

## Abstract

This document describes the technical aspect of the Driver Runtime task of InduSoft Web Studio (IWS) and a discussion of basic communications concepts.

The Driver Runtime task is responsible for exchanging data with other systems. Other systems could be other software (i.e.,: PC Based Control Software), an external device, such as a PLC, Robot, Remote I/O, Barcode reader, or any other device that supports a protocol that is implemented by the driver.

## Basic Communications Concepts

### Messages, Protocols, Abstraction Layers, and the OSI Model

Driver messages are pieces of structured communications sent over some physical medium, such as a serial connection, Ethernet, Wi-Fi, or even using a proprietary protocol and a specialized port. Messages are encoded with an addressing scheme that a receiving or “**listening**” station recognizes. Messages are constructed in layers called “**abstraction layers**” and each layer consists of a set of functionality providing a needed service to the preceding layer.

Message construction starts in the very highest layer, or the Application Layer, containing the core message to be delivered. Subsequently lower abstraction layers are added to the message until all the layers are encapsulated, ready for the receiving or listening station to understand and disassemble in reverse order. The way that messages are created, layered, and encoded is called the “**Protocol**” or more formally, “**Communications Protocol**”.

A communications protocol is basically a set of rules that end points in a telecommunication connection use when they communicate. This layering of the various functionalities is called the “**Protocol Layers**”. For instance, a common way to view how communications work is to use the “**OSI Model**” consisting of seven abstraction layers: Application, Presentation, Session, Transport, Network, Data Link, and Physical. Every communications protocol Definition Specification outlines the content or functionality that will be in each abstraction layer of that protocol (**Figure 1**)<sup>i</sup>.

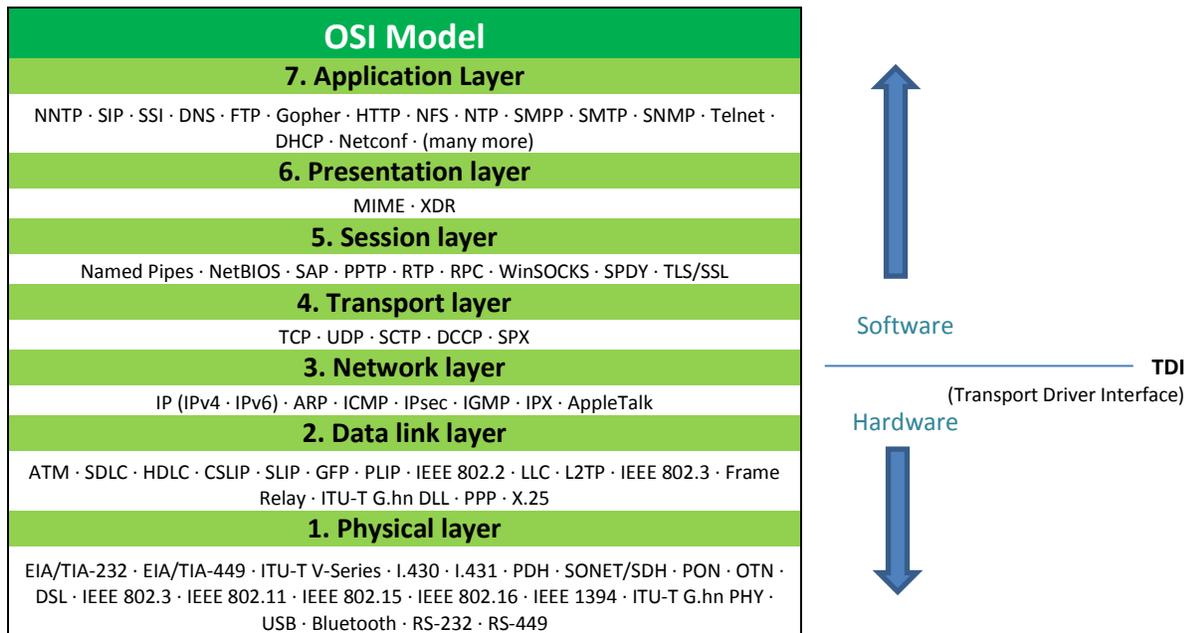


Figure 1: OSI Model and example items that could appear in the abstraction layers for any given communications protocol

When one node (source) sends a message to another node (target), it encapsulates the message elements through all the layers of the communications protocol specification. When the message arrives at the target node, it is de-encapsulated in the inverse order of layers (**Figure 2**) until the source or application message is finally uncovered. Each abstraction layer in the Target Node assumes that it is getting its information from, or communicating only with, an identical abstraction layer on the Source Node.

