Snap7

Reference manual

Davide Nardella

Rev.3 – January 2, 2014

Summary

Overview

Snap7 is an open source multi-platform Ethernet communication suite for interfacing natively with Siemens **S7 PLCs**. The new CPUs 1200/1500 and SINAMICS Drives are also partially supported.

Although it has been designed to overcome the limitations of OPC servers when transferring large amounts of high speed data in industrial facilities, it scales well down to small Linux based arm boards such as **Raspberry PI**, **BeagleBone Black**, **pcDuino, CubieBoard** and **UDOO.**

Three specialized components, **Client**, **Server** and **Partner**, allow you to definitively integrate your PC based systems into a PLC automation chain.

Main features

- Native multi-architecture design (32/64 bit).
- Platform independent, currently are supported Windows (from NT 4.0 up to Windows 8), Linux, BSD, Oracle Solaris 11 and Mac OSX.
- Fully scalable, starting from blade servers down to Raspberry PI board.
- No dependence on any third-party libraries, no installation needed, zero configuration.
- Three different native thread models for performance optimization: Win32 threads/ Posix threads / Solaris 11 threads.
- Two data transfer models: classic synchronous and asynchronous.
- Two data flow models: polling and unsolicited (PLC transfers data when it wants to).

Additional benefits

- Very easy to use, a full working server example is not bigger than the "Hello world".
- Hi level object oriented wrappers are provided, currently C/C++, .NET/Mono, Pascal, LabVIEW, Phyton with many source code examples.
- Multi-platform rich demos are provided.
- Many projects/makefiles are ready to run to easily rebuild Snap7 in any platform without the need of be a C++ guru.

Licensing

Snap7 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 Snap7 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 is however appreciated if you include it in your applications.

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.

IF ANYONE BELIEVES THAT, WITH SNAP7 PROJECT HAVE BEEN VIOLATED SOME COPYRIGHTS, PLEASE EMAIL US, AND ALL THE NECESSARY CHANGES WILL BE MADE.

Acnowledgments

A special thanks to Thomas W, without his S7comm wireshark plugin, this project would never have been born.

I also want to thank Thomas Hergenhahn for his libnodave.
Snap7 is not derived from it, but I happily used this library for a long time before writing my own library.

Thanks to Stephan Preeker and Gijs Molenaar for their Python wrapper

About this manual

This manual describes the package Snap7, what it consists of and how to use it.

It's written in **Internet English**, a kind of English with many syntax errors and a very questionable style.

The hope is that the message, though with much "syntax noise", it is understandable to all.

And, if the Internet English was not enough, I'm neither a writer nor a publishing expert, so, please, do not expect to read an award winner book.

You are warmly encouraged to send comments/corrections.

Rarely I read a manual from the beginning to the end, and I think that many other people do the same, so, many key concepts (such as PDU) are repeated throughout the entire manual to allow a "spot" consultation without the loss of information.

Convention

- Every concept or code snippet applies both for 32 and 64 bit architecture.
- Unless otherwise specified, **Unix** stands for Linux or BSD or Solaris.
- Unless otherwise specified, every concept or code snippet applies both for Unix and Windows regardless of the source code (C/C++/C#/Pascal).
- There is no special chapter about small ARM Linux boards (like Raspberry) because Snap7 offers the same functionality for these boards.

Snap7 Compatibility

Through the entire manual, you will find detailed info about the software implementation and about the hardware compatibility.

This is an only brief list to quickly know if Snap7 meets your working environment. As you can see, it was successfully tested into **54 OS/Distributions.**

As general rule any 16 bit OS/Compiler is definitely **not supported**.

OS

Microsoft Windows (x86-amd64)

	32	64
Windows NT Workstation 4.0 SP6	O	
Windows 2000 Professional	O	-
Windows 2003 Small Business Server	O	-
Windows 2003 Server R2	O	-
Windows 2008 Small Business Server	O	-
Windows 2008 Server RC2	O	O
Windows XP Professional SP3	O	O
Windows Vista	O	O
Windows 7 Home Premium	O	O
Windows 7 Professional	O	O
Windows 7 Ultimate	O	O
Windows 8 Professional	O	O
Windows 95	X	
Windows Me	X	

GNU-Linux (i386/i686-amd64)

	32	64
CentOS 6.4	O	O
Debian 6.0.6	O	-
Debian 7.0.0	O	O
Fedora 18	O	-
Fedora 19	O	-
Knoppix 7	O	-
LinuxMint 14	O	O
LinuxMint 15	O	O
OpenSuse 12.3	O	-
Red Hat 4.4.7-3	O	O
Semplice 4.0	O	O
Ubuntu 12.10	O	O
Ubuntu 13.04	O	O
Ubuntu 13.10	O	O
VectorLinux 7.0	O	-

GNU-Linux (arm v6/v7 boards)

	32	64
Raspberry PI - Raspbian “wheezy” (ARMHF)	O	
BeagleBone Black – Angstrom 2013.6 (ARMHF)	O	
pcDuino - Ubuntu 12.04 (ARMHF)	O	
CubieBoard 2 – Debian “wheezy” (ARMHF)	O	
UDOO – Ubuntu 12.04 LTS (ARMHF)	O	

BSD (i386-amd64)

	32	64
FreeBSD 9.1	O	O

OSX (i386-x86_64)

	32	64
Apple OSX 10.9.1 Mavericks	O	O

Oracle Solaris (i386-amd64)

	32	64
Solaris 11	O	-
OpenIndiana 151a7 (binary compatible with Solaris 11)	O	-

O	Built and tested
-	Compatible but not tested
X	Does not work
	Does not exists

missing OS releases / distributions / Platforms are to consider untested.

Wrappers

(Source code interface files and examples – see Snap7 Library API)

Pascal (snap7.pas)

Borland (or Inprise/CodeGear/Embarcadero) - Windows

	32	64
Delphi 2	O	
Delphi 3	O	
Delphi 4	O	
Delphi 5	O	
Delphi 6	O	
Delphi 7	O	
Delphi 8	O	
Delphi 2005	O	
Delphi 2006 (BDS 2006 / TurboDelphi)	O	
Delphi 2007	O	
Delphi 2008	O	
Delphi 2008 .NET	-	
Delphi 2009	O	
Delphi 2010	O	
Delphi XE	O	
Delphi XE2	O	O
Delphi XE3	O	O
Delphi XE4	O	O
Delphi XE5	O	O

missing releases are to consider untested

Borland - Linux

	32	64
Kylix (1.0/1.5)	-	

FreePascal - with Lazarus (when available)

	Windows	Linux	BSD	Sol 11	Linux Arm
FPC 2.4.0	32	32	-	-	-
FPC 2.6.0	32/64	32/64	32/64	-	32
FPC 2.6.2	32/64	32/64	32/64	-	-

CLR (snap7.net.cs)

Snap7 interface namespace is written in C#, the resulting compiled assembly **snap7.net.dll** can be used by all .net languages.

C# compiler	Windows	Linux	BSD	Sol 11	Linux Arm
Visual Studio 2008 (1)(2)	**32/64**				
Visual Studio 2010 (1)(2)	**32/64**				
Visual Studio 2012 (1)(3)	**32/64**				
Mono 2.10	**32**	**32/64**	**-**	**-**	

missing releases are to consider untested

(1) snap7.net.cs was compiled with the C# compiler supplied with Visual Studio, but the same compiler is part of the related **.NET SDK**
(2) Using .NET Framework 3.5 SP1
(3) Using .NET Framework 4.5

C++ (snap7.cpp/snap7.h)

C++ compiler	Windows	Linux	BSD	Sol 11	Linux Arm	OSX
Visual Studio 2008 (1)	**32/64**					
Visual Studio 2010 (1)	**32/64**					
Visual Studio 2012	**32/64**					
MinGW 32 4.7.2	**32**					
MinGW 64 4.7.1 (2)	**64**					
C++ Builder XE2 (3)	**32**					
C++ Builder XE3 (3)	**32**					
C++ Builder XE4 (3)	**32/64**					
C++ Builder XE5 (3)	**32/64**					
GNU g++ 4.6		**32/64**	**32/64**	**-**	**32**	
GNU g++ 4.7		**32/64**	**32/64**	**32**	**32**	
Solaris studio Compiler				**32** (4)		
Apple XCode 5.0						**64**

missing releases are to consider untested

C (snap7.h)

C compiler	Windows	Linux	BSD	Sol 11	Linux Arm	OSX
Visual Studio 2008 (1)	**32/64**					
Visual Studio 2010 (1)	**32/64**					
Visual Studio 2012	**32/64**					
MinGW 32 4.7.2	**32**					
MinGW 64 4.7.1 (2)	**64**					
C++ Builder XE2 (3)	**32**					
C++ Builder XE3 (3)	**32**					
Pelles C	**X**					
LCC-Win32	**X**					
LCC-Win64	**X**					
GNU GCC 4.6		**32/64**	**32/64**	**-**	**32**	
GNU GCC 4.7		**32/64**	**32/64**	**32**	**32**	
Solaris studio Compiler				**32** (4)		
Apple XCode 5.0						**64**

missing releases are to consider untested

(1) Express release needs **Windows Software Development Kit (SDK)** to compile 64 bit applications.
(2) This compiler (TDM 64-3) is able to produce also 32 bit binaries.
(3) snap7.lib must be converted with **coff2omf.exe** in order to be used with this compiler.

(4) Snap7 Library cannot be built with this compiler (see Rebuild Snap 7 chapter), but having a working **libsnap7.so** compiled with the GNU toolchain, the wrapper snap7.cpp works well in user programs with Oracle Solaris Studio Compiler.

LabVIEW (lv_snap7.dll / snap7.lvlib)

C compiler	Windows	Linux	BSD	Sol 11
LabVIEW 2010	**32**	**-**	**-**	**-**
LabVIEW 2013	**32/64**	**-**	**-**	**-**

missing releases are to consider untested

Python

Please refer to https://github.com/gijzelaerr/python-snap7

O	Tested - OK
32	32 Bit release
64	64 Bit release
-	Not tested
X	Does not work
	Does not exists

Siemens communications overview

Snap7, by design, only handles Ethernet S7 Protocol communications.

Why only Ethernet ?

Having said that we are not talking about the fieldbus, but we are focusing on PC-PLC communications, Ethernet has several advantages against Profibus/Mpi :

- It's faster, CP 1543-1 (for the newborn S71500) has a bandwidth of 1000 Mbps.
- It's more simply to troubleshoot, in 50% of cases a "ping" is good enough to solve your problems.
- It's cheaper, you don't need a special adapter to communicate (which, moreover, cannot not be used with any hardware and any virtual infrastructure as Snap7 does).
- If you use Snap7Server, many more non-Siemens panels/scada can be connected with your software.

Siemens PLCs, through their communication processors (CP) can communicate in Ethernet via two protocols:

Open TCP/IP and **S7 Protocol**.

The first is a standard implementation of the TCP/IP protocol, it's provided mainly to connect PLCs with non-Siemens hardware.

TCP/IP is a generic protocol, it only states how the packets must be transferred, and it doesn't know anything about their content.

Finally, TCP/IP is stream oriented (though Siemens FC/FB needs to packetize the data stream into blocks).

As said, it is a standard, so you don't need of Snap7 to use it, your preferred socket libraries are perfectly suitable.

S7 Protocol

S7 Protocol, is the backbone of the Siemens communications, its Ethernet implementation relies on ISO TCP (RFC1006) which, by design, is block oriented.

Each block is named **PDU** (Protocol Data Unit), its maximum length depends on the CP and is negotiated during the connection.

S7 Protocol is **Function oriented** or **Command oriented**, i.e. each transmission contains a command or a reply to it.
If the size of a command doesn't fit in a PDU, then it's split across more subsequent PDU.

Each command consists of
- A header.
- A set of parameters.
- A parameters data.
- A data block.

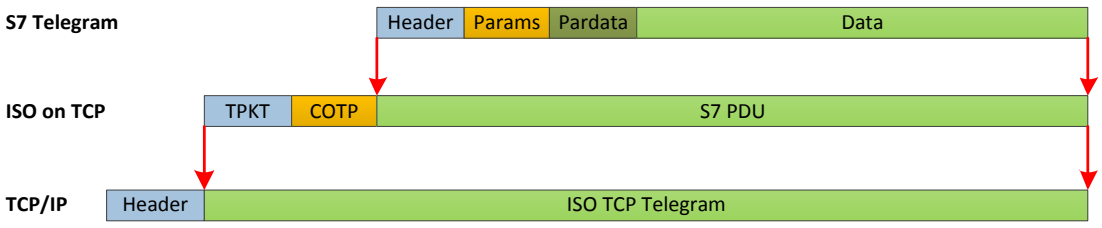
The first two elements are always present, the other are optional.

To understand:

***Write** this **data** into **DB 10** starting from the offset **4**.*
Is a command.

Write, DB, 10, 4 and data are the components of the command and are formatted in a message in accord to the protocol specifications.

S7 Protocol, ISO TCP and TCP/IP follow the well-known encapsulation rule : every telegram is the "payload" part of the underlying protocol.



Protocols Encapsulation

S7 Commands are divided into categories:

- **Data Read/Write**
- **Cyclic Data Read/Write**
- **Directory info**
- **System Info**
- **Blocks move**
- **PLC Control**
- **Date and Time**
- **Security**
- **Programming**

For a detailed description of their behavior, look at the functions list of Snap7Client that are arranged in the same way (Except for Cyclic Data I/O and Programming which are not implemented).

Siemens provides a lot of FB/FC (PLC side), **Simatic NET** software (PC side) and a huge excellent documentation about their use, but no internal protocol specifications.

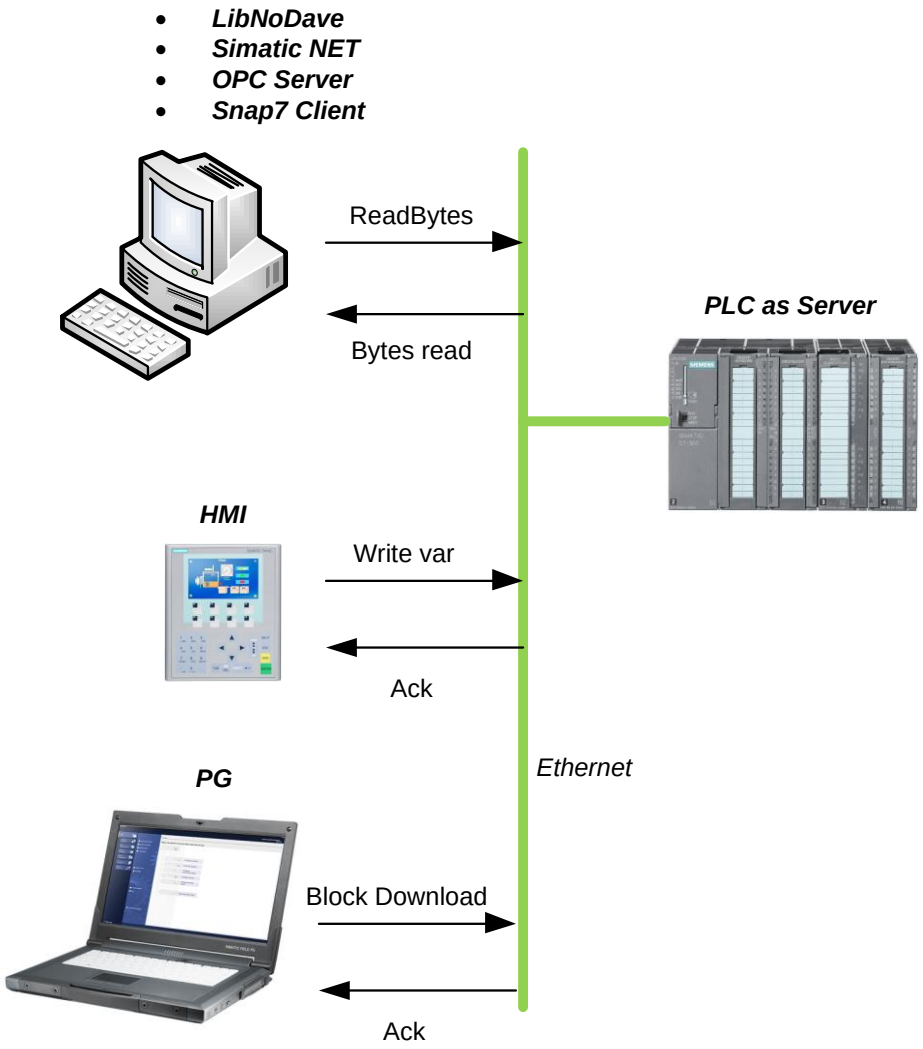
The Siemens theatre

In the Siemens communication theatre there are three actors:

1. **The Client**
2. **The Server**
3. **The Partner** (a.k.a. the peer in the classic computer dictionary).

And as in all good theater companies, they follow their script:

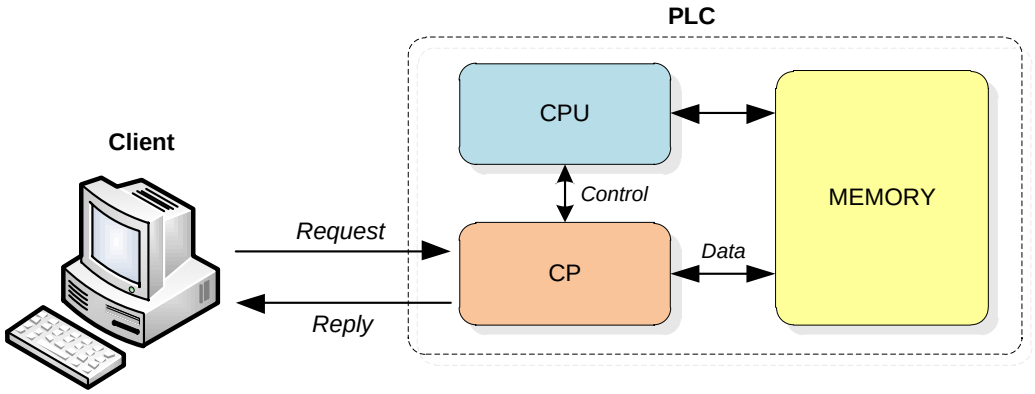
- The client can only query.
- The server can only reply.
- The partners can speak both on their own initiative.



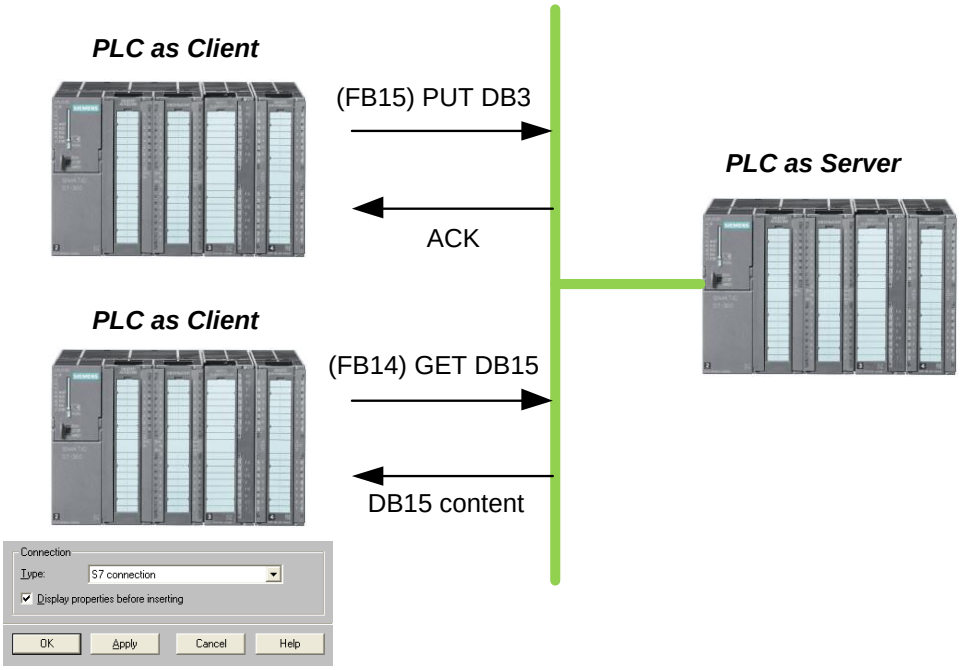
All three components on the left are **Clients**, they connect to the internal server of the Communication Processor (CP), and make an S7 Request.
The server replies with a S7 answer telegram.

No configuration is needed server side. The server service is automatically handled by the firmware of the CP.

The CP can be external such as CP343/CP443 or internal in 3XX-PN or 4XX-PN CPUs, they, however, work in the same way.



Also a PLC can work as Client, in this case the data read/write requests are made via FB14/FB15 (Get/Put), and a S7 connection, created with **NetPRO,** is needed.



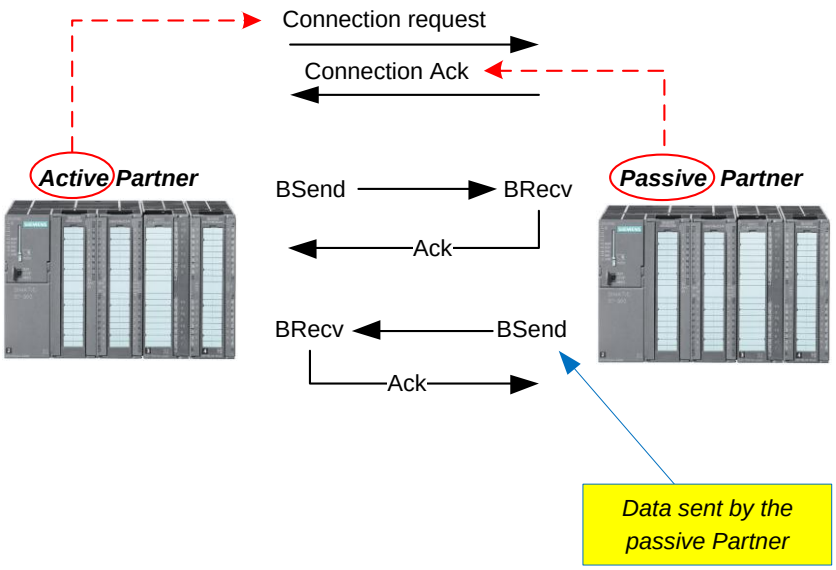
The **Partners** can exchange unsolicited data, i.e. once the connection is established, both can send data to the other partner.

This kind of communication often is named Client-Client by Siemens in their manuals.

The peer that requests the connection is named **Active Partner**, the peer that accepts the connection is named **Passive partner**.

The communication is performed via FB12/FB13 (S7300) or SFB12/SFB13 (S7400), their symbolic names are BSend/BRecv (Block Send / Block Recv).

An important remark is that : when PLC A calls BSend, BRecv must being call in PLC B in the same time, to complete the transaction.



For both partners an S7 Connection must be created with NetPro.
The Active partner must have the "Establish an active connection" option checked in the connection properties (You can find further details into the Snap7Partner description).

This kind of communication model is provided mainly to connect PLCs each other, Snap7Partner however, allows your software to act as active or passive partner in a PLCs network.

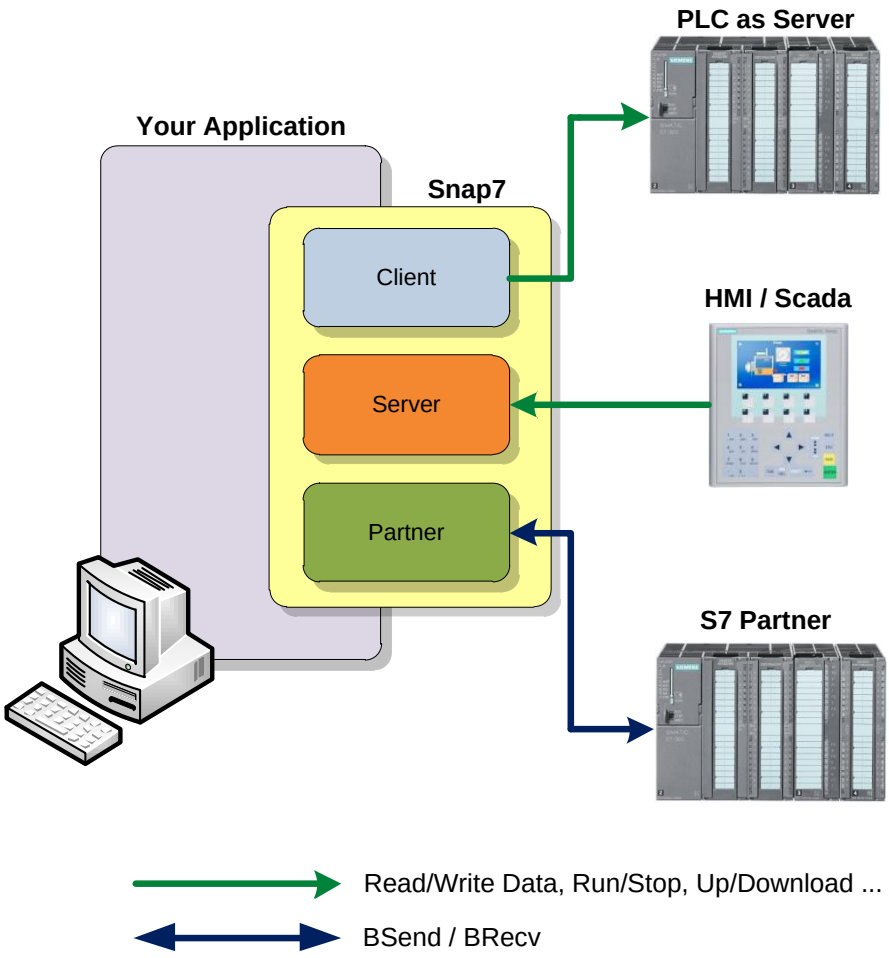
The Snap7 theatre

The purpose of Snap7 is to fully integrate your PC station into a PLC network, without anybody sees the difference.

To allow this, the Snap7 theater must be the same as the Siemens and contain the same actors.

Snap7 library contains three objects: the **Client**, the **Server** and the **Partner**.

- The three objects can be used simultaneously in the same application.
- Many objects of the same type can be instanced simultaneously.
- Many applications can use Snap7 simultaneously.



Snap7Client

A PLC client is the most well-known object, almost all PLC communication drivers on the market are clients.

Into S7 world, **LibNoDave**, **Prodave**, **SAPI-S7** (Simatic Net mn library) are clients. The same **OPC Server**, despite to its name, is a client against the PLC.

Finally, **Snap7Client** is a Client.

It fulfills almost completely the S7 protocol, you can read/write the whole PLC memory (In/Out/DB/Merkers/Timers/Counters), perform block operations (upload/download), control the PLC (Run/Stop/Compress..), meet the security level (Set Password/Clear Password) and almost all functions that Simatic Manager or Tia Portal allows.

You certainly have a clear idea about its use, its functions and their use are explained in detail into the **Client API** reference.

What I think is important to highlight, is its advanced characteristics.

Snap7 library is designed keeping in mind large industrial time-critical data transfers involving networks with dozen of PLCs.

To meet this, Snap7Client exposes three interesting features : **PDU independence**, **SmartConnect** and **Asynchronous data transfer**.

PDU independence

As said, every data packet exchanged with a PLC must fit in a PDU, whose size is fixed and varies from 240 up to 960 bytes.

All Snap7 functions completely hide this concept, the data that you can transfer in a single call depends only on the size of the available data.

If this data size exceeds the PDU size, the packet is automatically split across more subsequent transfers.

SmartConnect

When we try to connect to a generic server, basically two requirements must be met.

1. The hardware must be powered on.
2. A Server software listening for our connection must be running.

If the server is PC based, the first condition not always implies the second.
But for specialist hardware firmware-based such as the PLC, the things are different, few seconds after the power on all the services are running.

Said that, if we "can ping" a PLC we are almost sure that it can accept our connections.

The **SmartConnect** feature relies on this principle to avoid the TCP connection timeout when a PLC is off or the network cable is unwired.
Unlike the TCP connection timeout, The ping time is fixed and we can decide how much it should be.

When we call **`Cli_ConnectTo()`**, or when an active Snap7Partner needs to connect, first the PLC is "pinged", then, if the ping result was ok, the TCP connection is performed.

Snap7 uses two different ways to do this, depending on the platform:

Windows

The system library **iphlpapi.dll** is used, but it's is loaded dynamically because it's not officially supported by Microsoft (even if it is present in all platforms and now it's fully documented by MSDN).

If its load fails (very rare case), an ICMP socket is created to perform the ping. We use it as B-plan since we need administrative privileges to do this in Vista/Windows 7/Windows 8.

Unix (Linux/BSD/Solaris)

There is no system library that can help us, so the ICMP socket is created. Unluckily, to do this, our program needs root rights or the SUID flag set.

During the initialization, the library checks if the ping can be performed trying the above methods.

If all they fail, SmartConnect is disabled and all the clients (or Active partners) created will try to connect directly.

Now let's see how to take full advantage of this feature.

Let's suppose that we have a Client that cyclically exchanges data into a thread and we want a fast recovery in case of network problems or PLC power.

In the thread body we could write something like this:

C++

```
while (!TerminateCondition())
{
    if (Client->Connected())
    {
       PerformDataExchange();
       sleep(TimeRate); // exchange time interval
    }
    else
        if (Client->Connect()!=0)
            sleep(10); // small wait recovery time
}

//Supposing that TerminateCondition()is a bool function that
//returns true when the thread needs to be terminated.

//In Unix you have to use nanosleep() instead of sleep() or copy
//SysSleep() from snap_sysutils.cpp.
```

Pascal

```
while not TerminateCondition do
begin
    if Client.Connected then
    begin
       PerformDataExchange;
       Sleep(TimeRate); // exchange time interval
    end
    else
        if Client.Connect<>0 then
            Sleep(10); // small wait recovery time
end;

//Supposing that TerminateCondition()is a boolean function that
//returns true when the thread needs to be terminated.
```

In the examples are used the C++ and Pascal classes that you find into the wrappers.

Asynchronous data transfer

A **synchronous** function, is executed in the same thread of the caller, i.e. it exits only when its job is complete. Synchronous functions are often called *blocking functions* because they block the execution of the caller program.

An **asynchronous** function as opposite, consists of two parts, the first, executed in the same thread of the caller, which prepares the data (if any), triggers the second part and exits immediately.
The second one is executed in a separate thread and performs the body of the job requested, simultaneously to the execution of the caller program.

This function is also called *nonblocking.*

The choice of using one or the other model, essentially depends on two factors:

1. How much the parallel job is granular than the activity of the CPU.
2. How much, the job execution time, is greater than the overhead introduced by the synchronization.

A S7 protocol job consists of:

- Data preparation.
- Data transmission.
- Waiting for the response.
- Decoding of the reply telegram.

Each block transmitted is called PDU (protocol data unit) which is the greatest block that can be handled per transmission.

The "**max pdu size**" concept belongs to the IsoTCP protocol and it's negotiated during the S7 connection.

So, if our data size (plus headers size) is greater than the max pdu size, we need to split our packets and repeat the tasks of transmission and waiting.

"Waiting for the response" is the worst of them since it's the longest and the CPU is unused in the meantime.

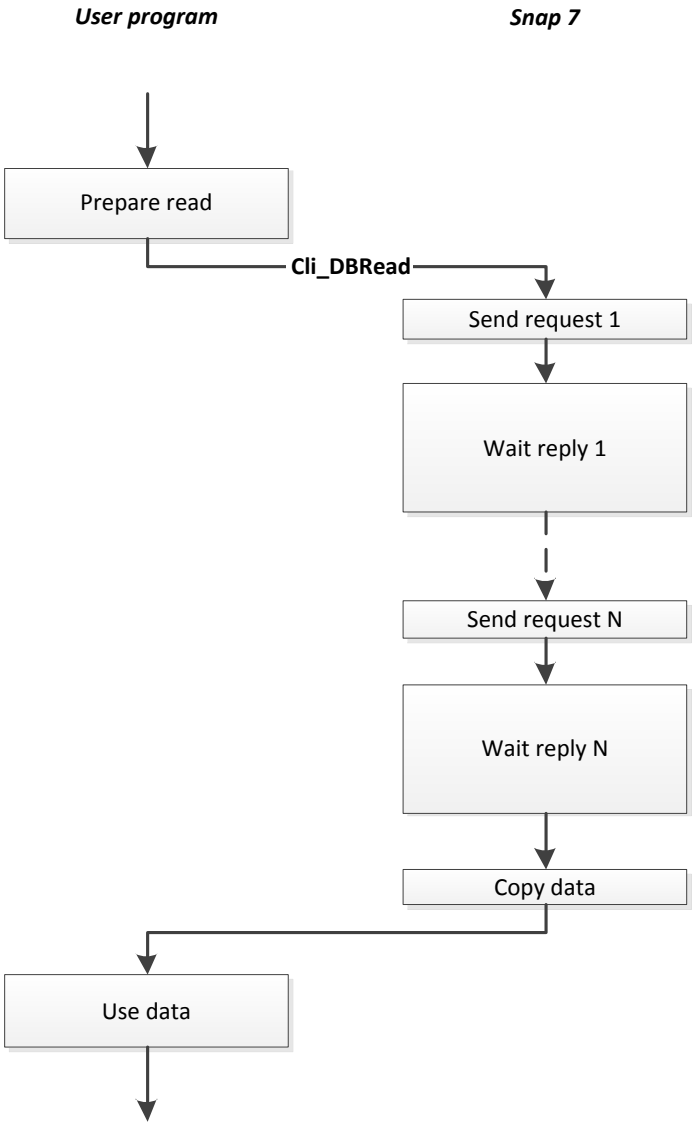
So, a S7 Job is definitely granular and could benefit from asynchronous execution.

"It could" because the advantage is zeroed by the synchronization overhead if the job consists of a single pdu.

The Snap7 Client supports both data flow models via two different set of functions that can be mixed in the same session:

Cli_<function name> and **Cli_As<function name>.**

The example in figure shows a call of Cli_DBRead that extends for more PDUs; during its execution the caller is blocked.



End of Job Completion

The asynchronous model in computer communications, however, has a great Achilles heel : **the completion**.

To understand:
The function is called, the job thread is triggered, the function exits and the caller work simultaneously to the thread.
At the end, we need to join the two execution flows, and to do this we need of a sort of synchronization.

An inappropriate completion model can completely nullify the advantage of asynchronous execution.

Basically there are three completion models:

- **Polling**
- **Idle wait**
- **Callback**

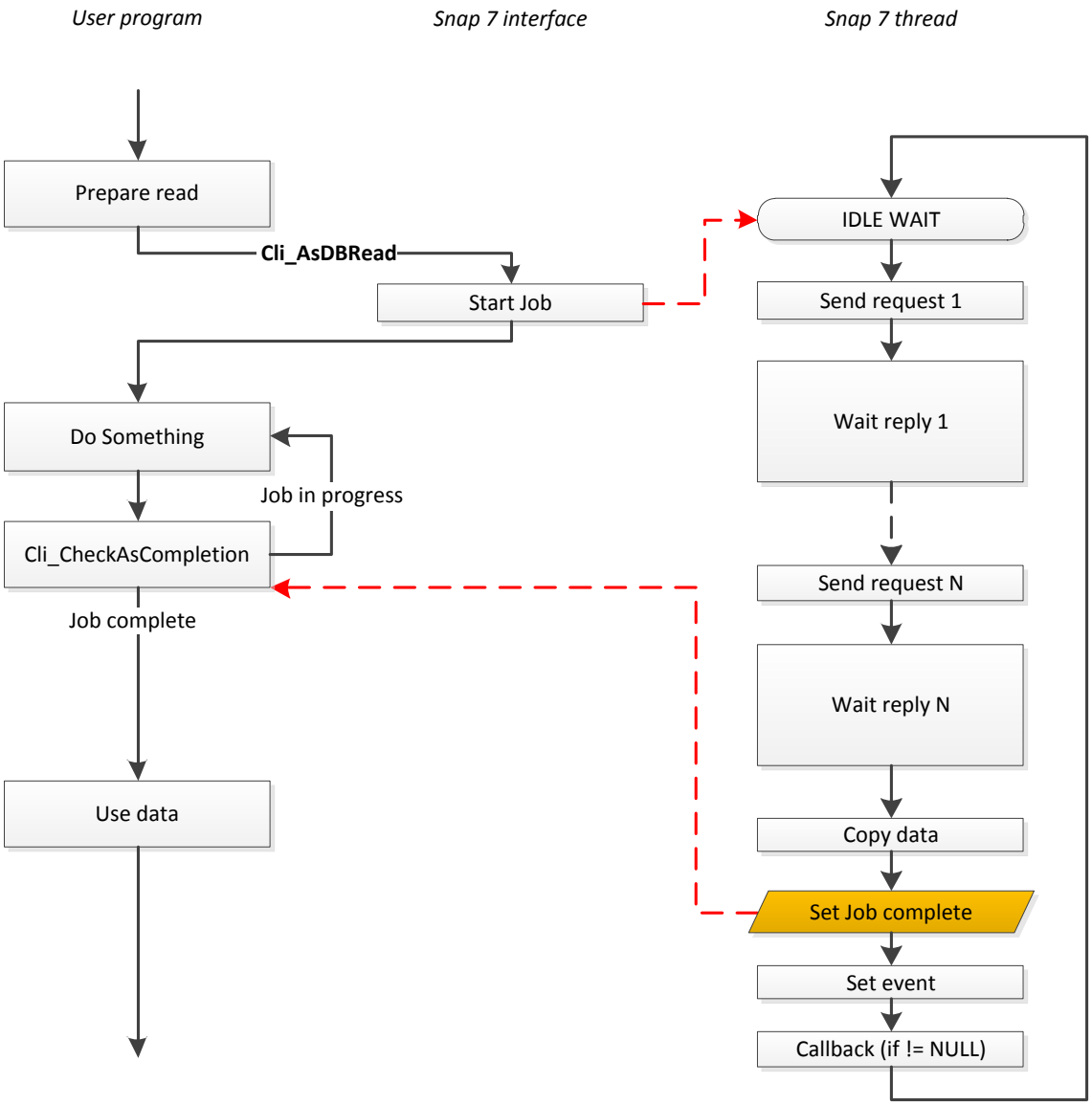
There is no better than the others, it depends on the context.

Snap7 Client supports all three models, or a combination of them, if wanted.

The **polling** is the simplest : after starting the process, we check the Client until the job is finished.

To do this we use the function **Cli_CheckAsCompletion()**; when called it quits immediately and returns the status of the job : finished or in progress.

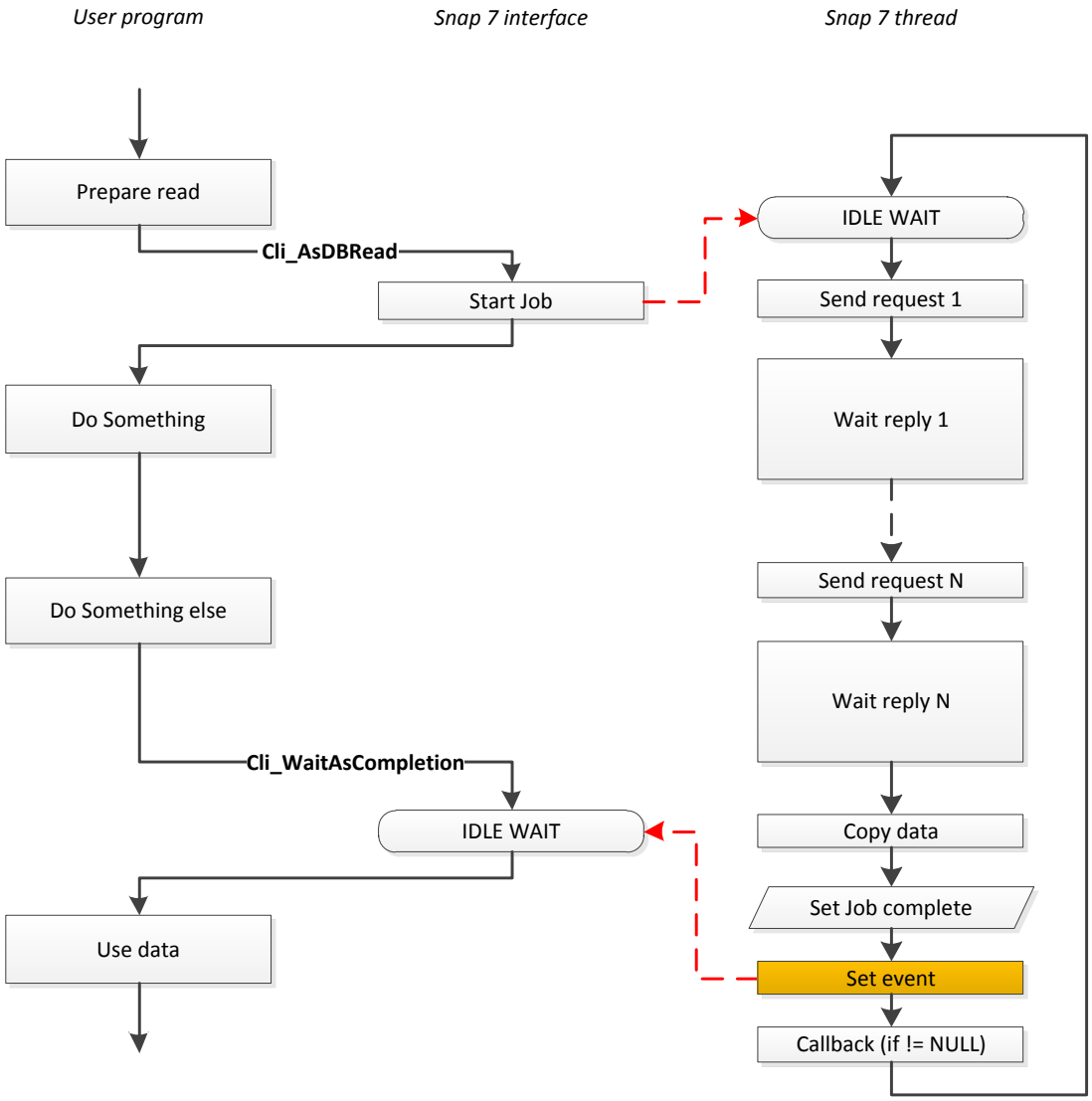
We can use it to avoid that our program becomes unresponsive during large data transfer.



