t LastError;
    int32_t Status;
    int32_t Connected;
    uint32_t JobTime;
}TDeviceStatus;
```

Field	Meaning
LastError	Last transaction error code (0 = success)
Status	Internal status after the last transaction
Connected	0 : Not connected; 1 : Connected
JobTime	Last transaction time (ms)

Internal status

Value	Meaning
0 : Unknown	Status unknown (maybe already created)
1 : OK	OK
2 : Timeout	Last transaction was timed out
3 : HWError	Socket or Serial port read/write error
4 : ProtoError	The device returned a Modbus error

Modbus Functions

> Note
> Please refer to **MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3** for the "Modbus" behavior of following functions

ReadCoils()

```
int broker_ReadCoils(XOBJECT& Broker, byte DeviceID, word Address, word Amount, void*
pUsrData);
```

Implementation of function `(0x01)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer
pUsrData	Pointer to transfer buffer

> Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

ReadDiscreteInputs()

```
int broker_ReadDiscreteInputs(XOBJECT& Broker, byte DeviceID, word Address, word Amount,
void* pUsrData);
```

Implementation of function `(0x02)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer

Parameter	Values
pUsrData	Pointer to transfer buffer

> ✎ Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

ReadHoldingRegisters()

```
int broker_ReadHoldingRegisters(XOBJECT& Broker, byte DeviceID, word Address, word
Amount, void* pUsrData);
```

Implementation of function `(0x03)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer
pUsrData	Pointer to transfer buffer

> ✎ Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

ReadInputRegisters()

```
int broker_ReadInputRegisters(XOBJECT& Broker, byte DeviceID, word Address, word Amount,
void* pUsrData);
```

Implementation of function `(0x04)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer
pUsrData	Pointer to transfer buffer

> Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

WriteSingleCoil()

Implementation of function `(0x05)`

```
int broker_WriteSingleCoil(XOBJECT& Broker, byte DeviceID, word Address, word Value);
```

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Value	Value to write

Return	Meaning
0	Success
Other	See the error tables

WriteMultipleRegisters()

```
int broker_WriteMultipleRegisters(XOBJECT& Broker, byte DeviceID, word Address, word
Amount, void* pUsrData);
```

Implementation of function `(0x10)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer
pUsrData	Pointer to transfer buffer

> Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

WriteSingleRegister()

```
int broker_WriteSingleRegister(XOBJECT& Broker, byte DeviceID, word Address, word
Value);
```

Implementation of function `(0x06)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Value	Value to write

Return	Meaning
0	Success
Other	See the error tables

ReadWriteMultipleRegisters()

```
int broker_ReadWriteMultipleRegisters(XOBJECT& Broker, byte DeviceID, word RDAddress,
word RDAmount, word WRAddress, word WRAmount, void* pRDUsrData, void* pWRUsrData);
```

Implementation of function `(0x17)`

Parameter	Values

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
RDAddress	Start address (first item index) to be read
RDAmount	Items number to read
WRAddress	Start address (first item index) to be write
WRAmount	Items number to write
pRDUsrData	Pointer to read buffer
pWRUsrData	Pointer to write buffer

Return	Meaning
0	Success
Other	See the error tables

WriteMultipleCoils()

```
int broker_WriteMultipleCoils(XOBJECT& Broker, byte DeviceID, word Address, word Amount,
void* pUsrData);
```

Implementation of function `(0x0F)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start address (first item index)
Amount	Items number to transfer
pUsrData	Pointer to transfer buffer

> Note
> With this function it is possible to overcome the limits of the Modbus specifications by transferring up to 65535 items.
> If the number of items is greater than the PDU size, the function automatically splits the request into several consecutive transactions.

Return	Meaning
0	Success
Other	See the error tables

MaskWriteRegister()

```
int broker_MaskWriteRegister(XOBJECT& Broker, byte DeviceID, word Address, word
AND_Mask, word OR_Mask);
```

Implementation of function `(0x16)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Register Index
AND_Mask	AND Mask
OR_Mask	OR Mask

This function code is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register.

The function’s algorithm is:

`Result = (Current_Content AND AND_Mask) OR (OR_Mask AND (NOT AND_Mask))`

Return	Meaning
0	Success
Other	See the error tables

ReadFileRecord()

```
int broker_ReadFileRecord(XOBJECT& Broker, byte DeviceID, byte RefType, word FileNumber,
word RecNumber, word RegsAmount, void* RecData);
```

Implementation of function `(0x14)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
RefType	Reference type (must be 6)
FileNumber	File Number to access
RecNumber	Record Number to read
RegsAmount	Number of registers to transfer
RecData	Pointer to Record data

> Note
> To simplify the use into the user program, I decided to avoid the use of nested structures, so this functions limits the number of record to be read to 1

Return	Meaning
0	Success
Other	See the error tables

WriteFileRecord()

```
int broker_WriteFileRecord(XOBJECT& Broker, byte DeviceID, byte RefType, word
FileNumber, word RecNumber, word RegsAmount, void* RecData);
```

Implementation of function `(0x15)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
RefType	Reference type (must be 6)
FileNumber	File Number to access
RecNumber	Record Number to write
RegsAmount	Number of registers to transfer
RecData	Pointer to Record data

> Note
> To simplify the use into the user program, I decided to avoid the use of nested structures, so this functions limits the number of record to write to 1

Return	Meaning
0	Success
Other	See the error tables

ReadFIFOQueue()

```
int broker_ReadFIFOQueue(XOBJECT& Broker, byte DeviceID, word Address, word& FifoCount,
void* FIFO);
```

Implementation of function `(0x18)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Address	Start register address (FIFO Index)
FifoCount	Number of registers which were read

Parameter	Values
FIFO	Pointer to our FIFO area (should contain at least 32 registers)

Return	Meaning
0	Success
Other	See the error tables

ReadExceptionStatus()

```
int broker_ReadExceptionStatus(XOBJECT& Broker, byte DeviceID, byte& Data);
```

Implementation of function `(0x07)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Data	Will contain the Device status

The normal response contains the status of the eight Exception Status outputs.
The outputs are packed into one data byte, with one bit per output.
The status of the lowest output reference is contained in the least significant bit of the byte.

The contents of the eight Exception Status outputs are device specific.

Return	Meaning
0	Success
Other	See the error tables

Diagnostics()

```
int broker_Diagnostics(XOBJECT& Broker, byte DeviceID, word SubFunction, void*
pSendData, void* pRecvData, word ItemsToSend, word& ItemsRecvd);
```

Implementation of function `(0x08)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
SubFunction	Request Sub function
pSendData	Pointer to data to send (our output buffer)
ItemsToSend	Items to send (each item is 16 bit wide)
ItemsRecvd	Items received (number of 16 bit words)

Return	Meaning
0	Success
Other	See the error tables

GetCommEventCounter()

```
int broker_GetCommEventCounter(XOBJECT& Broker, byte DeviceID, word& Status, word&
EventCount);
```

Implementation of function `(0x0B)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Status	Status word (returned)
EventCount	Events count (returned)

Return	Meaning
0	Success
Other	See the error tables

GetCommEventLog()

```
int broker_GetCommEventLog(XOBJECT& Broker, byte DeviceID, word& Status, word&
EventCount, word& MessageCount, word& NumItems, void* Events);
```

Implementation of function `(0x0C)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
Status	Status word (returned)
EventCount	Events count (returned)
MessageCount	Message Counter (returned) (*)
RegsAmount	Number of registers to transfer
NumItems	Number of items read (every event is one byte)
Events	Pointer to our incoming data buffer

(*) The message counter contains the quantity of messages processed by the remote device since its last restart, clear counters operation, or power–up.

Return	Meaning
0	Success
Other	See the error tables

ReportServerID()

```
int broker_ReportServerID(XOBJECT& Broker, byte DeviceID, void* pUsrData, int&
DataSize);
```

Implementation of function `(0x11)`

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
pUsrData	Pointer to our incoming data buffer
DataSize	Number of byte read (returned)

Return	Meaning
0	Success
Other	See the error tables

ExecuteMEIFunction()

```
int broker_ExecuteMEIFunction(XOBJECT& Broker, byte DeviceID, byte MEI_Type, void*
pWRUsrData, word WRSize, void* pRDUsrData, word& RDSize);
```

Implementation of function `(0x2B)`

MEI stands for Modbus Encapsulated Interface, i.e., this is a tunnel function.

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
MEI_Type	MEI Type
pWRUsrData	Pointer to our outcoming buffer
WRSize	Size (byte) to send
pRDUsrData	Pointer to our incoming buffer
RDSize	Size (byte) received

Return	Meaning
0	Success

Return	Meaning
Other	See the error tables

CustomFunctionRequest()

```
int broker_CustomFunctionRequest(XOBJECT& Broker, byte DeviceID, byte UsrFunction, void*
pUsrPDUWrite, word SizePDUWrite, void* pUsrPDURead, word& SizePDURead, word
SizePDUExpected);
```

This function can execute an **User Modbus Function**, i.e., a function not covered bye Modbus specifications

> Note
> SnapModbus only checks that the function number is not an "error response" i.e., the bit 7 **must be zero**

Parameter	Values
*Broker**	Object descriptor which came from the creation function
*DeviceID**	Device ID if the Broker is a FieldController, otherwise is ignored.
*UsrFunction**	User function code
*pUsrPDUWrite**	Pointer to our outcoming PDU buffer
*SizePDUWrite**	Size (byte) to send
*pUsrPDURead**	Pointer to our incoming PDU buffer
*SizePDURead**	Size (byte) received
*SizePDUExpected**	Size (byte) expected (*)

(*) If you know in advance the expected data size, otherwise pass 0.
This could be useful for serial communication to optimize the incoming telegram build.
For Ethernet communication this is meaningless.

Return	Meaning
0	Success
Other	See the error tables

RawRequest()

```
int broker_RawRequest(XOBJECT& Broker, byte DeviceID, void* pUsrPDUWrite, word
SizePDUWrite, void* pUsrPDURead, word& SizePDURead, word SizePDUExpected);
```

This is a low-level function that allow to send/receive an **arbitrary PDU**

> Note
> SnapModbus will manage the transaction calculating the in/out CRC, if needed, (RTU/ASCII)

> or creating the **MBAP** header (TCP/UDP)

Parameter	Values
Broker	Object descriptor which came from the creation function
DeviceID	Device ID if the Broker is a FieldController, otherwise is ignored.
pUsrPDUWrite	Pointer to our outcoming PDU buffer
SizePDUWrite	Size (byte) to send
pUsrPDURead	Pointer to our incoming PDU buffer
SizePDURead	Size (byte) received
SizePDUExpected	Size (byte) expected (*)

(*) If you know in advance the expected data size, otherwise pass 0.
This could be useful for serial communication to optimize the incoming telegram build.
For Ethernet communication this is meaningless.

Return	Meaning
0	Success
Other	See the error tables

Device

Creation/Destruction

CreateEthernetDevice()

```
void device_CreateEthernet(XOBJECT& Device, int Proto, byte DeviceID, const char*
Address, int Port);
```

Creates an **Ethernet Device** working with a given transport protocol

Parameter	Values
Device	Object descriptor returned
Proto	Transport Protocol
DeviceID	Device Address (ID)
Address	Device IP Address e.g. "192.168.1.15"
Port	Device Port, usually 502

Value	Protocol
0	TCP
1	UDP

Value	Protocol
2	RTU Over TCP
3	RTU Over UDP

CreateSerialDevice()

```
void device_CreateSerial(XOBJECT& Device, int Format, byte DeviceID, const char*
PortName, int BaudRate, char Parity, int DataBits, int Stops, int Flow);
```

Creates a **Serial Device** working with a given Data Format

Parameter	Values
Device	Object descriptor returned
Format	Data Format
DeviceID	Device Address (ID)
PortName	Serial Channel Name as known to the host OS (1)
Baudrate	Speed (bps) e.g. 9600, 19200, 115200 ...
Parity	Frame Parity
DataBits	Frame data bits : 7 or 8
StopBits	Frame stop bits : 1 or 2
Flow	Control Flow

(1) In Windows it will be "COM1", "COM2" and so on.. In Linux/BSD/macOS, please refer to the device list into `/dev`

Format

Value	Format
0	RTU
1	ASCII

Parity

Value	Parity
E or **e**	Even Parity
O or **o**	Odd Parity
N or **n**	No Parity

Flow

Value	Flow
0	No control flow

Value	Flow
1	Hardware control flow (RTS/CTS)

Destroy()

```
void device_Destroy(XOBJECT& Device);
```

Destroys the Object

Parameter	Values
Device	Object descriptor which came from the creation function

---

Object control functions

SetParam()

```
int device_SetParam(XOBJECT& Device, int ParamIndex, int Value);
```

These parameters apply directly to the Device, it's similar to Broker's SetLocalParam()

Parameter	Values
Device	Object descriptor which came from the creation function
ParamIndex	Parameter Index
Value	Value to set

Refer **here** for the parameter list.

Return	Meaning
0	Success
Other	See the error tables

GetSerialInterframe()

```
int device_GetSerialInterframe(XOBJECT& Device, int& InterframeDelay, int&
MaxInterframeDetected);
```

This is a debug/tuning function that allow us to know the max spourious interframe (ms) detected into the last transaction. Have a look to **Serial communications**.

It's specific for serial communications. For TCP/UDP Devices this is not used.

Parameter	Values
*Device**	Object descriptor which came from the creation function
*InterframeDelay**	The value that we set with **par_InterframeDelay**

Parameter	Values
*MaxInterframeDetected**	Value to set

Return	Meaning
0	Success
Other	See the error tables

SetCustomFunction()

```
int device_SetCustomFunction(XOBJECT& Device, byte FunctionID, int Value);
```

This function allows to enable/disable a given user-defined function into the Device.

Parameter	Values
Device	Object descriptor which came from the creation function
FunctionID	Function number
Value	**1** : Enables the function, **0** : Disables it

When an user function is enabled, the user function callback is called (if set).
only one callback is used for all user functions, see examples below.

Return	Meaning
0	Success
Other	See the error tables

C#

```
// Declare the delegate (common for all user functions)
static readonly TUsrFunctionRequest UsrFunctionRequest = new
TUsrFunctionRequest(OnUserFunction);

// Into the body (main or other method) set 2 user functions and the callback
Device.SetCustomFunction(0x41, true);
Device.SetCustomFunction(0x42, true);
Device.RegisterCallback(MBConsts.cbkUsrFunction,
Marshal.GetFunctionPointerForDelegate(UsrFunctionRequest), IntPtr.Zero);

// Callback code
static int OnUserFunction(IntPtr usrPtr, byte Function, IntPtr RxPDU, int RxPDUSize,
IntPtr TxPDU, ref ushort TxPDUSize)
{
   if (Function == 0x41)
   {
       // Do something
   }
   if (Function == 0x42)
   {
       // Do something else
   }
   return 0;
}
```

Delphi/Lazarus

```
// Into the body (main or other method) set 2 user functions and the callback
Device.SetCustomFunction($41, true);
Device.SetCustomFunction($42, true);
Device.RegisterCallback(cbkUsrFunction, @OnUserFunction, nil);

// Callback code
function OnUserFunction(usrPtr : Pointer; UsrFunction : byte; RxPDU : Pointer;
  RxPDUSize : word; TxPDU : Pointer; var TxPDUSize : word) : integer;
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
begin
  if UsrFunction = $41 then
  begin
      // Do something
  end;
  if UsrFunction = $42 then
  begin
      // Do something else...
  end;
  Result := 0;
end;
```

C++

```
// Into the body (main or other method) set 2 user functions and the callback
Device->SetCustomFunction(0x41, true);
Device->SetCustomFunction(0x42, true);
Device->RegisterCallback(cbkUsrFunction, (void*)OnUserFunction, NULL);

// Callback code (SNAP_API = __stdcall in Windows, nothing in Linux/FreeBSD/macOS)
int SNAP_API OnUserFunction(void* usrPtr, byte Function, void* RxPDU, uint16_t
RxPDUSize, void* TxPDU, uint16_t& TxPDUSize)
{
    if (Function == 0x41)
    {
        // Do something
    }
    if (Function == 0x42)
    {
        // Do something else
    }
    return 0;
}
```

Start()

```
int device_Start(XOBJECT& Device);
```

The device is started according to the set parameters.

TCP Device : The listener thread is created and the TCP socket is bound to the IP Address : Port
UDP Device : The listener thread is created and the UDP socket is bound to the IP Address : Port
SER Device : The listener thread is created and the Serial socket is bound to the port.

> Note
> In Unix-derived operating systems, to bind port 502 an application must have root privileges (must be run with `sudo`)

Parameter	Values
Device	Object descriptor which came from the creation function

Return	Meaning
0	Success
Other	See the error tables

Stop()

```
int device_Stop(XOBJECT& Device);
```

The device is stopped, all sockets are destroyed and all thread are closed.

Parameter	Values
Device	Object descriptor which came from the creation function

Return	Meaning
Always 0	Success

BindEthernet()/BindSerial()

```
int device_BindEthernet(XOBJECT& Device, byte DeviceID, const char* Address, int Port);
int device_BindSerial(XOBJECT& Device, byte DeviceID, const char* PortName, int
BaudRate, char Parity, int DataBits, int Stops, int Flow);
```

these functions allow you to (re)bind the Device to different Addresses/Ports.
The device must not be running.

BindEthernet()

Parameter	Values
Device	Object descriptor returned
DeviceID	Device Address (ID)
Address	Device IP Address e.g. "192.168.1.15"
Port	Device Port, usually 502

BindSerial()

Parameter	Values
Device	Object descriptor returned
DeviceID	Device Address (ID)
PortName	Serial Channel Name as known to the host OS (1)
Baudrate	Speed (bps) e.g. 9600, 19200, 115200 ...
Parity	Frame Parity
DataBits	Frame data bits : 7 or 8
StopBits	Frame stop bits : 1 or 2
Flow	Control Flow

(1) In Windows it will be "COM1", "COM2" and so on.. In Linux/BSD/macOS, please refer to the device list into `/dev`

Parity

Value	Parity
E or **e**	Even Parity
O or **o**	Odd Parity
N or **n**	No Parity

Flow

Value	Flow
0	No control flow
1	Hardware control flow (RTS/CTS)

Return (both functions)	Meaning
0	Success
Other	See the error tables

AddPeer()

```
int device_AddPeer(XOBJECT& Device, const char* Address);
```

A new item is inserted into the Peer List (Only TCP/UDP Devices)
Have a look at `par_DevPeerListMode` **here**

Parameter	Values
Device	Object descriptor which came from the creation function
Address	IP Address to insert (e.g. "192.168.1.34")

Return	Meaning
0	Success
Other	See the error tables

RegisterArea()

```
int device_RegisterArea(XOBJECT& Device, int AreaID, void* Data, int Amount);
```

A **shared resource** is registered.

Parameter	Values
Device	Object descriptor which came from the creation function
AreaID	Area Identifier (see below)
Data	Pointer to Area (or Area reference)
Amount	Number of items to share (bits or registers)

Return	Meaning
0	Success
Other	See the error tables

AreaID (mnemonic)	Value	Resource
mbAreaDiscreteInputs	0	Discrete Inputs

AreaID (mnemonic)	Value	Resource
mbAreaCoils	1	Coils
mbAreaInputRegisters	2	Input Registers
mbAreaHoldingRegisters	3	Holding Registers

When an Area is registered an internal `Critical Section Object` is created; it will used by `LockArea()/UnlockArea()` functions.

Sharing `regs_amount` registers using an array of 16 bit words.

C#

```
// Area declaration
static ushort[] HoldingRegisters = new ushort[regs_amount];
// Area registering
Device.RegisterArea(MBConsts.mbaHoldingRegisters, ref HoldingRegisters, regs_amount);
```

C++

```
// Area declaration
uint16_t HoldingRegisters[regs_amount];
// Area registering
Device->RegisterArea(mbAreaHoldingRegisters, &HoldingRegisters, regs_amount);
```

Delphi/Lazarus

```
// Area declaration
HoldingRegisters : packed array[0..regs_amount-1] of word;
// Area registering
Device.RegisterArea(mbAreaHoldingRegisters, @HoldingRegisters, regs_amount);
```

LockArea()/UnlockArea()

```
int device_LockArea(XOBJECT& Device, int AreaID);
int device_UnlockArea(XOBJECT& Device, int AreaID);
```

With these function we can manage the concurrent access to the shared areas as explained **here**

Parameter	Values
Device	Object descriptor which came from the creation function
AreaID	Area Identifier, same as **RegisterArea()** see above

Return	Meaning
0	Success
Other	See the error tables

This assures us that it will never happen that a client can read from the three registers old and new data at the same time.

Delphi/Lazarus

```
LockArea(mbaHoldingRegisters);
try
    HoldingRegisters[1]:=NewValue_1;
    HoldingRegisters[2]:=NewValue_2;
    HoldingRegisters[3]:=NewValue_3;
finally // to ensure to ever unlock the area
    UnlockArea(mbaHoldingRegisters);
end;
```

CopyArea()

```
int device_CopyArea(XOBJECT& Device, int AreaID, word Address, word Amount, void* Data,
int CopyMode);
```

Allows to safely copy a shared area.
See here why you need of it.

It executes in sequence:

1. LockArea()
2. Copy
3. UnlockArea()

Parameter	Values
Device	Object descriptor which came from the creation function
AreaID	Area Identifier (see below)
Address	Start Index of shared area
Amount	Number of items to copy (bits or registers)
Data	Pointer to user Buffer
CopyMode	Copy direction (see below)

CopyMode	Values
0 (Read)	The area is copied into our Buffer
1 (Write)	Our Buffer is copied into the Area

AreaID (mnemonic)	Value	Resource
mbAreaDiscreteInputs	0	Discrete Inputs
mbAreaCoils	1	Coils
mbAreaInputRegisters	2	Input Registers
mbAreaHoldingRegisters	3	Holding Registers

Return	Meaning

Return	Meaning
0	Success
Other	See the error tables

> Note
> If Start + Amount is greater than the Area limit, Amount is "trimmed" (no error is iussed)

PickEvent()/PickEventAsText()

```
int device_PickEvent(XOBJECT& Device, void* pEvent);
int device_PickEventAsText(XOBJECT& Device, char* Text, int TextSize);
```

Extract an event from the internal queue as explained **here**

The first returns the Event (to be analyzed), the second one its textual representation (for log purpose).

Parameter	Values
Device	Object descriptor which came from the creation function
pEvent	Pointer to the event
Text	Pointer to an Array of Ansi chars
TextSize	Array size

Return	Meaning
0	The queue is empty, nothing was extracted
1	Event or Text contains the extracted item

GetDeviceInfo()

```
int device_GetDeviceStatus(XOBJECT& Device, TDeviceInfo& DeviceInfo);
```

Reads the Device status

Parameter	Values
Device	Object descriptor which came from the creation function
DeviceInfo	Reference (Pointer to) of a TDeviceInfo struct

TDeviceStatus struct

```
typedef struct {
    int32_t Running;
    int32_t ClientsCount;   // only for TCP
    int32_t ClientsBlocked; // only for TCP/UDP
    int32_t LastError;
}TDeviceInfo;
```

Field	Meaning
Running	0 : Not running; 1 : Running
ClientsCount	Only TCP: number of connected clients
ClientsBlocked	Only TCP/UDP : number of blocked clients (*)
LastError	Last Device error code (0 = success)
JobTime	0 : Not used here

(*) See `par_DevPeerListMode` **here**

RegisterCallback()

```
int device_RegisterCallback(XOBJECT& Device, int CallbackID, void* cbRequest, void*
UsrPtr);
```

A callback is registered, as explained **here**

Parameter	Values
Device	Object descriptor which came from the creation function
CallbackID	Callback Selector (see below)
cbRequest	Pointer to Callback (or Callback reference)
UsrPtr	An user pointer returned unmodified by the Callback

Return	Meaning
0	Success
Other	See the error tables

Callback selector	value
cbkDeviceEvent	0
cbkPacketLog	1
cbkDiscreteInputs	2
cbkCoils	3
cbkInputRegisters	4
cbkHoldingRegisters	5
cbkReadWriteRegisters	6

Callback selector	value
cbkMaskRegister	7
cbkFileRecord	8
cbkExceptionStatus	9
cbkDiagnostics	10
cbkGetCommEventCounter	11
cbkGetCommEventLog	12
cbkReportServerID	13
cbkReadFIFOQueue	14
cbkEncapsulatedIT	15
cbkUsrFunction	16
cbkPassthrough	17

Callbacks prototypes

these are the prototypes as they are declared.

