Modicon Modbus Master (ASC/RTU/TCP) Driver

Filename	MODBUS.DLL
Manufacturer	Modicon
Devices	Any device compatible with Modbus protocol v1.1b
Protocol	Modbus v1.1b
Version	3.1.30
Last Update	04/01/2016
Platform	Win32 and Windows CE (Pocket PC ARM, HPC2000 ARM, HPC2000 X86, and HPC2000 MIPS)
Dependencies	No dependencies
Superblock Readings	Yes
Level	0

Introduction

This Driver implements the Modbus protocol, which allows an Elipse application to communicate with any slave device that implements this protocol in **ASCII**, **RTU**, or **TCP** modes.

This Driver always works as a master of a Modbus network. If users want to use a Driver to communicate with master devices, then Elipse's **Modbus Slave** Driver must be used, which can be downloaded at *Elipse's website*.

Modbus Driver, starting at version 2.00, was developed using Elipse's **IOKit** library. This library is responsible for implementing the physical layer access (**Serial**, **Ethernet**, **Modem**, or **RAS**). For more information about IOKit configuration, please check **IOKit User's Manual**.

It is recommended to start by reading topic **Quick Configuration Guide** if the device is fully compliant with standard Modbus protocol, defined by the *Modbus Organization (modbus.org)*, and if users only want to read or write bits and registers, without using more advanced Driver features.

For a complete understanding of all Driver functionality, it is recommended to start reading, in this order, chapters **Adding a Driver to an Elipse Application** and **Configuration**.

To create large scale applications, it is also recommended to read topic **Performance Tips**.

If users are not familiar with the protocol, please check the following topics:

- **Modbus Protocol**

- **Recommended Websites**

- **Supported Functions**

- **Special Functions**

Quick Configuration Guide

This topic describes all necessary steps to configure a Modbus Driver for communication with devices compliant with the standard protocol defined by the *Modbus Organization*, considering the most common configuration options.

If a device is fully compliant with the standard protocol, and users only want to read or write registers or bits, the next three topics are probably sufficient to configure this Driver:

- **Inserting a Driver**

- **Configuring a Driver**

- **Configuring I/O Tags**

Inserting a Driver

If using E3 or Elipse Power, please read topic **Adding a Driver to an Elipse Software Application - E3 or Elipse Power**.

If using Elipse SCADA, please read topic **Adding a Driver to an Elipse Software Application - Elipse SCADA**.

Then, read the **next step** on this Manual, which shows how to configure a Driver using its configuration window for the most common cases.

Adding a Driver to an Elipse Software Application

This section describes how to add a Modbus Driver to **E3 or Elipse Power** and **Elipse SCADA** applications.
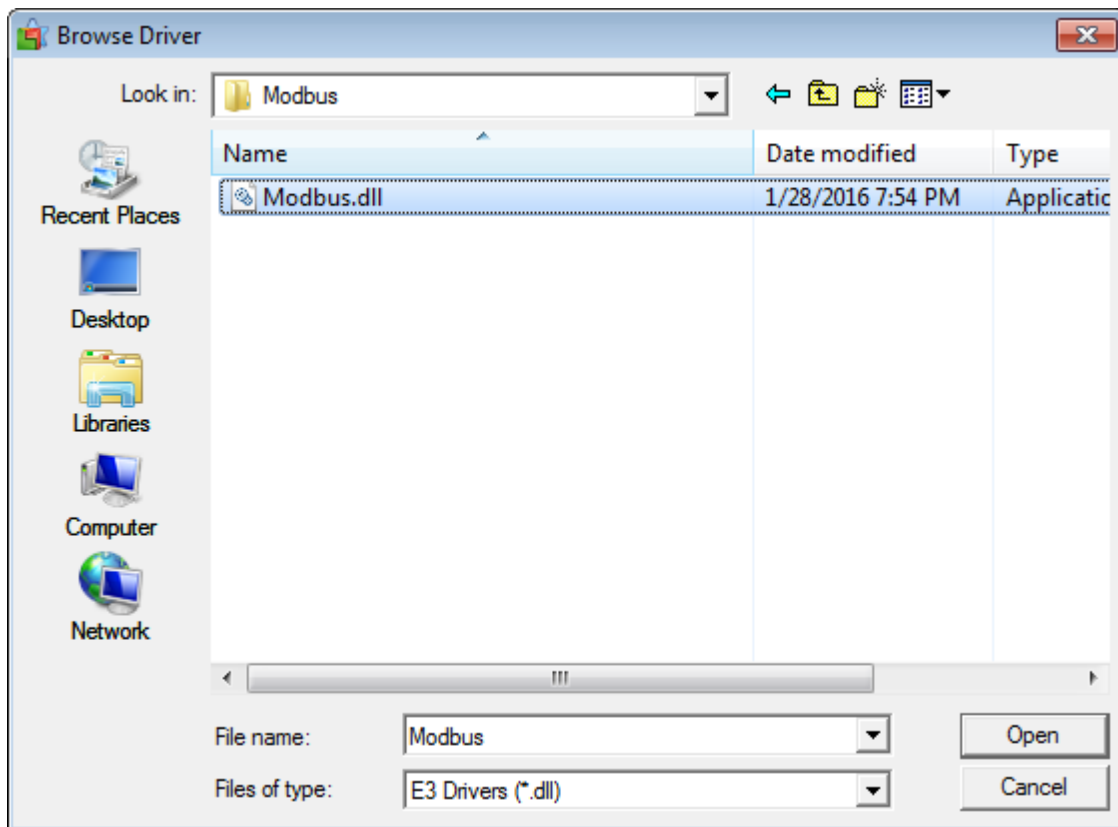
E3 or Elipse Power

On Organizer, right-click the **Server Objects - Drivers and OPC** item, select the **Insert I/O Driver in** option, and then select a project.
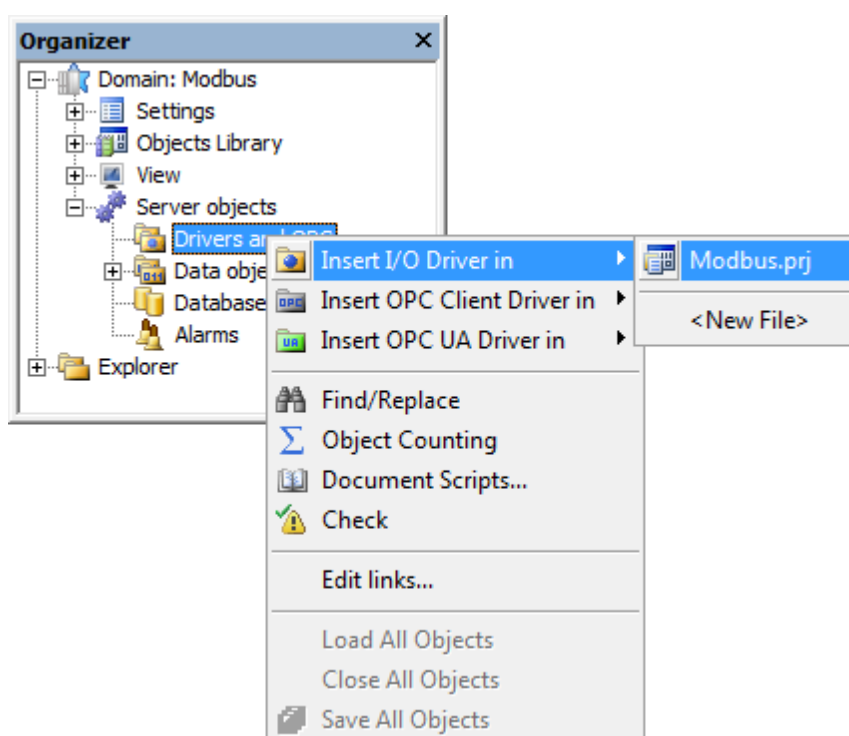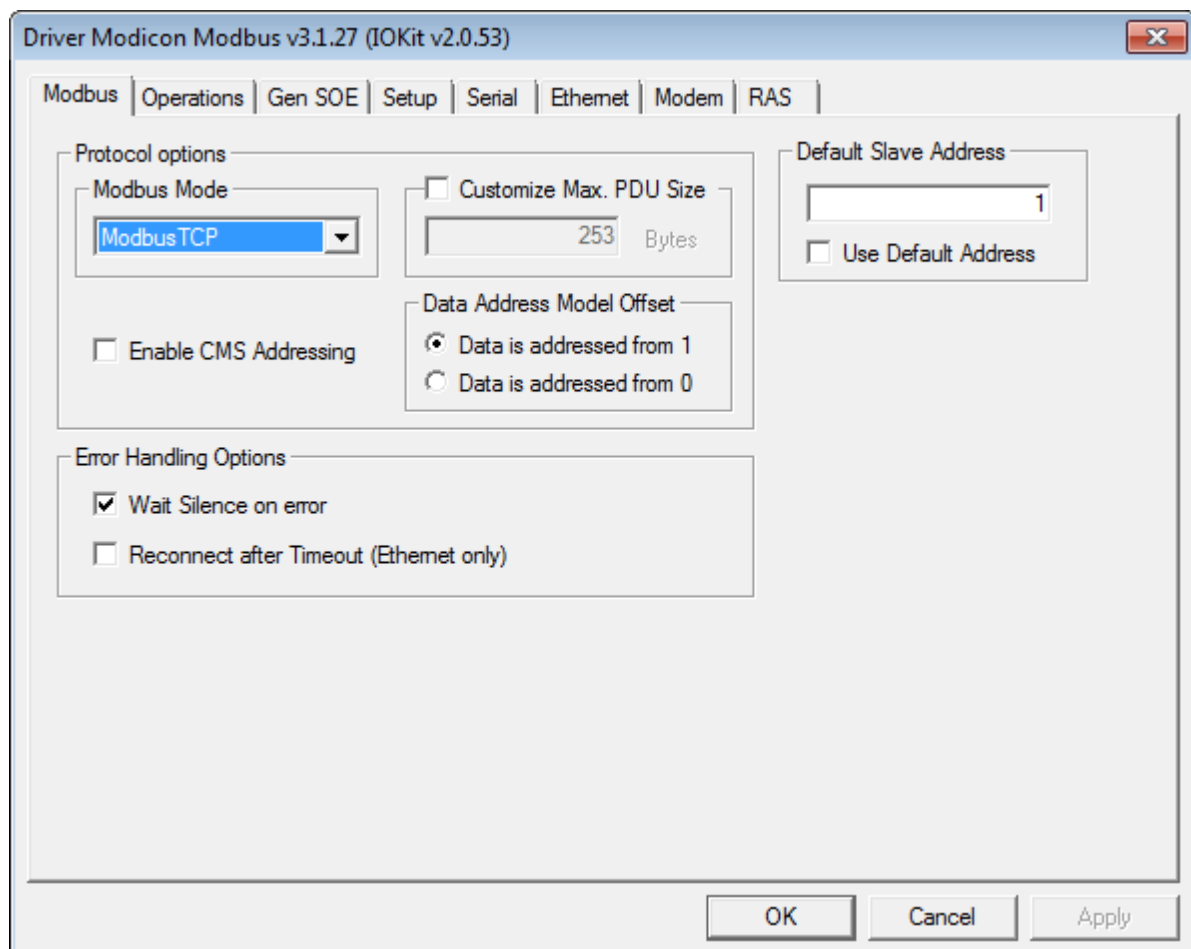

Adding a Driver to an E3 or Elipse Power application

On the window that opens, select a Driver (this file must be extracted to a folder on the computer in use) and click **Open**.
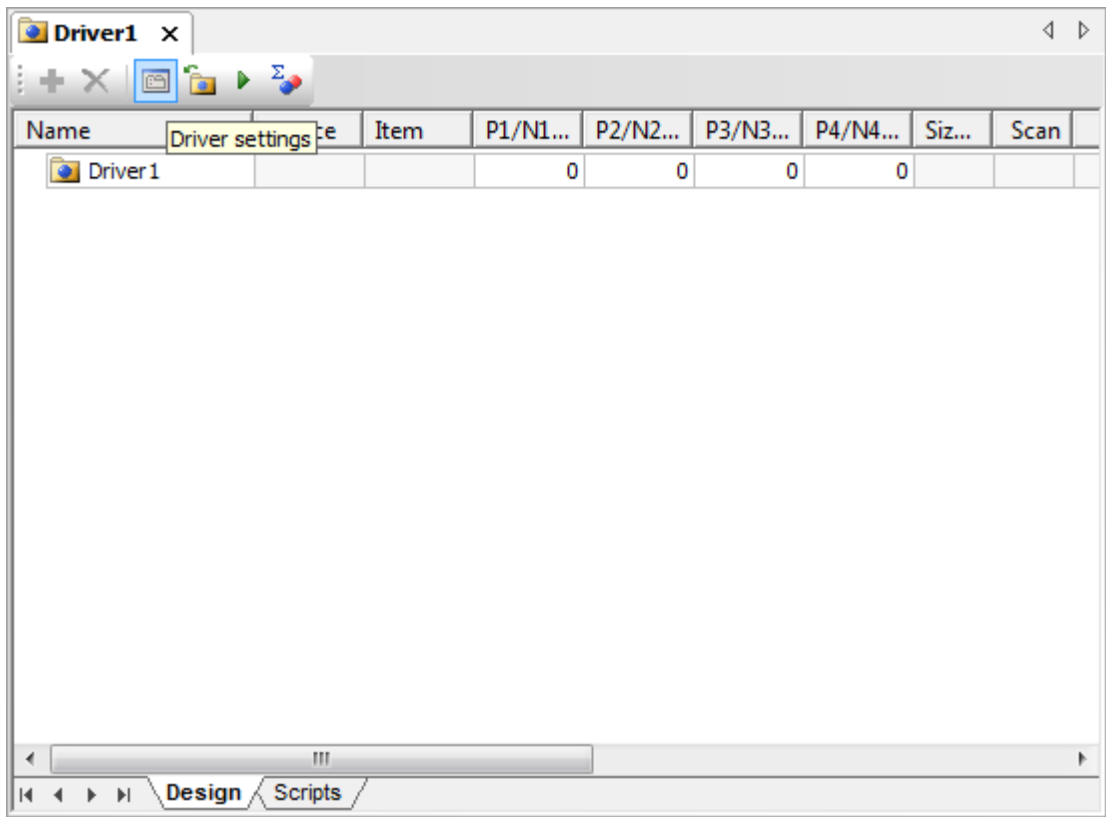

Browse Driver window

E3 or Elipse Power automatically opens Driver's configuration window, shown on the next figure.


Driver's configuration window

On the **second step** of topic **Quick Configuration Guide**, there is a procedure for a basic Driver configuration, for the most common usages. On topic **Properties** this configuration is presented in details.

Driver's configuration window can also be opened, later, by clicking 🖼 **Driver settings**, as shown on the next figure.

Driver settings option

After configuring Driver's properties, click **OK** so that E3 or Elipse Power opens Tag Browser window, allowing to insert pre-defined Tags in the application, based on the most used settings. The next figure shows Tag Browser's window. To add Tags, drag them from the list on the right (**Tags available from driver**) to the list on the left (**Current project tags**).
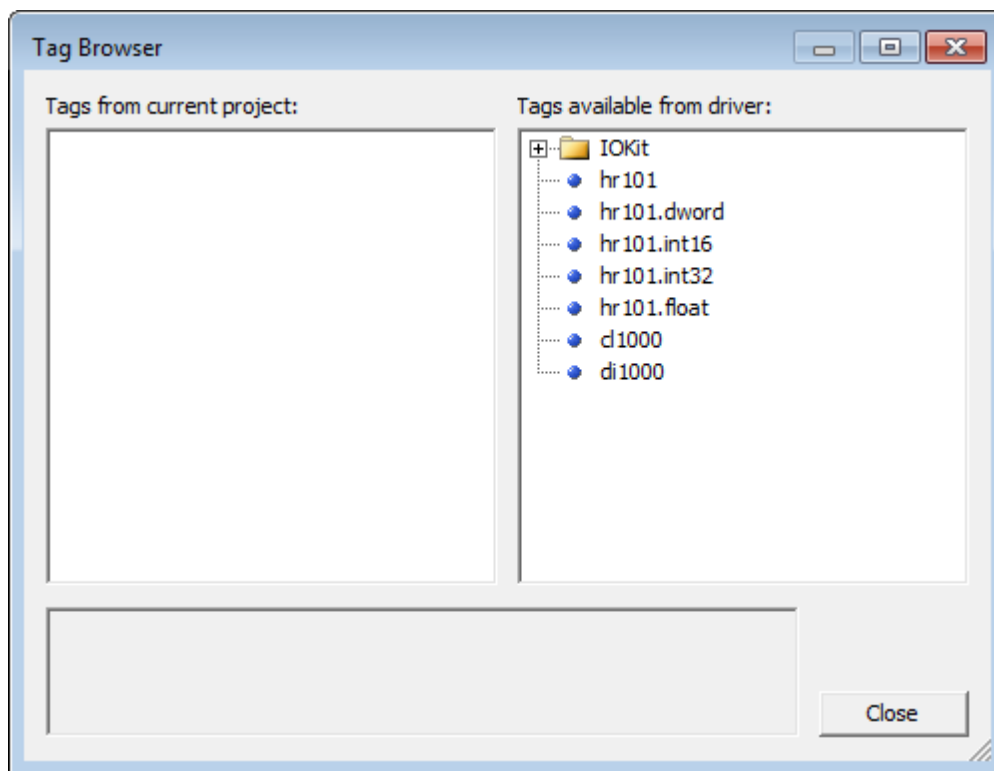
Tag Browser window

Tags available in Tag Browser are Tags **configured using Strings**, a new method that does not use the old concept of operations. Insert the most adequate ones to the application, editing their fields as needed. Tag Browser window can be opened later by clicking **Tag Browser**.

Elipse SCADA

On Organizer, select the **Drivers** item and click **New**.
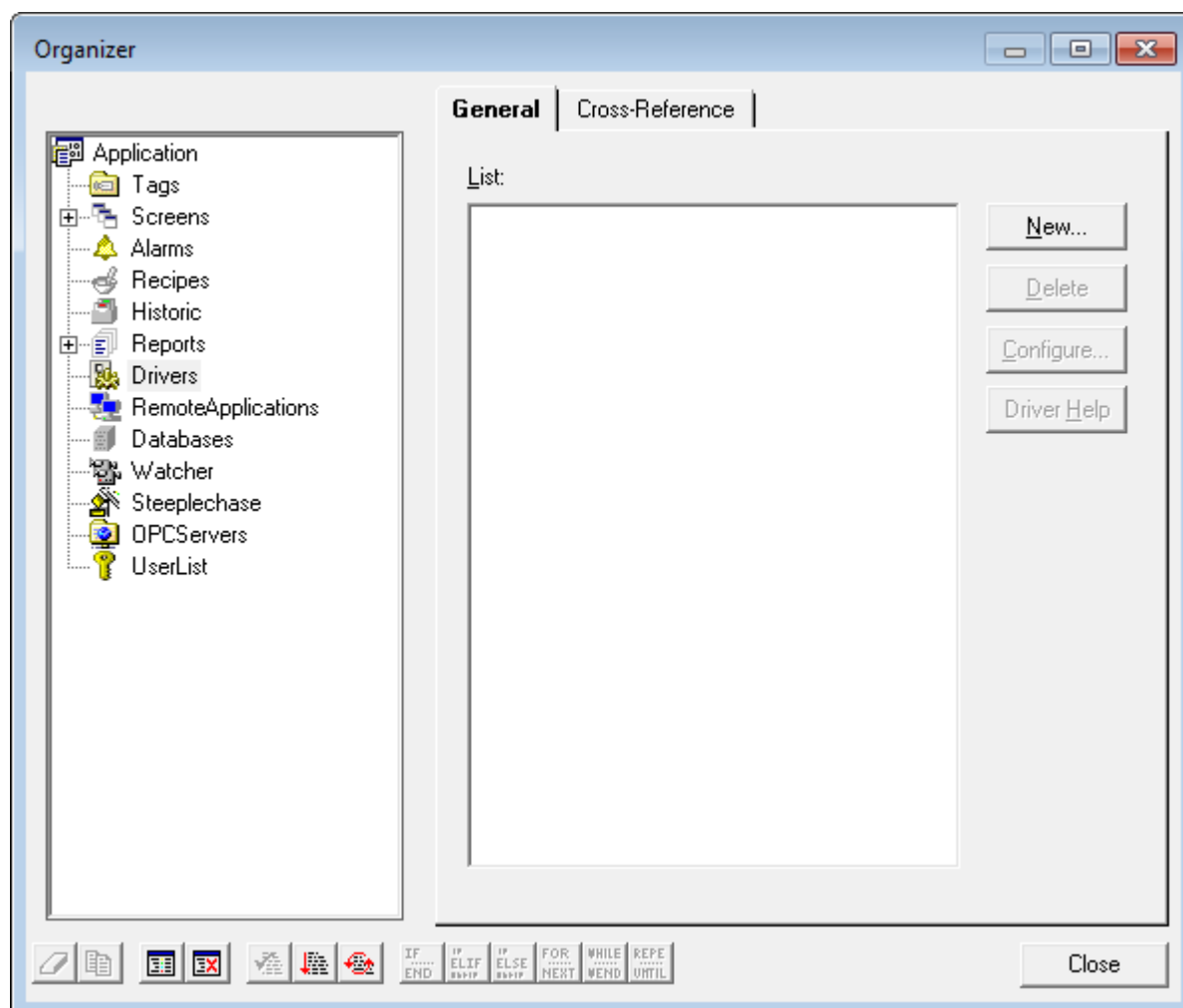
Adding a new Driver to Elipse SCADA

On the window that opens, select a Driver (this file must be extracted to a folder on the computer in use) and click **Open**.
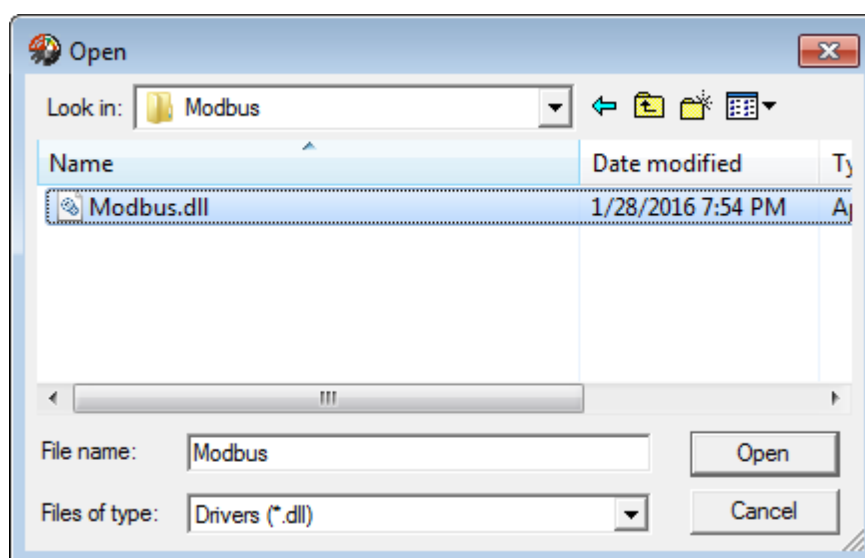
Selecting a Driver

This Driver is then added to the application.
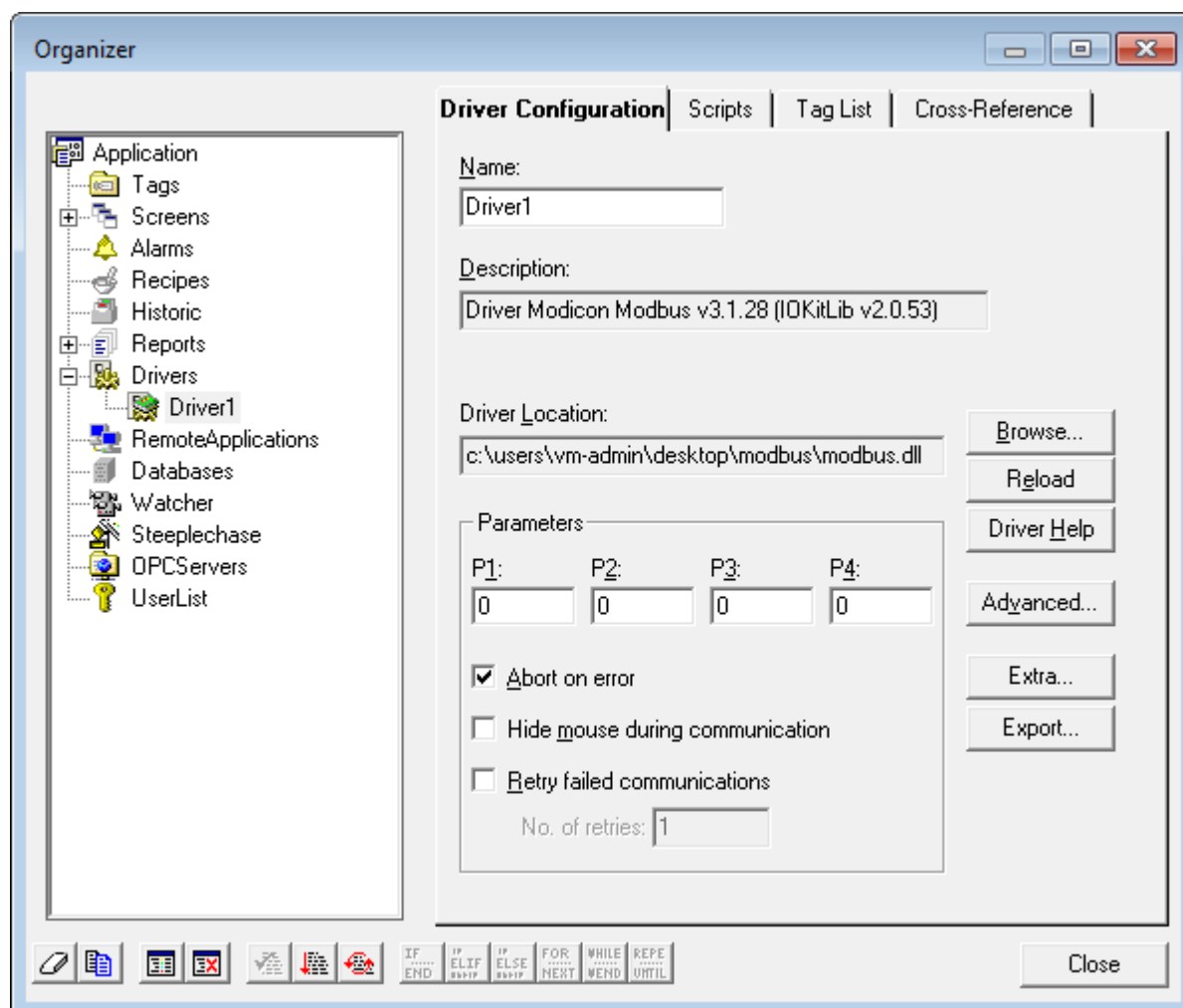
Driver in Organizer

For this Driver to work correctly, users still need to configured it on its configuration window. To open this window, showed on the next figure, click **Extra**.

Modicon Modbus Master (ASC/RTU/TCP) Driver v3.1.30

Driver's configuration window

The **second step** of topic **Quick Configuration Guide** shows how to configure a Driver for the most common usages, for devices complying to default Modbus protocol requirements. On topic **Properties** that configuration is described in details, including advanced configuration resources.