The **idle wait completion** waits until the job is completed or a Timeout expired. During this time our program is blocked but the CPU is free to perform other tasks.

To accomplish this are used specific OS primitives (events, semaphores..).

The function delegated to this is **Cli_WaitAsCompletion()**

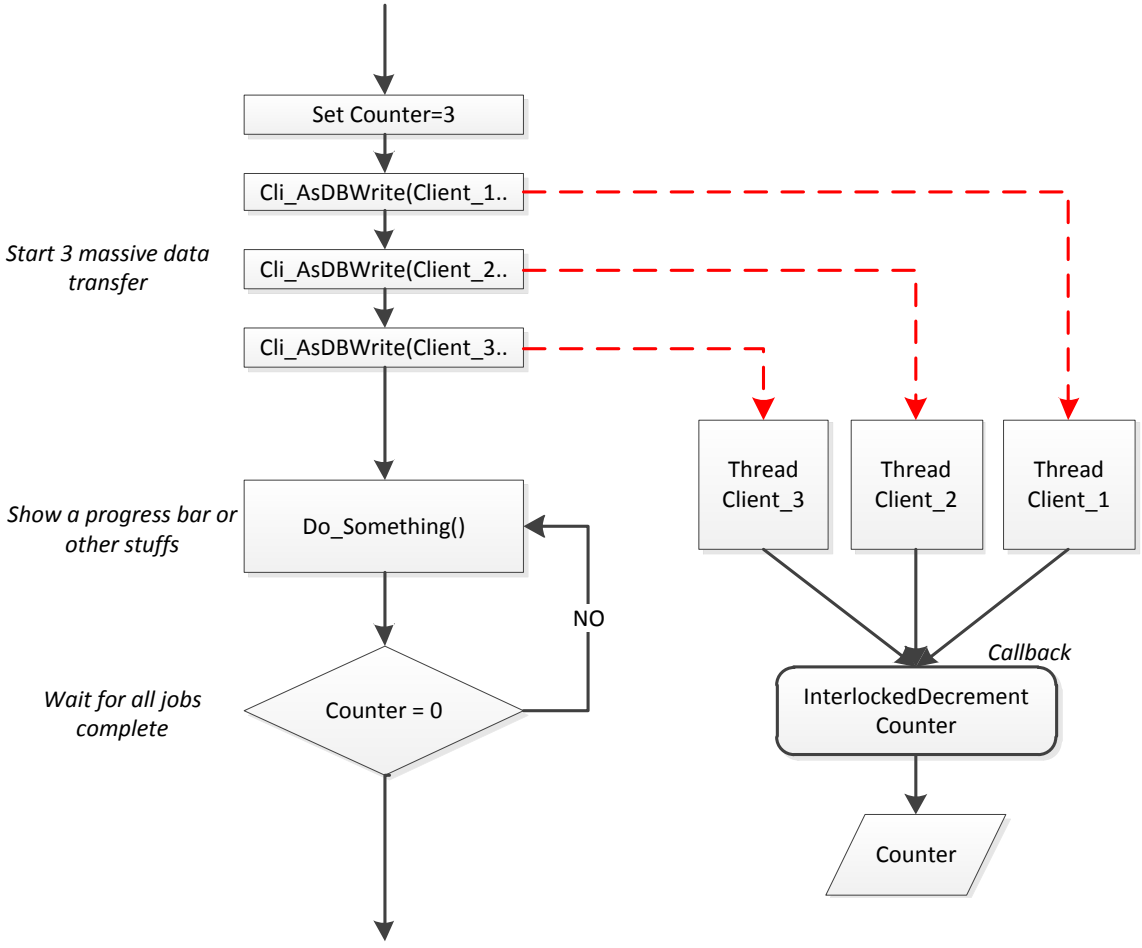


The **Callback** method is the more complex one:
When the job is terminated a user function (the so named callback) is invoked.

To use it, we must instruct the client about the callback (using **Cli_SetAsCallback()**) and we need to write the synchronization code inside it.

If it is used properly, this method can solve problems that cannot be solved with other libraries (as we will see speaking about the Snap7Partner).

In the picture we have several PLC and we need to make a "type changeover" in a production line.
And to do this we need to transfer a new large set of working parameters to each PLC.



Though the callback resides into the user's program, it's called into the Client thread, so be aware that calling another Client function inside the callback could lead to stack overflow.

Note

InterlockedDecrement is a synchronization primitive present in Windows/Unix that performs an atomic decrement by one on a variable.

Target Compatibility

As said, the S7 Protocol, is the backbone of the Siemens communications.

Many hardware components equipped with an Ethernet port can communicate via the S7 protocol, obviously not the whole protocol is fulfilled by them (would seem strange to download an FC into a CP343).

S7 300/400/WinAC CPU

They fully support the S7 Protocol.

S7 1200/1500 CPU

They use a modified S7 protocol with an extended telegram, 1500 series has advanced security functions (such as encrypted telegrams), however they can work in 300/400 compatibility mode and some functions can be executed, see also S7 1200/1500 Notes.

S7 200 / LOGO OBA7

These PLC have a different approach. See S7200 and LOGO chapters for a detailed description about their use with Snap7.

SINAMICS Drives

It' possible to communicate with the internal CPU, for some models (G120 for example) is also possible to change drive parameters.
A way to know what is possible to do with a given model, is to search what is possible to do with an HMI panel/Scada, since Snap7 can do the same things.

CP (Communication processor)

It's possible to communicate with them, and see their internal SDBs, although it's not such a useful thing, or you can use SZL information for debug purpose.

S7 Protocol partial compatibility list (See also LOGO / S7200)

	CPU						CP	DRIVE
	300	400	WinAC	Snap7S	1200	1500	343/443/IE	SINAMICS
DB Read/Write	O	O	O	O	O	O(3)	-	O
EB Read/Write	O	O	O	O	O	O	-	O
AB Read/Write	O	O	O	O	O	O	-	O
MK Read/Write	O	O	O	O	O	O	-	-
TM Read/Write	O	O	O	O	-	-	-	-
CT Read/Write	O	O	O	O	-	-	-	-
Read SZL	O	O	O	O	O	O	O	O
Multi Read/Write	O	O	O	O	O	O	-	O
Directory	O	O	O	O	-	-	O	(2)
Date and Time	O	O	O	O	-	-	-	O
Control Run/Stop	O	O	O	O	-	-	(1)	O
Security	O	O	O	O	-	-	-	-
Block Upload/Down/Delete	O	O	O	-	-	-	O	O

Snap7S = Snap7Server

(1) After the "Stop" command, the connection is lost, Stop/Run CPU sequence is needed.
(2) Tough DB are present and accessible, directory shows only SDBs.
(3) See S71500 Notes

S7 1200/1500 Notes

An external equipment can access to S71200/1500 CPU using the S7 "base" protocol, only working as an HMI, i.e. only basic data transfer are allowed.

All other PG operations (control/directory/etc..) must follow the extended protocol.

Particularly to access a DB in S71500 some additional setting plc-side are needed.

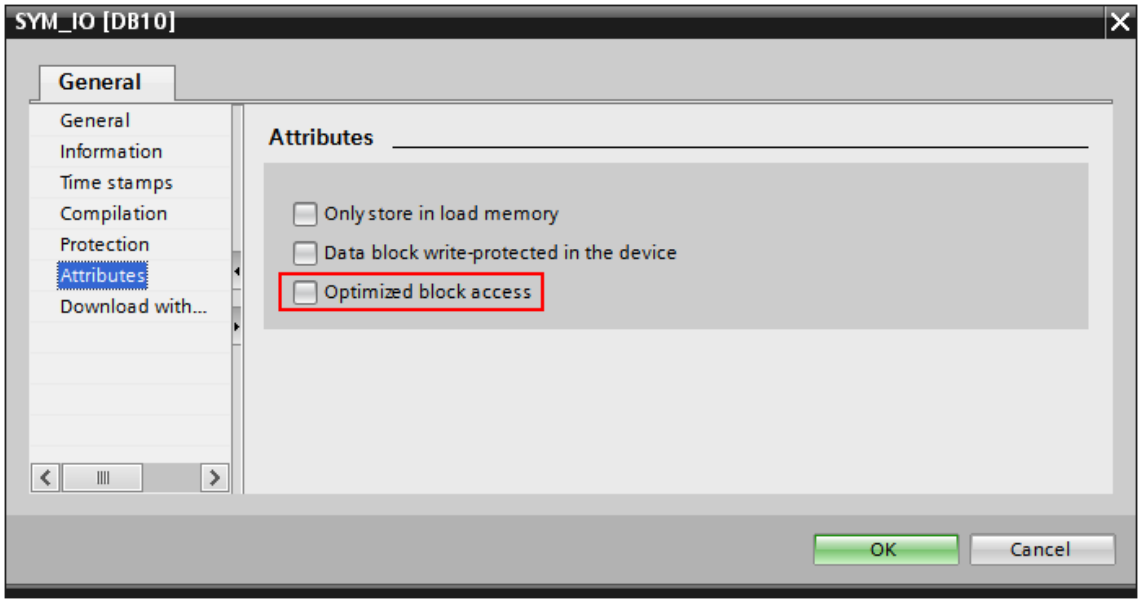
1. Only global DBs can be accessed.
2. The optimized block access must be turned off.
3. The access level must be "full" and the "connection mechanism" must allow GET/PUT.

Let's see these settings in TIA Portal V12

DB property

Select the DB in the left pane under "Program blocks" and press Alt-Enter (or in the contextual menu select "Properties…")

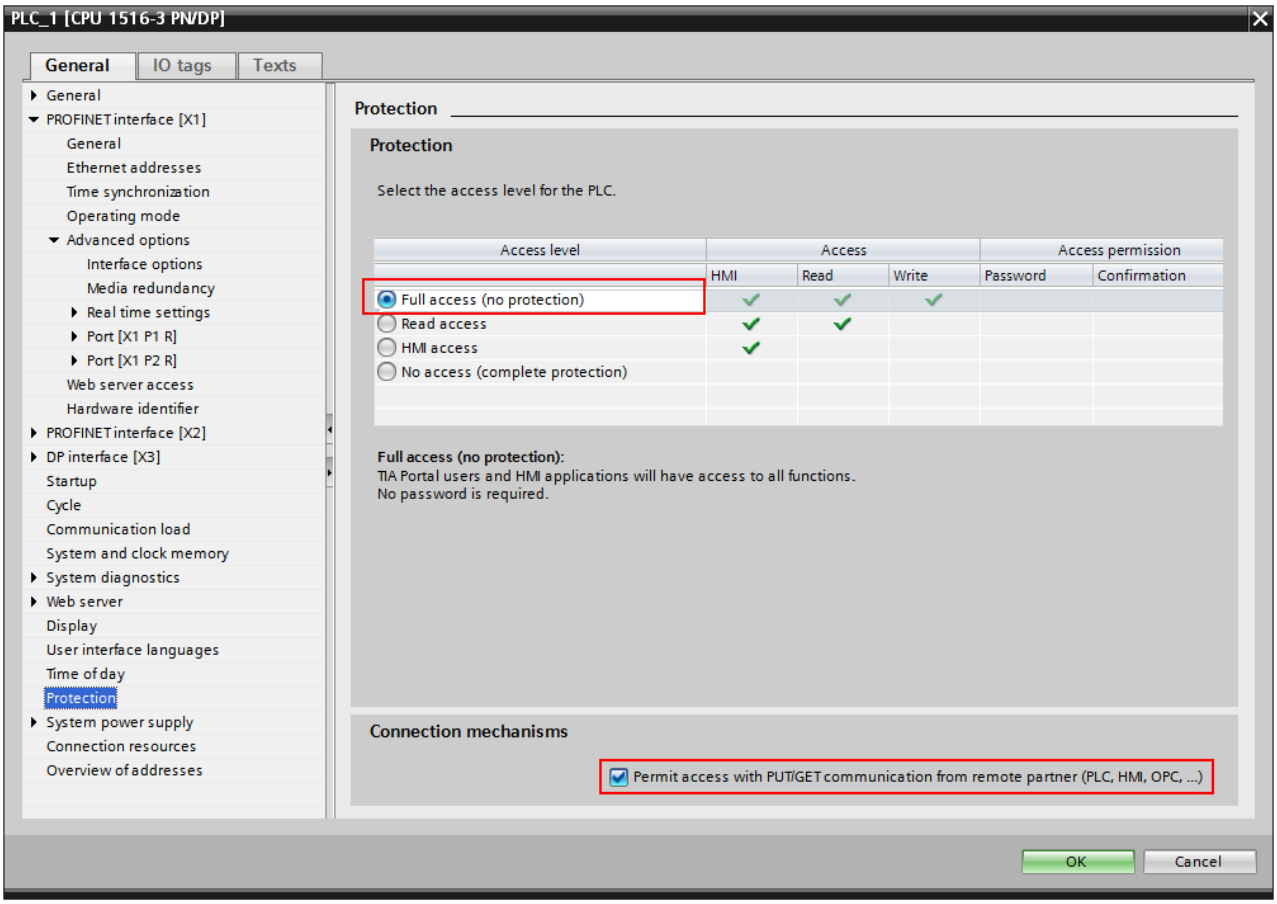
Uncheck Optimized block access, by default it's checked.



Protection

Select the CPU project in the left pane and press Alt-Enter (or in the contextual menu select "Properties…")

In the item Protection, select "Full access" and Check "Permit access with PUT/GET …." as in figure.



Snap7MicroClient

In the Snap7 project, *TSnap7MicroClient* is the ancestor of *TSnap7Client* class.

It's not exported, i.e. you cannot reference it from outside the library, and the only way to use it is to embed it in your C++ source code.

TSnap7MicroClient implements the body of all S7 Client jobs and the synchronous interface functions.

The exported *TSnap7Client*, only adds the remaining asynchronous functions but does not introduce any new S7 behavior.

Why we are speaking about an internal object ?

The micro client is thread independent and only relies on the sockets layer, i.e. you would embed it in your source code if:

- Your application will run in a micro-OS that has no threads layer.
- Your application will run in a realtime-OS (such as QNX) or in an OS that has not a standard threads layer (neither Windows nor posix). In this case you may create a native thread and use the micro client inside of it.

Micro client "extrapolation" is provided by design, there is a well-defined group of independent files to use.

See the chapter **Embedding Snap7MicroClient** for further information.

PLC connection

To connect a PLC, a client or an Active Partner must specificate three params : IP, Local TSAP, Remote TSAP.

The communication literature says:

"A Transport Services Access Point (TSAP) is an end-point for communication between the Transport layer (layer 4) and the Session layer in the OSI (Open Systems Interconnection) reference model. Each TSAP is an address that uniquely identifies a pecific instantiation of a service. TSAPs are created by concatenating the node's Network Service Access Point (NSAP) with a transport identifier, and sometimes a packet and/or protocol type."

We, field-mans, only need to know that the TSAP mechanism is used as a further addressing level in the S7 PLC network and contains informations about the resources involved as well.

In accord to the specifications, the TSAP is a generic indicator and could be also a string, in the S7 connection it's a 16 bit word.

Starting from the 1.1.0 release, Snap7 allows to specificate the TSAPs also in a Client connection, this to allow to connect with LOGO and S7200 that need particular TSAP values, just like a Snap7Partner.

To connect a PLC now there are 4 functions client-side, not to be used all in the same time of course, but grouped.

Let's see how do it dividing by two the PLC families:

S7 300/400/1200/1500/WinAC/Sinamics/Snap7Sever

Use **Cli_ConnectTo()** specifying **IP_Address, Rack, Slot** for the first connection, this functions set the internal parameters and connects to the PLC. if a TCP error occurred and a disconnection was needed, for reconnecting you can simply use **Cli_Connect()** which doesn't requires any parameters. Look at the reference of CLi_ConnectTo for a detailed explanation of Rack and Slot.

It's possible but it's not mandatory (Snap7 1.1.0) to specify the connection type via the function **Cli_SetConnectionType()** which must be called before Cli_ConnectTo(). By default the client connects as a **PG** (the programming console), with this function is possible change the connection resource type to **OP** (the Siemens HMI panel) or **S7 Basic** (a generic data transfer connection).
In the hardware configuration (Simatic Manager) of the CPU, under "Communication" tab, you can change, PLC-side, the connection's distribution, if you need.
PG, OP and S7 Basic communications are client-server connections, i.e. they don't require that the PLC have a connection designed by NetPro.
Note : This is an optimization function, if the client doesn't connect, the problem is **elsewhere**.

LOGO! 0BA7 /S7 200 via CP243

To connect to these PLC you need to call **Cli_SetConnectionParams()** and then **Cli_Connect().**

Cli_SetConnectionParams() needs of PLC IP Address, Local TSAP and Remote TSAP. There are two chapter dedicated to these PLC, for now let's say that TSAPs must follow what we wrote in the connection editors.

Remember that the TSAPs are crossed : Local TSAP PC-side is the Remote TSAP PLC-Side and vice-versa.

This function only sets the internal parameters, the real connection is made by Cli_Connect().

Note : when you use these functions don't call **Cli_SetConnectionType()**, it modifies the HI byte of the Remote TSAP !!!.

It's possible, but uncomfortable, to connect to a S7300 PLC using these functions as well.
To do this, use **0x0100** as Local TSAP and follow the next formula for the Remote TSAP.

```
RemoteTSAP=(ConnectionType<<8)+(Rack*0x20)+Slot; // C/C++/C#

RemoteTSAP:=(ConnectionType SHL 8)+(Rack*$20)+Slot; // Pascal
```

Where:

Connection Type	Value
PG	0x01
OP	0x02
S7 Basic	0x03..0x10

Remark

Snap7Server and Snap7Partner(in passive mode) accept any value for Local and Remote TSAP.

The internal connection mechanism is not really changed : all new functions are only for data-preparation.

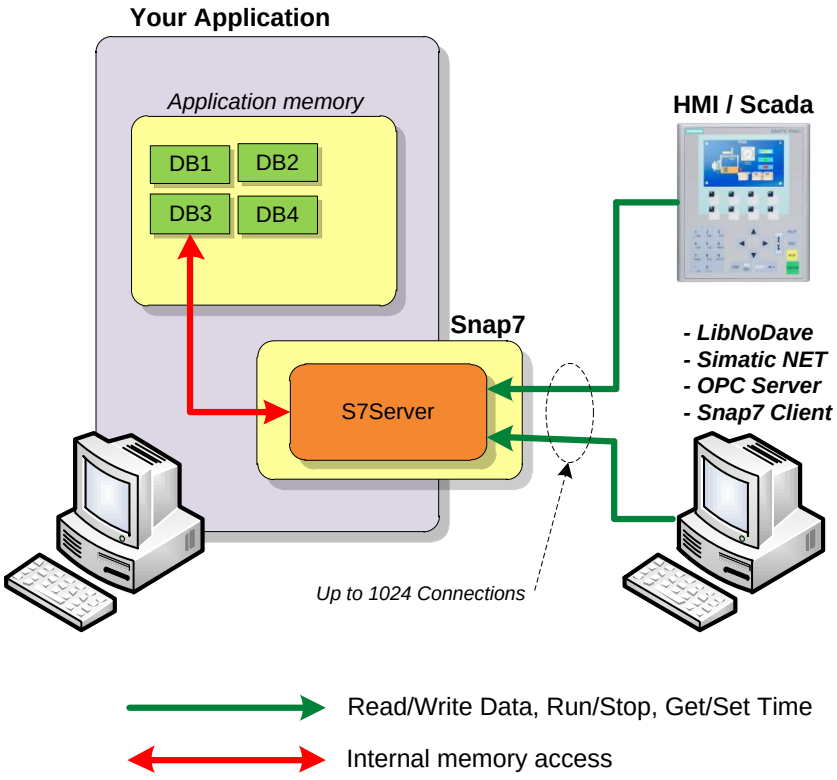
Snap7Server

Introduction

In spite of the fact that the Snap7Server is the easier object to use, initially it is probably the most complicated to digest.

Let's start to saying that Snap7Server neither is a kind of OPC Server nor is a program that gathers data from PLC and presents the results.

Snap7Server, just like a communication processor (CP), accepts S7 connections by external clients, and replies to their requests.



Just like the CPU that shares its resources with its CP, Your application must share its resources (memory blocks) with the server.

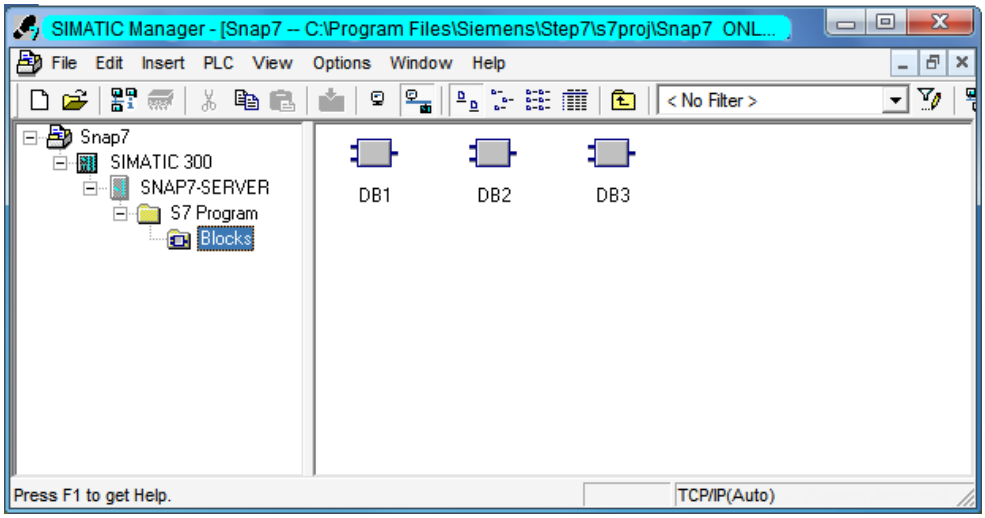
The mechanism is very simple:

- Your program allocates a memory block and says to the server "**this is your DB35**". Every time a client requests to read/write some byte from/to DB35, the server uses that block.
- If a client requests the access to an inexistent block (i.e. a block that you didn't shared) the server replies with an error of *resource not found*, just like a real PLC would do.

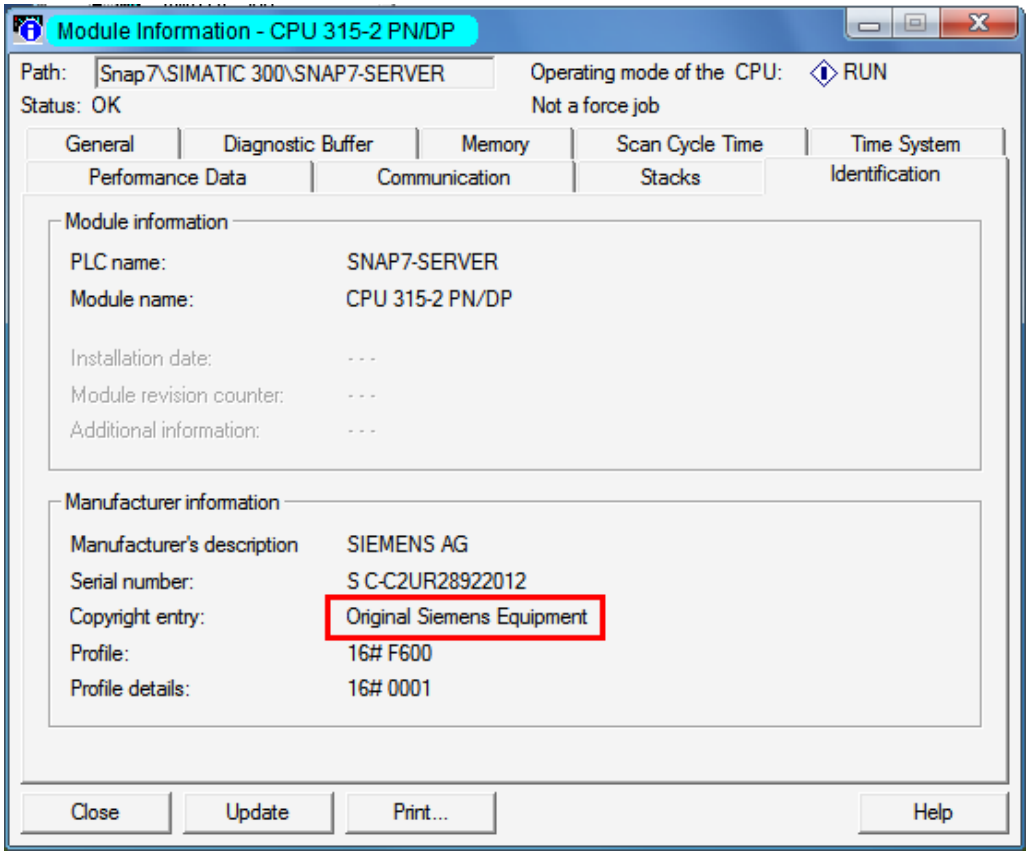
The client does not see any difference from a real PLC.

The simulation level is quite depth: S7 Manager (or TIA Portal) itself, sees your application as a CPU 315-2PN/DP.

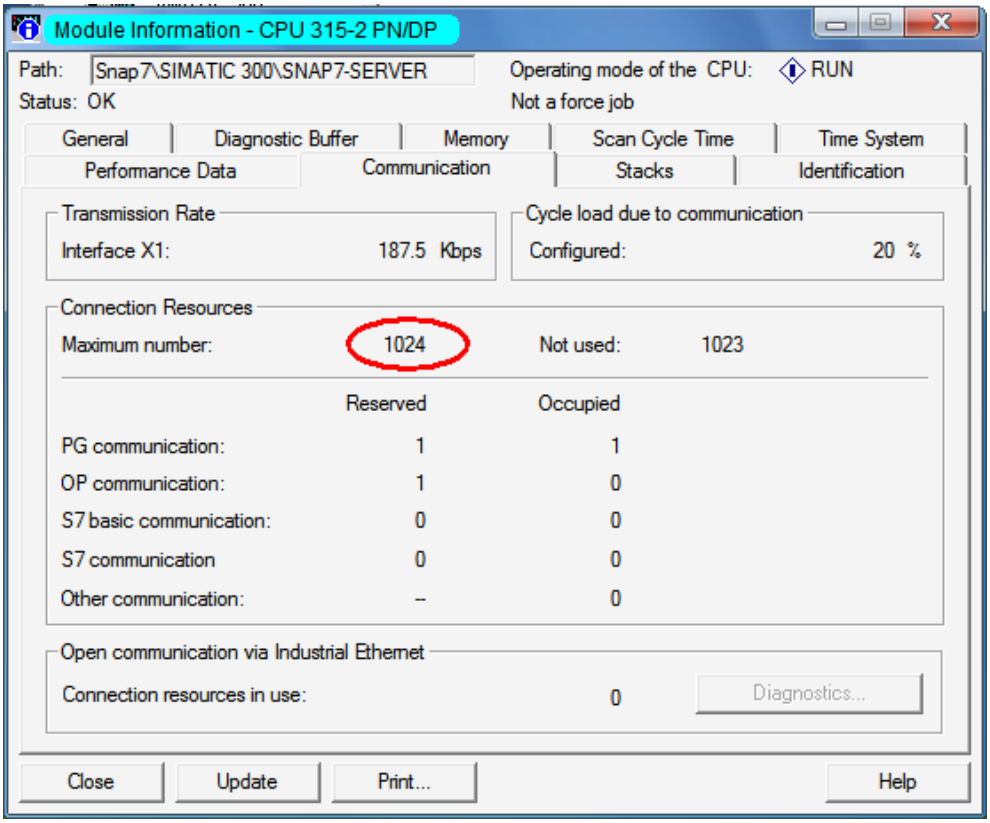
Online Project



Module information



Communication Info



Not being able to test Snap7Server with hundreds systems on the market, the key concept is:

- If Siemens Simatic manager sees the server as a real PLC, **more so** every client (Scada, hmi panel, PLC driver) will see the server as a real PLC.

You can obtain the preceding result with the simple following program :

```
#include <stdio.h>
#include <stdlib.h>
#include <cstring>
#include "snap7.h"

     TS7Server *Server;
     unsigned char DB1[512];  // Our DB1
     unsigned char DB2[128];  // Our DB2
     unsigned char DB3[1024]; // Our DB3
     unsigned char MB[2048];  // 2048 = CPU 315 Merkers amount

int main(int argc, char* argv[])
{
    int Error;
    Server = new TS7Server;
    Server->RegisterArea(srvAreaDB,     // We are registering a DB
                         1,             // Its number is 1 (DB1)
                         &DB1,          // Its address
                         sizeof(DB1));  // Its size
    // Do the same for DB2 and DB3
    Server->RegisterArea(srvAreaDB, 2, &DB2, sizeof(DB2));
    Server->RegisterArea(srvAreaDB, 3, &DB3, sizeof(DB3));
    // Let's share all Merkers from M0.0 to M2047.7
    Server->RegisterArea(srvAreaMK, 0, &MB, sizeof(MB));

    // Start the server onto the default adapter "0.0.0.0"
    Error=Server->Start();
    if (Error==0)
    {
       // Now the server is running ... wait a key to terminate
        getchar();
    }
    else
        printf("%s\n",SrvErrorText(Error).c_str());

    Server->Stop(); // not strictly needed
    delete Server;
}
```

Please refer to API reference for functions syntax.
You can find the C# and Pascal version into examples folders.

Notice that we shared also the merkers area.

Specifications

The Snap7Server is a multi-client multi-threaded server.

Once a connection is accepted, a new S7 worker thread is created which, from this moment will serve that client.
When the clients disconnects, the S7 worker is destroyed.

Up to 1024 (*) connection can be accepted, this value however can be changed via Srv_SetParam().

At the moment there is no Blacklist/Whitelist mechanism for filtering the connections, in a future release however it could be implemented (depending on the project audience).

The compatibility with Simatic Manager, of course, is not complete.

The server, as said, is a CP simulator **not** a SoftPLC, i.e. there isn't a MC7 program, compatible with Simatic Manager, to be edited, uploaded or downloaded : The business logic, if any, is your application program.

S7 functions implemented (in the current release)

- **Data I/O** (also via multivariable read/write)
  Read/Write DB, Mk, IPI, IPQ, Timers, and Counters.
- **Directory**
  List Blocks, List Blocks of Type, Block info.
- **Control** (1)
  Run/Stop, Compress and Copy Ram to Rom.
- **Date and Time** (2)
  Get/Set PLC Date and Time.
- **System Info**
  Read SZL
- **Security**
  Get/Set session password (3).

Some functions exist only to simulate a PLC presence, particularly :

(1) Run command is accepted and subsequent *get status* command will show the CPU as running, Stop command is accepted and subsequent *get status* command will show the CPU as stopped. But they have no practical effect on the server.
Compress and Copy Ram to Rom are accepted but (obviously) they do nothing.
(2) Get Date and time returns the Host (PC in which the server is running) date and time. Set date and time is accepted but the host date and time **is not modified**.
(3) Whatever password is accepted.

(*) The maximum number of open TCP connections depends also on the Host OS.

S7 functions not implemented (in the current release)

- **Block Upload/Download**
  Is accepted but the server replies that the operation cannot be accomplished because the security level is not met : we cannot download a block, a block must be created by the host application then shared with the server.
- **Programming functions**
  The server does not replies at all.
- **Cyclic data I/O**
  The server does not replies at all.

  Due to limit the memory footprint (the server must work fine into a Raspberry PI too), SDBs are not present; there is no scada, AFIK, that needs to access them.

Control flow

If you look at the previous small program, you see how to share resources between your application and the server.

We understood that the server replies automatically to the client requests, but :

There is a way to know what a client is requesting? Can we synchronize with it?

Is implemented a kind of log/debug mechanism?

Every time something happens in the server : when it is started, when it is stopped, when a client connects/disconnects or makes a request, an "event" is created.

The event is simply a struct defined as follow:

```
typedef struct{
    time_t EvtTime;     // Timestamp
    int EvtSender;      // Sender
    longword EvtCode;   // Event code
    word EvtRetCode;    // Event result
    word EvtParam1;     // Param 1 (if available)
    word EvtParam2;     // Param 2 (if available)
    word EvtParam3;     // Param 3 (if available)
    word EvtParam4;     // Param 4 (if available)
}TSrvEvent
```

EvtTime is the timestamp of the event, i.e. the date and time of its creation.

EvtSender is the IP of the Client involved in this event. The format is 32 bit integer to save memory, and can be converted into string, such as "192.168.0.34", using the socket function `inet_ntoa()` (Every OS socket layer has it).
If the event sender is the server itself (event generated on its startup for example), this value is 0.

EvtCode is the Event code, i.e. its identifier (see the list below).

EvtRetCode is the Event Result, it coincides with the result of the underlying S7 function if any, otherwise is 0.

EvtParam1..EvtParam4 are parameters whose meaning depends on the context.

In **snap_tcpsrv.h** and **s7_types.h** you will find all constants used.

EvtCode List

```
const longword evcServerStarted       = 0x00000001;
const longword evcServerStopped       = 0x00000002;
const longword evcListenerCannotStart = 0x00000004;
const longword evcClientAdded         = 0x00000008;
const longword evcClientRejected      = 0x00000010;
const longword evcClientNoRoom        = 0x00000020;
const longword evcClientException     = 0x00000040;
const longword evcClientDisconnected  = 0x00000080;
const longword evcClientTerminated    = 0x00000100;
const longword evcClientsDropped      = 0x00000200;
const longword evcReserved_00000400   = 0x00000400;
const longword evcReserved_00000800   = 0x00000800;
const longword evcReserved_00001000   = 0x00001000;
const longword evcReserved_00002000   = 0x00002000;
const longword evcReserved_00004000   = 0x00004000;
const longword evcReserved_00008000   = 0x00008000;
const longword evcPDUincoming         = 0x00010000;
const longword evcDataRead            = 0x00020000;
const longword evcDataWrite           = 0x00040000;
const longword evcNegotiatePDU        = 0x00080000;
const longword evcReadSZL             = 0x00100000;
const longword evcClock               = 0x00200000;
const longword evcUpload              = 0x00400000;
const longword evcDownload            = 0x00800000;
const longword evcDirectory           = 0x01000000;
const longword evcSecurity            = 0x02000000;
const longword evcControl             = 0x04000000;
const longword evcReserved_08000000   = 0x08000000;
const longword evcReserved_10000000   = 0x10000000;
const longword evcReserved_20000000   = 0x20000000;
const longword evcReserved_40000000   = 0x40000000;
const longword evcReserved_80000000   = 0x80000000;
```

The event generated follows 2 ways **: the events queue** and **the callbacks**

The Events queue is a FIFO list protected with a critical section to ensure events consistency and it's thread-safe.

Each S7 Worker inserts its events into the queue, your application extracts them using Srv_PickEvent().
If the queue is full, i.e. you don't call Srv_PickEvent or call it too slowly, the event is not inserted and it's simply discarded.

On calling Srv_ClearEvents() the queue is flushed.

The Event queue is designed for **log** purpose.

The next code snippet is extracted from ServerDemo (Pascal rich-demo).

- LogTimer is a cyclic timer procedure.
- Log is a Text Memo object.

```
procedure TFrmServer.LogTimer(Sender: TObject);
Var
  Event : TSrvEvent;
begin
  // Updates Log memo
  if Server.PickEvent(Event) then
  begin
    if Log.Lines.Count>1024 then // to limit the size
      Log.Lines.Clear;
    Log.Lines.Append(SrvEventText(Event));
  end;
  // Updates other Server Infos
  ServerStatus:=Server.ServerStatus;
  ClientsCount:=Server.ClientsCount;
end;
```

Finally, **SrvEventText()** returns the textual string of the event.

This is a sample output of this function:

```
2013-06-25 15:39:11 Server started
2013-06-25 15:39:24 [192.168.0.70] Client added
2013-06-25 15:39:24 [192.168.0.70] The client requires a PDU size of 480 bytes
2013-06-25 15:39:24 [192.168.0.70] Read SZL request, ID:0x0132 INDEX:0x0004 --> OK
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0000 INDEX:0x0000 --> OK
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0111 INDEX:0x0001 --> OK
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0424 INDEX:0x0000 --> OK
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0f74 INDEX:0x0000 --> OK
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0074 INDEX:0x0000 --> OK
```

Ok, but I do not want to be bored with messages about SZL or Date/Time, can I filter them ?

Yes ,you can filter them checking the EvtCode parameter but, even better, the server can make this for you.

If you look at the EvtCode List, you will notice that each event occupies one bit into a 32 bit word.
It's possible to pass to the server a BitMask word, named **LogMask**, whose bits will act as "and gate" for the events.

For example, this mask 0xFFFFFFFE has the first bit set to 0.
If you pass it to the server, every event except for "Server Started" will be stored.

Callbacks
While, as said, the Event queue is designed for log purpose, the callback mechanism is designed for control purpose.

A user function named Callback is invoked by the S7 Worker when an event is created.

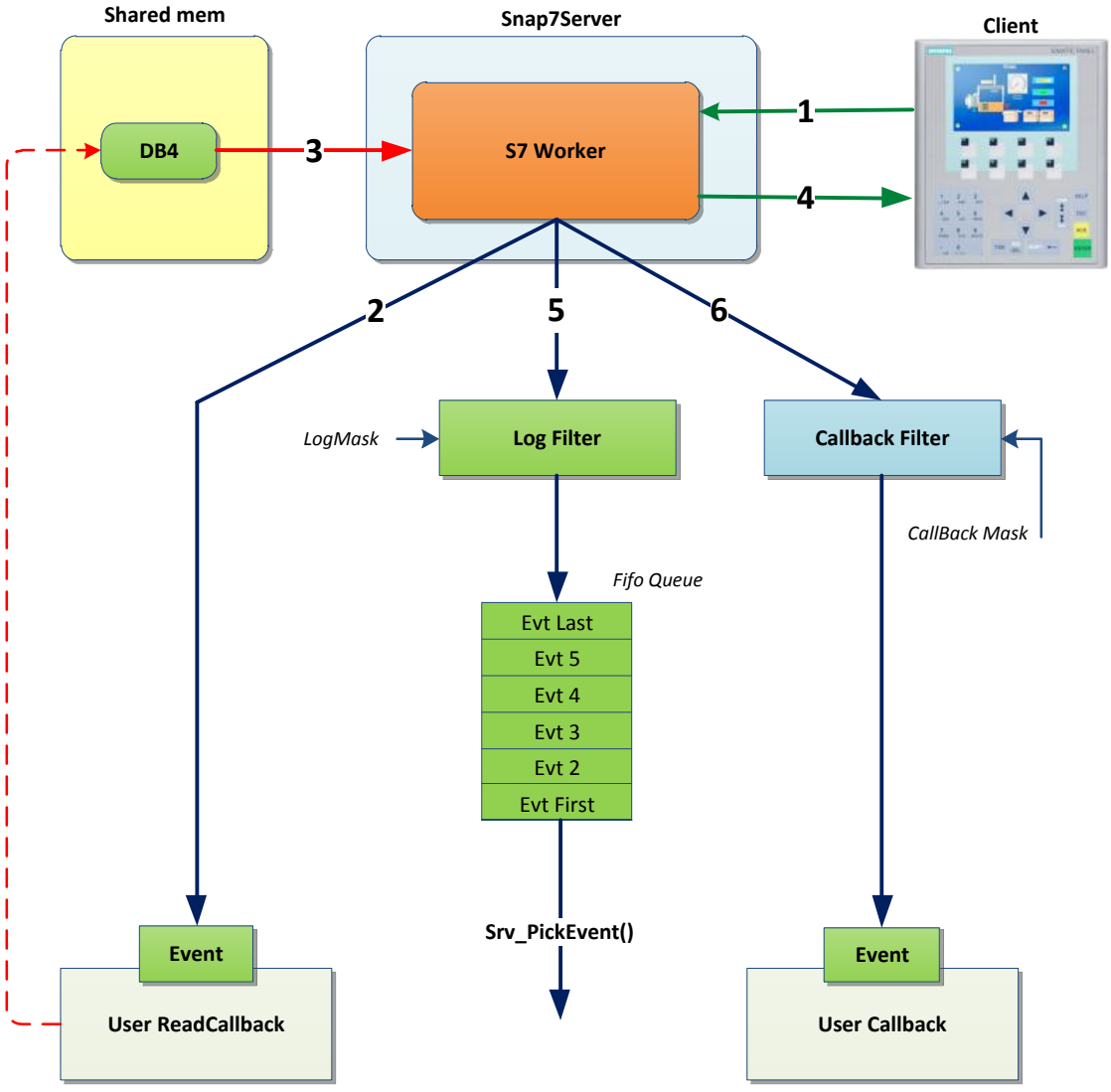
Also the callback mechanism is filtered with its own bitmask and is protected with a critical section.

There are two differents callbacks (from Snap7 1.1.0) the Read Callback and the Common Callback.
Both callbacks are executed in the same thread of the S7 Worker. The first is invoked on read request before performing the data transfer from Snap7Server to the Client, the second after the handshake with the client to avoid Client timeout since the code inside the callback locks the worker.

See Srv_SetReadEventsCallback() and Srv_SetEventsCallback() for further information about the callback prototype.
Log Mask and Callback mask are different, by default are set both to 0xFFFFFFFF (all enabled) on the server creation.

The purpose of the ReadEventCallback is for writing protocol converters or gateways.

Finally let's see the complete sequence.



1. The Client Requests to read some data from DB4.

The Worker:

2. Invokes the Read Callback (if assigned) passing it the read coordinates.
   Into the Read Callback we are able to modify DB4 if we want.
3. Gets the Data from DB4.
4. Sends the Data and Job result to the client.
5. Checks the Log Filter and inserts the Event into the queue.
6. Checks the Callback filter and, if the callback is assigned, calls the user function passing the event as parameter.
7. Becomes ready for further Client requests.

Remarks

The post-handshake event callback is called **also** on Data Read, so if you use the Read Callback, you should disable the read event into the Event mask to avoid a double notification for the same event.

Queue and callbacks mechanisms are not mandatory to be used in a program, if you don't pick the events from the queue or not assign a callback, nothing bad will happen.

Data consistency

Since the main application shares its resources with the server, a synchronization method is needed to ensure the data consistency.

When a memory block is shared via **Srv_RegisterArea(),** the server creates a block descriptor.

This descriptor contains

- Block number (it's used only if the block is a DB).
- Block memory address.
- Block size.
- A CriticalSection Object reference.

Just that object ensures the data consistency.