```
typedef void (SNAP_API* pfn_DeviceEvent)(void* usrPtr, void* PEvent, int Size);
typedef void (SNAP_API* pfn_PacketLog)(void* usrPtr, longword Peer, int Direction, void*
Data, int Size);
typedef int (SNAP_API* pfn_DiscreteInputsRequest)(void* usrPtr, word Address, word
Amount, void* Data);
typedef int (SNAP_API* pfn_CoilsRequest)(void* usrPtr, int Action, word Address, word
Amount, void* Data);
typedef int (SNAP_API* pfn_InputRegistersRequest)(void* usrPtr, word Address, word
Amount, void* Data);
typedef int (SNAP_API* pfn_HoldingRegistersRequest)(void* usrPtr, int Action, word
Address, word Amount, void* Data);
typedef int (SNAP_API* pfn_ReadWriteMultipleRegistersRequest)(void* usrPtr, word
RDAddress, word RDAmount, void* RDData, word WRAddress, word WRAmount, void* WRData);
typedef int (SNAP_API* pfn_MaskRegisterRequest)(void* usrPtr, word Address, word
AND_Mask, word OR_Mask);
typedef int (SNAP_API* pfn_FileRecordRequest)(void* usrPtr, int Action, word RefType,
word FileNumber, word RecNumber, word RegsAmount, void* Data);
typedef int (SNAP_API* pfn_ExceptionStatusRequest)(void* usrPtr, byte& Status);
typedef int (SNAP_API* pfn_DiagnosticsRequest)(void* usrPtr, word SubFunction, void*
RxItems, void* TxItems, word ItemsSent, word& ItemsRecvd);
typedef int (SNAP_API* pfn_GetCommEventCounterRequest)(void* usrPtr, word& Status, word&
EventCount);
typedef int (SNAP_API* pfn_GetCommEventLogRequest)(void* usrPtr, word& Status, word&
EventCount, word& MessageCount, void* Data, word& EventsAmount);
typedef int (SNAP_API* pfn_ReportServerIDRequest)(void* usrPtr, void* Data, word&
DataSize);
typedef int (SNAP_API* pfn_ReadFIFOQueueRequest)(void* usrPtr, word PtrAddress, void*
FIFOValues, word& FifoCount);
typedef int (SNAP_API* pfn_EncapsulatedIT)(void* usrPtr, byte MEI_Type, void*
MEI_DataReq, word ReqDataSize, void* MEI_DataRes, word& ResDataSize);
typedef int (SNAP_API* pfn_UsrFunctionRequest)(void* usrPtr, byte Function, void* RxPDU,
word RxPDUSize, void* TxPDU, word& TxPDUSize);
typedef int (SNAP_API* pfn_Passthrough)(void* usrPtr, byte DeviceID, void* RxPDU, word
RxPDUSize, void* TxPDU, word& TxPDUSize);
```

Into the examples:

- /examples/cpp/cpp_device.cpp
- /examples/c#/device/cs_device.cs
- /examples/pascal/device.pas

more conveniently, you will find their declarations and the usage

Parameters

There are several parameters that allow customization of the behavior of SnapMB It is possible to modify them using the functions **SetLocalParam()** or **SetRemoteDeviceParam()** for the Client/Controller and **SetParam()** for the Device. The two main parameters are **ParamIndex** and **Value**. let's see their meaning.

> Note
> All parameters accepted by the Client also apply to the FieldController.

---

par_TCP_UDP_Port `[TCP/UDP Client and Device]`

Allows to change the TCP/UDP port. changing this param when the Client is connected will cause its disconnection. Trying to change it in a Device when it's running will issue the error errDevOpNotAllowed

`Default = 502`

---

par_DeviceID `[RTU Device]`

Allows you to change the Device address (DeviceID), it must be between 1 and 255.

---

par_TcpPersistence `[TCP Client]`

1: the Client keeps the connection across the transactions.
0: the Client connects before a transaction and disconnects after it.

Since Connection/Disconnection introduces an overhead, use the auto-disconnection only for widely time-spaced transactions. This parameter is not used for UDP sockets (which are always unconnected)

`Default = 1`

---

par_DisconnectOnError `[TCP Client/TCP Device]`

- 1: If a logical error occurs the Client disconnects, the Device close the socket of the offending Client.
- 0: No disconnection is performed.

`Default = 1`

> Note
>
> - On Hardware/Network Low level error the Client/Device always disconnects, regardless of this parameter.
> - You don’t need explicitly to reconnect a client,it will automatically reconnect on the next transaction.

---

par_SendTimeout `[All]`

Represents the maximum timeout (ms) for a transmission (via Network or Serial).
It is advisable to keep this value low, generally the transmission is always instantaneous because the data is copied to the output queue, a write timeout error is a symptom of some serious problem.

```
Default Network = 200ms
Default Serial = 500ms
```

---

par_SerialFormat `[Serial Device]`

Allows to change the interface transport

- 0: RTU
- 1: ASCII

If the Device is running, the error **errDevOpNotAllowed** is issued

---

par_AutoTimeout `[Client]`

- 1: the Client autocalculates the best timeout for its Device, it will be the double of the mean between the minimum and maximum response time (of the correct transactions), the value is stored into AutoTimeCalc_ms.

- 0: **FixedTimeout_ms** param will be always used.

`Default = 1`

---

par_AutoTimeLimitMin `[Client]`

represents the minimum threshold (ms) below which an automatically calculated timeout cannot go.

---

par_FixedTimeout `[Client]`

represents the timeout (ms) that we always want to use.

`Default = 3000`

---

par_BaseAddress `[Client]`

Modbus specification states that the first element of a Device has Address = 1 but the transport protocol is zero-based.
It means that an Address parameter is decreased by 1 first to be passed to the protocol stack.
Setting `BaseAddress = 0` the Address parameter will not be decreased.

`Default = 1` in accord to Modbus specification.
Any value different from 0 or 1 will be ignored.

---

par_DevPeerListMode `[TCP/UDP Device]`

This parameter indicates how the peerlist should be considered.

- 0: (**plmDisabled**) the PeerList is not used, every incoming address is trusted.
- 1: (**plmAllowList**) the PeerList contains trusted addresses, a client can connect only if its address is into the list.
- 2: (**plmBlockList**) the addresses in the list are blocked, i.e., they cannot connect to the device

`Default = 0`

---

par_PacketLog `[Device]`
this parameter together with the **OnPacketLog** event, establishes when this event must be triggered (it is therefore a filter for this event).

- 0: (**PacketLog_NONE**): the event is never triggered
- 1: (**PacketLog_IN**) the event is triggered only for incoming packets.
- 2: (**PacketLog_OUT**) the event is triggered only for outcoming packets.
- 2: (**PacketLog_BOTH**) the event is triggered ever.

`Default = 2;`

> Note
> To receive an Event, its callback must be set.

---

par_InterframeDelay `[Serial Client/Device]`

This value indicates how many milliseconds we have to wait after receiving the last character, to determine the end of the telegram. It is only used for telegrams whose size is not known in advance (device telegrams).

Have a look here

`Default = 50`
After many test, 50 ms seems to be a good compromise. It's safe but not too high.

---

par_WorkInterval `[Device]`

Represents the work interval of the listener thread. Don't change it unless you have serious reasons to.

`Default = 100ms`

---

par_AllowSerFunOnEth `[TCP/UDP Device]`

Some functions, by Modbus specification are specific for Serial line:

- (0x07) Read Exception Status
- (0x08) Diagnostics
- (0x0B) Get Comm Event Counter
- (0x0C) Get Comm Event Log
- (0x11) Report Server ID

If this parameter is 0 and a TCP/UDP Device receives one of the above functions, an error (0x01) Illegal Function is issued to the Client.
if this parameter is 1, the Device will serve the request (provided that the relative Callback has been set)

`Default = 0`

---

par_MaxRetries `[Client]`

This parameter indicates the maximum number of retries a client can make in case of errors.

```
Default TCP Client = 2
Default RTU Client = 1
```

---

par_AttemptSleep `[Client]`

This parameter is related to the previous one and represents the time (ms) that the client must wait between one attempt and the next

`Default = 300 ms`

---

par_DisconnectTimeout `[TCP Device]`

this parameter represents the time (ms) beyond which a TCP Device disconnects its Client due to inactivity. Many devices have this behavior, perhaps due to limited resources or because they are unable to reuse a socket. In this context it is not strictly necessary as the Device is a multithreaded TCP server.

In any case, this possibility is also available. 0 as value will disable this behavior and the Device will never drop a connection.

`Default = 0`

---

par_DevicePassthrough `[Device]`

This is a structure parameter and indicates the way a device works.
If 1 the Device will not work and will call the Passthrough Callback at each request passing the received PDU as parameter and using the PDU set in the callback as response. However, the device will take care of calculating the CRC (RTU Device) and formatting the telegram.

This behavior is useful for debugging purposes or for implementing a gateway or a protocol converter.

`Default = 0`

Misc functions

ErrorText()

```
const char* ErrorText(int Error, char* Text, int TextSize);
```

Returns the textual representation of an error

Parameter	Values
Error	Error code
Text	Pointer to an Array of Ansi chars
TextSize	Array size (provide at least 256)

In you wrapper you will find this function declared in a more convenient way.

EventText()

```
const char* EventText(void* Event, char* Text, int TextSize);
```

Returns the textual representation of an event.

Parameter	Values
Event	Event reference
Text	Pointer to an Array of Ansi chars
TextSize	Array size (provide at least 256)

In you wrapper you will find this function declared in a more convenient way.

Errors

Almost all functions returns an integer (4 bytes) result.

If it's zero, the function succeeded, otherwise this code must be considered as an **Error**

This is composed by some fields

S	0	0	C	Code

S	Object Selector
0	Always zero
C	Category
Code	Error code

The Object selector is **who** raised the error.
The Category is the **contex**
The Code is the **Error number**

Object selector

```
const int errLibrary                  = 0x10000000;
const int errSerialClient             = 0x20000000;
const int errEthernetClient           = 0x30000000;
const int errFieldController          = 0x40000000;
const int errSerialDevice             = 0x50000000;
const int errEthernetDevice           = 0x60000000;
```

Category

```
const int errCategoryLibrary          = 0x00010000; // Library error
const int errCategorySerialSocket     = 0x00020000; // Serial socket error
const int errCategoryNetSocket        = 0x00030000; // TCP-UDP Error
const int errCategoryMBProtocol       = 0x00040000; // Protocol error (0x8X received)
const int errCategoryProcess          = 0x00050000; // Process error
```

Library errors

```
const int errNullObject               = 0x00000001; // Null object passed
const int errObjectInvalidMethod      = 0x00000002; // invalid method for this object
```

FieldController errors

```
const int errUndefinedBroker          = 0x0000000B;
const int errUndefinedClient          = 0x0000000C;
const int errDeviceIDZero             = 0x0000000D;
const int errDeviceAlreadyExists      = 0x0000000E;
const int errUndefinedController      = 0x0000000F;
const int errSomeConnectionsError     = 0x00000010;
const int errCommParamsMismatch       = 0x00000011;
```

Serial socket errors

```
const int errPortInvalidParams       = 0x00000001;
const int errPortSettingsTimeouts    = 0x00000002;
const int errPortSettingsParams      = 0x00000003;
const int errOpeningPort             = 0x00000004;
const int errPortReadTimeout         = 0x00000005;
const int errPortWriteTimeout        = 0x00000006;
const int errPortReadError           = 0x00000007;
const int errPortWriteError          = 0x00000008;
const int errBufferOverflow          = 0x00000009;
const int errPortGetParams           = 0x0000000A;
const int errPortLocked              = 0x0000000B;
const int errInterframe              = 0x0000000C;
```

Network socket errors
These errors come directly from underlying OS
Here is reported the function that parses them (mb_text.cpp)

```
const char* SocketTextOf(int Error)
{
 switch (Error)
 {
 case 0:                   return "\0";
 case WSAEINTR:            return "TCP/UDP: Interrupted system call\0";
 case WSAEBADF:            return "TCP/UDP: Bad file number\0";
 case WSAEACCES:           return "TCP/UDP: Permission denied\0";
 case WSAEFAULT:           return "TCP/UDP: Bad address\0";
 case WSAEINVAL:           return "TCP/UDP: Invalid argument\0";
 case WSAEMFILE:           return "TCP/UDP: Too many open files\0";
 case WSAEWOULDBLOCK:      return "TCP/UDP: Operation would block\0";
 case WSAEINPROGRESS:      return "TCP/UDP: Operation now in progress\0";
 case WSAEALREADY:         return "TCP/UDP: Operation already in progress\0";
 case WSAENOTSOCK:         return "TCP/UDP: Socket operation on non socket\0";
 case WSAEDESTADDRREQ:     return "TCP/UDP: Destination address required\0";
 case WSAEMSGSIZE:         return "TCP/UDP: Message too long\0";
 case WSAEPROTOTYPE:       return "TCP/UDP: Protocol wrong type for Socket\0";
 case WSAENOPROTOOPT:      return "TCP/UDP: Protocol not available\0";
 case WSAEPROTONOSUPPORT:  return "TCP/UDP: Protocol not supported\0";
 case WSAESOCKTNOSUPPORT:  return "TCP/UDP: Socket not supported\0";
 case WSAEOPNOTSUPP:       return "TCP/UDP: Operation not supported on Socket\0";
 case WSAEPFNOSUPPORT:     return "TCP/UDP: Protocol family not supported\0";
 case WSAEAFNOSUPPORT:     return "TCP/UDP: Address family not supported\0";
 case WSAEADDRINUSE:       return "TCP/UDP: Address already in use\0";
 case WSAEADDRNOTAVAIL:    return "TCP/UDP: Can't assign requested address\0";
 case WSAENETDOWN:         return "TCP/UDP: Network is down\0";
 case WSAENETUNREACH:      return "TCP/UDP: Network is unreachable\0";
 case WSAENETRESET:        return "TCP/UDP: Network dropped connection on reset\0";
 case WSAECONNABORTED:     return "TCP/UDP: Software caused connection abort\0";
 case WSAECONNRESET:       return "TCP/UDP: Connection reset by peer\0";
 case WSAENOBUFS:          return "TCP/UDP: No Buffer space available\0";
 case WSAEISCONN:          return "TCP/UDP: Socket is already connected\0";
 case WSAENOTCONN:         return "TCP/UDP: Socket is not connected\0";
 case WSAESHUTDOWN:        return "TCP/UDP: Can't send after Socket shutdown\0";
 case WSAETOOMANYREFS:     return "TCP/UDP: Too many references:can't splice\0";
 case WSAETIMEDOUT:        return "TCP/UDP: Connection timed out\0";
 case WSAECONNREFUSED:     return "TCP/UDP: Connection refused\0";
 case WSAELOOP:            return "TCP/UDP: Too many levels of symbolic links\0";
 case WSAENAMETOOLONG:     return "TCP/UDP: File name is too long\0";
 case WSAEHOSTDOWN:        return "TCP/UDP: Host is down\0";
 case WSAEHOSTUNREACH:     return "TCP/UDP: Unreachable peer\0";
 case WSAENOTEMPTY:        return "TCP/UDP: Directory is not empty\0";
 case WSAEUSERS:            return "TCP/UDP: Too many users\0";
 case WSAEDQUOT:            return "TCP/UDP: Disk quota exceeded\0";
 case WSAESTALE:            return "TCP/UDP: Stale NFS file handle\0";
 case WSAEREMOTE:           return "TCP/UDP: Too many levels of remote in path\0";
#ifdef SNAP_OS_WINDOWS
 case WSAEPROCLIM:          return "TCP/UDP: Too many processes\0";
 case WSASYSNOTREADY:       return "TCP/UDP: Network subsystem is unusable\0";
 case WSAVERNOTSUPPORTED:   return "TCP/UDP: Winsock DLL cannot support this
application\0";
 case WSANOTINITIALISED:    return "TCP/UDP: Winsock not initialized\0";
 case WSAEDISCON:           return "TCP/UDP: Disconnect\0";
 case WSAHOST_NOT_FOUND:    return "TCP/UDP: Host not found\0";
 case WSATRY_AGAIN:         return "TCP/UDP: Non authoritative - host not found\0";
 case WSANO_RECOVERY:       return "TCP/UDP: Non recoverable error\0";
 case WSANO_DATA:           return "TCP/UDP: Valid name, no data record of requested
type\0";
#endif
 case WSAEINVALIDADDRESS:   return "TCP/UDP: Invalid address\0";
 default:
  return "TCP/UDP: Other Socket Error ";
 }
}
```

Modbus protocol errors

```
const int errIllegalFunction         = 0x00000001; // Exception Code 0x01
const int errIllegalDataAddress      = 0x00000002; // Exception Code 0x02
const int errIllegalDataValue        = 0x00000003; // Exception Code 0x03
const int errSlaveDeviceFailure      = 0x00000004; // Exception Code 0x04
const int errAcknowledge             = 0x00000005; // Exception Code 0x05
const int errSlaveDeviceBusy         = 0x00000006; // Exception Code 0x06
const int errNegativeAcknowledge     = 0x00000007; // Exception Code 0x07
const int errMemoryParityError       = 0x00000008; // Exception Code 0x08
const int errGatewayPathUnavailable  = 0x00000010; // Exception Code 0x10
const int errGatewayTargetFailed     = 0x00000011; // Exception Code 0x11
```

Process errors

```
const int errInvalidBroadcastFunction= 0x00000001;
const int errInvalidParamIndex       = 0x00000002;
const int errInvalidAddress          = 0x00000003;
const int errInvalidDataAmount       = 0x00000004;
const int errInvalidADUReceived      = 0x00000005;
const int errInvalidChecksum         = 0x00000006;
const int errTimeout                 = 0x00000007;
const int errInvalidDeviceID         = 0x00000008;
const int errInvalidUserFunction     = 0x00000009;
const int errInvalidReqForThisObject = 0x0000000A;
```

Device errors

```
const int errDevUnknownAreaID        = 0x00000100; // Unknown Area ID
const int errDevAreaZero             = 0x00000101; // Area Amount = 0
const int errDevAreaTooWide          = 0x00000102; // Area Amount too wide
const int errDevUnknownCallbackID    = 0x00000103; // Unknown Callback ID
const int errDevInvalidParams        = 0x00000104; // Invalid param(s) supplied
const int errDevInvalidParamIndex    = 0x00000105; // Invalid param (SetParam())
const int errDevOpNotAllowed         = 0x00000106; // Cannot change because running
const int errDevTooManyPeers         = 0x00000107; // To many Peers for Deny/AcceptList
(only TCP)
const int errDevCannotRebindOnRun    = 0x00000108; // Device is running, stop first
```

Serial communications

Serial interfaces have changed over time, today it is almost impossible to find PCs with "native" serial interfaces. For this reason, SnapMB has been optimized to work with USB/RS485 (or USB/RS232) adapters.

The USB port allows higher communication speeds than the old serial port, it is an evolution of it, so there are no performance problems when using these adapters.

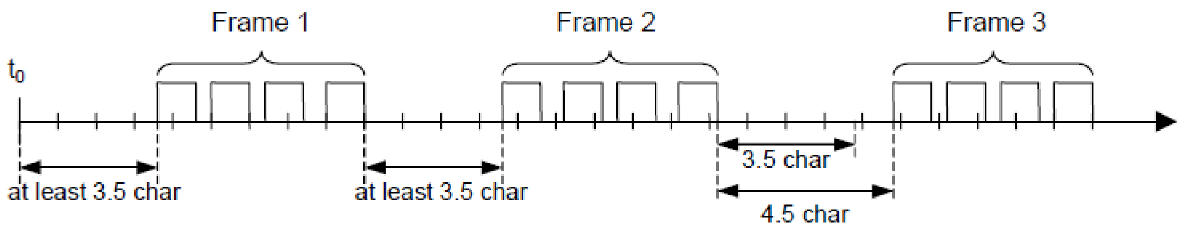
The problem is that the drivers of these devices behave differently than a classic serial.

Also due to the internal management of operating systems, they work like an Ethernet driver: Data is made available "by packet" and not character by character.

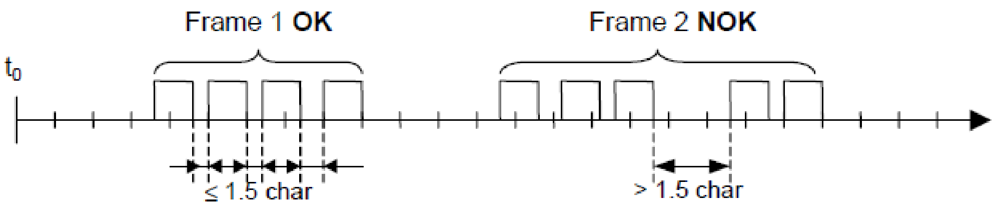
This makes it practically impossible to talk about inter-character delay and for the same reason, it is very difficult to determine the interframe interval, which by specification, is the way to determine when a telegram is complete.

In accord to:
MODBUS over Serial Line - Specification and Implementation Guide V1.02

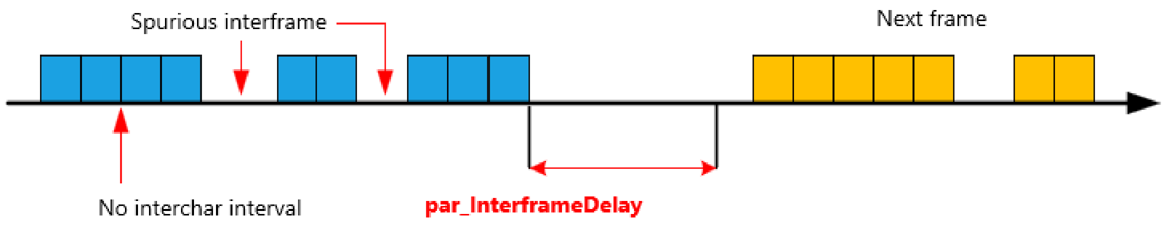
This is the interframe interval:



This the interchar interval:



Instead, this is what we get from the serial port when we use a USB adapter:



Which is completely out of specification.

We can deviate from this specification by also accepting these telegrams (provided the content complies with the specifications).

The remaining problem is that it is difficult to determine the end of a telegram, a delay between groups of characters could be due to the driver software. So, we are forced to increase the timeout which tells us when a telegram is complete.

SnapMB uses a different approach. Except in rare cases, we always know how long the expected telegram is, so, the telegram is parsed step by step until its "logical end", using the same approach that is used with Ethernet protocols.

This allows us to save several milliseconds in the data exchange of the "deterministic" functions. E.g., if we send a request for reading 3 registers, we expect 11 byte as answer.

Have a look at `TMBSerBroker::RecvRTUResponse()` method into **/src/core/mb_serbroker.cpp**

In some circumstances, unfortunately, we do not know the size of the incoming telegram in advance, so we have to determine its length via the interframe.

This, for example, is the case of the **Device**, which acquires all the telegrams transiting on the bus, then discarding those it is not interested in, therefore response telegrams, malformed telegrams, etc.

With reference to the figure, to determine the time interval beyond which we can consider a telegram complete if no other characters arrive, there is the **par_InterframeDelay** parameter.

This should be at least twice the largest spurious interframe.

To know the largest spurious interframe of the last successfully completed transaction, the **GetSerialInterframe()** function exists in the Device which also returns, for convenience, the current value of par_InterframeDelay.

USB Serial adapters

I tested several USB/RS485(232) adapters, those with the (original) **FTDI** chipset are the best, under Linux the driver allows the use of the `ASYNC_LOW_LATENCY` flag.

Windows, Linux

Nothing to declare, FTDI adapters are recognized out of the box (also in Raspberry OS).

FreeBSD

FTDI driver is separate module, so you will have to load it.\

Add to `/boot/loader.conf` the line