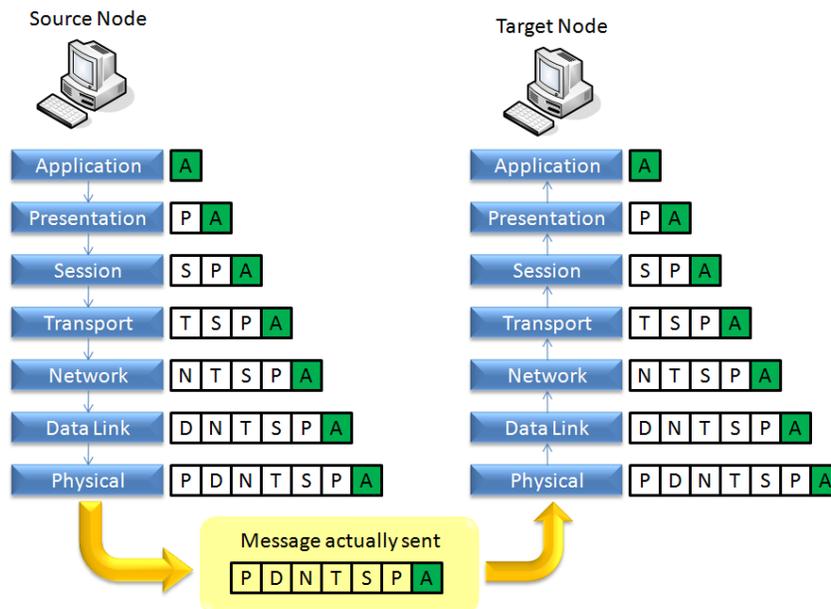


Figure 2: Creation of a message for any given communications protocol using the OSI Model

Each abstraction layer in every unique protocol is responsible for executing its functions and for integrating properly with its adjacent layers. It is important to note that there are standard protocols for some layers, such as “TCP” and “UDP”, which are “Transport Layer” protocols. Therefore, different protocols on the higher layers (Application, Presentation, and Session) can share the same transport layer when communicating data.

For example, both the InduSoft **MOTCP** driver, which implements the “Modbus using TCP” protocol, as well as the InduSoft **ABTCP** driver, which implements the “DF1 using TCP” protocol, share the TCP transport abstraction layer. The differences between the protocols implemented by these two drivers are only in the higher abstraction layers (Application, Presentation, and Session). The operating systems itself, along with corresponding device drivers for the network adapters with the physical hardware, implements the lower layers of these protocols. By adopting this concept of layers as defined in the OSI Model, any given Communications Protocol can be defined by a set of Protocol Layers.

## InduSoft Web Studio Internal Structure

The InduSoft Web Studio runtime (StudioManager.exe) has two main processes:

- **Studio Manager.exe:** Multi-thread process, which manages most of the runtime tasks.
- **Viewer.exe/ISSymbol.ocx:** Single-thread process, which manages the runtime graphic interface (screens) and exchanges data with Studio Manager via the TCP/IP Server task. The link between Viewer and Studio Manager is implemented over TCP/IP. Therefore, the Viewer module can run either on the same station where Studio Manager is running, on a remote station, or in an Internet Explorer Web Browser (hosting the ISSymbol.ocx plug-in), providing a wide level of flexibility and expansibility for the graphic interface (**Figure 3**).