The S7 worker "locks" the memory block every time it needs to access it, and unlocks it at the end.

To improve the performances, a **double-buffer** method is used: the S7 worker first receives the data into an internal buffer and then copies the content into the shared block.
Or, it copies the needed data from the shared block into the internal buffer before sending them.

Only the copy operation locks the block.

If you need data consistency, you must accomplish this rule.

For this purpose there are two functions : **Srv_LockArea()** and **Srv_UnlockArea()**.

You should use the first one to lock a memory block and the second one to unlock it.

On long operations I suggest you to adopt the same double-buffer strategy : use an internal buffer then transfer the data into the shared block.
Moreover, an exception raised when the block is locked will lead to a S7 worker freeze.

Note

The granularity of the consistency is the PDU size.

Multiple servers

In preparation to receive connections a socket must be bound to a 2-tuple value : **(IP Address, Listening Port).**

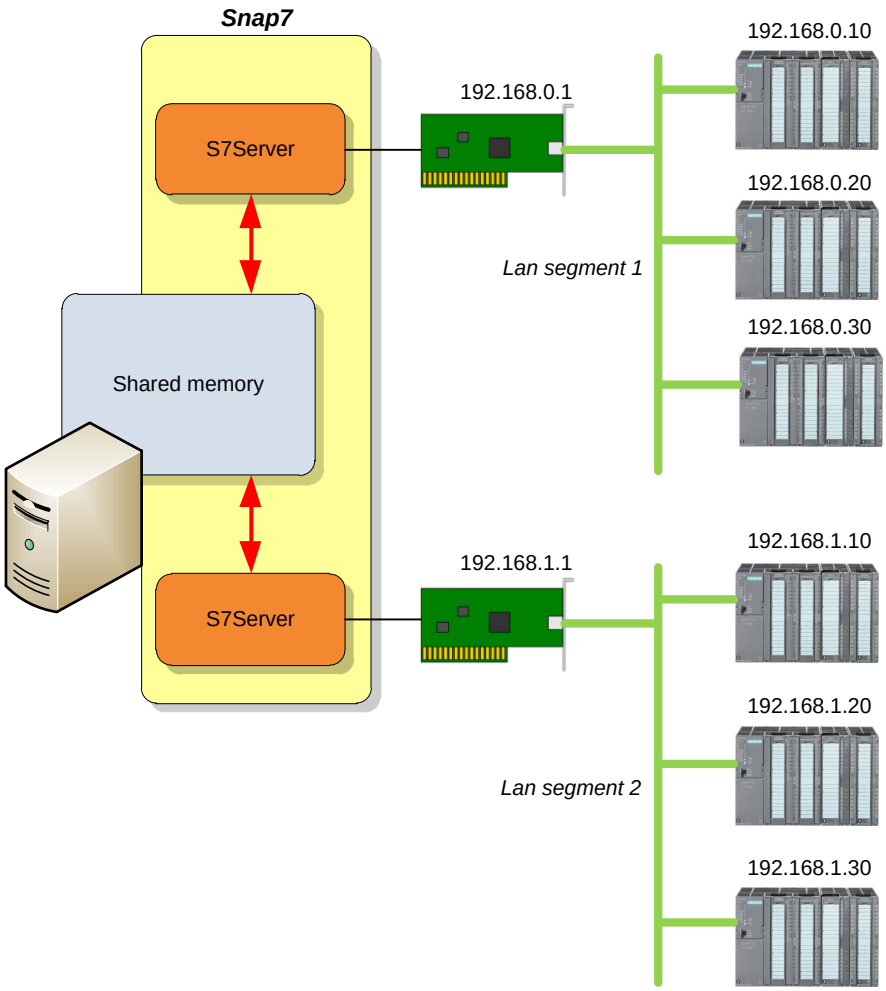
These coordinates are unique.

You can have a Telnet Server, a HTTP Server and a NTP Server running in the same machine because, though they have the same IP, they are listening on different ports.

You can have two HTTP Servers in the same machine that has two network adapters (i.e. two different IP addresses).

Established this (Berkley Sockets) rule, you can create multiple Snap7Servers but each of them **must be "started" onto a different network adapter**, because the listening port (ISO TCP - 102) cannot be changed.

It could be useful run two servers to share data between two different network segments.



If you plan to use a physical server, 16 adapters is the maximum suggested.
If you need more, consider to use a virtual infrastructure.

Troubleshooting

There are mainly three reasons for which a start error is generated:

The first, trivial, is that the bind address is wrong.

Windows

In the PC in which you are trying to start the server is installed the Step 7 / TIA Portal environment.

Step 7 has a server listening onto the port 102 : **s7oiehsx.**

To overcome this problem you can temporary stop it, by running one of the batch files:

stop_s7oiehsx_x86.bat (32 bit) or

stop_s7oiehsx_x64.bat (64 bit)

To re-run the Step 7 service use their counterpart **start_s7oiehsx_xxx.bat.**

You can find them in the examples folder.

Tanks to www.runmode.com

Unix

ISO TCP port is a *well-known port,* so in Unix you need **root** rights, or your application must have the SUID flag set, to bind a socket to this port.

There is not a workaround for this.

If a client does not connect with the server, check your **Firewall settings** (especially if the host OS is Windows 7 / Windows 8).

Step 7 Project

This is a sample **PG Project** of the Snap7Server, with this project you can:

- Connect the Step 7 Manager (or Tia Portal if you convert it) to the Snap7Server and see it online.
- Insert it in a multi-cpu project.
- Integrate a **WinCC flexible** project into it.

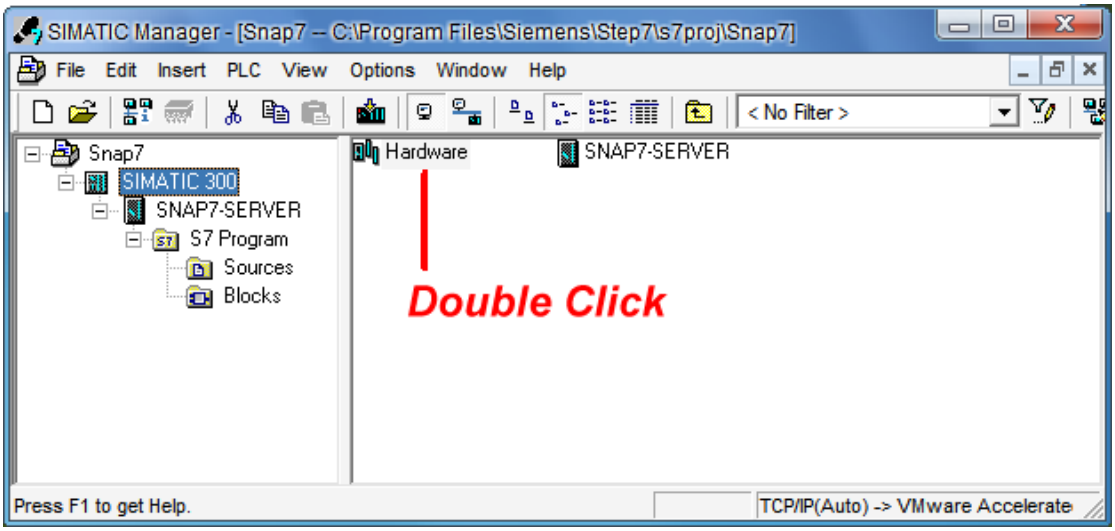
You can find it in the folder **examples/Step7/Server**

To use it you only need to setup the network IP address (as you would do, in a real project).

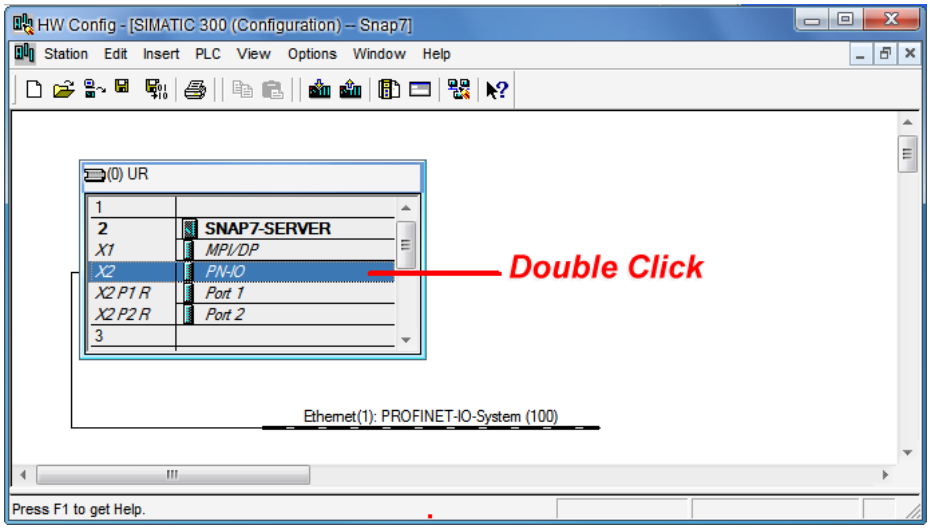
There are few steps to follow:

1) Load the project into Step 7 Simatic Manager.

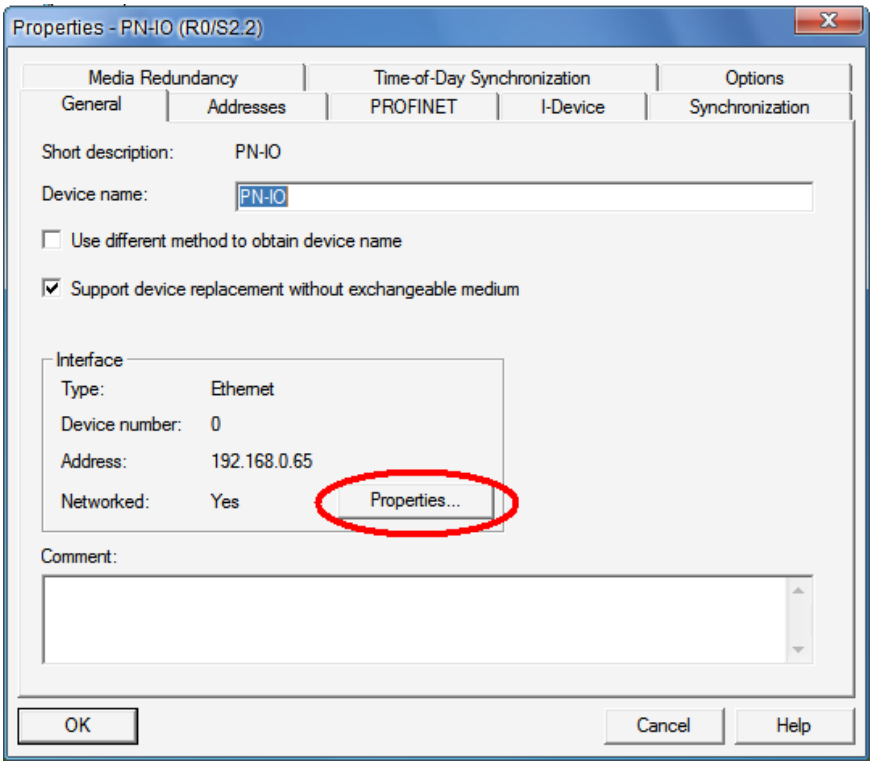
2) Open the Hardware Configuration



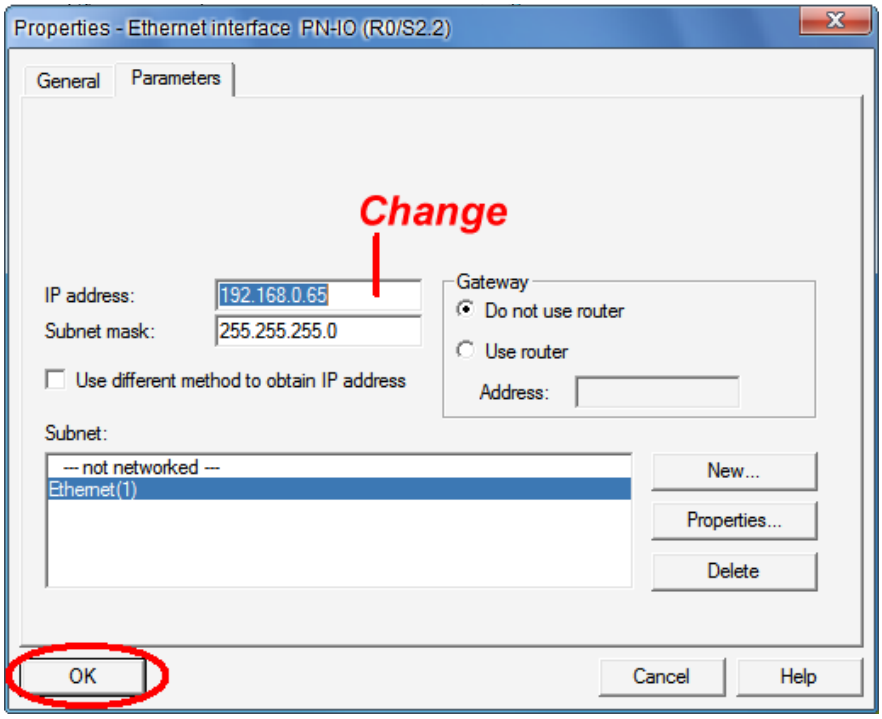
3) Open the PN-IO Interface editor



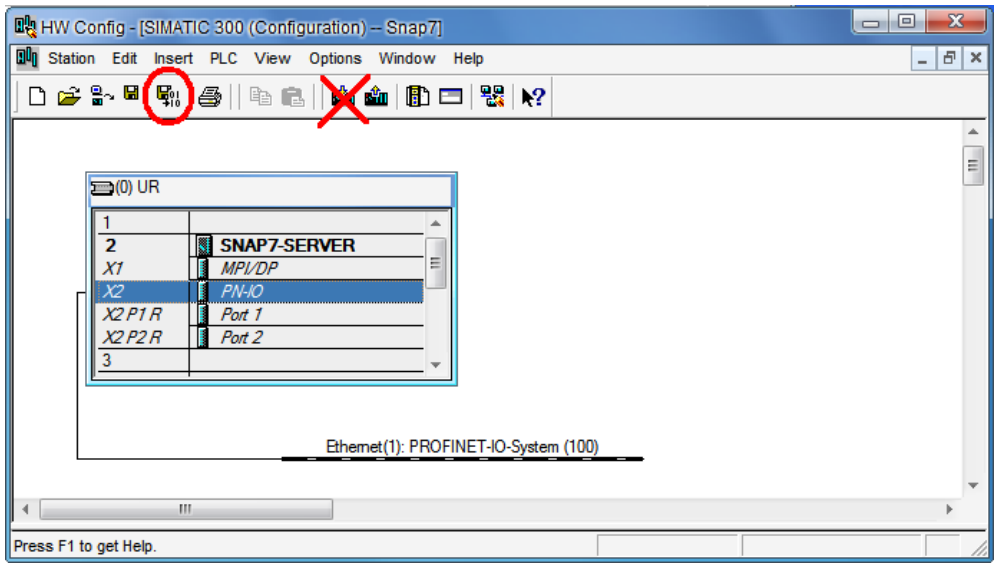
4) Open the Network properties editor



5) Set the IP Address of the PC in which the server will run (and confirm pressing the OK Button).



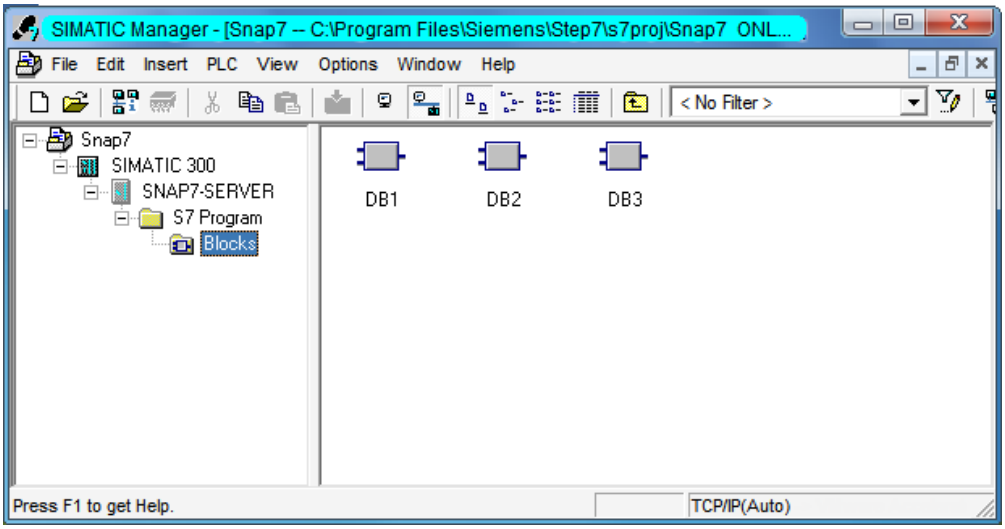
6) Close all previous windows, Save & Compile the Project, don't download it.



7) You can close the Hardware Configuration Editor, that's all.

In the host PC, run a Server program, any of the ones you find in the example folder or the rich-demo ServerDemo if you want.

Now you can go ONLINE with Simatic Manager, remember to set **TCP/IP Auto** interface.



You can modify the **offline** project adding DB, variables and so on, and link them to WinCC variables.
Remember, obviously, to create the same DBs into your application and to share them.

Remarks

Snap7Server is not visible via the "Display accessible nodes" function of Simatic manager, because to find the Ethernet nodes a Profinet packet (ServiceID=5, Discover, All) is used.

However, even WinAC, is not visible with this method if it's equipped only with IE (Industrial Ethernet) standard adapter.

Server Applications

The final 1.000.000 € (Eur. because I'm Italian) question is:

Why do I need this Object ?

Let's see two real scenarios:

Using PLC-aware hardware with your software

You have PC-based automation but:

- You don't want to create graphic screens.
- You don't want to "expose" the system with a standard keyboard.

With Snap7Server you can connect an HMI-Panel (Siemens/ESA/Pro-Face/Open source Scada) to your application and use a standard HMI-Builder.

Tomorrow you can migrate your application to another OS without be bored by graphic libraries porting.

Variant of the above:

You have a Raspberry-based (or other small Linux card) system and:

- You don't want to use an HDMI monitor.
- And you don't like a sad SSH interface.
- You like a small ISO-rail mounted box, wired using only two cables (network and power).

Finally:

Snap7Server allows your embedded hardware to be connected by hundred standard systems Siemens-Aware.

Integration in a PLC environment.

You build an analogic test bench to be inserted in an existing PLC-based production line and sadly realize that:

- Your bench is PC-based.
- The line has a commercial scada as supervisor.
- You don't want to buy a PLC as line interface layer.

The commercial scada can be smoothly interfaced with your application via a Snap7Server (moreover this solution is faster).

Snap7Partner

The Smart7Partner allows you to create a S7 peer to peer communication.

The Siemens model

Unlike the client-server model, where the client makes a request and the server replies to it, the peer to peer model sees two components with same rights, each of them can send data asynchronously.

The only difference between them is the one who is requesting the connection.

The partner that requests the connection is named **active,** the one that accept the connection is named **passive**.
Once the connection is established they can send unsolicited data.

The S7 protocol, as said, is command oriented, and the commands usually are executed by a server.

The partners communicate via the S7 Protocol, but using the untyped telegram "**segmented data send**".

This is not strictly a command, it's a data transfer that uses the S7 Protocol acknowledge mechanism.
In fact, it is not recognized by the client-server pair.

The communication is not fully asynchronous, i.e. there is no interrupt mechanism that says to the receiver that a packet is incoming : a partner to receive a packet , must be listening via a Block Recv function.

As said in *The Siemens Theatre* chapter, two PLC to communicate with this mechanism must have a S7 connection created with NetPro.

If the two PLC systems reside in the same project, Simatic Manager can keep track of the connections, otherwise both will have an **unspecified partner** as counterpart.
But, from the data transfer point of view there is no difference.

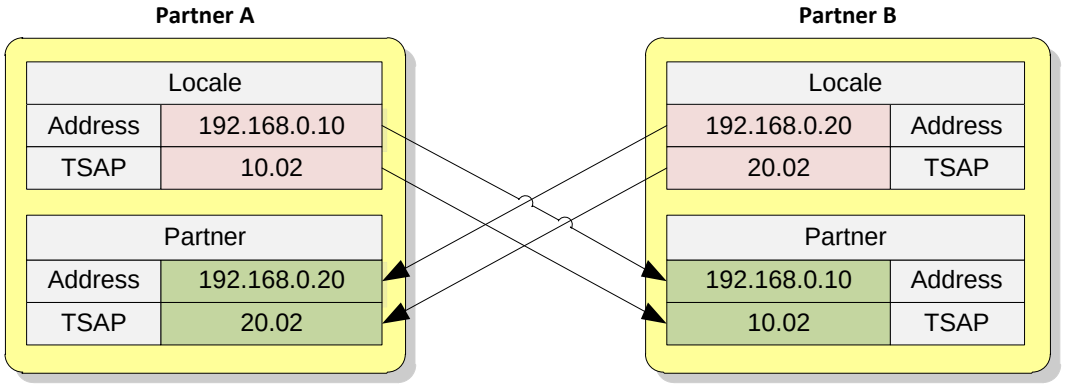
The partner addressing mechanism is not exactly easy to understand, however, let's try to simplify our life.

Two friends want to talk to each other, both they have a transceiver with antenna very directional.
To communicate, each needs to know their own geographical coordinates and those of the other : latitude, longitude and elevation, to properly orient the antenna.

Two PLC partners want to talk each other.
To communicate, each needs to know their own S7 Network coordinates and those of the other : **IP Address** and **TSAP**.

The TSAP concept belongs to the ISO/OSI Layer 4 but it's not necessary to know it deeply, (moreover the Siemens use, does not follow strictly the specification) let's say only that it is a number composed by **HI Part** and **LO Part**.

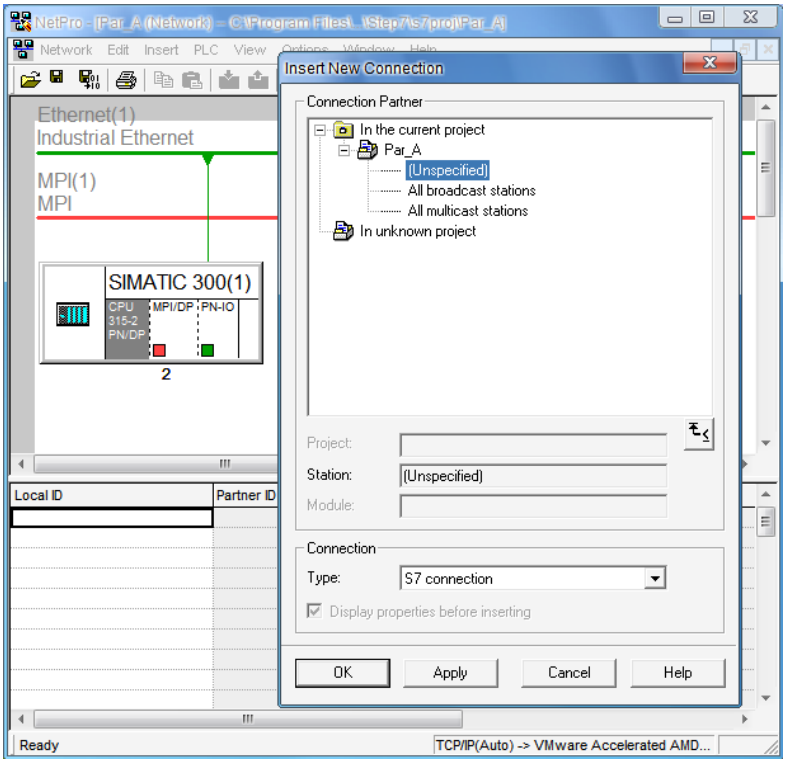
The main important thing is that all coordinates must be unique and **crossed,** just like a RS232 cable.



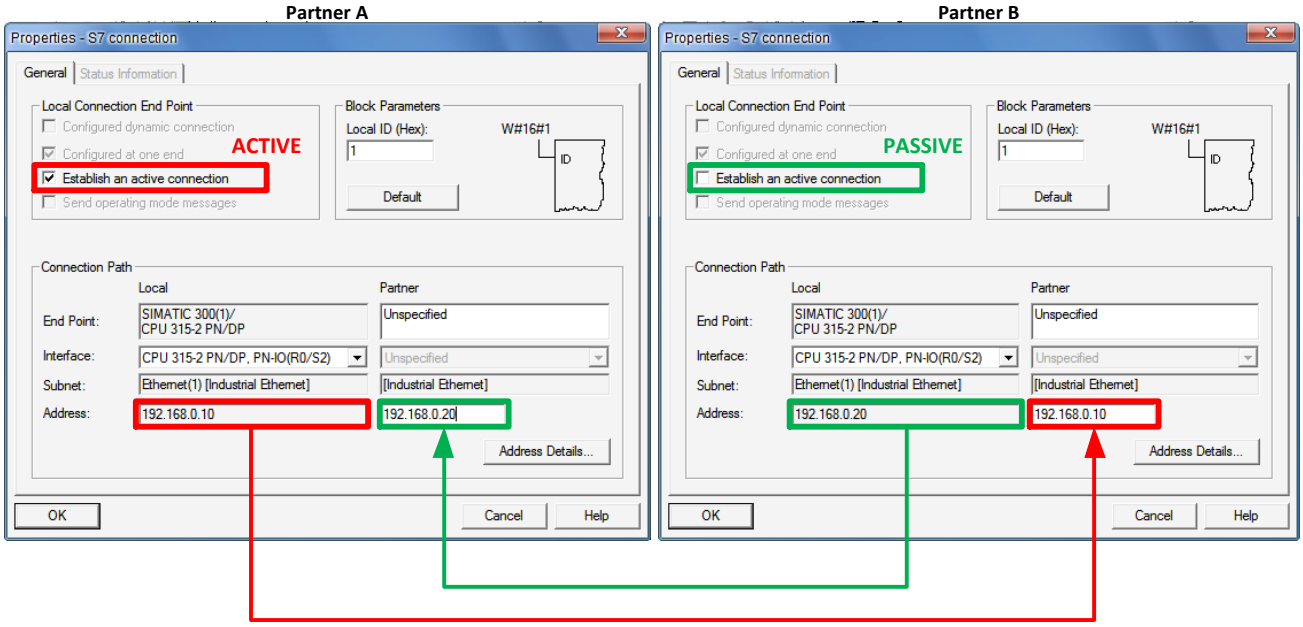
Note:
The HI part of the TSAP is not related with the lowest byte of the address in the example, it's just a coincidence.

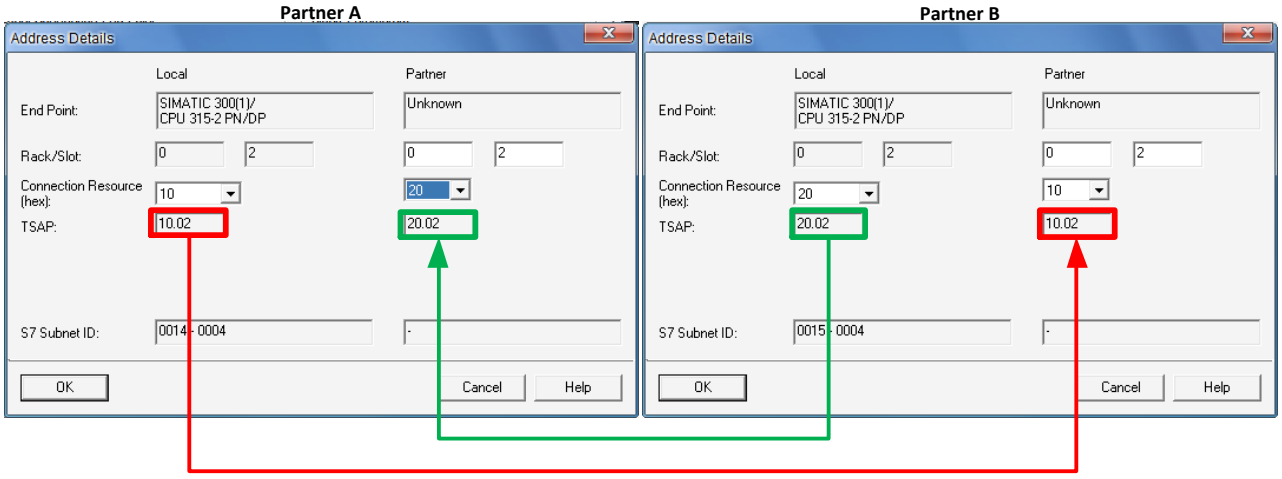
Now, Understood the mechanism, let's translate it into S7 NetPro.

For both projects, open NetPro, select the CPU and create a new connection (menu **Insert->New Connection**) as follow.



Then edit them; here we supposed that Partner A is the **active** one.





You cannot edit the TSAP value, it's obtained starting from Rack, Slot and Connection Resource as follow:

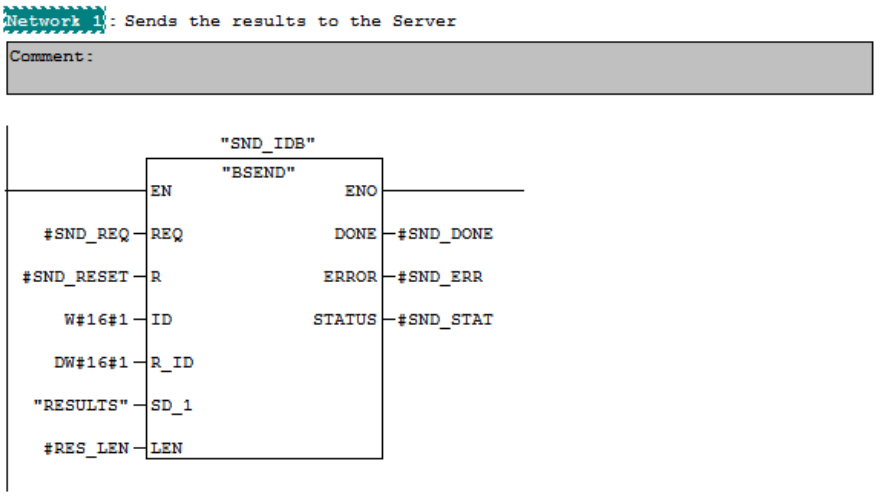
TSAP-HI = Connection Resource
TSAP-LO = <Rack * 2><Slot>.

In the description of the function **Cli_ConnectTo()** you can find a detailed description of what Rack and Slot are.

Local ID (in the picture is W#16#1) is the Connection ID that must be used into the communication FB, leave unchanged what NetPro proposes.

Pick **Connection Resource** from the List, any not already-used value in both partners is good enough.

With this configuration, the two partners can communicate via FB12/FB13 (S7300) or SFB12/SFB13 (S7400), BSend/BRecv.



The parameter **R_ID** allows a further level of routing, the pair BSEND/BRECV must have the same value for it.

There is an important remark about it (to avoid a big headache).

If you use FB12 (S7300) you can change it at runtime, it is sampled on the rising edge of the REQ parameter.

The S7400/WinAC SFB12, instead, samples it on the rising edge of the **stop->run** sequence, you can change it across two transmissions, but this has no effect : the telegram will always be sent with the original R_ID and there is no way to debug it, since it's an internal parameter.

The Snap7 model

The Snap7 model for the partner, faithfully follows the Siemens one, with some benefits:

1. No connection configuration is needed.
2. Two data send model :
    a. **Asynchronous** with three completion models.
    b. **Synchronous** - the caller is blocked until data are sent.
3. Two data receive models
    a. **Asynchronous** - a callback is invoked when there is an incoming packet
    b. **Synchronous** - calling BRecv just like using FB13.

There is a detailed description of the asynchronous data flow model in the Snap7Client, please refer to it for the theoretical part.

For the data exchange point of view the partners are equal as we will see, some remarks need to be done about the **connection process**.

When we create a partner we need to specify its type : **Active** or **Passive,** their behavior, as explained in the Siemens model, is different.
There is no way to change the partner type, once created.

Active Partner

It is quite simple to understand, it behaves like a client : requests a connection to the passive partner and waits the connection ack.

It's also easy to understand that we can create how many partners we want and connect them to their passive counterpart.

Passive Partner

The passive partner behaves like a server : it waits for a connection request, listening onto the IsoTCP port.

Said that, we could think that we can create only **one partner per adapter** since we cannot bind two socket to the same 2-tuple (Address, Port).

Luckily there is a programming trick, the **Connection Server** that works as follow:

We create as many passive partners we want.

Each of them, when started, says to the connection server :
"Boy, I'm waiting for a connection from a partner whose address is 192.168.10.30, call me if it passes here.."

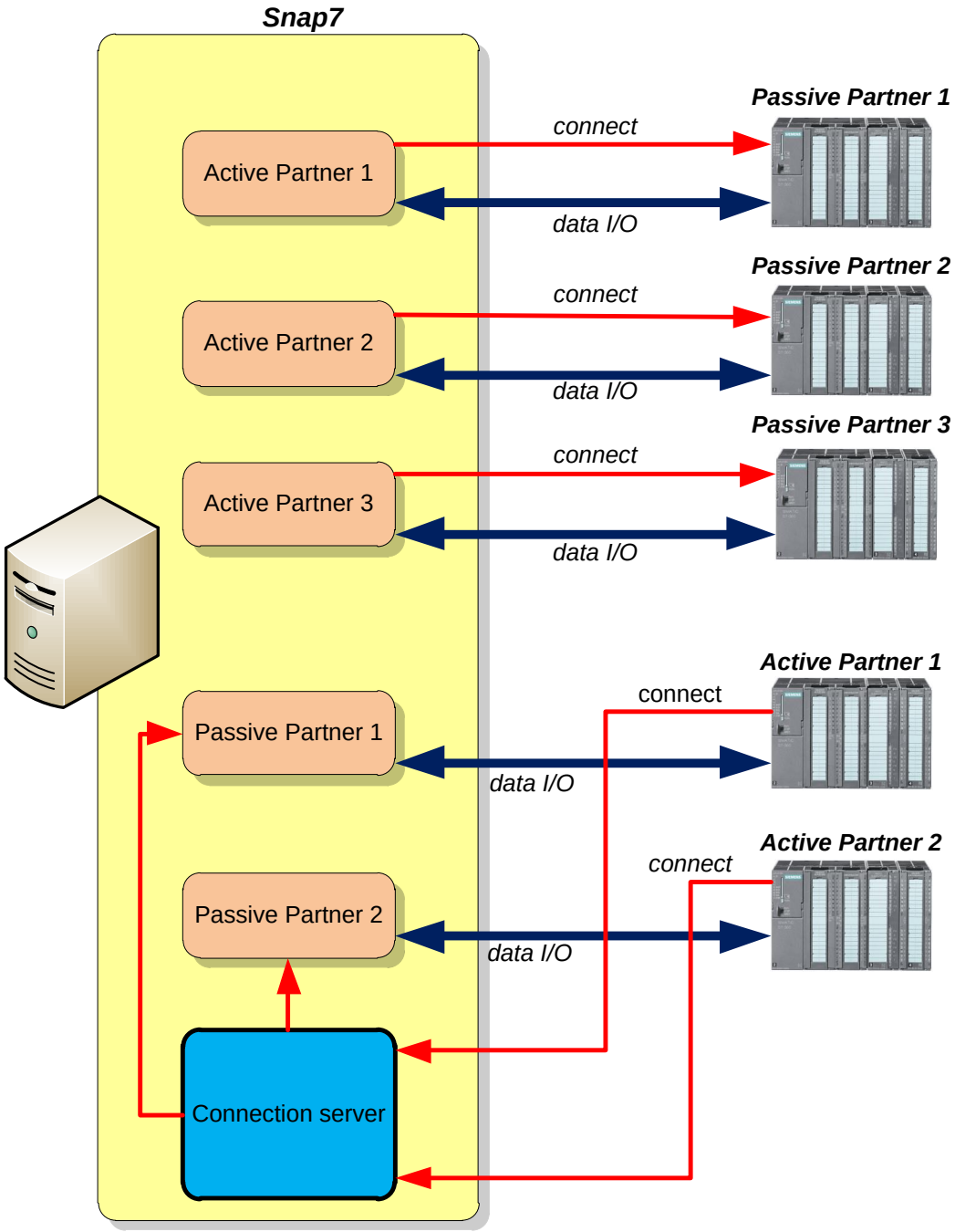
Thus the connection server adds it into its passive partners list.

When an Active Partner requests for a connection, the connection server reads its address and scans the list for someone that is waiting for it.
If there is such client, it accepts the connection and gives the connected socket to that passive client : the TCP connection now is established and they can communicate.

If no such partner is found in the list, the connection is refused.

The connection server behaves like a Snap7Server, the difference is that the Snap7Server accepts any connection and creates its workers.

More then one connection servers can be created if passive partners have different local addresses.
Active and passive partners can coexist in the same application, as follow :



In order to optimize the host "bindable" addresses, the connection server is created on the "start" of the first passive partner and is destroyed on the "stop" of the last passive partner.

The use of active partners does not involve the server creation.

Partner use

Let's see now the functions that Snap7 provides for working with partners.
You can find their exact syntax in the Partner API chapter, now we focus on their behavior.

Creation

A Partner must be created via the function Par_Create(int Active), where Active can be 1 or 0.

In the wrapper objects this parameter is boolean, and is internally converted to integer, this because is always preferable to avoid booleans parameters in multi-architecture/multi-compilers/multi OS programs.

Their differences stop here, from now every function is used for both partners.

Connection

Let's see how to proceed with an example.

We have two partners, the first one is a Snap7Partner, and the other is a PLC partner. As seen, we need to create a network connection with NetPro for the real PLC.

We create this connection as follow :

- Leave unchanged Local Address and Local TSAP that NetPro offers us.
- As "Partner Address" we insert the PC IP Address.
- As "Partner TSAP" we insert whatever unused value.
- If we created the Snap7Partner as Active, we "uncheck" the flag "Establish an active connection" and vice-versa.

The Snap7Partner doesn't need of a connection configuration, to work, it must be started with the "crossed" parameters.

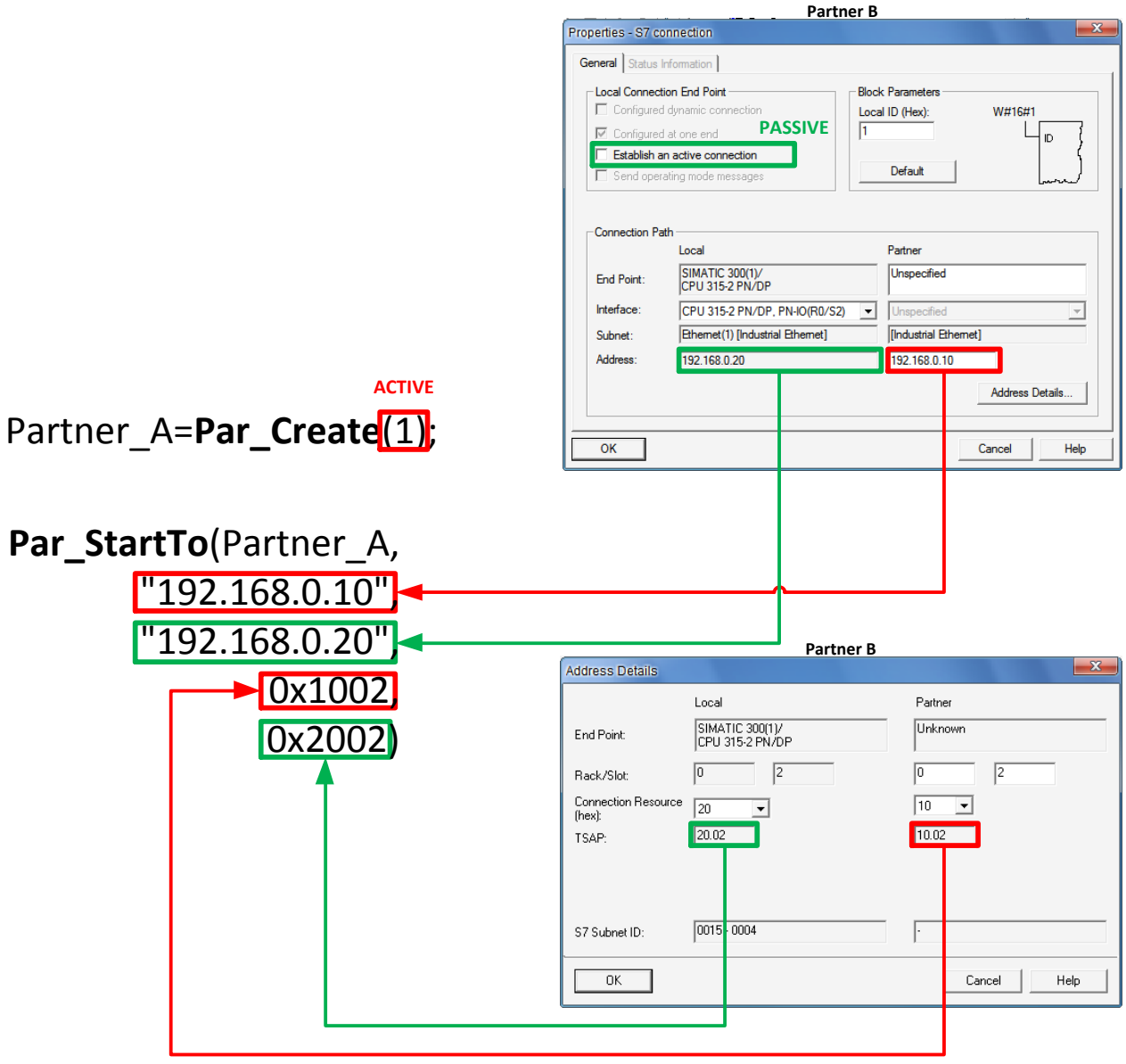
The function is:

```
Par_StartTo(S7Object Partner,
    const char *LocalAddress,
    const char *RemoteAddress,
    word LocTsap,
    word RemTsap);
```

Where

- LocalAddress is the IP Address of the PC in which our partner is running.
- RemoteAddress is the IP Address of the PLC.
- LocTsap is the local TSAP, we copy it from the PLC network configuration.
- RemTsap is the PLC TSAP, also this is copied from NetPro.