```
uftdi_load="YES"
```

macOS

The adapter is recognized out of the box. Into `/dev`, You will find two ports for each adapter: **tty.*** and **cu.***, the latter is the correct one to use.

As a second choice there are converters based on the **CP2102** chipset, these too are recognized by all operating systems, even if in some cases it is necessary to load the driver.

Finally there is the **CH340** chipset, it works quite well and is the cheapest, today this too is in the list of drivers supported by all operating systems.

Installation and commissioning

No installation is required, it is sufficient that in Windows PC **snapmb.dll** be present in the same folder of your application executable and, in Linux/BSD/Mac OS systems, that **libsnapmb.so** (or **libsnapmb.dylib**) is copied to `/usr/lib` or `/usr/local/lib`. The system libraries are statically linked to SnapMB, so it doesn't need of any runtime libraries, your programs will can work on "flat" PCs in portable mode, even from USB pendrive.

Usage

As said, some wrappers are provided, this mean that **you don’t need to know C++ to use SnapMB**. The wrapper is a “glue” code, i.e., a piece of code written in the same language that you are using, which internally interfaces with the library. Currently are provided wrappers for C/C++, C#, Object Pascal (Delphi or FPC/Lazarus) and LabVIEW, which can be compiled 32 or 64 bit, and more could come (if the project has any interest).

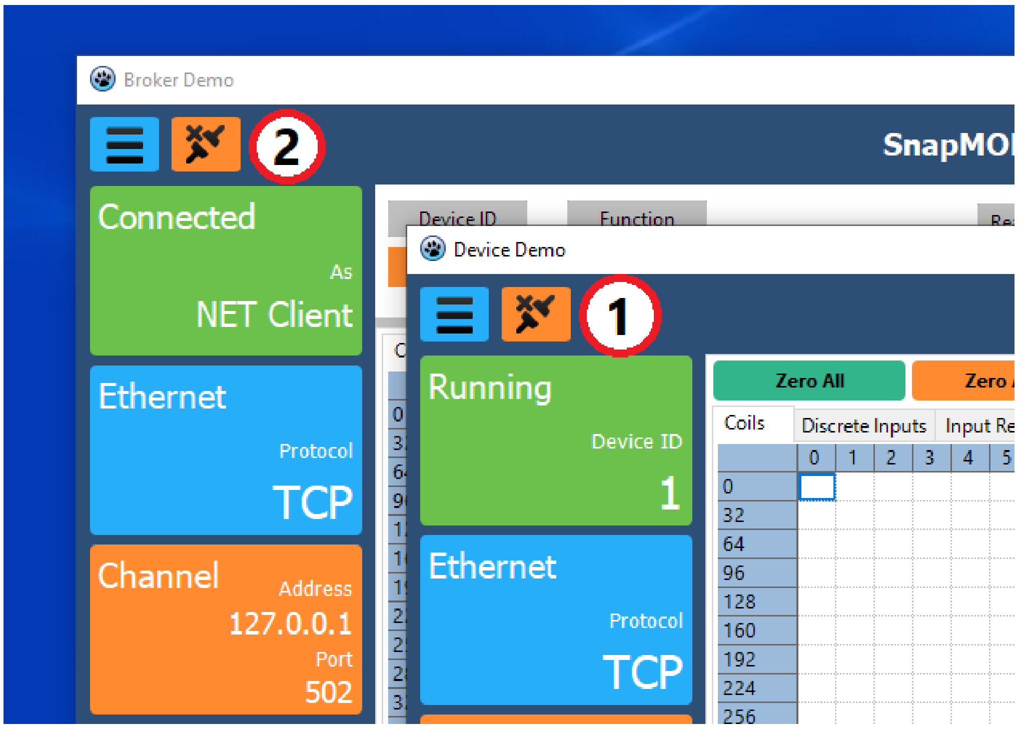
Plese refer to Examples and tools and LabVIEW chapters.

Testing SnapModbus

To test SnapModbus I recommend first of all to use the two tools WinBroker and WinDevice putting them in communication with each other.

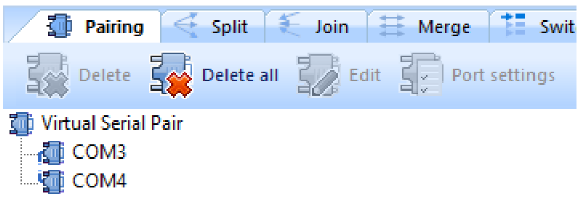
At startup, by default, both work in TCP on the local address 127.0.0.1 and port 502, so everything will work first time.

It is important to first start the Device and then the Client, otherwise the latter will not find an active Device.



So you can use two different PCs and change the two addresses accordingly.

To test the serial protocol you need of two serial adapters, or, you can use a **Virtual Serial Port Driver** like this.



It can create two virtual serial ports connected with a virtual null-modem cable.
Following the figure, you can connect the Client onto COM3 and the Device onto COM4.

Once you are comfortable with the communication, you can open the examples, try them out and modify them for your needs.

Examples and tools

Quick start

C++

```
#include "snapmb.h"

PSnapMBBroker Client_1;
PSnapMBBroker Client_2;
uint16_t HoldingRegisters[256];

int main(int argc, char* argv[])
{
// Creates 2 Clients the first TCP, the second Serial
    Client_1 = new TSnapMBBroker(ProtoTCP, "127.0.0.1", 502);
    Client_2 = new TSnapMBBroker(FormatRTU, "COM3", 115200, 'N', 8, 1, FlowNONE);

// Reads 16 Holding Registers from the Device N.1 starting from 7000
    Client_1->ReadHoldingRegisters(1, 7000, 16, &HoldingRegisters);
// Write the buffer to the Device N.2 starting from 4000
    Client_2->WriteMultipleRegisters(2, 4000, 16, &HoldingRegisters);

    delete Client_1;
    delete Client_2;
}
```

C#

```
using SnapModbus;

    static ushort[] Regs = new ushort[256];
    static SnapMBBroker Client_1;
    static SnapMBBroker Client_2;

static void Main(string[] args)
{
// Creates 2 Clients the first TCP, the second Serial
    Client_1 = new SnapMBBroker(MBConsts.ProtoTCP, "127.0.0.1", 502);
    Client_2 = new SnapMBBroker(MBConsts.FormatRTU, "COM3", 115200, 'N', 8, 1,
MBConsts.FlowNONE);

// Reads 16 Holding Registers from the Device N.1 starting from 7000
    Client_1.ReadHoldingRegisters(1, 7000, 16, Regs);
// Write the buffer to the Device N.2 starting from 4000
    Client_2.WriteMultipleRegisters(2, 4000, 16, Regs);
}
```

Object Pascal (Delphi/Lazarus)

```
Uses SnapMB;
Var
    Regs : packed array[0..255] of word;
    Client_1 : TSnapMBBroker;
    Client_2 : TSnapMBBroker;
Begin
// Creates 2 Clients the first TCP, the second Serial
    Client_1 := TSnapMBBroker.Create(mbTCP, '127.0.0.1', 502);
    Client_1 := TSnapMBBroker.Create(sfRTU, 'COM3', 115200, 'N', 8, 1, FlowNONE);

// Reads 16 Holding Registers from the Device N.1 starting from 7000
    Client_1.ReadHoldingRegisters(1, 7000, 16, @Regs);
// Write the buffer to the Device N.2 starting from 4000
    Client_2.WriteMultipleRegisters(2, 4000, 16, @Regs);

    Client_1.Free;
    Client_2.Free;
End;
```

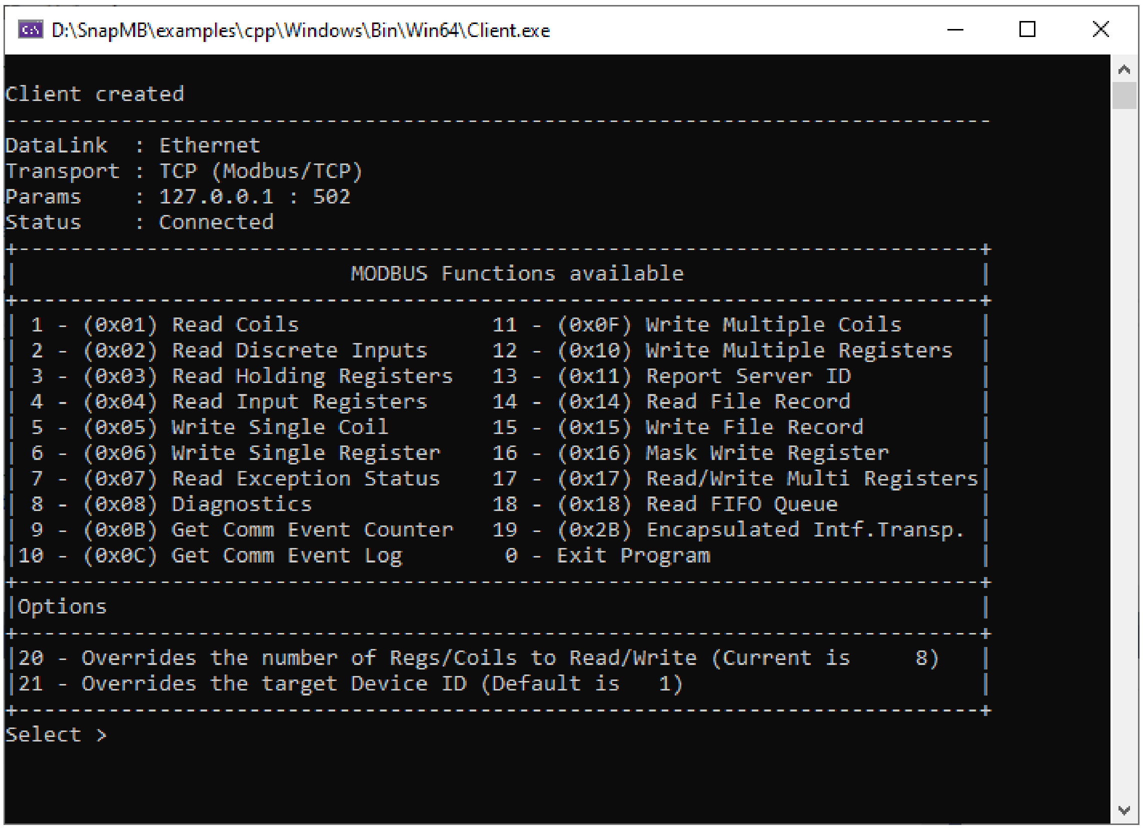
Examples

into /Examples folder you will find many examples to see how to use SnapModbus.

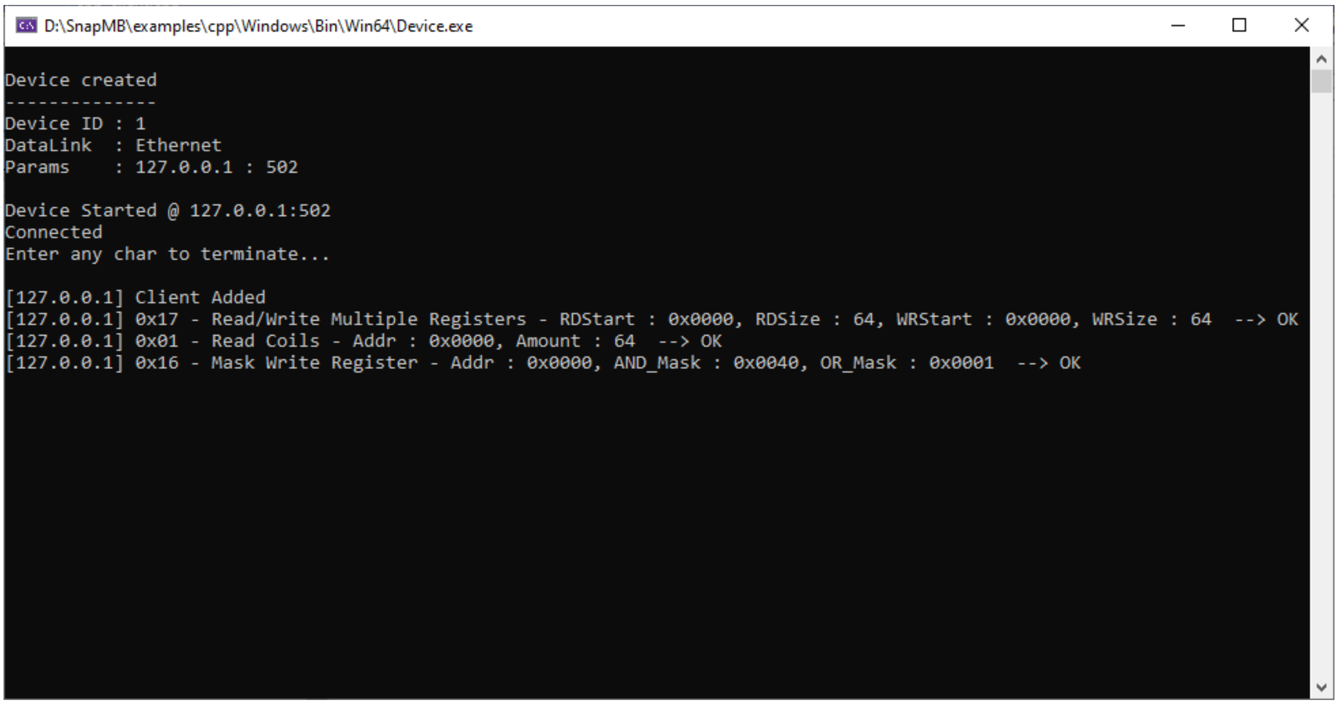
The languages used are C++, C# (Console and WinForm) , Object Pascal (Delphi and Lazarus), you can compile and run C++ examples them into Windows, Linux (all distros), FreeBSD, macOS.

I tested Object Pascal (Lazarus) examples in Windows and Linux, but they should work also into the other OS.

Makefiles and/or project are provided for all of them.



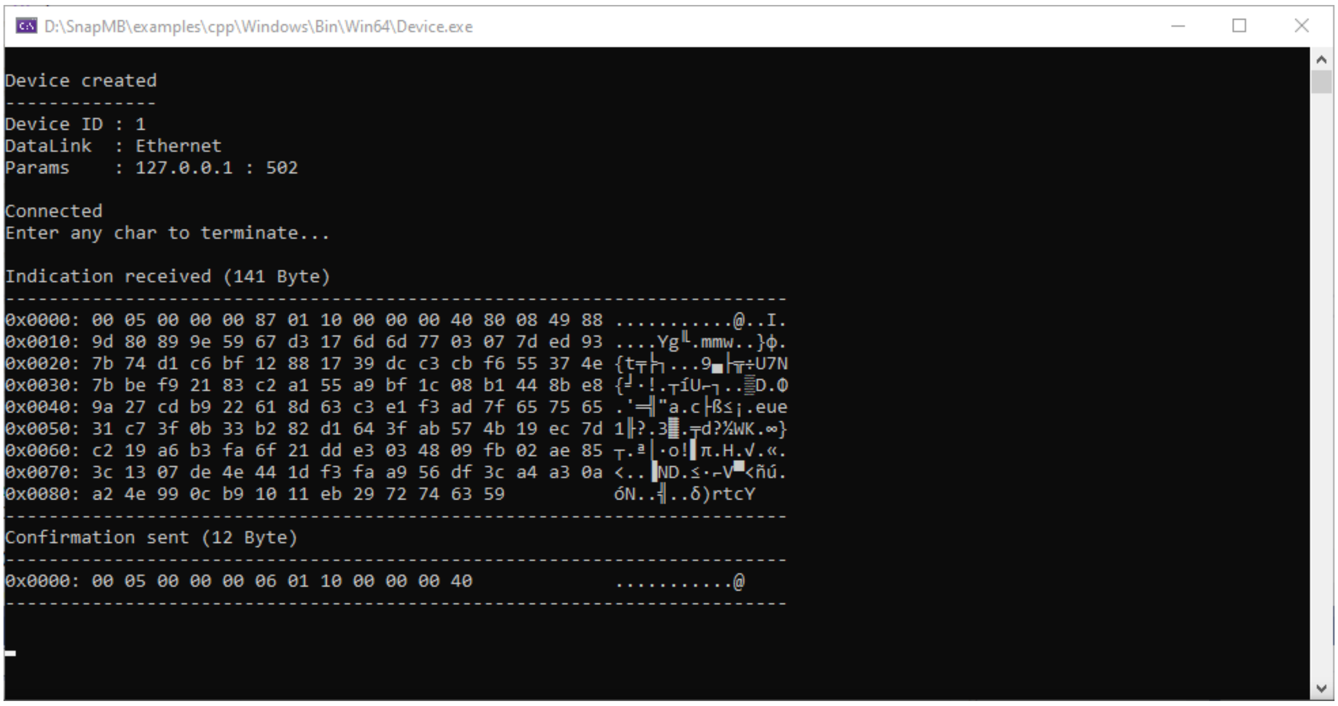
```
Client created
------------------------------------------------------------------------------
DataLink  : Ethernet
Transport : TCP (Modbus/TCP)
Params    : 127.0.0.1 : 502
Status    : Connected
+-----------------------------------------------------------------------------+
|                          MODBUS Functions available                        |
+-----------------------------------------------------------------------------+
| 1 - (0x01) Read Coils                 11 - (0x0F) Write Multiple Coils      |
| 2 - (0x02) Read Discrete Inputs       12 - (0x10) Write Multiple Registers  |
| 3 - (0x03) Read Holding Registers     13 - (0x11) Report Server ID          |
| 4 - (0x04) Read Input Registers       14 - (0x14) Read File Record          |
| 5 - (0x05) Write Single Coil          15 - (0x15) Write File Record         |
| 6 - (0x06) Write Single Register      16 - (0x16) Mask Write Register       |
| 7 - (0x07) Read Exception Status      17 - (0x17) Read/Write Multi Registers|
| 8 - (0x08) Diagnostics                18 - (0x18) Read FIFO Queue           |
| 9 - (0x0B) Get Comm Event Counter     19 - (0x2B) Encapsulated Intf.Transp. |
|10 - (0x0C) Get Comm Event Log          0 - Exit Program                     |
+-----------------------------------------------------------------------------+
|Options                                                                      |
+-----------------------------------------------------------------------------+
|20 - Overrides the number of Regs/Coils to Read/Write (Current is     8)    |
|21 - Overrides the target Device ID (Default is    1)                       |
+-----------------------------------------------------------------------------+
Select >
```



```
Device created
--------------
Device ID : 1
DataLink  : Ethernet
Params    : 127.0.0.1 : 502

Device Started @ 127.0.0.1:502
Connected
Enter any char to terminate...

[127.0.0.1] Client Added
[127.0.0.1] 0x17 - Read/Write Multiple Registers - RDStart : 0x0000, RDSize : 64, WRStart : 0x0000, WRSize : 64  --> OK
[127.0.0.1] 0x01 - Read Coils - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x16 - Mask Write Register - Addr : 0x0000, AND_Mask : 0x0040, OR_Mask : 0x0001  --> OK
```

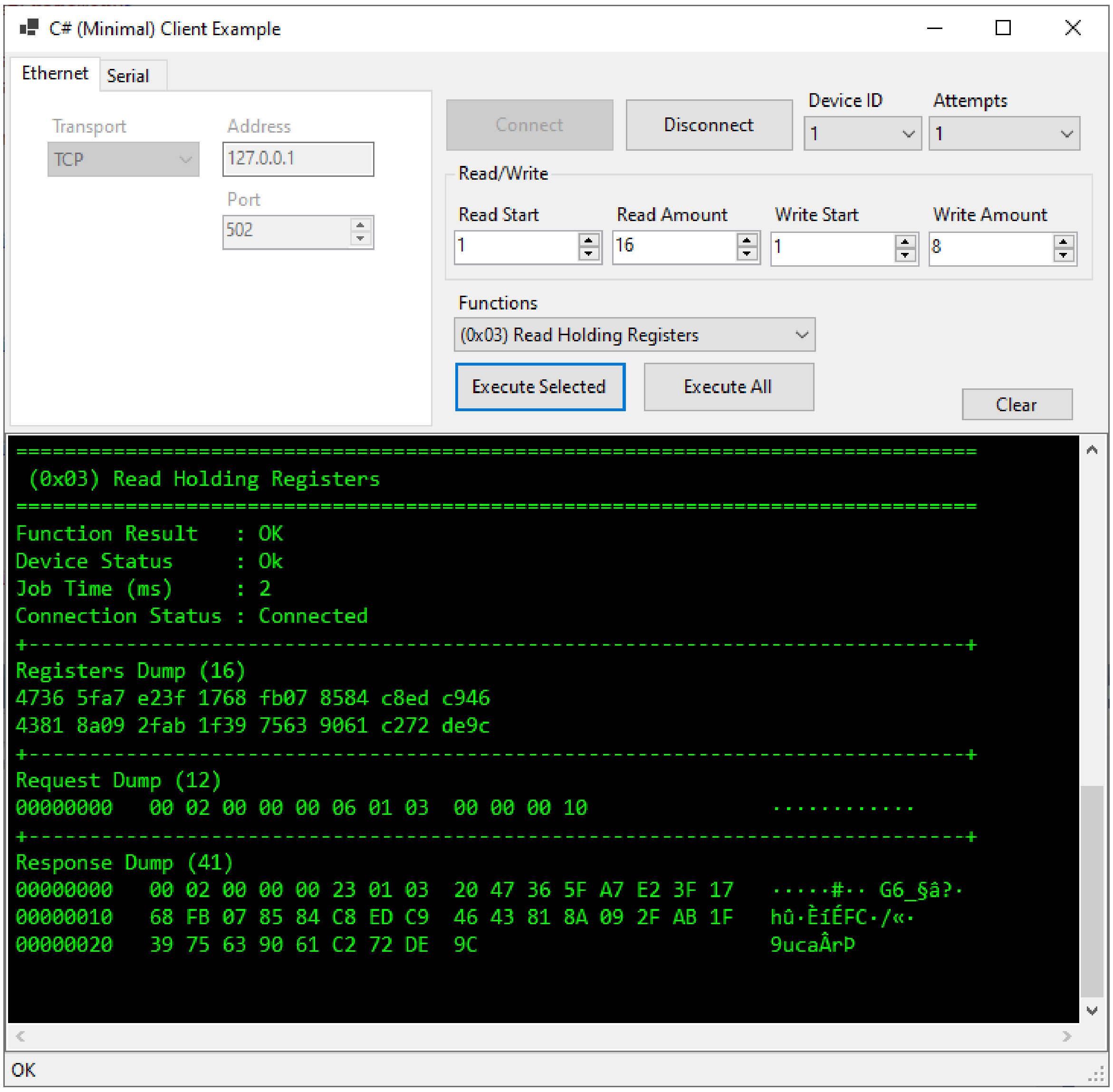


```
Device created
--------------
Device ID : 1
DataLink  : Ethernet
Params    : 127.0.0.1 : 502

Connected
Enter any char to terminate...