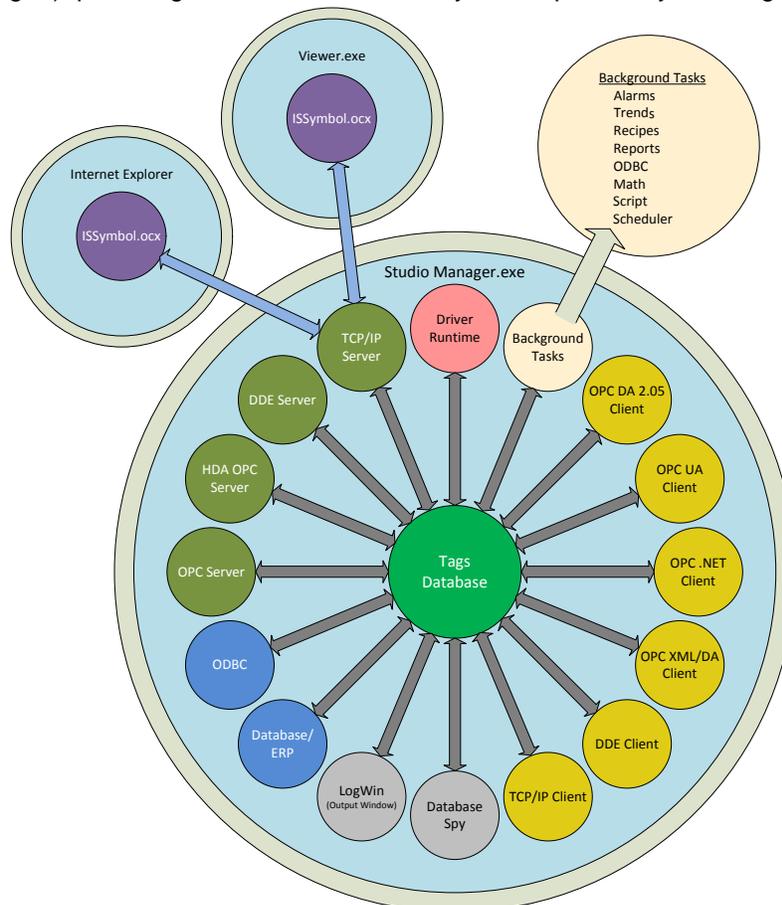


Figure 3: InduSoft Web Studio Internal Architecture

The Tags Database is the core of the IWS runtime engine. It keeps the current value of all tags configured in the application. When any task changes a tag value, that task sends a message to the Tags Database updating the tag value kept there. The Tags Database then sends a message to any other task that is using the tag to also update the current value to what is now stored in the Tags Database.

Using this scheme, the Tags Database becomes the main gateway keeping all tasks synchronized. Furthermore, the communication between the Tags Database and its tasks is by exception – only when a tag changes value. There is no polling involved. By utilizing this control method, internal synchronization and increased performance are then based on a solid and robust platform.

For example, when the Driver Runtime Task reads a new value from a device such as a PLC, it sends a message to the Tags Database. The Tags Database then sends a message to all other tasks that are using that specific tag at that moment, for example, the Viewer Task. When the Viewer receives the value from the Tags Database (through the TCP/IP Server task), it updates the objects on the open screens that are using the tag.

Even though the Viewer is an independent process from Studio Manager.exe, it works just like any other task (thread) of Studio Manager during Runtime. The only peculiarity of the Viewer module is the fact that it communicates with the Tags Database through the TCP/IP Server task, making the Viewer modular and flexible for distributed architectures.

It is important to emphasize the fact that the execution of all tasks is asynchronous and that the operating system executes them in a multi-tasking fashion. Even though the sequence of execution for the tasks is not preemptive, the multi-tasking nature of the operating system, allied with various IWS internal mechanisms, ensures optimal performance during the runtime. The execution of one task does not decrease the performance of the others.

## Driver Runtime Configuration

Each communication driver has its own syntax for station and register addressing. Some communication drivers may have protocol-specific parameters. However, all communication drivers supported by IWS share the same configuration interfaces (**Figure 4**).

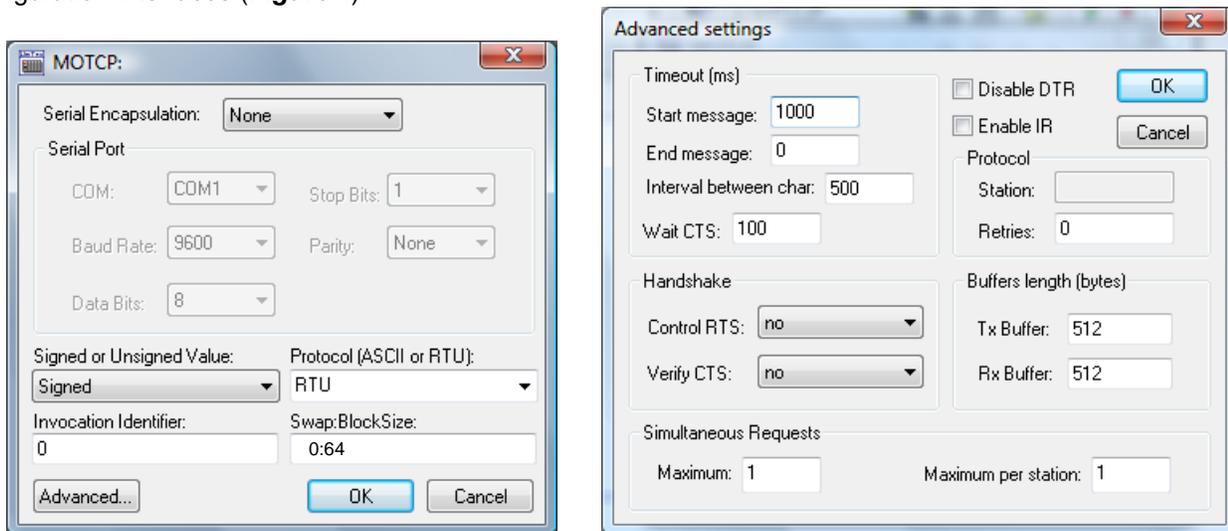


Figure 4: Driver settings dialogs-- Main and Advanced

InduSoft Web Studio provides the following basic interfaces to configure communication drivers:

- **Driver Settings:** Allows the user to adjust the settings that will be used by all communication worksheets configured for the specific communication driver.