This example supposes that the Snap7Partner is the active one and has "192.168.0.10" as IP Address.



Remark

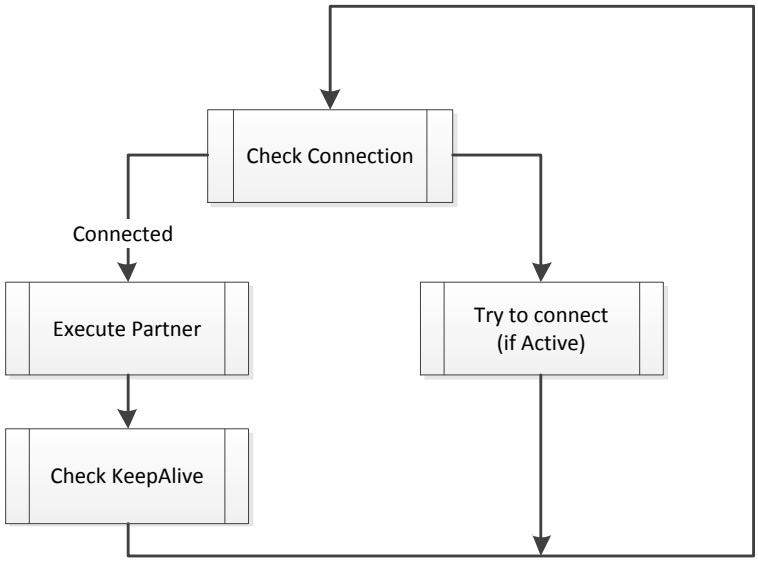
The remote TSAP parameter in a passive Snap7Partner has no effect, any connection request is accepted, and the only requirement is the IP Address.

Partner thread

The sending and receiving data process are externally independent , however is necessary, inside the partner, to synchronize them.

To best way to understand how this happens, is to examine their flowchart.

On start, the Snap7Partner creates a worker thread:



Partner Main Thread

It's an endless loop, if the connection checks are satisfied it executes the main work of the partner.

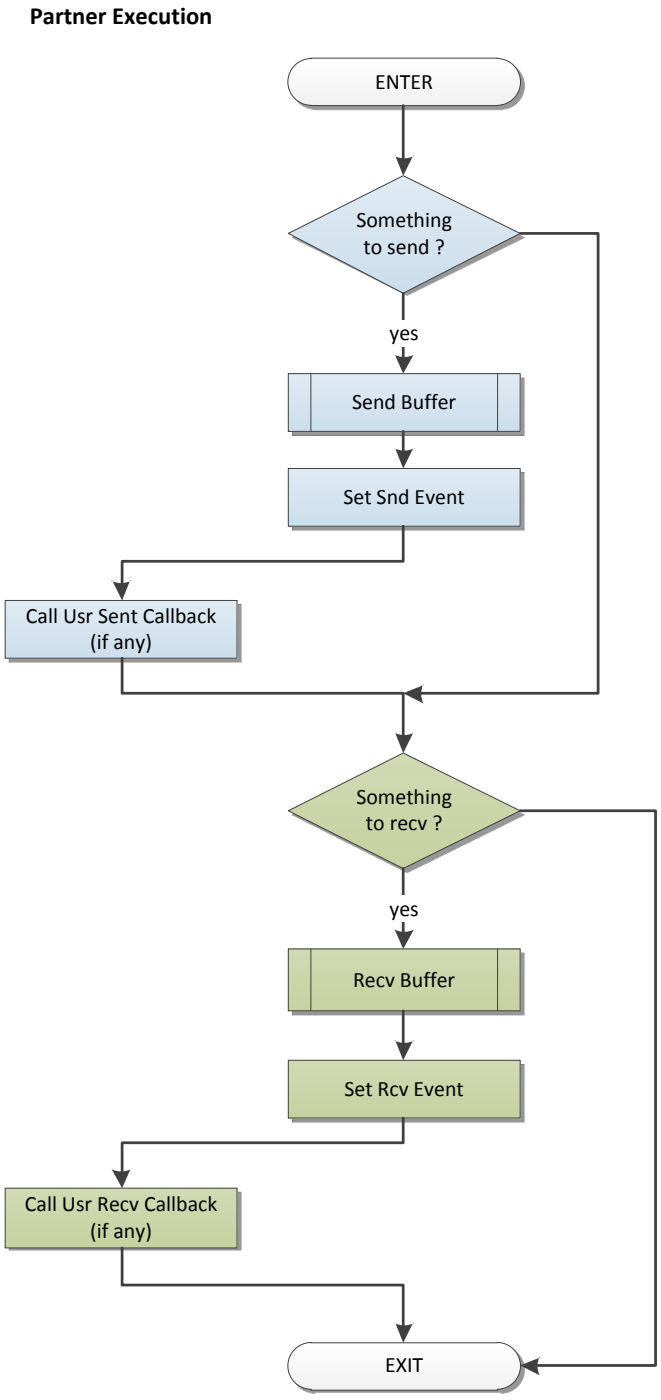
If a partner is connected (if active) or has an external connection (passive) has its status set as **Linked.**

The send and receive jobs are execute in the same thread but are mutually exclusive, this because a job involves many subsequent telegrams.

This is not a big penalty, since the communication channel of the PLC partner is arbitrated in the same half-duplex way.

At the end of the send task the Send Event is triggered and the user callback (if assigned) is invoked; the same happens at the end of the recv task.

To keep the synchronization, the send event is always cleared when a send job is started; the recv event is cleared by the BRecv function.



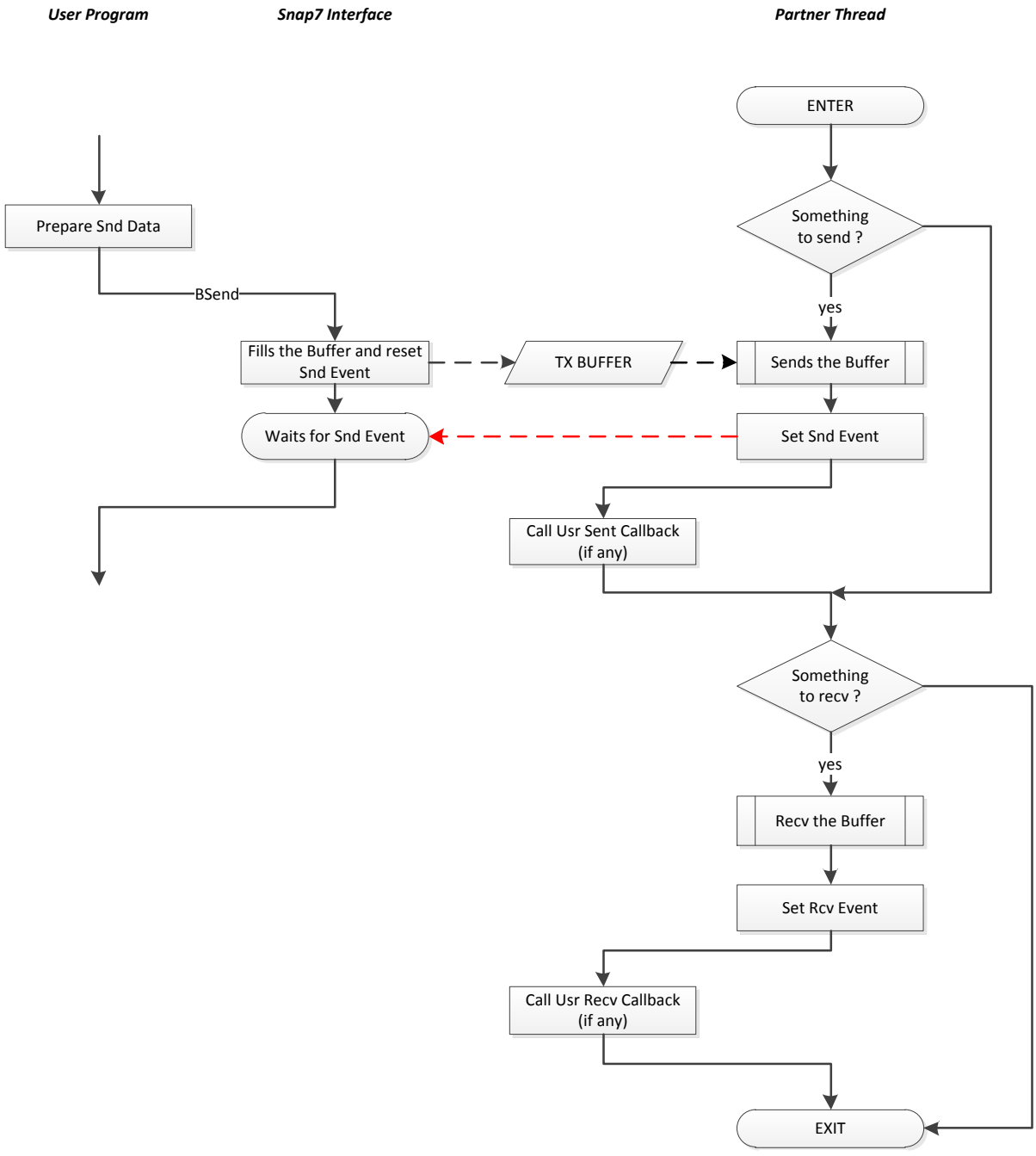
The Event/Callback mechanism, as we saw for the Snap7Client, allows a flexible way to handle the data flow.

Data Send

To send data to a remote partner we can use two functions : **BSend** and **AsBSend**.

The first is synchronous i.e. the caller execution is blocked until the job is finished. To be pedantic, this is not a "pure" synchronous function, since its body is executed inside the worker thread, but the result is the same.

BSend

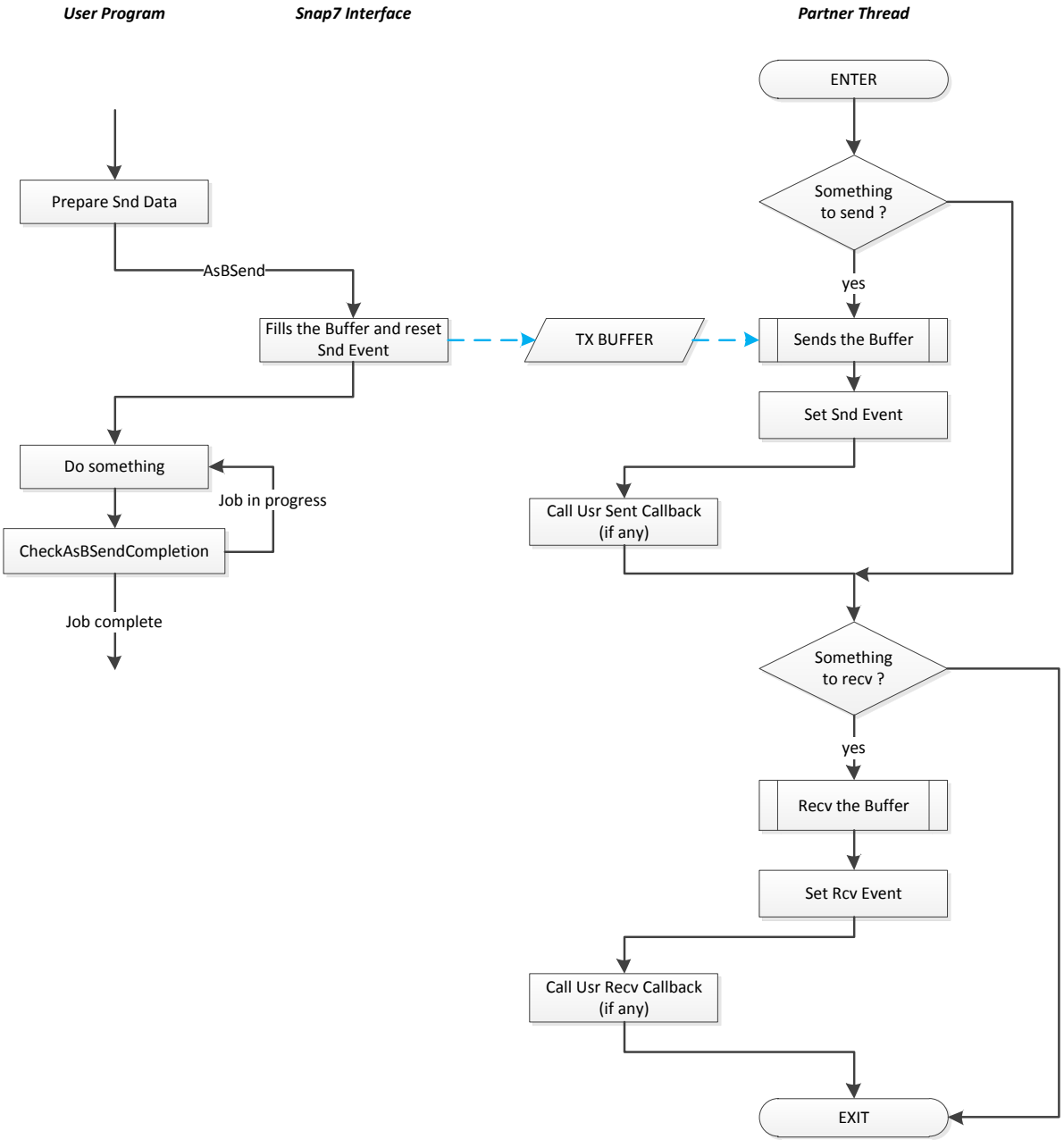


The function terminates when the data are correctly received by the remote partner or when the timeout expired.

AsBSend is asynchronous, the data are copied in the TX Buffer and the function terminates immediately.

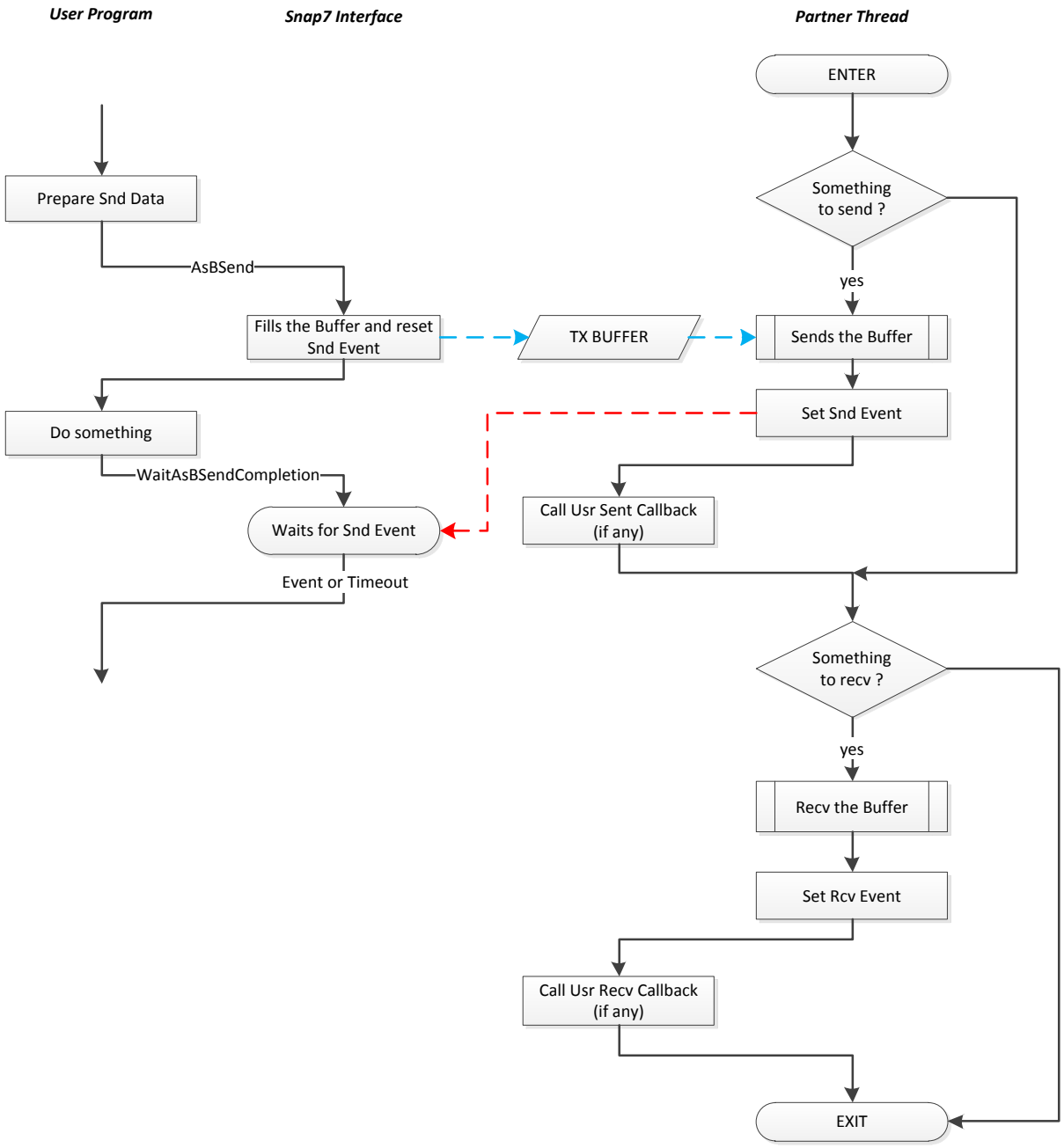
To know when the job is complete we can use three methods :

1) The polling via **CheckAsBSendCompletion** function



CheckAsBSendCompletion terminates immediately returning the job state : complete or in progress.

2) Use the function **WaitAsBSendCompletion**



This function terminates when the event is triggered or when the timeout expired.

Notice that BSend = AsBSend + WaitAsBSendCompletion.

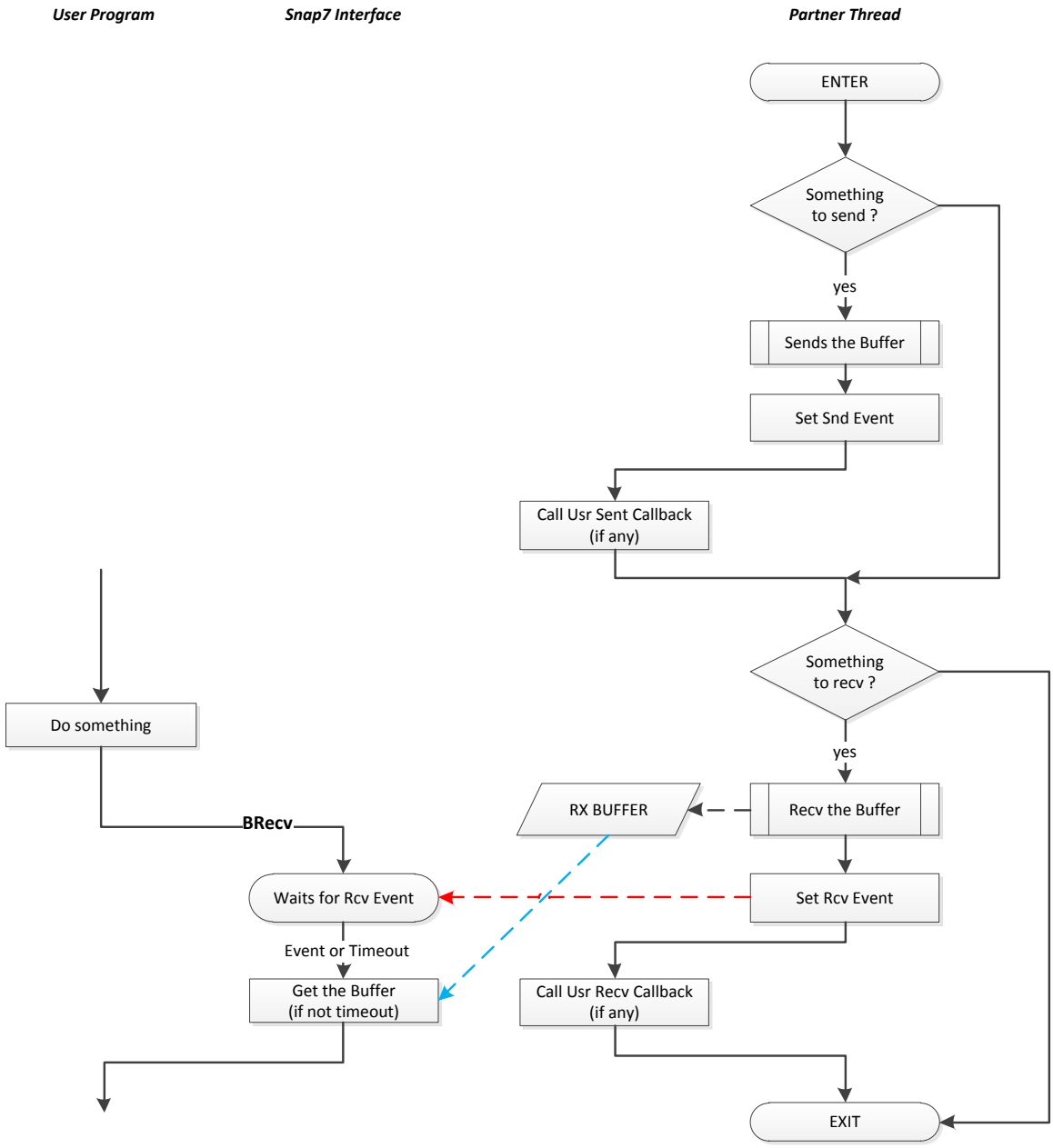
3) Write a synchronization code into the callback.

Data Recv

We can receive a data packet in three ways :

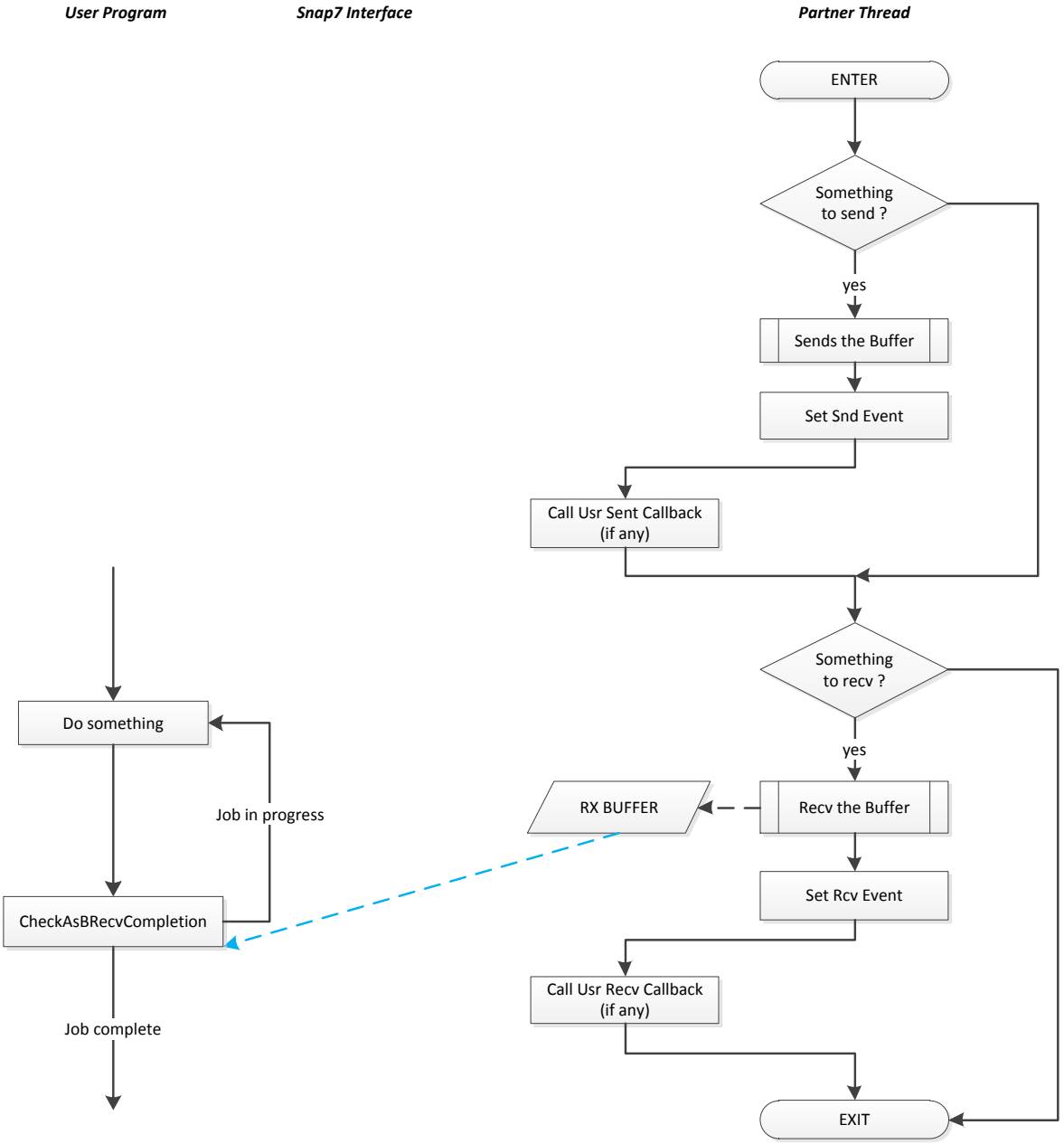
1. Using **BRecv** synchronously.
2. Polling
3. Writing user code into a **callback**.

BRecv



Obviously, we can use BRecv only if we know that the remote partner is sending something, otherwise a timeout error is returned.

Polling



For info about the callback method see ***Partner Applications.***

Partner Applications

Surely you can insert your PC based automation in a PLCs context where they communicate with BSend/BRecv, but this is only a beneficial side effect.

The main purpose of Snap7Partner is the massive data collecting in industrial facilities using the robustness of the S7 Protocol.

Suppose to have a production line with the following characteristics:

- 75 stations PLC based.
- Takt Time : 5.0 sec./pcs.
- Each station must record its own process data into a server.
- These data cannot be queued, each station needs the data of the preceding one.

After excluding any polling mechanism, we need a "**store on demand**" model.
To realize this is very simple with Snap7Partner.

Let's suppose to create a TStation class that contains a Snap7Partner linked to the station PLC.

ParDataIncoming is the gateway callback that each partner will invoke when an incoming data packet will be ready (1).

The station class initializes the partner callback passing it the ParDataIncoming address and its reference "this" (or *Self* in Pascal) as **usrPtr**; so usrPtr is the fingerprint of the station.

When a Partner receives a data packet, it invokes the callback supplying:

- **usrPtr** that received during the initialization.
- The operation result **opResult** that is not zero if an error was detected.
- The address of the buffer in which is stored the packet : **pData.**
- The packet **Size** in bytes (2).
- The parameter **R_ID** that was passed to FB12

The callback "casts" usrPtr to a station reference and calls its member **StoreData** passing it the remaining parameters.

StoreData now can check opResult and record the data where it wants…

An important note is that Snap7Partner completes the handshake with the PLC **before calling the callback**.
This means that the record process time can be extended up to the next packet incoming without worrying of the S7 timeout.

Notes

(1) If you program in C# there is no "plain" gateway callback, the partner will invoke directly a delegate method of the station class (see C# partner examples).

(2) This means that we don't need to know in advance the packet size, it can vary across the transmissions.

C++

```
// Class definition
class TStation()
{
private:
    TS7Partner *Partner;
public:
    TStation();
    void StoreData(longword R_ID, void *pData, int Size);
};

// Class implementation
TStation::TStation()
{
    // Partner creation
    Partner = new TS7Partner(true); // or false
    // Callback set
    Partner->SetBRecvCallback(ParDataIncoming, this);
    // "this" parameter is the fingerprint of TStation instance.
}
void TStation::StoreData(longword R_ID, void *pData, int Size)
{
// Store the received data
}

// Callback shared between all partners
void S7API ParDataIncoming(void *usrPtr, int opResult,
    longword R_ID, void *pData, int Size,)
{
    // Cast usrPtr to TStation
    TStation *MyStation  = (TStation *) usrPtr;
    // Call the member
    if (opResult==0)
        MyStation->StoreData(R_ID, pData, Size);
}
```

News from 1.1.0

LOGO! 0BA7

LOGO is a small Siemens PLC suited for implementing simple automation tasks in industry and building management systems.

It's very user friendly and the last model is equipped with an ethernet port for both programming and data exchange.

The Snap7 focus is on to S7300/400 systems, but due to several requests, i decided to manage this PLC as well.

Communication

Due to its architecture, the LOGO communication is different from its Siemens cousins.

It implements two Ethernet protocols, the first that we can call **PG protocol,** is used by the software LOGO Comfort (the developing tool) to perform system tasks such as program upload/download, run/stop and configuration.
The second, used for data exchange, is the well known (from the Snap7 point of view) S7 Protocol.

They are very different, and the first is not covered by Snap7 because is a stand-alone protocol that is not present, Is far I know, in different contexts.

Although LOGO uses the S7 Protocol, some small changes (new connection functions added) were needed in Snap7 to manage it as PLC Server by a Snap7Client.

You can use LOGO as Client connecting it to a Snap7Server or you can connect a Snap7Client to a LOGO set as Server.

Finally, to communicate with LOGO, the Ethernet connections must be designed with LOGO Comfort in advance.
Of course I will show you how.

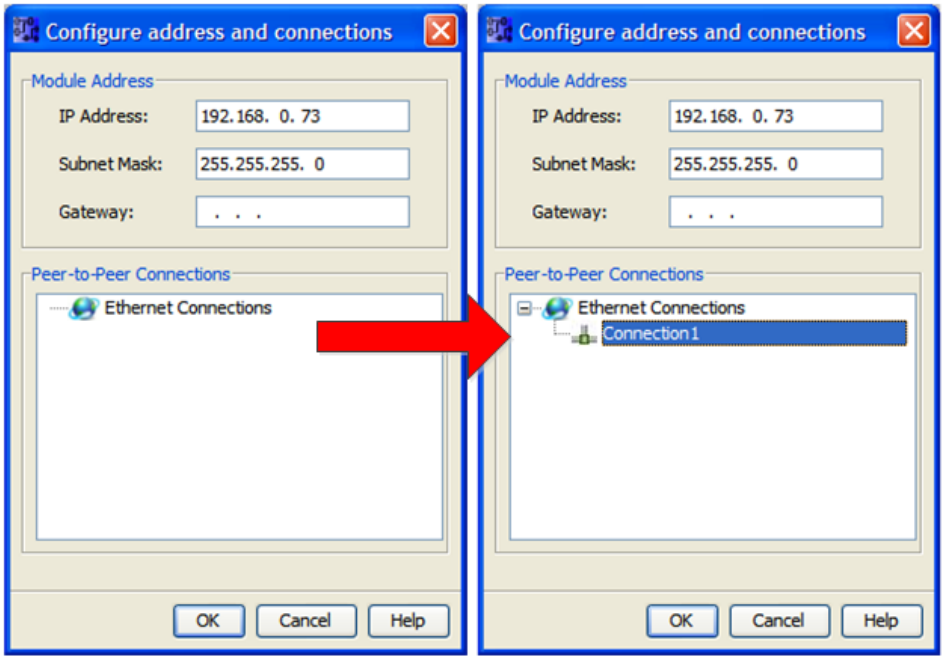
For both type of communication (Client-Server or Server-Client) LOGO must be set as MASTER (i.e. NORMAL mode as LOGO Comfort says).

I assume that your LOGO Comfort is already set and connected to the LOGO.

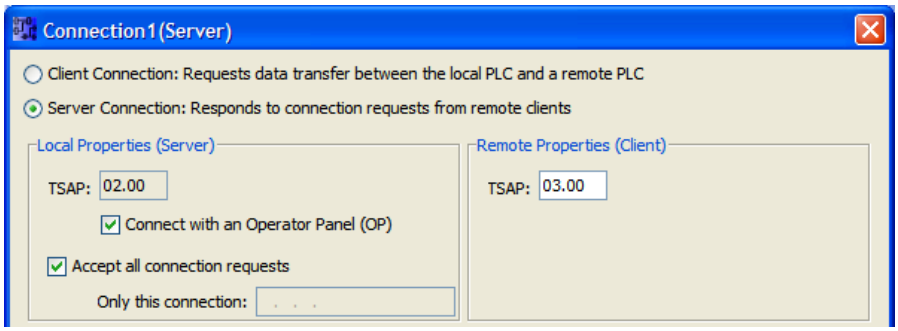
LOGO as Server

Configuring a **server connection** allows you to connect LOGO with Snap7Client for reading and writeing the memory just like an HMI panel would do.

- In the **Tools** menu choose the **Ethernet Connections** item.
- Right click on ”Ethernet Connections” and click ”Add connections” to add a connection



- Double-click the new connection created and edit its parameters selecting **Server Connection**.

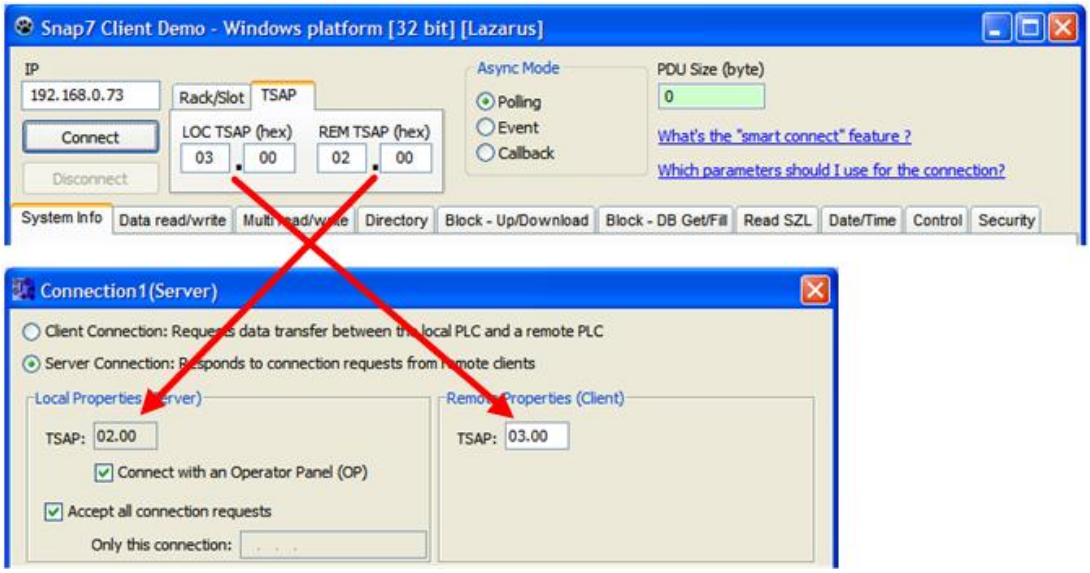


Note:

1. "Connect with an operator panel" checkbox can be checked or unchecked.
2. If you uncheck "Accept all connections" you must specify the PC address (for now I suggest you to do it checked).

You can chose for Remote TSAP the same value of the Local TSAP, in the example I used two different vaules to remark (as you will see) the **crossing parameters**.

- Confirm the dialog, close the connection editor and **download** the configuration into the LOGO.
- The LOGO is ready, to test it run the new ClientDemo, insert the LOGO IP Address and select the TSAP Tab for the connection as in figure.



Notice that the Local TSAP of the Client corresponds to the Remote TSAP of the LOGO and vice-versa. This is the key concept for the S7 connections.

The LOGO memory that we can Read/Write is the **V** area that is seen by all HMI (and Snap7 too) as **DB 1.**
Into it are mapped all LOGO resources organized by bit, byte or word.

There are several tutorials in the Siemens site that show how to connect an HMI (via WinCC flexible or TIA) to the LOGO and the detailed map.
Please refer to them for further informations.

Finally, to connect to the LOGO by program with the same parameters of above:

```
Client->SetConnectionParams("192.168.0.73", 0x0300, 0x0200); // C++
Client->Connect();

Client.SetConnectionParams("192.168.0.73", 0x0300, 0x0200);  // C#
Client.Connect();

Client.SetConnectionParams('192.168.0.73', $0300, $0200);    // Pascal
Client.Connect;
```

LOGO as Client

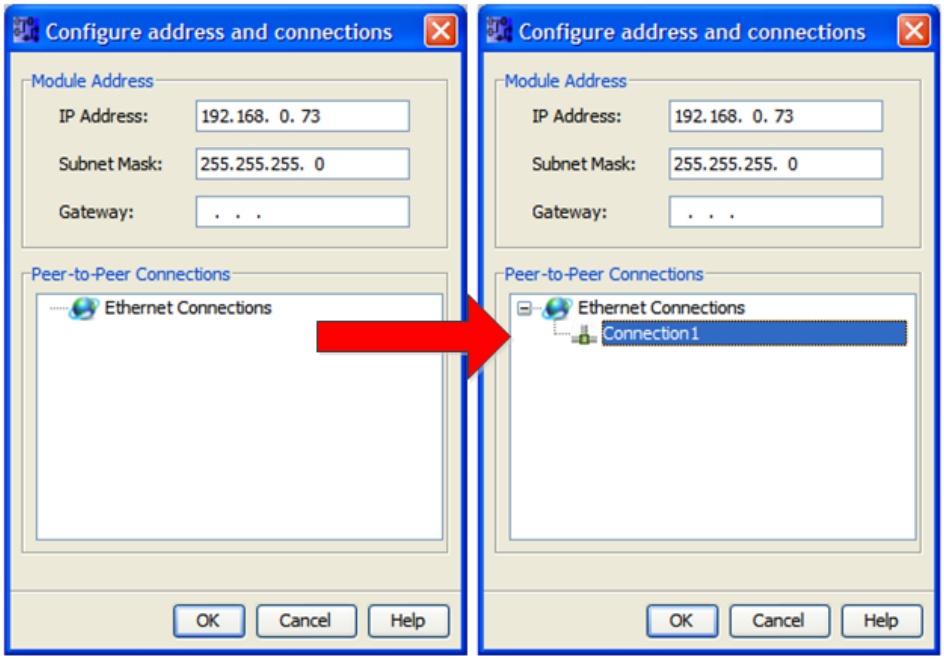
LOGO can work as Client in two way :

1. Explicit - using a client connection.
2. Implicit - via Network I/O blocks.

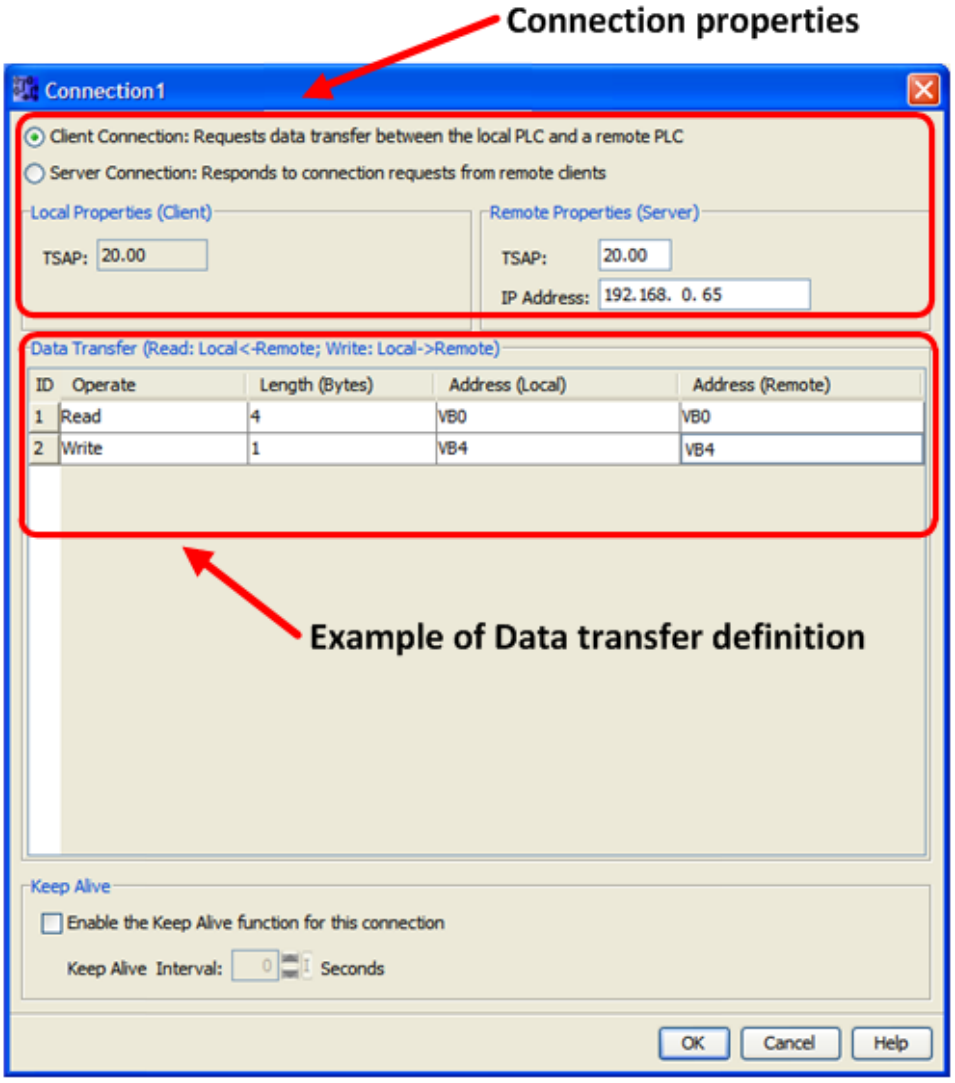
In both cases you can connect LOGO to a Snap7Server that acts like a slave.