Configuring a Driver

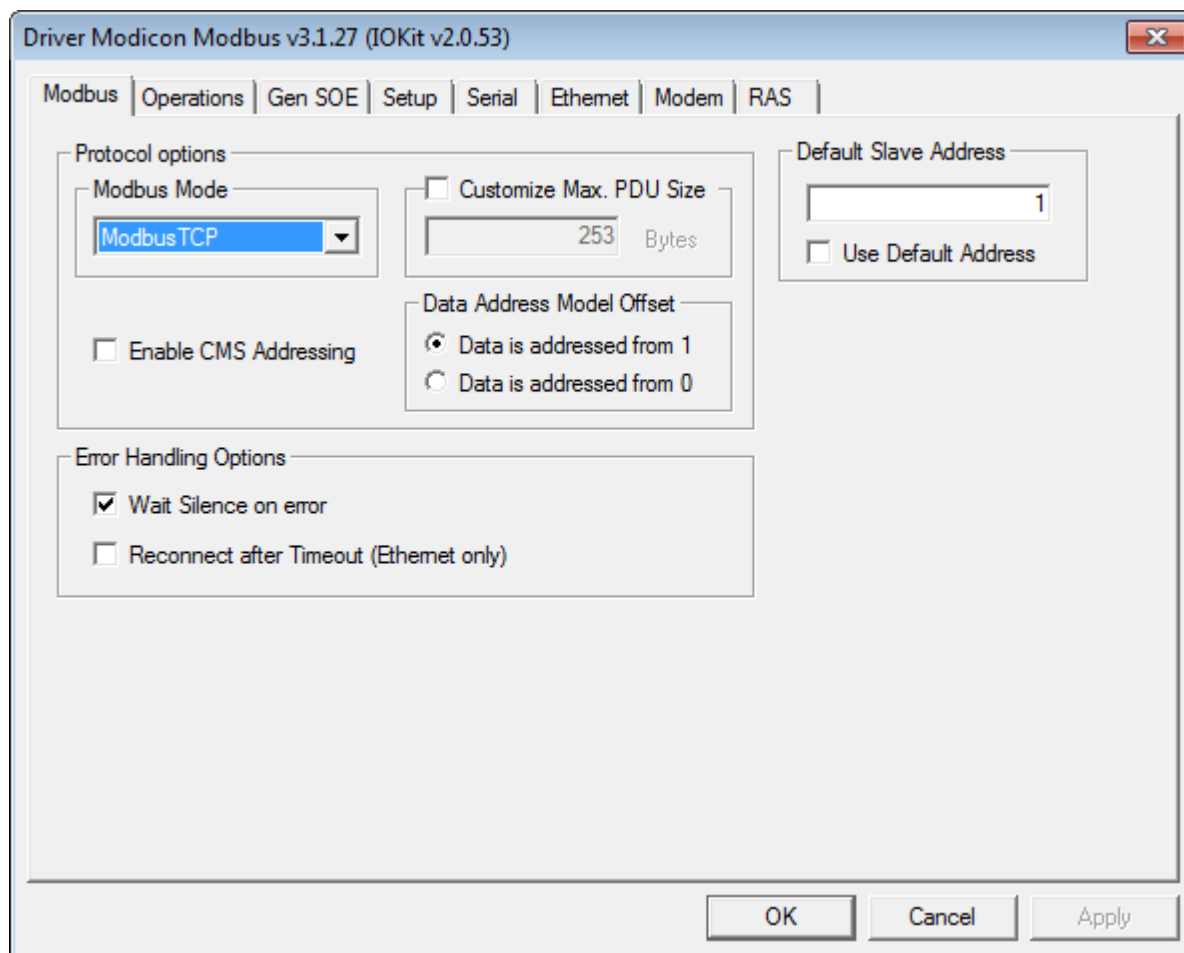After inserting a Driver in an application, users must open Driver's configuration window, as explained on topics **E3 or Elipse Power** or **Elipse SCADA**. With this configuration window open, follow these steps:

1. Configure communication's physical layer:

    a. On **Setup** tab, select a physical layer (**Serial**, **Ethernet**, **Modem**, or **RAS**) to use when connecting with a device.

    b. Configure the selected physical layer on its corresponding tab (**Serial**, **Ethernet**, **Modem**, or **RAS**).

    c. For more information about configuring a physical layer, please check **IOKit User's Manual**.

2. On **Modbus** tab, select the protocol mode (**RTU**, **ASCII**, or **TCP**) used by the device. As a general rule, users must select **RTU** or **ASC** (for most devices it is **RTU**) for **Serial** or **Modem** physical layers, or **TCP** for **Ethernet** or **RAS** physical layers. The other options usually can be kept with their default configurations. For more information about all options on this tab, please check topic **Modbus Tab**.

> **NOTE**: For new applications, it is strongly recommended to avoid using **ModbusRTU** (**RTU** mode) encapsulated in an **Ethernet TCP/IP** layer. However, if by any reason, for legacy applications, users must use **ModbusRTU** encapsulated in **TCP/IP**, please do not forget to enable the **Reconnect after Timeout** option, described on topic **Modbus Tab**.

3. When creating applications for newer Elipse Software products such as E3, Elipse Power, or Elipse OPC Server, users can use Tag configuration by **Strings** (**Device** and **Item** fields). If this is the case, please go to the **next step** of this guide.

4. If users still need to use the old numerical configuration (*N/B* parameters), used by Elipse SCADA, it is important to check the **Operations** tab. There are seven default operations already pre-configured on the Driver. Operations are configurations of functions and data formats that later are referenced by application's Tags. These seven default operations, already available when a Driver is loaded for the first time, are the most common ones. Evaluate all reading and writing functions and the data types used for each operation, and check which ones are needed for the application. If these seven pre-defined operations do not fit the application needs, users must edit them or even create new operations. If this is the case, please read topic **Operations Tab**. The next table lists all seven pre-defined operations.

Pre-defined operations

OPERATION	READING FUNCTION	WRITING FUNCTION	DATA TYPE	PURPOSE
1	3: Read Holding Registers	16: Write Multiple Registers	**Word**	Reading and writing unsigned 16-bit integers
2	3: Read Holding Registers	16: Write Multiple Registers	**DWord**	Reading and writing unsigned 32-bit integers
3	3: Read Holding Registers	16: Write Multiple Registers	**Int16**	Reading and writing signed 16-bit integers
4	3: Read Holding Registers	16: Write Multiple Registers	**Int32**	Reading and writing signed 32-bit integers
5	3: Read Holding Registers	16: Write Multiple Registers	**Float**	Reading 32-bit floating point values
6	3: Read Multiple Coils	15: Write Multiple Coils	**Bit**	Reading and writing bits
7	2: Read Discrete Inputs	None	**Bit**	Reading bits from a data block of Discrete Inputs

NOTE: These seven default operations are configured assuming that a device complies with Modbus' default byte order, *big endian*, in which the most significant bytes come first. If a device does not comply with that standard, please check topic **Operations Tab** for more information about configuring operations for different byte orders.

For more information about configuring this Driver, please read topic **Configuration**.

The **next step** demonstrates how to configure I/O Tags based on pre-defined operations.

Configuring I/O Tags

This section describes how to configure I/O Tags in E3, Elipse Power and in old Elipse SCADA for the most common usages.

Tag Configuration in E3 and in Elipse Power

Configuring I/O Tags in E3 and in Elipse Power can be performed using the new **String configuration** method or using the old **numerical configuration** method, compatible with Elipse SCADA. For new projects, it is recommended to use String configuration, which improves application's legibility and maintenance.

Next there is a description of the recommended procedures for **String** configuration and also for the old numerical configuration, if it is needed for legacy applications.

String Configuration

To add **String**-configured Tags, users have an option to import Tag Browser's pre-defined models, as explained on topic **Adding a Driver to an Elipse Software Application**. To do so, users must keep the **Show Operations in Tag Browser** option disabled on **Operations tab**, and then open Tag Browser by clicking 📷 **Tag Browser**.

To add a new Tag to an application without Tag Browser, follow these steps:

1. On Organizer, double-click the Driver, select the **Design** tab, click ➕ **Add**, and then select the **I/O Tag** item, according to the next figure.
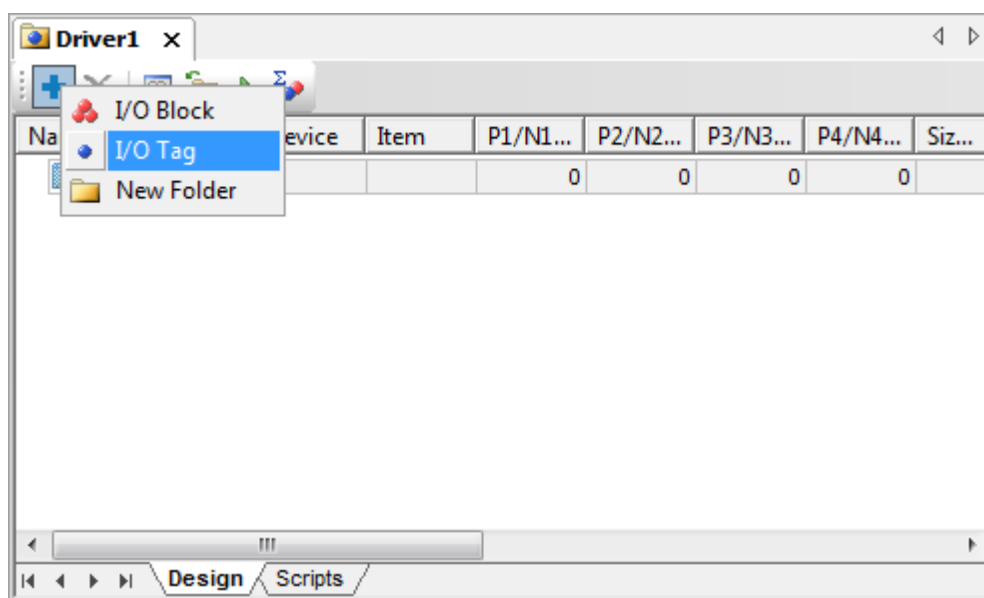
Adding a new I/O Tag

2. On the **Adding IOTag** window, configure the **Quantity** field with value 1 (one) and specify a name for this Tag in the **Name** field. Click **OK** to create a new Tag.

3. On the **Device** column, type the numerical value of device's *Slave Id* to communicate, followed by a colon, such as "1:" for a *Slave Id* equal to 1 (one). Notice that, in an **Ethernet TCP/IP** layer, this value is sometimes ignored, and only the IP address is used and the port configured on **Ethernet** tab, which must be declared on device's documentation.

4. On the **Item** column, specify a mnemonic for the **address space** (a set of reading and writing Modbus functions) followed by the register's or bit's address. For *Holding Registers*, the address space is "hr" or "shr", and this last one does not allow writing in blocks, because it uses the writing function **06** (*Write Single Register*), while the "hr" address space uses the writing function **16** (*Write Multiple Registers*). Both use the reading function **03** (*Read Holding Registers*). For *Coils* use "cl" or "scl". Again, the difference is that the last one, which uses function **05** (*Force Single Coil*), does not write to blocks. Next, there are some examples of configurations for the **Item** column.

a. Reading or writing *Holding Register* 150 using functions **03** and **16** (writing multiple registers): **Item** must be equal to "hr150".

b. Reading or writing *Holding Register* 150 using functions **03** and **06** (*writing simple registers*): **Item** must be equal to "shr150".

c. Reading or writing a *Coil* with address FFF0h (65520) using functions **01** and **15** (writing multiple bits): **Item** must be equal to "cl65520" or "cl&hFFF0" (prefix "&h" can be used to provide addresses in hexadecimal format).

d. Reading or writing a *Coil* with address FFF0h (65520) using functions **01** and **05** (writing simple bits, one by one): **Item** must be equal to "scl65520" or "scl&hFFF0" (prefix "&h" can be used to provide addresses in hexadecimal format).

5. For more information about other features of **String** configuration, such as other Modbus functions, special functions, and different data types, please check topic **String Configuration**.

6. Tag addressing must correspond to the Modbus address map on the device, which must be declared on manufacturer's documentation. When in doubt, please check topic **Addressing Tips**.

Prefer simple Tags (called PLC Tags in Elipse SCADA) rather than Block Tags, keeping the Superblock feature enabled (the **EnableReadGrouping** property set to True), leaving group optimization to the application and to the Driver. For more details, please check topic **Superblock Reading**.

As an example, the next figure shows Tags configured by **Strings**.

Example of Tags configured by Strings

Numerical Configuration

For E3 or Elipse Power, users can use Tag Browser to create Tags with pre-defined operations, **numerically configured**. To do so, select the **Show Operations in Tag Browser** option on **Operations** tab.

Tag Browser's window, shown on the next figure, is opened when clicking **OK** on Driver's configuration window.
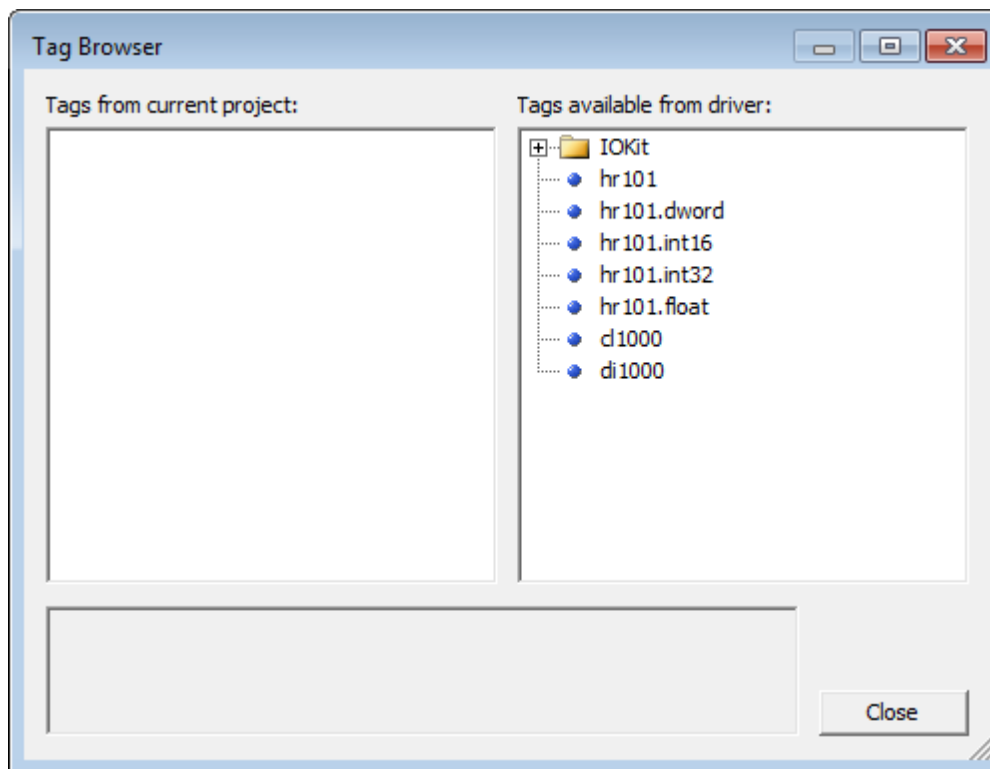
Tag Browser for numerical configuration of Tags

To add a new Tag to an application, follow these steps:

1. Drag Tags from the list **Tags available from driver** to the list **Current project tags**, as described on topic **E3 or Elipse Power**. For many devices, operation **1**, the most common, should be sufficient, drag **Op1<word>** Tag to the list of project Tags. Notice that, assuming that several Tags are needed with the same operation, which is a normal situation, users can drag the same Tag several times (notice that an application adds sequential numbers to the default name). Users can also add a Tag from each operation and then later create other copies in Organizer.

2. Close Tag Browser and configure the *N4/B4* parameter of each Tag with the register or bit address to read or write, according to device's register map. This address map must be described on manufacturer's documentation. When in doubt, please check topic **Addressing Tips**.

3. Also configure the *N1/B1* parameter of each Tag with device's address (*Slave Id*) to access in each case. This parameter is usually configured on the device and, to determine it, please check manufacturer's documentation or technical support when in doubt.

4. Rename all Tags, if needed, with a more significant name for the application.

Configure simple Tags (called PLC Tags in old Elipse SCADA) rather than Block Tags, keeping the Superblock feature enabled (the **EnableReadGrouping** property set to True), leaving group optimization to the application and to the Driver. For more information, please check topic **Superblock Reading**.

Tag Configuration in Elipse SCADA

Elipse SCADA does not support Tag Browser, thus it is necessary to configure I/O Tags manually. Users must create Tags with the following configuration:

- **N1/B1**: Device address (Slave Id)

- **N2/B2**: Operation Code

- **N3/B3**: Not used, leave it 0 (zero)

- **N4/B4**: Address of a Modbus register or bit

Notice that, for this Driver, simple Tag's *N* parameters have the same meaning as Block Tag's *B* parameters, so they are described together.

When in doubt about which value to configure in the *N4/B4* parameter, please check topic **Addressing Tips**.

As Elipse SCADA does not support Superblocks, it is recommended to create Block Tags, grouping adjacent or close registers, to read the maximum number of registers in the smallest number of protocol requests.

Also notice that, once a device supports **default protocol limits for the size of a communication frame**, due to **Automatic Block Partition** feature, there is no need to worry about exceeding the maximum block size supported by this protocol, because this Driver already creates the appropriate subdivisions during communication.

Final Considerations

If users only want to use Driver's default operations, and if a device complies with the standard protocol defined by Modbus Organization, the three steps presented in this Quick Configuration Guide should be sufficient to configure this Driver.

For larger applications, it is recommended to read topic **Optimization Tips**.

More details on I/O Tag configuration are provided on topic **Configuring an I/O Tag**.

The Modbus Protocol

The **Modbus Protocol** was initially developed by Modicon in 1979, and today it is an open standard, maintained by the Modbus Organization (modbus.org), and implemented by hundreds of manufacturers in thousands of devices. Schneider Electric, current controller of Modicon, transferred protocol rights to the Modbus Organization in April 2004, and committed to keep Modbus as an open protocol. Its specification can be downloaded for free at Organization's website (*www.modbus.org*), and protocol's usage is free of licensing fees.

This protocol is based on command and response messages, positioned at layer 7 of the OSI model (application layer), which provides client and server communication among devices connected to different types of networks. It offers services with functions defined by an eight-bit code. There are three categories of function codes:

- **Public function codes**: Protocol's well-defined functions, guaranteed to be unique, validated by the Modbus community, and publicly documented in MB IETF RFC. They can assume values ranging from **1 to 64**, from **73 to 99**, and from **111 to 127**.

- **User-defined function codes**: Non-standard functions, which do not need Modbus.org approval, without any guarantee of being unique, and freely implementable. They can assume values ranging from **65 to 72** and from **100 to 110**.

- **Reserved function codes**: Codes with values inside the range of public functions, currently used by some manufacturers for legacy products, and not publicly available anymore. Examples are **9**, **10**, **13**, **14**, **41**, **42**, **90**, **91**, **125**, **126**, and **127** codes. For more information, please check **Annex A** of protocol's specification (version 1.1b), which is available at *protocol's official website*.

This Driver currently implements 11 of all 19 public functions defined on the current version of protocol's specification (1.1b), as well as some specific manufacturer's functions or related to specific Driver features, known as **Special Functions**. All public functions implemented are described on topic **Supported Functions**. The following protocol's public functions are not yet supported:

- **Function 08**: Diagnostic

- **Function 11**: Get Com event counter

- **Function 12**: Get Com Event Log

- **Function 17**: Report Slave ID

- **Function 22**: Mask Write Register

- **Function 23**: Read/Write Multiple Registers

- **Function 24**: Read FIFO queue

- **Function 43**: Read Device Identification

If users want to implement one of these functions, please contact *Elipse Software's commercial department*.

Recommended Websites

Modicon Modbus Master (ASC/RTU/TCP) Driver is available for download (at no cost) at Elipse Software's *Drivers download* area.

More information about the Modbus protocol can be found at *www.modbus.org*, protocol's official website.

Elipse Modbus Simulator is available for download (at no cost) at Elipse Software's *E3 download* area.

Modsim Modbus Slave Simulator, probably the best known in its category, can be purchased at *www.win-tech.com/html/modsim32.htm*. This software emulates a device, allowing communication with this Driver.

There is also a free alternative called **Free Modbus PLC Simulator**, available for download at *www.plcsimulator.org*.

Other alternatives for simulators and software tools related to this protocol can be found at *protocol's official website*.

Supported Functions

Modbus protocol functions supported by this Driver are described next.

Reading functions

- **01**: **Bit** Reading (*Read Coil Status - 0x*)

- **02**: **Bit** Reading (*Read Input Status - 1x*)

- **03**: **Word** Reading (*Read Holding Registers - 4x*)

- **04**: **Word** Reading (*Read Input Registers - 3x*)

- **07**: Status Reading (*Read Exception Status*)

- **20**: File Record Reading (*Read File Register - 6x*)

Writing functions

- **05**: **Bit** Writing (*Force Single Coil - 0x*)

- **06**: Simple **Word** Writing (*Preset Single Register - 4x*)

- **15**: **Bit** Writing (*Force Multiple Coils - 0x*)

- **16**: **Word** Writing (*Preset Multiple Registers - 4x*)

- **21**: File Record Writing (*Write File Register - 6x*)

Detailed information about each one of these functions can be found on Modbus protocol's specification, available for download at *Modbus Organization's* website.

In addition to protocol's standard functions, as already stated, this Driver also implements special functions, not defined by the protocol, usually related to mass memory readings. A list of all supported special functions can be checked on topic **Special Functions**. A complete Driver configuration is described on topic **Configuration**.

If users want to add support to a new function in this Driver, please contact *Elipse Software's commercial department*.

Special Functions

Special reading and writing functions are Driver functions not defined by the standard Modbus protocol. They were developed to meet specific features of certain devices, or else to provide, in a standardized way by this Driver, features not available in the standard protocol. The Modbus Driver, in its current version, includes the following special functions:

Reading Functions

- **65 03**: Mass Memory Reading (*ABB MGE 144*), described in details on topic **Reading Mass Memory Registers in ABB MGE 144 Meters**

- **GE SOE**: Event Reading (*GE PAC RX7 Systems*), described in details on topic **Reading an Event Buffer in GE PAC RX7 Controllers**

- **SP SOE**: Event Reading (*Schneider Electric SEPAM series Relays*), described in details on topic **Reading Events in Schneider Electric SEPAM Series 20, 40, and 80 Relays**

- **GenSOE**: SOE reading with a generic algorithm, implemented by a resident software in the slave device (PLC), described in details on topic **Elipse Software's Generic SOE Reading Algorithm**

Writing Functions

- **65 01**: Restarts (performs a reset operations) a power meter (*ABB MGE 144*). This command is sent as a Tag's simple writing command (**Write**). Tag's **Value** field is ignored by this Driver and can be left in 0 (zero). For more information, please check device's manual

- **65 02**: Zeroes the maximum and minimum memory (*ABB MGE 144*). This command is sent as a Tag's simple writing command (**Write**). Tag's **Value** field is ignored by this Driver and can be left in 0 (zero). For more information, please check device's manual

Notice that this Driver's special functions, except for the writing function **65 01**, are directly or indirectly related to mass memory reading of registers of their respective devices. For more information, please check topic **Mass Memory Reading**. For a description on how to configure operations and Tags using these functions, please check topic **Configuration**.

Configuration

This section describes how to configure a Modbus Driver. The following topics are discussed:

- **Properties**

- **Configuring Tags**

- **Mass Memory Reading**

Properties

Driver properties can be configured at design time or at run time. A runtime configuration is also called an **Offline Mode Configuration**, and it is described on a specific topic.

At design time, this Driver can be configured using its configuration window, displayed on the next figure.
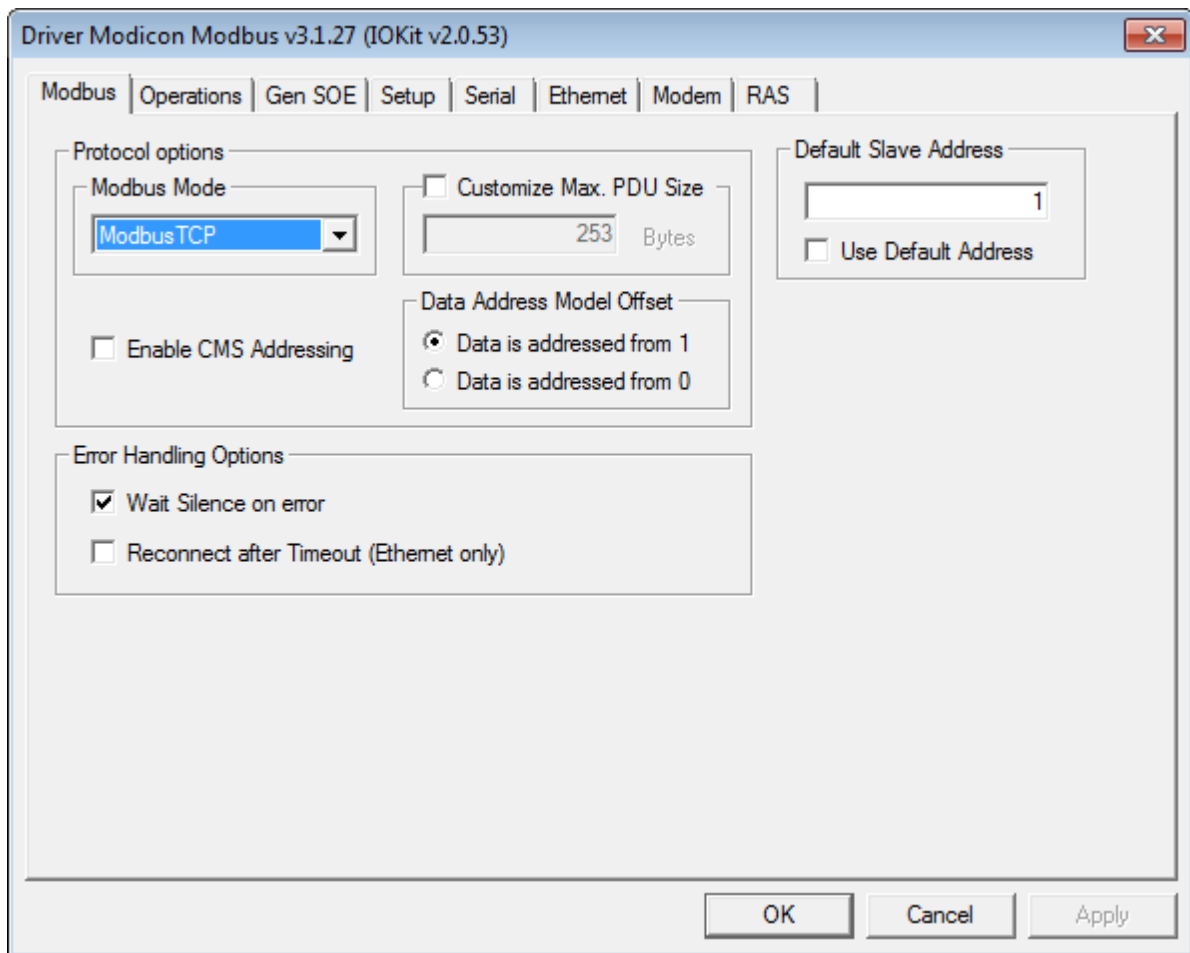
Driver's configuration window

To open Driver's configuration window in E3 or Elipse Power, double-click the Driver object in Organizer and click 🖽 **Driver settings**, as displayed on the next figure.
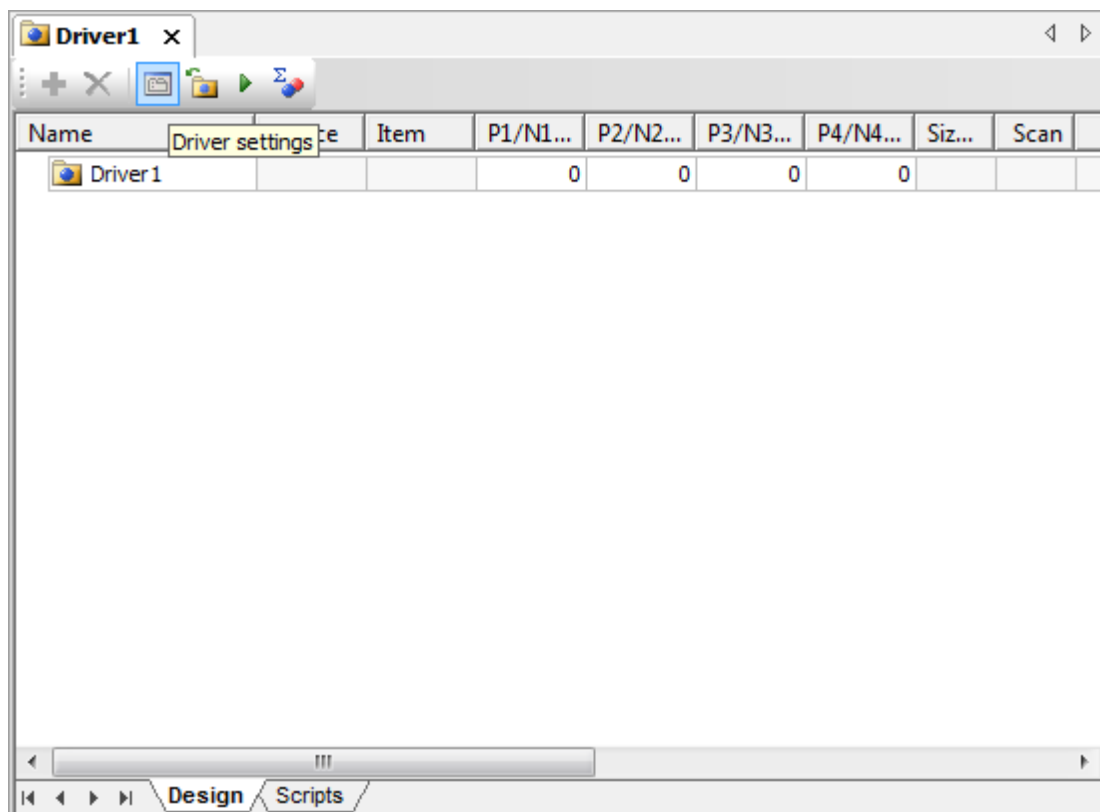
Driver settings option

In Elipse SCADA, on the other hand, Driver's configuration window can be opened by clicking **Extra**, in

application's Organizer.

