INFRANOR

# Gem Drive Studio

## Commissioning manual of the Communication Server

Issue : **1.2**

# Content

# 1. Introduction

PC softwares developed for the parameterization of INFRANOR's latest generation drives are based on a modular client/server architecture.

One of the interests of this architecture is the possibility to develop independent client modules using the same server.

The purpose of the Communication Server module is to manage the physical accesses to the fieldbus by adapting the communication protocol to the various peripherals (CANopen, RS232, ...).

This manual provides all information allowing users to develop their own client module by using the Communication Server module.

# 2. Minimum configuration

The server module has to be installed on a PC with following minimum configuration:

- Processor: 1 GHz,
- RAM: 512 MB,
- Operating System: Windows© XP (Service Pack 2) or Server 2003
- Microsoft .NET Framework V3.0 (or higher) installed.

# 3. Installation of the server module

The installation set contains:

- The installer of the server.
- A client model (source code) written in VB.NET.
- A documentation.

Unzip the ZIP file and launch the Setup.exe file.
Follow the procedure to install the server on the PC.

The sources of the client model can be edited under the Microsoft Visual Studio environment. A free express version is available on the Microsoft website for download.

During the installation of the server on the PC, four keys are created in the Windows base register:

<u>One key containing the name of the executable allowing to launch the server</u>:

Key: HKEY_USER
Subkey: Default\Software\DriveServer\Infos
Name of the key: DriveServerName

<u>One key containing the access path to this executable</u>:

Key: HKEY_USER
Subkey: Default\Software\DriveServer\Infos
Name of the key: DriveServerPath

One key containing the server version:

Key: HKEY_USER
Subkey: Default\Software\DriveServer\Infos
Name of the key: DriveServerVersion

One key containing the access path to the object dictionary files:

Key: HKEY_USER
Subkey: Default\Software\DriveServer\Infos
Name of the key: DictionariesPath

The client module can use these keys later for launching/stopping the server.

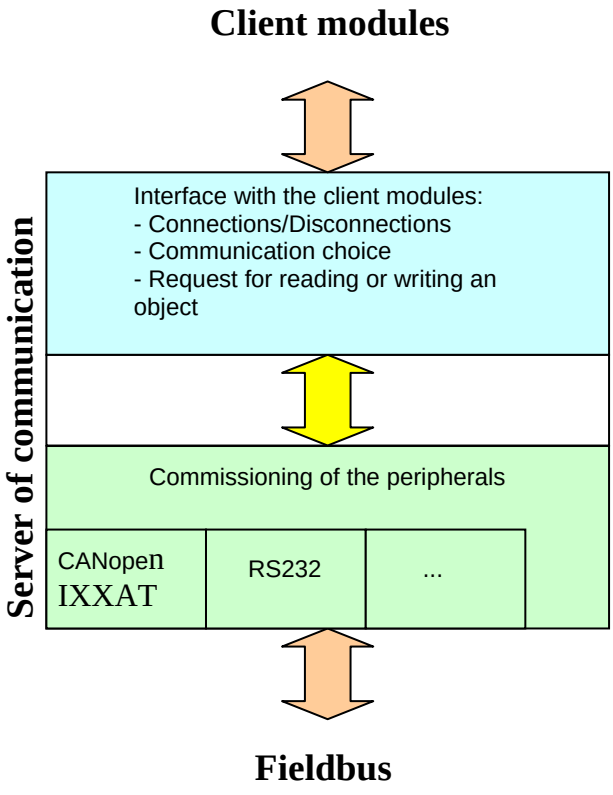
# 4. Description of the server

## 4.1. Architecture

The server consists of:

- a part dedicated to the communication between server and client modules,

- a part managing the hardware peripherals (RS232, CANopen1, CANopen2, ...).



The whole management of the hardware peripherals is made by the server. So, any new peripheral implemented on the server will not require the recompiling of the client modules.

Adding a new communication channel will involve the development of a new module internal to the server as well as an incrementation of the server version, ensuring the compatibility with all former versions.