Client connection

- In the **Tools** menu choose the **Ethernet Connections** item.
- Right click on ”Ethernet Connections” and click ”Add connections” to add a connection

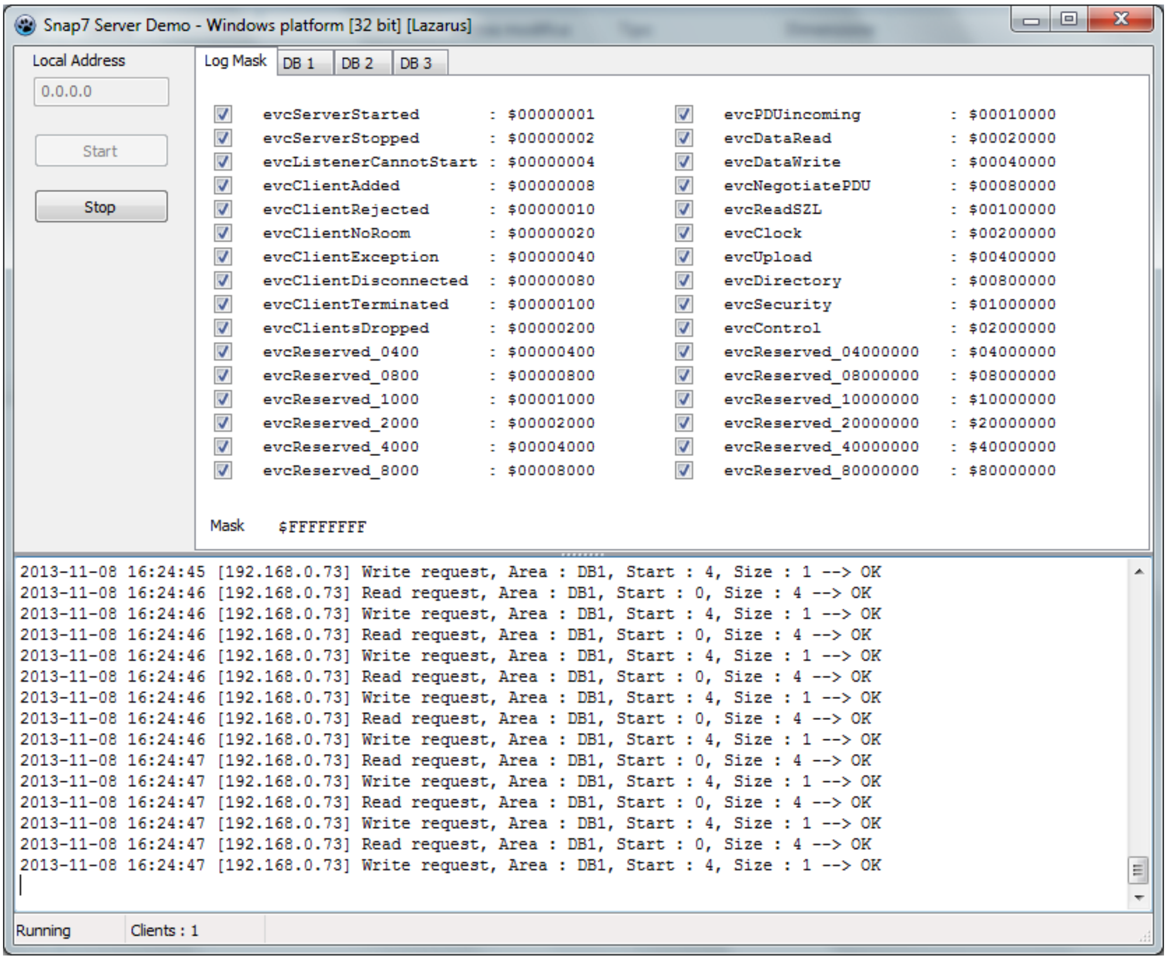


- Double-click the new connection created and edit its parameters selecting **Client Connection**.



- Insert the Server IP Address, as TSAP you can use whatever you want, since Snap7Server doesn't care of it.
- In the second area you can specify the data exchange area.
- As usual, confirm everything and download the configuration into the LOGO.

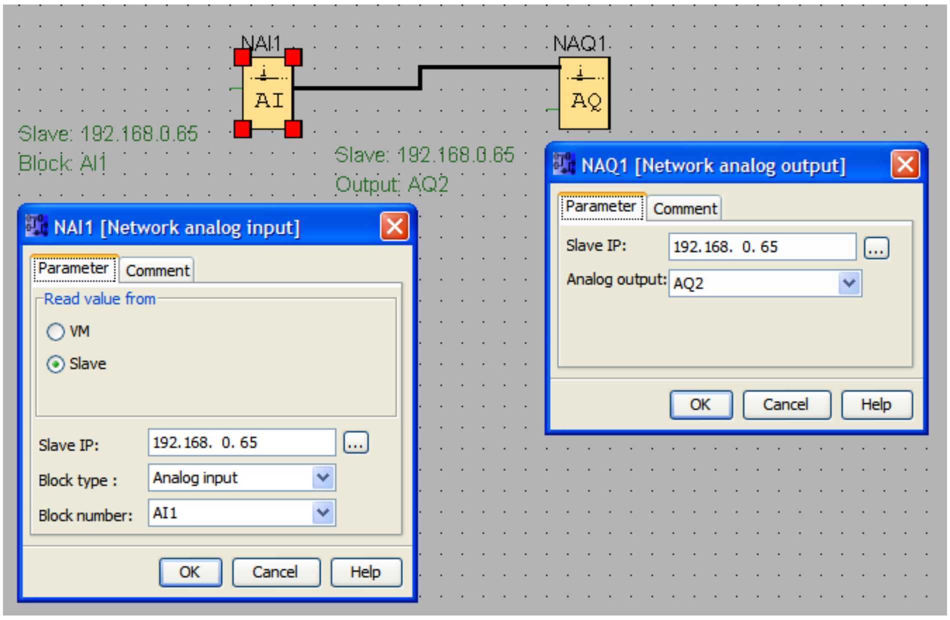
At this point you can run the Server Demo, and with the above configuration you should see something similar to this :



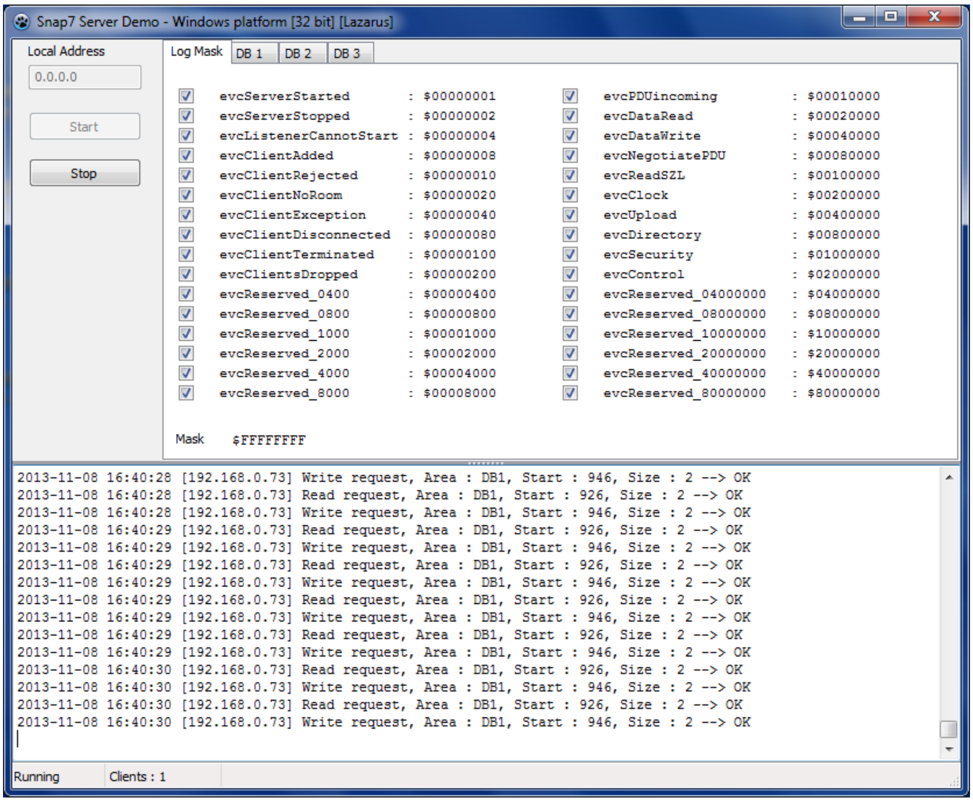
Four bytes of read request and one byte of write request, as we expected.

Network I/O Blocks

Open a Diagram and, as example, create a "loopback" between a network input and a network output.
Each network node requests a parametrization as in figure.
Here our LOGO sees Snap7Server as a LOGO Slave.



And this is what Snap7Server should show us:



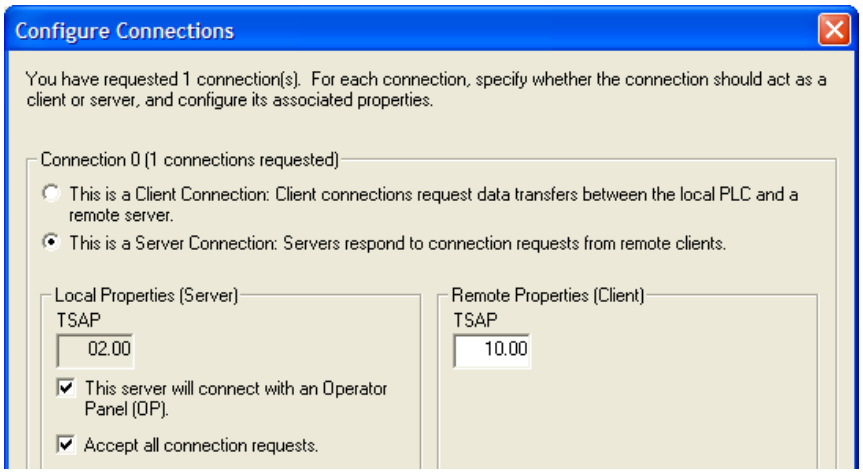
Two bytes of read request and two byte of write request, as we expected.

S7 200 (via CP243)

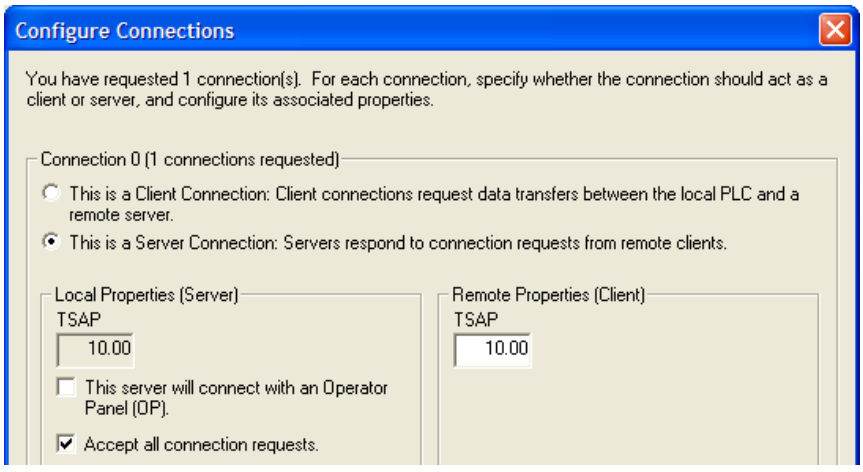
S7200 was out of the scope for Snap7 because beginning November 2013 the S7-200 product family will enter into the Phase Out stage of its product life cycle, but after working with LOGO also this PLC can be accessed since it uses the same connection mechanism.

Consider experimental the support for this PLC

As said, the connection is very similar to that of LOGO, you need to design a connection using the Ethernet wizard of MicroWin as in figure.



or



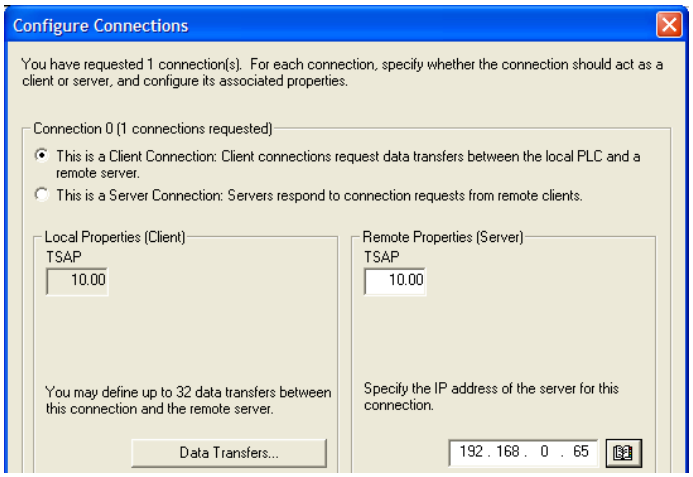
In the first case the PLC expects to be connected to an OP and you must supply LocalTSAP = **0x1000** and RemoteTSAP = **0x0200** to the SetConnectionParams function.

If you make a S7200 HMI project, the runtime of WinCC itself uses these parameters.

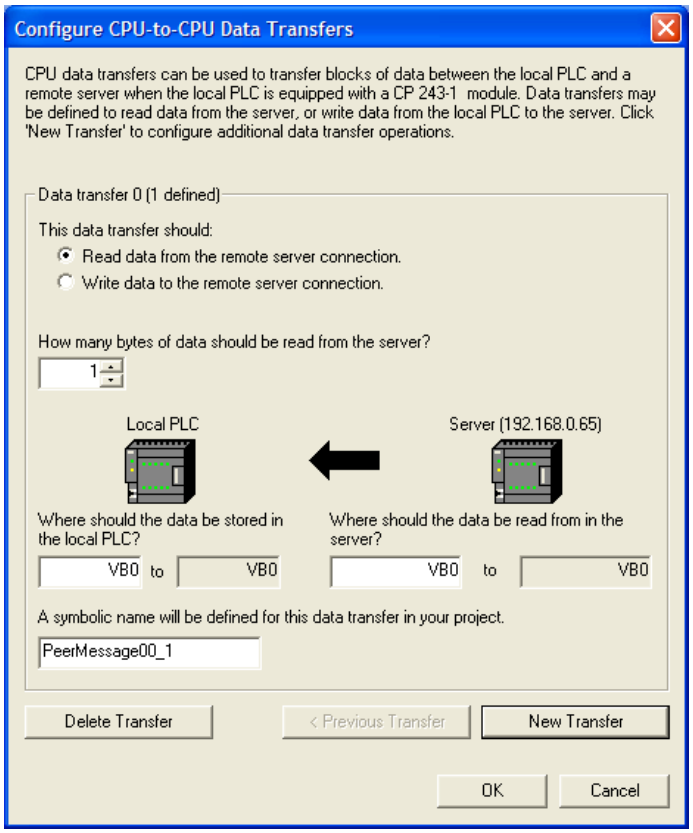
In the second case you should use LocalTSAP = **0x1000** and RemoteTSAP = **0x1000.**

Server side

To connect S7200 to a Snap7Server you need a Client connection PLC-side :



then, clicking "Data Transfers" button you can set the Exchange Data areas.



Don't care about TSAPs, Snap7Server accepts any values.

Snap7 Library API

API conventions

Snap7 exposes a simple and unified way to access to its objects.

Each object is created via the function **xxx_Create()** where xxx stands for Cli, Srv or Par.
This function returns a **Handle** that you must store and not modify.

Once the object is created, to use it you must call its working functions, passing them its handle.

MyServer=Srv_Create();
ReturnValue=Srv_<working function>(MyServer, <other params>);

The handle is an internal pointer (not a 24 bit descriptor), thus its size is either 32 or 64 bit, depending on the platform.

Provided wrappers define the type S7Object as native integer and handle it internally, I suggest to use them, or if you don't plan to use wrappers, copy its definition or use an untyped pointer.

At the end, we must destroy the object via the xxx_Destroy() function, passing the handle **by reference**.

Srv_Destroy(&MyServer).

xxx_Destroy() first checks that the handle value is not NULL, destroys the object, then sets to zero the handle. So, erroneous multiple call to xxx_destroy will not lead to access violation.

For each function, into the API Reference, **C** and **Pascal** prototype are present.
For **C#** and object-oriented functions please refer to wrappers interface files.

Wrappers

A **shared library** or **shared object** is a file that is intended to be shared by executable files and further shared objects files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.

A shared library has a well-defined interface by which the functions are invoked, which consists of *the* **function name** and the **calling convention** (stdcall, cdecl, etc..).

Beside the benefits of the shared libraries that make them a pillar of computing there are some drawbacks:

- **The library approach is fully procedural**. Full object-oriented languages (such as Java or C#) need a special interface code to interface them.

- Must be very careful about the **parameters type** and **calling convention**, especially if the library is meant to be used in a multi-architecture / multi-platform environment (32/64 bit – Windows/Unix).

In the libraries context, a **wrapper** is a piece of source code that works as glue between the user's source code and the binary library. And it should be considered part of the library (unmodifiable).

Wrappers supplied with Snap7 are **object-oriented**, they not only translate the syntax but give you a more comfortable way to work.

Example:

We want to read 16 byte from DB32 of an S7300 PLC whose address is "192.168.10.100".

We need to:

1. Create a Snap7Client.
2. Connect it to the PLC.
3. Read the DB.
4. Destroy the Client (the disconnection is automatic on destroy).

To do this, we simply include the wrapper in our source code and use the Client class as follow:

Pascal

```
Uses Snap7;

Var
  MyDB32 : packed array[0..255] of byte; // generic buffer
  MyClient : TS7Client;

Procedure SymplyGet;
Begin
  MyClient:=TS7Client.Create;
  MyClient.ConnectTo('192.168.10.100',0,2);
  MyClient.DBRead(32,       // DB Number
                   0,       // Start from
                  16,       // How many
                  @MyDB32); // Target address
  MyClient.Free;
End;
```

C#

```
Using Snap7;

  byte[] MyDB32 = new byte[256];
  static S7Client MyClient;

static void SymplyGet()
{

  MyClient = new S7Client();
  MyClient.ConnectTo("192.168.10.100",0,2);
  MyClient.DBRead(32, 0, 16, MyDB32);
  MyClient = null;
}
```

C++

```
#include "snap7.h";

  byte MyDB32[256]; // byte is a portable type of snap7.h
  TS7Client *Client;

void SymplyGet()
{

  MyClient = new TS7Client();
  MyClient->ConnectTo("192.168.10.100",0,2);
  MyClient->DBRead(32, 0, 16, &MyDB32);
  delete MyClient;
}
```

These are only a code snippets, functions return values should be checked…

If your preferred language is plain **C** you don't have objects, but the job is still very simply.

Here **MyClient** is not a class reference but it's a handle passed to the function.

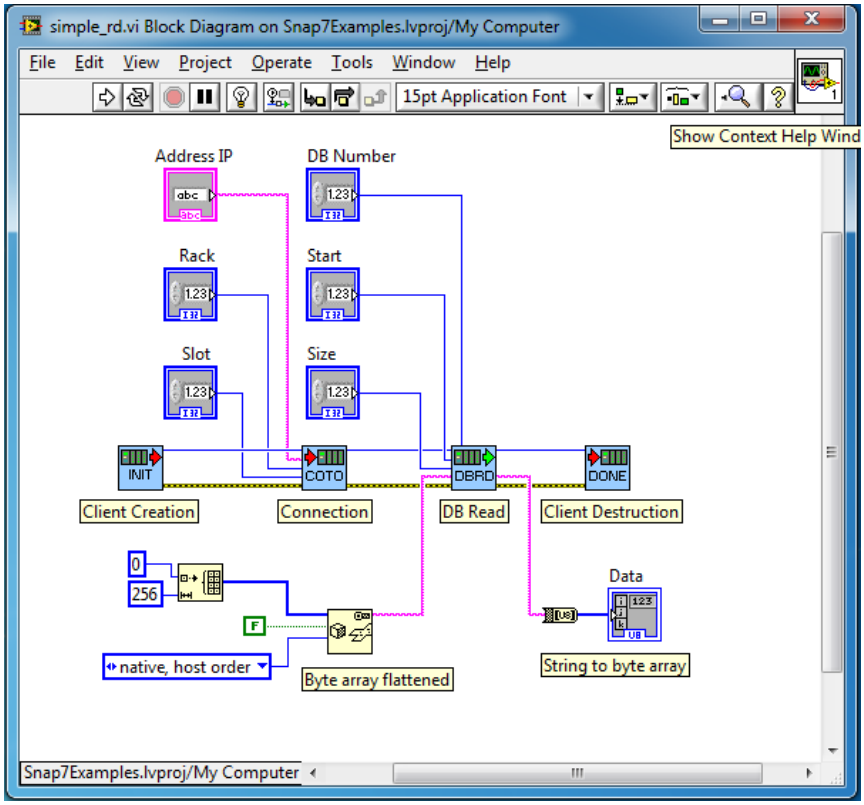
```
#include "snap7.h";

  byte MyDB32[256];
  S7Object Client;  // It's a native integer

void SymplyGet()
{

  MyClient = Cli_Create();
  Cli_ConnectTo(MyClient, "192.168.10.100",0,2);
  Cli_DBRead(MyClient, 32, 0, 16, &MyDB32);
  Cli_Destroy(&MyClient); // passed by ref
}
```

In the folder ***examples*** you will find many examples ready to run.

LabVIEW

LabVIEW is a special case, there is a specific chapter dedicated to it since many structural considerations must be done.

However this minimalist (but working) vi reads 16 bytes from a DB.



Accessing internal parameters

To allow a fine tuning of the behavior of Snap7 objects, is provided an expandable method to access their internal parameters.

xxx_GetParam() and **xxx_SetParam()** (where xxx is Cli, Srv or Par).
The first function allows you to read a parameter and the second to write it.

The declaration is:

```
int xxx_GetParam(S7Object TheObject, int ParamNumber, void *pValue);
int xxx_SetParam(S7Object TheObject, int ParamNumber, void *pValue);
```

The first parameter, as usual, is the object handle.

ParamNumber is an integer that identifies unequivocally a parameter (see the table below).

pValue is a pointer to the variable that contains (SetParam) or will receive (GetParam) the parameter value.

ParamNumbers are already defined in the wrappers provided for all languages.

	Value	CLI	SRV	PAR	
p_u16_LocalPort	1		O		Socket local Port.
p_u16_RemotePort	2	O		O	Socket remote Port.
p_i32_PingTimeout	3	O		O	Client Ping timeout.
p_i32_SendTimeout	4	O		O	Socket Send timeout.
p_i32_RecvTimeout	5	O		O	Socket Recv timeout.
p_i32_WorkInterval	6		O	O	Socket worker interval.
p_u16_SrcRef	7	O		O	ISOTcp Source reference.
p_u16_DstRef	8	O		O	ISOTcp Destination reference.
p_u16_SrcTSap	9	O		O	ISOTcp Source TSAP.
p_i32_PDURequest	10	O		O	Initial PDU length request.
p_i32_MaxClients	11		O		Max Clients allowed.
p_i32_BSendTimeout	12			O	BSend completion sequence timeout.
p_i32_BRecvTimeout	13			O	BRecv completion sequence timeout.
p_u32_RecoveryTime	14			O	Disconnection recovery time.
p_u32_KeepAliveTime	15			O	Time for (PLC) partner alive.

For further help, the parameter name contains info about the parameter type:

u16	pValue points to an unsigned 16 bit integer
i16	pValue points to a signed 16 bit integer
u32	pValue points to an unsigned 32 bit integer
i32	pValue points to a signed 32 bit integer
u64	pValue points to an unsigned 64 bit integer
i64	pValue points to a signed 64 bit integer

The parameters mean will be clear reading the functions reference.

Let's see some examples:

C++

```
…
// Sets the MyClient Ping time to 500 ms
int32_t PingTime=500;
Cli_SetParam(MyClient, p_i32_PingTimeout, &PingTime);

// Sets the MyServer max number of Clients to 128
int32_t MaxClients=128;
Srv_SetParam(MyServer, p_i32_MaxClients, &MaxClients);

// Gets the current MyPartner BRecv Timeout (ms)
int32_t BRecvTimeout;
Par_GetParam(MyPartner, p_i32_BRecvTimeout, &BRecvTimeout);
//<- Here BRecvTimeout contains the value
…
```

Pascal

```
…
// Sets the MyClient Ping time to 500 ms
Var
  PingTime : integer;
PingTime:=500;
Cli_SetParam(MyClient, p_i32_PingTimeout, @PingTime);

// Sets the MyServer max number of Clients to 128
Var
  MaxClients : integer;
MaxClients:=128;
Srv_SetParam(MyServer, p_i32_MaxClients, @MaxClients);

// Gets the current MyPartner BRecv Timeout (ms)
Var
  BRecvTimeout : integer;
Par_GetParam(MyPartner, p_i32_BRecvTimeout, @BRecvTimeout);
//<- Here BRecvTimeout contains the value
…
```

Some notes:

C# has no generic pointers, so to avoid the use of ***unsafe code,*** overloaded methods are used into the interface class.

Not all parameters have meaning for all the three objects, `p_i32_MaxClients`, for example, are server specific and are not recognized by a Client or a Partner.

Default values of these parameters are already fine-tuned.

These functions are meant for an advanced/experimental use, don't use them if "something seems to working bad" and in any case, look at the library source code to see how they operate.

Client API Reference

Administrative functions

These functions allow controlling the behavior a Client Object.

Function	Purpose
Cli_Create	Creates a Client Object.
Cli_Destroy	Destroys a Client Object.
Cli_ConnectTo	Connects a Client Object to a PLC.
Cli_SetConnectionType	Sets the connection type (PG/OP/S7Basic)
Cli_ConnectionParamst	Sets Address, Local and Remote TSAP for the connection.
Cli_Connect	Connects a Client Object to a PLC with implicit parameters.
Cli_Disconnect	Disconnects a Client.
Cli_GetParam	Reads an internal Client parameter.
Cli_SetParam	Writes an internal Client Parameter.

Cli_Create

Description

Creates a Client and returns its handle, which is the reference that you have to use every time you refer to that client.

The maximum number of clients that you can create depends only on the system memory amount

Declaration

```
S7Object Cli_Create();

function Cli_Create : S7Object;
```

Parameters

No parameters

Example

```
S7Object Client;     // Declaration

Client=Cli_Create(); // Creation
// Do something

Cli_Destroy(Client); // Destruction
```

Remarks

The handle is a memory pointer, so its size varies depending on the platform (32 or 64 bit). If you use the wrappers provided it is already declared as native integer, otherwise you can store it into a “pointer type” var.

Simply store it, it should not be changed ever.

Cli_Destroy

Description

Destroy a Client of given handle.
Before destruction the client is automatically disconnected if it was.

Declaration

```
void Cli_Destroy(S7Object *Client);

procedure Cli_Destroy(var Client : S7Object);
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by reference.

Example

```
S7Object Client;     // Declaration

Client=Cli_Create(); // Creation
// Do something

Cli_Destroy(Client); // Destruction
```

Remarks

The handle is passed by reference and it is set to NULL by the function. This allows you to call Cli_Destroy() more than once without worrying about memory exceptions.

Cli_SetConnectionType

Description

Sets the connection resource type, i.e the way in which the Clients connects to a PLC.

Declaration

```
int Cli_SetConnectionType(S7Object Client, word ConnectionType);

function Cli_SetConnectionType(Client : S7Object;
    ConnectionType : word) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
ConnectionType	Unsigned 16 bit integer.	In	See the table

Connection type table

Connection Type	Value
PG	0x01
OP	0x02
S7 Basic	0x03..0x10

Return value

- 0 : The Parameters was successfully written.
- `errLibInvalidObject` : The Client parameter was invalid.

Cli_ConnectTo

Description

Connects the client to the hardware at (IP, Rack, Slot) Coordinates.

Declaration

```
int Cli_ConnectTo(S7Object Client, const char *Address,
   int Rack, int Slot);

function Cli_ConnectTo(Client : S7Object; Address : PAnsiChar;
   Rack, Slot :  integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Address	Pointer to Ansi String	In	PLC/Equipment IPV4 Address ex. “192.168.1.12”
Rack	Integer	In	PLC Rack number (see below)
Slot	Integer	In	PLC Slot number (see below)

Return value

- 0 : The Client is successfully connected (or was already connected).
- Other values : see the Errors Code List.

Rack and Slot

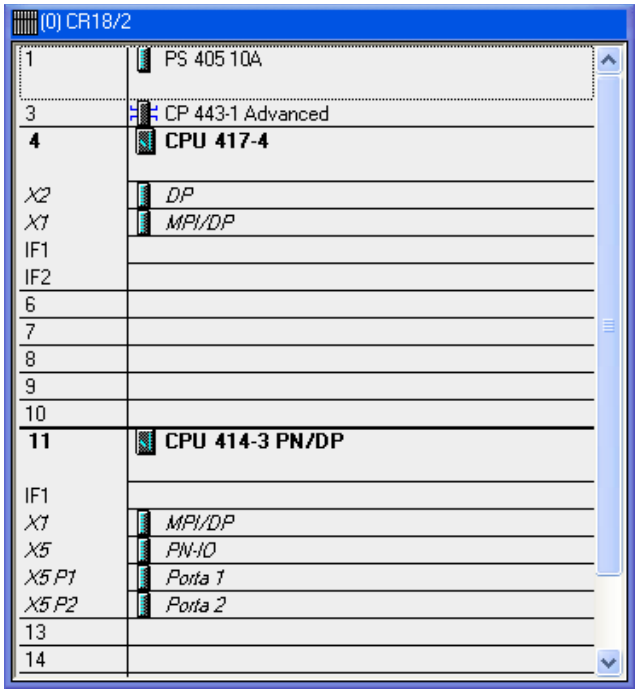
In addition to the IP Address, that we all understand, there are two other parameters that index the unit : **Rack** (0..7) and **Slot** (1..31) that you find into the hardware configuration of your project, for a physical component, or into the Station Configuration manager for WinAC.

There is however some special cases for which those values are fixed or can work with a default as you can see in the next table.

	Rack	Slot	
S7 300 CPU	0	2	Always
S7 400 CPU	Not fixed		Follow the hardware configuration.
WinAC CPU	Not fixed		Follow the hardware configuration.
S7 1200 CPU	0	0	Or 0, 1
S7 1500 CPU	0	0	Or 0, 1
CP 343	0	0	Or follow Hardware configuration.
CP 443	Not fixed		Follow the hardware configuration.
WinAC IE	0	0	Or follow Hardware configuration.

In the worst case, if you know the IP address, run **ClientDemo**, set 0 as Rack and try to connect with different values of Slot (1..31).

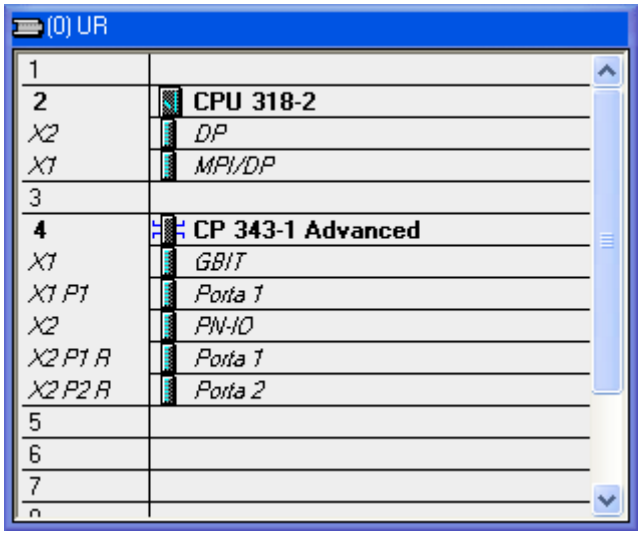
Let's see some examples of hardware configuration:



S7 400 Rack

	Rack	Slot
CPU 1	0	4
CPU 2	0	11
CP 443-1	0	3

The same concept for WinAC CPU which index can vary inside the PC Station Rack.



S7300 Rack

	Rack	Slot
CPU	0	2
CP 343-1	0	4

Cli_SetConnectionParams

Description

Sets internally (IP, LocalTSAP, RemoteTSAP) Coordinates.

Declaration

```
int Cli_SetConnectionParams(S7Object Client, const char *Address,
   word LocalTSAP, word RemoteTSAP);

function Cli_SetConnectionParams(Client : S7Object; Address : PAnsiChar;
   LocalTSAP, RemoteTSAP : word) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Address	Pointer to Ansi String	In	PLC/Equipment IPV4 Address ex. “192.168.1.12”
LocalTSAP	16 bit unsigned Integer	In	Local TSAP (PC TSAP)
RemoteTSAP	16 bit unsigned Integer	In	Remote TSAP (PLC TSAP)

Return value

- 0 : The Parameters were successfully written.
- `errLibInvalidObject` : The Client parameter was invalid.

Remarks

This function must be called just before **Cli_Connect().**

Cli_Connect

Description

Connects the client to the PLC with the parameters specified in the previous call of **Cli_ConnectTo()** or **Cli_SetConnectionParams().**

Declaration

```
int Cli_Connect(S7Object Client);

function Cli_Connect(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The Client is successfully connected (or was already connected).
- Other values : see the Errors Code List.

Remarks

This function can be called only after a previous of **`Cli_ConnectTo()`** or **`Cli_SetConnectionParams()`** which internally sets Address and TSAPs.

Cli_Disconnect

Description

Disconnects "gracefully" the Client from the PLC.

Declaration

```
int Cli_Disconnect(S7Object Client);

function Cli_Disconnect(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The Client is successfully disconnected (or was already disconnected).
- Other values : see the Errors Code List.

Remarks

If Client parameter is a valid handle, this function returns always 0, it can be called safely multiple times.
This function is called internally by **`Cli_Destroy()`** too.

Cli_GetParam

Description

Reads an internal Client object parameter.

Declaration

```
int Cli_GetParam(S7Object Client, int ParamNumber, void *pValue);

function Cli_GetParam(Client : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that will receive the parameter value.

Return value

- 0 : The parameter was successfully read.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Cli_SetParam

Description

Sets an internal Client object parameter.

Declaration

```
int Cli_SetParam(S7Object Client, int ParamNumber, void *pValue);

function Cli_SetParam(Client : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that contains the parameter value.

Return value

- 0 : The parameter was successfully set.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Data I/O functions

These functions allow the Client to exchange data with a PLC.

Function	Purpose
Cli_ReadArea	Reads a data area from a PLC.
Cli_WriteArea	Writes a data area into a PLC.
Cli_DBRead	Reads a part of a DB from a PLC.
Cli_DBWrite	Writes a part of a DB into a PLC.
Cli_ABRead	Reads a part of IPU area from a PLC.
Cli_ABWrite	Writes a part of IPU area into a PLC.
Cli_EBRead	Reads a part of IPI area from a PLC.
Cli_EBWrite	Writes a part of IPI area into a PLC.
Cli_MBRead	Reads a part of Merkers area from a PLC.
Cli_MBWrite	Writes a part of Merkers area into a PLC.
Cli_TMRead	Reads timers from a PLC.
Cli_TMWrite	Write timers into a PLC.
Cli_CTRead	Reads counters from a PLC.
Cli_CTWrite	Write counters into a PLC.
Cli_ReadMultiVars	Reads different kind of variables from a PLC simultaneously.
Cli_WriteMultiVars	Writes different kind of variables into a PLC simultaneously.

Cli_ReadArea

Description

This is the main function to read data from a PLC.
With it you can read DB, Inputs, Outputs, Merkers, Timers and Counters.

Declaration

```
int Cli_ReadArea(S7Object Client, int Area, int DBNumber, int Start,
    int Amount, int WordLen, void *pUsrData);

function Cli_ReadArea(Client : S7Object; Area, DBNumber, Start,
    Amount, WordLen : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	Mean
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Area	integer	In	Area identifier.
DBNumber	integer	In	DB Number if Area = S7AreaDB, otherwise is ignored.
Start	integer	In	Offset to start
Amount	integer	In	Amount of words to read (1)
Wordlen	integer	In	Word size
pUsrData	Pointer to memory area	In	Address of user buffer.

(1) Note the use of the parameter name "amount", it means quantity of words, not byte size.

Area table

	Value	Mean
S7AreaPE	0x81	Process Inputs.
S7AreaPA	0x82	Process Outputs.
S7AreaMK	0x83	Merkers.
S7AreaDB	0x84	DB
S7AreaCT	0x1C	Counters.
S7AreaTM	0x1D	Timers

WordLen table

	Value	Mean
S7WLBit	0x01	Bit (inside a word)
S7WLByte	0x02	Byte (8 bit)
S7WLWord	0x04	Word (16 bit)
S7WLDWord	0x06	Double Word (32 bit)
S7WLReal	0x08	Real (32 bit float)
S7WLCounter	0x1C	Counter (16 bit)
S7WLTimer	0x1D	Timer (16 bit)

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

As said, every data packet exchanged with a PLC must fit in a PDU, whose size is fixed and varies from 240 up to 960 bytes.

This function completely hides this concept, the data that you can transfer in a single call depends only on the size available of the data area (i.e. obviously, you cannot read 1024 bytes from a DB whose size is 300 bytes).

If this data size exceeds the PDU size, the packet is automatically split across more subsequent transfers.

If either **S7AreaCT** or **S7AreaTM** is selected, WordLen must be either **S7WLCounter** or **S7WLTimer** (However no error is raised and the values are internally fixed).

Your buffer should be large enough to receive the data.

Particularly:

Buffer size (byte) = Word size * Amount

Where:

	Word size
S7WLBit	1
S7WLByte	1
S7WLWord	2
S7WLDWord	4
S7WLReal	4
S7WLCounter	2
S7WLTimer	2

Notes

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsReadArea.

Cli_WriteArea

Description

This is the main function to write data into a PLC. It's the complementary function of Cli_ReadArea(), the parameters and their meanings are the same.

The only difference is that the data is transferred from the buffer pointed by pUsrData into PLC.

Declaration

```
int Cli_WriteArea(S7Object Client, int Area, int DBNumber, int Start,
    int Amount, int WordLen, void *pUsrData);

function Cli_WriteArea(Client : S7Object; Area, DBNumber, Start,
    Amount, WordLen : integer; pUsrData : pointer) : integer;
```

See **Cli_ReadArea()** for parameters and remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsWriteArea.

Cli_DBRead

Description

This is a lean function of Cli_ReadArea() to read PLC DB.

It simply internally calls Cli_ReadArea() with

- Area = S7AreaDB.
- WordLen = S7WLBytes.

Declaration

```
int Cli_DBRead(S7Object Client, int DBNumber, int Start, int Size,
    void *pUsrData);

function Cli_DBRead(Client : S7Object; DBNumber, Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DBNumber	integer	In	DB Index (0..0xFFFF).
Start	integer	In	Offset to start
Size	integer	In	Size to read (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsDBRead.

Cli_DBWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC DB.

It simply internally calls Cli_WriteArea() with

- Area = S7AreaDB.
- WordLen = S7WLBytes.

Declaration

```
int Cli_DBWrite(S7Object Client, int DBNumber, int Start, int Size,
    void *pUsrData);

function Cli_DBWrite(Client : S7Object; DBNumber, Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DBNumber	integer	In	DB Index (0..0xFFFF).
Start	integer	In	Offset to start
Size	integer	In	Size to Write (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsDBWrite.

Cli_ABRead

Description

This is a lean function of Cli_ReadArea() to read PLC process outputs .

It simply internally calls Cli_ReadArea() with

- Area = S7AreaPA.
- WordLen = S7WLBytes.

Declaration

```
int Cli_ABRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_ABRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to read (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsABRead.

Cli_ABWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC process outputs.

It simply internally calls Cli_WriteArea() with

- Area = S7AreaPA.
- WordLen = S7WLBytes.

Declaration

```
int Cli_ABWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_ABWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to Write (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsABWrite.

Cli_EBRead

Description

This is a lean function of Cli_ReadArea() to read PLC process inputs .

It simply internally calls Cli_ReadArea() with

- Area = S7AreaPE.
- WordLen = S7WLBytes.

Declaration

```
int Cli_EBRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_EBRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to read (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsEBRead.

Cli_EBWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC process inputs .

It simply internally calls Cli_WriteArea() with

- Area = S7AreaPE.
- WordLen = S7WLBytes.

Declaration

```
int Cli_EBWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_EBWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to Write (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsEBWrite.

Cli_MBRead

Description

This is a lean function of Cli_ReadArea() to read PLC Merkers .

It simply internally calls Cli_ReadArea() with

- Area = S7AreaMK.
- WordLen = S7WLBytes.

Declaration

```
int Cli_MBRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_MBRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to read (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsMBRead.

Cli_MBWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC Merkers.

It simply internally calls Cli_WriteArea() with

- Area = S7AreaMK.
- WordLen = S7WLBytes.

Declaration

```
int Cli_MBWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_MBWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Size	integer	In	Size to Write (bytes)
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsMBWrite.

Cli_TMRead

Description

This is a lean function of Cli_ReadArea() to read PLC Timers .

It simply internally calls Cli_ReadArea() with

- Area = S7AreaTM.
- WordLen = S7WLTimer.

Declaration

```
int Cli_TMRead(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_TMRead(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Amount	integer	In	Number of timers.
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsTMRead.

Buffer size (bytes) needed is Amount * 2.

Cli_TMWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC Timers .

It simply internally calls Cli_WriteArea() with

- Area = S7AreaTM.
- WordLen = S7WLTimer.

Declaration

```
int Cli_TMWrite(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_TMWrite(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Amount	integer	In	Number of timers.
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsTMWrite.

Buffer size (bytes) needed is Amount * 2.

Cli_CTRead

Description

This is a lean function of Cli_ReadArea() to read PLC Counters.

It simply internally calls Cli_ReadArea() with

- Area = S7AreaCT.
- WordLen = S7WLCounter.

Declaration

```
int Cli_CTRead(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_CTRead(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Amount	integer	In	Number of counters.
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_ReadArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsCTRead.

Buffer size (bytes) needed is Amount * 2.

Cli_CTWrite

Description

This is a lean function of Cli_WriteArea() to Write PLC Counters.

It simply internally calls Cli_WriteArea() with

- Area = S7AreaCT.
- WordLen = S7WLCounter.

Declaration

```
int Cli_CTWrite(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_CTWrite(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Start	integer	In	Offset to start
Amount	integer	In	Number of counters.
pUsrData	Pointer to memory area	In	Pointer user buffer.

See **Cli_WriteArea()** for remarks.

Remarks

If you need a large data transfer you may consider to use the asynchronous counterpart Cli_AsCTWrite.

Buffer size (bytes) needed is Amount * 2.

Cli_ReadMultiVars

Description

This is function allows to read different kind of variables from a PLC in a single call. With it you can read DB, Inputs, Outputs, Merkers, Timers and Counters.

Declaration

```
int Cli_ReadMultiVars(S7Object Client, PS7DataItem Item, int ItemsCount);

function Cli_ReadMultiVars(Client : S7Object;
    Items : PS7DataItems; ItemsCount : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Item	Pointer to struct.	In	Pointer to the first element of a TS7DataItem array.
ItemsCount	integer	In	Number of Items to read.