This configuration window is divided into several tabs, some of them for IOKit configuration, and others are Driver-specific. For Modbus Driver, the **Modbus**, **Operations**, and **Gen SOE** tabs are specific. All other tabs are for IOKit configuration, and they are not described on this Manual. For more information about IOKit configuration, please check **IOKit User's Manual**.

The next topics describe Driver's specific tabs and also the runtime configuration, in **Offline Mode**, using scripts.

- **Modbus Tab**

- **Operations Tab**

- **Gen SOE Tab**

- **Offline Mode Configuration**

Modbus Tab

The **Modbus** tab allows configuring Driver and protocol parameters, as shown on the next figure.
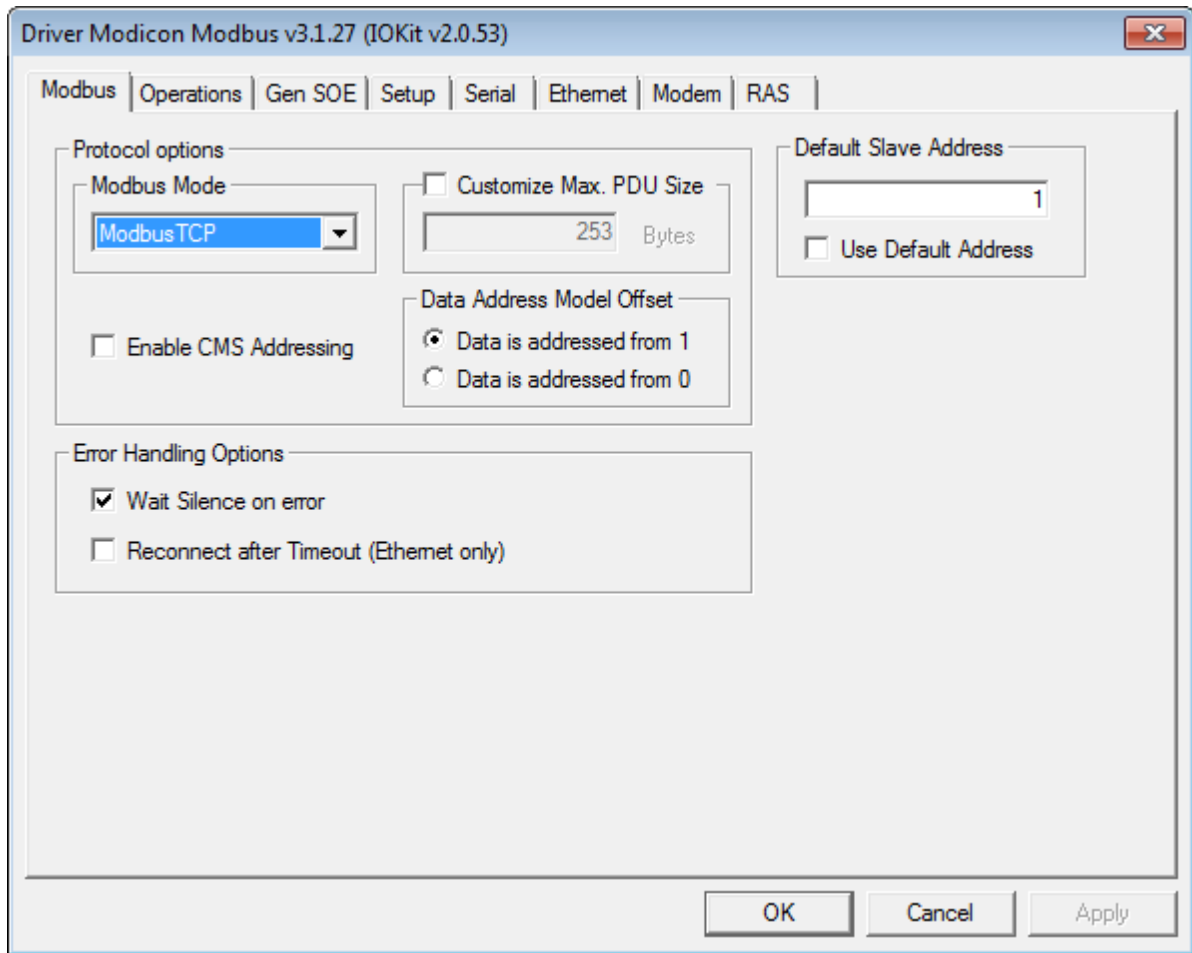
Modbus tab

The next sections describe all configuration options available on this tab.

Protocol Options

This group of options gathers options referring to variations on protocol's standards, according to the next

---

table.

Available protocol options on Modbus tab

OPTION	DESCRIPTION
Modbus Mode	On this combo box users can select a protocol mode to use. Protocol modes are variations defined by the standard for a better adaptation to different physical layers (**Serial**, **Ethernet TCP/IP**, **RAS**, etc.). There are three available options: • **RTU mode**: Default mode for use in serial communications. Includes a 16-bit CRC. • **ASCII mode**: Also used in serial communications, it is used in simpler devices, which do not support **RTU** mode requirements. It uses ASCII characters for transmission, where each byte contains two ASCII characters (one per nibble), thus it is less efficient than **RTU** mode, and rarely found on the market. Uses LRC (*Longitudinal Redundancy Checking*) for error checking. • **ModbusTCP mode**: Used for communication in **TCP/IP** mode. Includes a field for transaction check and does not have an error check system. The transaction field allows discarding delayed responses, thus avoiding that a Driver assumes as a valid response for the current command the response frames from previous commands. This situation may occur if previous modes are encapsulated in TCP/IP.
Customize Max. PDU Size	If enabled, this option allows defining a maximum custom size for PDU (*Protocol Data Unit*). A PDU is a part of the protocol that does not vary between modes (**ModbusTCP**, **ASCII**, and **RTU**) and contains a data area. The number of data bytes supported in each communication is given by this value minus the header bytes, which depend on the Modbus function used. If disabled, the maximum considered size is the **default value defined by the Modbus protocol version 1.1b**, with 253 bytes. This is the recommended option for most devices.
Enable CMS Addressing	This option must be only used in devices that support the **TeleBUS** protocol. If enabled, this Driver accepts a 16-bit **Word** as its slave address, that is, users can define values above 255 and below 65535 as a slave address. In this case, a slave address is then defined in the protocol by three bytes. In addition, the **Default Slave Address** option stops working.

OPTION	DESCRIPTION
Data Address Model Offset	This option enables or disables the default protocol's data offset, by one unit. Available options are: • **Data is addressed from 1 (default)**: The address provided (address of the **Item** field in **String** configuration or the *N4/B4* parameter in numerical configuration) is decremented by 1 (one) before sending it to a device. This offset is part of protocol's specification, therefore this is the default option. • **Data is addressed from 0**: The user-provided address is used in protocol requests, without changes. As a general rule, select the first option if device's register mapping starts at 1 (one) and the second option if it starts at 0 (zero). Also check if the manufacturer uses additional offsets from the old **Modbus Convention**. For more information, please check the **next section**.

TIP: Avoid using protocol's **RTU** mode encapsulated in **Ethernet TCP/IP** layer. If there is a need to encapsulate serial communication for devices using **Modbus RTU** in **TCP/IP**, there are gateways available on the market that not only encapsulate serial communication in Ethernet TCP/IP, but also convert **Modbus RTU** to **Modbus TCP**. As a last option, if using **Modbus RTU** in an **Ethernet TCP/IP** layer is inevitable, remember to enable the **Reconnect after Timeout** option, described on the next table.

Data Address Model Offset

This configuration option, described on the **previous table**, is a source of frequent doubts when addressing I/O Tags, because there are many variations in how it is implemented by manufacturers. Next there is more information about this addressing.

In protocol's standard data model, four data blocks are defined (or address spaces): *Discrete Inputs*, *Coils*, *Input Registers*, and *Holding Registers*. In each one of these blocks, data elements are addressed starting at 1 (one). On the other hand, the communication frame's specification defines a PDU with addresses that range from 0 (zero) to 65535. The relation between the address provided by the PDU and the address of data elements, therefore, has an offset of 1 (one), that is, if in a request's PDU there is an address 0 (zero), the data element to access is the address 1 (one).

With this option on **Modbus** tab, users can select whether this Driver sets that value automatically, thus allowing the use of data element's address on Tags (default option) or the value sent in the PDU is the same provided on Tag configuration (the N4/B4 parameter in **numerical configuration**). There are devices that comply with Modbus standard in their address maps (starting at one) and other devices that map their values without a default offset, directly using the value of the address on communication's frame (starting at zero).

In addition to this single offset, there are still devices that use the old offset standard used by Modicon, the company that created the protocol, which is known as **Modbus Convention**, detailed on topic **Addressing Tips**. Please check device's manual for information on the register map used. When in doubt, please check manufacturer's technical support.

NOTE:The **Data Address Model Offset** option used to be named **Use Older Address** on versions earlier than version 2.03, where the **Data is addressed from 1** option is equivalent to the old **Use Older Address** option enabled, and the **Data is addressed from 0** option is equivalent to the **Use Older Address** option disabled.

Other Options

The next table describes all other options on this tab, referring to Driver's behavior.

**Other available options on Modbus tab**

OPTION	DESCRIPTION
Default Slave Address	This feature allows configuring a default address for slaves, so that it is not necessary to configure them in each Tag. To use this feature, configure the *Slave Id* (the *N1/B1* parameter on **numerical configuration** or the **Device** field on **String configuration**) as 1000, that is, all Tags with their *Slave Id* equal to 1000 have this value replaced by the value configured in the **Default Slave Address** option. Users can also force using a default address in all Tags, regardless of the value configured in *Slave Id*, by selecting the **Use Default Address** option.
Wait Silence on error	If this option is enabled, after every communication error a Driver remains in loop, receiving data until a time-out occurs. This clears the reception channel and prevents problems in future communications due to the reception of delayed bytes still in transit at the time of the error, and which may be confused with a response to a new command.
Reconnect after Timeout (Ethernet only)	With this option enabled, after any time-out error in device's frame reception, this Driver performs a disconnection and a reconnection to the physical layer, clearing the connection from possible delayed frames still in transit, which may affect future requests. This option must be always enabled if using **Modbus RTU** in an **Ethernet TCP/IP** layer is inevitable in legacy systems, as the **RTU** mode does not have a transaction control, therefore it is not always possible to distinguish a correct response frame from a delayed one resulting from a previous reading, possibly from another address, which failed by a time-out. For new projects, it is strongly recommended **NOT** using **Modbus RTU** or **Modbus ASC** modes in an **Ethernet TCP/IP** layer. Notice that users must keep the **Retry failed connection every** option enabled on IOKit's **Setup** tab, so that this Driver reconnects after a time-out. Otherwise, this time-out only generates a disconnection and the application is responsible for managing this new connection.

NOTE: The old **Swap Address Delay** option was removed from the configuration window in version 2.08. This Driver still supports it in pre-existing applications and allows enabling it by script (please check topic **Offline Mode Configuration**). For new applications, it is recommended to use the **Inter-frame Delay** option on IOKit's **Serial** tab, which replaces this old option with benefits.

Operations Tab

This topic describes how to configure the **Operations** tab on Driver's configuration window, where all operations used on I/O Tags are defined, as shown on the next figure.

Configuring operations is no longer used when **configuring Tags by Strings**, it is only used in the old **numerical configuration** (*N/B* parameters) on Elipse SCADA.
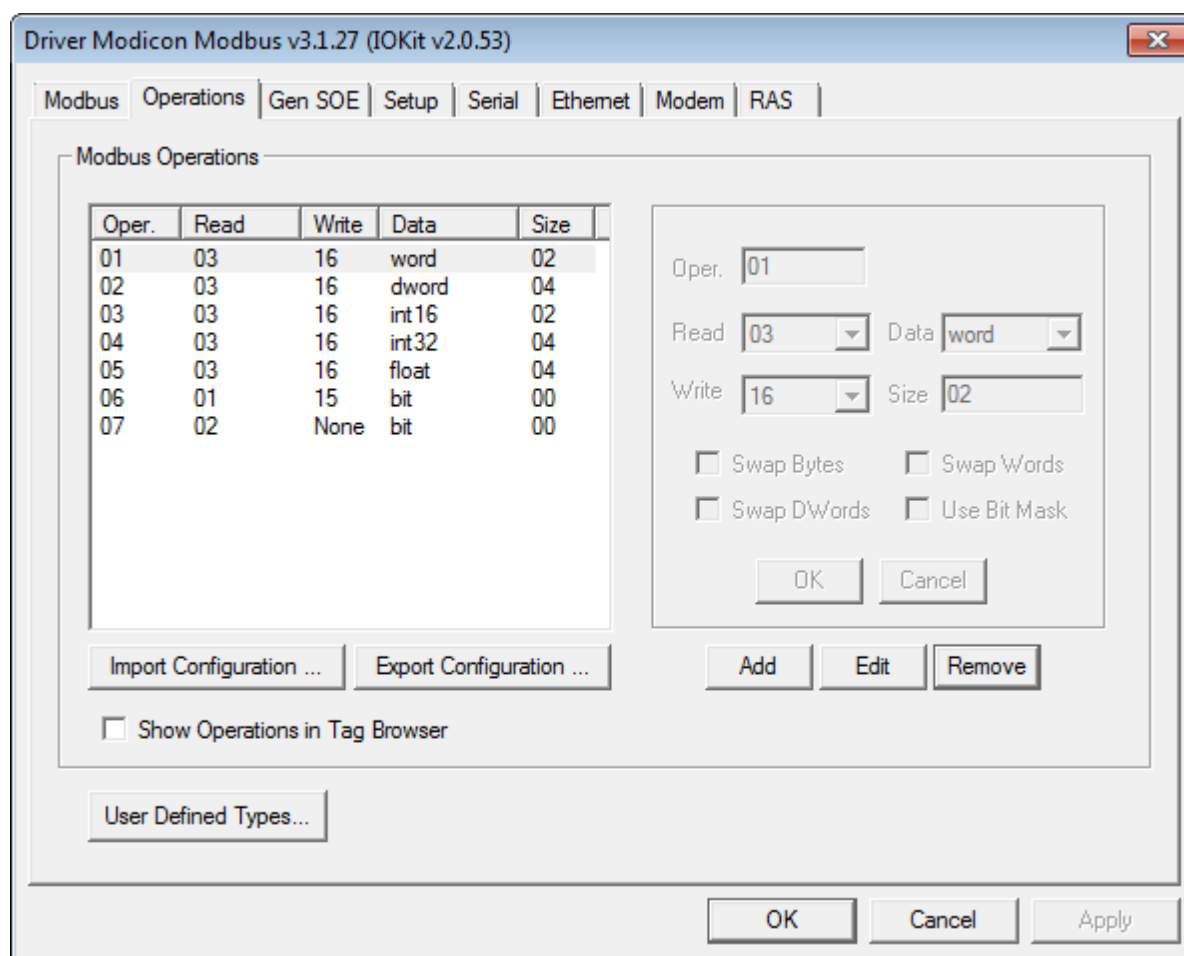
Operations tab on Driver's configuration window

Operations

For a proper usage of this Driver, users must define which reading and writing Modbus functions are used for each I/O Tag. To do so, if Tag configuration is performed using the old **N/B numerical parameters** in Elipse SCADA, select the **Operations** tab on configuration window.

For this Driver, **Operations** are configurations defining how each I/O Tag performs data writings and readings to and from a device.

An operation is nothing more than a definition of a pair of protocol functions, one for writing and another one for reading, and a specification for additional conversions on the format of data that can be linked to application Tags. In other words, in Modbus Driver the *N* or *B* numerical parameters of I/O Tags do not directly reference protocol functions, but rather pre-configured operations, which by their turn not only inform functions (protocol's **native** or even **special ones**) to use when communicating, as well as the way native protocol data must be interpreted.

Configuration of I/O Tag parameters is described later on topic **Configuring an I/O Tag**. Next, there is a description of the configuration of operations, which must be later linked to each I/O Tag.

Functions

The Modbus protocol defines reading and writing functions, which can access distinct address spaces on a device, and with specific data types. Functions **03** and **16**, for example, protocol's most used ones, are responsible for reading and writing *Holding Registers*, which are simply 16-bit unsigned integer values (**Words**).

Modbus protocol's default functions provide data only in basic 16-bit **Bit** and **Word** formats. There are no additional data formats in protocol's specification.

A list of all Modbus functions supported by this Driver, which can be set to the configured operations, can be checked on topic **Supported Functions**.

In addition to protocol functions, this Driver also contains some **Special Functions** that are not part of protocol's standard, with a proprietary format and commonly used for reading events (SOE).

Data Formatting

In addition to allow linking functions (from the protocol or special ones) to specific Tags, operations also allow defining an additional formatting to apply to data, providing support to additional data types, not specified by the protocol, such as 32-bit (**Float**) and 64-bit (**Double**) floating point values. Supported data types are described on topic **Supported Data Types**.

It is important to notice that, when 32- and 64-bit data types are defined in operations, users must define protocol functions that work with 16-bit registers. This way, reading data with more than 16 bits results in reading several 16-bit Modbus registers from a device, that is, to read a Tag linked to an operation defining a 32-bit **Float** data type, this Driver must read two consecutive 16-bit registers from a device, concatenate them, and then perform a conversion to a **Float** format.

Users can also define eight-bit data types (**Byte**, **Int8**, or **Char**) in operations. Notice that, as protocol functions do not allow reading and writing isolated bytes, for each two Block Elements of eight-bit types, this Driver is forced to access a distinct 16-bit register on a device. For that reason, this Driver does not allow writing eight-bit data types to Tags, to isolated Block Elements, or to Blocks with odd or unitary sizes. Writing eight-bit data types must be always performed even-size Blocks.

User-Defined Data Types

In addition to pre-defined data types (native or built-in data types) described on topic **Supported Data Types**, this Driver also allows user-defined data types. These data types must be declared on a specific window, by clicking **User Defined Types**, at the bottom of **Operations** tab. Such data types are structures created from pre-defined data types. For more information about user-defined data types, please check topic **User-Defined Data Types**.

Byte Order

In addition to protocol's reading and writing functions and the data type used, each operation also allows setting additional manipulations to bytes, related to a byte order, that is, the order of bytes inside every value. These are called swap options (*Swap Bytes*, *Swap Words*, and *Swap DWords*). Such options only need to

---

be enabled for devices that do not respect protocol's default byte order.

The Modbus protocol defines that its 16-bit values always use a byte order known as *big endian*, also known as *Motorola*, because it is used by this manufacturer. The *big endian* standard always defines a byte order so that the most significant byte of each value comes first. Thus, as an example, when reading the hexadecimal value 1234h, the device first sends the most significant byte 12h and then the least significant one, 34h.

For devices that do not implement protocol's default byte order, and use another one known as *little endian* or *Intel*, data is sent with the least significant bytes first. Users must then enable those swap options to reverse that byte order.

There are also devices that use different byte orders for 32- and 16-bit types. For devices that, for example, use Modbus' default byte order (*big endian*) for 16-bit types, but provide 32-bit data with the least significant **Word** first (*little endian*), users must only enable the **Swap Words** option, leaving the **Swap Bytes** option deselected. There are basically three possible situations:

- For devices that provide data using Modbus' default byte order (*Motorola* or *big endian*), with the most significant bytes first, users must left all swap options disabled. This is the most common situation.

- For devices using another byte order standard, with the least significant bytes first (*little endian*), users must enable all swap options referring to the data type used, that is, for 16-bit types, enable the **Swap Bytes** option. For 32-bit data types, enable the **Swap Bytes** and **Swap Words** options. For 64-bit types, all three swap options must be enabled.

- In the least common case, devices that use different byte orders for different data sizes, providing, for example, the most significant byte of each **Word** first, but the least significant **Word** of each **DWord** first, then users must evaluate in which case each swap option must be enabled, so that it converts a value returned by a device to protocol's default *big endian* format.

> **NOTE**: All mentioned swap options have no effect for **Bit** data types or for data types with an eight-bit size (**Byte**, **Char**, and **Int8**). Swapping occurs inside every data type, that is, the **Swap Words** option has no effect for 16-bit data types, as well as the **Swap DWords** option has no effect for 32-bit data types. **BCD** data types also do not allow swapping.

To check if a device uses some unusual byte order format, check its manufacturer's documentation. In case this information is not found on that documentation, please contact manufacturer's technical support.

The topic **Frequently Asked Questions** contains tips on byte order configurations for some devices that are known to use swap options.

Bit Mask

The **Use Bit Mask** option is an advanced feature, used in specific and unusual cases where users want to read only a bit from the value returned by a device, but they cannot use application's bit mapping.

For most users, application's bit mapping fields are the best alternative to access bit masks, and there is no need to use this Driver's feature.

This feature was initially created to allow reading bits from *Holding Registers* by specialized E3 libraries, in situations that prevented the usage of application's bit mapping.

In this case, this Driver reads a value from a device normally and then masks it, to return to Tag's **Value** field only the specified bit (**0** or **1**). The definition of a bit number to return is performed on I/O Tag's *N3/B3*

parameter.

The **Use Bit Mask** option can only be used with integer data types with 16 bits or more (**Int16**, **Int32**, **Word**, or **DWord**). In addition, operations that enable this option can only be used for reading. The Modbus function for writing operations (**Write**) that use this mask option can be defined as **None**.

Driver's Default Operations

By default, when a new Driver is added to an application, this Driver is already created with seven default operations, described on the next table.

Default operations

OPERATION	READING FUNCTION	WRITING FUNCTION	DATA TYPE	PURPOSE
1	3 - Read Holding Registers	16 - Write Multiple Registers	**Word**	Reading and writing unsigned 16-bit integers
2	3 - Read Holding Registers	16 - Write Multiple Registers	**DWord**	Reading and writing unsigned 32-bit integers
3	3 - Read Holding Registers	16 - Write Multiple Registers	**Int16**	Reading and writing signed 16-bit integers
4	3 - Read Holding Registers	16 - Write Multiple Registers	**Int32**	Reading and writing signed 32-bit integers
5	3 - Read Holding Registers	16 - Write Multiple Registers	**Float**	Reading an writing 32-bit floating point values
6	3 - Read Holding Registers	15 - Write Multiple Coils	**Bit**	Reading and writing bits
7	2 - Read Discrete Inputs	None	**Bit**	Reading bits from a Discrete Input data block.

These operations are the most commonly used, and operation 1 is the most common one. For most devices, select all operations needed among the ones provided by default, there is no need to create new operations or change the configuration of these default operations.