Each driver technical reference (help) manual (<DriverName>.pdf) provides a detailed description about the driver settings; however the fields shown in **Table 1** deserve special attention.

Field	Description
<b>Serial Encapsulation</b>	This option is useful to encapsulate <b>serial</b> protocols through TCP/IP, UDP/IP or Modem. Obviously, this option applies only for serial protocols, where the user can use an external converter (e.g.: Ethernet to Serial) to de-encapsulate the protocol on the other end. This option must be set to <b>None</b> for protocols that are natively non-serial, such as Modbus using TCP/IP (MOTCP driver).
<b>Timeout – Start Message</b>	When sending messages to the device (e.g.: PLC), the driver waits for a response from it. If the response does not arrive within the timeout period of time set in this field, the message is discharged indicating time-out error, and the next message is executed. If the <b>Retries</b> field is set with a value greater than zero, the same message is sent again and the timeout time is reset. If the communication fails the number of retries set by the user, the message is discharged indicating error, and the next message is executed.
<b>Simultaneous Requests</b>	This option allows the user to configure the driver to support simultaneous connections. In other words, during the runtime, more than one instance (thread) of the driver is created in order to execute communication messages (commands) in parallel. This option is not applicable for protocols that do not support simultaneous requests in the same physical layer, such as serial protocols. This setting is described in details in the following sections of this document.

**Note:** The Simultaneous Requests dialog is enabled by default only for drivers where this option has been fully tested. Therefore, this option is not enabled by default for some drivers that could support it (in theory). In order to enable this option, you can edit the <DriverName>.INI file for the driver from the \DRV sub-folder of InduSoft Web Studio, including the following settings in the **[CommParam]** section:

- EnableSimultaneousRequests=1
- SimultaneousRequestsMaximumMax=32
- SimultaneousRequestsPerStationMax=16

**Table 1: Fields in the Driver Settings configuration**

- **Main Driver Sheet:** The Main Driver Sheet or “MDS” is a single worksheet available for each communication driver. The main advantage of the MDS is to provide a user-friendly interface to configure all communication addresses of the application. The user simply needs to associate one tag to each station/address and configure one of the following actions for each tag:
  - **Read**
  - **Write**
  - **Read+Write**

When the driver runs, the MDS creates virtual groups for reading commands, the size of which will not exceed the maximum block size supported by the protocol. The READ command is executed using the READ BLOCK Method (reads a block of addresses in each message), triggered whenever the System Tag “**BlinkSlow**” toggles (600ms by default). The WRITE command is executed using the WRITE ITEM Method (writes only one tag value to one address in each message), triggered when the respective tag changes value.

**Note:** Some communication drivers do not support the MDS interface.

- **Standard Driver Sheet:** The user can configure several Standard Driver (work)Sheets or “**SDS**” for each communication driver. The main advantage of the SDS is to provide a flexible solution to divide the communication addresses into different groups manually, and control how these groups will be managed dynamically, during the runtime. The addresses configured in each SDS cannot exceed the maximum block size supported by the protocol.

**Note:** If the maximum block size is exceeded, the message will fail.

The user can control the actions of each SDS using the commands shown in **Table 2**.

Command	Action	Method
<b>Write on Tag Change</b>	When any tag configured on the SDS changes of value, its value is written to the device.	WRITE ITEM
<b>Write Trigger</b>	When the tag configured in this field changes of value (e.g.: toggles), the values of all tags configured in the SDS are written to the device.	WRITE BLOCK
<b>Read Trigger</b>	When the tag configured in this field changes of value (e.g.: toggles), the values of all addresses configured in the SDS are read from the device.	READ BLOCK
<b>Enable Read When Idle</b>	The values of all addresses configured in the SDS are read from the device when the driver does not have messages with higher priority to be executed at the given moment.	READ BLOCK

**Note:** Please consult the driver technical reference (help) manual (<DriverName>.pdf) for a detailed description about all the settings which are available for communication driver that you are using.

**Table 2: Standard Driver Sheet Commands**

## Driver Runtime Internal Structure

### Introduction

The “**Driver Runtime**” is a common task shared by all communication drivers supported by IWS. Each communication driver is a modular component (<DriverName>.dll) which implements a specific communications protocol. As mentioned previously, the InduSoft **MOTCP** communication driver implements the “**Modbus using TCP/IP**” protocol.

The Driver Runtime task is able to accommodate more than one communication driver simultaneously. The maximum number of communication drivers supported simultaneously by the Driver Runtime task is limited by the product license.