TS7DataItem struct

	Type	Dir.	
Area	integer 32	In	Area identifier.
Wordlen	integer 32	In	Word size
Result	integer 32	Out	Item operation result (2)
DBNumber	integer 32	In	DB Number if Area = S7AreaDB, otherwise is ignored.
Start	integer 32	In	Offset to start
Amount	integer 32	In	Amount of words to read (1)
pData	Pointer to memory area	In	Address of user buffer.

(1) Note the use of the parameter name "amount", it means quantity of words, not byte size.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

(2) Since could happen that some variables are read, some other not because maybe they don't exist in PLC. **Is important to check the single item Result**.

Remarks

Due the different kind of variables involved , there is no split feature available for this function, so **the maximum data size must not exceed the PDU size**. Thus there isn't an asynchronous counterpart of this function.

The advantage of this function becomes big when you have many small noncontiguous variables to be read.

Example

C++

```
void MultiRead()
{
      // Buffers
      byte MB[16]; // 16 Merker bytes
      byte EB[16]; // 16 Digital Input bytes
      byte AB[16]; // 16 Digital Output bytes
      word TM[8];  // 8 timers
      word CT[8];  // 8 counters

      // Prepare struct
      TS7DataItem Items[5];

      // Merkers
      Items[0].Area     =S7AreaMK;
      Items[0].WordLen  =S7WLByte;
      Items[0].DBNumber =0;         // Don't need DB
      Items[0].Start    =0;         // Starting from 0
      Items[0].Amount   =16;        // 16 Items (bytes)
      Items[0].pdata    =&MB;       // Address of buffer
      // Digital Input bytes
      Items[1].Area     =S7AreaPE;
      Items[1].WordLen  =S7WLByte;
      Items[1].DBNumber =0;         // Don't need DB
      Items[1].Start    =0;         // Starting from 0
      Items[1].Amount   =16;        // 16 Items (bytes)
      Items[1].pdata    =&EB;       // Address of buffer
      // Digital Output bytes
      Items[2].Area     =S7AreaPA;
      Items[2].WordLen  =S7WLByte;
      Items[2].DBNumber =0;         // Don't need DB
      Items[2].Start    =0;         // Starting from 0
      Items[2].Amount   =16;        // 16 Items (bytes)
      Items[2].pdata    =&AB;       // Address of buffer
      // Timers
      Items[3].Area     =S7AreaTM;
      Items[3].WordLen  =S7WLTimer;
      Items[3].DBNumber =0;         // Don't need DB
      Items[3].Start    =0;         // Starting from 0
      Items[3].Amount   =8;         // 8 Timers (16 bytes)
      Items[3].pdata    =&TM;       // Address of buffer
      // Counters
      Items[4].Area     =S7AreaCT;
      Items[4].WordLen  =S7WLCounter;
      Items[4].DBNumber =0;         // Don't need DB
      Items[4].Start    =0;         // Starting from 0
      Items[4].Amount   =8;         // 8 Counters (16 bytes)
      Items[4].pdata    =&CT;       // Address of buffer

      Client->ReadMultiVars(&Items[0],5);
}
```

Cli_WriteMultiVars

Description

This is function allows to write different kind of variables into a PLC in a single call. With it you can write DB, Inputs, Outputs, Merkers, Timers and Counters.

It's the complementary function of Cli_ReadMultiVars(), the parameters and their meanings are the same.

Declaration

```
int Cli_WriteMultiVars(S7Object Client, PS7DataItem Item, int
ItemsCount);

function Cli_WriteMultiVars(Client : S7Object;
    Items : PS7DataItems; ItemsCount : integer) : integer;
```

See **Cli_ReadMultiVars ()** for parameters and remarks.

Directory functions

These functions give you detailed information about the blocks present in a PLC.

Function	Purpose
Cli_ListBlocks	Returns the AG blocks amount divided by type.
Cli_ListBlocksOfType	Returns the AG blocks list of a given type.
Cli_GetAgBlockInfo	Returns detailed information about a block present in AG.
Cli_GetPgBlockInfo	Returns detailed information about a block loaded in memory.

Cli_ListBlocks

Description

This function returns the AG blocks amount divided by type.

Declaration

```
int Cli_ListBlocks(S7Object Client, TS7BlocksList *pUsrData);

function Cli_ListBlocks(Client : S7Object;
   pUsrData : PS7BlocksList) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct.	In	Pointer to TS7BlocksList struct.

TS7BlocksList struct

	Type	Dir.	
OBCount	integer 32	Out	OB amount in AG
FBCount	integer 32	Out	FB amount in AG
FCCount	integer 32	Out	FC amount in AG
SFBCount	integer 32	Out	SFB amount in AG
SFCCount	integer 32	Out	SFC amount in AG
DBCount	integer 32	Out	DB amount in AG
SDBCount	integer 32	Out	SDB amount in AG

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Example

See ListBlockOfTpe() Example

Cli_ListBlocksOfType

Description

This function returns the AG list of a specified block type.

Declaration

```
int Cli_ListBlocksofType (S7Object Client, int BlockType,
    TS7BlocksOfType *pUsrData, int *ItemsCount);

function Cli_ListBlocksOfType(Client : S7Object;
    BlockType : integer; pUsrData : PS7BlocksOfType;
    var ItemsCount : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockType	integer 32	In	Type of Block that we need
BlockNum	integer 32	In	Number of Block
pUsrData	Pointer	in	Address of the user buffer
ItemsCount	Pointer to integer 32	In	Buffer capacity
		Out	Number of items found

BlockType values

	Value	Type
Block_OB	0x38	OB
Block_DB	0x41	DB
Block_SDB	0x42	SDB
Block_FC	0x43	FC
Block_SFC	0x44	SFC
Block_FB	0x45	FB
Block_SFB	0x46	SFB

TS7BlocksOfType, by default, is defined as a packed array of 8192 16-bit word.

```
typedef word TS7BlocksOfType[0x2000];
```

8192 is the maximum number of blocks that a CPU S7417-4 can hold.

ItemsCount
In input indicates the user buffer capacity, in output how many items were found.
The function reads the list into the internal buffer, if **ItemsCount** is smaller than the data uploaded, only *ItemsCount* elements are copied into the user buffer and **errCliPartialDataRead** is returned.
The minimum expected value for ItemsCount is 1, otherwise **errCliInvalidBlockSize** error is returned.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

Each item of the user array will contain a block number.

Example extracted from *client.cs*, look at it for the missing functions **`Check()`**.

C#

```
static void ListBlocks()
{
    S7Client.S7BlocksList List = new S7Client.S7BlocksList();
    ushort[] DBList = new ushort[0x1000];
    int ItemsCount = DBList.Length;

    int res = Client.ListBlocks(ref List);
    if (Check(res, "List Blocks in AG"))
    {
        Console.WriteLine("  OBCount  : " + List.OBCount.ToString());
        Console.WriteLine("  FBCount  : " + List.FBCount.ToString());
        Console.WriteLine("  FCCount  : " + List.FCCount.ToString());
        Console.WriteLine("  SFBCount : " + List.SFBCount.ToString());
        Console.WriteLine("  SFCCount : " + List.SFCCount.ToString());
        Console.WriteLine("  DBCount  : " + List.DBCount.ToString());
        Console.WriteLine("  SDBCount : " + List.SDBCount.ToString());
    }
    else
        return;
    // List Blocks of Type (DB)
    res = Client.ListBlocksOfType(S7Client.Block_DB, DBList, ref ItemsCount);
    if (Check(res, "DB List in AG"))
    {
        if (ItemsCount > 0)
        {
            for (int i = 0; i < ItemsCount; i++)
                Console.WriteLine("  DB " + DBList[i].ToString());
        }
        else
            Console.WriteLine("NO DB found");
    }
}
```

Cli_GetAgBlockInfo

Description

Returns detailed information about an AG given block.

This function is very useful if you need to read or write data in a DB which you do not know the size in advance (see **MC7Size** field)

This function is used internally by Cli_DBGet().

Declaration

```
int Cli_GetAgBlockInfo(S7Object Client, int BlockType, int BlockNum,
    TS7BlockInfo *pUsrData);

function Cli_GetAgBlockInfo(Client : S7Object; BlockType, BlockNum :
integer;
    pUsrData : PS7BlockInfo) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockType	integer 32	In	Type of Block that we need
BlockNum	integer 32	In	Number of Block
pUsrData	Pointer to struct.	in	Pointer to TS7BlockInfo struct.

BlockType values

	Value	Type
Block_OB	0x38	OB
Block_DB	0x41	DB
Block_SDB	0x42	SDB
Block_FC	0x43	FC
Block_SFC	0x44	SFC
Block_FB	0x45	FB
Block_SFB	0x46	SFB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

TS7BlockInfo struct

```
typedef struct {
    int BlkType;       // Block Type (see SubBlkType table)
    int BlkNumber;     // Block number
    int BlkLang;       // Block Language (see LangType Table)
    int BlkFlags;      // Block flags (bitmapped)
    int MC7Size;       // The real size in bytes
    int LoadSize;      // Load memory size
    int LocalData;     // Local data
    int SBBLength;     // SBB Length
    int CheckSum;      // Checksum
    int Version;       // Version (BCD 00<HI><LO>)
    char CodeDate[11]; // Code date
    char IntfDate[11]; // Interface date
    char Author[9];    // Author
    char Family[9];    // Family
    char Header[9];    // Header
} TS7BlockInfo, *PS7BlockInfo ;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

This struct is filled by the function, some fields require additional info:

SubBlockType table

	Value	Type
SubBlk_OB	0x08	OB
SubBlk_DB	0x0A	DB
SubBlk_SDB	0x0B	SDB
SubBlk_FC	0x0C	FC
SubBlk_SFC	0x0D	SFC
SubBlk_FB	0x0E	FB
SubBlk_SFB	0x0F	SFB

LangType table

	Value	Block Language
BlockLangAWL	0x01	AWL
BlockLangKOP	0x02	KOP
BlockLangFUP	0x03	FUP
BlockLangSCL	0x04	SCL
BlockLangDB	0x05	DB
BlockLangGRAPH	0x06	GRAPH

For an exhaustive example see ClientDemo (rich demos package).

Cli_GetPgBlockInfo

Description

Returns detailed information about a block present in a user buffer.
This function is usually used in conjunction with **Cli_FullUpload()**.

An uploaded a block saved to disk, could be loaded in a user buffer and checked with this function.

Declaration

```
int Cli_GetPgBlockInfo(S7Object Client, void *pBlock,
    TS7BlockInfo *pUsrData, int size);

function Cli_GetPgBlockInfo(Client : S7Object; pBlock : pointer;
    pUsrData : PS7BlockInfo; size integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pBlock	Pointer	In	Address of the user buffer that contains the block.
pUsrData	Pointer to struct.	in	Pointer to TS7BlockInfo struct.
size	integer 32	in	Size (bytes) of user buffer.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

For a detailed description of TS7BlockInfo see **Cli_GetAgBlockInfo**

With this function in conjunction with block oriented functions it's possible to create a "backup manager".

The rich demo ClientDemo shows how to upload/download/delete and get detailed block information.

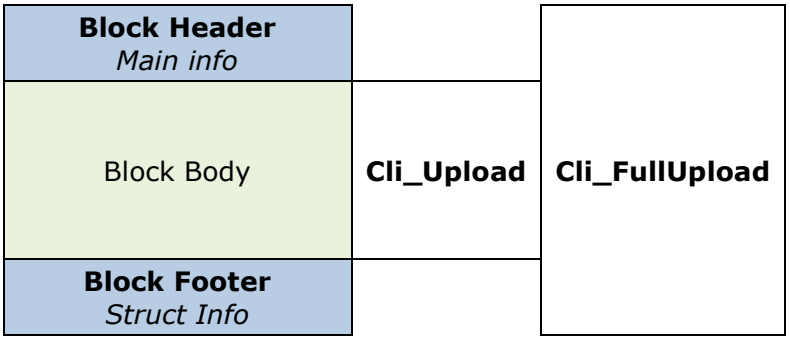
Block oriented functions

These functions allow you to move blocks from/to the PLC and delete them.

AG Block Structure

A block (OB, FB, DB , etc.) in AG consists of:

- A **Header** containing main block info such as MC7Size, date and time etc.
- A **Body** containing the data.
- A **Footer** containing info about data struct such as number and type of elements of a DB. Additional block info such as author, family etc...



With these functions we can upload a block from AG in two ways : **fully** or **data only** depending on our needed, using `Cli_FullUpload()` or `Cli_Upload(). (1)`

For the blocks downloading, however, there are some limitations:

1. Only **full blocks** can be downloaded into AG via `Cli_Download()`.
2. A modified block, i.e. a block to which we have made changes to the body area, could be refused by the CPU.

Upload, download and deletion are subject to the security level set.
See Security functions for more info.

(1) S7 Protocol itself, only provides full upload, Snap7Client internally extracts the data body into Cli_Upload.

Function	Purpose
Cli_FullUpload	Uploads a block from AG with Header and Footer infos.
Cli_Upload	Uploads a block from AG.
Cli_Download	Download a block into AG.
Cli_Delete	Delete a block into AG.
Cli_DBGet	Uploads a DB from AG using DBRead.
Cli_DBFill	Fills a DB in AG with a given byte.

Cli_FullUpload

Description

Uploads a block from AG.
The whole block (including header and footer) is copied into the user buffer.

Declaration

```
int Cli_FullUpload(S7Object Client, int BlockType, int BlockNum,
    void *pUsrData, int *Size);

function Cli_FullUpload(Client : S7Object; BlockType, BlockNum : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockType	integer 32	In	Type of Block that we need
BlockNum	integer 32	In	Number of Block
pUsrData	Pointer	in	Address of the user buffer
size	Pointer to integer 32	In	Buffer size available
		Out	Bytes uploaded

BlockType values

	Value	Type
Block_OB	0x38	OB
Block_DB	0x41	DB
Block_SDB	0x42	SDB
Block_FC	0x43	FC
Block_SFC	0x44	SFC
Block_FB	0x45	FB
Block_SFB	0x46	SFB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The function is performed into the internal buffer, if **size** is smaller than the data uploaded, only *size* bytes are copied and **errCliPartialDataRead** is returned.

Example extracted from client.cpp (see)

```
void UploadSDB0()
{
   byte Buffer[4096]; // 4 K buffer
   int Size = sizeof(Buffer);

   int res=Client->Upload(Block_SDB, 0, &Buffer, &Size);
   if (Check(res,"Block Upload (SDB 0)"))
   {
        printf("Dump (%d bytes) :\n",Size);
        hexdump(&Buffer,Size);
   }
}
```

Cli_Upload

Description

Uploads a block body from AG.
Only the block body (but header and footer) is copied into the user buffer.

Declaration

```
int Cli_Upload(S7Object Client, int BlockType, int BlockNum,
    void *pUsrData, int *Size);

function Cli_Upload(Client : S7Object; BlockType, BlockNum : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockType	integer 32	In	Type of Block that we need
BlockNum	integer 32	In	Number of Block
pUsrData	Pointer	in	Address of the user buffer
size	Pointer to integer 32	In	Buffer size available
		Out	Bytes uploaded

BlockType values

	Value	Type
Block_OB	0x38	OB
Block_DB	0x41	DB
Block_SDB	0x42	SDB
Block_FC	0x43	FC
Block_SFC	0x44	SFC
Block_FB	0x45	FB
Block_SFB	0x46	SFB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The function reads the block data into the internal buffer, if **size** is smaller than the data uploaded, only *size* bytes are copied into the user buffer and **errCliPartialDataRead** is returned.

See Cli_FullUpload Example.

Cli_Download

Description

Downloads a block into AG.
A whole block (including header and footer) must be available into the user buffer.

Declaration

```
int Cli_Download(S7Object Client, int BlockNum, void *pUsrData, int *Size);

function Cli_Download(Client : S7Object; BlockNum : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockNum	integer 32	In	New Block number (or -1)
pUsrData	Pointer	in	Address of the user buffer
size	Integer 32	In	Buffer size

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

A block ready to be downloaded already contains info about block type and block number.
If the parameter BlockNum is -1, the block number is not changed used else the block is downloaded with a different number (just like a “Download As…”).

Cli_Delete

Description

Deletes a block into AG.

Warning

There is no undo function available.

Declaration

```
int Cli_Delete(S7Object Client, int BlockType, int BlockNum);

function Cli_Delete(Client : S7Object; BlockType,
    BlockNum : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
BlockType	integer 32	In	Type of Block to delete
BlockNum	integer 32	In	Number of Block to delete

BlockType values

	Value	Type
Block_OB	0x38	OB
Block_DB	0x41	DB
Block_SDB	0x42	SDB
Block_FC	0x43	FC
Block_SFC	0x44	SFC
Block_FB	0x45	FB
Block_SFB	0x46	SFB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Cli_DBGet

Description

Uploads a DB from AG.

This function is equivalent to Cli_Upload() with BlockType = Block_DB but it uses a different approach so **it's not subject to the security level set**.

Only data is uploaded.

Declaration

```
int Cli_DBGet(S7Object Client, int DBNumber, void *pUsrData, int *Size);

function Cli_DBGet(Client : S7Object; DBNumber : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DBNumber	integer 32	In	DB Number
pUsrData	Pointer	in	Address of the user buffer
size	Pointer to integer 32	In	Buffer size available
		Out	Bytes uploaded

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This function first gathers the DB size via Cli_GetAgBlockInfo then calls Cli_DBRead.

Cli_DBFill

Description

Fills a DB in AG with a given byte without the need of specifying its size.

Declaration

```
int Cli_DBFill(S7Object Client, int DBNumber, int FillChar);

function Cli_DBFill(Client : S7Object; DBNumber : integer;
    FillChar : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DBNumber	integer 32	In	DB Number
FillChar	Integer 32	in	Byte pattern

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

Fillchar is an integer for efficiency reasons, only the lowest byte is used.

Date/Time functions

These functions allow to read/modify the date and time of a PLC.

Imagine a production line in which each PLC saves the data with date/time field inside, it is very important that the date be up to date.

Both CP X43 and internal PN allow to synchronize date and time but you need an NTP server, and in some cases (old hardware or CP343-1 Lean or old firmware release) this doesn't work properly.

Snap7 Client, using the same method of S7 Manager, always works.

Function	Purpose
Cli_GetPlcDateTime	Returns the PLC date/time.
Cli_SetPlcDateTime	Sets the PLC date/time with a given value.
Cli_SetPlcSystemDateTime	Sets the PLC date/time with the host (PC) date/time.

Cli_GetPlcDateTime

Description

Reads PLC date and time.

Declaration

```
int Cli_GetPlcDateTime(S7Object Client, tm *DateTime);

function Cli_GetPlcDateTime(Client : S7Object;
    var DateTime : TCPP_tm) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DateTime	Pointer to struct	In	Address of C++ tm struct.

Struct tm is C++ specific, if you use the wrappers provided, you don't need to care about it, since *tm* is internally converted to the native date/time format.

Pascal

```
function TS7Client.GetPlcDateTime(Var DateTime : TDateTime) : integer;
```

C#

```
public int GetPlcDateTime(ref DateTime DT);
```

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Cli_SetPlcDateTime

Description

Sets the PLC date and time.

Declaration

```
int Cli_SetPlcDateTime(S7Object Client, tm *DateTime);

function Cli_SetPlcDateTime(Client : S7Object;
    var DateTime : TCPP_tm) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
DateTime	Pointer to struct	In	Address of C++ tm struct.

Struct tm is C++ specific, if you use the wrappers provided, you don't need to care about it, since *tm* is internally converted to the native date/time format.

Pascal

```
function TS7Client.SetPlcDateTime(Var DateTime : TDateTime) : integer;
```

C#

```
public int SetPlcDateTime(ref DateTime DT);
```

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Cli_SetPlcSystemDateTime

Description

Sets the PLC date and time in accord to the PC system Date/Time.

Declaration

```
int Cli_SetPlcSystemDateTime(S7Object Client);

function Cli_SetPlcSystemDateTime(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

System info functions

these functions access to **SZL** (or **SSL** - System Status List) to give you all the same information that you can get from S7 Manager.

System Status List

The system status list (SSL) describes the current status of a programmable logic controller.

The contents of the SSL can only be read using information functions but cannot be modified. The partial lists are virtual lists, in other words, they are only created by the operating system of the CPUs when specifically requested.

You can access to system status list using **SFC 51** too "RDSYSST."

To read a partial list you must specify its **ID** and **Index**.

For a detailed description of SZL see:

§33 of **"System Software for S7-300/400 System and Standard Functions".**

Function	Purpose
Cli_ReadSZL	Reads a partial list of given ID and Index.
Cli_ReadSZLList	Reads the list of partial lists available in the CPU.
Cli_GetOrderCode	Returns the CPU order code.
Cli_GetCpuInfo	Returns some information about the AG.
Cli_GetCpInfo	Returns some information about the CP (communication processor).

Cli_ReadSZL

Description

Reads a partial list of given **ID** and **INDEX**.

Declaration

```
int Cli_ReadSZL(S7Object Client, int ID, int Index,
    TS7SZL *pUsrData, int *Size);

function Cli_ReadSZL(Client : S7Object; ID, Index : integer;
    pUsrData : PS7SZL; var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
ID	integer 32	In	List ID
Index	integer 32	In	List Index
pUsrData	Pointer to struct	in	Address of the user buffer
size	Pointer to integer 32	In	Buffer size available
		Out	Bytes read

TS7SZL struct

```
// See §33.1 of "System Software for S7-300/400 System and
// Standard Functions"
// and see SFC51 description too

typedef struct {
   word LENTHDR;
   word N_DR;
} SZL_HEADER, *PSZL_HEADER;

typedef struct {
   SZL_HEADER Header;
   byte Data[0x4000-4];
} TS7SZL, *PS7SZL;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

	Type	Dir.	Mean
LENTHDR	unsigned Integer 16	Out	Length of a data record of the partial list in bytes
N_DR	unsigned Integer 16	Out	Number of data records contained in the partial list.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The function is performed into the internal buffer, if **size** is smaller than the data uploaded, only *size* bytes are copied and **`errCliPartialDataRead`** is returned.

Remarks

LENTHDR and N_DR are HI-LOW order swapped, the data buffer is unchanged.

Cli_ReadSZLList

Description

Reads the directory of the partial lists.

Declaration

```
int Cli_ReadSZLList(S7Object Client, TS7SZLList *pUsrData,
    int *ItemsCount);

function Cli_ReadSZLList(Client : S7Object; pUsrData : PS7SZLList;
    var ItemsCount : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct	in	Address of the user buffer list
ItemsCount	Pointer to integer 32	In	Buffer capacity
		Out	Number of items found

TS7SZLList struct

```
// See §33.1 of "System Software for S7-300/400 System and
// Standard Functions"
// and see SFC51 description too

typedef struct {
   word LENTHDR;
   word N_DR;
} SZL_HEADER, *PSZL_HEADER;

typedef struct {
   SZL_HEADER Header;
   word List[0x2000-2]; // HI-LO Swapped
} TS7SZLList, *PS7SZLList;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

	Type	Dir.	
LENTHDR	unsigned Integer 16	Out	Length of a data record of the partial list in bytes
N_DR	unsigned Integer 16	Out	Number of data records contained in the partial list.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

Not all **ID** address a valid partial list, use this function to know what are.

ItemsCount
In input indicates the user buffer capacity, in output how many items were found.

The function reads the list into the internal buffer, if **ItemsCount** is smaller than the data uploaded, only *ItemsCount* elements are copied into the user buffer and **`errCliPartialDataRead`** is returned.

LENTHDR, N_DR and List HI-LOW order swapped.

Cli_GetOrderCode

Description

Gets CPU order code and version info.

Declaration

```
int Cli_GetOrderCode(S7Object Client, TS7OrderCode *pUsrData);

function Cli_GetOrderCode(Client : S7Object;
    pUsrData : PS7OrderCode) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct	in	Address of the user Order code buffer

TS7OrderCode struct

```
typedef struct {
   char Code[21]; // Order Code
   byte V1;       // Version V1.V2.V3
   byte V2;
   byte V3;
} TS7OrderCode, *PS7OrderCode;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Example extracted from client.cpp (see)

```
void OrderCode()
{
     TS7OrderCode Info;
     int res=Client->GetOrderCode(&Info);
     if (Check(res,"Catalog"))
     {
        printf("  Order Code : %s\n",Info.Code);
        printf("  Version    : %d.%d.%d\n",
          Info.V1,Info.V2,Info.V3);
     };
}
```

Cli_GetCpuInfo

Description

Gets CPU module name, serial number and other info.

Declaration

```
int Cli_GetCpuInfo(S7Object Client, TS7CpuInfo *pUsrData);

function Cli_GetCpuInfo(Client : S7Object;
    pUsrData : PS7CpuInfo) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct	in	Address of the user CPU info buffer

TS7CpuInfo struct

```
typedef struct {
   char ModuleTypeName[33];
   char SerialNumber[25];
   char ASName[25];
   char Copyright[27];
   char ModuleName[25];
} TS7CpuInfo, *PS7CpuInfo;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Example extracted from client.cpp (see)

```
void CpuInfo()
{
     TS7CpuInfo Info;
     int res=Client->GetCpuInfo(&Info);
     if (Check(res,"Unit Info"))
     {
          printf("  Module Type Name : %s\n",Info.ModuleTypeName);
          printf("  Serial Number    : %s\n",Info.SerialNumber);
          printf("  AS Name          : %s\n",Info.ASName);
          printf("  Module Name      : %s\n",Info.ModuleName);
     };
}
```

Cli_GetCpInfo

Description

Gets CP (communication processor) info.

Declaration

```
int Cli_GetCpInfo(S7Object Client, TS7CpInfo *pUsrData);

function Cli_GetCpInfo(Client : S7Object;
    pUsrData : PS7CpInfo) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct	in	Address of the user CP info buffer

TS7CpInfo struct

```
typedef struct {
   int MaxPduLengt;
   int MaxConnections;
   int MaxMpiRate;
   int MaxBusRate;
} TS7CpInfo, *PS7CpInfo;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Example extracted from client.cpp (see)

```
void CpInfo()
{
    TS7CpInfo Info;
     int res=Client->GetCpInfo(&Info);
     if (Check(res,"Communication processor Info"))
     {
          printf("  Max PDU Length    : %d bytes\n",Info.MaxPduLengt);
          printf("  Max Connections   : %d \n",Info.MaxConnections);
          printf("  Max MPI Rate      : %d bps\n",Info.MaxMpiRate);
          printf("  Max Bus Rate      : %d bps\n",Info.MaxBusRate);
     };
}
```

PLC control functions

With these control function it's possible to Start/Stop a CPU and perform some other maintenance tasks.

Function	Purpose
Cli_PlcHotStart	Puts the CPU in RUN mode performing an HOT START.
Cli_PlcColdStart	Puts the CPU in RUN mode performing a COLD START.
Cli_PlcStop	Puts the CPU in STOP mode.
Cli_CopyRamToRom	Performs the Copy Ram to Rom action.
Cli_Compress	Performs the Compress action.
Cli_GetPlcStatus	Returns the CPU status (running/stopped).

Cli_PlcHotStart

Description

Puts the CPU in RUN mode performing an HOT START.

Declaration

```
int Cli_PlcHotStart(S7Object Client);

function Cli_PlcHotStart(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This function is subject to the security level set.

Cli_PlcColdStart

Description

Puts the CPU in RUN mode performing a COLD START.

Declaration

```
int Cli_PlcColdStart(S7Object Client);

function Cli_PlcColdStart(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This function is subject to the security level set.

Cli_PlcStop

Description

Puts the CPU in STOP mode.

Declaration

```
int Cli_PlcStop(S7Object Client);

function Cli_PlcStop(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This function is subject to the security level set.

Cli_CopyRamToRom

Description

Performs the Copy Ram to Rom action.

Declaration

```
int Cli_CopyRamToRom(S7Object Client, int Timeout);

function Cli_CopyRamToRom(Client : S7Object; Timeout : integer) :
integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Timeout	Integer 32	In	Maximum time expected to complete the operation (ms).

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

Not all CPUs support this operation.

The CPU must be in STOP mode.

Cli_Compress

Description

Performs the Memory compress action.

Declaration

```
int Cli_Compress(S7Object Client, int Timeout);

function Cli_Compress(Client : S7Object; Timeout : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Timeout	Integer 32	In	Maximum time expected to complete the operation (ms).

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

Not all CPUs support this operation.

The CPU must be in STOP mode.

Cli_GetPlcStatus

Description

Returns the CPU status (running/stopped).

Declaration

```
int Cli_GetPlcStatus(S7Object Client, int *Status);

function Cli_GetPlcStatus(Client : S7Object;
    Var Status : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Status	Pointer to Integer 32	In	Address of Status variable.

Status values

	Value	
S7CpuStatusUnknown	0x00	The CPU status is unknown.
S7CpuStatusRun	0x08	The CPU is running.
S7CpuStatusStop	0x04	The CPU is stopped.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Security functions

With these functions is possible to know the current protection level, and to set/clear the current session password.

The correct name of the below functions Cli_SetSessionPassword and Cli_ClearSessionPassword, would have to be **Cli_Login** and **Cli_Logout** to avoid misunderstandings about their scope.

Especially because, if you look at the source code, there is an encoding function that translates the plain password before send it to the PLC.

PASSWORD HACKING IS VERY FAR FROM THE AIM OF THIS PROJECT, MOREOVER YOU NEED TO KNOW THE CORRECT PASSWORD TO MEET THE CPU SECURITY LEVEL.

Detailed information about the protection level can be found in §33.19 of **"System Software for S7-300/400 System and Standard Functions".**

Function	Purpose
Cli_SetSessionPassword	Send the password to the PLC to meet its security level.
Cli_ClearSessionPassword	Clears the password set for the current session (logout).
Cli_GetProtection	Gets the CPU protection level info.

Cli_SetSessionPassword

Description

Send the password to the PLC to meet its security level.

Declaration

```
int Cli_SetSessionPassword(S7Object Client, char *Password);

function Cli_SetSessionPassword(Client : S7Object;
   Password : PAnsiChar) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Password	Pointer to Ansi String	In	8 chars string

Return value

- 0 : The Client is successfully connected (or was already connected).
- Other values : see the Errors Code List.

Remarks

A password accepted by a PLC is an 8 chars string, a greater password will be trimmed, and a smaller one will be "right space padded".

Cli_ClearSessionPassword

Description

Clears the password set for the current session (logout).

Declaration

```
int Cli_ClearSessionPassword(S7Object Client);

function Cli_ClearSessionPassword(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.

Return value

- 0 : The Client is successfully connected (or was already connected).
- Other values : see the Errors Code List.

Cli_GetProtection

Description

Gets the CPU protection level info.

Declaration

```
int Cli_GetProtection(S7Object Client, TS7Protection *pUsrData);

function Cli_GeProtection(Client : S7Object;
    pUsrData : PS7Protection) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer to struct	in	Address of the user Protection info buffer

TS7Protection struct

```
typedef struct {
   word  sch_schal;
   word  sch_par;
   word  sch rel;
   word  bart_sch;
   word  anl_sch;
} TS7Protection, *PS7Protection;
```

for Pascal and C# definition see **snap7.pas** and **snap7.net.cs**

S7Protection Values

	Values	
sch_schal	1, 2, 3	Protection level set with the mode selector.
sch_par	0, 1, 2, 3	Password level, 0 : no password
sch_rel	0, 1, 2, 3	Valid protection level of the CPU
bart_sch	1, 2, 3, 4	Mode selector setting (1:RUN, 2:RUN-P, 3:STOP, 4:MRES, 0:undefined or cannot be determined)
anl_sch;	0, 1, 2	Startup switch setting (1:CRST, 2:WRST, 0:undefined, does not exist of cannot be determined)

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Low level functions

Snap7 hides the IsoTCP underlying layer. With this function however, it's possible to exchange an IsoTCP telegram with a PLC.

Function	Purpose
Cli_IsoExchangeBuffer	Exchanges a given S7 PDU (protocol data unit) with the CPU.

Cli_IsoExchangeBuffer

Description

Exchanges a given S7 PDU (protocol data unit) with the CPU.

Declaration

```
int Cli_IsoExchangeBuffer(S7Object Client, void *pUsrData, int *Size);

function Cli_IsoExchangeBuffer (Client : S7Object; pUsrData : pointer,
   var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pUsrData	Pointer	In	Address of the user buffer.
Size	Pointer to integer 32	In	Buffer size available
		Out	Reply telegram size

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

The S7 PDU supplied is encapsulated into an IsoTCP telegram then is sent to the CPU.

Finally, the S7 PDU is extracted from the reply telegram and is copied into the user buffer.

Look at **S7_types.h** for more info about S7 PDU.

No size check is performed : use a large enough buffer.

Miscellaneous functions

These are utility functions.

Function	Purpose
Cli_GetExecTime	Returns the last job execution time in milliseconds.
Cli_GetLastError	Returns the last job result.
Cli_GetPduLength	Returns info about the PDU length (requested and negotiated).
Cli_ErrorText	Returns a textual explanation of a given error number.
Cli_GetConnected	Returns the connection status of the client.

Cli_GetExecTime

Description

Returns the last job execution time in milliseconds.

Declaration

```
int Cli_GetExecTime(S7Object Client, int *Time);

function Cli_GetExecTime(Client : S7Object;
   var Time : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Time	Pointer to integer 32	In	Address of the time variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Cli_GetLastError

Description

Returns the last job result.

Declaration

```
int Cli_GetLastError(S7Object Client, int *LastError);

function Cli_GetLastError(Client : S7Object;
   var LastError : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
LastError	Pointer to integer 32	In	Address of the LastError variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Cli_GetPduLength

Description

Returns info about the PDU length.

Declaration

```
int Cli_GetPduLength(S7Object Client, int *Requested, int *Negotiated);

function Cli_GetPduLength(Client : S7Object;
   var Requested, Negotiated : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Requested	Pointer to integer 32	In	Address of the PDU Req. variable
Negotiated	Pointer to integer 32	In	Address of the PDU Neg. variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

During the S7 connection Client and Server (the PLC) negotiate the PDU length.

PDU requested can be modified with **Cli_SetParam().**

It's useful to know the PDU negotiated when we need to call **Cli_ReadMultivar()** or **Cli_WriteMultiVar().**
All other data transfer functions handle by themselves this information and split automatically the telegrams if needed.

Cli_ErrorText

Description

Returns a textual explanation of a given error number.

Declaration

```
int Cli_ErrorText(int Error, char *Text, int TextLen);

function Cli_ErrorText(Error : integer, Text : PAnsiChar;
   TextLen :  integer) : integer;
```

Parameters

	Type	Dir.	
Error	Integer 32	In	Error code
Text	Pointer to Ansi String	In	Address of the char array
TextLen	Integer 32	In	Size of the char array

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This is a translation function, so there is no need of a client handle.

The messages are in (internet) English, all they are in s7_text.cpp.

Cli_GetConnected

Description

Returns the connection status

Declaration

```
int Cli_GetConnected(S7Object Client, int *IsConnected);

function Cli_GetConnected(Client : S7Object;
   var IsConnected : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
IsConnected	Pointer to integer 32	In	Address of the IsConnected variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

IsConnected is 0 if the client is not connected otherwise contains an integer !=0.

Asynchronous functions

These functions are executed in a separate thread simultaneously to the execution of the caller program.

Function	Purpose
Cli_AsReadArea	Reads a data area from a PLC.
Cli_AsWriteArea	Writes a data area into a PLC.
Cli_AsDBRead	Reads a part of a DB from a PLC.
Cli_AsDBWrite	Writes a part of a DB into a PLC.
Cli_AsABRead	Reads a part of IPU area from a PLC.
Cli_AsABWrite	Writes a part of IPU area into a PLC.
Cli_AsEBRead	Reads a part of IPI area from a PLC.
Cli_AsEBWrite	Writes a part of IPI area into a PLC.
Cli_AsMBRead	Reads a part of Merkers area from a PLC.
Cli_AsMBWrite	Writes a part of Merkers area into a PLC.
Cli_AsTMRead	Reads timers from a PLC.
Cli_AsTMWrite	Write timers into a PLC.
Cli_AsCTRead	Reads counters from a PLC.
Cli_AsCTWrite	Write counters into a PLC.
Cli_AsListBlocksOfType	Returns the AG blocks list of a given type.
Cli_AsReadSZL	Reads a partial list of given ID and Index.
Cli_AsReadSZLList	Reads the list of partial lists available in the CPU.
Cli_AsFullUpload	Uploads a block from AG with Header and Footer infos.
Cli_AsUpload	Uploads a block from AG.
Cli_AsDownload	Download a block into AG.
Cli_AsDBGet	Uploads a DB from AG using DBRead.
Cli_AsDBFill	Fills a DB in AG with a given byte.
Cli_AsCopyRamToRom	Performs the Copy Ram to Rom action.
Cli_AsCompress	Performs the Compress action.

Cli_SetAsCallback

Description

Sets the user callback that the Client object has to call when the asynchronous data transfer is complete.

Declaration

```
int Cli_SetAsCallback(S7Object Client, pfn_CliCompletion pCompletion,
    void *usrPtr);

function Cli_SetAsCallback(Client : S7Object; pCompletion,
    usrPtr : pointer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
pCompletion	Pointer to function	In	Pointer to the Callback function
usrPtr	Pointer	In	User pointer passed back

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The expected callback is defined as:

```
typedef void (S7API *pfn_CliCompletion) (void *usrPtr, int opCode,
   int opResult);
```

Where **S7API** is **__stdcall** only for Windows.

This function must be present into your source code, so let's see also how is defined across other languages:

Pascal:

```
TS7CliCompletion = procedure(usrPtr : Pointer; opCode, opResult :
integer);
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
```

C#

```
public delegate void S7CliCompletion(IntPtr usrPtr, int opCode,
    int opResult);
```

usPtr is an optional parameter (meaning that it can be NULL) that is useful to switch the context from API-procedural to object-oriented (except for C#).

Let's suppose that we have a TStation class that uses a Client, **CliCompletion** must be a plain function (because the OS doesn't know anything about classes members).

Q : How can we instruct our Snap7Client to let him call our class member ?
A : Storing the class instance in usrPtr.

Examples:

C++

```
// Class definition
class TStation()
{
private:
    TS7Client *Client;
public:
    TStation();
    void TransferComplete(int opCode, int opResult);
};


// This is the plain function (API-procedural context)
void S7API CliCompletion(void *usrPtr, int opCode, int opResult)
{
    // Cast usrPtr to Station
    TStation *MyStation  = (TStation *) usrPtr;
    // Call the member
    MyStation->TransferComplete(opCode, opResult);
}

// This is the TStation member (OO Context)
void TStation::TransferComplete(int opCode, int opResult)
{
    if (opResult==0)
        DoSomething();
    else
        DoSomethingElse();
}

TStation::TStation()
{
    // Client creation and callback set
    Client = new TS7Client();
    // Callback set
    Client->SetAsCallback(CliCompletion, this);
    // "this" parameter is the fingerprint of TStation instance.
}
```

Pascal

```
// Class definition
TStation = class
private
    Client : TS7Client;
public
    constructor Create;
    procedure TransferComplete(opCode, opResult : integer);
end;

// This is the plain function (API-procedural context)
procedure CliCompletion(usrPtr : pointer, opCode,
    opResult : integer);
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
begin
   // Cast usrPtr to Station and call the method
   TStation(usrPtr).TransferComplete(opCode, opResult);
end;

// This is the TStation member (OO Context)
procedure TStation.TransferComplete(opCode, opResult : integer);
begin
    if opResult=0 then
        DoSomething
    else
        DoSomethingElse;
end;

constructor TStation.Create;
begin
    // Client creation and callback set
    Client := TS7Client.Create;
    Client.SetAsCallback(CliCompletion, self);
    // "self" parameter is the fingerprint of TStation instance.
end;
```

C#

Here the thing is simpler, using a delegate we don't need to cast usrPtr.

```
class TStation
{
    static S7Client Client;

    static void CompletionProc(IntPtr usrPtr, int opCode,
        int opResult)
    {
        if (opCode == 0)
            DoSomething();
        else
            DoSomethingElse();
    }

    public TStation()
    {
        Client = new S7Client();
        Client.SetAsCallBack(CompletionProc, IntPtr.Zero);
    }
}
```

Remarks

To disable the callback calling after an asynchronous job, call Cli_SetAsCallback with CompletionProc=NULL

Cli_CheckAsCompletion

Description

Checks if the current asynchronous job was done and terminates immediately.

Declaration

```
int Cli_CheckAsCompletion(S7Object Client, int *opResult);

function Cli_CheckAsCompletion(Client : S7Object;
   var opResult : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
opResult	Pointer to Integer 32	In	Operation Result

Return value

	Value	
JobComplete	0	Job was done
JobPending	1	Job in progress
errLibInvalidObject	-2	Invalid handled supplied

If Return value is JobComplete, **opResult** contains the function result, i.e. the same value that we would have if we had called the synchronous function.

Remarks

Use this function inside a **while cycle** only in conjunction with other operations and always inserting a small delay to avoid CPU waste time.

Wrong use of function : the cycle is wasting the CPU time, consider to use Cli_WaitAsCompletion.

```
while (Cli_CheckAsCompletion(MyClient, opResult)!=JobComplete)
{};
```

Correct use of function .

```
while (Cli_CheckAsCompletion(MyClient, opResult)!=JobComplete)
{
    DoSomething();
    Sleep(1);
};
```

Cli_WaitAsCompletion

Description

Waits until the current asynchronous job is done or the timeout expires.

Declaration

```
int Cli_WaitAsCompletion(S7Object Client, int Timeout);

function Cli_WaitAsCompletion(Client : S7Object;
   Timeout : integer) : integer;
```

Parameters

	Type	Dir.	
Client	Native Integer	In	The handle as return value of Cli_Create(), passed by value.
Timeout	Integer 32	In	Operation Timeout (ms)

Return value

This function returns the Job result, i.e. the same value that we would have if we had called the synchronous function.

- 0 : The asynchronous function was accomplished with no errors.
- 0x02200000 - **errCliJobTimeout** if timeout expired.
- Other values : see the Errors Code List.

Remarks

This function uses native OS primitives (events, signals..) to avoid CPU time waste.

Cli_AsReadArea

Description

This is the asynchronous counterpart of **Cli_ReadArea.**
See it for the parameters explanation.