Defining New Operations

To add a new operation to a Driver, click **Add**.
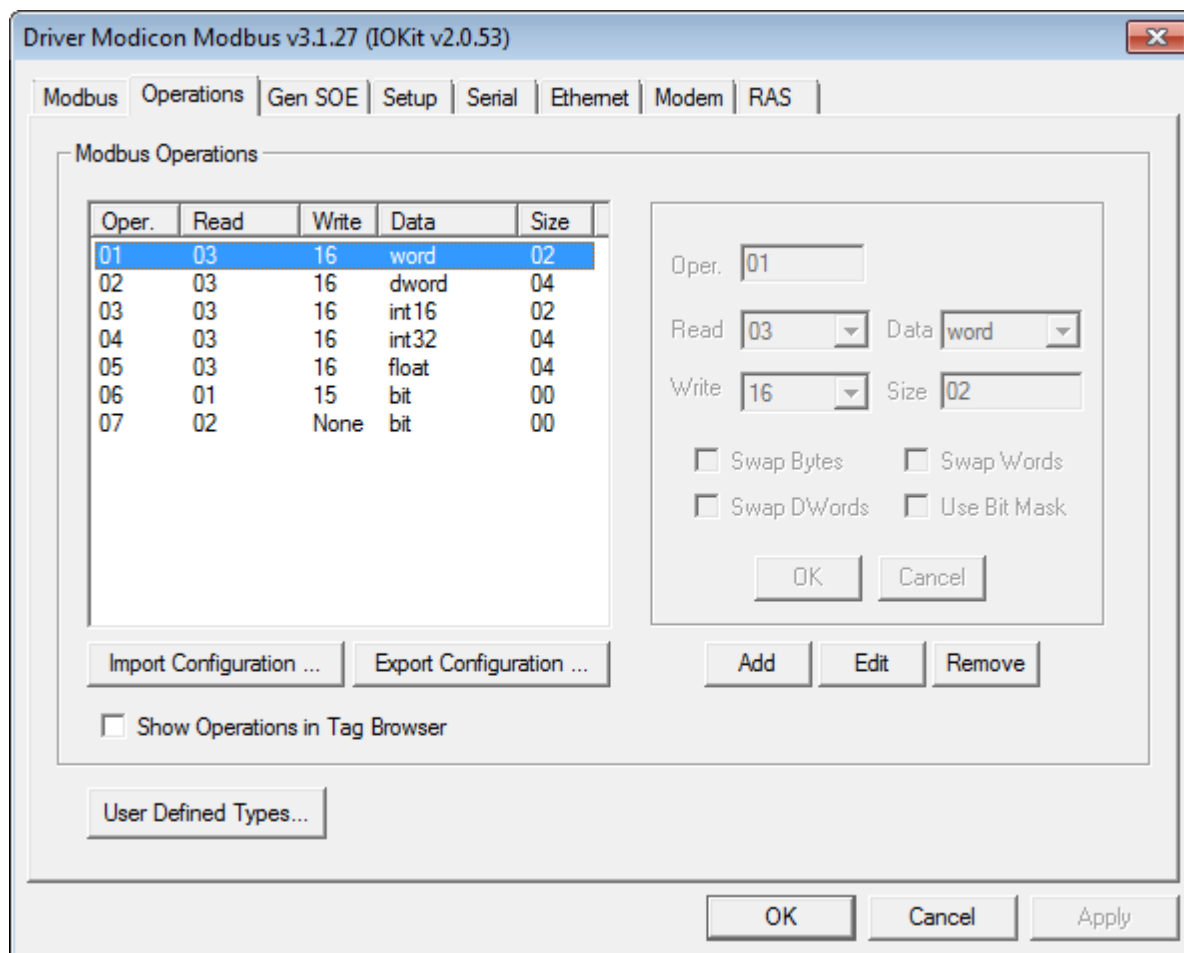
Adding a new operation

To configure this new operation, select a number for it (this number is used on I/O Tag's *N2/B2* parameter), which function to use for reading, and which function to use for writing, as well as informing a data type to be read or written by this Driver. Notice that, when clicking **Add**, this Driver already suggests a value that is not in use for this new operation.

For more information about supported data types, please check topic **Supported Data Types**. All other fields can be configured as needed. The next table contains a description of these fields.

Field options for operations

OPTION	DESCRIPTION
Size	A size **in bytes** of each element of the selected data type must be informed. This field is automatically filled in for fixed-size data types, such as **Byte**, **Word**, and **Int16**, and it must be filled in for **String** and **BCD** data types. For **Strings**, this size defines exactly the number of bytes sent or received for each **String** value, that is, for each Tag or Block Element. If the **String** read or written has a shorter size, the remaining bytes are filled in with zeroes to complete its configured size. The **String** data type in this Driver has no defined maximum limit size, this limit is the **maximum allowed by the protocol** for frame's data area of a certain function.
Swap Bytes	Indicates that this Driver must reverse the byte order, one by one, to retrieve a value.

OPTION	DESCRIPTION
Swap Words	Indicates that this Driver must reverse the byte order, two by two (in **Words**), to retrieve a value.
Swap DWords	Indicates that this Driver must reverse the byte order, four by four (in **DWords**), to retrieve a value.
Use Bit Mask	Enables a bit masking of registers, using the *N3/B3* parameter. This option only affects readings and can only be used with integer data types, signed or unsigned, with at least 16 bits of size (**Int16**, **Int32**, **Word**, or **DWord**). Operations with this option enabled cannot be used for writing. **For most users, it is recommended to use application's bit mapping, and leave this option deselected** (please check the **specific section**).

Protocol functions that can be configured in operations' **Read** and **Write** fields are described on topic **Supported Functions**. The next table describes each one of the available options.

Available options on Operations tab

OPTION	DESCRIPTION
Import Configuration	This option allows importing configurations for operations from versions prior to Modbus Master/ Slave Driver version 2.0, which stored these configurations on a modbus.ini file. This Driver does not use INI files anymore to store such configurations, which are now stored on the application file. For more information, please check topic **Import and Export Operations**.
Export Configuration	This option executes the opposite operation of the previous option, generating an INI file containing all operation configurations, in the current format or in the same format of this Driver's previous versions. This way, users can store operation configurations of a certain device on a file, and these configurations can be used by other applications. For more details, please check topic **Import and Export Operations**.
Show Operations in Tag Browser	If this option **is not selected** (default), templates of **Tags configured by Strings** (**Device** and **Item** fields) are displayed on Tag Browser. If **it is selected**, templates of **Tags numerically configured** (*N/B* parameters) are displayed on Tag Browser. When creating new instances of this Driver, this option is deselected by default. In legacy applications, when Driver's version is updated from a version previous to 3.1, this option is already selected, keeping the behavior of previous versions.
Add	Adds a new operation to the list.
Edit	Updates the selected operation on the list (equivalent to double-clicking an item).
Remove	Removes the selected operation from the list.

Supported Data Types

The next table lists Driver's native data types, which can be defined when configuring I/O Tags.

As explained on topics **String Configuration** and **Operations Tab**, the Modbus protocol itself only supports **Bit** and **Word** data types (16 bits) for the **most commonly used functions** implemented by this Driver (the only exception is currently **function 7**). All other Driver's data types are converted to **Word** at protocol's level, for reading from or writing to a device or slave device.

Also notice that this Driver supports **User-Defined Data Types**, defined as structures with elements composed by the native data types listed on the next table.

On the next table, data types use the same denominations of mnemonics for the **data type** field, when Tags are **configured by Strings**. For the old **numerical configuration**, the same denominations are also used on **Data** column of Driver's configuration window (on **Operations Tab**). In some cases, frequent alternative denominations are displayed between parentheses.

Available options for data types

TYPE	RANGE	DESCRIPTION
Char	-128 to 127	Eight-bit word, character. Writing must always occur in blocks with even size (**Words**)
Byte	0 to 255	Unsigned eight-bit word. Writing must always occur in blocks with even size (**Words**)
Int8	-128 to 127	Signed eight-bit word. Writing must always occur in blocks with even size (**Words**)
Int16	-32768 to 32767	Signed 16-bit integer
Int32	-2147483648 to 2147483647	Signed 32-bit integer
Word (or UInt)	0 to 65535	Unsigned 16-bit integer
DWord (or ULong)	0 to 4294967295	Unsigned 32-bit integer (**Double Word**)
Float	-3.4E38 to 3.4E38	32-bit floating point (**IEEE 754**) (four bytes: EXP F2 F1 0)
Float_GE	-1.427E+45 to 1.427E+45	32-bit floating point used by GE, not compatible with **IEEE 754**. It is used in GE GEDE UPS devices, with an eight-bit exponent $2^{[-128 \ldots +127]}$ and 24-bit mantissa $[-2^{23} \ldots +(2^{23}-1)]$. (four bytes: EXP F2 F1 F0). For more information, please check device's documentation
Double (or Real)	-1.7E308 to 1.7E308	64-bit floating point (**IEEE 754**)

TYPE	RANGE	DESCRIPTION
String	Does not apply	Text in **ANSI** format, with a determined number of eight bit ASCII characters (**Chars**)
BCD	Check description	BCD (*Binary-Coded Decimal*) numerical value. When using this data type, an application must provide a positive and integer decimal value, sent in **BCD** format, respecting the specified size. The **Size** field, for **BCD** types, refers to the number of bytes sent to represent a value. As in BCD encoding each figure is converted to a nibble, then the allowed values must have a maximum number of figures equal to double the size of the value specified in the **Size** field, that is, if a value of two is selected for the **Size** field, the maximum value that can be sent is 9999. Likewise, if **Size** is equal to four, the maximum value is then 99999999. Allowed values for the **Size** field for **BCD** types are two (**Word**) and four (**Double Word**). For more information about the BCD encoding, please check topic **BCD Encoding**
GE_events	Check description	Data type used when reading an event buffer (SOE) from a GE PAC RX7 PLC. Its definition is only allowed for operations that use the special reading function **GE SOE**. These events are returned as blocks with two Elements, with timestamps defined by the controller. For more information, please check topic **Reading an Event Buffer in GE PAC RX7 Controllers**
Bit	0 (zero) or 1 (one)	This data type is automatically selected when a bit-access function is selected. Bit-access functions are **01**, **02**, **05**, and **15**. The **Size** field is not used for **Bit** types. When using this data type, each Tag or Block Tag Element represents a bit

TYPE	RANGE	DESCRIPTION
SP_events	Check description	Data type used when reading events (SOE) in Schneider Electric relays from SEPAM 20, 40, and 80 series. Its definition is only allowed for operations that use the special reading function **SP SOE**. These events are returned as a three-Element Block, with a device-provided timestamp. For more information, please check topic **Reading Events in Schneider Electric Relays from SEPAM 20, 40, and 80 Series**
GenTime	1/1/1970 00:00 to 31/12/2035 23:59:59.999 (please check the next note)	Date and time type composed by an eight-bit structure, originally created for use when reading events using **generic SOE algorithm (GenSOE)**. This data type can be used with other Modbus protocol functions, in addition to **GenSOE**. This format is read internally as a structure of **Words**, and the only valid swap function for this type is **Swap Bytes**. Representation of this type in PLC memory is described on topic **GenTime Data Type**. For more information about this type, please check topic **Elipse Software's Generic SOE Reading Algorithm**
Sp_time	1/1/1970 00:00 to 31/12/2035 23:59:59.999 (please check the next note)	Date and time data type composed by an eight-byte structure, used by Schneider Electric relays from SEPAM 20, 40, and 80 series, usually to represent a timestamp. For more information, please check device's documentation
UTC64d	1/1/1970 00:00 to 31/12/2035 23:59:59.999 (please check the next note)	Date and time data type represented in **Double** format (**64-bit IEEE 754**), with seconds since 1/1/1970 00:00
UTC32	1/1/1970 00:00 to 31/12/2035 23:59:59.999 (please check the next note)	Date and time data type represented as an unsigned 32-bit integer (**DWord** or **UInt**), with seconds since 1/1/1970 00:00. This format does not represent milliseconds, which are always considered as 0 (zero)

GenTime Type

GenTime is a date and time data type originally defined and added to this Driver for use with **Elipse Software's Generic SOE Reading Algorithm**. It is, however, a generic data type that can be easily used with almost any PLC.

In an application, that is, in values of Tags and Block Tag Elements, as well as in Tag's **Timestamp** field, this data type, as well as all other Driver's date and time data types, is represented by an application's native date and time data type. For more information about other date and time data types supported by this Driver, please check topic **Supported Data Types**. For more information about application's date and time data types, please check their respective user's manual (there are some differences between VBScript used in Elipse SCADA and the one used in E3 and Elipse Power).

In the PLC or slave device, this data type is represented by a structure composed by four 16-bit registers (eight bytes), as displayed on the next table.

<div align="center">

Structure of registers

</div>

OFFSET	CONTENT	BIT MAPPING (16 BITS)	RANGE (DECIMAL)
0	Year	AAAAAAAA AAAAAAAA	Between 0 and 65535
1	Day / Month	DDDDDDDD MMMMMMMM	Between 0 and 65535
2	Hour / Minute	HHHHHHHH MMMMMMMM	Between 0 and 65535
3	Second / Millisecond	SSSSSSMM MMMMMMMM	Between 0 and 65535

The base address (*offset* 0), for attribution on Tag's *N4/B4* parameter accessing data, contains the year. The next register (*offset* 1) contains the day as the most significant byte and the month as the least significant byte. The offset 2 contains the hour represented in the most significant byte and the minutes in the least significant byte. The fourth register contains the four most significant bits of the **Word** representing seconds, and the remaining bits (the two least significant of the most significant and the integer's least significant byte) representing milliseconds.

Notice that each Tag referencing this data type forces a Driver to read a block of four Modbus registers in the device to represent the value of each Tag or Block Element to return a valid value.

Advantages of this data type are its simplicity (it is easily generated in a PLC ladder), its milliseconds precision, and its relative compression, as it does not need native support by the PLC or slave device.

User-Defined Data Types

User-defined data types, or structures, after configured on **User Defined Types** configuration window, can be

---

used by Driver operations the same way as pre-defined data types.

These data types are in fact structures whose elements may have different **native data types**, that is, a user-defined data type is merely a structure defined based on Driver's pre-defined data types (native or built-in data types), allowing users to configure Block Tags where each Element may have a different native data type.

Users can use almost all data types pre-defined by the Driver in their structures. The only data types not allowed are **Bit** data types, eight-bit data types, variable-size data types, such as **String** and **BCD**, and event data types linked to specific SOE functions.

Once a data type is defined, users can link it to any Tag, as long as it uses Modbus functions supporting **Words**, that is, it is not allowed to link a user-defined data type to an operation that defines as its reading function (**Read**) the **01** function, for example, because that function only reads bits.

In addition to the definition of structure elements, whose values are returned in Block Elements, users can also define the type of a Tag's timestamp, as well as the default address for that structure, address that is used for the *B4* parameter of Tags available using E3's Tag Browser.

Applications

User-defined data types were originally implemented for use with **Elipse Software's Generic SOE Reading Algorithm (Gen SOE)**, as this algorithm performs a reading on data structure tables.

In addition to using with SOE's generic algorithm, this feature can also be used to group different data types on a single Block Tag, optimizing communication in applications without Superblocks, such as Elipse SCADA, or if the device in use for some reason does not allow using Superblocks (please check topic **Superblock Reading**).

Configuring User-Defined Data Types

Configuration of user-defined data types is performed on a specific window, by clicking **User Defined Types** on **Operations Tab** of Driver's configuration window, as shown on the next figure.
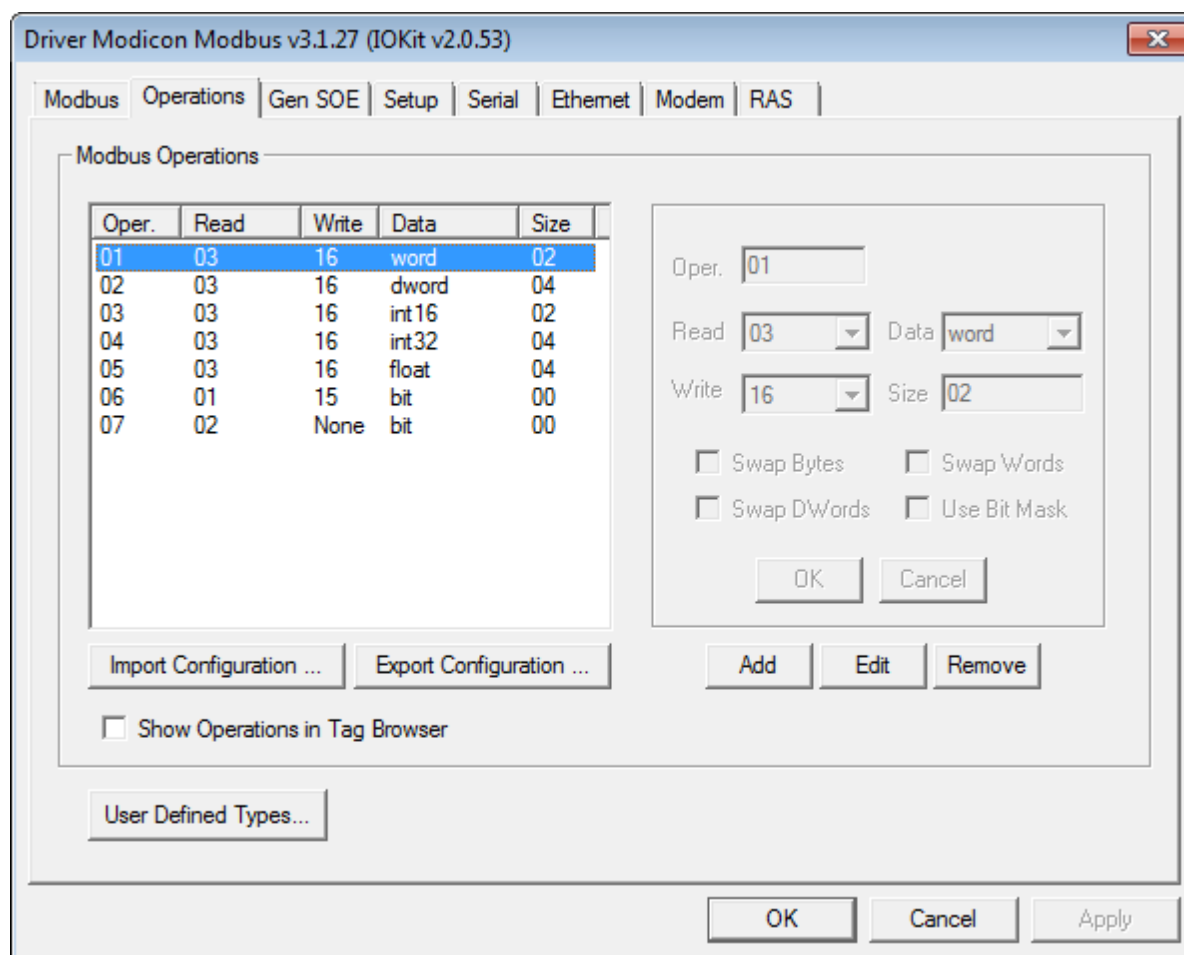
Operations tab of Driver's configuration window

This window to configure user-defined data types allows editing files with structure configurations. When opening this window for the first time, it displays the default configuration file (with comments), which defines three example types that appear commented with multiple-line comments ("/*" and "*/"), as explained later.

The next figure displays the configuration window for user-defined data types, with a small file defining these three example data types.
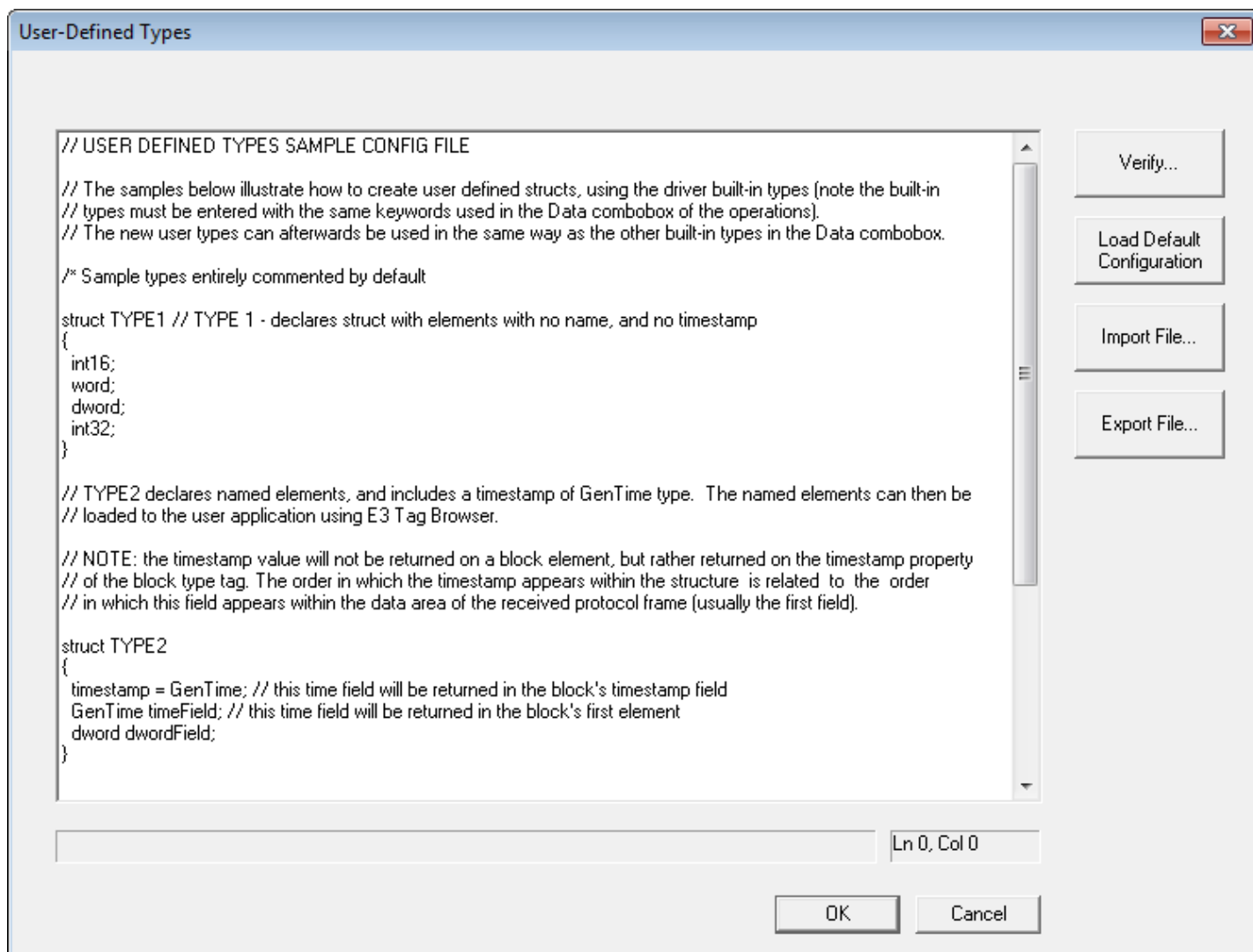
Configuration of user-defined data types

Notice that line comments always start with "//", identifying what is on the right, on the same line, as a comment, following the pattern for line comments of the C++ programming language, which is also used for other languages such as Java and C#.

Comments with multiple lines are also supported, still following the same C++ syntax, starting by "/*" and finishing by "*/". Notice that the example file that accompanies this Driver already applies that comment format to its example types, leaving them commented by default. Remove the lines indicated by "/* Sample types entirely commented by default" and "*/" (without quotation marks) so that all three example data types are ready to use.

As the text of the configuration file changes, the status bar displays the result of this file's syntactical analysis, in real-time. This status bar displays a "Status: OK!" message if no errors are detected in this file.

The line and column of cursor position in the edit box are always displayed on the right side of the status bar. Errors displayed on the status bar always reference the line and column number where it was detected.

This check can be also fully performed by clicking **Verify** and, in case of any error, cursor is then automatically placed on the error line.

The definition of each type has the following syntax (elements inside brackets are optional):

```
struct <Type Name>
{
  [timestamp = <date and time type>;]
  [DefaultAddress = <address>;]
  <type> [name of element 1];
  <type> [name of element 2];
  <type> [name of element 3];
[...]
  <type> [name of element n];
}
```

Where:

- **struct**: Keyword, lower case, starting the definition of a user-defined data type.

- **<Type Name>**: Name by which this new data type is identified by the Driver. This is the name displayed on the combo box **Data**, when configuring operations. Its maximum size is six characters.

- **timestamp**: Optional field indicating that this structure contains a device-defined timestamp, which must be returned in Tag's **Timestamp** field. Each structure can have only one timestamp. The order in which it appears on this structure affects the position in which this field is read in the frame returned by a device (notice that in Tags this value is returned only in the **Timestamp** field). Any date and time data types supported by this Driver can be defined. In the current version, this Driver supports date and time data types **GenTime**, **Sp_time**, **UTC64d**, and **UTC32**. For more information about data types, please check topic **Supported Data Types**.

- **DefaultAddress**: Optional field specifying a default address value, used to fill the *B4* parameter of Tags in Tag Browser referencing operations containing this structure. Address values can be provided in decimal or hexadecimal format. To use the later format, users must precede the number with the "0x" prefix (for example, using "0x10" to encode the decimal value 16 to hexadecimal).

- **<date and time type>**: Pre-defined date and time data types for this Driver, which can be used as a timestamp by a slave device. In the current version of this Driver, native data types **GenTime**, **Sp_time**, **UTC32**, and **UTC64d** are accepted.

- **<type>**: Element's data type. It must be defined as one of Driver's pre-defined data types, and written as it appears on the combo box **Data**, on the configuration window for operation parameters, respecting lower and upper case. **Bit** data types, eight-bit data types, and variable-size data types, such as **BCD** and **String**, are not allowed.

- **[name of element]**: Optional parameter defining a name for each Block Element. If defined, determines a name for Block Elements in Tags displayed in E3's Tag Browser. If not defined in structure's declaration, this Driver then sets default names to Elements in Tag Browser, with the keyword "Element" followed by the index of that Element in the Block ("Element1", "Element2", etc.).

Importing and Exporting

The **Import File** and **Export File** options allow importing and exporting a configuration file with user-defined data types to text files on disk. These options can be used to create backup copies of a file, or to share it among several Drivers. This file is always saved and read using Windows default ANSI format (**Windows-1252** Charset). Future versions of this Driver may include support for other formats.

In addition to copying a file to disk, users can also use shortcut keys CTRL + A (**Select All**), CTRL + C (**Copy**), and CTRL + V (**Paste**) to copy and paste this file's content to another Text Editor.

The **Load Default Configuration** option loads the default configuration file again in the editor, the same file loaded when the configuration window is opened for the first time.

NOTE: When clicking **Cancel**, all changes performed on the file are discarded by this Driver. By clicking **OK**, this file is then stored in the application. This operation performs a full check on this file and if any error is identified, this error is then displayed and this window is not closed. If users want to save these changes with pending errors, export this file or copy and paste it to another Text Editor.

Using User-Defined Data Types with Tags Configured using Strings

Names for user-defined data types can be used as mnemonics for the **Type** field in the **Item** parameter, such as with Driver's native data types, as long as this name has been previously declared, as defined earlier in this topic.

IMPORTANT: As in E3 the **Item** field is not case-sensitive, to use user-defined data types in this field, the user-defined data types must not be case-insensitive, that is, users must not define, for example, a data type named as "type1" and another one as "TYPE1". If this happens, users cannot use that user-defined data type in the **Item** field until that name is fixed.

For more information on Tag configuration using **Strings**, please check topic **String Configuration**. Example:

1. Read or write Holding Registers (functions **03** and **06**) of address 100 from a device with *Id* 5, interpreted as a user-defined data type named "mytype", with *Slave Id* in the **Item** field:

   a. **Device**: "" (empty **String**)

   b. **Item**: "5:shr100.mytype"

NOTE: Swap options (byte order) for user-defined data types are only effective on the elements of the defined structure, not on the entire structure, that is, if the **Swap Words** option is enabled, all elements with more than 16 bits have their **Words** swapped. 16-bit elements, however, are not changed.

Using User-Defined Data Types on Numerical Configuration

After defining new data types in the configuration file on the **User-Defined Types** window, these data types are available for use in Driver operations. Remember that only operations that use Modbus functions for access to 16-bit registers, such as functions **03**, **04**, **06**, and **16**, allow user-defined data types.