The server remains independent from the data types. For each parameter (CAN object), the server reads or writes a sequence of bytes on the selected peripheral and the calling client will have to format the data according to the object type. The information regarding the type is given by the object dictionary corresponding to the drive model and version.

## 4.2. Environment of the development

The communication server has been developed under .NET environment in order to offer maximum compatibility with the present systems.

The client modules can be written by using one of the NET languages (C#, VB.NET, C++, Delphi.NET....).

The communication between communication server and modules is based on the WCF technology or on Windows Communication Foundation. This technology appeared with version 3.0 of the Framework .NET.

WCF provides a unified programming model allowing to build distributed applications.

This technology is based on 3 important elements:

A/ The service

A WCFservice is a software unit implementing a contract. These services are listed in an interface (in the meaning of object) which integrates the list of proposed operations by each service (service contract).

B/ The address

The address is just a URL defining the service location.

C/ The protocol or "binding"

This is the method used for communicating with the WCF service. Framework .NET 3.0 proposes 9 various "bindings":

- BasicHttpBinding
- WSHttpBinding
- WSDualHttpBinding
- WSFederationHttpBinding
- NetTcpBinding
- NetNamedPipeBinding
- NetMsmqBinding
- NetPeerTcpBinding
- MsmqIntegrationBinding

## 4.3. Required files

### 4.3.1. Installed files

The services developed in the communication server are compiled in a DLL library file and proposed via an interface (**DriveInterface.dll**).

The services of the communication server requires a host on which they can be executed. This is the purpose of the **DriveHost.exe** executable.

In order to develop a client module and to use the server services, a reference must be added to the DriveInterface.dll file.
The server is started by launching the executable file DriveHost.exe.

### 4.3.2. Configuration files of server and client

The configuration of a Windows Communication Foundation (WCF) service with a configuration file allows to provide termination point and service behaviour data at the moment of the execution rather than at the compiling.

These files must be located in the same directory as the executable. They allow to define the service termination points as well as their behaviour.

**Client side:**

The configuration file (app.config) must be located in the same directory as the executable.
Example of its content:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="longTimeoutBinding"
                 receiveTimeout="infinite"
                 sendTimeout="infinite">
          <reliableSession enabled ="true" inactivityTimeout ="infinite"
ordered="false"/>
          <security mode="None"/>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint
        address="net.tcp://localhost:8018/AllDriveServices"
        binding="netTcpBinding"
        bindingConfiguration="longTimeoutBinding"
        contract="DriveInterface.IDriveService"
        name="NetTcpBinding_IDriveService" />
    </client>
  </system.serviceModel>
</configuration>
```

**Server side:**

The configuration file is installed in the same directory as the DriveHost.exe file.
Its content is the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="longTimeoutBinding"
                 receiveTimeout="infinite"
                 sendTimeout="infinite">
          <reliableSession enabled ="true" inactivityTimeout ="infinite"
ordered="false"/>
          <security mode="None"/>
        </binding>
      </netTcpBinding>
    </bindings>
    <services>
      <service name="DriveService.AllDriveServices">
        <endpoint
            address="net.tcp://localhost:8018/AllDriveServices"
            bindingConfiguration="longTimeoutBinding"
            binding="netTcpBinding"
            contract="DriveInterface.IDriveService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

In this example, clients will connect to the address `net.tcp://localhost:8018/AllDriveServices` for using the services of the server and dialogue by using TCP sockets.

Configuration files can be modified. For example, it is possible to add <endpoint> tags with various addresses and different bindings.

So, when adding the following lines into the App.config file, the service will also be available in http:

```
<endpoint address="http:// localhost:8018/AllDriveServices "
binding="basicHttpBinding" contract=" DriveInterface.IDriveService "/>
```

# 5. Function list

## Fct_AddClientReference

Description

This function allows to reference a client module on the server.

Input parameters

Use of the Callbacks (boolean).

Output parameters

None.

Feedback value

- Client number ("Short" type).

Note: This number must be used later for all exchanges with the server.

## Fct_RemoveClientReference

Description

This function allows to unreference the client module of the server.

Input parameters

- Client number ("Short" type).

Output parameters

None.

Feedback value

- Number of clients still referenced on the server ("Byte" type).

## Fct_ReadServerFeatures

Description

This function allows to know the list of functionalities implemented on the communication server.

Input parameters

None.

Output parameters

None.

Feedback value

- 32 bit word. Each bit represents a supported functionality ("UInteger" type)

The summary below gives the list of the integrated functions for the various server versions. If the corresponding bit is set at 1, then the function is implemented.

Version 1.0 (first version)
Bit 0: Basic simulator (reading only)

## Fct_CommConfig

Description

This function allows to configure the communication by displaying a window grouping the various available peripherals.

Input parameters

- Client number ("Short" type)

Output parameters

- Selected communication type ("SByte" type):
    0: Serial link
    1: CANopen (IXXAT peripherals)

- Name of the selected peripheral ("String" type)

- Communication speed ("UInteger" type)

Feedback value

- Communication status ("SByte" type):

    0: Peripheral stopped
    1: Peripheral started
    -1: Peripheral faulty

## Sub_ReadCommConfig

Description
This function allows to read the communication status. The returned information allow to avoid calling the Fct_CommConfig function if a peripheral is already started and the client wants to remain on the same peripheral.

Input parameters

- None.

Output parameters

- Selected communication type ("SByte" type):
    0: Serial link
    1: CANopen (IXXAT peripherals)

- Selected peripheral type "(String" type)

- Communication speed ("UInteger" type)

- Communication status ("SByte" type):

    0: Peripheral stopped
    1: Peripheral started
    -1: Peripheral faulty

## Sub_StopPeripheral

Description

This function stops the started peripheral.

Input parameters

- None.

Output parameters

- None.

## Fct_ReadPort

Description

This function allows to read a parameter (CANopen object) on one of the drives connected to the fieldbus.

Input parameters

- Client number ("Short" type)

- Node number (CAN address) of the drive ("Short" type)

- Parameter index in hexadecimal ("String" type)

- Parameter sub-index in hexadecimal ("String" type)

- Timeout value in seconds (optional, value = 1 by default) ("Short" type)

Output parameters

- Byte table containing the read value ("Byte" type)

Feedback value

- Command status ("Integer" type):

    - 1: Feedback OK
    - 2: Server busy (in this case, renew the request)
    - 3: Faulty feedback
    - Other values: "Abort" code (see CANopen Communication Profile DS-301).

### Fct_WritePort

Description

This function allows to read a parameter (CANopen objetc) on one of the drives connected to the fieldbus.

Input parameters

- Client number ("Short" type)

- Node number (CAN address) of the drive ("Short" type)

- Parameter index in hexadecimal ("String" type)

- Parameter sub-index in hexadecimal ("String" type)

- Table containing the bytes to be written ("Byte" type)

- Timeout value in seconds (optional, value = 1 by default) ("Short" type)

Output parameters

- None.

Feedback value ("Integer" type)

- Command status:

    - 1: Feedback OK
    - 2: Server busy (in this case, renew the request)
    - 3: Faulty feedback
    - Other values: "Abort" code (see CANopen Communication Profile DS-301).

# 6. Development of a client module

## 6.1. Pre-requested

The client module must refer to the **DriveInterface.dll** interface file and to the mechanisms of the WCF technology grouped within the space named **System.ServiceModel**.

The DriveInterface.dll file is located in the server installation directory.

## 6.2. Object dictionaries

The object dictionaries allow to recognize the list and features of the parameters belonging to a drive. These dictionaries are managed as versions and can be modified according to the firmware evolutions.
Concretely, an object dictionary is an XML file which contains a header allowing the identification of the versions, followed by a list and description of the parameters.

For each parameter, the object dictionary specifies:

- The index
- The sub-index
- The name
- The type (*)
- The access type (**)
- The limit values
- The possibility or not to map this object into a PDO message
- The behaviour (modification when enabled or disabled, ...)
- The category (parameter of motor, regulator, communication, application, ...)

(*) Possible values:

- 0x02: 8 bit signed whole number (Integer8)
- 0x03: 16 bit signed whole number (Integer16)
- 0x04: 32 bit signed whole number (Integer32)
- 0x05: 8 bit unsigned whole number (Unsigned8)
- 0x06: 16 bit unsigned whole number (Unsigned16)
- 0x07: 32 bit unsigned whole number (Unsigned32)
- 0x09: Character sequence (VisibleString)
- 0x0A: Character (ByteString)

(**) Possible values:

- rw: Accessible in reading or writing
- ro: Accessible in reading only
- wo: Accessible in writing only

In fact, the dictionary contains the information which are normally contained in the EDS (Electronic Data Sheet) file available for any drive running with the CANopen protocol. Information contained in the object dictionary are more complete than those contained in the EDS file.

For each new version of the communication server, the directory containing the object dictionary pertaining to the drives will be updated.
This directory is located in the "Dictionaries" file of the server installation directory.

The name of this directory is given by a key recorded in the Windows base register when installing the server (see chapter "Installation of the server module").

## 6.3. Commissioning of the server

### 6.3.1. Creation of the WCF communication channel

The creation of a communication channel with the server requires the ChannelFactory object. This object allows to generate a class which will authorize a client to send to and/or receive from a service.

This channel can only be defined from the client to the server or in both directions. In this case, it is called "duplex channel". The communication from the server to the client is made by means of "callback" procedures.

The following code (example in VB .NET) allows to create this communication channel between client and server.

Note: This object is an interface type object (service contract).