Indication received (141 Byte)
------------------------------------------------------------------------
0x0000: 00 05 00 00 00 87 01 10 00 00 00 40 80 08 49 88 ...........@..I.
0x0010: 9d 80 89 9e 59 67 d3 17 6d 6d 77 03 07 7d ed 93 ....Yg╙.mmw..}φ.
0x0020: 7b 74 d1 c6 bf 12 88 17 39 dc c3 cb f6 55 37 4e {t╤╞┐...9▄├╦÷U7N
0x0030: 7b be f9 21 83 c2 a1 55 a9 bf 1c 08 b1 44 8b e8 {╛·!.┬íU⌐┐..▒D.Φ
0x0040: 9a 27 cd b9 22 61 8d 63 c3 e1 f3 ad 7f 65 75 65 .'═╣"a.c├ß≤¡.eue
0x0050: 31 c7 3f 0b 33 b2 82 d1 64 3f ab 57 4b 19 ec 7d 1╟?.3▓.╤d?½WK.∞}
0x0060: c2 19 a6 b3 fa 6f 21 dd e3 03 48 09 fb 02 ae 85 ┬.ª│·o!▌π.H.√.«.
0x0070: 3c 13 07 de 4e 44 1d f3 fa a9 56 df 3c a4 a3 0a <..▐ND.≤·⌐V▀<ñú.
0x0080: a2 4e 99 0c b9 10 11 eb 29 72 74 63 59          óN..╣..δ)rtcY
------------------------------------------------------------------------
Confirmation sent (12 Byte)
------------------------------------------------------------------------
0x0000: 00 05 00 00 00 06 01 10 00 00 00 40             ...........@
------------------------------------------------------------------------
```

C# WinForm example



Transport: TCP
Address: 127.0.0.1
Port: 502
Device ID: 1
Attempts: 1
Read Start: 1
Read Amount: 16
Write Start: 1
Write Amount: 8
Functions: (0x03) Read Holding Registers

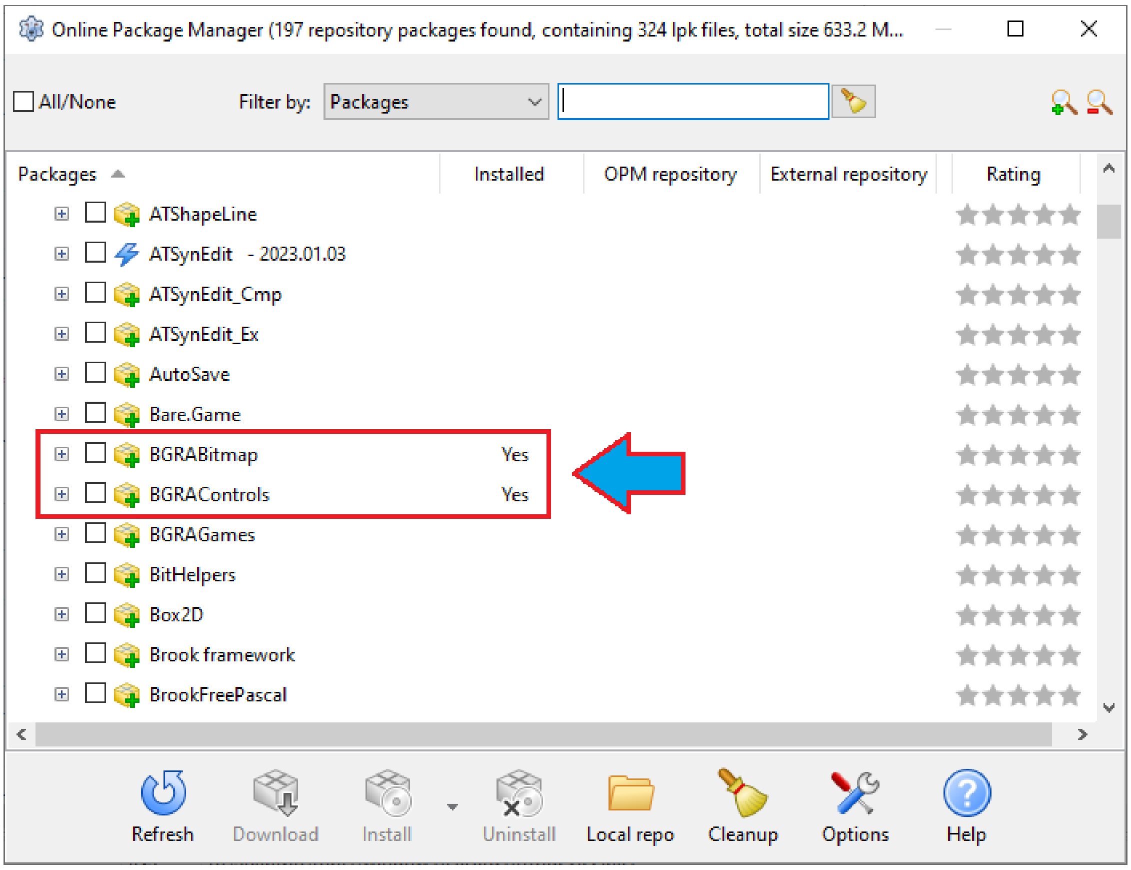
```
==============================================================================
 (0x03) Read Holding Registers
==============================================================================
Function Result   : OK
Device Status     : Ok
Job Time (ms)     : 2
Connection Status : Connected
+----------------------------------------------------------------------------+
Registers Dump (16)
4736 5fa7 e23f 1768 fb07 8584 c8ed c946
4381 8a09 2fab 1f39 7563 9061 c272 de9c
+----------------------------------------------------------------------------+
Request Dump (12)
00000000   00 02 00 00 00 06 01 03  00 00 00 10            ............
+----------------------------------------------------------------------------+
Response Dump (41)
00000000   00 02 00 00 00 23 01 03  20 47 36 5F A7 E2 3F 17  ·····#·· G6_§â?·
00000010   68 FB 07 85 84 C8 ED C9  46 43 81 8A 09 2F AB 1F  hû·ÈíÉFC·/«·
00000020   39 75 63 90 61 C2 72 DE  9C                       9ucaÂrÞ
```

OK

Tools

The tools are **big demos** that you can study or use whether or not you intend to work with the SnapModbus library.
They are written in Lazarus and the source code is provided.

If you want to rebuild them, you need **BGRAControls** and **BGRABitmaps** packages, which can be installed via Online package manager of Lazarus.



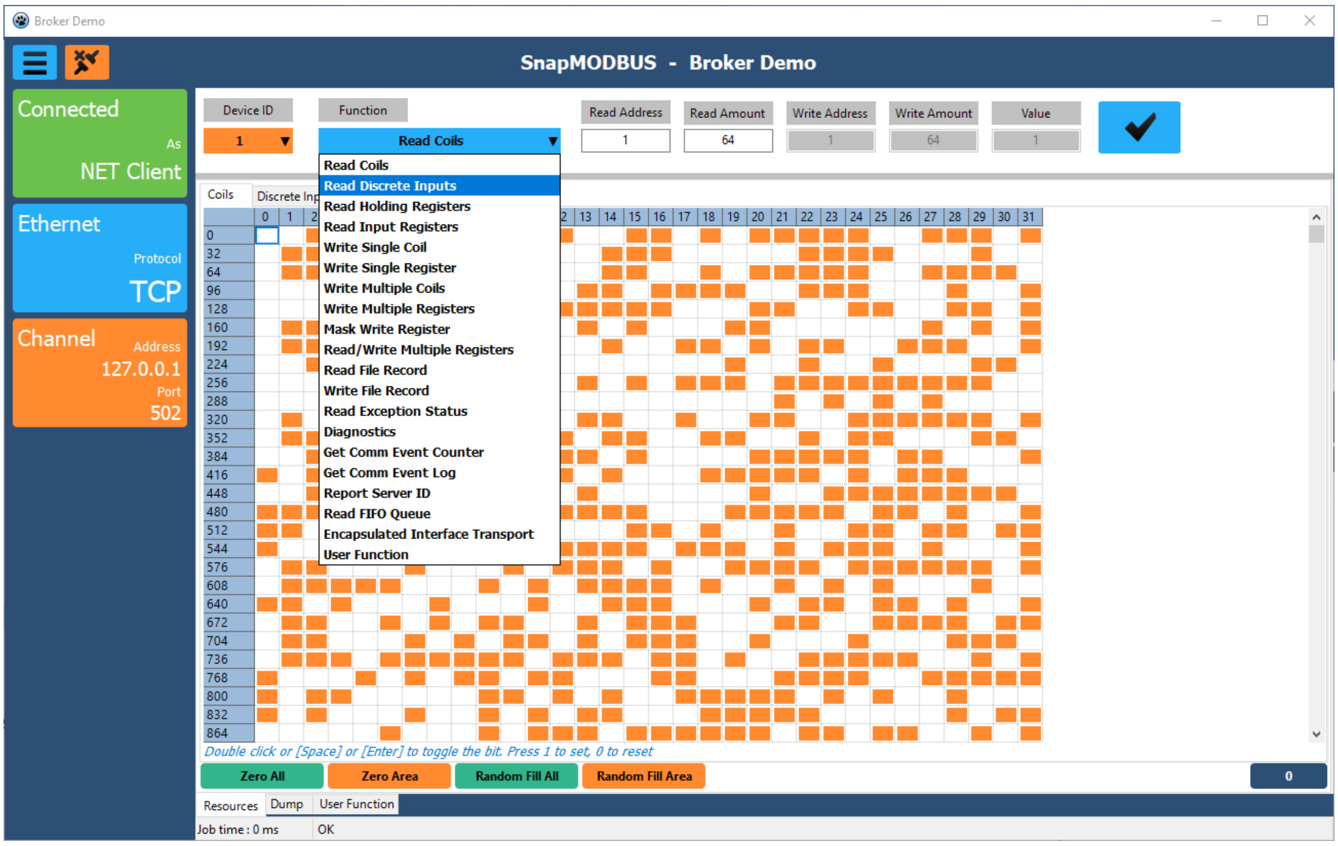
Online Package Manager (197 repository packages found, containing 324 lpk files, total size 633.2 M...

All/None   Filter by: Packages

Packages	Installed	OPM repository	External repository	Rating
ATShapeLine				
ATSynEdit - 2023.01.03				
ATSynEdit_Cmp				
ATSynEdit_Ex				
AutoSave				
Bare.Game				
BGRABitmap	Yes			
BGRAControls	Yes			
BGRAGames				
BitHelpers				
Box2D				
Brook framework				
BrookFreePascal				

WinBroker

Simulates a Broker (Ethernet or Serial), you can do almost everything with it, including the build of custom functions.



Broker Demo

SnapMODBUS - Broker Demo

Connected As NET Client

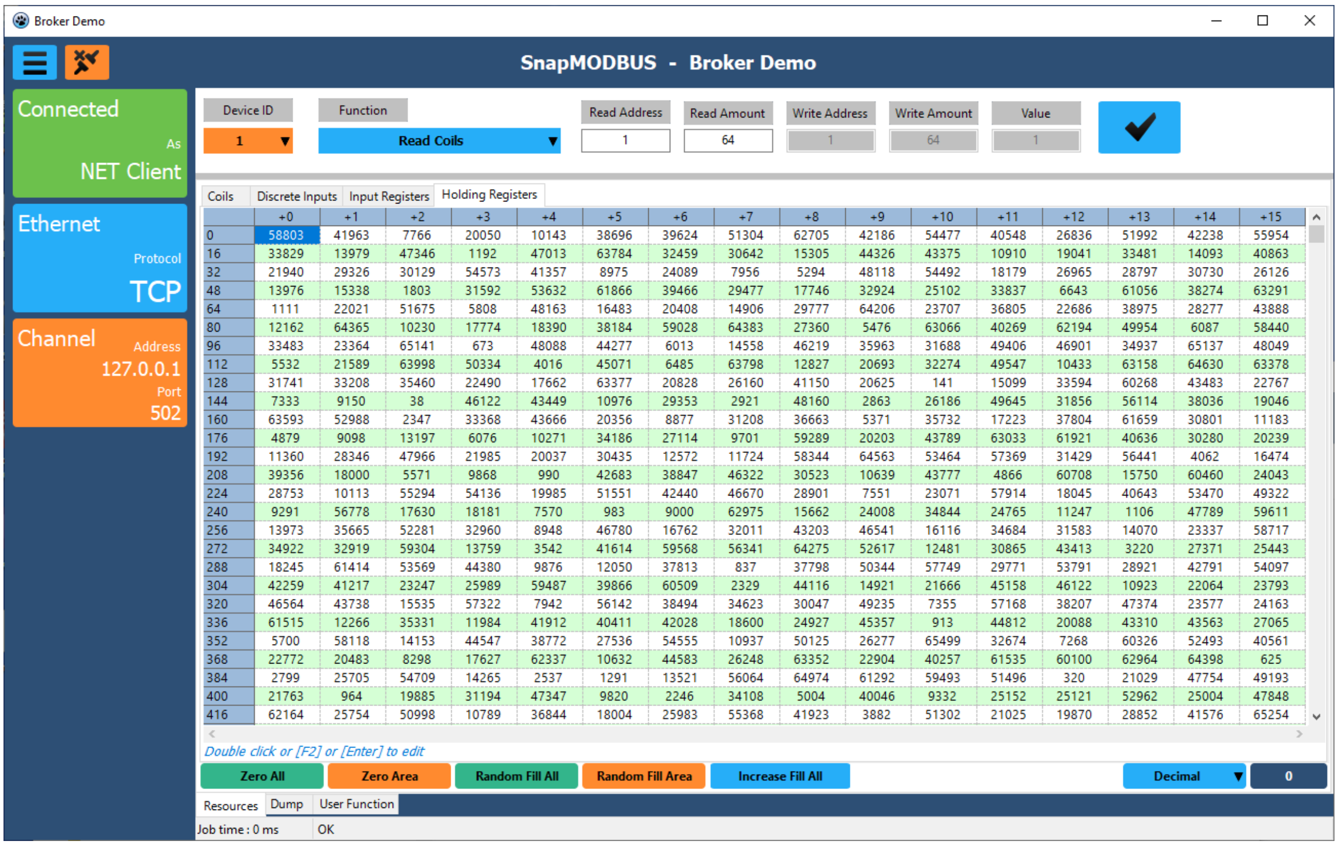
Ethernet Protocol TCP

Channel Address 127.0.0.1 Port 502

Device ID	Function	Read Address	Read Amount	Write Address	Write Amount	Value
1	Read Coils	1	64	1	64	1

Double click or [Space] or [Enter] to toggle the bit. Press 1 to set, 0 to reset

Job time : 0 ms   OK

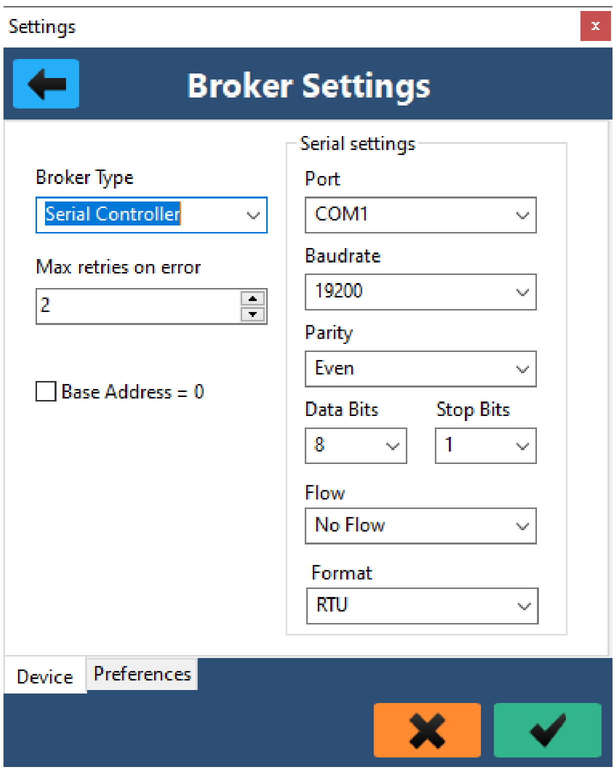


Device ID: 1
Function: Read Coils
Read Address: 1
Read Amount: 64
Write Address: 1
Write Amount: 64
Value: 1

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
0	58803	41963	7766	20050	10143	38696	39624	51304	62705	42186	54477	40548	26836	51992	42238	55954
16	33829	13979	47346	1192	47013	63784	32459	30642	15305	44326	43375	10910	19041	33481	14093	40863
32	21940	29326	30129	54573	41357	8975	24089	7956	5294	48118	54492	18179	26965	28797	30730	26126
48	13976	15338	1803	31592	53632	61866	39466	29477	17746	32924	25102	33837	6643	61056	38274	63291
64	1111	22021	51675	5808	48163	16483	20408	14906	29777	64206	23707	36805	22686	38975	28277	43888
80	12162	64365	10230	17774	18390	38184	59028	64383	27360	5476	63066	40269	62194	49954	6087	58440
96	33483	23364	65141	673	48088	44277	6013	14558	46219	35963	31688	49406	46901	34937	65137	48049
112	5532	21589	63998	50334	4016	45071	6485	63798	12827	20693	32274	49547	10433	63158	64630	63378
128	31741	33208	35460	22490	17662	63377	20828	26160	41150	20625	141	15099	33594	60268	43483	22767
144	7333	9150	38	46122	43449	10976	29353	2921	48160	2863	26186	49645	31856	56114	38036	19046
160	63593	52988	2347	33368	43666	20356	8877	31208	36663	5371	35732	17223	37804	61659	30801	11183
176	4879	9098	13197	6076	10271	34186	27114	9701	59289	20203	43789	63033	61921	40636	30280	20239
192	11360	28346	47966	21985	20037	30435	12572	11724	58344	64563	53464	57369	31429	56441	4062	16474
208	39356	18000	5571	9868	990	42683	38847	46322	30523	10639	43777	4866	60708	15750	60460	24043
224	28753	10113	55294	54136	19985	51551	42440	46670	28901	7551	23071	57914	18045	40643	53470	49322
240	9291	56778	17630	18181	7570	983	9000	62975	15662	24008	34844	24765	11247	1106	47789	59611
256	13973	35665	52281	32960	8948	46780	16762	32011	43203	46541	16116	34684	31583	14070	23337	58717
272	34922	32919	59304	13759	3542	41614	59568	56341	64275	52617	12481	30865	43413	3220	27371	25443
288	18245	61414	53569	44380	9876	12050	37813	837	37798	50344	57749	29771	53791	28921	42791	54097
304	42259	41217	23247	25989	59487	39866	60509	2329	44116	14921	21666	45158	46122	10923	22064	23793
320	46564	43738	15535	57322	7942	56142	38494	34623	30047	49235	7355	57168	38207	47374	23577	24163
336	61515	12266	35331	11984	41912	40411	42028	18600	24927	45357	913	44812	20088	43310	43563	27065
352	5700	58118	14153	44547	38772	27536	54555	10937	50125	26277	65499	32674	7268	60326	52493	40561
368	22772	20483	8298	17627	62337	10632	44583	26248	63352	22904	40257	61535	60100	62964	64398	625
384	2799	25705	54709	14265	2537	1291	13521	56064	64974	61292	59493	51496	320	21029	47754	49193
400	21763	964	19885	31194	47347	9820	2246	34108	5004	40046	9332	25152	25121	52962	25004	47848
416	62164	25754	50998	10789	36844	18004	25983	55368	41923	3882	51302	21025	19870	28852	41576	65254

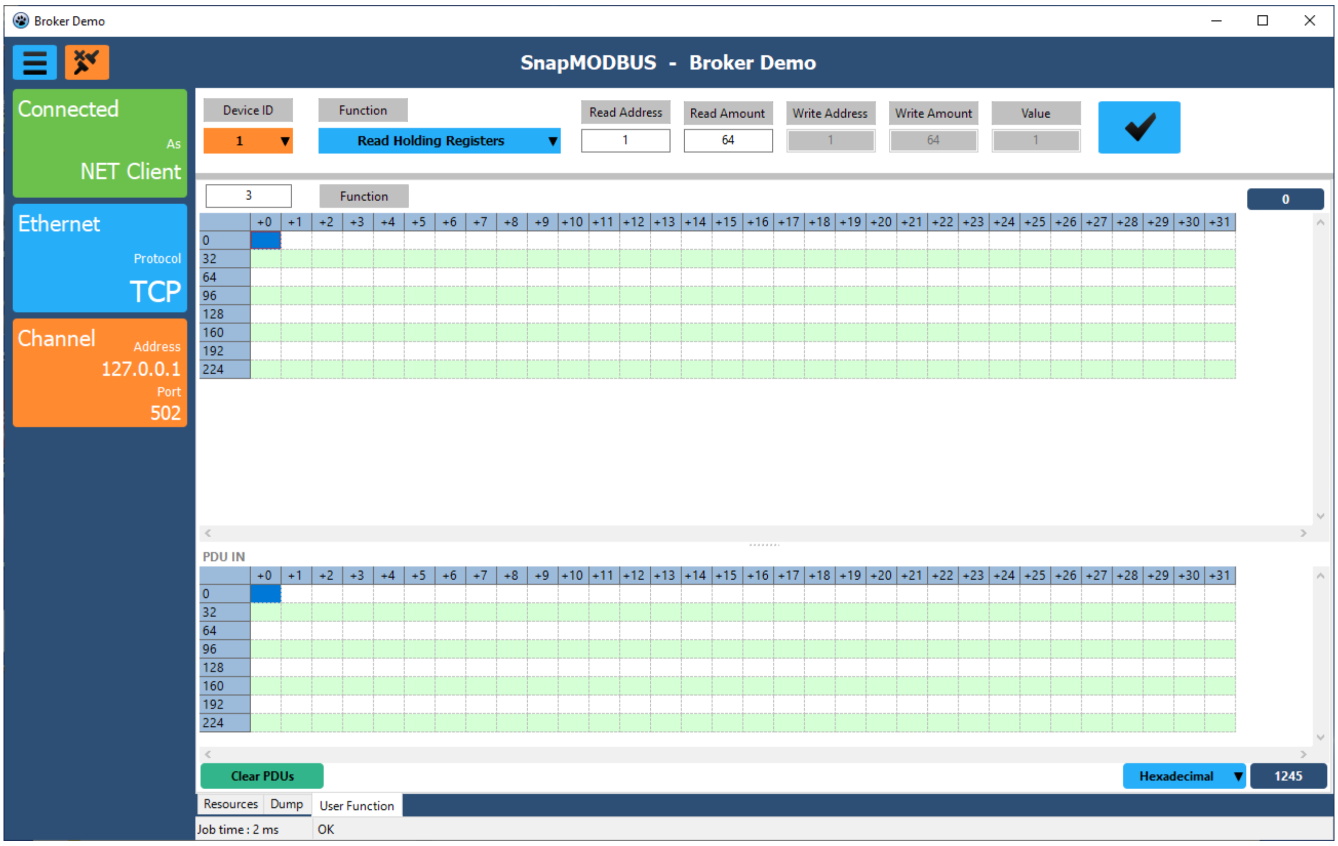
Double click or [F2] or [Enter] to edit

Job time : 0 ms   OK



Broker Type: Serial Controller
Max retries on error: 2
☐ Base Address = 0

Serial settings
Port: COM1
Baudrate: 19200
Parity: Even
Data Bits: 8
Stop Bits: 1
Flow: No Flow
Format: RTU


Broker Demo

SnapMODBUS - Broker Demo

Connected As NET Client

Ethernet Protocol TCP

Channel Address 127.0.0.1 Port 502

Device ID: 1
Function: Read Holding Registers
Read Address: 1
Read Amount: 64
Write Address: 1
Write Amount: 64
Value: 1

Function: 3

PDU IN

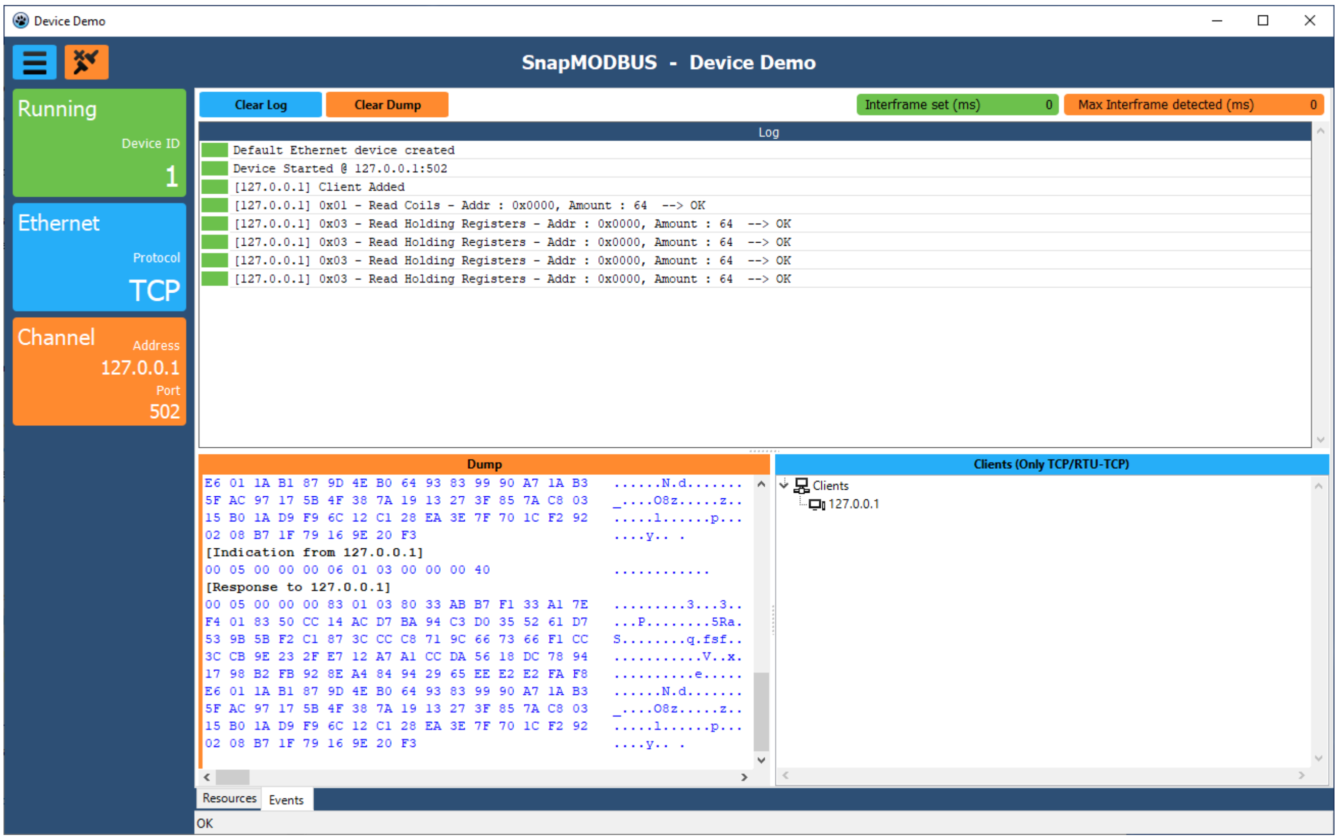
Clear PDUs | Hexadecimal | 1245

Resources | Dump | User Function

Job time : 2 ms | OK

WinDevice

Simulates a Device (Ethernet or Serial).


Device Demo

SnapMODBUS - Device Demo

Running Device ID 1

Ethernet Protocol TCP

Channel Address 127.0.0.1 Port 502

Clear Log | Clear Dump | Interframe set (ms) 0 | Max Interframe detected (ms) 0

Log