The next figure displays the configuration of a new operation that uses a user-defined data type (structure) named **TYPE3**, showed on the previous example, after clicking **Add**.
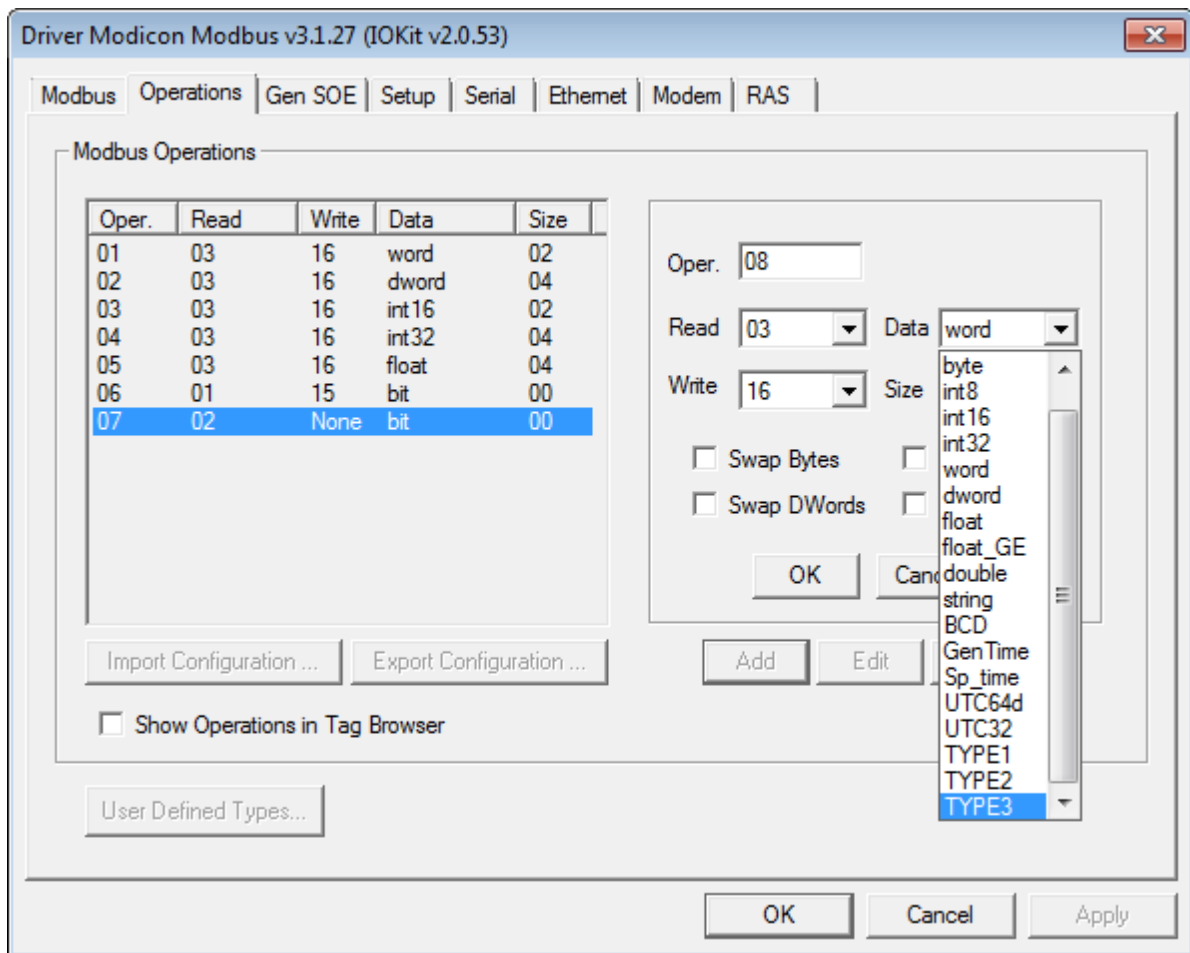
Adding a user-defined data type

NOTE: Swap options for user-defined types are only effective on the elements of the defined structure and not on the entire structure, that is, if the **Swap Words** option is enabled, all elements with more than 16 bits have their **Words** swapped. 16-bit elements, however, are not changed.

After defining a new operation, by using the new type **TYPE3**, define a Block Tag with that same data type and size equal to the number of elements of that structure, as shown on the next figure.
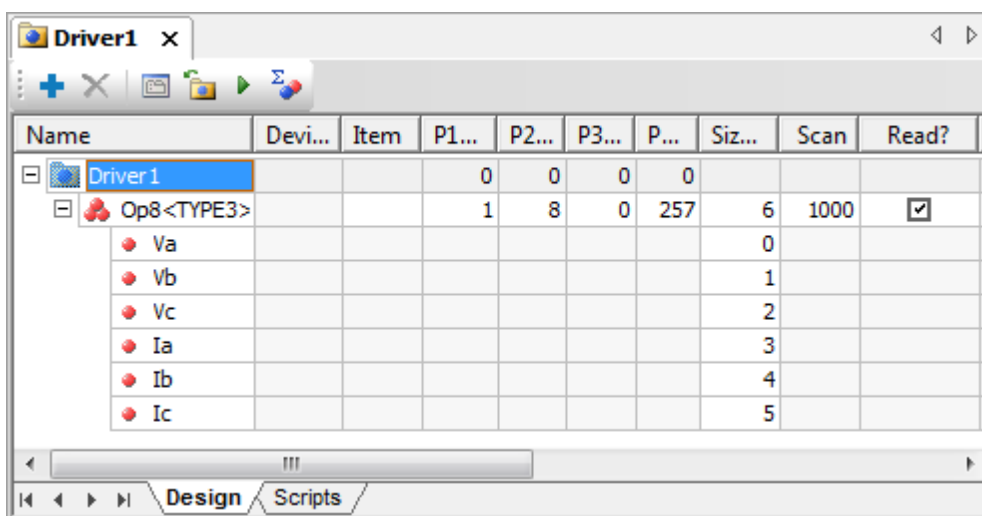
Declaring Tags using structures in E3 or Elipse Power

If a name was defined for each element of this structure, then users can use E3's Tag Browser to include a Block Tag referring the desired data type in an application, without typing it again. To use this feature, users must select the **Show Operations in Tag Browser** option on **Operations** tab. The next figure shows how to perform this procedure.
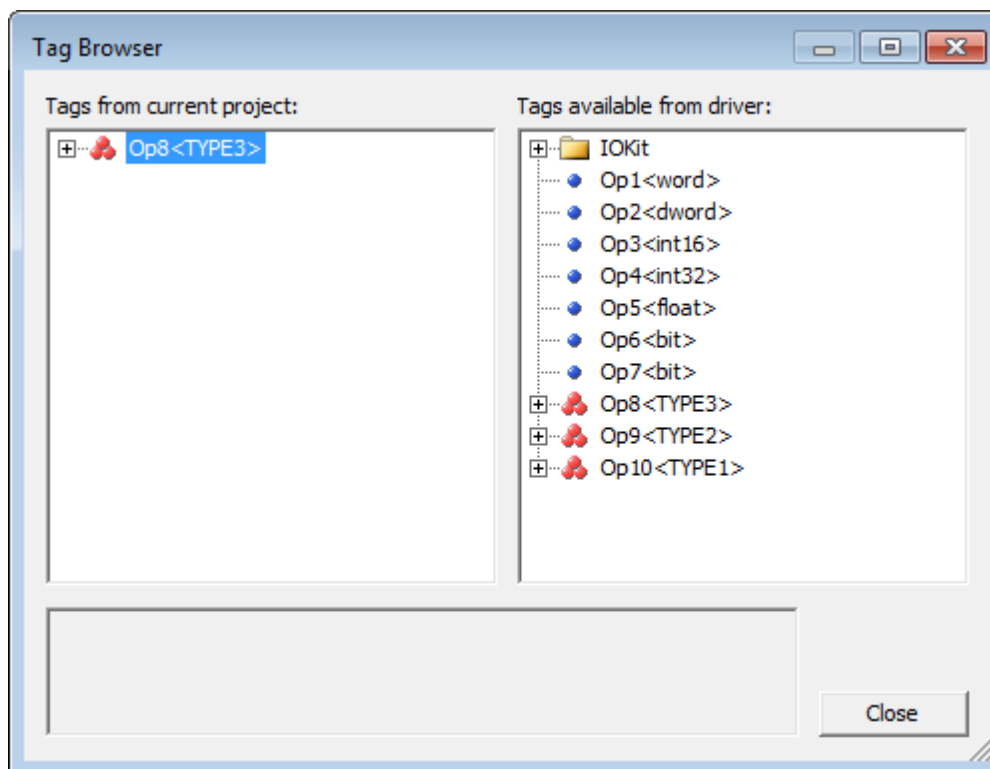
Using Tag Browser to define Tags using structures

As this figure suggests, click 🗁 on Driver's **Design** tab to open Tag Browser and drag the desired data type from the list **Tags available from driver** to the list **Tags from current project**.

Event-Reported Reading

User-defined data types or structures are generally used to define events in PLC's memory, and can be used with **Elipse Software's Generic SOE Reading Algorithm**. However, if users want to read events organized in PLC's memory, such as a sequence of structures, in an operation that only uses **protocol's public reading function**, that is, without using special functions with the SOE algorithm, such procedure can be performed in two ways:

- **Block Reading**: Create a Block with a number of Elements that is a multiple of the number of elements of user's data structure. For example, a user-defined data type or structure with two elements that represent events collected on an arrangement in PLC's memory. If users want to read a block with five events, they must define a Block Tag with 10 Elements. Thus, a single reading from this Tag retrieves all events at once.

- **Event-Reported Reading**: Uses a sequence of Tag's **OnRead** events to read a data block. With it, considering the example on the previous item, instead of creating a Tag with 10 Elements, users only need to create a single Block Tag with two Elements, configuring its *B3* parameter as "5". This way, when performing a Tag reading, E3 calls Tag's **OnRead** event five times, and at each call the Elements and properties of this Block Tag contain data related to a specific event. The most common usage for Tags reported by events is storing events read directly from a historical database. This is easily performed using Historic's **WriteRecord** method previously linked to this Tag, on the **OnRead** event of the event-reported Tag. For more information, please check topic about Event-Reported Tags on **E3 User's Manual**.

In other words, every I/O Tag using structures and using a **protocol's public reading function** (this resource does not work for **special SOE functions**), becomes an Event-Reported Tag if its *B3* parameter is configured with a non-null value.

In case of special SOE functions, such as the **Gen SOE function**, the event-reported return is defined by the function's proprietary algorithm itself.

For more information about I/O Tag configurations, please check topic **Configuring an I/O Tag**.

> **IMPORTANT**: When reading mass memory events in event-reported Tags in E3, disable Tag's dead band (the **EnableDeadBand** property configured as False) and also the linked Historic object (the **DeadBand** property equal to zero), to avoid loosing events with close values. It is also important to disable the historic by scan (in E3, the **ScanTime** property equal to zero). This ensures that new events are only stored using the **WriteRecord** method, executed in Tag's **OnRead** event, thus avoiding duplicated events.

Importing and Exporting Operations

Importing and exporting operations can be performed on Driver's **Operations** tab, by clicking **Import Configuration** or **Export Configuration**, as shown on the next figure.
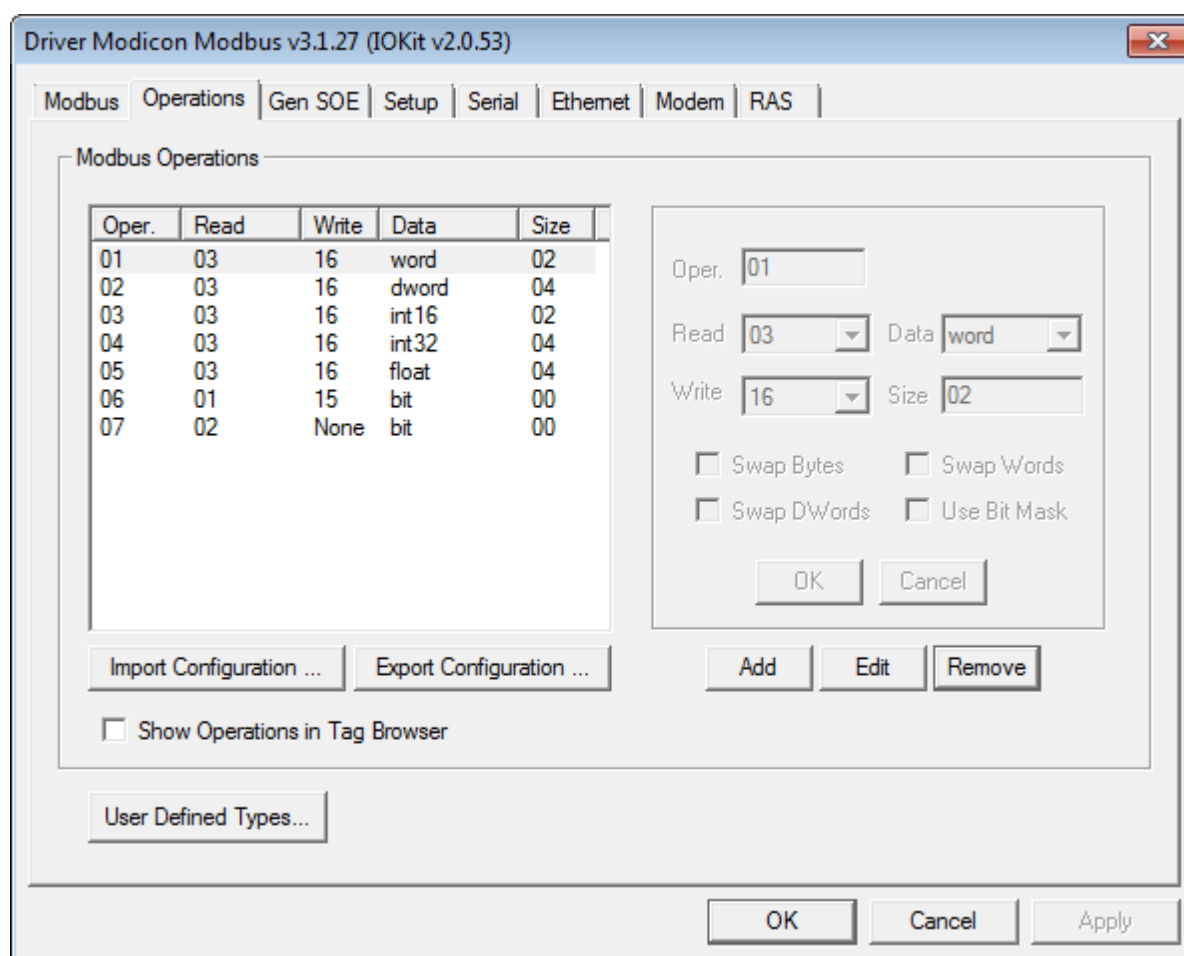
Options for importing and exporting operations

These options allow importing and exporting operation configurations displayed on **Modbus Operations** frame to INI files.

On this Driver's versions previous to 2.00, operation configurations were performed on a modbus.ini file, which was loaded during Driver's initialization. modbus.ini files of these old versions still can be loaded on the current Driver version, by using the import option.

> **NOTE**: Driver operations used to be called **Driver Functions** in initial versions. This name was then changed to **Driver Operations** due to some cases when users confused it with **Protocol Functions**.

Importing

Importing configuration files is very simple. Click **Import Configuration** and select an INI file. The Driver must load operation configurations, which immediately appear on **Modbus Operations** frame. This Driver allows importing files generated on previous versions.

Exporting

Exporting files with operation configurations can be performed to share the same operation configurations among different Driver objects, as well as performing backup copies of operation configurations of a certain device.

Another possible use is exporting configurations to a modbus.ini file compatible with previous Driver versions, allowing to load these configurations on previous versions. This is not advisable but, if inevitable in case of legacy applications, users must consider the following situations.

When clicking **Export Configuration**, a window with two options is then opened, as in the following figure.

Export options
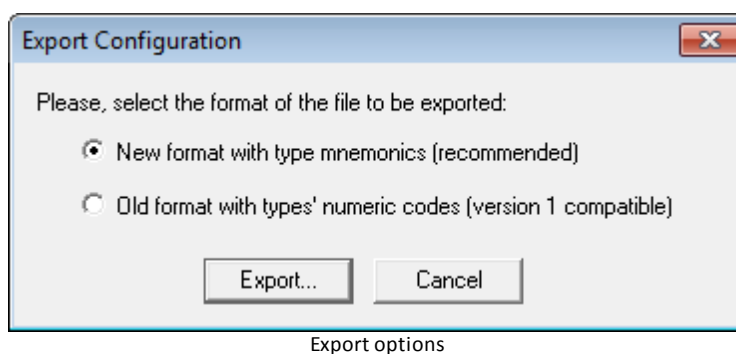
On this window, users must select between exporting based on the new format (**New format with type mnemonics**), with displayed data types defined as **Strings** (mnemonics), or based on the old format (**Old format with types' numeric codes**), in which data types were identified by a numerical value, corresponding to the position where they appeared on the combo box **Data** on **Operations** tab.

This new format is more legible, making it easy to debug, and is used on the most recent versions of this Driver, and it is the most recommended option.

The old format, on the other hand, must be selected only if it is indispensable to export for versions previous to version 2.08 of this Driver.

Notice that, to export modbus.ini files successfully to load them to versions previous to 2.00, operations cannot define any new data type not implemented on the destination version, nor define operations that use the **Use bit mask** parameter, or the import operation may fail.

Generally, it is recommended to avoid exporting configurations to previous versions, preferring updated Driver versions.

Gen SOE Tab

The goal of this tab is to concentrate configuration options for **Elipse Software's Generic SOE Reading Algorithm**. The next figure displays all options on this tab.
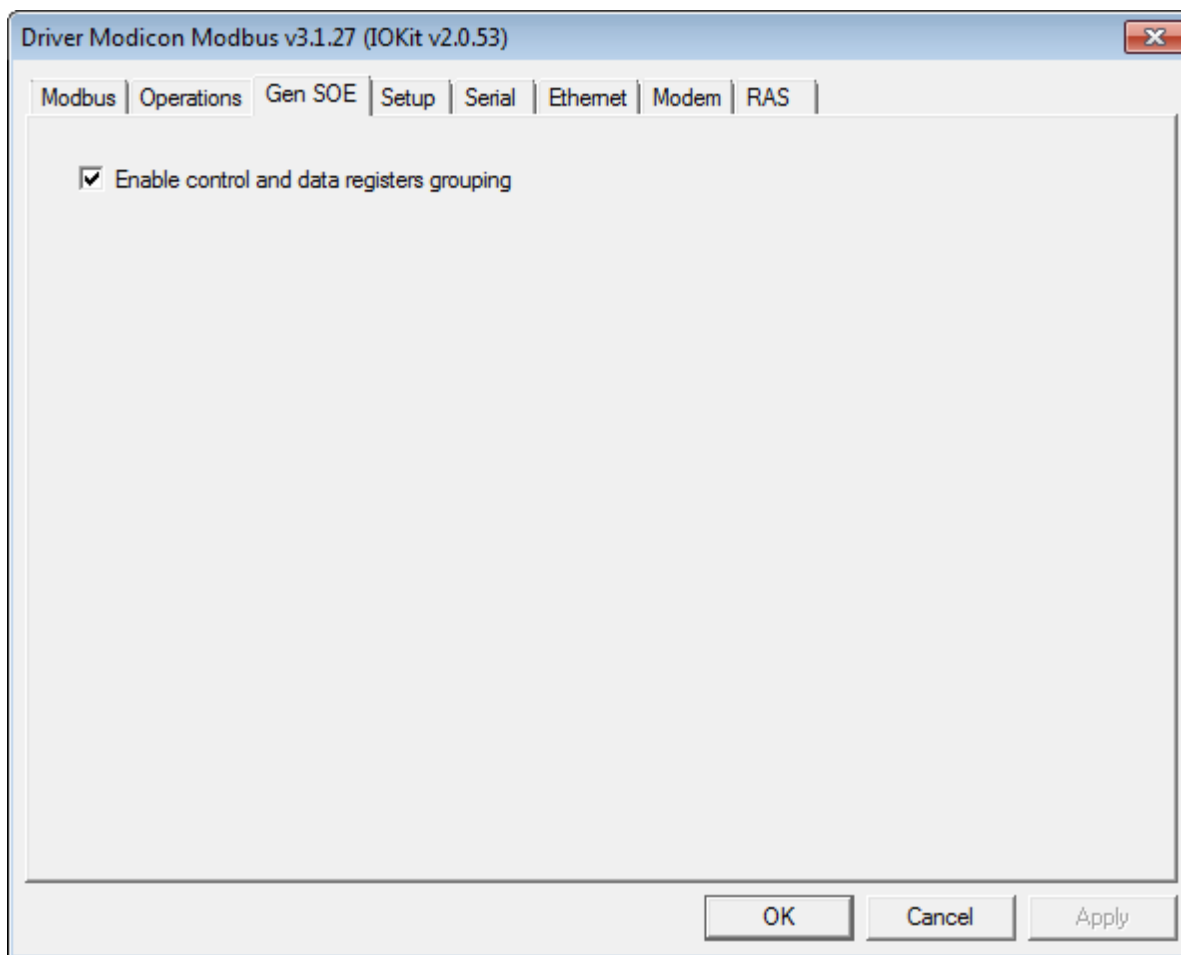
Gen SOE tab

On the current version, there is only one configuration option:

- **Enable control and data registers grouping (default True)**: Enables grouping control and data registers, to perform the minimum possible number of readings. If this option is enabled, this Driver starts reading tables already trying to read the maximum number of protocol-allowed registers, be them control or data registers, and possibly reading the entire table in a single reading. Such procedure usually optimizes Tag scanning the same way as Superblocks, because the time spent for reading large blocks is usually less than the time needed to perform several readings of the same amount of data, although this may depend on the PLC. ATOS PLCs do not allow grouped reading of control registers and data structures, thus requiring this option to be disabled.

Offline Mode Configuration

Driver configurations can also be accessed at run time if a Driver starts in **Offline** mode, as explained on **IOKit User's Manual**, by using the **String**-type parameters described on the next table.

Available parameters

PARAMETER	TYPE
ModiconModbus.ModbusMode	Integer: • **0**: Modbus RTU • **1**: Modbus ASCII • **2**: Modbus TCP
ModiconModbus.Olderaddr	**Boolean** (0 or 1): • **0**: Data is addressed from 0 (zero) • **1**: Data is addressed from 1 (one)
ModiconModbus.UseDefaultSlaveAddress	**Boolean** (0 or 1)

PARAMETER	TYPE
ModiconModbus.DefaultSlaveAddress	Unsigned integer
ModiconModbus.UseSwapAddressDelay	**Boolean** (0 or 1)
ModiconModbus.SwapAddressDelay	Integer, with a delay interval in milliseconds. **NOTE**: Obsolete option kept for compatibility reasons. For new applications, please use the **Inter-frame delay** option on IOKit's **Serial** tab
ModiconModbus.WaitSilenceOnError	**Boolean** (0 or 1)
ModiconModbus.EnableCMSAddressing	**Boolean** (0 or 1)
ModiconModbus.EnCustomizeMaxPDUSize	**Boolean** (0 or 1)
ModiconModbus.MaxPDUSize	Integer
ModiconModbus.ConfigFile	**String** containing a configuration file with Driver operations. This file can be exported and imported on **Operations** tab on Driver's configuration window
ModiconModbus.EnableReconnectAfterTimeout	**Boolean** (0 or 1): • **0**: Time-out does not generates a disconnection from physical layer • **1**: In case of a time-out, when in an **Ethernet** physical layer, this Driver performs a physical layer disconnection and a reconnection
ModiconModbus.UserTypesConfigFile	Configuration **String** for **user-defined data types** (structures). This is the same configuration file available on Driver's configuration window (**User-Defined Types**)
ModiconModbus.EnableGenSOERegGrouping	**Boolean** (0 or 1): • **0**: Event-reading algorithm first reads control registers, and then event data • **1**: Generic SOE reading is grouped to its maximum, not only reading control registers first, but also the maximum possible events

For more information about **Offline** configurations at run time, please check **IOKit User's Manual**.

Configuring Tags

This topic describes configurations for several types of Tags supported by this Driver. Tags are divided into two categories, described on the following topics:

- **Configuring an I/O Tag**

- **Configuring Special Tags**

Configuring an I/O Tag

Driver's I/O Tags are Tags that allow communication with devices.

I/O Tags allow reading and writing Modbus registers from and to slave devices, by using **Modbus protocol functions**, or even **special functions**. This Driver does not differentiate between Block and simple Tags, in case of I/O Tags, that is, I/O Tags work the same way as a Block with a single Element.

Data is read from a device using protocol-supported formats, that is, registers with integer values of 16 bits,

bytes, or sets of bits, depending on the protocol's function used. For more information about protocol functions, please check specifications at *protocol's official website*.

I/O Tags can be configured in two ways, described in the next topics:

- **String Configuration**: This is the newest method that can be used with Elipse E3, Elipse Power, and Elipse OPC Server, recommended for new projects. It is not supported by Elipse SCADA

- **Numerical Configuration**: Old method used in Elipse SCADA

String Configuration

The **String** configuration of **I/O Tags** is performed using the **Device** and **Item** fields of each Tag.

That new configuration method does not work with Elipse SCADA, which still uses the old **numerical configuration** (*N* and *B* parameters).

N and *B* parameters are not used when configuring by **Strings** and they must be left in 0 (zero).

Configuration by **Strings** makes Tag configuration more readable, making it easy to configure and maintain applications.

Block Reading

Tags configured by **Strings** can be simple Tags or Block Tags, with their **Device** and **Item** fields with the same syntax.

> **NOTE**: The grouping service of Tags performed by **Superblocks** is not available for Tags using configuration by **Strings**. If users want optimization through superblock readings, application Tags must use only **Numerical Configuration**.

Device Field

In the **Device** field, the *Slave Id* (device's identification address) must be provided as a number between 1 (one) and 255 followed by a colon, such as "1:", "101:", "225:" etc.

Please remember that, in Modbus protocol, the *Slave Id* 255 is reserved for broadcasting, which only makes sense if used by writing operations, because there is no return from devices, or else there would be conflicts.

Optionally, the *Slave Id* may appear at the beginning of the **Item** field, explained later, and in this case the **Device** field must be left empty. This option is detailed next.

Item Field

The **Item** field must provide all addressing and operation information to be performed by a Tag, using the following syntax:

`<address space><address>[.<type>[<type size>]][.<byte order>][/bit]`

Where parameters between brackets are optional.

As mentioned earlier, users can alternatively provide a *Slave Id* at the beginning of the **Item** field, such as the next example:

```
<slave id>:<address space><address>[.<type>[<type size>]][.<byte order>][/bit]
```

In this case, as already explained, users must leave the **Device** field empty.

The next examples show the most common usage cases (notice that quotation marks must not be added to the application):

1. Reading or writing a Holding Register (functions **03** and **16**) on address 100 of a device with *Id* 1, and *Slave Id* provided in the **Device** field:

    a. **Device**: "1:"

    b. **Item**: "hr100"

2. Reading or writing a Holding Register (functions **03** and **16**) on address 120 of a device with *Id* 3, and *Slave Id* provided in the **Item** field:

    a. **Device**: "" (empty **String**)

    b. **Item**: "3:hr120"

3. Reading or writing a Coil (functions **01** and **15**) on address 65535 (FFFFh) of a device with *Slave Id* 4, here provided in the **Item** field:

    a. **Device**: "" (empty **String**)

    b. **Item**: "4:cl&hFFFF" (or optionally "4:cl65535")

The next figure shows examples of **String**-configured Tags on E3 Developer's Tutorial.