Declaration

```
int Cli_AsReadArea(S7Object Client, int Area, int DBNumber, int Start,
    int Amount, int WordLen, void *pUsrData);

function Cli_AsReadArea(Client : S7Object; Area, DBNumber, Start,
    Amount, WordLen : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsWriteArea

Description

This is the asynchronous counterpart of **Cli_WriteArea.**
See it for the parameters explanation.

Declaration

```
int Cli_AsWriteArea(S7Object Client, int Area, int DBNumber, int Start,
    int Amount, int WordLen, void *pUsrData);

function Cli_AsWriteArea(Client : S7Object; Area, DBNumber, Start,
    Amount, WordLen : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsDBRead

Description

This is the asynchronous counterpart of **Cli_DBRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsDBRead(S7Object Client, int DBNumber, int Start, int Size,
    void *pUsrData);

function Cli_AsDBRead(Client : S7Object; DBNumber, Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsDBWrite

Description

This is the asynchronous counterpart of **Cli_DBWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsDBWrite(S7Object Client, int DBNumber, int Start, int Size,
    void *pUsrData);

function Cli_AsDBWrite(Client : S7Object; DBNumber, Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsABRead

Description

This is the asynchronous counterpart of **Cli_ABRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsABRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsABRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsABWrite

Description

This is the asynchronous counterpart of **Cli_ABWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsABWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsABWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsEBRead

Description

This is the asynchronous counterpart of **Cli_EBRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsEBRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsEBRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsEBWrite

Description

This is the asynchronous counterpart of **Cli_EBWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsEBWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsEBWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsMBRead

Description

This is the asynchronous counterpart of **Cli_MBRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsMBRead(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsMBRead(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsMBWrite

Description

This is the asynchronous counterpart of **Cli_MBWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsMBWrite(S7Object Client, int Start, int Size, void *pUsrData);

function Cli_AsMBWrite(Client : S7Object; Start,
    Size : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsTMRead

Description

This is the asynchronous counterpart of **Cli_TMRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsTMRead(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_AsTMRead(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsTMWrite

Description

This is the asynchronous counterpart of **Cli_TMWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsTMWrite(S7Object Client, int Start, int Amount, void
*pUsrData);

function Cli_AsTMWrite(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsCTRead

Description

This is the asynchronous counterpart of **Cli_CTRead.**
See it for the parameters explanation.

Declaration

```
int Cli_AsCTRead(S7Object Client, int Start, int Amount, void *pUsrData);

function Cli_AsCTRead(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsCTWrite

Description

This is the asynchronous counterpart of **Cli_CTWrite.**
See it for the parameters explanation.

Declaration

```
int Cli_AsCTWrite(S7Object Client, int Start, int Amount, void
*pUsrData);

function Cli_AsCTWrite(Client : S7Object; Start,
    Amount : integer; pUsrData : pointer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsListBlocksOfType

Description

This is the asynchronous counterpart of **Cli_ListBlocksOfType.**
See it for the parameters explanation.

Declaration

```
int Cli_AsListBlocksofType(S7Object Client, int BlockType,
    TS7BlocksOfType *pUsrData, int *ItemsCount);

function Cli_AsListBlocksOfType(Client : S7Object;
    BlockType : integer; pUsrData : PS7BlocksOfType;
    var ItemsCount : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsReadSZL

Description

This is the asynchronous counterpart of **Cli_ReadSZL.**
See it for the parameters explanation.

Declaration

```
int Cli_AsReadSZL(S7Object Client, int ID, int Index,
    TS7SZL *pUsrData, int *Size);

function Cli_AsReadSZL(Client : S7Object; ID, Index : integer;
    pUsrData : PS7SZL; var Size : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsReadSZLList

Description

This is the asynchronous counterpart of **Cli_ReadSZLList.**
See it for the parameters explanation.

Declaration

```
int Cli_AsReadSZLList(S7Object Client, TS7SZLList *pUsrData,
    int *ItemsCount);

function Cli_AsReadSZLList(Client : S7Object; pUsrData : PS7SZLList;
    var ItemsCount : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsFullUpload

Description

This is the asynchronous counterpart of **Cli_FullUpload.**
See it for the parameters explanation.

Declaration

```
int Cli_AsFullUpload(S7Object Client, int BlockType, int BlockNum,
    void *pUsrData, int *Size);

function Cli_AsFullUpload(Client : S7Object; BlockType, BlockNum :
integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsUpload

Description

This is the asynchronous counterpart of **Cli_Upload.**
See it for the parameters explanation.

Declaration

```
int Cli_AsUpload(S7Object Client, int BlockType, int BlockNum,
    void *pUsrData, int *Size);

function Cli_AsUpload(Client : S7Object; BlockType, BlockNum : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Cli_AsDownload

Description

This is the asynchronous counterpart of **Cli_Download.**
See it for the parameters explanation.

Declaration

```
int Cli_AsDownload(S7Object Client, int BlockNum, void *pUsrData, int
*Size);

function Cli_ASDownload(Client : S7Object; BlockNum : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsDBGet

Description

This is the asynchronous counterpart of **Cli_DBGet.**
See it for the parameters explanation.

Declaration

```
int Cli_AsDBGet(S7Object Client, int DBNumber, void *pUsrData, int *Size);

function Cli_AsDBGet(Client : S7Object; DBNumber : integer;
    pUsrData : pointer; var Size : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsDBFill

Description

This is the asynchronous counterpart of **Cli_DBFill.**
See it for the parameters explanation.

Declaration

```
int Cli_AsDBFill(S7Object Client, int DBNumber, int FillChar);

function Cli_AsDBFill(Client : S7Object; DBNumber : integer;
    FillChar : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsCopyRamToRom

Description

This is the asynchronous counterpart of **Cli_CopyRamToRom.**
See it for the parameters explanation.

Declaration

```
int Cli_AsCopyRamToRom(S7Object Client, int Timeout);

function Cli_AsCopyRamToRom(Client : S7Object; Timeout : integer) :
integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it’s preferable to use the Synchronous function.

Cli_AsCompress

Description

This is the asynchronous counterpart of **Cli_Compress.**
See it for the parameters explanation.

Declaration

```
int Cli_AsCompress(S7Object Client, int Timeout);

function Cli_AsCompress(Client : S7Object; Timeout : integer) : integer;
```

Return value

- 0 : The Asynchronous Job was successfully started.
- 0x00300000 (**errCliJobPending)** : Another job is running.
- Other values : see the Errors Code List.

Remarks

The function starts the job and terminates immediately. To know the job completion you can use one of the above functions.

If the data size to exchange is lesser or equal than the PDU negotiated it's preferable to use the Synchronous function.

Server API Reference

Administrative functions

These functions allow controlling the behavior a Server Object.

Function	Purpose
Srv_Create	Creates a Server Object.
Srv_Destroy	Destroys a Server Object.
Srv_StartTo	Starts a Server Object onto a given IP Address.
Srv_Start	Starts a Server Object onto the default adapter.
Srv_Stop	Stops the Server.
Srv_GetParam	Reads an internal Server parameter.
Srv_SetParam	Writes an internal Server Parameter.

Srv_Create

Description

Creates a Server and returns its handle, which is the reference that you have to use every time you refer to that Server.

The maximum number of Servers that you can create depends only on the system memory amount and on the network adapters amount.

Declaration

```
S7Object Srv_Create();

function Srv_Create : S7Object;
```

Parameters

No parameters

Example

```
S7Object Server;     // Declaration

Server=Srv_Create(); // Creation
// Do something

Srv_Destroy(Server); // Destruction
```

Remarks

The handle is a memory pointer, so its size varies depending on the platform (32 or 64 bit). If you use the wrappers provided it is already declared as native integer, otherwise you can store it into a “pointer type” var.

Simply store it, it should not be changed ever.

Srv_Destroy

Description

Destroy a Server of given handle.
Before destruction the Server is stopped, all clients disconnected and all shared memory blocks released.

Declaration

```
void Srv_Destroy(S7Object *Server);

procedure Srv_Destroy(var Server : S7Object);
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by reference.

Example

```
S7Object Server;     // Declaration

Server=Srv_Create(); // Creation
// Do something

Srv_Destroy(Server); // Destruction
```

Remarks

The handle is passed by reference and it's set to NULL by the function. This allows you to call Srv_Destroy() more than once without worrying about memory exceptions.

Srv_GetParam

Description

Reads an internal Server object parameter.

Declaration

```
int Srv_GetParam(S7Object Server, int ParamNumber, void *pValue);

function Srv_GetParam(Server : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that will receive the parameter value.

Return value

- 0 : The parameter was successfully read.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Srv_SetParam

Description

Sets an internal Server object parameter.

Declaration

```
int Srv_SetParam(S7Object Server, int ParamNumber, void *pValue);

function Srv_SetParam(Server : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that contains the parameter value.

Return value

- 0 : The parameter was successfully set.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Srv_StartTo

Description

Starts the server and binds it to the specified IP address and the IsoTCP port.

Declaration

```
int Srv_StartTo(S7Object Server, const char *Address);

function Srv_StartTo(Client : S7Object; Address : PAnsiChar) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
Address	Pointer to Ansi String	In	Adapter IPV4 Address ex. “192.168.1.12” (1)

(1) If “0.0.0.0” is supplied, the default adapter is used.

Return value

- 0 : The Server is successfully started (or was already running).
- Other values : see the Errors Code List.

Srv_Start

Description

Starts the server and binds it to the IP address specified in the previous call of **Srv_StartTo().**

Declaration

```
int Srv_Start(S7Object Server);

function Srv_Start(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.

Return value

- 0 : The Server is successfully started (or was already running).
- Other values : see the Errors Code List.

Remarks

If Srv_StartTo() was not previously called, "0.0.0.0" is assumed as IP address.

Srv_Stop

Description

Stops the server, disconnects gracefully all clients, destroys al S7 workers and unbinds the listener socket from its address.

Declaration

```
int Srv_Stop(S7Object Server);

function Srv_Stop(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.

Return value

- 0 : The Server is successfully stopped (or was already stopped).
- Other values : see the Errors Code List.

Shared memory functions

These functions allow to share data between the user application and the server.

Function	Purpose
Srv_RegisterArea	Shares a given memory area with the server.
Srv_UnRegisterArea	“Unshares” a memory area previously shared.
Srv_LockArea	Locks a shared memory area.
Srv_UnlockArea	Unlocks a previously locked shared memory area.

Srv_RegisterArea

Description

Shares a memory area with the server. That memory block will be visible by the clients.

Declaration

```
int Srv_RegisterArea(S7Object Server, int AreaCode, word Index,
   void *pUsrData, int Size);

function Srv_RegisterArea(Server : S7Object; AreaCode : integer;
   Index : word; pUsrData : pointer; Size : integer) : integer;
```

Parameters

	Type	Dir.	Mean
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
AreaCode	integer 32	In	Area identifier.
Index	integer 16	In	DB Number if Area = srvAreaDB, otherwise is ignored.
pUsrData	Pointer to memory area	In	Address of user buffer.
Size	integer 32	In	Size of user buffer

AreaCode values

	Value	Mean
S7AreaPE	0	Process Inputs.
S7AreaPA	1	Process Outputs.
S7AreaMK	2	Merkers.
S7AreaCT	3	Timers
S7AreaTM	4	Counters.
S7AreaDB	5	DB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Srv_UnRegisterArea

Description

"Unshares" a memory area previously shared with Srv_RegisterArea().
That memory block will be no longer visible by the clients.

Declaration

```
int Srv_UnregisterArea(S7Object Server, int AreaCode, word Index);

function Srv_UnregisterArea(Server : S7Object; AreaCode : integer;
   Index : word) : integer;
```

Parameters

	Type	Dir.	Mean
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
AreaCode	integer 32	In	Area identifier.
Index	integer 16	In	DB Number if Area = srvAreaDB, otherwise is ignored.

AreaCode values

	Value	Mean
S7AreaPE	0	Process Inputs.
S7AreaPA	1	Process Outputs.
S7AreaMK	2	Merkers.
S7AreaCT	3	Timers
S7AreaTM	4	Counters.
S7AreaDB	5	DB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Srv_LockArea

Description

Locks a shared memory area.

Declaration

```
int Srv_LockArea(S7Object Server, int AreaCode, word Index);

function Srv_LockArea(Server : S7Object; AreaCode : integer;
   Index : word) : integer;
```

Parameters

	Type	Dir.	Mean
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
AreaCode	integer 32	In	Area identifier.
Index	integer 16	In	DB Number if Area = srvAreaDB, otherwise is ignored.

AreaCode values

	Value	Mean
S7AreaPE	0	Process Inputs.
S7AreaPA	1	Process Outputs.
S7AreaMK	2	Merkers.
S7AreaCT	3	Timers
S7AreaTM	4	Counters.
S7AreaDB	5	DB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Srv_UnlockArea

Description

Unlocks a previously locked shared memory area.

Declaration

```
int Srv_UnlockArea(S7Object Server, int AreaCode, word Index);

function Srv_UnockArea(Server : S7Object; AreaCode : integer;
   Index : word) : integer;
```

Parameters

	Type	Dir.	Mean
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
AreaCode	integer 32	In	Area identifier.
Index	integer 16	In	DB Number if Area = srvAreaDB, otherwise is ignored.

AreaCode values

	Value	Mean
S7AreaPE	0	Process Inputs.
S7AreaPA	1	Process Outputs.
S7AreaMK	2	Merkers.
S7AreaCT	3	Timers
S7AreaTM	4	Counters.
S7AreaDB	5	DB

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Control flow functions

These functions allow to setup/handle the events generated by a server.

Function	Purpose
Srv_SetEventsCallback	Sets the user callback that the Server object has to call when an event is created.
Srv_GetMask	Reads the specified filter mask.
Srv_SetMask	Writes the specified filter mask.
Srv_PickEvent	Extracts an event (if available) from the Events queue.
Srv_ClearEvents	Empties the Event queue.

Srv_SetEventsCallback

Description

Sets the user callback that the Server object has to call when an event is created.

Declaration

```
int Srv_SetEventsCallback(S7Object Server, pfn_SrvCallBack pCallBack,
    void *usrPtr);

function Srv_SetEventsCallback(Server : S7Object; pCallBack,
    usrPtr : pointer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
pCallBack	Pointer to function	In	Pointer to the Callback function
usrPtr	Pointer	In	User pointer passed back

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The expected callback is defined as:

```
typedef void (S7API *pfn_SrvCallBack) (void * usrPtr, PSrvEvent PEvent,
   int Size);
```

Where **S7API** is `__stdcall` only for Windows.

And PSrvEvent is a pointer to TSrvEvent defined as follow:

```
typedef struct{
    time_t EvtTime;     // Timestamp
    int EvtSender;      // Sender
    longword EvtCode;   // Event code
    word EvtRetCode;    // Event result
    word EvtParam1;     // Param 1 (if available)
    word EvtParam2;     // Param 2 (if available)
    word EvtParam3;     // Param 3 (if available)
    word EvtParam4;     // Param 4 (if available)
}
```

snap7.net.cs and **snap7.pas** contain the C# and Pascal definition for this struct.

usPtr is an optional parameter (meaning that it can be NULL) that is useful to switch the context from API-procedural to object-oriented (except for C#).

Size is the Event size (for a future backward compatibility).

This function must be present into your source code, so let's see also how is defined across other languages:

Pascal:

```
TSrvCallBack = procedure(usrPtr : pointer; PEvent : PSrvEvent;
    Size : integer);
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
```

C#

```
public delegate void TSrvCallback(IntPtr usrPtr, ref USrvEvent Event,
    int Size);
```

Remarks

The call of user Callback is subject to mask filtering (see Snap7Server **Control flow** chapter for further information)

To clear the Callback, call this function with pCallBack=NULL

Srv_SetReadsEventsCallback

Description

Sets the user callback that the Server object has to call when a Read event is created.

Declaration

```
int Srv_SetReadEventsCallback(S7Object Server, pfn_SrvCallBack pCallBack,
    void *usrPtr);

function Srv_SetReadEventsCallback(Server : S7Object; pCallBack,
    usrPtr : pointer) : integer;
```

Remarks

Refer to Srv_SetEventsCallBack for parameters, result and CallBack prototype.

The **EvtRetCode** will be always 0.

Srv_GetMask

Description

Reads the specified filter mask.

Declaration

```
int Srv_GetMask(S7Object Server, int MaskKind, longword *Mask);

function Srv_GetMask(Server : S7Object; MaskKind : integer;
    var Mask : longword) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
MaskKind	Integer	In	Kind of the Mask.
Mask	Pointer to unsigned 32	In	Pointer to the variable that will receive the mask value.

MaskKind values

	Value	Mean
mkEvent	0	Event Mask
mkLog	1	Log Mask

Return value

- 0 : The mask was successfully read.
- Other values : see the Errors Code List.

Srv_SetMask

Description

Writes the specified filter mask.

Declaration

```
int Srv_SetMask(S7Object Server, int MaskKind, longword Mask);

function Srv_SetMask(Server : S7Object; MaskKind : integer;
    Mask : longword) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
MaskKind	Integer	In	Kind of the Mask.
Mask	Unsigned integer 32	In	Value of the Mask.

MaskKind values

	Value	Mean
mkEvent	0	Event Mask
mkLog	1	Log Mask

Return value

- 0 : The mask was successfully set.
- Other values : see the Errors Code List.

Srv_PickEvent

Description

Extracts an event (if available) from the Events queue.

Declaration

```
int Srv_PickEvent(S7Object Server, TSrvEvent *pEvent,
   int *EvtReady);

function Srv_PickEvent(Server : S7Object;
   var Event : TSrvEvent; var EvtReady : integer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
pEvent	Pointer to struct	In	Address of user Event variable
EvtReady	Pointer to Integer 32	In	Address of user EvtReady var.

If an Event was available EvtReady=1 otherwise EvtReady=0.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

see Snap7Server **Control flow** chapter for further information.

Object-oriented wrappers expose a more convenient bool function.

Srv_ClearEvents

Description

Empties the Event queue.

Declaration

```
int Srv_ClearEvents(S7Object Client);

function Srv_ClearEvents(Client : S7Object);
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Miscellaneous functions

These are utility functions.

Function	Purpose
Srv_GetStatus	Returns the last job execution time in milliseconds.
Srv_SetCpuStatus	Returns the last job result.
Srv_EventText	Returns a textual explanation of a given event.
Srv_ErrorText	Returns a textual explanation of a given error number.

Srv_GetStatus

Description

Reads the server status, the Virtual CPU status and the number of the clients connected.

Declaration

```
int Srv_GetStatus(S7Object Server, int *ServerStatus, int *CpuStatus,
   int *ClientsCount);

function Srv_GetStatus(Server : S7Object; var ServerStatus,
   CpuStatus, ClientsCount : integer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
ServerStatus	Pointer to Integer 32	In	Pointer to the variable that will receive the Server Status.
CpuStatus	Pointer to Integer 32	In	Pointer to the variable that will receive the Virtual CPU Status.
ClientsCount	Pointer to Integer 32	In	Pointer to the variable that will receive the Clients count.

ServerStatus values

	Value	Mean
SrvStopped	0	The Server is stopped.
SrvRunning	1	The Server is Running.
SrvError	2	Server Error.

CpuStatus values

	Value	Mean
S7CpuStatusUnknown	0x00	Unknown.
S7CpuRun	0x08	CPU in RUN
S7CpuStop	0x04	CPU in Stop

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remark

The CPU status can be changed by a client calling the related S7 Control function (Cold Start/ Warm Start / Stop) or programmatically, server side, calling the function **Srv_SetCpuStatus()**.

Srv_SetCpuStatus

Description

Sets the Virtual CPU status.

Declaration

```
int Srv_SetCpuStatus(S7Object Server, int CpuStatus);

function Srv_SetCpuStatus(Server : S7Object;
   CpuStatus : integer) : integer;
```

Parameters

	Type	Dir.	
Server	Native Integer	In	The handle as return value of Srv_Create(), passed by value.
CpuStatus	Integer 32	In	Value of Virtual CPU Status.

CpuStatus values

	Value	Mean
S7CpuStatusUnknown	0x00	Unknown.
S7CpuRun	0x08	CPU in RUN
S7CpuStop	0x04	CPU in Stop

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Srv_ErrorText

Description

Returns a textual explanation of a given error number.

Declaration

```
int Srv_ErrorText(int Error, char *Text, int TextLen);

function Srv_ErrorText(Error : integer, Text : PAnsiChar;
   TextLen :  integer) : integer;
```

Parameters

	Type	Dir.	
Error	Integer 32	In	Error code
Text	Pointer to Ansi String	In	Address of the char array
TextLen	Integer 32	In	Size of the char array

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This is a translation function, so there is no need of a Server handle.

The messages are in (internet) English, all they are in s7_text.cpp.

Srv_EventText

Description

Returns a textual explanation of a given event.

Declaration

```
int Srv_EventText(TSrvEvent *Event, char *Text, int TextLen);

function Srv_EventText(var Event : TSrvEvent, Text : PAnsiChar;
   TextLen :  integer) : integer;
```

Parameters

	Type	Dir.	
Event	Pointer to struct	In	Address of user Event variable
Text	Pointer to Ansi String	In	Address of the char array
TextLen	Integer 32	In	Size of the char array

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Remarks

This is a translation function, so there is no need of a Server handle.

The messages are in (internet) English, all they are in s7_text.cpp.

The event string is formatted as follow:

<Date and time><Sender><Description>[<Parameters>][<Result>]

This is an example:

```
2013-06-25 15:39:25 [192.168.0.70] Read SZL request, ID:0x0424 INDEX:0x0000 -->
OK
```

Partner API Reference

Administrative functions

These functions allow controlling the behavior a Partner Object.

Function	Purpose
Par_Create	Creates a Partner Object.
Par_Destroy	Destroys a Partner Object.
Par_StartTo	Starts a Partner Object onto a given IP Address.
Par_Start	Starts a Partner Object onto the previous parameters supplied.
Par_Stop	Stops the Partner.
Par_GetParam	Reads an internal Partner parameter.
Par_SetParam	Writes an internal Partner Parameter.
Par_SetSendCallback	Sets the user callback that the Partner object has to call when the asynchronous data sent is complete.
Par_SetRecvCallback	Sets the user callback that the Partner object has to call when a data packet is incoming.

Par_Create

Description

Creates a Partner and returns its handle, which is the reference that you have to use every time you refer to that Partner.

Declaration

```
S7Object Par_Create(int Active);

function Par_Create : S7Object;
```

Parameters

	Type	Dir.	
Active	Integer	In	0 : A Passive Partner will be created. 1 : An Active partner will be created.

Example

```
S7Object Partner;     // Declaration

Partner=Par_Create(1); // Active Partner Creation
// Do something

Par_Destroy(Partner); // Destruction
```

Remarks

The handle is a memory pointer, so its size varies depending on the platform (32 or 64 bit). If you use the wrappers provided it is already declared as native integer, otherwise you can store it into a "pointer type" var.

Simply store it, it should not be changed ever.

Par_Destroy

Description

Destroy a Partner of given handle.
Before destruction the Partner is stopped, all clients disconnected and all shared memory blocks released.

Declaration

```
void Par_Destroy(S7Object *Partner);

procedure Par_Destroy(var Partner : S7Object);
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by reference.

Example

```
S7Object Partner;     // Declaration

Partner=Par_Create(); // Creation
// Do something

Par_Destroy(Partner); // Destruction
```

Remarks

The handle is passed by reference and it's set to NULL by the function. This allows you to call Par_Destroy() more than once without worrying about memory exceptions.

Par_GetParam

Description

Reads an internal Partner object parameter.

Declaration

```
int Par_GetParam(S7Object Partner, int ParamNumber, void *pValue);

function Par_GetParam(Partner : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that will receive the parameter value.

Return value

- 0 : The parameter was successfully read.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Par_SetParam

Description

Sets an internal Partner object parameter.

Declaration

```
int Par_SetParam(S7Object Partner, int ParamNumber, void *pValue);

function Par_SetParam(Partner : S7Object; ParamNumber : integer;
    pValue :  pointer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
ParamNumber	Integer	In	Parameter number.
pValue	Pointer	In	Pointer to the variable that contains the parameter value.

Return value

- 0 : The parameter was successfully set.
- Other values : see the Errors Code List.

Since the couple GetParam/SetParam is present in all three Snap7 objects, there is a detailed description of them (*Internal parameters*).

Par_StartTo

Description

Starts the Partner and binds it to the specified IP address and the IsoTCP port.

Declaration

```
int Par_StartTo(S7Object Partner, const char *LocalAddress,
    const char *LocalAddress, word LocTsap, word RemTsap);

function Par_StartTo(Client : S7Object; LocalAddress, RemoteAddress :
    PAnsiChar; LocTsap, RemTsap : word) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
LocalAddress	Pointer to Ansi String	In	PC host IPV4 Address
RemoteAddress	Pointer to Ansi String	In	PLC IPV4 Address
LocTsap	Unsigned Integer 16	In	Local TSAP
RemTsap	Unsigned Integer 16	In	PLC TSAP

(1) If "0.0.0.0" is supplied, the default adapter is used.

Return value

- 0 : The Partner is successfully started (or was already running).
- Other values : see the Errors Code List.

Remark

See § **Snap7Partner->The Snap7 Model** for further information.

Par_Start

Description

Starts the Partner and binds it to the parameters specified in the previous call of **Par_StartTo().**

Declaration

```
int Par_Start(S7Object Partner);

function Par_Start(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.

Return value

- 0 : The Partner is successfully started (or was already running).
- Other values : see the Errors Code List.

Remarks

This function can be called only after a previous **`Par_StartTo()`** which internally sets Addresses, and TSAPs.

Par_Stop

Description

Stops the Partner, disconnects gracefully the remote partner.

Declaration

```
int Par_Stop(S7Object Partner);

function Par_Stop(Client : S7Object) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.

Return value

- 0 : The Partner is successfully stopped (or was already stopped).
- Other values : see the Errors Code List.

Par_SetSendCallback

Description

Sets the user callback that the Partner object has to call when the asynchronous data sent is complete.

Declaration

```
int Par_SetSendCallback(S7Object Partner, pfn_ParSendCompletion
pCompletion,
    void *usrPtr);

function Par_SetSendCallback(Partner : S7Object; pCompletion,
    usrPtr : pointer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
pCompletion	Pointer to function	In	Pointer to the Callback function
usrPtr	Pointer	In	User pointer passed back

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The expected callback is defined as:

```
typedef void (S7API *pfn_ParSendCompletion) (void *usrPtr,
   int opResult);
```

Where **S7API** is **__stdcall** only for Windows.

This function must be present into your source code, so let's see also how is defined across other languages:

Pascal:

```
TParBSendCompletion = procedure(usrPtr : Pointer;
    opResult : integer);
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
```

C#

```
public delegate void S7ParSendCompletion(IntPtr usrPtr,
    int opResult);
```

usPtr is an optional parameter (meaning that it can be NULL) that is useful to switch the context from API-procedural to object-oriented (except for C#).

See **Partner Applications** for an example of callback use.

Par_SetRecvCallback

Description

Sets the user callback that the Partner object has to call when a data packet is incoming.

Declaration

```
int Par_SetRecvCallback(S7Object Partner, pfn_ParRecvCallback
pCompletion,
    void *usrPtr);

function Par_SetRecvCallback(Partner : S7Object; pCompletion,
    usrPtr : pointer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
pCompletion	Pointer to function	In	Pointer to the Callback function
usrPtr	Pointer	In	User pointer passed back

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

The expected callback is defined as:

```
typedef void (S7API *pfn_ParRecvCallback) (void *usrPtr,
   int opResult, longword R_ID, void *pData, int Size);
```

Where **S7API** is **__stdcall** only for Windows.

This function must be present into your source code, so let's see also how is defined across other languages:

Pascal:

```
TParRecvCallback = procedure(usrPtr : Pointer; opResult : integer;
    R_ID : longword; pData : Pointer; Size : integer);
{$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
```

C#

```
public delegate void S7ParSendCompletion(IntPtr usrPtr,
    int opResult, uint R_ID, IntPtr pData, int Size);
```

usPtr is an optional parameter (meaning that it can be NULL) that is useful to switch the context from API-procedural to object-oriented (except for C#).

pData points to the internal Partner buffer.

Size is the amount (bytes) of the incoming packet.

opResult is the transfer job result, 0 : ok, other see Error Code List.

Data Transfer functions

These functions allow the Partner to exchange data with its counterpart into a PLC.

Function	Purpose
Par_BSend	Sends a data packet to the partner.
Par_AsBSend	Sends an asynchronous data packet to the partner.
Par_CheckAsBSendCompletion	Checks if the current asynchronous job was completed.
Par_WaitAsBSendCompletion	Waits until the current asynchronous send job is done.
Par_BRecv	Receives a data packet from the partner.
Par_CheckAsBRecvCompletion	Checks if a packed received was received.

Par_BSend

Description

Sends a data packet to the partner. This function is synchronous, i.e. it terminates when the transfer job (send+ack) is complete.

Declaration

```
int Par_BSend(S7Object Partner, longword R_ID, void *pUsrData, int Size);

function Par_BSend(Partner : S7Object; R_ID : longword;
   pUsrData : Pointer; Size : integer) : integer;
```

Parameters

	Type	Dir.	Mean
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
R_ID	unsigned integer 32	In	Routing User parameter.
pUsrData	Pointer to memory area	In	Address of user buffer.
Size	integer	In	Size (byte) of the user buffer

R_ID is a routing parameter : the same value must be supplied to the BRecv FB.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_AsBSend

Description

Sends a data packet to the partner. This function is asynchronous, i.e. it terminates immediately, a completion method is needed to know when the transfer is complete.

Declaration

```
int Par_AsBSend(S7Object Partner, longword R_ID, void *pUsrData,
   int Size);

function Par_AsBSend(Partner : S7Object; R_ID : longword;
   pUsrData : Pointer; Size : integer) : integer;
```

Parameters

	Type	Dir.	Mean
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
R_ID	unsigned integer 32	In	Routing User parameter.
pUsrData	Pointer to memory area	In	Address of user buffer.
Size	integer	In	Size (byte) of the user buffer

R_ID is a routing parameter : the same value must be supplied to the BRecv FB.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_CheckAsBSendCompletion

Description

Checks if the current asynchronous send job was completed and terminates immediately.

Declaration

```
int Par_CheckAsCompletion(S7Object Partner, int *opResult);

function Par_CheckAsCompletion(Partner : S7Object;
   var opResult : integer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
opResult	Pointer to Integer 32	In	Operation Result

Return value

	Value	
JobComplete	0	Job was done
JobPending	1	Job in progress
errLibInvalidObject	-2	Invalid handled supplied

If Return value is JobComplete, **opResult** contains the function result, i.e. the same value that we would have if we had called the synchronous function.

Remarks

Use this function inside a **while cycle** only in conjunction with other operations and always inserting a small delay to avoid CPU waste time.

Wrong use of function : the cycle is wasting the CPU time, consider to use Par_WaitAsBSendCompletion.

```
while (Par_CheckAsBSendCompletion(MyPartner, opResult)!=JobComplete)
{};
```

Correct use of function .

```
while (Par_CheckAsBSendCompletion(MyPartner, opResult)!=JobComplete)
{
    DoSomething();
    Sleep(1);
};
```

Par_WaitAsBSendCompletion

Description

Waits until the current asynchronous send job is done or the timeout expires.

Declaration

```
int Par_WaitAsBSendCompletion(S7Object Partner, longword Timeout);

function Par_WaitAsBSendCompletion(Partner : S7Object;
   Timeout : longword) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
Timeout	Unsigned Integer 32	In	Operation Timeout (ms)

Return value

This function returns the Job result, i.e. the same value that we would have if we had called the synchronous function.

- 0 : The asynchronous function was accomplished with no errors.
- 0x00B00000 - **errParSendTimeout** if timeout expired.
- Other values : see the Errors Code List.

Remarks

This function uses native OS primitives (events, signals..) to avoid CPU time waste.

Par_BRecv

Description

Receives a data packet from the partner. This function is synchronous, it waits until a packet is received or the timeout supplied expires.

Declaration

```
int Par_BRecv(S7Object Partner, longword *R_ID, void *pUsrData,
   int *Size, longword Timeout);

function Par_BRecv(Partner : S7Object; var R_ID : longword;
   pUsrData : Pointer; var Size : integer; Timeout : longword) : integer;
```

Parameters

	Type	Dir.	Mean
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
R_ID	Pointer to unsigned integer 32	In	Address of Routing User parameter.
pUsrData	Pointer to memory area	In	Address of user buffer.
Size	Pointer to integer 32	In	Size (byte) of received packet
Timeout	Unsigned Integer 32	In	Operation Timeout (ms)

R_ID is the routing parameter that the remote partner supplied to its BSend FB.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_CheckAsBRecvCompletion

Description

Checks if a packed received was received.

Declaration

```
int Par_CheckAsBRecvCompletion(S7Object Partner, int *opResult,
   longword *R_ID, void *pData, int *Size);

function Par_CheckAsBRecvCompletion(Partner : S7Object;
   var opResult : integer; var R_ID : longword; pData : Pointer;
   var Size : integer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
opResult	Pointer to Integer 32	In	Operation Result
R_ID	Pointer to unsigned integer 32	In	Address of Routing User parameter.
pData	Pointer to memory area	In	Address of packet buffer.
Size	Pointer to integer 32	In	Size (byte) of received packet

Return value

	Value	
JobComplete	0	Packet ready
JobPending	1	Packet not ready
errLibInvalidObject	-2	Invalid handled supplied

If Return value is JobComplete, **opResult** contains the function result, i.e. the same value that we would have if we had called the synchronous function.

Remarks

opResult, R_ID, pData and Size will contain valid values only if a packet was received, i.e. Return value = JobComplete.

Miscellaneous functions

These are utility functions.

Function	Purpose
Par_GetTimes	Returns the last send and recv jobs execution time in milliseconds.
Par_GetStats	Returns some statistics.
Par_GetLastError	Returns the last job result.
Par_GetStatus	Returns the Partner status.
Par_ErrorText	Returns a textual explanation of a given error number.

Par_GetTimes

Description

Returns the last send and recv jobs execution time in milliseconds.

Declaration

```
int Par_GetTimes(S7Object Partner, longword *SendTime,
   longword *RecvTime);

function Par_GetTimes(Partner : S7Object;
   var SendTime, RecvTime : integer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
SendTime	Pointer to unsigned integer 32	In	Address of the send time variable
RecvTime	Pointer to unsigned integer 32	In	Address of the send time variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_GetStats

Description

Returns some statistics.

Declaration

```
int Par_GetTimes(S7Object Partner, longword *BytesSent,
   longword *BytesRecv, longword *SendErrors, longword *RecvErrors);

function Par_GetTimes(Partner : S7Object; var BytesSent, BytesRecv,
   SendErrors, RecvErrors : longword) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
BytesSent	Pointer to unsigned integer 32	In	Amount of bytes sent.
BytesRecv	Pointer to unsigned integer 32	In	Amount of bytes received.
SendErrors	Pointer to unsigned integer 32	In	Amount of send errors.
RecvErrors	Pointer to unsigned integer 32	In	Amount of recv errors.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_GetLastError

Description

Returns the last job result.

Declaration

```
int Par_GetLastError(S7Object Partner, int *LastError);

function Par_GetLastError(Partner : S7Object;
   var LastError : integer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
LastError	Pointer to integer 32	In	Address of the LastError variable

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_GetStatus

Description

Returns the Partner status.

Declaration

```
int Par_GetStatus(S7Object Partner, int *Status);

function Par_GetStatus(Partner : S7Object;
   var Status : integer) : integer;
```

Parameters

	Type	Dir.	
Partner	Native Integer	In	The handle as return value of Par_Create(), passed by value.
Status	Pointer to Integer 32	In	Pointer to the variable that will receive the Partner Status.

Status values

	Value	Mean
par_stopped	0	Stopped.
par_connecting	1	Running, active and trying to connect.
par_waiting	2	Running, passive and waiting for a connection
par_connected	3	Connected.
par_sending	4	Sending data.
par_receibing	5	Receiving data.
par_binderror	6	Error starting passive partner.

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

Par_ErrorText

Description

Returns a textual explanation of a given error number.

Declaration

```
int Par_ErrorText(int Error, char *Text, int TextLen);

function Par_ErrorText(Error : integer, Text : PAnsiChar;
   TextLen :  integer) : integer;
```

Parameters

	Type	Dir.	
Error	Integer 32	In	Error code
Text	Pointer to Ansi String	In	Address of the char array
TextLen	Integer 32	In	Size of the char array

Return value

- 0 : The function was accomplished with no errors.
- Other values : see the Errors Code List.

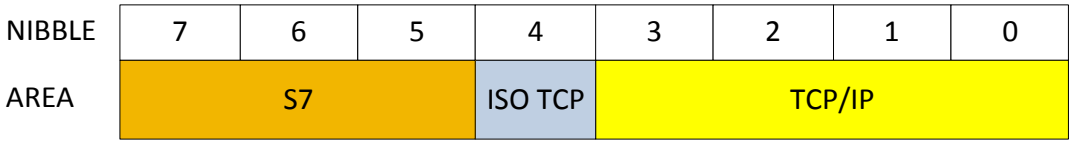
Remarks

This is a translation function, so there is no need of a Partner handle.

The messages are in (internet) English, all they are in s7_text.cpp.

API Error codes

Due to the classes layering, the result error code (a 32 bit value) is composed by three fields, as in figure:



Errors encoding

The first 16 bits (4 nibbles) represent a TCP/IP error that is raised by the OS socket layer. It is set in snap_msgsock.cpp.

The 5th nibble represent an ISO TCP error and is set in s7_isotcp.cpp.

The last three nibbles represent an S7 protocol error, they are shared between the objects TSnap7MicroClient/TSnap7Client, TSnap7Server and TSnap7Partner.

The three fields can contain contemporary a valid value and normally it is so.To isolate a layer, the error must be "and masked".

You can find, into the wrappers provided, the complete list of them.

For TCP/IP code please refer to your OS reference.

ISO TCP Error table

Mnemonic	HEX	Meaning
errIsoConnect	1	Iso Connection error.
errIsoDisconnect	2	Iso Disconnection error.
errIsoInvalidPDU	3	Malformatted PDU suppled.
errIsoInvalidDataSize	4	Bad Datasize passed to send/recv function.
errIsoNullPointer	5	Null pointer supplied.
errIsoShortPacket	6	A short packet received.
errIsoTooManyFragments	7	Too many packets without EoT flag (>64)
errIsoPduOverflow	8	The sum of fragments data exceeds the maximum packet size.
errIsoSendPacket	9	An error occurred during send.
errIsoRecvPacket	A	An error occurred during recv.
errIsoInvalidParams	B	Invalid TSAP params supplied.
errIsoResvd_1	C	Reserved (unused)
errIsoResvd_2	D	Reserved (unused)
errIsoResvd_3	E	Reserved (unused)
errIsoResvd_4	F	Reserved (unused)

Client Errors Table

Mnemonic	HEX	Meaning
errNegotiatingPDU	001	Error during PDU negotiation.
errCliInvalidParams	002	Invalid param(s) supplied to the current function.
errCliJobPending	003	A Job is pending : there is an async function in progress.
errCliTooManyItems	004	More than 20 items where passed to a MultiRead/Write area function.
errCliInvalidWordLen	005	Invalid Wordlen param supplied to the current function.
errCliPartialDataWritten	006	Partial data where written : The target area is smaller than the DataSize supplied.
errCliSizeOverPDU	007	A MultiRead/MultiWrite function has datasize over the PDU size.
errCliInvalidPlcAnswer	008	Invalid answer from the PLC.
errCliAddressOutOfRange	009	An address out of range was specified.
errCliInvalidTransportSize	00A	Invalid Transportsize parameter was supplied to a Read/WriteArea function.
errCliWriteDataSizeMismatch	00B	Invalid datasize parameter supplied to the current function.
errCliItemNotAvailable	00C	Item requested was not found in the PLC.
errCliInvalidValue	00D	Invalid value supplied to the current function.
errCliCannotStartPLC	00E	PLC cannot be started.
errCliAlreadyRun	00F	PLC is already in RUN stare.
errCliCannotStopPLC	010	PLC cannot be stopped.
errCliCannotCopyRamToRom	011	Cannot copy RAM to ROM : the PLC is running or doesn't support this function.
errCliCannotCompress	012	Cannot compress : the PLC is running or doesn't support this function.
errCliAlreadyStop	013	PLC is already in STOP state.
errCliFunNotAvailable	014	Function not available.
errCliUploadSequenceFailed	015	Block upload sequence failed.
errCliInvalidDataSizeRecvd	016	Invalid data size received from the PLC.
errCliInvalidBlockType	017	Invalid block type supplied to the current function.
errCliInvalidBlockNumber	018	Invalid block supplied to the current function.
errCliInvalidBlockSize	019	Invalid block size supplied to the current function.
errCliDownloadSequenceFailed	01A	Block download sequence failed.
errCliInsertRefused	01B	Insert command (implicit command sent after a block download) refused.
errCliDeleteRefused	01C	Delete command refused.
errCliNeedPassword	01D	This operation is password protected.
errCliInvalidPassword	01E	Invalid password supplied.
errCliNoPasswordToSetOrClear	01F	There is no password to set or clear : the protection is OFF.
errCliJobTimeout	020	Job timeout.
errCliPartialDataRead	021	Partial data where read : The source area is greater than the DataSize supplied.
errCliBufferTooSmall	022	The buffer supplied is too small.
errCliFunctionRefused	023	Function refused by the PLC.
errCliInvalidParamNumber	024	Invalid param number suppilied to Get/SetParam.
errCliDestroying	025	Cannot perform : the client is destroying.
errCliCannotChangeParam	026	Cannot change parameter because connected.

Server Errors Table

Mnemonic	HEX	Meaning
errSrvCannotStart	001	The server cannot be started.
errSrvDBNullPointer	002	A null was passed as area pointer.
errSrvAreaAlreadyExists	003	Trying to re-registering an area.
errSrvUnknownArea	004	Area code unknown.
errSrvInvalidParams	005	Invalid param(s) supplied to the current function.
errSrvTooManyDB	006	Trying to registering too many DB (>2048)
errSrvInvalidParamNumber	007	Invalid param number suppilied to Get/SetParam.
errSrvCannotChangeParam	008	Cannot change parameter because running.

Partner Errors Table

Mnemonic	HEX	Meaning
errParAddressInUse	002	Another passive partner is waiting for the same active address.
errParNoRoom	003	Trying to create too many partners for the current connection server.(>256)
errServerNoRoom	004	Trying to allocate too many connection servers (>256)
errParInvalidParams	005	Invalid param(s) supplied to the current function.
errParNotLinked	006	Cannot execute : the partner is not linked.
errParBusy	007	Partner busy : cannot send (Send or Recv sequence in progress)
errParFrameTimeout	008	Send or Recv sequence timeout.
errParInvalidPDU	009	Invalid PDU received, maybe a client (not a partner) is trying to communicate.
errParSendTimeout	00A	Timeout occurred in send function.
errParRecvTimeout	00B	Timeout occurred in recv function.
errParSendRefused	00C	Send refused by the PLC partner.
errParNegotiatingPDU	00D	Error during PDU negotiation.
errParSendingBlock	00E	Error during block send.
errParRecvingBlock	00F	Error during block recv.
errBindError	010	Cannot bind the address supplied.
errParDestroying	011	Cannot perform : the partner is destroying.
errParInvalidParamNumber	012	Invalid param number suppilied to Get/SetParam.
errParCannotChangeParam	013	Cannot change parameter because running.

Snap7 package

Snap7 package is named **snap7-full-x.y.z**.

- **x** is the Major Version.
- **y** is the minor version.
- **z** is the bugfix release.

Do not be scared by the size of the package, it contains many files because it's multi architecture and multi-platform.

Let's see how to navigate within the project that is divided into the following folders:

- **/build**
- **/doc**
- **/examples**
- **/release**
- **/rich-demos**
- **/src**
- **/LabVIEW**

[build]

This folder contains all you need to build the library, it is further divided into:

- **bin**
  Library output directory divided by platform-os, Unix OS target is the same of which you are running the compiler.
  Example : if you run "make -f i386_linux.mk all" under Ubuntu 13.10, you will find in bin/i386-linux a library that can run into Ubuntu 13.10 and in all OS derivative of Ubuntu that have the same GLIBC release.
- **temp**
  Intermediate objects/temp files, can be safety emptied.
- **unix**
  Unix (Linux/BSD/Solaris) makefiles directory.
- **windows**
  Windows projects/makefiles directory divided by compilers

You can find detailed information about library rebuild in the chapter **Rebuild Snap7**.

[doc]

Here you can find the project documentation.

[examples]

This folder contains source code examples divided by programming language.

- **cpp**
- **dot.net**
  suitable for Microsoft .NET and Mono 2.10
- **pascal**
  suitable for Delphi and FreePascal/Lazarus
- **plain-c**
- **Step 7**
  Contains plc-side examples, they are Step 7 projects (V5.5) that can be easily converted with TIA Portal V11 or V12.
- **temp**
  Intermediate objects/temp files, can be safety emptied.

All the sources are multi-architecture and multi-platform, into the platform-specific subfolder you will find projects/makefiles to build them.

[release]

This folder contains all files that are strictly needed to work with Snap7 : binary libraries and wrappers.

See **/release/deploy.html** for the updated list

[rich-demos]

While the examples are working "code snippets" to see how to use the library, rich demos are graphic programs that show almost all Snap7 features.

They are written using Lazarus (pascal) because:

- It's multi-platform.
- It's a powerful RAD that allow writing complex an nice programs in a breeze.

These demos don't have external dependencies, all they can be compiled with a fresh copy of Lazarus.

Originally these programs were written using Delphi and then converted (automatically with Lazarus).

For each platform supported there is a subfolder containing the projects ready to run.

However also the LabView examples can be considered rich-demos since they offer a graphical interface.
Anyway you need LabVIEW environment to run them.

[src]

This folder contains the Snap7 source code, please refer to **Snap7 source code** chapter for further information.

[LabVIEW]

This folder contains all you need to interface your LabView programs with Snap7.

See LabVIEW chapter for further information.

LabVIEW

NI LabVIEW is a software for systems design wich 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 not "exposed". You know that a cluster (a struct) contains some elements, but you don't know where in the memory they are, i.e. you don't know their physical address.

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

Let's see how can we interface LabVIEW with Snap7, keeping in mind these two major differences (execution and data storage) against the traditional programming languages.

The wrapper provided consists of:

1. A LabVIEW library (Snap7.lvlib) that contains a set of Vis. Each vi "wraps" a Snap7 function via the **Call Library function node**.
2. A "glue" DLL (lv_snap7.dll) that interfaces the Vis with Snap7.dll. It re-exports the typed-data functions and supplies new data adapter procedures for the untyped-data functions.

Since many of Snap7 functions only make sense only in a procedural context :

- **Asynchronous functions**
  All asynchronous functions are not exported, because are completely useless, indeed, in some cases, they can be harmful.
  LabVIEW is an inherently concurrent, adding a synchronization layer will complicate uselessly the execution flow.
- **Callbacks**
  LabVIEW cannot natively pass a pointer to a VI for use as a callback function in a DLL, a C wrapper must be used as workaround to provide an interface between the DLL and an user event. This is not a trivial task due to the data-driven nature of the language.
  At the end, the Event Structure must be used in a While Loop because when the Event Structure executes, it will only wait for and handle exactly one event. The Snap7 polling functions must be used instead. This is, imho, the better solution, because they are simple to use and, above all, because LabVIEW has very efficient mechanisms to optimize the parallel executions.

DLL Calling

To understand LabVIEW Snap7 interface, it's important to know how the Call Library function node works.

This is not a commercial book, rewriting base concepts that are already well explained has not much sense.
So, to explain this argument, I selected these two pages :

http://www.ni.com/white-paper/4877/en/

https://decibel.ni.com/content/docs/DOC-9080

As you can see in them, LabVIEW provides two ways to pass complex data to a DLL:

1. Adapt To Type
2. String (as LabVIEW string handle).

The first method is used when the data structure is well known in advance, i.e. when we wire it to the call library node.

All Snap7 functions which declare a struct (in snap7.h) as input use this method.

The second method allows to write VIs that accept, as input, generic buffers encapsulated in a string.

All Snap7 VIs that read/write an untyped buffer use the second method and the data adaption is made in lv_snap7.dll. The string type, in spite of its name, can contain anything since it has in head its length.

Generic buffers

Let see how Snap7 vi manage untyped buffers examining as example the **Cli_DBGet()** function of the client.

This is the C prototype of the function as exported by snap7.dll.