```
Default Ethernet device created
Device Started @ 127.0.0.1:502
[127.0.0.1] Client Added
[127.0.0.1] 0x01 - Read Coils - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x03 - Read Holding Registers - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x03 - Read Holding Registers - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x03 - Read Holding Registers - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x03 - Read Holding Registers - Addr : 0x0000, Amount : 64  --> OK
```

Dump

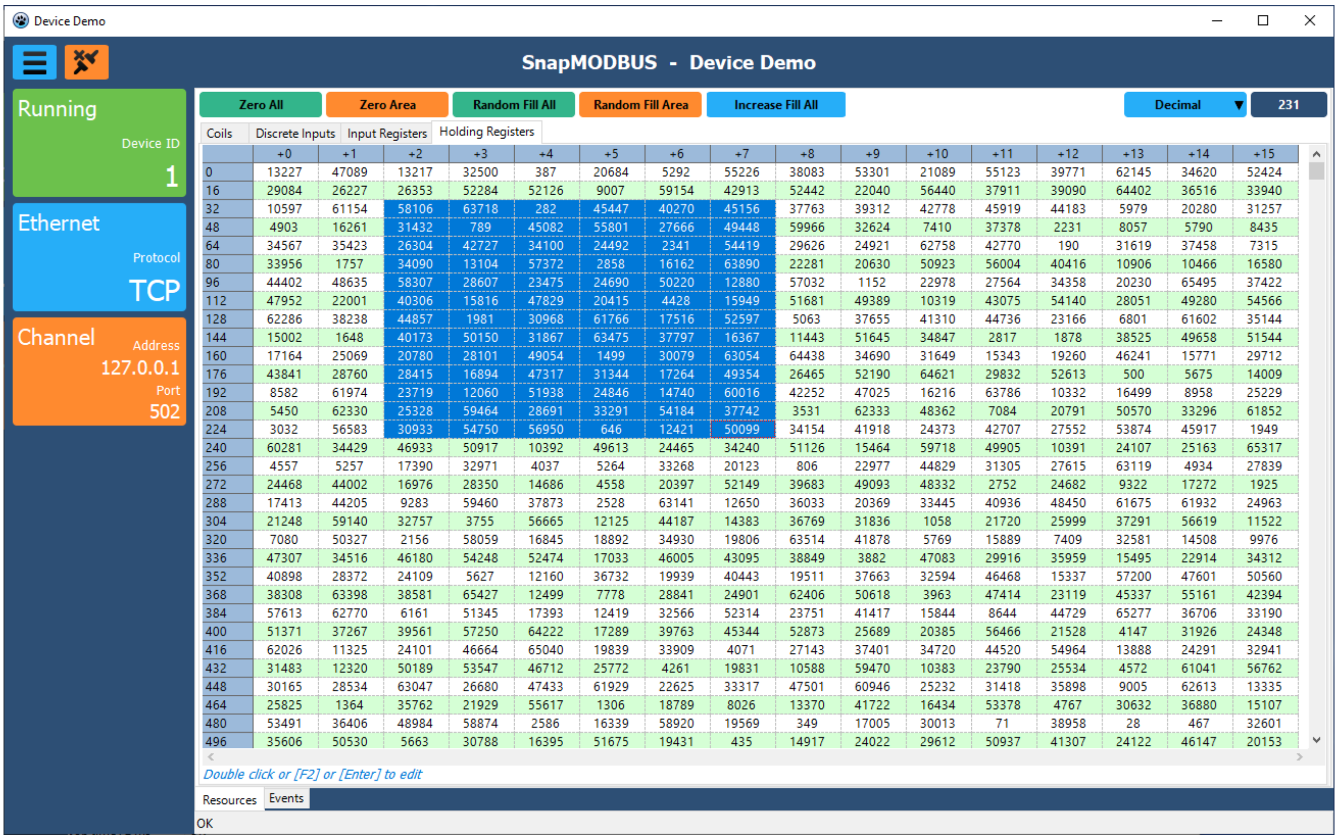
```
E6 01 1A B1 87 9D 4E B0 64 93 83 99 90 A7 1A B3   ......N.d.......
5F AC 97 17 5B 4F 38 7A 19 13 27 3F 85 7A C8 03   _....O8z.....z..
15 B0 1A D9 F9 6C 12 C1 28 EA 3E 7F 70 1C F2 92   .....l......p...
02 08 B7 1F 79 16 9E 20 F3                        ....y.. .
[Indication from 127.0.0.1]
00 05 00 00 00 06 01 03 00 00 00 40               ............
[Response to 127.0.0.1]
00 05 00 00 00 83 01 03 80 33 AB B7 F1 33 A1 7E   .........3...3..
F4 01 83 50 CC 14 AC D7 BA 94 C3 D0 35 52 61 D7   ...P........5Ra.
53 9B 5B F2 C1 87 3C CC C8 71 9C 66 73 66 F1 CC   S........q.fsf..
3C CB 9E 23 2F E7 12 A7 A1 CC DA 56 18 DC 78 94   ...........V..x.
17 98 B2 FB 92 8E A4 84 94 29 65 EE E2 E2 FA F8   ..........e.....
E6 01 1A B1 87 9D 4E B0 64 93 83 99 90 A7 1A B3   ......N.d.......
5F AC 97 17 5B 4F 38 7A 19 13 27 3F 85 7A C8 03   _....O8z.....z..
15 B0 1A D9 F9 6C 12 C1 28 EA 3E 7F 70 1C F2 92   .....l......p...
02 08 B7 1F 79 16 9E 20 F3                        ....y.. .
```

Clients (Only TCP/RTU-TCP)

Clients
127.0.0.1

Resources | Events

OK



	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
0	13227	47089	13217	32500	387	20684	5292	55226	38083	53301	21089	55123	39771	62145	34620	52424
16	29084	26227	26353	52284	52126	9007	59154	42913	52442	22040	56440	37911	39090	64402	36516	33940
32	10597	61154	58106	63718	282	45447	40270	45156	37763	39312	42778	45919	44183	5979	20280	31257
48	4903	16261	31432	789	45082	55801	27666	49448	59966	32624	7410	37378	2231	8057	5790	8435
64	34567	35423	26304	42727	34100	24492	2341	54419	29626	24921	62758	42770	190	31619	37458	7315
80	33956	1757	34090	13104	57372	2858	16162	63890	22281	20630	50923	56004	40416	10906	10466	16580
96	44402	48635	58307	28607	23475	24690	50220	12880	57032	1152	22978	27564	34358	20230	65495	37422
112	47952	22001	40306	15816	47829	20415	4428	15949	51681	49389	10319	43075	54140	28051	49280	54566
128	62286	38238	44857	1981	30968	61766	17516	52597	5063	37655	41310	44736	23166	6801	61602	35144
144	15002	1648	40173	50150	31867	63475	37797	16367	11443	51645	34847	2817	1878	38525	49658	51544
160	17164	25069	20780	28101	49054	1499	30079	63054	64438	34690	31649	15343	19260	46241	15771	29712
176	43841	28760	28415	16894	47317	31344	17264	49354	26465	52190	64621	29832	52613	500	5675	14009
192	8582	61974	23719	12060	51938	24846	14740	60016	42252	47025	16216	63786	10332	16499	8958	25229
208	5450	62330	25328	59464	28691	33291	54184	37742	3531	62333	48362	7084	20791	50570	33296	61852
224	3032	56583	30933	54750	56950	646	12421	50099	34154	41918	24373	42707	27552	53874	45917	1949
240	60281	34429	46933	50917	10392	49613	24465	34240	51126	15464	59718	49905	10391	24107	25163	65317
256	4557	5257	17390	32971	4037	5264	33268	20123	806	22977	44829	31305	27615	63119	4934	27839
272	24468	44002	16976	28350	14686	4558	20397	52149	39683	49093	48332	2752	24682	9322	17272	1925
288	17413	44205	9283	59460	37873	2528	63141	12650	36033	20369	33445	40936	48450	61675	61932	24963
304	21248	59140	32757	3755	56665	12125	44187	14383	36769	31836	1058	21720	25999	37291	56619	11522
320	7080	50327	2156	58059	16845	18892	34930	19806	63514	41878	5769	15889	7409	32581	14508	9976
336	47307	34516	46180	54248	52474	17033	46005	43095	38849	3882	47083	29916	35959	15495	22914	34312
352	40898	28372	24109	5627	12160	36732	19939	40443	19511	37663	32594	46468	15337	57200	47601	50560
368	38308	63398	38581	65427	12499	7778	28841	24901	62406	50618	3963	47414	23119	45337	55161	42394
384	57613	62770	6161	51345	17393	12419	32566	52314	23751	41417	15844	8644	44729	65277	36706	33190
400	51371	37267	39561	57250	64222	17289	39763	45344	52873	25689	20385	56466	21528	4147	31926	24348
416	62026	11325	24101	46664	65040	19839	33909	4071	27143	37401	34720	44520	54964	13888	24291	32941
432	31483	12320	50189	53547	46712	25772	4261	19831	10588	59470	10383	23790	25534	4572	61041	56762
448	30165	28534	63047	26680	47433	61929	22625	33317	47501	60946	25232	31418	35898	9005	62613	13335
464	25825	1364	35762	21929	55617	1306	18789	8026	13370	41722	16434	53378	4767	30632	36880	15107
480	53491	36406	48984	58874	2586	16339	58920	19569	349	17005	30013	71	38958	28	467	32601
496	35606	50530	5663	30788	16395	51675	19431	435	14917	24022	29612	50937	41307	24122	46147	20153

Double click or [F2] or [Enter] to edit

OK

LabVIEW

NI LabVIEW is a software for systems design which uses a graphical language, named "G" (not to be confused with the more pleasant G-point), to build complex laboratory and automation applications.

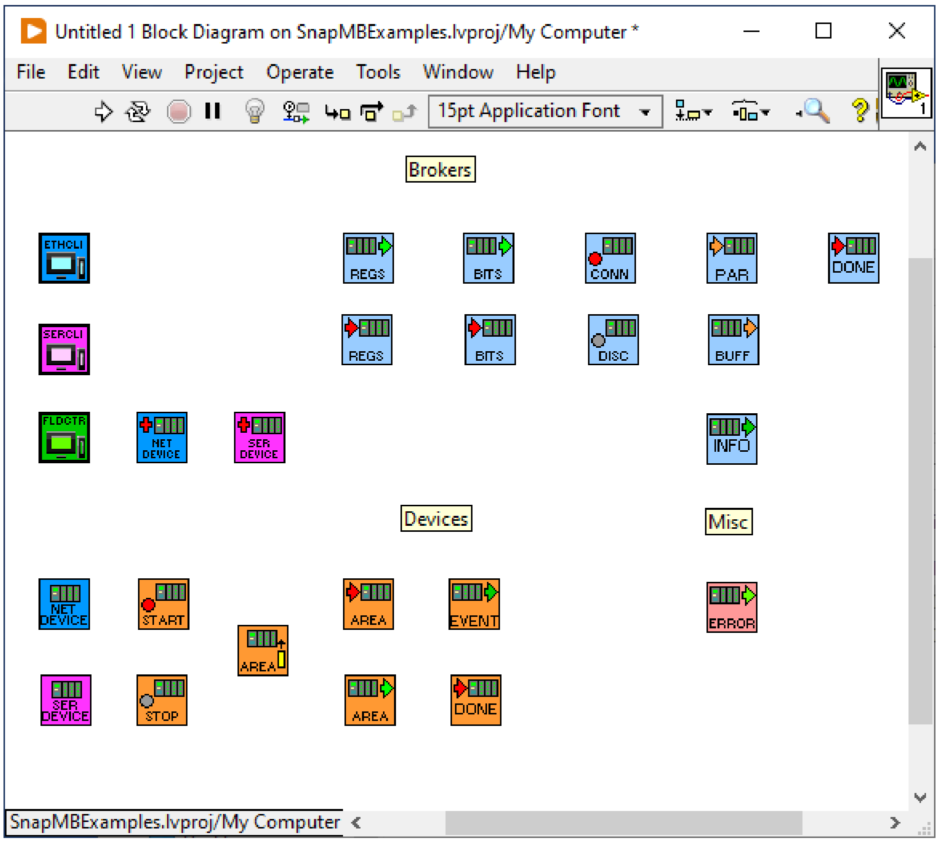
In a G program, the execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available.

LabVIEW offers the same data types/structures as other programming languages, but they are propagated using a different approach.

From this point of view we can consider G as a managed language.

Let’s see how can we interface LabVIEW with SnapModbus, keeping in mind these two major differences (execution and data storage) against the traditional programming languages.

The wrapper provided consists of a series of VI, each of them encapsulates a SnapMB function via the Call Library function node.



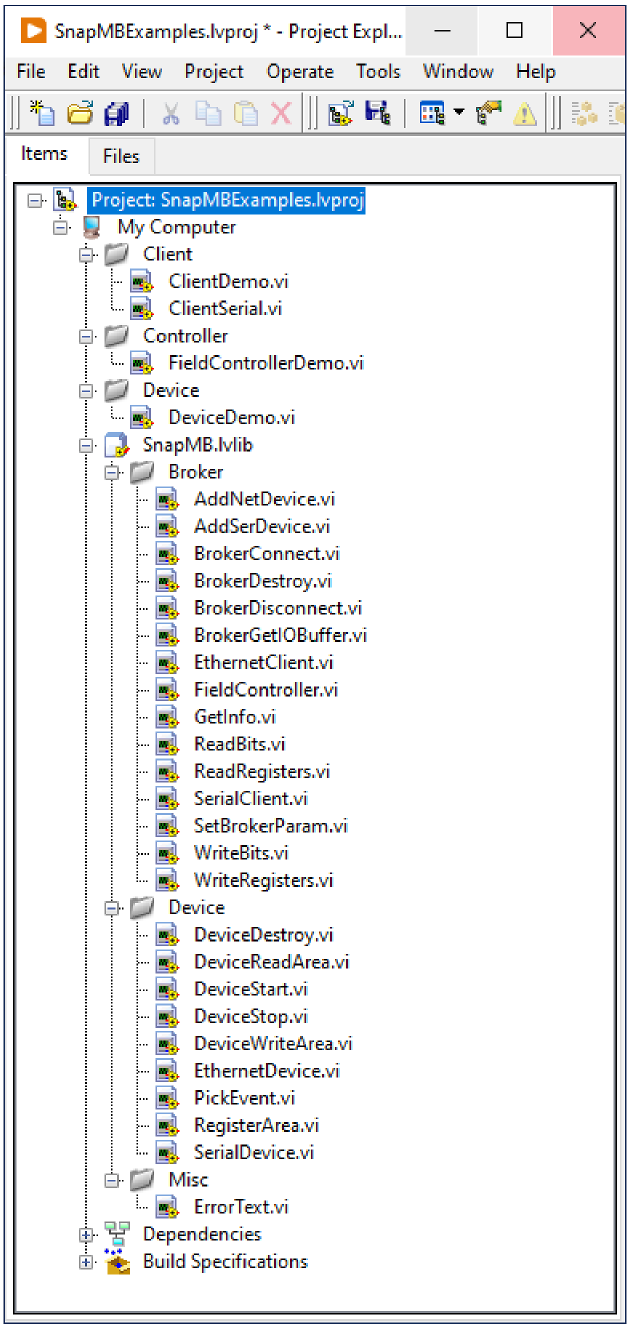
Not all SnapMB functions have been wrapped because some of them **can't be used** (see below) and others are of more sporadic use, the fact remains that you can include them very easily using the VIs you find already written as guidelines.

However, all the functions for data exchange towards Coils, Holding registers, Discrete inputs and Input registers have been implemented, both on the broker and on the device side.

LabVIEW cannot handle the callbacks, i.e. is not possible to call LV pseudo code, from an external binary library.
For this reason, the only way to work with a Device is using the **shared resources**, for which, the RegisterArea and ReadArea VI are supplied.

> ✎ Note
> All SnapModbus functions are thread-safe, so LabVIEW VI are set to "run in any thread".

under `/LabVIEW/Examples/` folder you will find **SnapMBExamples.lvproj**
which contains the demos and **SnapMB.lvlib** i.e. the VI library.

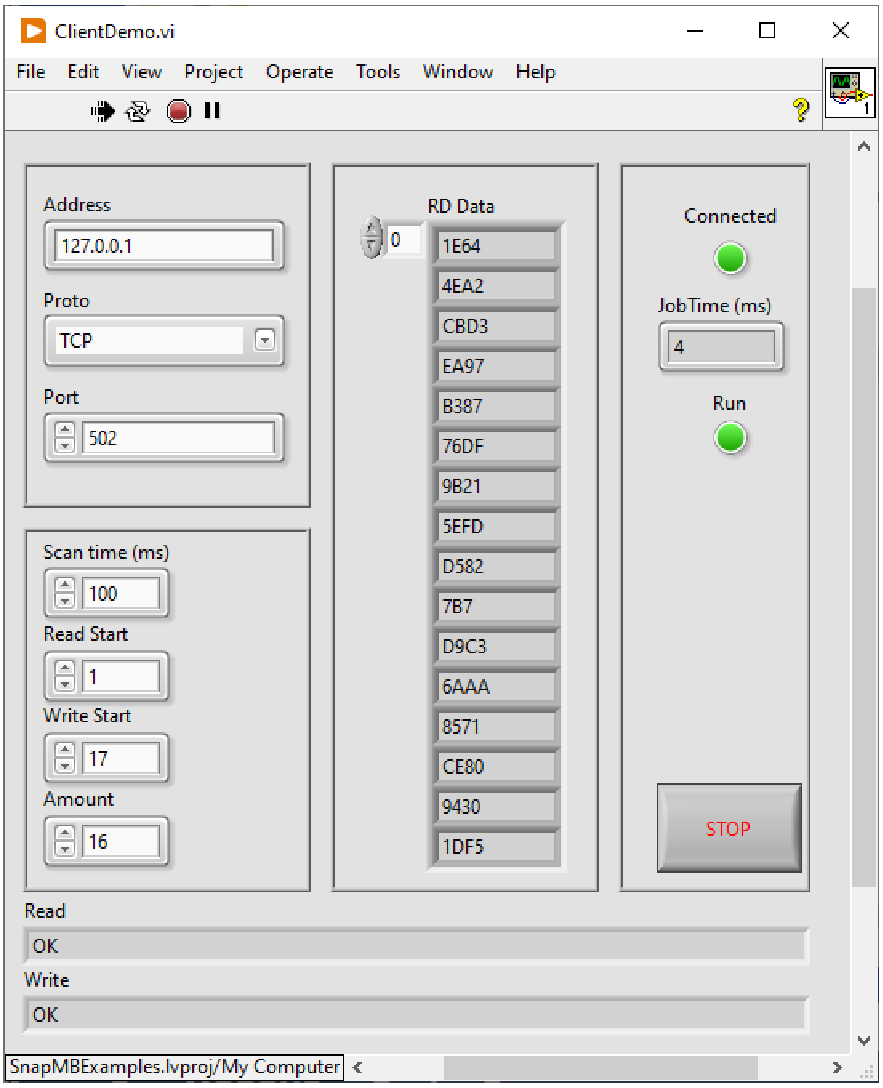


SnapMBExamples.lvproj * - Project Expl...