The Driver Runtime task can trigger messages (reading or writing commands) at a faster rate than the Communication Driver is able to execute. To accommodate this situation, there is a buffer between the Driver Runtime and the Communication Driver, called the “**Driver Messages Queue**”. The Driver Runtime inserts messages into the queue when reading or writing commands are triggered. The Communication Driver removes the messages from the queue after executing them.

## How the Driver Task Messages are Prioritized

The way that Driver Task Messages are placed into the Driver Messages Queue depends on two factors: Timestamp and Priority.

Messages with higher priority are executed before messages with lower priority, regardless of their Timestamps (time when the messages were triggered). Messages with the same priority are executed according to their timestamp, in a FIFO (First In, First Out) manner.

Because the Driver Runtime inserts the messages into the queue in the order that they must be executed (which is based on the priority and timestamp of each message), the Communication Driver executes the topmost message on the message stack **first**, removing it from the queue after executing it, **regardless if the communication was successful or not**.

**Note:** If the user specified a number of retries in the communication driver **Settings >> Advanced** dialog, the communication driver keeps retrying to execute the message the number of times specified by the user in case the communication fails. When the number of retries expires, the message is removed from the queue, even if the communication was not successful.

**Figure 5** and **Table 3** illustrate how the Driver Runtime task inserts the messages in the queue according to their priority/timestamp and how the Communication Driver executes the message from the top of the message stack and removes it after executing it.

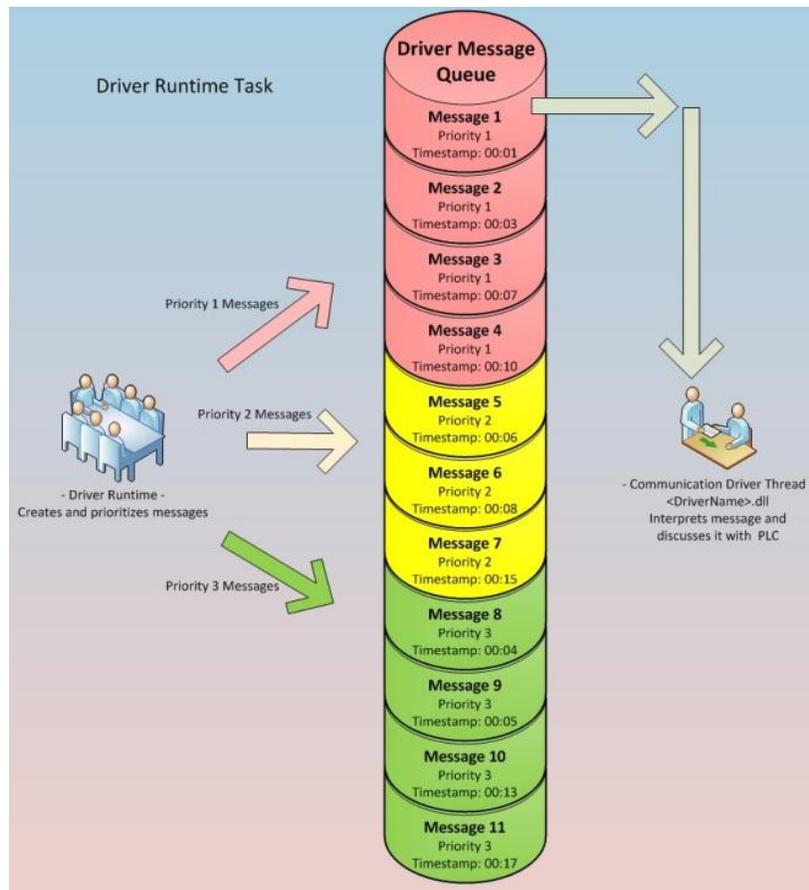


Figure 5: Driver Runtime creating the message queue

Priority	Worksheet Type	Command
Priority 1 (highest priority)	SDS (Standard Driver Sheet)	Any command (Write on Tag Change, Write Trigger, Read Trigger, or Enable Read When Idle), as long as the check-box <b>Increase Priority</b> is checked.
Priority 2	SDS (Standard Driver Sheet)	<b>Write on Tag Change</b>
	MDS (Main Driver Sheet)	<b>Write</b>
Priority 3 (lowest priority)	SDS (Standard Driver Sheet)	<b>Write Trigger, Read Trigger, or Enable Read When Idle</b>
	MDS (Main Driver Sheet)	<b>Read</b>

**Note:** Prior to InduSoft Web Studio v6.1+SP5 (Build 61.5.00.00), messages from the Enable Read When Idle commands had an even lower priority (priority 4). Then, these messages would not even be sent to the queue, unless there was at least one driver instance (thread) available (idle) to execute it. This behavior was not necessary wrong, but it led to confusion, so it was modified for newer versions.