String-configured Tags on E3 Developer's Tutorial

Mandatory and Optional Fields

Mandatory fields for all Tags are described next and individually detailed later in this topic:

1. **Address space**: A mnemonic defining a set of protocol's reading and writing functions to use (please check the **table** with all supported mnemonics later on this topic).

2. **Address**: A numerical value identifying the item's address (register or bit) to read or write within the defined address space. These values can be provided in decimal, hexadecimal, or octal. For decimal values, no prefix is needed, or optionally a "&d" prefix can be used. For hexadecimal values, use the "&h" prefix (for example, "&hFFFF"). For octal values, use the "&ho" prefix (for example, "&o843"). This address can have an offset relative to the address sent in the communication frame, which depends on the convention used by the manufacturer. When in doubt about addressing conventions, please check topic **Addressing Tips (Modbus Convention)**. Particularly, check whether the device implements the default offset of protocol's Data Model (please check the **Data Address Model Offset** options on **Modbus tab**).

The next fields are **optional**, used for extensions to the default protocol or for compatibility with devices not fully compliant with the protocol (they are also individually detailed later in this topic):

1. **Type**: Defines how bytes from the data area of the communication frame must be interpreted. If omitted, the default types of the protocol for the respective functions are used, that is, **Word** for functions to access registers and **Bit** for functions to access digital data (Coils and Discrete Inputs). Please check the **mnemonics table** for all supported types later in this topic.

2. **Type size**: Users must specify this field only for variable-size types, such as **BCD** and **String**. Its numerical value indicates the size of the type in bytes.

3. **Byte order**: Mnemonic indicating the byte order of numerical values. If omitted, protocol's default order is then used, with the most significant bytes coming first in the communication frame, which is called *big endian*. Please check more information on the specific section about this option, later in this topic.

4. **Bit**: Allows returning a specific bit of an integer value, which obviously only makes sense in Modbus functions returning integer values (**Words**). Usually, users are advised to not use this feature, preferring application's bit mapping. Bit 1 (one) is the least significant and, the greater the value, the most significant is the bit. The maximum allowed value obviously depends on the type size, which is currently 64 for **Double** types. This field corresponds to the old **Use bit mask** option on **numerical configuration**. Please check more information on this option on topic **Operations Tab**.

Exceptions

Modbus protocol's **functions 20 and 21**, which access files, use a slightly different syntax from the one described previously:

```
fr<file>.<register>[.<type>[<type size>]][.<byte order>][/bit]
```

Example:

- **Device**: "" (empty **String**)

- **Item**: "1:fr4.101" (reading register 101 from file 4 on *Slave Id* 1)

Specifically for the **GenSOE special function** (*Elipse Generic SOE*):

```
elsoe<N>.<initial addr.>[.<type>[<type size>]][.<byte order>][/bit]
```

Specifically for the **SP SOE special function** (*Sepam SOE*), to read all events:

```
spsoe<event type>.<initial addr.>[.<type>][.<byte order>][/bit]
```

Specifically for the **SP SOE special function** (*Sepam SOE*), to read events from specific points:

```
ptspsoe<event type>.<event addr.>
```

Specifically for the **GE SOE special function** (*GE PAC RX7 SOE*):

```
gesoe<tag type + point index>.<queue base addr.>
```

Please check specific topics about the special functions mentioned previously for more information about configuring the respective Tags using **Strings**.

Address Space

Instead of explicitly defining Modbus functions or special reading and writing functions to use, as performed on the old **numerical configuration** and its concept of **operations**, when configuring using **Strings** users define an address space through mnemonics listed on the next table, already linked to protocol's pre-defined functions and their respective native data types.

Address spaces and mnemonics

ADDRESS SPACE	MNEMONIC	NATIVE TYPE	READING OR WRITING FUNCTIONS	COMMENTS
Holding Register	hr	**Word** (16-bit)	Functions **03** and **16**	Functions **03** and **16** are protocol's most used ones (Modbus class 0)
Single Holding Register	shr	**Word** (16-bit)	Functions **03** and **06**	Function **06** writes to the same registers of function **16**, the difference is that function **16** can only write in Blocks, while function **06** writes to a single register at a time, but with less overhead.
Coil	cl	**Bit**	Functions **01** and **15**	
Single Coil	scl	**Bit**	Functions **01** and **05**	Function **05** writes to the same registers of function **15**, but cannot write in Blocks, therefore with less overhead
Discrete Input	di	**Bit**	Functions **02** and **None** (read-only)	
Input Register	ir	**Word** (16-bit)	Functions **04** and **None** (read-only)	
Exception Status	es	**Byte**	Functions **07** and **None** (read-only)	
File Register	fr	**Word** (16-bit)	Functions **20** and **21**	
ABB MGE 144 - Mass Memory Reading	abbmge	**Word** (16-bit)	Functions **65 03** and **None** (read-only)	
ABB MGE 144 - Reset	abbmge.rst	Not used	Functions **None** for reading and **65 01** for writing	
ABB MGE 144 - Zeroes Maximum and Minimum Memory	abbmge.rstmxmn	Not used	Functions **None** for reading and **65 02** for writing	
GE PAC RX7 SOE - Reading	gesoe	**GE_events**	Functions **GE SOE** for reading and **None** for writing	

ADDRESS SPACE	MNEMONIC	NATIVE TYPE	READING OR WRITING FUNCTIONS	COMMENTS
SEPAM SOE Events	spsoe	**SP_events**	Functions **SP SOE** for reading and **None** for writing	Collects from the meter and returns a structure (with **SP_events** type) to the Tag with events from any points (please check topic **SEPAM SOE**)
SEPAM SOE Single Point Events	ptspsoe	**Int32**	Functions **SP SOE** for reading and **None** for writing	Collects from the meter and returns to the Tag an integer value from the **Edge** field, relative to events from a specific point (please check topic **SEPAM SOE**)
Elipse Generic SOE	elsoe	**Word** (16-bit)	Functions **GEN SOE** for reading (Modbus function **03** with additional algorithms) and **16** for writing	

Data Types

The table on the previous section lists all protocol's native data types, according to the Modbus functions used, as well as some specific data types used in **special functions** (non-standard). For Tags returning these native data types, the *Data Type* parameter can be omitted from the **Item** field's **String**.

If users must interpret native data in a different way, which is common among devices using Modbus, they must specify the data type to use, as explained in this section.

A list with all native data types supported by this Driver, as well as their description, can be checked on topic **Supported Data Types**.

The next table lists all mnemonics used in the *<type>* parameter of the **Item** field for each supported data type, Driver-native, and also an alias or an alternative name.

Supported data types

TYPE	MNEMONIC	ALIAS
Char	char	ch
Byte	byte	by or u8
Int8	int8	i8
Int16	int16	i16
Int32	int32	i32
Word or UInt	word	u16
DWord or ULong	dword	u32
Float	float	f

TYPE	MNEMONIC	ALIAS
Float_GE	float_GE	fge
Double or Real	double	d
String	string	s
BCD	BCD	bcd
GenTime	GenTime	gtm
Sp_time	Sp_time	sptm
UTC64d	UTC64d	-
UTC32	UTC32	-

User-Defined Data Types

In addition to the data types listed on the previous table, users can also provide mnemonics for user-defined data types or structures (please check topic **User-Defined Data Types**).

To use user-defined data types in the **Item** field, however, the names of these data types must not be case-insensitive, as the **Item** field does not differentiate upper and lower case. If that happens, the Driver does not allow using these data types in the **Item** field (please check topic **User-Defined Data Types**).

Examples

1. Reading or writing Holding Registers (functions **03** and **16**) to or from address 100 of a device with *Id* 1, interpreted as a **DWord**, with *Slave Id* in the **Device** field:

   a. **Device**: "1:"

   b. **Item**: "hr100.u32" or "hr100.dword", or if a hexadecimal is convenient, "hr&h64.u32"

2. Reading or writing Holding Registers (functions **03** and **16**) to or from address 150 of a device with *Id* 3, interpreted as a **Float**, with *Slave Id* in the **Item** field:

   a. **Device**: "" (empty **String**)

   b. **Item**: "3:hr150.f" or "3:hr150.float", or if a hexadecimal is convenient, "3:hr&h96.f"

3. Reading or writing Holding Registers (functions **03** and **16**) to or from address 1500 of a device with *Id* 5, interpreted as a **Double**, with *Slave Id* in the **Item** field:

   a. **Device**: "" (empty **String**)

   b. **Item**: "5:hr1500.d" or "5:hr1500.double", or if a hexadecimal is convenient, "5:hr&h5DC.d"

4. Reading or writing Holding Registers (functions **03** and **06**) to or from address 100 of a device with *Id* 5, interpreted as a **Word**, with *Slave Id* in the **Item** field:

   a. **Device**: "" (empty **String**)

   b. **Item**: "5:shr100" or "5:shr100.u16", or "5:shr100.word". Notice that, because it is a **Word**, Modbus protocol's native data type for Holding Registers, the data type can be omitted

5. Reading or writing Holding Registers (functions **03** and **06**) to or from address 100 of a device with *Id* 5, interpreted as a **user-defined data type** named "mytype", with *Slave Id* in the **Item** field:

   a. **Device**: "" (empty String)

b. **Item**: "5:shr100.mytype"

> **NOTE**: The address space of Holding Registers in Modbus protocol contains 16-bit registers. Therefore, to read 32-bit data types, such as **DWord** or **Float**, users must read two "hr" addresses for each Tag accessed. Likewise, reading a **Double**-type Tag demands the reading of four Holding Register addresses. For the same reasons, reading and writing "hr" Tags with a **Byte** data type can only be performed in pairs. On the device, each Holding Register address always contains two bytes.

Size of Data Types

BCD- and **String**-data types, as they have a variable size, demand the specification of a data type size, in bytes, right after their data type.

Notice that **only data types 2 and 4 are valid** (2 and 4 bytes or 4 and 8 digits) for **BCD** data types. Examples:

1. Reading or writing Holding Registers (functions **03** and **16**) to or from address 100 of a device with *Id* 1, interpreted as a **10-byte-String** (five "hr" registers), with *Slave Id* in the **Device** field:

    a. **Device**: "1:"

    b. **Item**: "hr100.s10"

2. Reading or writing Holding Registers (functions **03** and **16**) to or from address 100 of a device with *Id* 1, interpreted as an **eight-digit-BCD** (four bytes or two "hr" registers), with *Slave Id* in the **Item** field:

    a. **Device**: "" (empty **String**)

    b. **Item**: "1:hr100.bcd4"

Byte Order

As explained on the syntax of the previous section, users can add an optional byte order parameter in Tag's **Item** field to specify a byte ordering for devices that do not comply with protocol's standard. If a device complies with Modbus protocol's default ordering, this field can be omitted.

If distorted values are read, which can be observed on early tests with a device, and if these values, converted to hexadecimal, are correct after inverting the position of some bytes, please read this section carefully.

The Modbus protocol uses the big endian format by default, where values are formatted with their most significant coming first in communication frames. This is the default format used by this Driver. There is, however, a large amount of devices in the market that use values with other combinations for byte ordering.

As an example, if a Driver reads a value equal to "1234h" (or "4660" in decimal), by default this Driver waits that data be sent with a byte sequence equal to 12h and 34h. If the device uses an inverted default, which is called little indian, then the byte 34h is sent first and then the byte 12h, and the Driver may interpret it as 3412h, or 13330 in decimal, unless these two bytes were inverted before interpreting.

For 32-bit values, there are cases when **Word** values are swapped, but with bytes inside **Word** values keeping their default ordering. For example, the value 12345678h can be received as 56781234h. There are also other situations, with several different combinations for ordering.

To allow communication with these devices that do not follow protocol's standard byte order, this Driver allows users to configure Tags by specifying the order to use.

The *byte order* parameter corresponds to the **swap options** from the old **numerical configuration**, and it may have values "b0", "b1", "b2", "b3", "b4", "b5", "b6", "b7", "alias", or else "alias2" (please check the next table).

If the byte order parameter is omitted, a data is interpreted as the protocol's default, which is equivalent to the "b0" code.

The next table indicates that swap operations (*Swap Bytes*, *Swap Words*, and *Swap DWords*) are performed for each ordering mnemonic (from "b0" to "b7").

Swapping operations

	SWAP BYTES	SWAP WORDS	SWAP DWORDS	ALIAS	ALIAS 2 (SWAPS)	BYTE ORDER*
b0				msb	-	by7 by6 by5 by4 by3 by2 by1 by0
b1	X			-	sb	by6 by7 by4 by5 by2 by3 by0 by1
b2		X		-	sw	by5 by4 by7 by6 by1 by0 by3 by2
b3	X	X		-	sb.sw	by4 by5 by6 by7 by0 by1 by2 by3
b4			X	-	sdw	by3 by2 by1 by0 by7 by6 by5 by4
b5	X		X	-	sb.sdw	by2 by3 by0 by1 by6 by7 by4 by5
b6		X	X	-	sw.sdw	by1 by0 by3 by2 by5 by4 by7 by6
b7	X	X	X	lsb	sb.sw.sdw	by0 by1 by2 by3 by4 by5 by6 by7

* 64-bit (where "by0" is "lsb" and "b7" is "msb")

That is, "b0" does not perform any swap operation on data bytes, keeping the original ordering of bytes received from the device, which is equivalent to deselecting the **swap options on Operations tab** from the old **numerical configuration**.

"b1", on the other hand, performs a byte swapping, two by two, that is, when receiving a **Word** (unsigned 16-bit integer) with the hexadecimal value 0102h, the value returned to the Tag is 0201h, with its bytes swapped. It is equivalent to the old **Swap Bytes** option.

"b2" performs a **Word** swapping, that is, bytes two by two, which obviously affects only 32-bit data or larger. This is the same as selecting the **Swap Words** option from the old numerical configuration. As an example, if the value 01020304h is received from a device, the value used for application Tags is 03040102h.

"b3" performs byte and **Word** swapping, which is equivalent to the old **Swap Bytes** and **Swap Words** options enabled simultaneously. In this case, the value 01020304h becomes 04030201h.

Likewise, "b4" performs a **DWord** swapping for 64-bit values, which corresponds to the **Swap DWords** option from the old **numerical configuration**, that is, the value 1122334455667788h is interpreted as 5566778811223344h. And so on for all other codes.

The last two table columns specify aliases that users can use for readability, that is, instead of using a "b0" code, users can use an "sw.sdw" alias, and so on.

How to Select the Correct Byte Order?

In most cases, device's documentation specifies the byte order used, or how to configure it (there are devices that allow that configuration).

In cases where device's documentation does not contain a configuration, users must contact manufacturer's technical support.

If there is no reliable information, users must perform empirical tests, analyzing the returned values, in hexadecimal, comparing them to the expected values and observing if there are byte order inversions that may explain the differences.

There are basically three situations:

1. For devices providing data using Modbus protocol's default byte order (big endian or Motorola), with the most significant bytes coming first, users must omit this parameter or define it as "b0". This is the most common situation.

2. For devices using another byte order standard, with the least significant bytes coming first (little endian), users must enable all swap options referring to the data type used, which corresponds to the "b7" mnemonic.

3. In the least common case, there are devices that use different byte orders for different data sizes, providing for instance the most significant byte of each **Word** first, but the least significant **Word** of each **DWord** first. Therefore, users must evaluate in which case each swap option must be enabled, thus converting the value returned by the device to protocol's default big endian format.

> **NOTE**: All mentioned swap options have no effect for **Bit** data types or eight-bit-size types (**Byte**, **Char**, and **Int8**). Swapping occurs inside each data type, that is, the **Swap Words** option has no effect for 16-bit data types, as well as the **Swap DWords** option has no effect for 32-bit data types. **BCD** data types do not allow swapping operations either.

The topic **Frequently Asked Questions** lists a few known cases, already observed on technical supports. Examples:

1. Reading or writing Holding Registers (functions **03** and **16**) to or from address 1500 of a device with *Id* 5, interpreted as a **Double** without byte inversion, with *Slave Id* in the **Item** field:

    a. **Device**: "" (empty **String**)

    b. **Item**: "5:hr1500.d" or "5:hr1500.double", or else "5:hr1500.d.b0"

2. Reading or writing Holding Registers (functions **03** and **16**) to or from address 1500 of a device with *Id* 5, interpreted as a **Double** with the least significant byte of each **Word** coming first, and with *Slave Id* in the **Item** field:

a. **Device**: "" (empty **String**)

b. **Item**: "5:hr1500.d.b1" or "5:hr1500.double.b1", or else "5:hr1500.double.sb"

3. Reading or writing Holding Registers (functions **03** and **16**) to or from address 1500 of a device with *Id* 5, interpreted as a **Double** with the least significant byte coming first (little endian), and with *Slave Id* in the **Item** field:

a. **Device**: "" (empty **String**)

b. **Item**: "5:hr1500.d.b7" or other variations, such as "5:hr1500.d.lsb" and "5:hr1500.d.sb.sw.sdw"

Driver's Special Tags

In addition to all Tags described previously, users can configure Driver's **Special Tags** using **Strings**, which are described in details on their specific topics (click an item for more information).

Special Tags

DEVICE	ITEM	OPERATION
	ForceWaitSilence	Writing
<slave id>:	LastExceptionCode	Reading or writing

Numerical Configuration

Numerical configuration is performed using *N* and *B* parameters of **I/O Tags**, not using the **Device** and **Item** fields available in Elipse E3 or Power, which must be left empty.

This configuration method must be used with Elipse SCADA and with legacy applications. In applications using newer products, such as Elipse E3, Elipse Power, or Elipse OPC Server, it is recommended to use **String configuration**.

I/O Tags configured numerically reference **operations** previously configured on configuration window.

Operations

As already explained on topic **Operations tab**, this Driver supports other data types in addition to protocol's native data types. For this reason, the concept of **Operation** was created on this Driver.

On operations using Modbus functions that read and write bits, such as protocol functions **1**, **5**, and **15**, this Driver always map binary values of each bit to Block Elements, where each Element represents the value of a specific bit.

Operations with eight-bit data types, such as the **Byte** type, always imply, obviously, on reading at least two bytes (a 16-bit Modbus register). To prevent surprises, this Driver requires that eight-bit data writings be performed in pairs, that is, writing Blocks with an even number of Elements. These operations must be referenced using I/O Tag's *N2/B2* parameters, as described later.

Configuration Parameters of I/O Tags

The following configuration applies to I/O Tag's *N* parameters, as well as to I/O Block Tag's *B* parameters.

- **N1/B1**: Address of a slave device (PLC) on the network (*Slave Id*). This address is used on serial networks and can vary from 1 to 247. This parameter can be also configured with value 0 (zero). Thus,

this Tag works in **Broadcast** mode, sending a message to all slave devices (PLC) on the network. In **Ethernet** (**Modbus TCP** mode), the address generally used is the IP address, but the *Slave Id* can still be used when the IP address references a gateway connected to a device network (usually an RS485 network, with Modbus RTU, using a gateway capable of performing a conversion from **Modbus TCP** to **Modbus RTU**).

> **NOTE**: In **Broadcast** mode with *N1* equal to 0 (zero), users cannot perform readings, only writings. In this mode all devices on the network are addressed, receiving the written value and not returning any response, to avoid network conflicts.

- **N2/B2**: Operation code. References an operation added on Driver's configuration window (please check topic **Operations Tab**).

- **N3/B3**: Additional parameter. This parameter is not generally used and can be kept in 0 (zero). It is only used in four situations:

  - **Modbus functions 20 and 21**: For operations that use these functions for file access (functions **20 and 21**), the *N3/B3* parameters specify the file to access.

  - **Use Bit Mask**: For Tags referencing operations with the **Use Bit Mask** option enabled, the *N3/B3* parameter specifies the number of the bit to access (please check topic **Operations Tab**).

  - **User-Defined Data Types**: For operations that use structures, if the *B3* parameter is greater than 0 (zero), it defines the return of an event-reported block array, by using a sequence of Tag's **OnRead** events (please check topic **User-Defined Data Types**).

  - **Gen SOE Special Function**: For operations that use the **Gen SOE** special reading function, the *N3/B3* parameter indicates the size of the linked table in the PLC or slave device memory, as the maximum number of supported events (please check topic **Elipse Software's Generic SOE Reading Algorithm**).

- **N4/B4**: Register, variable, or bit address on the slave device (PLC) to read or write, according to device's register map (please check device's documentation). It is important to correctly configure the **Data Address Model Offset** option (please check topic **Modbus Tab**) and check if manufacturer's documentation does not use offsets used by old Modbus devices, known as **Modbus Convention**.

- **Size/Index (Block Tags only)**: Each Block Element represents a data value of a type defined in the operation used (the *N2/B2* parameter). Notice that this protocol only supports **Bit** or **Word** types. Thus, if this operation selects the **DWord** type (32-bit) for each Block Element, this Driver must read two consecutive registers from a device.

Special Tags

In addition to I/O Tags (Tags referencing operations), there are also special Tags to execute specific Driver functions. These Tags are described on topic **Configuring Special Tags**.

Addressing Tips (Modbus Convention)

On topic **Configuring an I/O Tag**, Tag addressing (*N4/B4* parameters on **numerical configuration**) is described based on the most recent Modbus protocol specification (version 1.1b). However, there are devices that still use the old offset addressing convention known as Modbus Convention, which adds offsets to an address. This topic explains how to address Tags if device's register mapping still follows that old

---

convention, originated from initial Modicon specification, not included on the current specification.

The address provided in the Tag is sent together with protocol's request frame, with or without the default offset of 1 (one), required by the Modbus Data Model specified by the protocol, according to the configuration in the **Data Model Offset** field, on **Modbus** tab of Driver's configuration window.

In addition to this default offset of 1 (one), defined on the current Modbus standard (version 1.1b), some manufacturers still use the old Modicon standard, known as **Modbus Convention**, with an offset that can be added to the address, and whose value depends on the Modbus function used, or more specifically, depends on which address space this function accessed originally. Such additional offset must be ignored when defining Tag addresses on this Driver. Later on this Manual there are more examples. The next table lists all offsets used by the **Modbus Convention** standard.

Modbus Convention standard offsets

DATA TYPE (STANDARD DATA MODEL)	MODBUS FUNCTION	OFFSET
Coils	**01**: Read Coils (0x) **05**: Write Single Coil (0x) **15**: Write Multiple Coils (0x)	000000
Discrete Inputs	**02**: Read Discrete Inputs (1x)	10000
Input Registers	**04**: Read Input Registers (3x)	30000
Holding Registers	**03**: Read Holding Registers (4x) **06**: Write Single Register (4x) **16**: Write Multiple Registers (4x)	40000
File Register (old Extended Memory file)	**20**: Read General Reference (6x) **21**: Write General Reference (6x)	60000

If device's register map uses this convention, users must follow this procedure to determine the addresses to attribute to Tags, in the **Item** field when **configuring by Strings** or in N4 or B4 parameters for **numerical configuration**:

1. On **Modbus tab**, select the **Data is addressed from 1** option.

2. Subtract from the address displayed on device's manual the offset shown on the previous table for the Modbus function used. **TIP**: Remove the fifth digit from right to the left.

Notice that, in devices that use this old convention, users can determine which Modbus functions can be used to access each register or bit using the offset used in its address.

Examples

ADDRESS WITH OFFSET (DEVICE)	ADDRESS ON I/O TAG (ITEM OR N4/B4)	MODBUS FUNCTION
01234	1234	**01**: Read Coils **05**: Write Single Coil **15**: Write Multiple Coils
11234	1234	**02**: Read Discrete Inputs
31234	1234	**04**: Read Input Registers
41234	1234	**03**: Read Holding Registers **06**: Write Single Register **16**: Write Multiple Registers

ADDRESS WITH OFFSET (DEVICE)	ADDRESS ON I/O TAG (ITEM OR N4/B4)	MODBUS FUNCTION
45789	5789	**03**: Read Holding Registers **06**: Write Single Register **16**: Write Multiple Registers
65789	5789	**20**: Read General Reference **21**: Write General Reference

Automatic Block Partition

Starting with version 2.00, Modbus Driver now has a feature called **Automatic Block Partition**. With this feature, this Driver manages the division of blocks larger than **protocol limits**. Thus, users do not need to worry about exceeding the maximum limit for block size, because this Driver divides blocks in the correct sizes during communication with a device, if any Block Tag exceeds the maximum allowed size.

Starting with version 2.01, this Driver also supports **Superblock Readings**. With this feature enabled, users do not need to group variables into Block Tags aiming to improve performance, it is possible to use only Tags without degrading performance. And as the Superblock algorithm already considers the maximum block size allowed by the protocol, when this feature is used this Driver also does not need to use Automatic Partition feature.

In cases when, due to device's specificity (please check topic **Superblock Reading**), it is not possible to enable the **EnableReadGrouping** property in E3 or Elipse Power (a property that enables Superblocks), or if users are using the old Elipse SCADA, which does not support grouping (Superblocks), then they must use Automatic Block Partition to ignore protocol limits.

IMPORTANT: Superblock grouping in E3 and Elipse Power, as well as Driver's Automatic Block Partition, require that a device supports all limits established by standard Modbus (please check topic **Maximum Limit for the Size of Blocks Supported by the Protocol**). There are devices, however, that support lower limits. For this automatic block partition to work in these cases, as well as Superblock grouping, starting with version 2.03 this Driver allows customizing the maximum supported limit for PDU (*Protocol Data Unit*). To do so, on Driver's configuration window, on **Modbus** tab, enable the **Customize Max. PDU Size** option and configure the maximum size of bytes supported by this device for PDU. If this device has different limits for each function, it may be necessary to perform that grouping manually (please check topic **Superblock Reading**).

The article *KB-23112* in Elipse Knowledgebase presents a summary of questions related to Tag grouping and Block resizing in Modbus Driver, discussed here and in other topics (please check topics **Superblock Reading** and **Optimization Tips**).

Superblock Reading (Grouping)

Starting with version 2.01, this Driver supports a **Superblock Reading** feature. This feature is supported by E3 and Elipse Power, and it can be enabled via Driver's **EnableReadGrouping** property in Organizer. When this property is configured in True, users do not need to worry about block resizing.

With this feature, it is possible (and usually recommended) to create applications with only simple Tags (PLC Tags in Elipse SCADA) without performance issues, because group optimization on readings is automatically performed during communication. The next figure displays the configuration of the **EnableReadGrouping** property in E3 or Elipse Power.
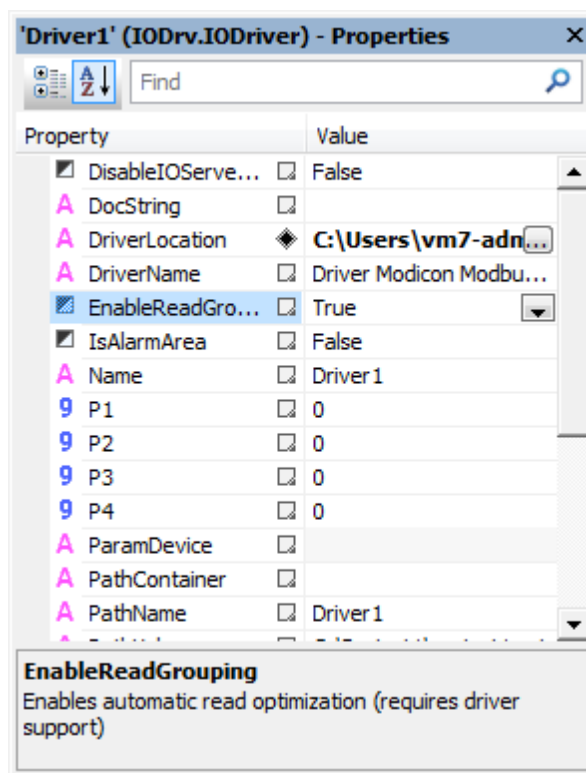
EnableReadGrouping property

Elipse SCADA does not support Superblocks. The behavior when reading Tags in Elipse SCADA is identical to E3 and Elipse Power when the **EnableReadGrouping** property is configured as False. In both cases, this Driver relies on **Automatic Block Partition**, and it can divide blocks with sizes larger than **protocol limits** into smaller blocks during communication. In these cases, users must consider that grouping when defining application Tags, as seen later on this topic.