```
Project: SnapMBExamples.lvproj
  My Computer
    Client
      ClientDemo.vi
      ClientSerial.vi
    Controller
      FieldControllerDemo.vi
    Device
      DeviceDemo.vi
    SnapMB.lvlib
      Broker
        AddNetDevice.vi
        AddSerDevice.vi
        BrokerConnect.vi
        BrokerDestroy.vi
        BrokerDisconnect.vi
        BrokerGetIOBuffer.vi
        EthernetClient.vi
        FieldController.vi
        GetInfo.vi
        ReadBits.vi
        ReadRegisters.vi
        SerialClient.vi
        SetBrokerParam.vi
        WriteBits.vi
        WriteRegisters.vi
      Device
        DeviceDestroy.vi
        DeviceReadArea.vi
        DeviceStart.vi
        DeviceStop.vi
        DeviceWriteArea.vi
        EthernetDevice.vi
        PickEvent.vi
        RegisterArea.vi
        SerialDevice.vi
      Misc
        ErrorText.vi
    Dependencies
    Build Specifications
```

Let's see some examples with their diagrams. To run them, if you don't have real hardware ready, you can use **WinBroker** and **WinDevice** tools.

ClientDemo


ClientDemo.vi
File Edit View Project Operate Tools Window Help
STOP
SnapMBExamples.lvproj/My Computer

Address: 127.0.0.1

Proto: TCP

Port: 502

Scan time (ms): 100

Read Start: 1

Write Start: 17

Amount: 16

RD Data: 0

RD Data
1E64
4EA2
CBD3
EA97
B387
76DF
9B21
5EFD
D582
7B7
D9C3
6AAA
8571
CE80
9430
1DF5

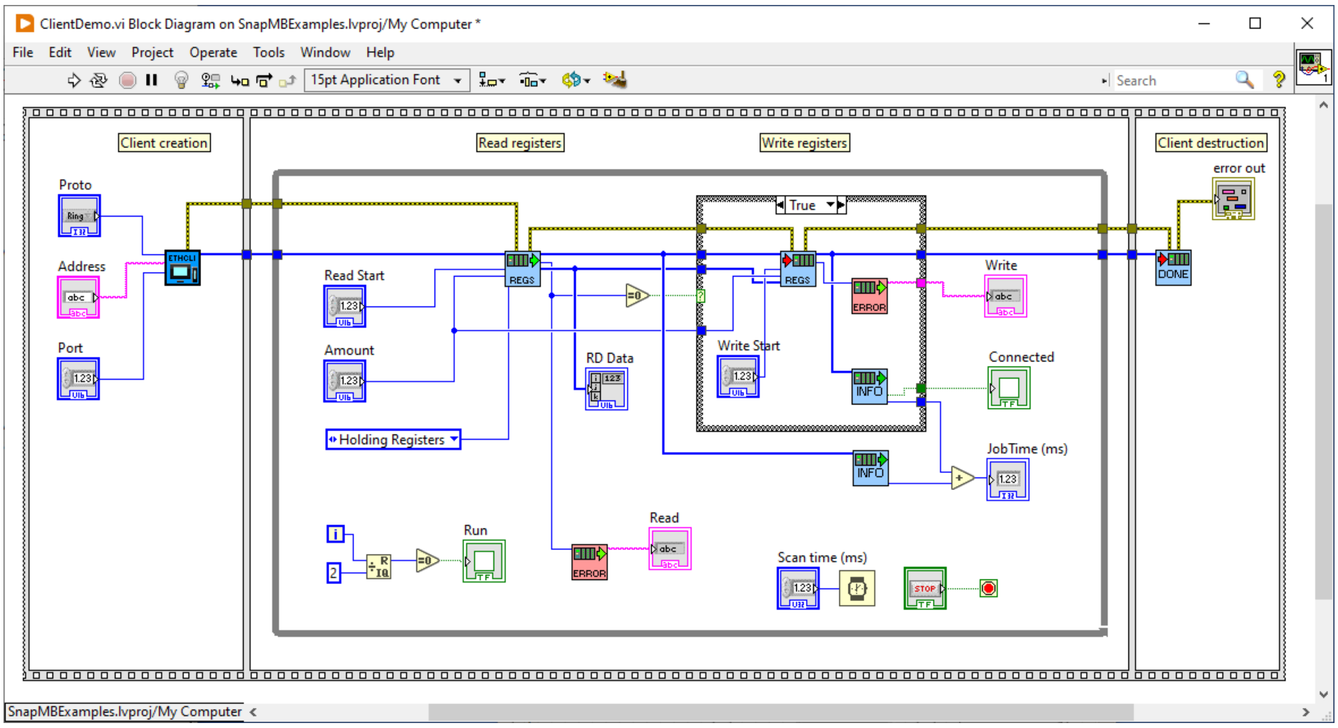
Connected: ●

JobTime (ms): 4

Run: ●

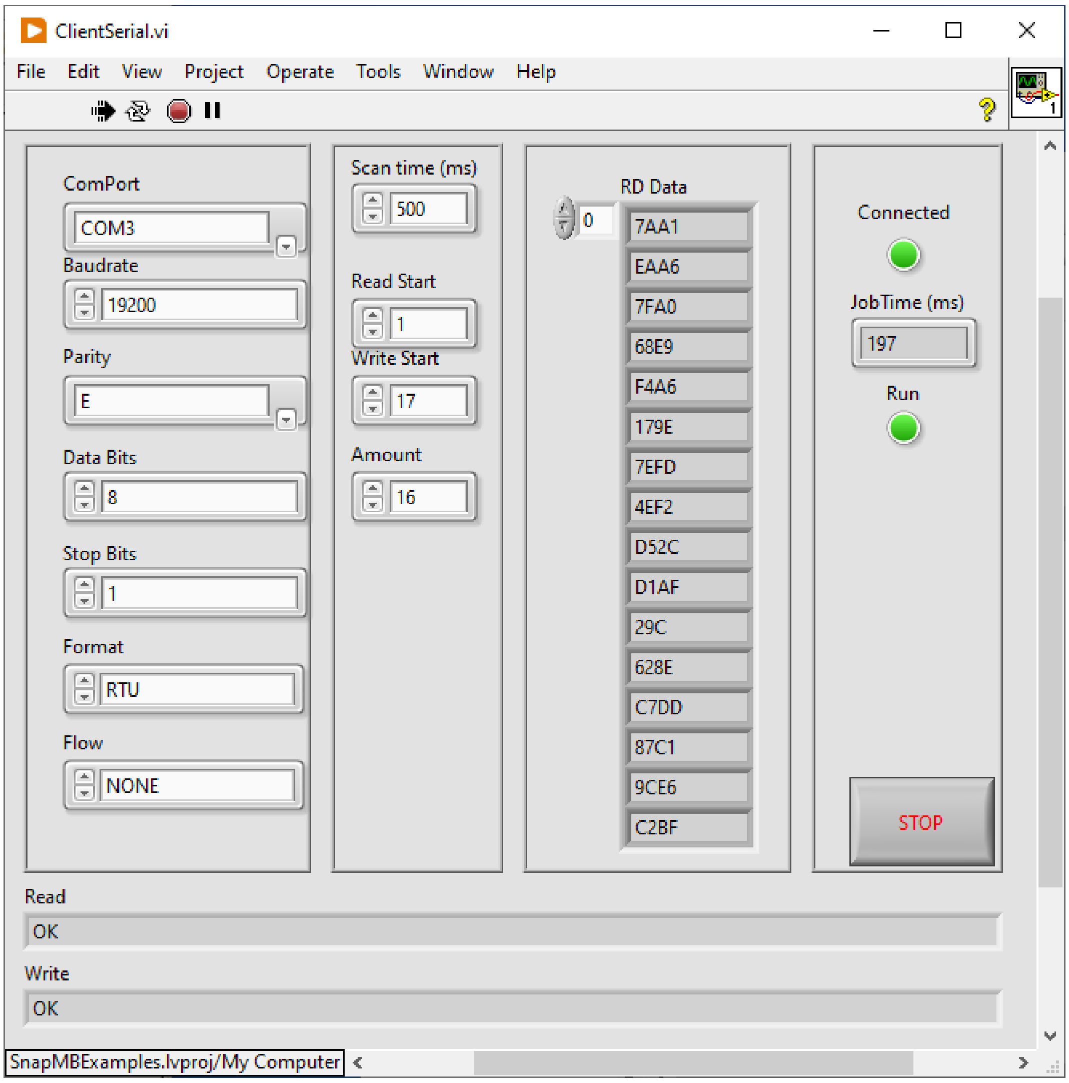
Read: OK

Write: OK



ClientSerial

Same as above, but using a Serial Client.


ClientSerial.vi
File Edit View Project Operate Tools Window Help
STOP
SnapMBExamples.lvproj/My Computer

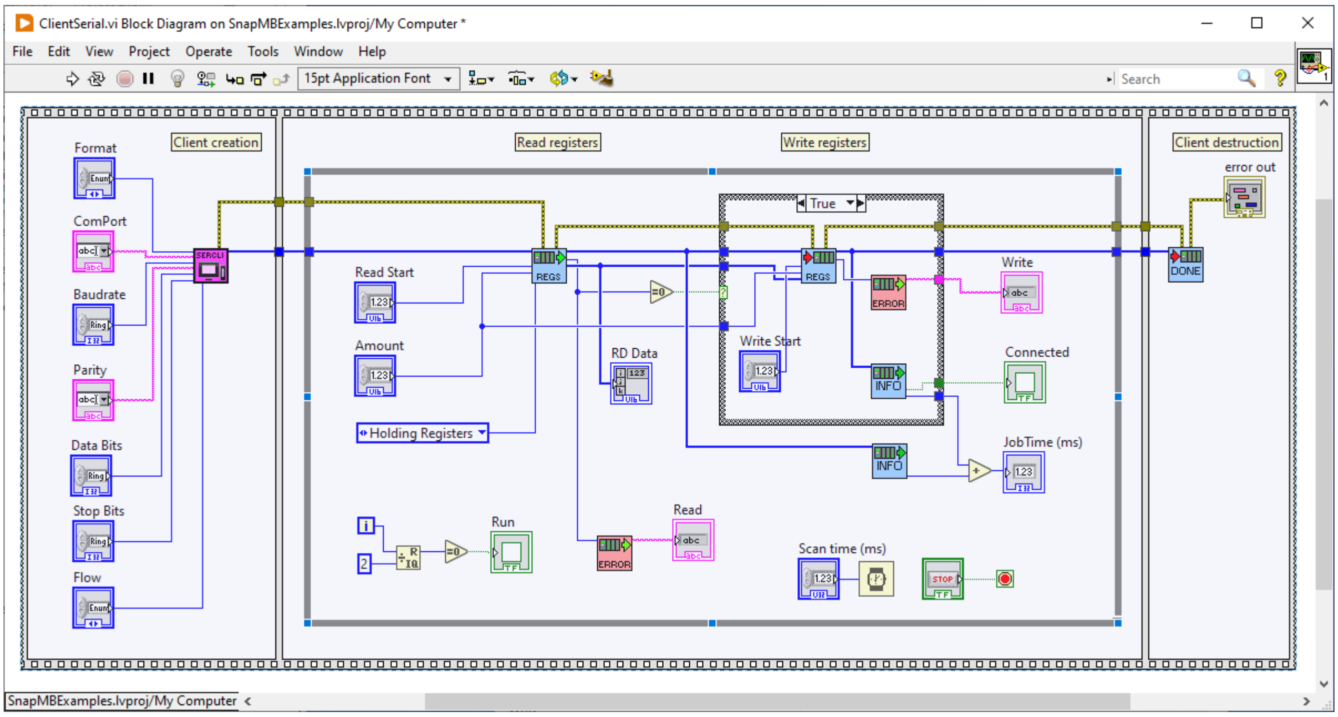
ComPort: COM3
Baudrate: 19200
Parity: E
Data Bits: 8
Stop Bits: 1
Format: RTU
Flow: NONE

Scan time (ms): 500
Read Start: 1
Write Start: 17
Amount: 16

RD Data: 0
7AA1
EAA6
7FA0
68E9
F4A6
179E
7EFD
4EF2
D52C
D1AF
29C
628E
C7DD
87C1
9CE6
C2BF

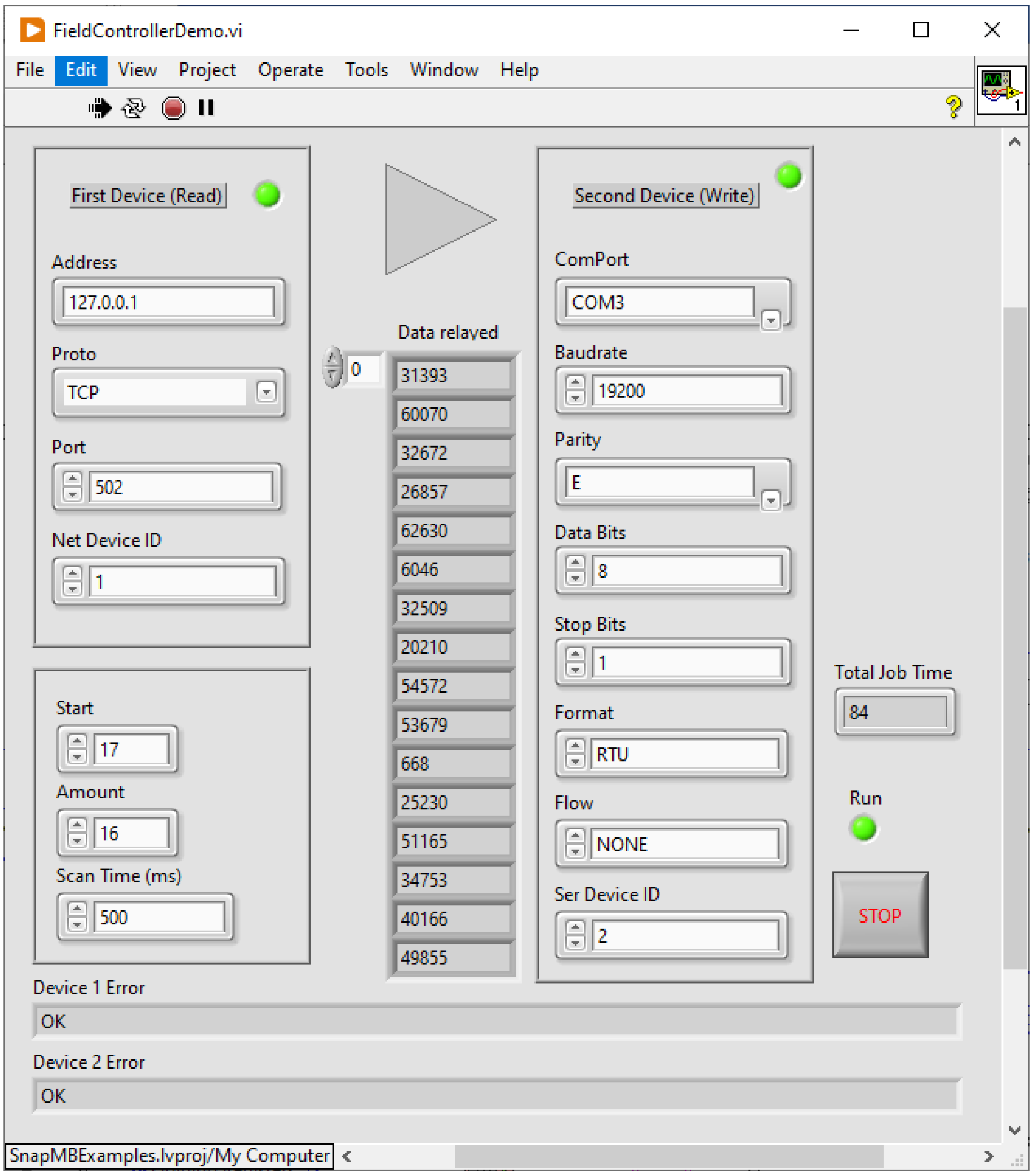
Connected
JobTime (ms): 197
Run

Read: OK
Write: OK



FieldControllerDemo

Two Devices are added, the first Ethernet and the second Serial, look at FieldController usage for more info.
The program read some registers from the first device and write them to the second.


FieldControllerDemo.vi

File Edit View Project Operate Tools Window Help

First Device (Read)

Address: 127.0.0.1

Proto: TCP

Port: 502

Net Device ID: 1

Start: 17

Amount: 16

Scan Time (ms): 500

Data relayed: 0

31393
60070
32672
26857
62630
6046
32509
20210
54572
53679
668
25230
51165
34753
40166
49855

Second Device (Write)

ComPort: COM3

Baudrate: 19200

Parity: E

Data Bits: 8

Stop Bits: 1

Format: RTU

Flow: NONE

Ser Device ID: 2

Total Job Time: 84

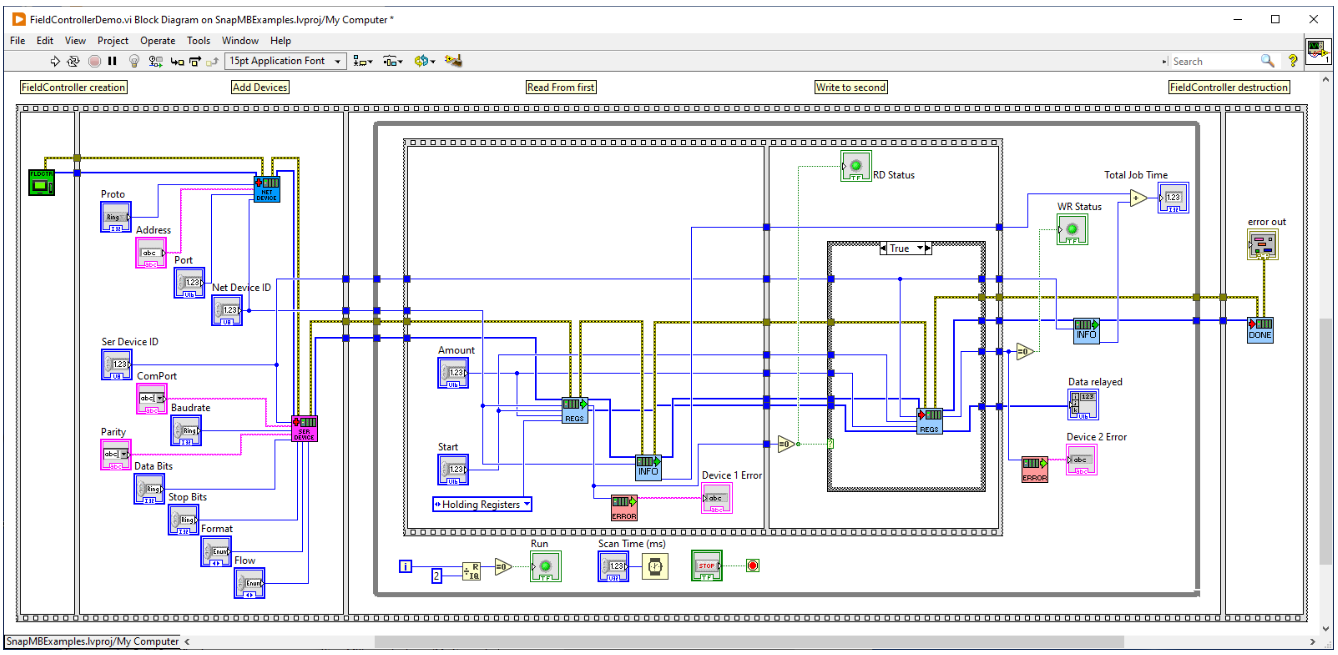
Run

STOP

Device 1 Error: OK

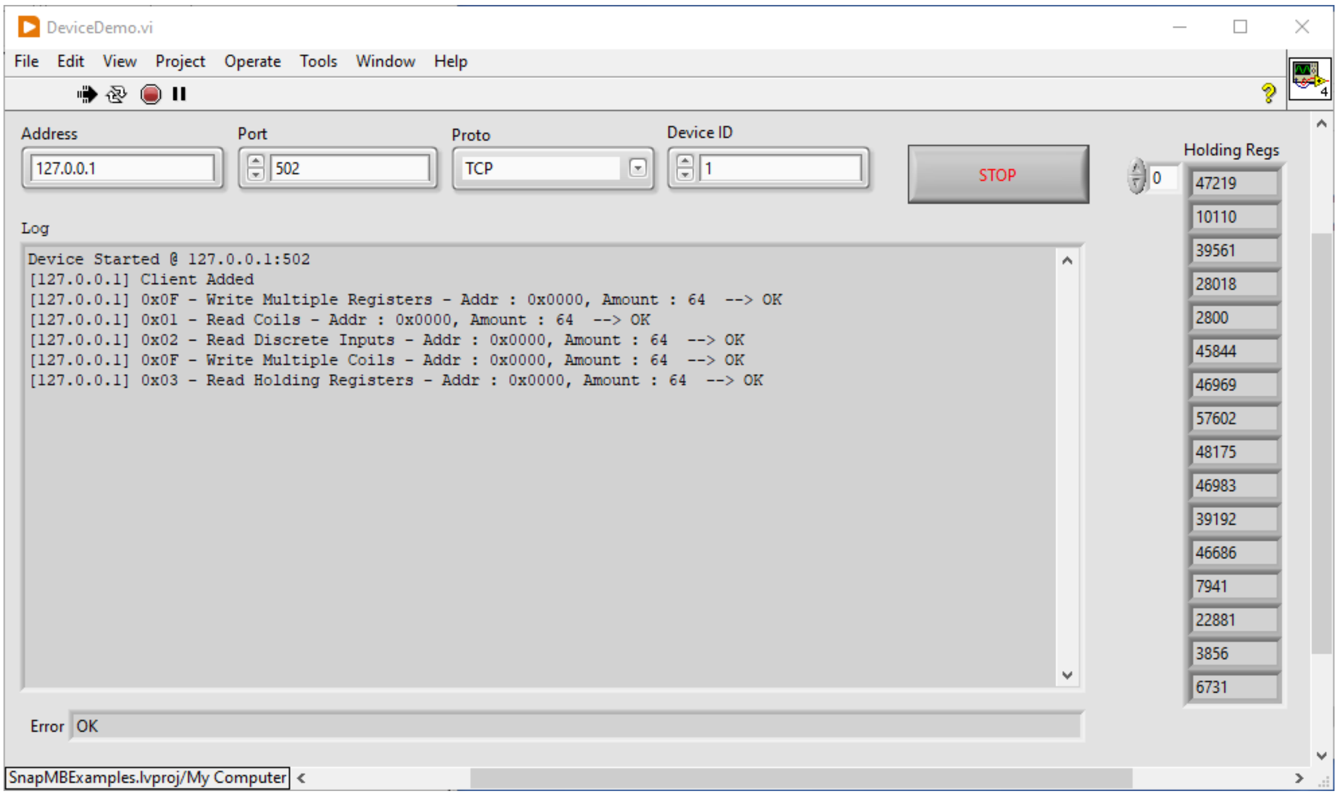
Device 2 Error: OK

SnapMBExamples.lvproj/My Computer



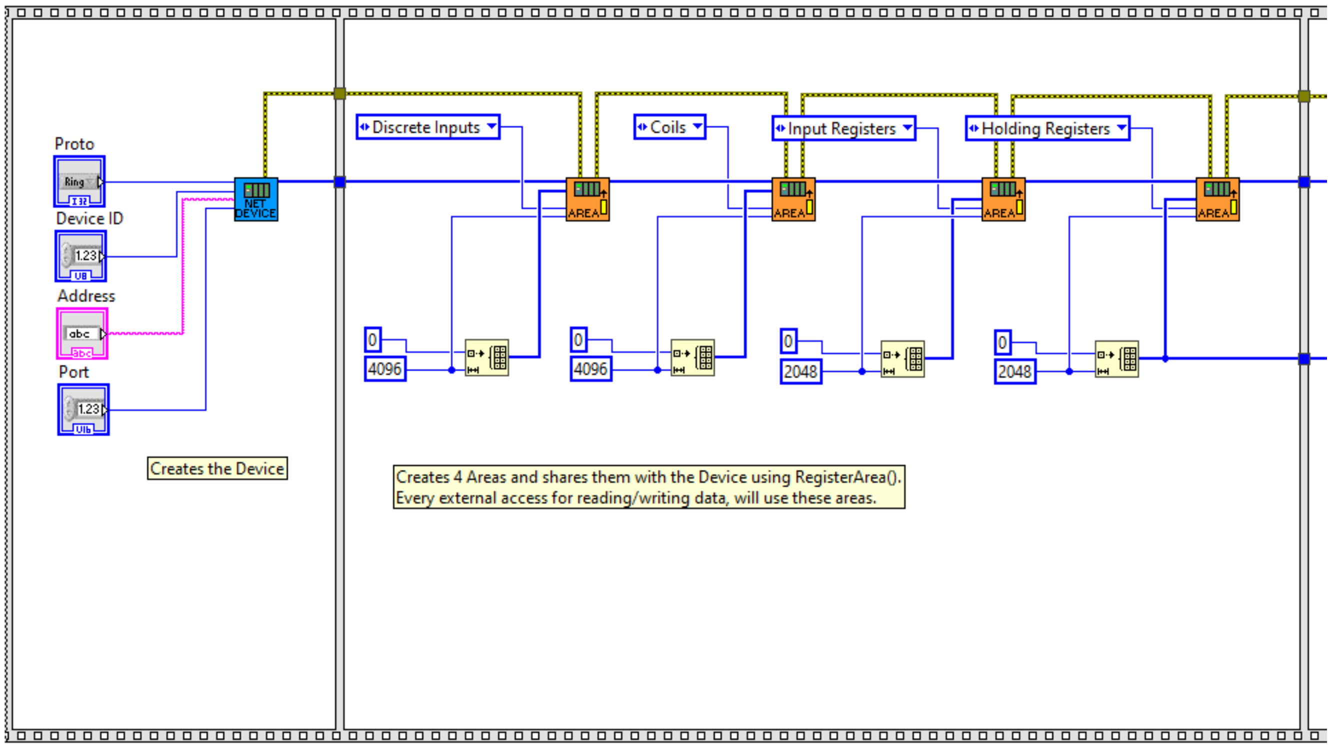
DeviceDemo

A simple Device which shares the resources and uses ReadArea to access them safely (see Concurrency)

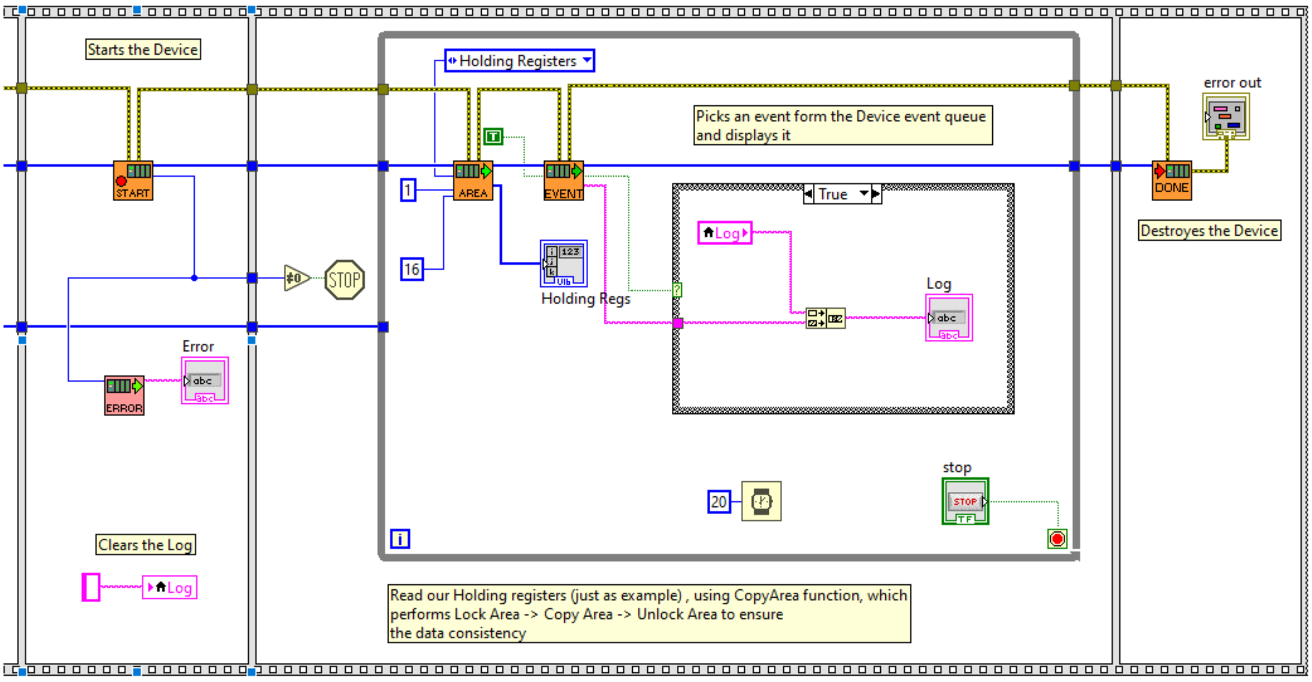