Table 3: Message Type and Priority

### Communication Driver Modes

The simplest scenario for the Communication Driver interface is one instance of a Communication Driver exchanging data with one device (**Figure 6**).

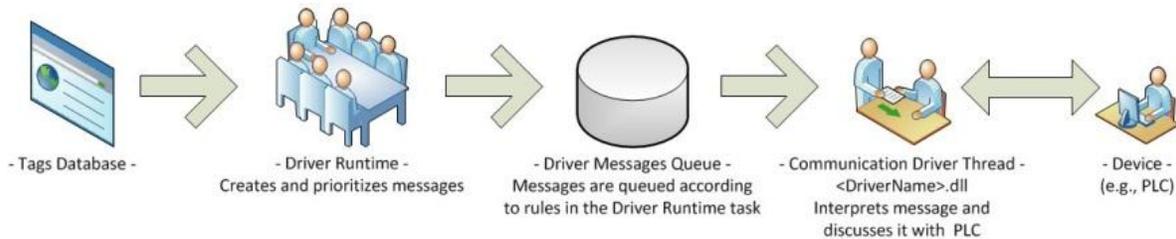
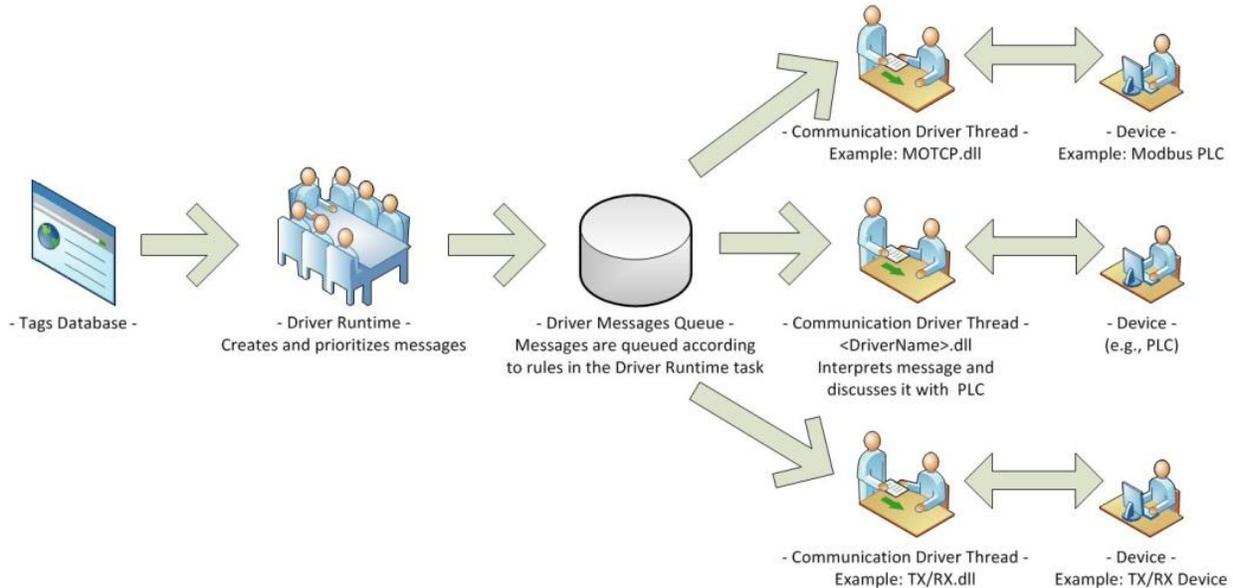