NOTES:

- Automatic grouping is performed based on application Tags in advise. Whenever new Tags enter or leave advise, the Superblock algorithm redefines this grouping, that is, Superblocks to be read automatically, at run time, including only Tags in advise.
- Only Tags using numerical configuration can be grouped by the Superblock service. Tags using configuration by Strings are not grouped by the Superblock service.

IMPORTANT: Superblock grouping in E3, as well as Driver's **Automatic Block Partition**, require that a device supports the limits established by standard Modbus (please check topic **Maximum Limit for the Size of Blocks Supported by the Protocol**). There are devices, however, that support lower limits. For automatic block partition and Superblock grouping to work on these cases, starting with version 2.03, this Driver allows customizing the maximum limit supported for PDU (*Protocol Data Unit*). To do so, on Driver's configuration window, **Modbus** tab, enable the **Customize Max. PDU Size** option and configure the maximum size of bytes supported for PDU on this device. If this device supports different limits for each type of function, users must perform a manual grouping (please check further on this topic), observing all limits described on manufacturer's documentation.

Identifying Devices that do not Support Automatic Grouping (Superblocks)

Superblock algorithm considers all limits and addressing spaces defined by standard Modbus protocol. For devices that implement Modbus protocol with small variations, some additional advanced configurations may be necessary to use this Superblock feature, if its usage appears viable. In these cases, it is necessary to disable automatic grouping (the **EnableReadGrouping** property configured as False), and then perform a

manual grouping. The following conditions may prevent using Superblocks, or may require additional advanced configurations:

- Devices that define maximum limits for block sizes lower than protocol's standard limit (**a limit of 253 bytes for PDU**). **Solution**: Configure the **Customize Max. PDU Size** option, on **Modbus** tab.

NOTE: There are devices whose PDU limits vary according to the Modbus function used. In these cases, if it is necessary to use functions with different limits, it is also necessary to disable Superblocks (the **EnableReadGrouping** property configured as False), by manually grouping Tags (please check later on this topic).

- Devices with discontinuities (undefined address intervals inserted between valid intervals) on the register map. **Solution**: Once it is impossible to inform to the Superblock algorithm which intervals cannot be inserted in blocks, usually it is not possible to use Superblocks. Disable Superblocks (the **EnableReadGrouping** property configured as False) and manually group all Tags.

- Devices that do not support block readings. **Solution**: Disable Superblocks (the **EnableReadGrouping** property configured as False) and define simple Tags.

- Devices that only allow defining blocks in pre-determined addresses and with fixed sizes. **Solution**: Disable Superblocks (the **EnableReadGrouping** property configured as False) and define simple Tags (PLC Tags in Elipse SCADA) or Blocks according to device's specification.

Manual Grouping

Usually, the larger the grouping of variables in blocks, the less reading requests are needed to complete a scan cycle of application Tags, thus increasing Tag's update speed. For this reason, if it is not possible use automatic grouping (Superblocks), it is preferable to create Block Tags containing as many variables as possible, instead of creating simple Tags (PLC Tags in Elipse SCADA).

Notice that, due to **Automatic Block Partition** feature, there is no need to prevent exceeding protocol's maximum limits, as long as a device supports **protocol's default maximum limits**. If this device does not support these limits, but defines fixed limits, valid for all supported Modbus functions, users must configure the **Customize Max. PDU Size** option, on **Modbus** tab.

If a device supports different limits for each supported function, automatic partitioning can be also unfeasible. In these cases, an application developer must also consider device's limits, and define blocks respecting these limits.

For a manual grouping, using **User-Defined Types** can increase possibilities of grouping, by allowing to gather on a single Block Tag variables from the same addressing space, that is, a single Modbus function, but with different data types (the defined structure may have elements with different data types).

For more tips, please check topic **Optimization Tips**. The article *KB-23112* in Elipse Knowledgebase presents a summary of questions related to Tag grouping and block resizing in Modbus Driver, discussed here and on other topics.

Configuring Special Tags

In addition to **I/O Tags**, this Driver also supports a few Special Tags that allow an application to trigger features not related to data reading or writing. Unlike I/O Tags, Special Tags do not reference **Driver operations** in **numerical configuration**. The next topics describe in details all supported Special Tags:

- **Forcing a Wait Silence**

- **Reading the Last Exception Code**

Forcing a Wait Silence

Special Tag used to discard all pending data from communication until it finds a time-out, indicating that there is no more data to receive.

This service can be configured on **Modbus** tab, to occur whenever detecting a communication error. With this Tag, however, users can execute this service at any time by an application.

This Special Tag is executed using a Tag's writing command. Its value, written to the Tag, is ignored.

String Configuration

- **Device**: Not used, this field must be left blank

- **Item**: "ForceWaitSilence"

Numerical Configuration

- **N1**: Not used, can be left in zero

- **N2**: 9001

- **N3**: Not used, can be left in zero

- **N4**: Not used, can be left in zero

- **Value**: Not used, can be left in zero

Reading the Last Exception Code

As already mentioned on this Manual, Special Tags for reading the last exception code are used to read the last exception code sent from a certain slave device.

Such codes are automatically stored by this Driver in internal registers, which can then be accessed using this Tag. In addition, at each successful communication with a certain device where no exception was returned, this Driver automatically zeroes the associated register.

Exception Codes

Exception codes are used by a slave device (PLC) to report a failure when executing a certain function. Slave devices do not return exceptions in case of communication failures, a situation where these devices simply do not respond. Exception codes are returned by slaves in situations where a master request (in case of a Driver) was successfully received, but could not be executed for any reason, such as trying to read or write to a non-existent register. In this case, the returned exception code indicates the type of error occurred, that is, the reason that a Driver's request, although correctly received, could not be completed.

The specification of Modbus protocol defines nine exception codes. The list of protocol's default exceptions can be checked on topic **List of Protocol's Default Exceptions**. In addition to these codes, some manufacturers define additional codes, specific to their devices. Such codes must be documented on device's

manual. If they are not, please check with manufacturer's technical support.

String Configuration

- **Device**: Numerical value of device's Id (*Slave Id*) followed by a colon. Example: "1:", "2:", "3:", etc.

- **Item**: "LastExceptionCode"

Numerical Configuration

- **B1**: Slave device's address (*Slave Id*)

- **B2**: 9999

- **B3**: Not used, can be left in zero

- **B4**: Not used, can be left in zero

Values returned on Block Elements:

- **Element 1 (index 0)**: Exception code returned by a device (please check topic **List of Protocol's Default Exceptions**)

- **Element 2 (index 1)**: *N2/B2* parameter of the I/O Tag generating this exception

- **Element 3 (index 2)**: *N3/B3* parameter of the I/O Tag generating this exception

- **Element 4 (index 3)**: *N4/B4* parameter of the I/O Tag generating this exception

- **Element 5 (index 4)**: *Size* parameter of the I/O Tag generating this exception

- **Element 6 (index 5)**: *Device* parameter of the I/O Tag generating this exception

- **Element 7 (index 6)**: *Item* parameter of the I/O Tag generating this exception

Using a Special Tag

The most common usage for this Tag during a normal scan of function Tags is via an exception Tag's **OnRead** event. In this case, a script must first reject null values, because these values indicate that exceptions were not received. Next, users can handle that exception by executing the adequate procedures, according to the received code. It is advisable to zero the exception register when leaving a script, to indicate that this exception was already handled. Please check the following example, written in Elipse Basic (Elipse SCADA):

```
// TagExc Tag's OnRead event
// Note: For this example, consider TagExc
// with automatic reading and writing enabled

If TagExc == 0
  Return
EndIf

If TagExc == 1
  ... // Handles exception 1
ElseIf TagExc == 2
  ... // Handles exception 2
Else
  ... // Handles all other exceptions
EndIf

TagExc = 0 // Zeroes the exception register
```

This is another example, written in VBScript (Elipse E3 and Elipse Power):

```
' TagExc Tag's OnRead event
' Note: For this example, consider TagExc
' with automatic reading and writing enabled

Sub TagExc_OnRead()
  If Value = 0 Then
    Exit Sub
  End If

  If Value = 1 Then
    ... ' Handles exception 1
  ElseIf Value = 2 Then
    ... ' Handles exception 2
  Else
    ... ' Handles all other exceptions
  End If

  Value = 0 ' Zeroes the exception register
End Sub
```

In writing operations by script, on the other hand, where users must test for returned exceptions right after sending a command, users must first zero the exception register. That avoids an eventual exception provoked by a writing command to be confused with another pre-existing one. Execute the writing operation and test a Special Tag's value, which must return 0 (zero) if no exception was received. In case it returns a value different from 0 (zero), then users can properly handle that received exception. Please check the following example, written in Elipse Basic (Elipse SCADA):

Modicon Modbus Master (ASC/RTU/TCP) Driver v3.1.30

```
// Note: For this example, consider TagExc
// with automatic reading and writing enabled
// and TagVal with automatic writing disabled

TagExc = 0 // Zeroes the exception register

TagVal.WriteEx(10) // Writes the value 10

If TagExc <> 0
  ... // Handles this exception
EndIf
```

This is another example, written in VBScript (Elipse E3 and Elipse Power):

```
' Note: For this example, consider TagExc
' with automatic reading and writing enabled
' and TagVal with automatic writing disabled

' Zeroes the exception register
Application.GetObject("Tags.TagExc").Value = 0

' Writes the value 10
Application.GetObject("Tags.TagVal").WriteEx(10)

If Application.GetObject("Tags.TagExc").Value <> 0 Then
  ... ' Handles the exception
End If
```

NOTE: This Special Tag returns, in addition to an exception code (returned on Element zero), also Tag parameters whose communication provoked that exception. If this information is not needed, users can read the same register using a simple Tag (a PLC Tag in Elipse SCADA), without using a Block Tag. In this case, all recommended procedures remain the same.

Mass Memory Reading

This Driver allows defining, on operations, special reading functions for collecting mass memory from slave devices. Such functions do not exist in the protocol, and imply in using specific algorithms for reading events from devices, which may read or write in several registers, by using one or more protocol functions.

Callback Readings

Starting with version 2.08, this Driver implements callback readings, a feature available in E3 (starting with version 3.0) and in Elipse Power, which optimizes performance of mass memory readings. With this feature, an application delegates to a Driver a Tag scan for reading mass memory events. In other words, an application does not need to keep asking a Driver at each scan period. Instead, a Driver performs a verification of new events on a device and collects events as soon as they become available, and sends them to an application.

Special Functions for Mass Memory Readings

On the current version of this Driver, the following functions for reading sequences of events (SOE) are supported:

- **GE SOE**: Performs event collecting from GE PAC RX7 PLCs. For more information, please check topic **Reading an Event Buffer from GE PAC RX7 Controllers**

- **SP SOE**: Collects events from Schneider Electric SEPAM series 20, 40, and 80 relays. For more information, please check topic **Reading Events from Schneider Electric Relays from SEPAM 20, 40, and 80 Series**

- **GenSOE**: This function uses a generic Algorithm created by Elipse, **Elipse Modbus SOE**, which can be used by most Programmable Controllers. It requires the creation of an analogous programming procedure on PLC's programming (*ladder*). For more information, please check topic **Elipse Software's Generic SOE Reading Algorithm**

- **65 03**: Special function for reading mass memory events from ABB MGE 144 meters. For more information, please check topic **Reading Mass Memory Registers from ABB MGE 144 Meters**

Reading an Event Buffer from GE PAC RX7 Controllers

An event buffer can be read using three types of Tags: **Event-reported Tags**, **Event-reported Tags by point**, and **Real-time Tags**.

Event-reported Tags

Event-reported Tags return, at each reading operation, all events stored in Driver's internal buffer, and they can be **configured by Strings** or **numerically**.

String Configuration

- **Device**: "<slave id>:"

- **Item**: "gesoe0.<Base address of an event stack>"

Numerical Configuration

To read an event buffer from GE PAC RX7 using numerical configuration, users must define, on Driver's configuration window, an operation that uses as its reading function the **special function GE SOE**. Its data type must be defined as **GE_events**.

- **B1**: Slave ID

- **B2**: Code of the operation defined with **GE SOE** function

- **B3**: 0 (zero)

- **B4**: Base address of PLC's event stack

At each scan on this Tag, this Driver checks if there are events on controller's buffer. If there are events, this Driver starts an event-reading thread, which is executed in background, not blocking the scan of all other Tags. After finishing the reading of a Driver's buffer, this event-reported Tag returns the set of events read on that scan.

Returned events generate a sequence of **OnRead** events on this Tag. For each read event, E3 updates Tag fields (Element values and timestamp) with values from a certain event, and calls the **OnRead** event once.

This event's script must be defined by users, and it is generally used to insert Tag's data in a Historic.

Every event is represented by a Block with two Elements, with its **Timestamp** field read from a device. Fields from the respective reading Block Tag are displayed on the next table.

Block Tag fields

OFFSET	MEANING	DATA TYPE	RANGE OF VALUES
0	Point identification	Byte	Between 0 and 15
1	Point status	Byte	Between 0 and 1

For more information about event-reported Tags, please check the specific topic on **E3 User's Manual**.

IMPORTANT: When reading mass memory events in event-reported Tags in E3, disable Tag's dead band (the **EnableDeadBand** property configured as False) and also the linked Historic object (the **DeadBand** property equal to zero), to avoid loosing events with near values. It is also important to disable historic by scan (in E3, the **ScanTime** property equal to zero). This ensures that new events are only stored using the **WriteRecord** method, executed on Tag's **OnRead** event, avoiding duplication of events.

Event-Reported Tags by Point

Starting with version 2.5 of this Driver, it is possible to use a new Tag to download events from a specific point.

This Tags works as the previous one, except that it returns only events from a specific point.

Different from the previous event, the returned value only contains a single Element with a status value of a point, so that only one Tag can be used. This Tag must be configured as follows:

String Configuration

- **Device**: "<Slave Id>:"

- **Item**: "gesoe<200 + Point's index>.<base address of event queue>"

Numerical Configuration

- **N1**: Slave ID

- **N2**: Code of the operation defined with **GE SOE** function

- **N3**: 200 + Point's index (for example, for point 2, configure *N3* as 202)

- **N4**: Base address of PLC's event stack

For more information about event-reported Tags, please check a specific topic on **E3 User's Manual**.

Real-time Tags

These Tags return the most recent event already read for a specific point. These events are stored on Driver's internal memory for each PLC's event reading, with their respective timestamps read from a device. This Tag

uses the following parameters:

String Configuration

- **Device**: "<Slave Id>:"

- **Item**: "gesoe<100 + Point's index>.<base address of event queue>"

Numerical Configuration

- **N1**: Slave ID

- **N2**: Operation Code

- **N3**: 100 + Point's index

- **N4**: Base address of PLC's event stack

- **Value**: Point's status

Reading Events from Schneider Electric Relays from SEPAM 20, 40, and 80 Series

To read SEPAM relays, the offset model of addressing must be configured as **Data is addressed from 0**, on **Modbus tab**. Reading these events is performed using two basic types of Tags:

- **Tag for collecting all table events (mandatory)**: Performs a collect of all table events from a certain memory zone on a device. This Tag, in addition to returning all events read to an event-reported application, still stores events read in a Driver's internal buffer, to remove them using readings on Tags for reading a single event, described next.

- **Tag for reading a single event (optional)**: Returns events received from a specified relay, with a certain address, type, and zone. This Tag does not perform a direct reading from a device, but returns events from Driver's internal buffer, stored during the reading of a Tag for collecting all table events, described previously, that is, to be able to read events with this type of Tag, a **Collecting all events**-type Tag must be already active, with its scan enabled. This Tag is useful when users need to get events from a specific type and source (relay, zone, address, and type). An example of usage is an association to Screen objects, displaying the status of a certain event address. Although this Tag returns the same information returned by the previous Tag, its usage prevents users from creating filters, **Select Case** clauses in VBScript, or any other method to separate several types of events returned by a **Tag for collecting all events by script** in an application.

An application must implement a Tag for collecting all events for each table or event zone to collect on each relay, because it is during the reading of this Tag that actual events are collected from a device. Configuration for these two Tags is presented next.

Tag for Collecting all Table Events (Event Zone)

This is an event-reported Tag. Its typical usage is inserting events in a linked Historic, by using Historic's **WriteRecord** method, called on Tag's **OnRead** event. At each reading, that is, at each Tag's scan period, this Driver can collect up to four new events from a device. This is the maximum number of events that each relay's event zone contains at each reading request.

As it is during the reading of this Tag that events are effectively collected from a device, event if its data is not used directly, that is, even if there is no need to store all events in a Historic, its implementation is mandatory for single-event Tags to return data. A Tag for collecting all events must be configured as a Block Tag with three Elements, and with the following parameters:

String Configuration

- **Device**: "<slave id>:"

- **Item**: "spsoe<Zone or Event Table (1 or 2)>"

Example: To read Zone 1 of Slave 1, **Device** is equal to "1:" and **Item** is equal to "spsoe1". Alternatively, **Device** can be equal to "" and **Item** equal to "1:spsoe1" (please check the topic **String Configuration**).

Numerical Configuration

To use numerical configuration, users must define, on **Operations tab**, an operation that uses as its reading function the **special function SP SOE**. Its data type is automatically defined as **SP_events**, as soon as the **SP SOE** reading function is selected.

- **B1**: 1000 + Slave address (relay) on the network (between 1 and 247)

- **B2**: **Code of the operation** configured with **special reading function SP SOE**

- **B3**: 0 (zero)

- **B4**: Zone or event table (1 or 2)

The next table describes the meaning of these three Block Elements, which have their values returned as reported by events.

Meaning of Block Elements (SP_events data type)

OFFSET	MEANING	DATA TYPE	RANGE OF VALUES
0	Type of event	**Word**	From 0 to 65535 (800H for **Remote Annunciation**, **Internal Data**, and **Logic Input**)
1	Address of the event	**Word**	References bit addresses from 1000H to 105FH
2	Ramp up or down	**Word**	• **00**: Ramp down • **01**: Ramp up

For more information about event-reported Tags, please check topic **Tags Reported by Events** on **E3 User's Manual**.

Tag for Reading a Single Event

This is also an event-reported Ta, and users can use its **OnRead** event for storage in a Historic. Notice that this does not prevent it to be treated as a normal Tag (a real-time Tag), in case only its most recent value is relevant. As this Driver only reads an internal buffer, it is advisable to define a very low scan time, even lower than the one from the other type of Tag. CPU consumption at each scan can be considered as not significant. It is suggested to configure it as half the scan period for a Tag for collecting all events.

As already mentioned, this Tag is used to get a status for a certain event address, without parsing or performing filters on events that arrive by the previous Tag, by script, or by any other means. A typical application would be linking it to Screen objects.

A Tag for reading a single event, as already mentioned, does not perform a reading of events from a device, but from a Driver's internal buffer, previously filled during the reading of a Tag for collecting all events. This Tag returns a single Element, an event-reported one, and it can be configured as a simple Tag (it does not need to be a Block Tag). This Driver accepts up to eight events accumulated by event point, that is, for each combination of relay, zone, type, and event address, in its internal buffer. If there is an overflow, that is, if more than eight events from a single point are returned without any single-event Tag collecting them, this Driver starts to discard older events. The correct configuration of a scan time may prevent data loss.

This Tag must be configured with the following parameters:

String Configuration

- **Device**: "<slave id>:"

- **Item**: "ptspsoe<Event type (800H by default)>.<Event bit address) + Event zone offset* (please check the next table)>"

Example: To read **800H**-type events at address 1 of zone 2, **Device** must be equal to "1:" and **Item** must be equal to "ptspsoe&h800.&h8001". Alternatively, **Device** can be equal to "" and **Item** equal to "1:ptspsoe&h800.&h8001" (please check topic **String Configuration**).

Numerical Configuration

- N1: Slave address (relay) on the network (between 1 and 247)

- N2: **Code of the operation** configured with **special reading function SP SOE**

- N3: Event type (0800H by default, according to manufacturer's documentation)

- N4: Event address (Event bit address) + Event zone offset, as described on the next table

Options for event address (Events Zone Offset)

EVENT ZONE	EVENT ZONE OFFSET
1	0
2	8000H (8000 in hexadecimal)

Examples:

- **Event Address**: 102FH, Event Zone = 1 ® N4 = 102FH + 0 = 102FH

- **Event Address**: 102FH, Event Zone = 2 ® N4 = 102FH + 8000H = 902FH

NOTE: To represent values in hexadecimal in Elipse E3 and in Elipse Power, users must use prefix "&H" (for example, &H10 = 16). In Elipse SCADA, use suffix "h" (for example, 10h = 16). On this Manual, however, the uppercase suffix "H" is used to indicate values in hexadecimal format.

- **Value**: Returns ramp up or down, as described on the next table

Available options for Value

VALUE	MEANING
00	Ramp down
01	Ramp up

- **Timestamp**: The **Timestamp** property represents the date and time an event was actually read from a relay, during the reading of a Tag for collecting all events previously described

For more information about relay events, their meanings and addressing, please check manufacturer's documentation. For more information about event-reported Tags, please check topic **Tags Reported by Events** on **E3 User's Manual**.

Elipse Software's Generic SOE Reading Algorithm

The Modbus protocol does not define a default method for reading events from a device. For this reason, it is common that manufacturers create their own algorithms for reading events from devices that support Modbus protocol.

The generic algorithm for Sequencing of Events (SOE) of this Driver (**Elipse Modbus SOE**) was developed by Elipse Software to provide a default alternative for reading events from programmable controllers that do not have a native version of this feature, provided that these controllers meet some basic requirements of memory space and programming features, and also allow the creation of tables and control registers described later.

This algorithm allows storing and reading events from almost all programmable controllers, in an optimized way, by using features already implemented and validated in Modbus Driver.

Event reading on Modbus Driver follows a standard procedure, defined by Elipse Software, reading events from **tables created in PLC's memory** or in slave devices by its resident application (*ladder*).

To use this algorithm, users must define Tags using the **special function Gen SOE**, which can be performed using either the new **String configuration** (**Device** and **Item** fields) or using the old **numerical configuration**

(*N* and *B* parameters). Tag configuration is described later on topic **Acquisition Procedure in an Application**.

During the process of reading events, the **special function GenSOE** always uses the Modbus function **03** (*Read Holding Register*) to read registers from a device. For writing during the update of control registers, the default function used is Modbus function **16** (*Write Multiple Registers*). By using the numerical configuration, users can select the writing function **06** (*Write Single Registers*), in the rare case of devices that do not support function **16** (the opposite is more common), by using operation's **Write** field, on **Operations tab**.

PLC's resident software (*ladder* or equivalent) must keep updated all control registers that provide information to this Driver, such as the number of events available for reading and the address of the last register to read.

A device can keep more than one **event table**, in different memory addresses, containing different data types. Each table must be preceded by their respective **control registers**, in adjacent addresses. This table is formed by a **circular buffer** in contiguous addresses, accumulating events or data for collecting by this Driver at each collect procedure (download of events).

Users can define distinct data formats (events) for each defined table, which are usually defined as a **data structure**, and may contain event's timestamp field. Events can also be defined using a Driver's **native data type**. In this case, users cannot define a **Timestamp** field in the PLC (the timestamp is sent with the date of reading), and the event contains a single field, which can be represented by a simple Tag (a PLC Tag in Elipse SCADA).

> **NOTE**: SOE algorithm always uses Modbus protocol's function **03** (*Read Holding Registers*) to read registers from a device. For writing registers, the default Modbus function used is **16** (*Write Multiple Registers*). Users can also select function **06** (*Write Single Register*) only on **numerical configuration**, by using the **Write** field of the respective operation, on **Operations tab**.

The next topics describe in details this algorithm, its implementation in a PLC software (*ladder*), and how to perform its reading by using Driver's Tags:

- **Event Table**

- **Acquisition Procedure in a PLC**

- **Acquisition Procedure in an Application**

Event Table

As already mentioned on topic **Elipse Software's Generic SOE Reading Algorithm**, each event table keeps events in a circular buffer. The circular buffer of each table is defined by its initial address, or base address, contiguous to control registers, and by its maximum number of registers, which defines its final limit. The following table shows a layout of events inside a table's circular buffer.

Layout of events in a circular buffer

EVENT	TIMESTAMP	ELEMENT1	ELEMENT2	ELEMENT3	...	ELEMENTN
1						
2						
3						
4						
5						

EVENT	TIMESTAMP	ELEMENT1	ELEMENT2	ELEMENT3	…	ELEMENTN
…						
N						

Each row on the previous table represents a stored event, usually represented by a structure, or by **User-Defined Data Types**.

Notice that, in the example of the previous table, the first element of that event structure is a timestamp. This field, whose presence is not mandatory and that does not need to appear in the first position, defines Tag's **Timestamp** property and it is not returned in its Elements (for more information, please check topic **User-Defined Data Types**).

It is also possible to define events with **Driver's native data types**, and in this case there is only one data Element in each event, without timestamp.

Events must be inserted in this circular buffer in ascending order, returning to the base address after reaching the upper limit of this circular buffer. The following control registers must be defined for each table:

- **Table status**: Must be kept by the PLC, indicating the number of events available for reading in the circular buffer. It must be updated by the device whenever new events are added to the circular buffer, or after finishing the collecting of events by an application, which can be detected via **Acquisition status**.

- **Recording pointer**: This value indicates an index, starting at 0 (zero), of the position where a device must insert the next event. It must be incremented by the device after each new event insertion in the circular buffer, moving back to the base address after reaching buffer's upper limit. Notice that this value must not be provided in units of Modbus registers, but in event positions, and it must be incremented by one unit at every new event inserted, regardless of the number of Modbus registers occupied for each event in the circular buffer. With this, the maximum allowed value for this pointer is given by the formula **MaxWritePtr = (Size of the circular buffer / Size of event structure) - 1**.

- **Acquisition status**: Indicates the number of registers already read by a Driver at each individual event reading. After each reading, this Driver writes in this register the number of registers it successfully read. Slave's resident application (*ladder*) must immediately subtract the value written by this Driver from **Table status** and then zeroes **Acquisition status**.

- **Reserved**: This register is currently not used. It may be used on future versions of this Driver, and it can be kept in 0 (zero) in its current version.

As already mentioned, the base address of this circular buffer, that is, the address in which an event table starts, must be contiguous to control registers.