```
Device Started @ 127.0.0.1:502
[127.0.0.1] Client Added
[127.0.0.1] 0x0F - Write Multiple Registers - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x01 - Read Coils - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x02 - Read Discrete Inputs - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x0F - Write Multiple Coils - Addr : 0x0000, Amount : 64  --> OK
[127.0.0.1] 0x03 - Read Holding Registers - Addr : 0x0000, Amount : 64  --> OK
```

I have divided the diagram into two parts for better readability.



Part 1



Picks an event form the Device event queue and displays it

Read our Holding registers (just as example) , using CopyArea function, which performs Lock Area -> Copy Area -> Unlock Area to ensure the data consistency

part 2

Rebuild SnapModbus

SnapModbus is written in "flat" C++, i.e., without dependencies. There is a snap_platform.h file with some "definitions" to adjust the included files, but all sources are **the same** for all operating systems and platforms.

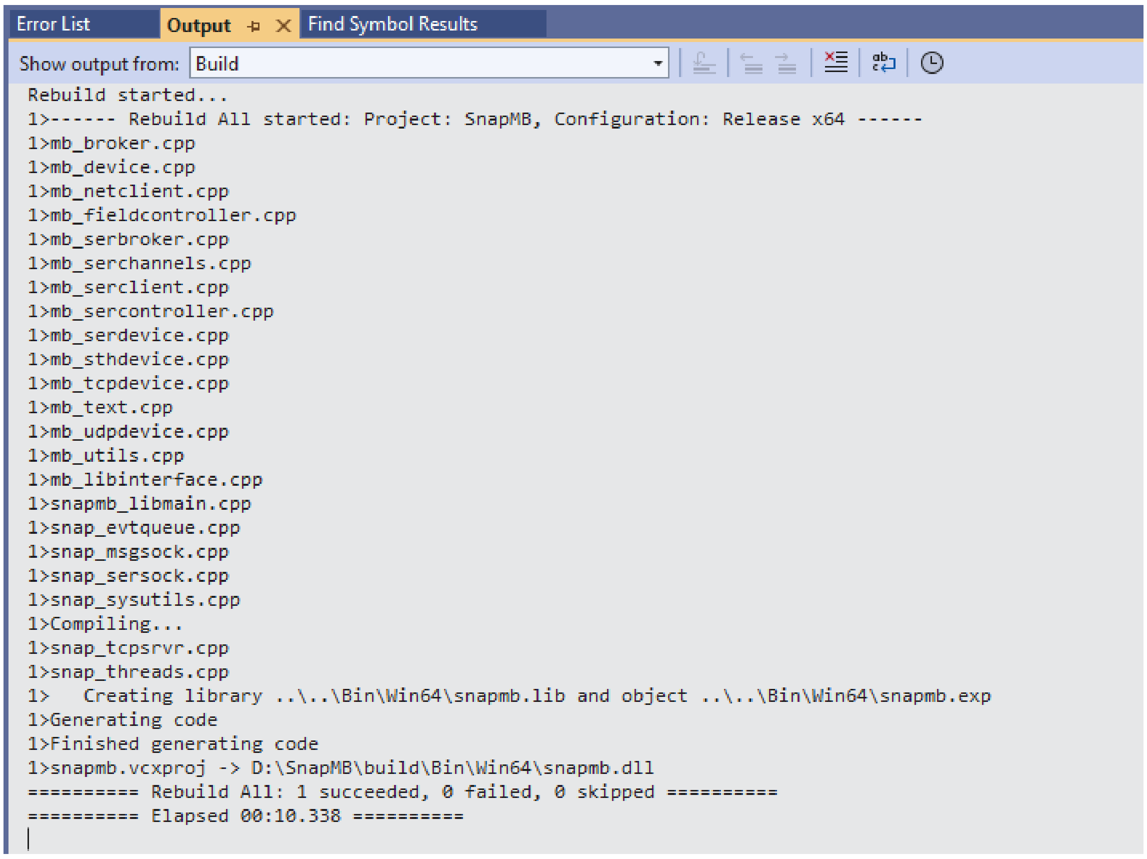
Windows

Visual Studio Community Edition

Into

`/build/windows/VisualStudio`

you will find **snapmb.sln**, just open it. It should be compatible also with preceding versions.



```
Rebuild started...
1>------ Rebuild All started: Project: SnapMB, Configuration: Release x64 ------
1>mb_broker.cpp
1>mb_device.cpp
1>mb_netclient.cpp
1>mb_fieldcontroller.cpp
1>mb_serbroker.cpp
1>mb_serchannels.cpp
1>mb_serclient.cpp
1>mb_sercontroller.cpp
1>mb_serdevice.cpp
1>mb_sthdevice.cpp
1>mb_tcpdevice.cpp
1>mb_text.cpp
1>mb_udpdevice.cpp
1>mb_utils.cpp
1>mb_libinterface.cpp
1>snapmb_libmain.cpp
1>snap_evtqueue.cpp
1>snap_msgsock.cpp
1>snap_sersock.cpp
1>snap_sysutils.cpp
1>Compiling...
1>snap_tcpsrvr.cpp
1>snap_threads.cpp
1>   Creating library ..\..\Bin\Win64\snapmb.lib and object ..\..\Bin\Win64\snapmb.exp
1>Generating code
1>Finished generating code
1>snapmb.vcxproj -> D:\SnapMB\build\Bin\Win64\snapmb.dll
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
========== Elapsed 00:10.338 ==========
```

MinGW

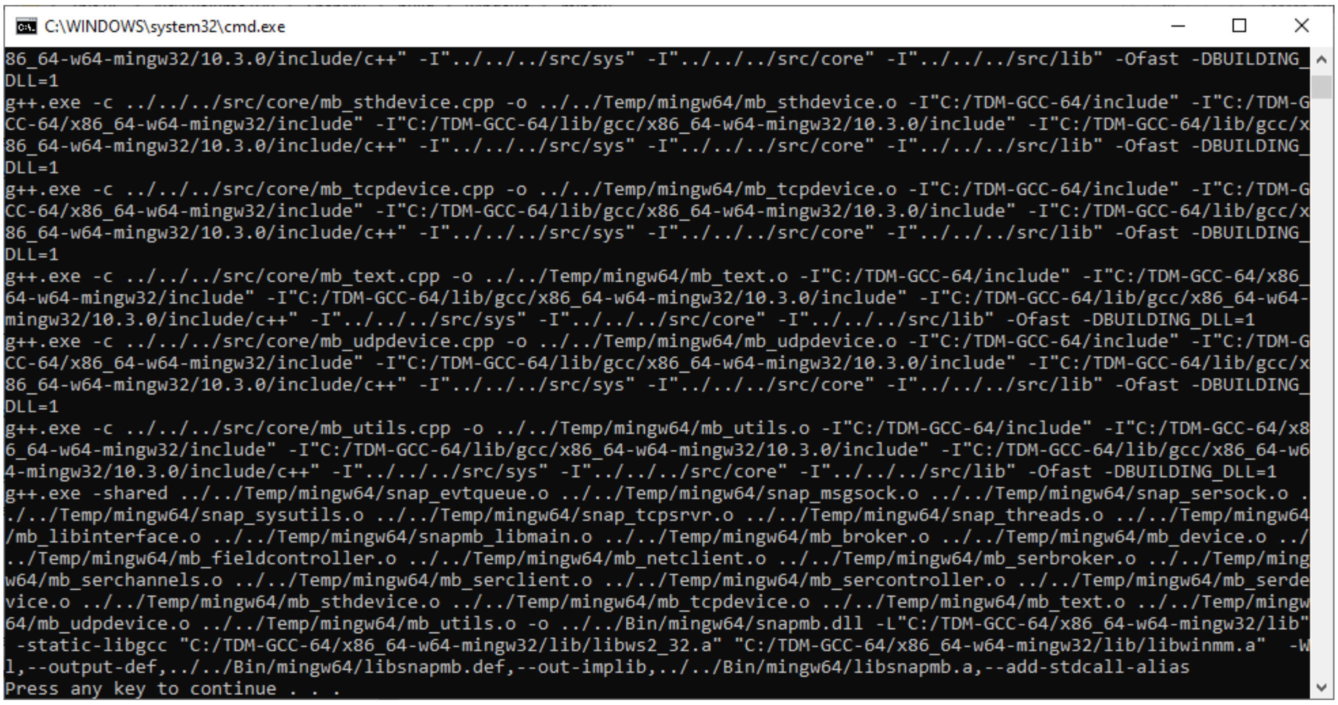
Go to

```
/build/windows/mingw
```

and run **build32** or **build64**,
a pause statement will allow to see the compilation result also if you run them by double-clicking in explorer.

I used TDM-GCC, you can download it from https://jmeubank.github.io/tdm-gcc/

Maybe you need to modify the batch files, specify the installation path and the release, currently I installed the release 10.3.0 in C:
The libraries will be generated into `\build\bin\mingw32` or `\build\bin\mingw64`

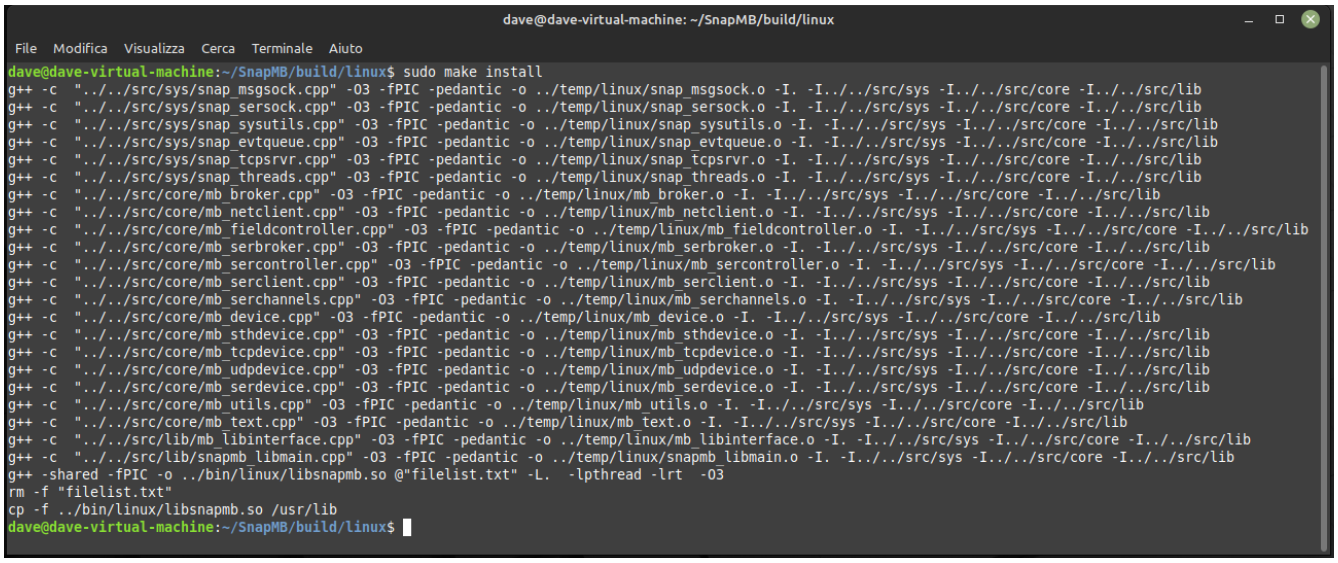


```
86_64-w64-mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_
DLL=1
g++.exe -c ../../../src/core/mb_sthdevice.cpp -o ../../Temp/mingw64/mb_sthdevice.o -I"C:/TDM-GCC-64/include" -I"C:/TDM-G
CC-64/x86_64-w64-mingw32/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/10.3.0/include" -I"C:/TDM-GCC-64/lib/gcc/x
86_64-w64-mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_
DLL=1
g++.exe -c ../../../src/core/mb_tcpdevice.cpp -o ../../Temp/mingw64/mb_tcpdevice.o -I"C:/TDM-GCC-64/include" -I"C:/TDM-G
CC-64/x86_64-w64-mingw32/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/10.3.0/include" -I"C:/TDM-GCC-64/lib/gcc/x
86_64-w64-mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_
DLL=1
g++.exe -c ../../../src/core/mb_text.cpp -o ../../Temp/mingw64/mb_text.o -I"C:/TDM-GCC-64/include" -I"C:/TDM-GCC-64/x86_
64-w64-mingw32/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/10.3.0/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-
mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_DLL=1
g++.exe -c ../../../src/core/mb_udpdevice.cpp -o ../../Temp/mingw64/mb_udpdevice.o -I"C:/TDM-GCC-64/include" -I"C:/TDM-G
CC-64/x86_64-w64-mingw32/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/10.3.0/include" -I"C:/TDM-GCC-64/lib/gcc/x
86_64-w64-mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_
DLL=1
g++.exe -c ../../../src/core/mb_utils.cpp -o ../../Temp/mingw64/mb_utils.o -I"C:/TDM-GCC-64/include" -I"C:/TDM-GCC-64/x8
6_64-w64-mingw32/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w64-mingw32/10.3.0/include" -I"C:/TDM-GCC-64/lib/gcc/x86_64-w6
4-mingw32/10.3.0/include/c++" -I"../../../src/sys" -I"../../../src/core" -I"../../../src/lib" -Ofast -DBUILDING_DLL=1
g++.exe -shared ../../Temp/mingw64/snap_evtqueue.o ../../Temp/mingw64/snap_msgsock.o ../../Temp/mingw64/snap_sersock.o .
./../Temp/mingw64/snap_sysutils.o ../../Temp/mingw64/snap_tcpsrvr.o ../../Temp/mingw64/snap_threads.o ../../Temp/mingw64
/mb_libinterface.o ../../Temp/mingw64/snapmb_libmain.o ../../Temp/mingw64/mb_broker.o ../../Temp/mingw64/mb_device.o ../
../Temp/mingw64/mb_fieldcontroller.o ../../Temp/mingw64/mb_netclient.o ../../Temp/mingw64/mb_serbroker.o ../../Temp/ming
w64/mb_serchannels.o ../../Temp/mingw64/mb_serclient.o ../../Temp/mingw64/mb_sercontroller.o ../../Temp/mingw64/mb_serde
vice.o ../../Temp/mingw64/mb_sthdevice.o ../../Temp/mingw64/mb_tcpdevice.o ../../Temp/mingw64/mb_text.o ../../Temp/mingw
64/mb_udpdevice.o ../../Temp/mingw64/mb_utils.o -o ../../Bin/mingw64/snapmb.dll -L"C:/TDM-GCC-64/x86_64-w64-mingw32/lib"
 -static-libgcc "C:/TDM-GCC-64/x86_64-w64-mingw32/lib/libws2_32.a" "C:/TDM-GCC-64/x86_64-w64-mingw32/lib/libwinmm.a"  -W
l,--output-def,../../Bin/mingw64/libsnapmb.def,--out-implib,../../Bin/mingw64/libsnapmb.a,--add-stdcall-alias
Press any key to continue . . .
```

Linux (All distros)

Go to `/build/linux`
and run

```
sudo make install
```



```
dave@dave-virtual-machine:~/SnapMB/build/linux$ sudo make install
g++ -c  "../../src/sys/snap_msgsock.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_msgsock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_sersock.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_sersock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_sysutils.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_sysutils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_evtqueue.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_evtqueue.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_tcpsrvr.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_tcpsrvr.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_threads.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snap_threads.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_broker.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_broker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_netclient.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_netclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_fieldcontroller.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_fieldcontroller.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serbroker.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_serbroker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_sercontroller.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_sercontroller.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serclient.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_serclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serchannels.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_serchannels.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_device.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_device.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_sthdevice.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_sthdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_tcpdevice.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_tcpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_udpdevice.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_udpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serdevice.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_serdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_utils.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_utils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_text.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_text.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/lib/mb_libinterface.cpp" -O3 -fPIC -pedantic -o ../temp/linux/mb_libinterface.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/lib/snapmb_libmain.cpp" -O3 -fPIC -pedantic -o ../temp/linux/snapmb_libmain.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -shared -fPIC -o ../bin/linux/libsnapmb.so @"filelist.txt" -L.  -lpthread -lrt  -O3
rm -f "filelist.txt"
cp -f ../bin/linux/libsnapmb.so /usr/lib
dave@dave-virtual-machine:~/SnapMB/build/linux$
```

`libsnapmb.so` by default will be copied into `/usr/lib`, to change this path you can set the `LibInstall` variable,
e.g.