```
int S7API Cli_DBGet(S7Object Client, int DBNumber, void *pUsrData,
    int *Size);
```

This function reads an entire DB of given Number into the buffer pointed by pUsrData.

The first two parameters are simple to manage since they are simple typed vars.

pUsrData is the pointer to a generic buffer.

Size, in input must contain the size of the buffer supplied, in output contains the DB size, i.e how many bytes were read. If the buffer size is less than the DB size an error is returned (but however the buffer contains the partial data read).

The adapter function exported by lv_snap7.dll has this prototype:

```
int S7API lv_Cli_DBGet(S7Object Client, int DBNumber,
    PLVString *pStringData, int *SizeGet);
```

The first two parameters are the same of Cli_DBGet().

PLVString is defined as follow:

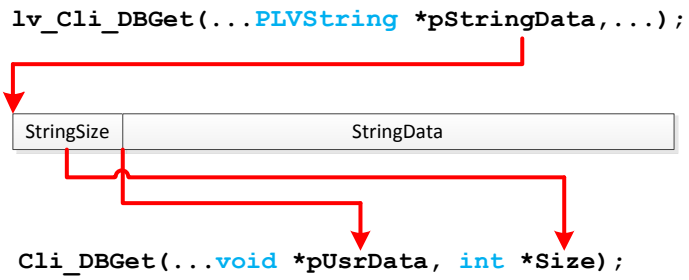
```
typedef struct {
   int32_t size;     // Block Size
   byte Data[1];     // Data
}
```

(byte is a portable 32/64 bit "byte" defined in snap7.h).

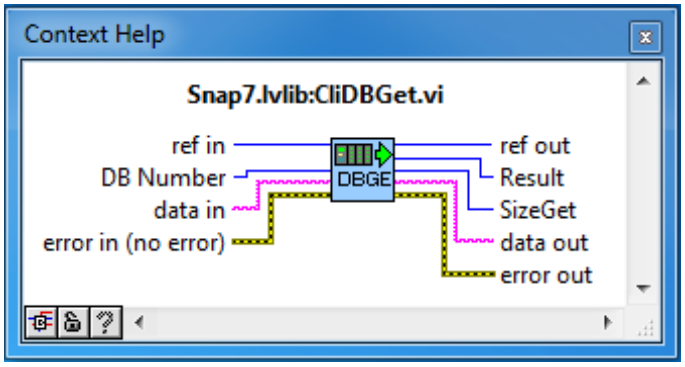
This is the body of the adapter.

```
int S7API lv_Cli_DBGet(S7Object Client, int DBNumber,
  PLVString *pStringData, int &SizeGet)
{
    int32_t Size = *pint32_t(*pStringData); // String size
    pbyte pUsrData=pbyte(*pStringData) + sizeof(int32_t);
    int Result=Cli_DBGet(Client, DBNumber, pUsrData, &Size);
    SizeGet=Size;
    return Result;
// Note : the buffer size check is performed into Cli_DBGet
}
```

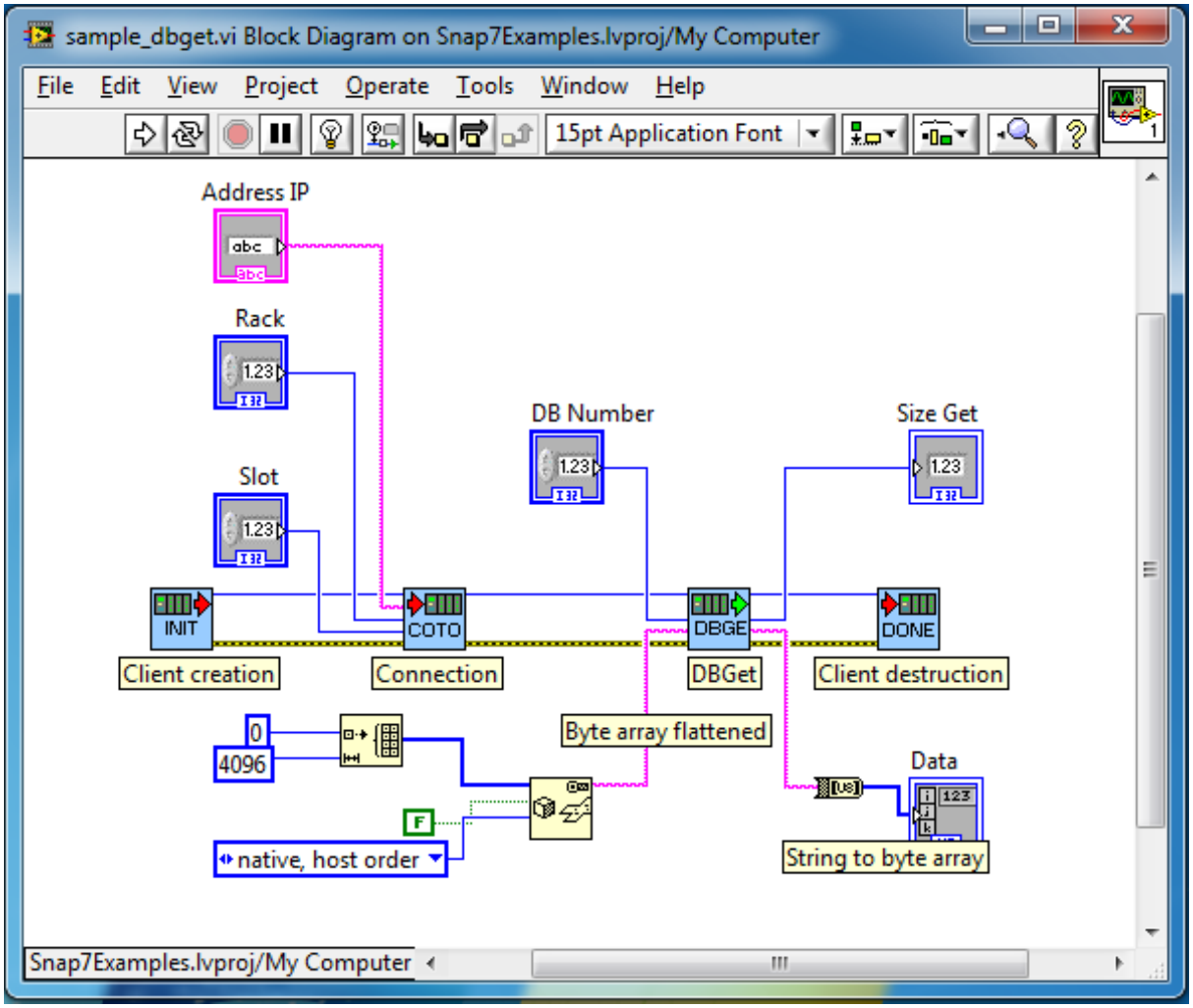
That implements this concept:



On the LabVIEW side the vi **CliDBGet.vi** is defined as follow:



Finally, a very minimalist (but working) program to read a DB in a 4K buffer :

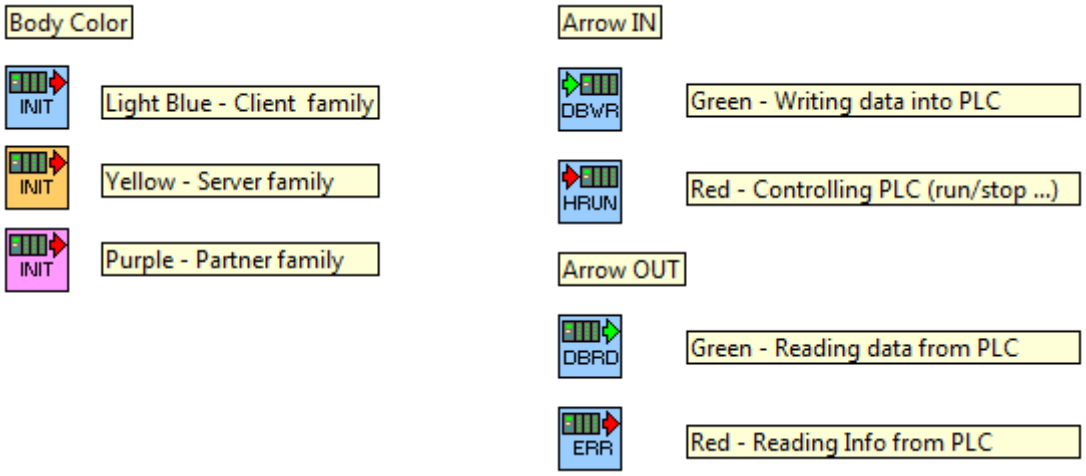


The upper wire across all blocks is the Client reference generated by CliCreate, internally it's a intptr_t, externally is stored into a 64 bit integer (for using in 64 bit architectures).

Conventions

Graphic

Surely Snap7 VI icons will not be exposed to the New York Museum of Modern Art, but they follow a useful convention that helps to identify them at a glance.



Naming

As said, all the VIs access to the Snap7 through the LV interface library, so for each function we have three entity :

1. VI name
2. LV library function name
3. Snap7 function name

They are linked following this rule :

VI name	**<object><Function name>**
LV function	**lv_<object>_<Function name>**
Snap7 function	**<object>_<Function name>**

Example

VI name	**SrvRegisterArea**
LV function	**lv_Srv_RegisterArea**
Snap7 function	**Srv_RegisterArea**

Release

Everything you need is stored into LabVIEW folder that is divided as follow:

[\Examples] contains a LabVIEW project which groups many examples. They are further divided into three folders (\Client, \Server and \Partner) and are autonomous, i.e. you can run them without loading the project.

[\lib] that contains the library Snap7.lvlib and all the interface vi.

[\lib\windows] contains lv_snap7.dll and snap7.dll, they are the deploy libraries.

[\lib\win32] contains 32 bit version of lv_snap7.dll and snap7.dll, they are the build libraries (see LabVIEW_32.bat).

[\lib\win64] contains 64 bit version of lv_snap7.dll and snap7.dll, they are the build libraries (see LabVIEW_64.bat).

[\lib_build] contains three projects to compile lv_snap7.dll.

[\lib_build\VS2012_LV] Visual Studio 2012 solution.

[\lib_build\MinGW32] MinGW32 makefile and batch file for 32 bit.

[\lib_build\MinGW64] MinGW64 makefile and batch file for 64 bit.

[\lib_src] contains the source files of lv_snap7.dll.

[\lib_tmp] contains temporary compilation files (can be safety emptied).

LabVIEW 32 bit and LabVIEW 64 bit (native) use different library models, please follow these rules:

- If you plan to use Snap7 in 32 or 64 bit systems with 32 bit LabVIEW run **LabVIEW_32.bat** before opening any project.
- If you plan to use Snap7 in 64 bit systems with 64 bit LabVIEW run **LabVIEW_64.bat** before opening any project.

These batch files merely copy lv_snap7.dll and snap7.dll from the platform folder (win32 or win 64) to the deploy folder (windows).

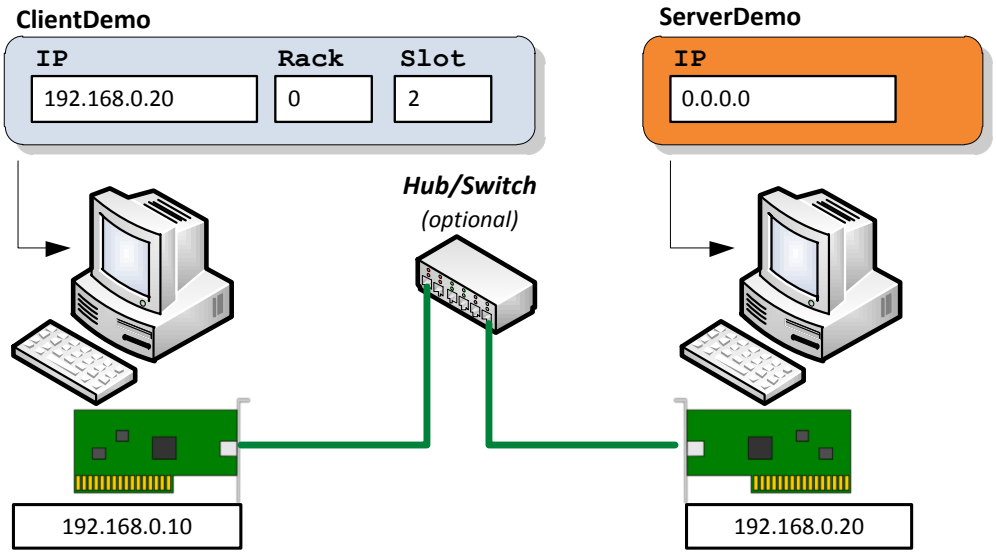
Final remarks

- All the Snap7 blocks are **thread safe.**
- All the Snap7 blocks, in which a string handle is passed, check the string size against the Size parameter passed to avoid program crash. If the string size is less than the size param, the latter is trimmed, the function is performed but an error of partial data read or write is produced.
- If you need to rebuild lv_snap7.dll and snap7.dll **use the same c++ compiler for both if you plan to use them in 64 bit environments.**
  There is still no unified ABI convention for 64 bit systems, the problem is not the dll itself, but the .lib file needed to link them.
- For lv_snap7.dll are valid all concepts exposed in "Rebuild Snap7".
- **snap7.dll** must reside in the same folder of **lv_snap7.dll**, and their architecture must match (32/64 bit).

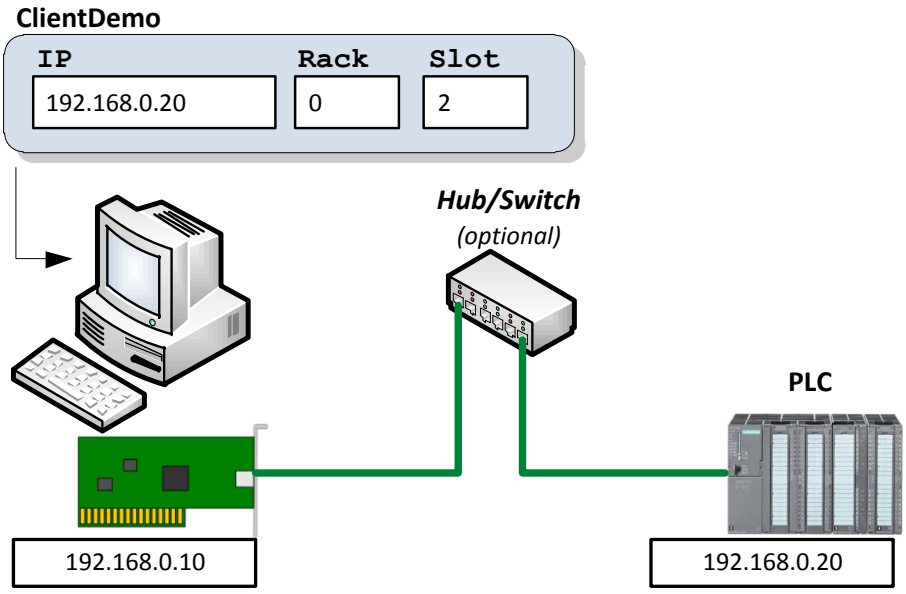
Testing Snap7

Here we are supposing to use the rich demos provided, but also the examples can be suitable for this task.

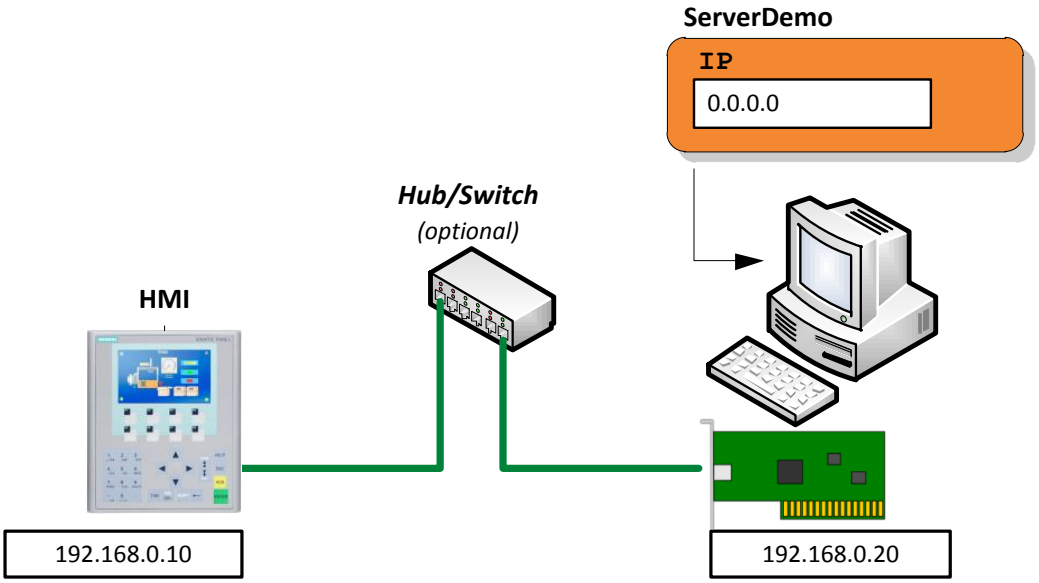
Testing the Client and the Server using two PC.



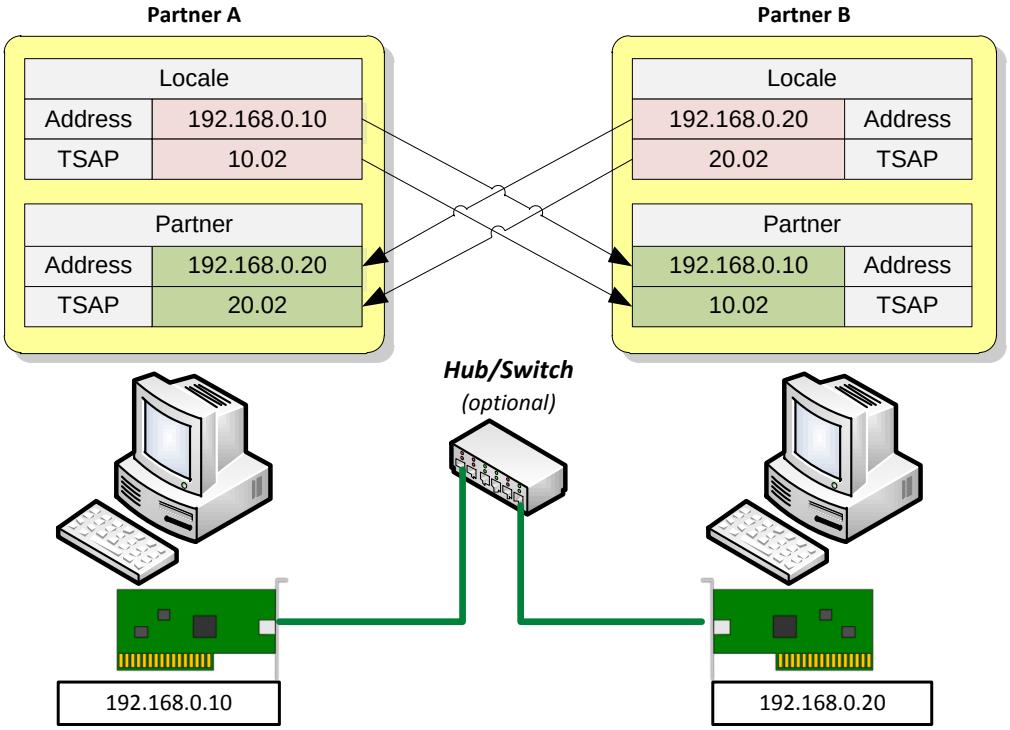
Testing the Client using a PC and a PLC.



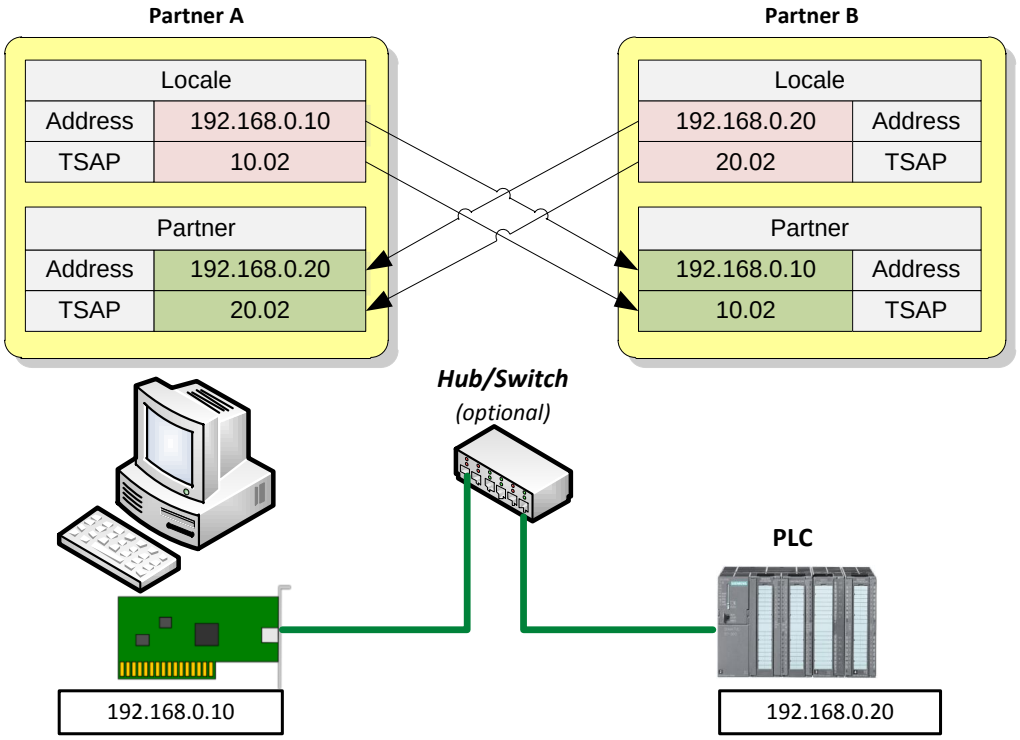
Testing the Server using a PC and an HMI.



Testing the Partner using two PC.



Testing the Partner using a PC and a PLC.



Remarks

Be very careful using clientdemo with production machines.

Snap7 source code

Snap7 is written in ISO C++.

The source code is split into three folders.

sys

Contains base classes, the socket communication layer, the threads layer and some platform dependent files, the files are named *snap_xxxxx*.

These files are the lower layer of the **Snap Project**, Snap7 is the S7 implementation, SnapModbus … is coming soon.

core

Contains all files related to Snap7, the implementation of the IsoTCP and S7 protocol.

lib

Contains the interface files for creating the library.

There is a reason for this subdivision

- If you want to write a multiplatform Ethernet packet driver, you can use only the **sys** folder.
- If you want to embed Snap7 into your source code, include all but the **lib** folder.

Style

Snap7 is written in "C with objects" with two level of dependency.

The lowest allows to embed the **Snap7MicroClient** into your program. It was designed to be as much as possible friend of the "Embedded C++".

No STL, exceptions, dynamic memory or threads are used, The **virtual** directive of some members can be deleted.

The resulting code should be "romable".

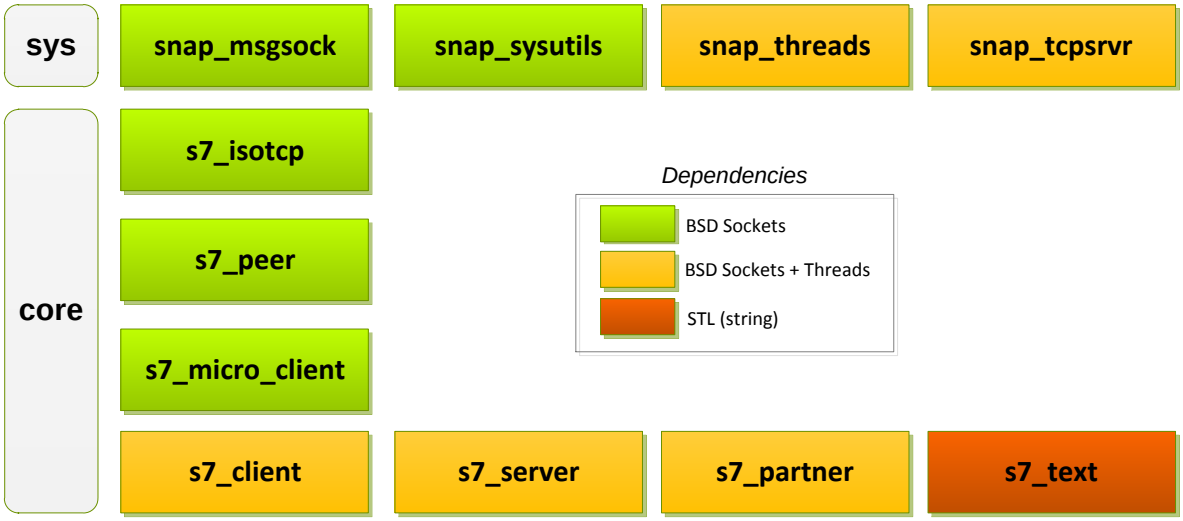
Next paragraph will explain how to embed the micro client.

Embedding Snap7MicroClient

As said, Snap7Microclient is not accessible from the outside of the library.

To embed it into your source code you need to include into your project these files:

- sys\snap_msgsock.cpp
- sys\snap_sysutils.cpp
- core\s7_isotcp.cpp
- core\s7_peer.cpp
- core\s7_micro_client.cpp



Rebuild Snap7

Due to its design (32 and native 64 bit) to rebuild the source code you need of a C99+ compiler, since *stdint.h* is **strictly required**.

Windows

Normally you do not need to recompile the libraries unless you modified the source code.

The **32** bit release already works in any version of Windows, either 32 or 64 bit, desktop or server, starting from Windows **NT4 SP6** up to **Windows 8**.

The **64** bit release only works in all native 64 bit platforms, including the little-known Windows XP 64 Professional. To link them, your software itself must be native 64 bit.

In the folder **build\windows** there are 5 solutions ready to run, *Host platform* indicates where the compiler runs, *and Target platform* indicates where the library produced runs:

	Host Platform	Target Platform	Notes
MinGW32 – 4.7.2	Win32/Win64	Win32/Win64	1
MinGW64 – 4.7.1	Win32/Win64	Win64	
VS2008	Win32/Win64	Win32/Win64	2
VS2010	Win32/Win64	Win32/Win64	2
VS2012 (upd.2)	Win32/Win64	Win32/Win64	2,3

1) This compiler is the only one compatible with Windows NT 4.0 /Windows 2000
2) Express release needs **Windows Software Development Kit (SDK)** to compile 64 bit applications.
3) Windows Vista 32 bit is the "oldest" platform supported by VS2012 without update 2.

Most likely **Wine** too is a suitable host platform for them, but currently is untested.

MinGW 32bit 4.7.2

It is assumed that **MinGW** compiler is installed in C:\MinGW32 and its release is 4.7.2.
If not, you need to modify make.bat and makefile.

- In **make.bat** change the path instruction in the first line to point to the correct compiler path.
- In **makefile** change the vars **MINGW** and **MINREL** (path and release).
  Pay attention to not leave trailing spaces after the text.

If you don't have it at all:

Go to
http://sourceforge.net/projects/orwelldevcpp/files/Compilers/MinGW/

download **MinGW 4.7.2.7z** and unpack it in c:\ .

No further settings are needed.

To build Snap7, open a command prompt into the working folder **build\windows\MinGW32** and run "make all" (or "make clean" if you want to clean the project).

Into **build\bin\win32** you will find **snap7.dll** and **snap7.lib**, the latter is the dynamic library import file to be used with C/C++ compilers (other languages don't need it).

Libstdc++ are statically linked, so you don't need to distribute them with your software.

Remarks

This compiler is the only one that supports Windows NT 4.0/Windows 2000.

MinGW 64 bit 4.7.1

It is assumed that **MinGW** compiler is installed in C:\MinGW64 and its release is 4.7.1.
If not, you need to modify make.bat and makefile.

- In **make.bat** change the path instruction in the first line to point to the correct compiler path.
- In **makefile** change the vars **MINGW** and **MINREL** (path and release).
  Pay attention to not leave trailing spaces after the text.

If you don't have it at all:

Go to
http://sourceforge.net/projects/orwelldevcpp/files/Compilers/TDM-GCC/

download **TDM-GCC 4.7.1 (4.7.1-tdm64-3).7z** and unpack it in c:\ .

No further settings are needed.

To build Snap7, open a command prompt into the working folder **build\windows\MinGW64** and run "make all" (or "make clean" if you want to clean the project).

Into **build\bin\win64** you will find **snap7.dll** and **snap7.lib**, the latter is the dynamic library import file to be used with C/C++ compilers (other languages don't need it).

Libstdc++ are statically linked, so you don't need to distribute them with your software.

Microsoft Visual Studio

There are three solutions ready into the folders **build\windows**, open them with the IDE of Visual Studio, choose **Win32** or **x64** in the combo box of the platform and build the solution.

Base libraries are statically linked, you don't need to distribute *Microsoft Visual C++ 20xx Redistributable* with your application.

Into **build\bin\win32** or **build\bin\win64** you will find **snap7.dll** and **snap7.lib**, the latter is the dynamic library import file to be used with C/C++ compilers (other languages don't need it).

Visual Studio 2012

All 32 bit DLLs (not only the snap7 one) generated by this compiler **don't work** in Windows NT4.0/ 2000 / XP.
This because an (unrequested) reference to the **GetTickCount64** function is generated.

This function is present in kernel32.dll starting from Vista.

To produce DLL compatible with Windows XP you need to install the **Update 2** and choose into the project options **Visual Studio 2012 - Windows XP (v110_xp)** as platform set.

Visual Studio 2008 should not be suitable since it is not compliant with C99. I made a small patch in snap7_platform.h to include it since it is a very largely used compiler.

Visual Studio Express (2008-2012) is suitable but you need to download **Windows Software Development Kit (SDK)** to compile the 64 bit release of the library.

Finally, if you need the compatibility with NT 4.0 and Windows 2000 family, these compilers cannot be used at all.

Embarcadero C++ builder

Snap7 is compiled fine by this compiler but there is an issue : all the dynamic library import files generated (.lib) are not compatible with all the other Windows C/C++ compilers that use the Microsoft standard, included MinGW, and vice-versa.

Microsoft lib files are in **coff** format.
Borland lib files are in **omf** format.

In any case :

- In the **bin** folder of C++Builder there is the utility **coff2omf.exe** that converts from coff to omf.
- In the Microsoft **masm** package (V8+) there is the utility omf2coff.exe that converts omf to coff.

This issue only affects the *.lib* file, the dll generated works fine.

Finally, to produce 64 bit libraries, you need at least **Embarcadero C++ Builder XE3 upd.1**.

Currently there is no ready project for this compiler.

Unix

The libraries rebuilding is normal under Unix even though you don't modify the source code because **Libgcc** (the only dependence of snap7) changes accordingly to the OS/distribution.

If you experience an error regarding Libgcc or other libraries version with libsnap7.so released, **don't try strange alchemy** (such as downloading them), Snap7 relies only on system libraries, your OS may malfunction if you replace them.

You have the full source code of Snap7 and to rebuild it is a very simple task, since all makefiles are ready.

The only requirement is that **GNU ToolChain** (g++ and Make) is present, and this is true for the main OS and distributions : BSD, Solaris, Linux (Debian, Ubuntu, Red Hat, SlackWare, etc.), even small ARM cards that I tested had them.

Otherwise, you need to install them using the current package manager of your OS (apt, pkgtool, Yum, etc).

Anyhow, if you type "GNU ToolChain" followed by your OS name in Google, surely you will find very detailed information.

To know if the GNU Toolchain is correctly installed, open a terminal and enter:

`g++ --version` and
`make --version`.

You should see the compiler and make utility release.

Linux x86/x64

Open a terminal and go to **build/unix**, there type :

make –f <architecture>_linux.mk all to rebuild the library

make –f <architecture>_linux.mk clean to clean the project

make –f <architecture>_linux.mk install to rebuild and copy the library in **usr/lib**

for the third option you need be root or use **sudo make** …

Where <architecture> can be **i386** or **x86_64.**

Let's suppose that you have a 32 bit release of Ubuntu, you must type:

```
make –f i386_linux.mk all
```

In the folder **bin/<architecture>-linux/** you will find **libsnap7.so.**

You need to copy it into /usr/lib or set accordingly **LD_PATH_LIBRARY**.

Warning

Some 64 bit distributions (as CentOS and Red Hat) need libsnap7.so into **/usr/lib64** instead of /usr/lib.

Linux Arm boards

There are two makefiles ready to use with **single/dual/quad-core** ARM boards.

- **arm_v6_linux.mk** for V6 ARMHF boards (like Raspberry PI).
- **arm_v7_linux.mk** for V7 ARMHF boards (like BeagleBone, pcDuino, Cubieboard 2 and UDOO).

(They are a bit different, the second one needs -mword-relocations)

As usually, in **build/unix** , type :

```
make –f arm_v6_linux.mk all (or clean or install)
```

or

```
make –f arm_v7_linux.mk all (or clean or install)
```

In the folder **bin/arm_vX-linux/** you will find **libsnap7.so.**

You need to copy it into /usr/lib or set accordingly **LD_PATH_LIBRARY.**

Remarks

The build process is quite slow on these boards cause the switch **–pedantic** that ensures the ISO compliance. You can disable it temporarily to speed up the process.

BSD

The working folder is, as usual, **build/unix**, but the command must be:

gmake –f <architecture>_bsd.mk all (or clean or install).

Where <architecture> can be **i386** or **x86_64.**

In the folder **bin/<architecture>-bsd/** you will find **libsnap7.so.**

You need to copy it into /usr/lib or set accordingly **LD_PATH_LIBRARY.**

Oracle Solaris 11

Solaris 11 has an excellent native IDE and compiler, but you must be aware of some issues related to it.

If you want to produce a shared library and use Oracle Solaris Studio, you **must** set GNU as Tool Collection in "project properties" (NetBeans do it by default).

You can use Solaris studio with "OracleSolarisStudio" compiler only if you are embedding the library source code into your program, otherwise it **doesn't work**.

And, in any case, remember to include "Socket and Network Service Library" and "Solaris Threads" as libraries (in linker options), **don't use pthreads**, snap7 provides direct support for native Solaris threads.

(i.e. -lsocket -lnsl -lthread)

Or, if you like a quiet life, use the makefile provided and simply type:

```
gmake –f i386_solaris.mk all (or clean or install)
```

or

```
gmake –f x86_64_solaris.mk all (or clean or install)
```

The correct native Solaris libraries are already set into the makefile.

As usual, in the folder **/bin/<architecture>-solaris/** you will find **libsnap7.so.**

Remarks

Neither Spark architecture nor X86_64 were tested. Please report feedbacks if you do it.

Apple OSX

Apple OSX derived from BSD i.e. it belongs to the Unix family, however there is a separated folder to build it.

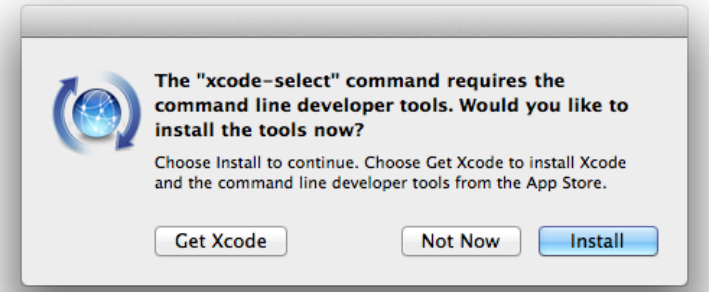
To rebuild Snap7, Xcode command line tools are needed.

From Xcode 4.3 they are not automatically installed with the main software and you need to download them.

It's very simple, open a terminal and type:

xcode-select --install

In the next window press Install button. That's all.



Open a terminal and go to **build/osx**, there type :

`make -f <architecture>_osx.mk all` to rebuild the library

`make -f <architecture>_osx.mk clean` to clean the project

`make -f <architecture>_osx.mk install` to rebuild and copy the library in **usr/lib**

for the third option you need be root or use **`sudo make`** …

Where <architecture> can be **i386** or **x86_64.**

In the folder **/build/bin/<architecture>-osx/** you will find **libsnap7.so.**

You need to copy it into /usr/lib or set accordingly **LD_PATH_LIBRARY**.