```
'Normal channel
'Public myChannelFactory As ServiceModel.ChannelFactory(Of
DriveInterface.IDriveService) = Nothing

'Duplex channel (by using the callback procedures)
Public myChannelFactory As ServiceModel.DuplexChannelFactory(Of
DriveInterface.IDriveService) = Nothing

Public myService As DriveInterface.IDriveService

Dim Instance As New InstanceContext(New CbCallbackClass())

myChannelFactory = New DuplexChannelFactory(Of
DriveInterface.IDriveService)(Instance, "NetTcpBinding_IDriveService")

myService = myChannelFactory.CreateChannel()
```

The following code line allows to close the communication channel:

```
myChannelFactory.Abort()
```

### 6.3.2. Launching the server

The first client must launch the server. The actions required for it are the following:

- ⇨ Check whether the server is installed, by verifying the availability of the keys (DriveServerName and/or DriveServerPath) in the base registers.
- ⇨ Check whether the server is already launched, by verifying if its name (see DriveServerName key) is in the list of active Windows processes.
- ⇨ Launch the server (see DriverServerPath key) if not already done.

Reminder: The server is not a Windows application (application with window). It is a DLL which contains procedures called by the clients. Launching the server consists in creating a communication channel (in the meaning of Windows Communication Foundation) between this server and the client modules.

When the server is launched, a new icon is displayed in the notification area of the Windows tool bar (systray). Clicking on this icon displays a menu which allows to stop the server.

### 6.3.3. Connection with the server

When a client is connecting to the server, it references on the server and gets a client number which will be used later in all exchanges between client module and server module.

### 6.3.4. Selection of a communication peripheral

For launching a reading or writing instruction on one of the drives connected to the fieldbus, a communication peripheral must be selected among all available peripherals (RS232, CAN,...).

When the first client is connecting to the server (referencing), then the server automatically starts the last peripheral used, if it is available.

For using another peripheral, it is possible, from the client module, to call a server procedure which opens a configuration window of the communication. In this window, the user can stop the running peripheral, if it has started, and select another peripheral and launch its initialization.

Note: When a user selects a communication peripheral and clicks on the button initializing this peripheral, this one is immediately started by the server. If there are other client modules, those will have to use the same peripheral as long is it will not have been stopped. But it is possible to stop the running peripheral from any client connected to the server.

### 6.3.5. Use

When a client module is connected to the server and a peripheral is started, it is possible to use the functions allowing to read or write parameters in a drive connected to the fieldbus (for more details, see chapter "Function list").

### 6.3.6. Disconnection and stopping of the server

When a client disconnects from the server, it is unreferenced from the server and its client number is released.