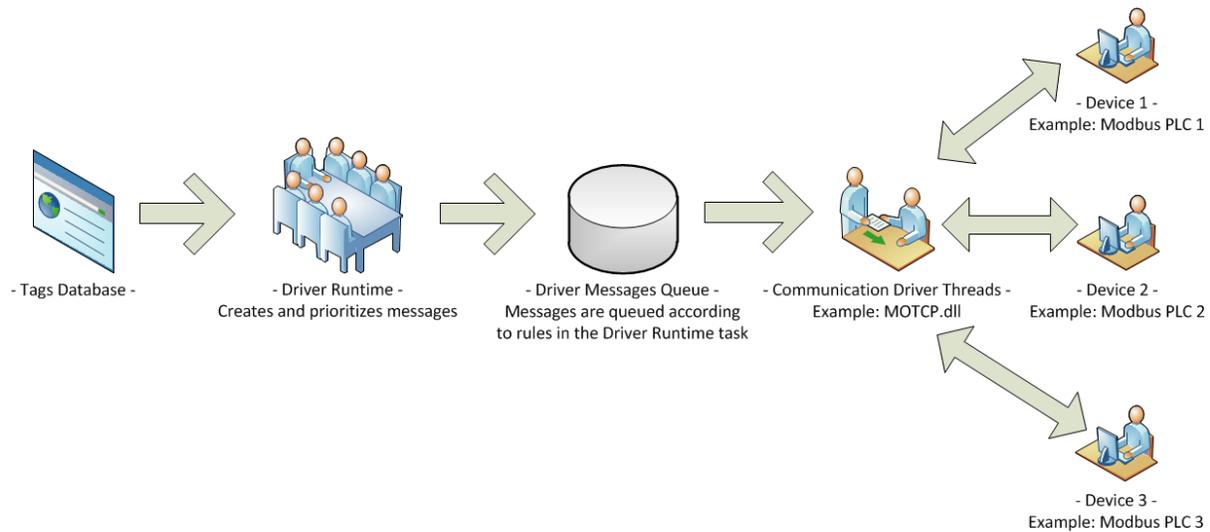
Figure 6: Driver Runtime Task and Driver Communication Thread (<DriverName>.dll)

The Driver Runtime task is able to handle more than one Communication Driver simultaneously. The maximum number of Communication Drivers supported simultaneously by the Driver Runtime task is limited by the product license (**Figure 7**).



**Figure 7: Driver Runtime Task with Several Communication Drivers**

Additionally, the same Communication Driver can exchange data with several devices (**Figure 8**).



**Figure 8: Driver Runtime with a single instance of the driver talking to multiple devices**

Each instance of a Communication Driver executes the messages from the Driver Messages Queue synchronously. This means that the Communication Driver does not execute the next message in the queue until the current message is completely executed (i.e., receives a successful reply from the device, receives an error from the device, or times-out).

This behavior is suitable for many small and medium sized systems. However, in large systems, this scenario might not provide expected performance, because the Communication Driver becomes a bottleneck when communicating with several devices, or even with one single device.

When the Communication Driver is exchanging data with more than one device, and at least one of them is not available (i.e., disconnected from the network), this situation can significantly decrease the performance of the whole communication interface. This is because the Communication Driver does not execute the subsequent messages in the message stack for other device(s) properly connected to the network during the time that it is attempting execution of the message(s) for the disconnected device. Since the “**Time-Out**” time is typically set in order of seconds, it can cause serious delays in the communications to all devices.

InduSoft Web Studio provides a solution for these architecture bottlenecks by using “**Simultaneous Connections**” for the same Communication Driver.

### Simultaneous Connections

The user can configure the number of simultaneous connections for the same Communication Driver in the **Settings** >> **Advanced** dialog interface of the driver (**Figure 9**). During runtime, IWS creates an independent instance (thread) of the Communication Driver for each simultaneous connection configured by the user.

**Note:** The maximum number of simultaneous requests depends on the device and protocol specifications. Please consult the device manufacturer's documentation for more information about the device and protocol that you are using.

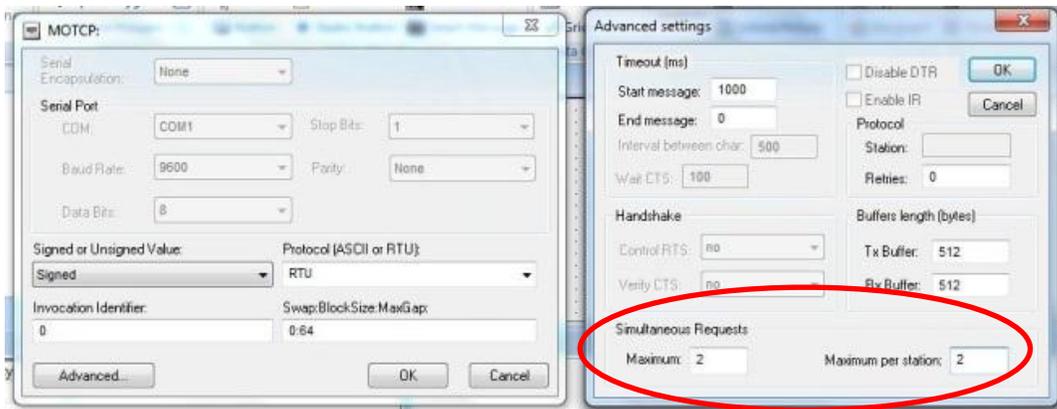


Figure 9: Driver Settings and Advanced Settings Showing Simultaneous Connections

Figure 10 illustrates the internal structure of the communication interface of IWS when the user configures two simultaneous connections for the same device.