```
sudo make install LibInstall=<MyPath>
```

Some 64 bit distributions (CentOS and Red Hat) need `libsnapmb.so` into `/usr/lib64` instead of `/usr/lib` So you should use

```
LibInstall=/usr/lib64
```

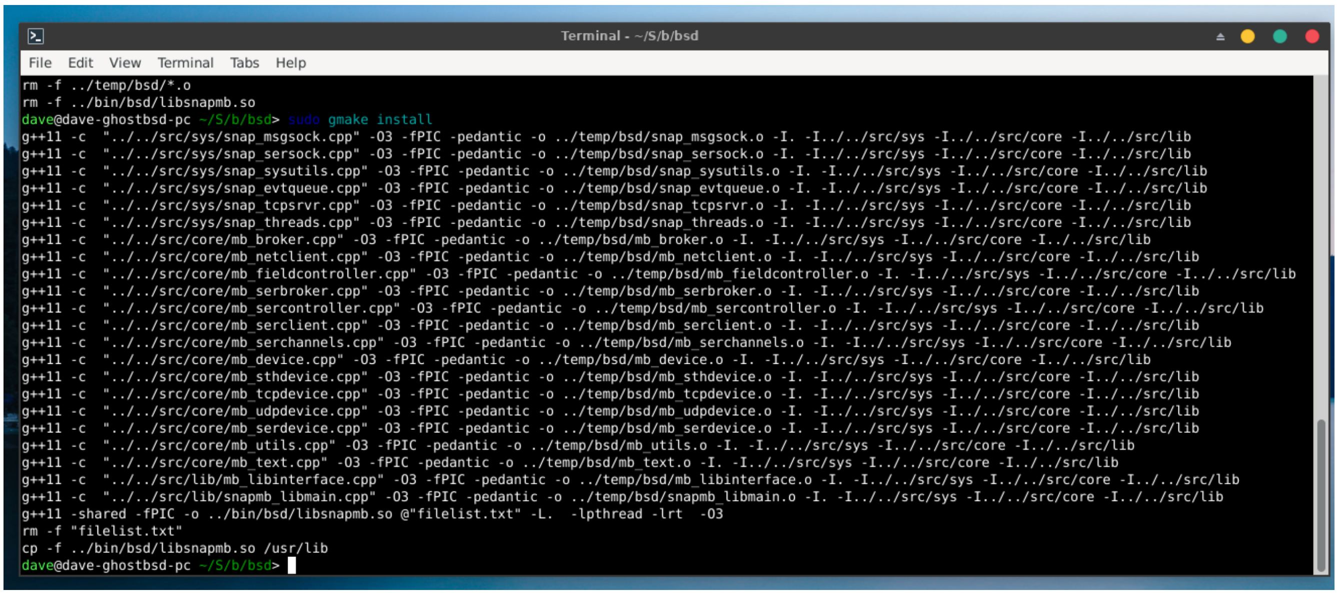
FreeBSD

Go to
`/build/bsd`
and run

```
sudo gmake install
```

in FreeBSD gcc is a port, so the compiler present will be `g++[version index]`, by default (I used FreeBSD 13.1) the variable Compiler is set to g++11, if you have a different version, you can choose one of these options:

1. Change the variable `Compiler` into the makefile
2. Run `sudo gmake install Compiler=g++[YourVersion]`
3. Create a symbolic link into /bin `g++` --> `g++[YourVersion]`



```
rm -f ../temp/bsd/*.o
rm -f ../bin/bsd/libsnapmb.so
dave@dave-ghostbsd-pc ~/S/b/bsd> sudo gmake install
g++11 -c  "../../src/sys/snap_msgsock.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_msgsock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/sys/snap_sersock.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_sersock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/sys/snap_sysutils.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_sysutils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/sys/snap_evtqueue.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_evtqueue.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/sys/snap_tcpsrvr.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_tcpsrvr.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/sys/snap_threads.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snap_threads.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_broker.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_broker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_netclient.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_netclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_fieldcontroller.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_fieldcontroller.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_serbroker.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_serbroker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_sercontroller.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_sercontroller.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_serclient.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_serclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_serchannels.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_serchannels.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_device.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_device.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_sthdevice.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_sthdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_tcpdevice.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_tcpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_udpdevice.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_udpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_serdevice.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_serdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_utils.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_utils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/core/mb_text.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_text.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/lib/mb_libinterface.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/mb_libinterface.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -c  "../../src/lib/snapmb_libmain.cpp" -O3 -fPIC -pedantic -o ../temp/bsd/snapmb_libmain.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++11 -shared -fPIC -o ../bin/bsd/libsnapmb.so @"filelist.txt" -L.  -lpthread -lrt  -O3
rm -f "filelist.txt"
cp -f ../bin/bsd/libsnapmb.so /usr/lib
dave@dave-ghostbsd-pc ~/S/b/bsd>
```

---

macOS

Open a terminal and type

```
g++
```

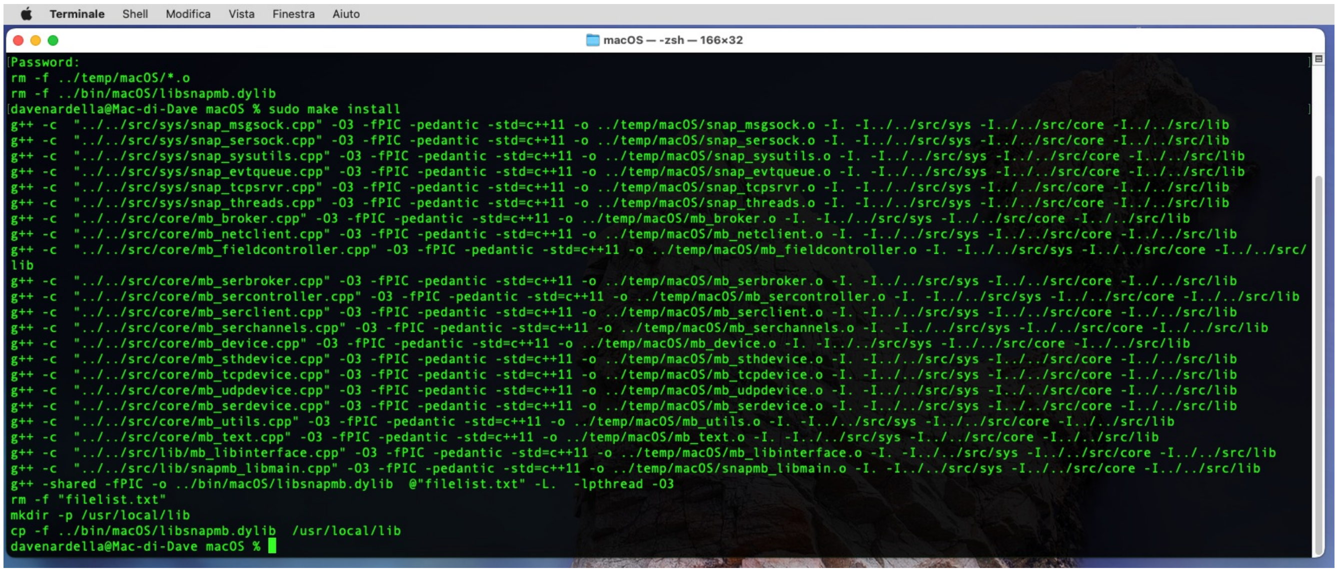
if the compiler is not installed will appear a window asking for install it, confirm.

Then, go to into

`/build/macOS`

and run

```
sudo make install
```



```
Password:
rm -f ../temp/macOS/*.o
rm -f ../bin/macOS/libsnapmb.dylib
davenardella@Mac-di-Dave macOS % sudo make install
g++ -c  "../../src/sys/snap_msgsock.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_msgsock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_sersock.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_sersock.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_sysutils.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_sysutils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_evtqueue.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_evtqueue.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_tcpsrvr.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_tcpsrvr.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/sys/snap_threads.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snap_threads.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_broker.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_broker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_netclient.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_netclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_fieldcontroller.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_fieldcontroller.o -I. -I../../src/sys -I../../src/core -I../../src/
lib
g++ -c  "../../src/core/mb_serbroker.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_serbroker.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_sercontroller.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_sercontroller.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serclient.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_serclient.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serchannels.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_serchannels.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_device.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_device.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_sthdevice.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_sthdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_tcpdevice.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_tcpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_udpdevice.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_udpdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_serdevice.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_serdevice.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_utils.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_utils.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/core/mb_text.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_text.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/lib/mb_libinterface.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/mb_libinterface.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -c  "../../src/lib/snapmb_libmain.cpp" -O3 -fPIC -pedantic -std=c++11 -o ../temp/macOS/snapmb_libmain.o -I. -I../../src/sys -I../../src/core -I../../src/lib
g++ -shared -fPIC -o ../bin/macOS/libsnapmb.dylib  @"filelist.txt" -L.  -lpthread -O3
rm -f "filelist.txt"
mkdir -p /usr/local/lib
cp -f ../bin/macOS/libsnapmb.dylib  /usr/local/lib
davenardella@Mac-di-Dave macOS % 
```

By default will be generated `libsnapmb.dylib` unless you change the variable `LibExt` e.g.:

```
sudo make install LibExt=so
```

Stress tests

When working on industrial plants, the equipment often remains on and running for long periods, so it is essential that the software involved is reliable.

It is **important** to use test methods that go beyond the usual work/fail method, which is only useful in debugging.

Two types of stress-tests were performed on SnapModbus.

- Concurrency
- Error Memory Check

> Note
> In the distribution there are also the sources of the stress tests, so, if you want, you can reproduce or modify the tests according to your needs.

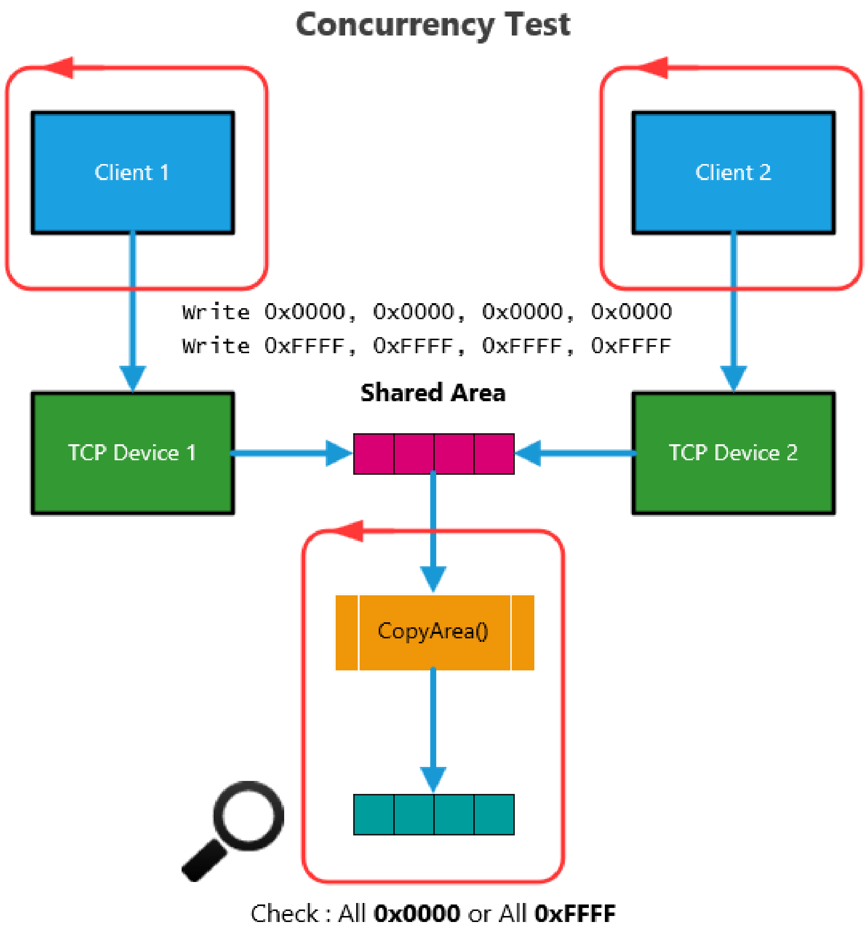
Concurrency test

With this test we verify the lock/unlock mechanisms of the memory areas (see here).

SnapModbus is very flexible, and allows heavy use of multithreading, allowing multiple objects to work simultaneously.

Therefore, it is important that, even in harsh conditions, it never happens that the data is corrupted, or that it is partially read, due to simultaneous accesses from multiple objects.

This is the principle scheme used.


Concurrency Test
Client 1
Client 2
Write 0x0000, 0x0000, 0x0000, 0x0000
Write 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF
Shared Area
TCP Device 1
TCP Device 2
CopyArea()
Check : All **0x0000** or All **0xFFFF**

An area of four registers is shared between two Devices working simultaneously on two different ports and our application.

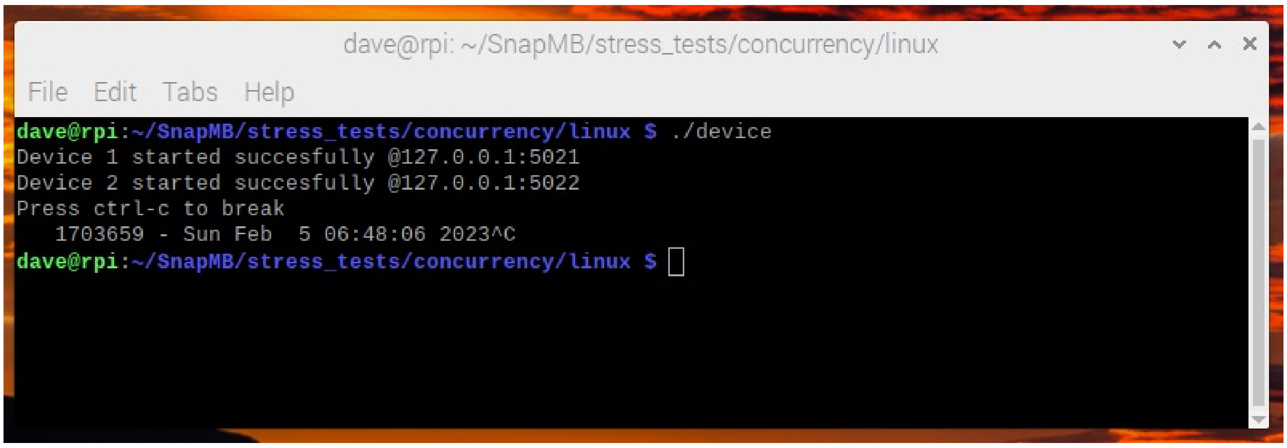
Two Clients running in two independent applications are connected to the two Devices and both write alternately in the registers two patterns:

- 0x0000, 0x0000, 0x0000, 0x0000
- 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF

The application continuously reads the four registers, using the safe function CopyArea(), and checks that they always contain **a single pattern** ie all four are 0x0000 or 0xFFFF.

If the lock/unlock management doesn't work, the data occasionally gets partially overwritten, because the writing of the values is not atomic and can be interrupted by another thread.

This was the result.



```
dave@rpi:~/SnapMB/stress_tests/concurrency/linux $ ./device
Device 1 started succesfully @127.0.0.1:5021
Device 2 started succesfully @127.0.0.1:5022
Press ctrl-c to break
   1703659 - Sun Feb  5 06:48:06 2023^C
dave@rpi:~/SnapMB/stress_tests/concurrency/linux $
```

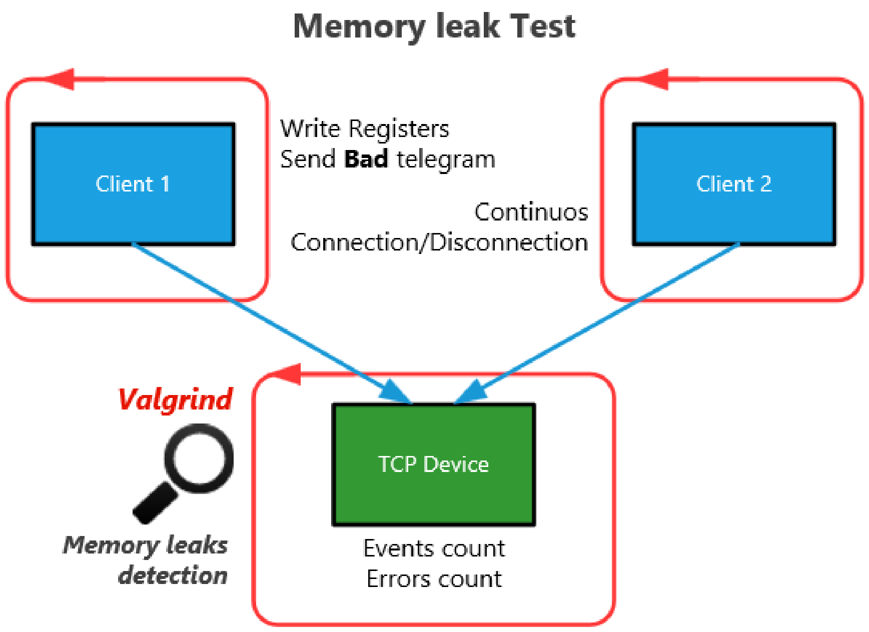
1.703.659 checks were made without errors.

Error memory check

With this test we verify that there are no memory errors and that our library does not have memory leaks or that it does not "eat" memory in the long run.

For this test I used Valgrind, a program which allows both to intercept memory errors and to keep track of block allocations/de-allocations, verifying that they are consistent.

This is the principle scheme used.



There are two clients working the two different threads connected to our Device at the same time.

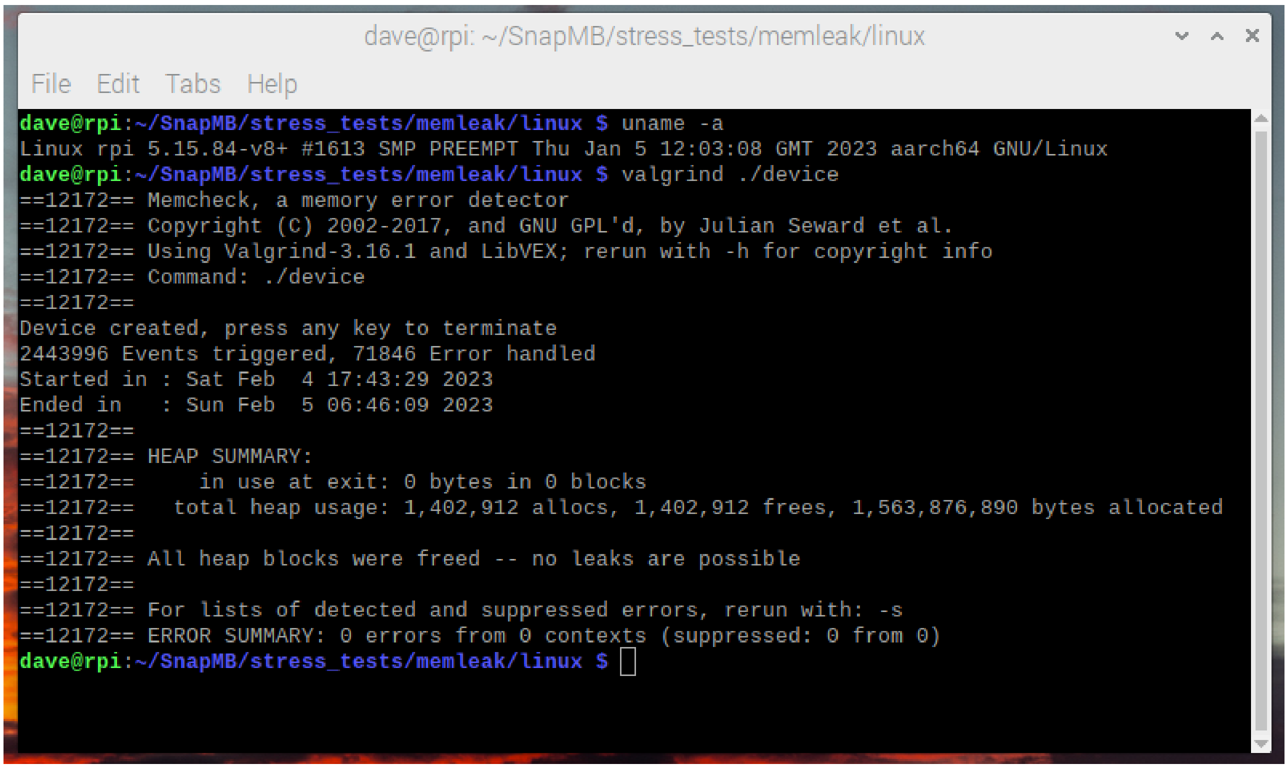
The first client writes to registers continuously, and occasionally sends bad telegrams. The second Client connects and disconnects continuously.

With the first client we test the ability to handle critical situations and, above all, that these do not cause errors in the memory.

With the second client we test that the continuous connections/disconnections, which in the Device cause the creation/destruction of many socket-threads, are correctly managed without memory leaks.

Our program, which runs inside Valgrind, counts the total events and error handling encountered due to bad telegrams received from Client 1.

This was the result.



```
dave@rpi:~/SnapMB/stress_tests/memleak/linux $ uname -a
Linux rpi 5.15.84-v8+ #1613 SMP PREEMPT Thu Jan 5 12:03:08 GMT 2023 aarch64 GNU/Linux
dave@rpi:~/SnapMB/stress_tests/memleak/linux $ valgrind ./device
==12172== Memcheck, a memory error detector
==12172== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12172== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==12172== Command: ./device
==12172==
Device created, press any key to terminate
2443996 Events triggered, 71846 Error handled
Started in : Sat Feb  4 17:43:29 2023
Ended in   : Sun Feb  5 06:46:09 2023
==12172==
==12172== HEAP SUMMARY:
==12172==     in use at exit: 0 bytes in 0 blocks
==12172==   total heap usage: 1,402,912 allocs, 1,402,912 frees, 1,563,876,890 bytes allocated
==12172==
==12172== All heap blocks were freed -- no leaks are possible
==12172==
==12172== For lists of detected and suppressed errors, rerun with: -s
==12172== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dave@rpi:~/SnapMB/stress_tests/memleak/linux $ 
```

- **2.443.996** Events detected
- **71.848** Errors succesfully handled
- **1.402.912** Memory blocks correctly allocaded and deallocated
- **1.536.876.890** byte of memory managed during the run
- **0** memory errors detected by Valgrind

References

1. Brief but complete description of the protocol in its various forms (**Wikipedia**)
2. Modbus **organization**
3. Official **specifications**

License

SnapModbus is distributed as a binary shared library with full source code under GNU Library or Lesser General Public License version 3.0 (LGPLv3)

Basically this means that you can distribute your commercial software linked with SnapModbus without the requirement to distribute the source code of your application and without the requirement that your application be itself distributed under LGPL.

A small mention to the project or the author is however appreciated if you include it in your applications.

SnapModbus examples, including wrappers and demos, are not covered by any license, thus they are **completely free**.

Disclaimer of Warranty

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.