Disconnecting from the server does not mean stopping the server. But the disconnection procedure called by the client specifies in return the number of clients remaining connected. If this number is equal to 0, it is recommended to stop the server because the process is not more used.

On server side, if a communication peripheral is started, it is automatically stopped after disconnection of the last client.

For stopping the server, just stop the process under Windows from the keys described above.

### 6.3.7. The callbacks

The callbacks are procedures of the client called by the server.
The setup of these procedures is only effective if the input parameter of the referencing procedure on the server (`Fct_AddClientReference`) is set at 1.

Attention: The client module must mandatorily contain the declaration of these procedures. If one of these procedures is not used, the procedure body can be empty but the procedure will however be declared.

The list of the presently implemented callbacks is the following :

**Sub_ServerNotification**

Description: This procedure is used only for Gem Drive Studio projects. It allows to inform a client module about a project modification.

Input parameters

- Notification type (Short)
    - 0: The notification indicates that the client must store his project configuration
    - 1: The notification indicates that the project file has been modified. The other parameters contain the new project file path and the list of the connected devices.
    - 2: The notification indicates that the client module must stop.

- Access path to the project file (string type)

- NodeID list of the connected devices separated by "#" characters (string type).

Output parameters

- None.

**Sub_ServerCommandProgress**

Description: As soon as a transfer size (reading or writing) exceeds 50 bytes, this procedure is called by the server every second for indicating the total transfer byte number and the number of bytes still to be transferred.

Input parameters

- Number of bytes to be transferred (Integer type)

- Number of bytes remaining for transfer (Integer type)

Output parameters

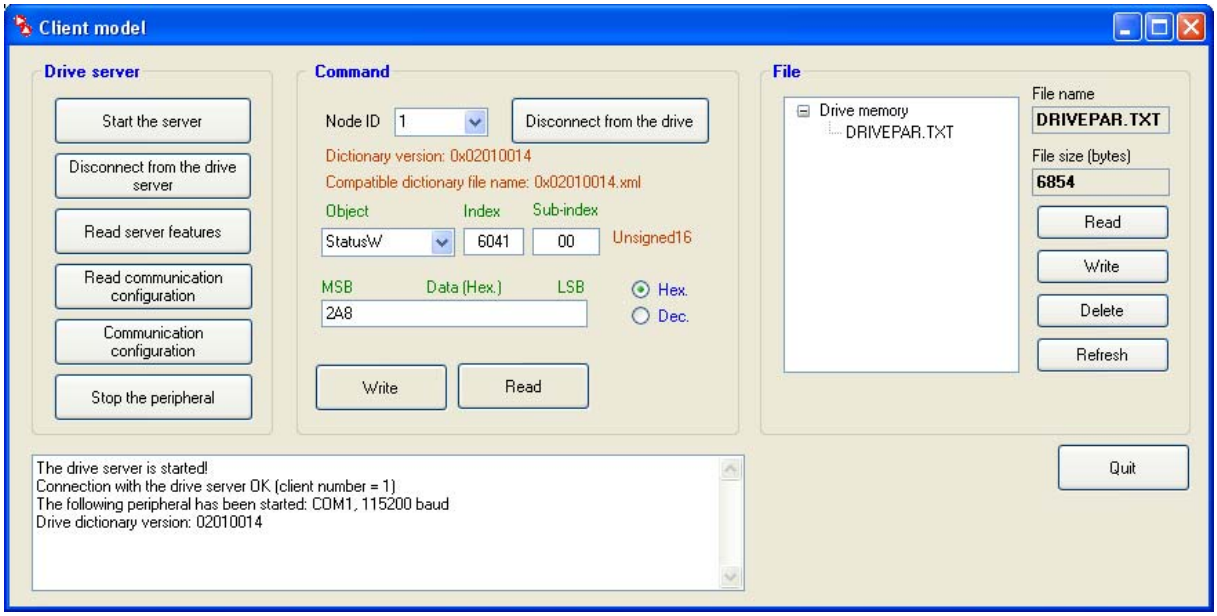
- None.

## 6.4. Example: client model

The installer of the communication server contains a Visual Studio project named ClientModel. According to its name, this project contains the commented source files corresponding to a client module template.

Compiling this project requires to add the **DriveInterface.dll** file in the reference list.
Reminder: this file is located in the installation directory of the communication server.

The main interface contains several buttons allowing to use the various possibilities of the communication server:



Various possible actions:

**A/ Starting the server**

=> Use of the server installation keys for launching/controlling the server process.

**B/ Connection to / disconnection from the server**

=> Creation of the WCF communication channel
=> Client referencing on the server (assignment of a client number).
=> Delisting of the client (release of the client number).

**C/ Selection of the communication peripheral**

=> Display of the configuration window of the communication.
=> Start/Stop of the peripheral.

**D/ Identification of a drive on the bus**

=> Connection to a drive by using its address.