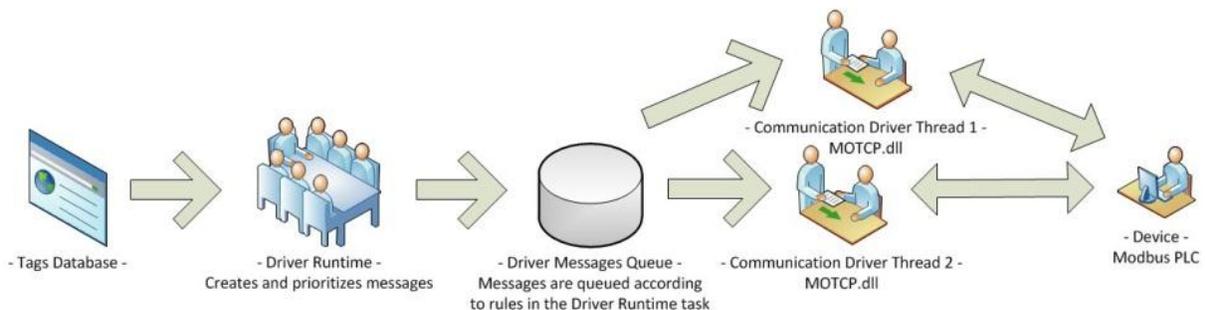


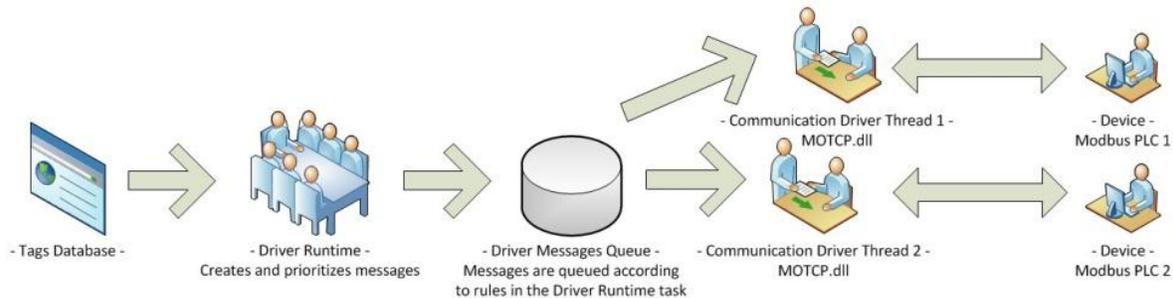
Figure 10: Driver showing two simultaneous connections to the same device

It is important to mention that the instances (threads) of the driver are created in memory, and are transparent to the user.

Each simultaneous connection of the Communication Driver works in parallel with the others. When an instance executes one message, it removes it from the queue and looks for the next suitable message in the queue starting at the top of the message stack.

IWS implements various mechanisms to make sure that simultaneous connections of the driver do not attempt to execute the same message in the Driver Messages Queue. When one instance (thread) of the communication driver looks for a message in the Driver Messages Queue, it might not take the topmost message, if it is addressed for a device which is already in communication with the maximum number of connections (other communication driver instances) configured by the user. In this scenario, the communication driver that is idle looks for the topmost message for a device that is not in communication with the maximum number of connections configured by the user – which may not be necessarily the topmost message in the queue.

**Figure 11** illustrates the internal structure of the communication interface of IWS when the user configures two simultaneous connections, with one per device. In this scenario, there will be one instance (thread) of the communication driver for each device. Should one device not respond to the driver requests, this situation will not decrease the communication performance of the other device.



**Figure 11: Driver showing two simultaneous connections to two different devices**

**Note:** Simultaneous connections cannot be implemented with serial protocols because the physical layer does not support asynchronous requests on the serial line (peer-to-peer). Moreover, drivers that use third-party libraries will support simultaneous connections only if the third-party library supports it too.

## Revision Table

Revision	Author	Date	Comments
A	Fabio Terezhin	January 11, 2008	▪ Initial Draft Revision
B	Fabio Terezhin	March 17, 2009	▪ Modified the priority for the command Enable Read When Idle, based on the changes implemented in the Build 61.5.0.0.
C	Richard Clark	April 30, 2013	▪ Added explanation of OSI Model and Abstraction Layers, updated drawings, rewrote explanations, corrected spelling and syntax errors. Updated IWS graphic to current version. Approved for publication – F Terezhin on May 6, 2013

<sup>i</sup> MORE INFORMATION ABOUT THE OSI MODEL AND THE VARIOUS FUNCTIONALITIES AND PROTOCOLS WITHIN EACH ABSTRACTION LAYER IS AVAILABLE AT:  
[http://en.wikipedia.org/wiki/OSI\\_model](http://en.wikipedia.org/wiki/OSI_model)