=> Reading of its object dictionary version and association with a dictionary file available in the library.
=> Creation of a CANopen object list from the analysis of the object dictionary.

**E/ Reading/writing of a parameter on a drive**

=> Reading of an object with formatting according to the type
=> Writing of an object

**F/ Getting the file system of a drive working**

=> Reading of the file list available in a drive
=> Reading/Writing/Deletion of a file.

All objects regarding the file system are detailed in the appendix of this document.

# 7. Appendix: File system objects

## 7.1. File name

There are several possible commands on the file contained in the drive. Before each command, the file name must be communicated to the device.

Object: Index 0x5F40, Sub-index 0x0
Type: String
Value: File name in format 8.3 (Example: DRIVEPAR.TXT)

## 7.2. Opening a file for reading

Object: Index 0x5F42, Sub-index 0x1
Type: Integer32
Value to be written: File length in bytes

The use of this object requires to previously specify a file name with object 0x5F40/0x0.

## 7.3. Reading a file

Object: Index 0x5F48, Sub-index 0x0
Type: Unsigned8
Value to be read: Byte array
The use of this object requires to previously open the file with object 0x5F42/0x1

## 7.4. Creating a file

Object: Index 0x5F42, Sub-index 0x2
Type: Integer32
Value to be written: File length in bytes
The use of this object requires to previously specify a file name with object 0x5F40/0x0.

## 7.5. Writing a file

Object: Index 0x5F49, Sub-index 0x0
Type: Unsigned8
Value to be written: Byte array
The use of this object requires to previously create the file with object 0x5F42/0x2

## 7.6. Deleting a file

Object: Index 0x5F42, Sub-index 0x4
Type: Integer16
Value to be written: 0
The use of this object requires to previously specify a file name with object 0x5F40/0x0.

## 7.7. Closing a file

Object: Index 0x5F42, Sub-index 0x3
Type: Integer32
Value to be written: 0
The use of this object requires to previously open the file for reading or for writing.

## 7.8. Searching a file

Object: Index 0x5F42, Sub-index 0x5
Type: Integer16
Value to be read: File status

Return values:

-2 Wrong file name
-1 A file is already opened
0 File not found
1 File found
2 file corrupted
The use of this object requires to previously specify a file name with object 0x5F40/0x0.

## 7.9. File list

Object: Index 0x5F4A, Sub-index 0x0
Type: String
Value to be read: List and size of the files

The returned string contains the list and the size of files contained in the device memory.

Example:

DRIVEPAR.TXT 2621
USER_PAR.TXT 1582
2 file(s).

## 7.10. Return codes

The following return codes concern the opening, closure, reading, writing and deleting commands on the file.

| | | |
|---|---|---|
| 0 | FILE_OK | Command OK |
| -1 | FILE_EXIST | A file already exists with the same name |
| -2 | FILE_ACCESS | Access conflict: A file is already opened |
| -3 | DSK_FULL | Memory is full for writing |
| -4 | FILE_NOTFOUND | File not found |
| -7 | FILE_RDERROR | Reading error |
| -5 | FILE_EOF | End of file |
| -6 | FILE_NOTOPENED | The file is not opened |
| -8 | FILE_WRERROR | Writing error |
| -9 | FILE_OPENED | The file is opened |
| -10 | FILE_NAME | File name error |
| -11 | FILE_TOO_BIG | File size too large |
| -12 | FILE_CRC32 | CRC32 error |