Control registers, on the other hand, must be also set on contiguous addresses, on the same order presented previously, allowing their reading on a single operation, that is, assuming that the base address of control registers for a certain table is 100, these are the addresses for the other registers:

Register addresses

REGISTER	ADDRESS
Table Status	100
Recording Pointer	101
Acquisition Status	102
Reserved	103

REGISTER	ADDRESS
Circular Buffer's Base Address	104

On topic **Acquisition Procedure in a PLC** there is a description of a step-by-step acquisition procedure or algorithm based on a slave device (PLC). On the next topic, **Acquisition Procedure in an Application**, there is a discussion on how to configure an application for acquiring table events.

Acquisition Procedure in a PLC

This topic presents a discussion about an algorithm for collecting events from the point of view of a PLC or slave device. Its goal is to clarify to developers what must be implemented in PLC's resident application (*ladder*).

A device must start inserting events in ascending order, starting at table's base address, that is, starting at the circular buffer. For each new event inserted, the Recording Pointer must be incremented, pointing to the next available position in that buffer.

This Driver performs a reading starting from the oldest to the newest event. The starting address of this reading is calculated by this Driver using the value of Recording Pointer and Table Status.

If the number of available events is greater than protocol's **maximum allowed into a single communication frame**, this Driver performs multiple block readings, updating the value of Acquisition Status after finishing this process with the total amount of events read.

NOTE: If a device does not respect the default limit of 253 bytes for PDU, then users must configure the **Customize Max. PDU Size** option, on **Modbus** tab, according to supported limits, which must be described on manufacturer's documentation.

When detecting a non-null value written by this Driver to Acquisition Status, a PLC's or device's application must immediately subtract Acquisition Status' value from Table Status' value and then zeroes Acquisition Status. With Acquisition Status zeroed again, this Driver can start a new acquisition at any time.

A PLC can insert new events to a table during PLC's acquisition process, as long as there is no overflow on circular buffer, that is, as long as its writing pointer does not exceed its reading pointer, by incrementing Table Status.

An event collecting or downloading procedure is finished when Table Status is zeroed. All collected events are then provided to an application via event-reported Tags, as described on the next topic.

The next figure shows a flow chart, as a UML Activity Diagram, with a suggestion of implementation for a PLC logic. Notice that some variations are possible, for example discarding the oldest event in case of overflow, which can be evaluated by a developer, depending on the context.

UML Activity Diagram (PLC)
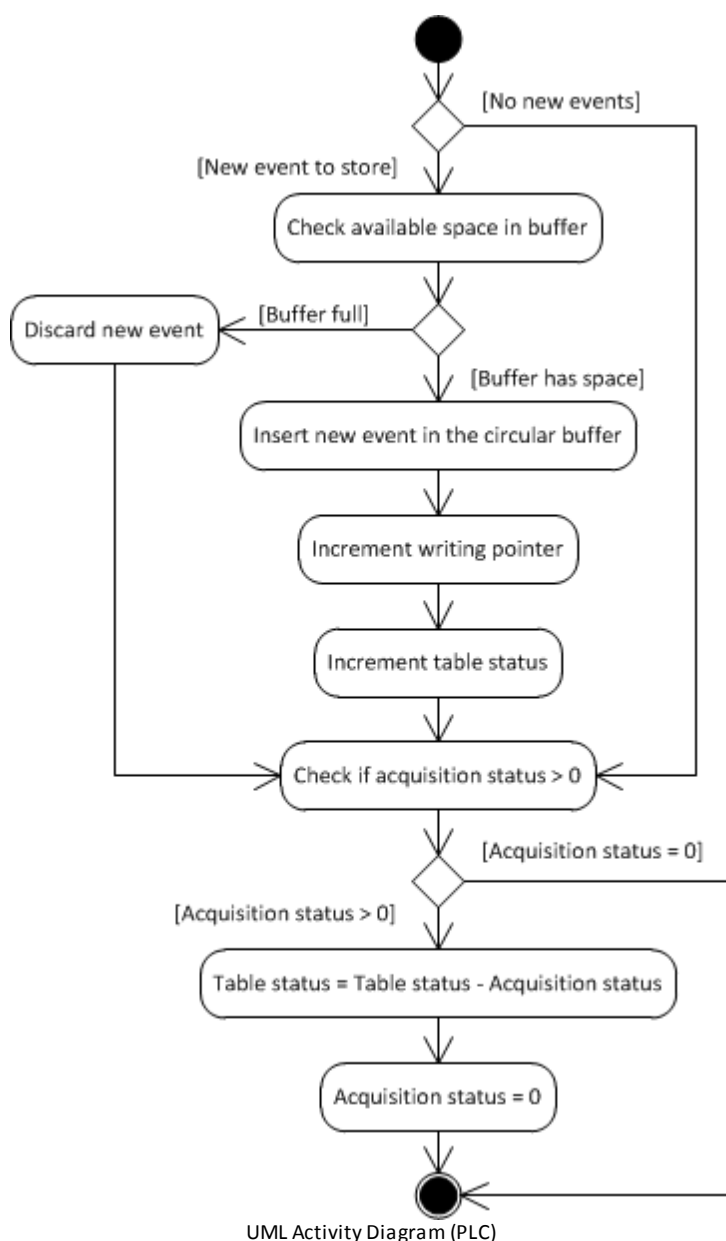
Timestamp

As already mentioned, every event is composed by a structure containing one or more data elements (usually, but not necessarily, represented by **User-Defined Data Types**).

If structures (user-defined data types) are used, then users can associate a timestamp to each PLC-provided event. In this case, the value of the **Timestamp** field must be provided in a structure field, in PLC memory, in the order it was declared in the configuration file, and its value is not displayed in any Block Element, it is only returned in the **Timestamp** property of the linked Tag.

As explained on topic **User-Defined Data Types**, any date and time type supported by this Driver can be used. The **GenTime** data type, however, was specially created for use with Elipse Modbus SOE, due to an easy definition in PLC's resident application (*ladder*).

If a millisecond precision is needed, another option is consider Driver's **UTC32** data type, represented as an integer with only 32 bits (4 bytes) with seconds starting at 1/1/1970, without a milliseconds representation, considered as 0 (zero).

The next topic, **Acquisition Procedure in an Application**, describes how to configure an application for collecting events accumulated in a PLC or programmable slave device.

Acquisition Procedure in an Application

This topic contains a detailed explanation on the configuration of an application to acquire events accumulated in a PLC or programmable slave device.

Reading events in an application is performed using Tags that reference the special reading function **Gen SOE**. Tag's data type defines the structure of events stored in the device's event table. If a **Driver's native data type** is defined (a built-in type), each event contains only one element of this type, without a device-provided timestamp (a timestamp represents the instant events were collected). On the other hand, if **user-defined data types** are used, then users can define structures for events, including timestamps, as explained later on this topic.

Next, there is a description of the configuration of Tags using the new methodology of configuring by **Strings** (**Device** and **Item** fields), as well as the old numerical configuration used by Elipse SCADA (*N* and *B* parameters).

String Configuration

- **Device**: "<Slave Id>:"

- **Item**: "elsoe<N>.<start add.>[.<type>[<type size>]][.<byte order>][/bit]"

Where:

- **N**: "<Slave Id>:"

- **start add.**: Address of the first control register, using the value defined on the example table of topic **Event Table**

- **type**: Native or user-defined data type used for each event (please check topic **String Configuration**)

- **type size**: Used only for variable-size data types (please check topic **String Configuration**)

- **byte order**: Byte ordering. It can be omitted for devices fully compliant with protocol's standard (please check the **Byte Order** item on topic **String Configuration** for more information). When structures are used, it only affects their individual elements (please check topic **User-Defined Data Types**)

- **bit**: Bit masking. Usually can be omitted, it would hardly be used here (please check the **Bit** field on topic **String Configuration**)

Example:

- **Device**: "1:"

- **Item**: "elsoe150.&h101.TYPE3"

The **TYPE3** data type is defined as follows in Driver's default example file (please check topic **User-Defined Data Types**).

```
// This type has an UTC32-type timestamp
// and a few named elements
struct TYPE3
{
  DefaultAddress = 0x101;
  timestamp = UTC32;
  float Va;
  float Vb;
  float Vc;
  float Ia;
  float Ib;
  float Ic;
}
```

This is a **Structure** data type containing six data fields and a timestamp. Therefore, this Tag must be a Block with six Elements to represent that structure.

Notice that, as already explained, timestamp's value, although it occupies registers in the PLC, does not need Block Tag Elements, because its value is returned in Tag's timestamp field.

> **NOTE**: The *N* parameter informs the size of a table as the maximum number of events, not as Modbus records. Together with the *Start Address* parameter, it indirectly informs the final address or upper limit of that table. The size of table's data area, therefore, in number of Modbus registers, is the product of *N* by the size of each event in number of Modbus registers, that is, in 16-bit **Words**.

Numerical Configuration (N and B Parameters)

To configure Tags for reading Elipse SOE using a numerical configuration, users must configure an operation on **Operations tab**, using the **special function GenSOE**.

The next figure shows an example of a new operation added using special function **GenSOE** with a **Word** data type.
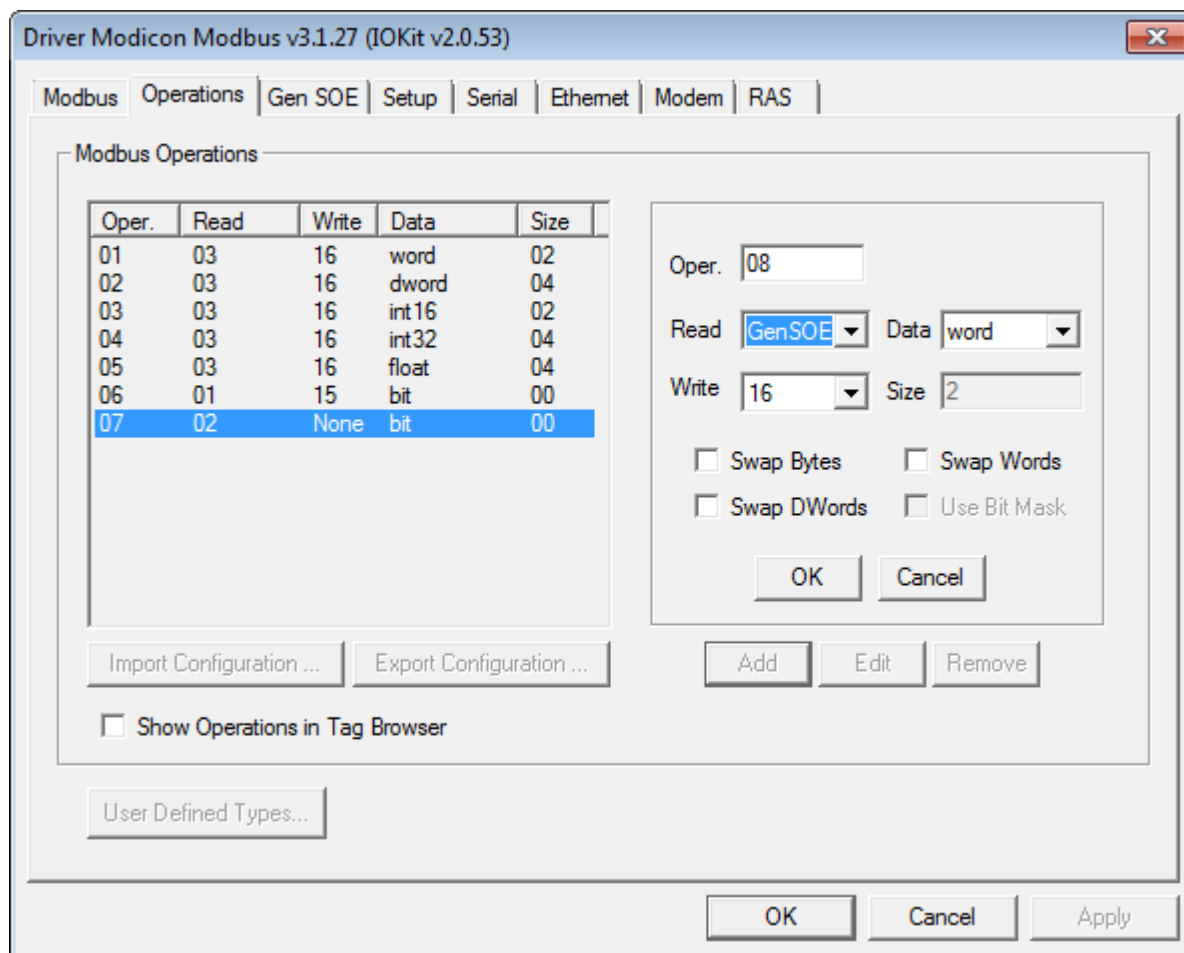
Special function GenSOE

Notice that function **16** (*Write Multiple Registers*) was selected as its writing function, which is the most commonly used function. However, users are encouraged to use function **06** (*Write Single Register*) whenever supported by a device.

The next figure shows the same operation with a user-defined data type **TYPE3** (please check topic **User-Defined Data Types**), which is a data type defined in the example default configuration file, available with this Driver when it is firstly added to the application, used as an example in this topic.
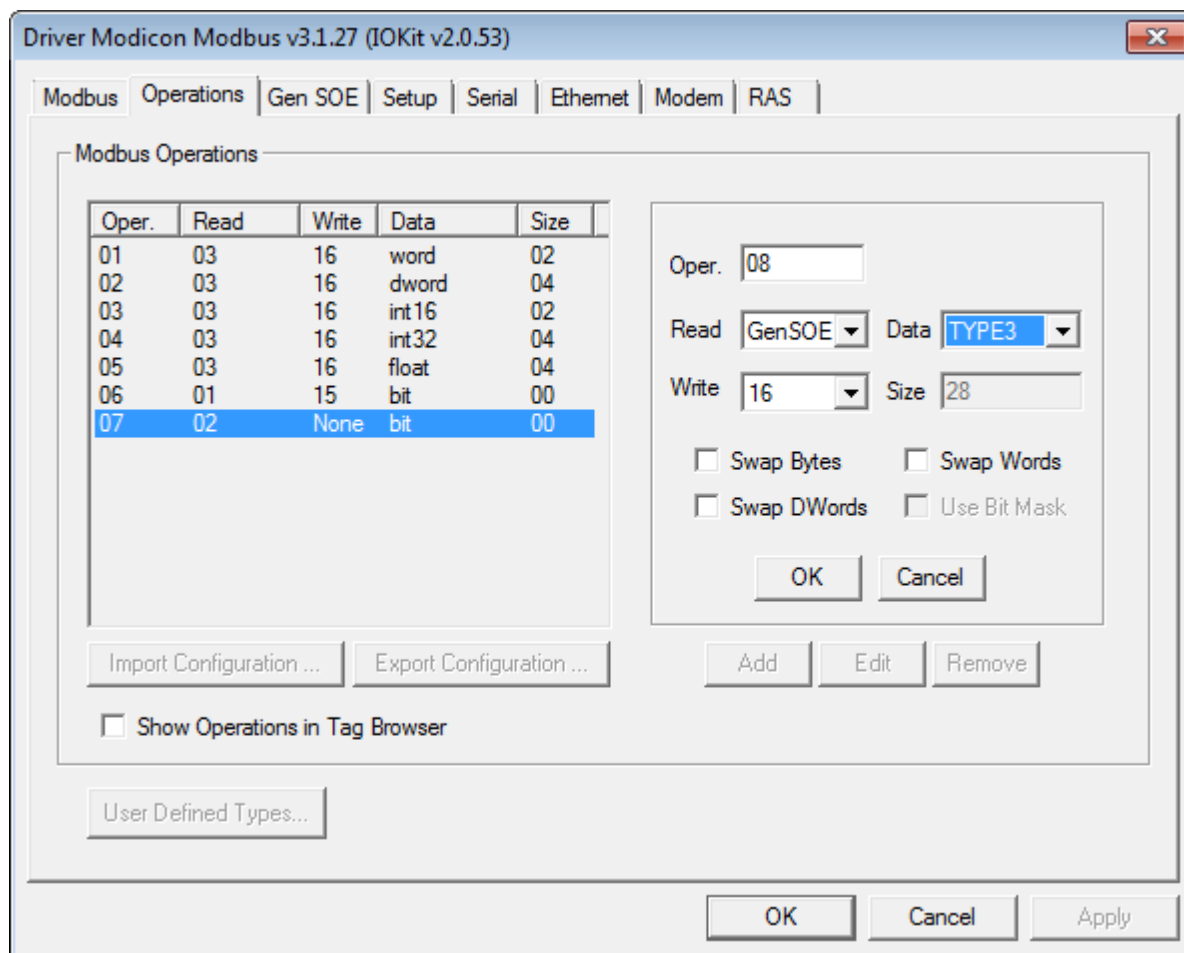
Configuration using the GenSOE function and a user-defined data type

Data type **TYPE3** is defined in Driver's example file as follows:

```
// This type has an UTC32-type timestamp
// and a few named elements
struct TYPE3
{
  DefaultAddress = 0x101;
  timestamp = UTC32;
  float Va;
  float Vb;
  float Vc;
  float Ia;
  float Ib;
  float Ic;
}
```

This is, therefore, a **Structure** data type with six data fields, a timestamp, and a default address (Tag's *B4* parameter) equal to "101H" (257 in decimal). To read it, users must define a Communication Block Tag with six Elements and the following configuration:

- **B1**: Slave device's address (PLC) on the network (*Slave Id*)

- **B2**: 8 (a previously defined operation with special function **Gen SOE**)

- **B3**: N (size of a device's table, as the maximum number of events that table can contain)

- **B4**: 100 (address of the first control register, using a value defined on the example table of topic **Event Table**)

- **Size**: 6

> **NOTE**: The *B3* parameter informs the size of a table as the maximum number of events, not as Modbus records. Together with the *B4* parameter, it indirectly informs the final address or upper limit of that table. The size of table's data area, therefore, in number of Modbus records, is the product of *B3* by the size of each event in number of Modbus records, that is, in 16-bit **Words**.

Notice that, if E3's Tag Browser is used to insert a Tag in an application, as explained on topic **User-Defined Data Types**, Tag Elements are already named according to the name given to structure elements when they were declared. Tag Browser can be opened by clicking 🖼 on Driver's **Design** tab.

Usage

Once defined an appropriate Tag (or Tags), enable its scan and let the Driver collect events from their respective table, whenever new events are detected.

Tags linked to the **GenSOE** function (**elsoe** when configuring by **Strings**) are always **event-reported**. This means, as already explained on topic **User-Defined Data Types**, that this Driver can return several events on a single reading operation, that is, on a single interval of a Tag's scan.

This means that this Driver returns a set of events (for the previous example, sets of blocks with six data fields and a timestamp) at once, which produces a sequence of **OnRead** events on a Tag, one for each event (a block with six data fields and a timestamp) returned by this Driver.

For detailed instructions on the right way to handle event-reported Tags, please check topic **Tags Reported by Events** on **E3 User's Manual**. **Elipse SCADA** User's Manual also contains an analogous topic.

In short, the usual way of handling event-reported Tags is by adding a call to the **WriteRecord** method of a previously linked Historic object on Tag's **OnRead** event, ensuring that all events reaching this Historic are registered. In this case, this Historic must be configured without a dead band (the **DeadBand** property set to zero) and disabling historic by scan (in E3, the **ScanTime** property set to zero). Tag's **EnableDeadBand** property must also be configured to False.

> **IMPORTANT**: When reading mass memory events in E3's event-reported Tags, disable Tag's dead band (the **EnableDeadBand** property configured as False) and also in the linked Historic object (the **DeadBand** property configured as zero), to avoid losing events with close values. It is also important to disable historic by scan (in E3, the **ScanTime** property configured as zero). This ensures that new events are only stored using the **WriteRecord** method, executed on Tag's **OnRead** event, avoiding duplicated events.

Optimization and Compatibility

Some devices, such as PLCs by ATOS, do not support block readings using data types with different structures. In practice, this prevents a Driver to read data from control and event registers as a single block. To collect PLC events with these restrictions, users must disable the **Enable Control and Data Registers Grouping** option on **Gen SOE tab**.

Reading Mass Memory Registers from ABB MGE 144 Meters

To read mass memory registers from ABB MGE 144 meters, users must configure Tags using the special reading function **65 03**, as described on this topic.

The special function **65 03** is ABB's proprietary and it is practically identical to protocol's standard function **03**

(*Read Holding Registers*), differing only on returned data, referring to ABB meter's mass memory.

Data is returned as a **Word** (as in function **03**), with protocol's default byte order (big endian). Therefore, there is no need to enable any swap function (*Swap Bytes*, *Swap Words*, or *Swap DWords*).

Meter's register map, specifying data to read as well as its correct configuration, must be checked on meter's manufacturer-provided documentation.

This Driver also contains two special writing functions specific for this meter, functions **65 01** and **65 02**. For more information about those special writing functions, please check topic **Special Functions** and also device's documentation.

String Configuration

- **Device**: "<slave id>:"

- **Item**: "abbmge<address>[.<type>[<type size>]][.<byte order>][/bit]"

Where:

- **Address**: Address of the Modbus register to read

- **Type**: Data type. If omitted, assumes default **Word**. For more information, please check topic **String Configuration**

- **Type size**: Used only for variable-sized data types. For more information, please check topic **String Configuration**

- **Byte order**: Byte ordering. If omitted, assumes protocol's default. For more information, please check topic **String Configuration**

- **Bit**: Bit masking. Usually omitted (prefer application's bit masks). For more information, please check topic **String Configuration**

Numerical Configuration (N or B Parameters)

- **N1/B1**: Slave Id

- **N2/B2**: Code of the **operation** configured with function **65 03** (check next)

- **N3/B3**: Not used, leave in 0 (zero)

- **N4/B4**: Register's address

To configure Tags numerically, users must first add an operation with function **65 03** on **Operations tab** of Driver's configuration window, as shown on the next figure.
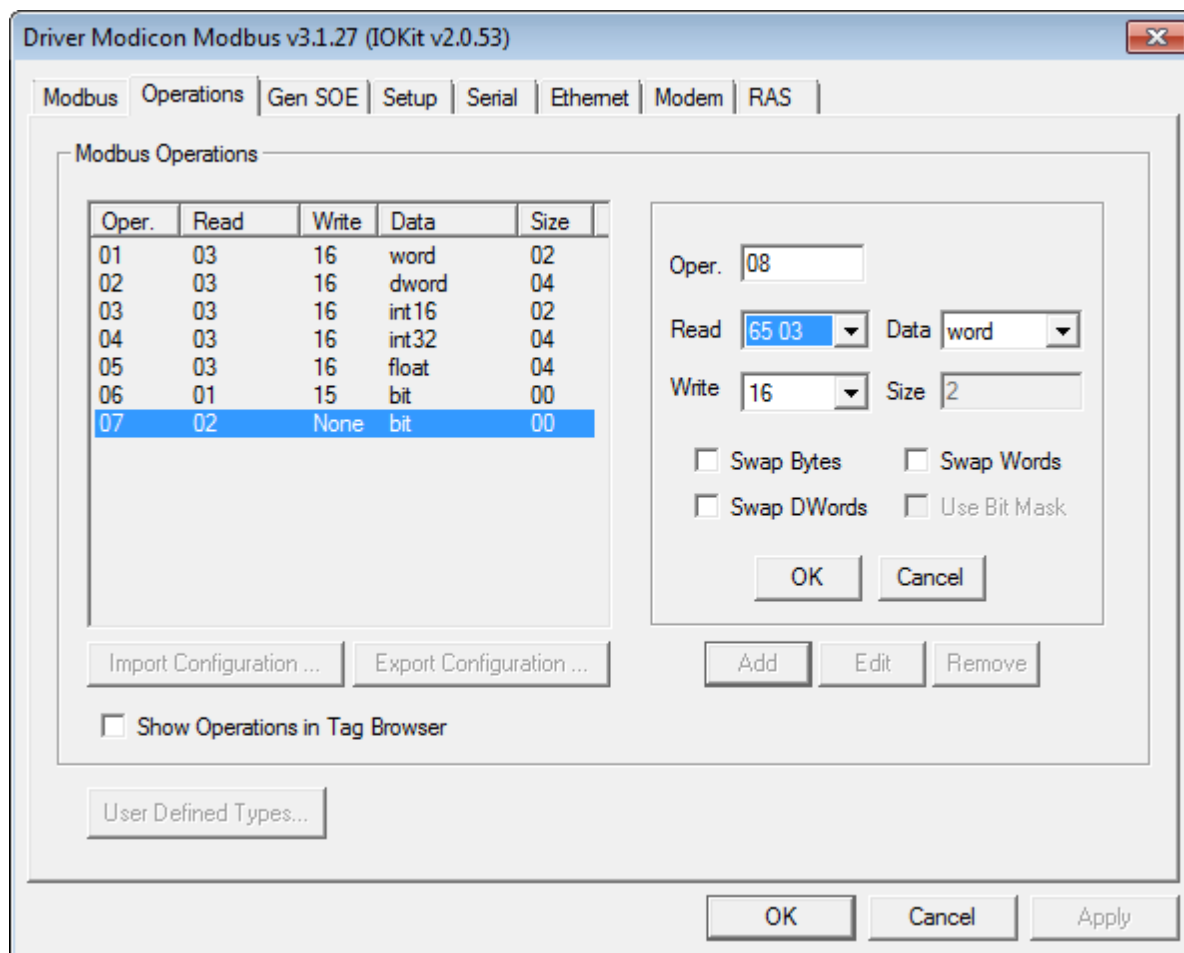
Creating an operation with special function 65 03

Appendix

This appendix contains the following topics:

- **Optimization Tips**

- **Frequently Asked Questions**

- **List of Devices that Communicate with Modbus**

- **List of Protocol's Default Exceptions**

- **Maximum Limit for the Size of Blocks Supported by the Protocol**

- **BCD Encoding**

Optimization Tips

This topic enumerates some optimization tips to communicate with slave devices.

Tips for Configuring a Driver for E3 and Elipse Power

- Use Superblocks whenever possible, preferring to create simple Tags (PLC Tags in Elipse SCADA) instead of Block Tags (please check topic **Superblock Reading**).

- If Superblocks are not an option, prefer to create Block Tags, grouping the largest possible number of variables in the smallest number of blocks (please check the text about manual grouping on topic

Superblock Reading).

- Consider all recommendations of article *Performance tips for E3*, in Elipse Knowledgebase.

- In case of high-latency networks, reduced bandwidth, or packet loss, please also read article *E3's Network Settings for networks with high latency, low bandwidth and/or packet loss*.

- In high-latency networks, configure higher time-outs, considering the expected latency. Remember that a time-out is only effective on delays, it does not interfere with performance on normal usage situations.

Tips for Configuring a Driver for Elipse SCADA

- Prefer to create Block Tags, by grouping the largest possible number of variables in the smallest number of blocks (please check the text about manual grouping on topic **Superblock Reading**).

- Consider all recommendations of article *Developing applications with optimum performances*, also available in Elipse Knowledgebase.

- In high-latency networks, configure higher time-outs, considering the expected latency. Remember that a time-out is only effective on delays, it does not interfere with performance on normal usage situations.

Tips for Configuring or Programming a Device

- If possible, group all variables defined by a resident application (*ladder*) that have the smallest scan time, in contiguous addresses in PLC's memory. The total scan time of Tags highly depends on the capacity of grouping variables in communication blocks.

Frequently Asked Questions

Known peculiarities of Twido devices by Schneider

- The application is trying to read a Float-type value, but it is failing. The value displayed by this PLC is completely different from the one displayed by the application for the same address.

  - **Answer**: This device does not use protocol's default byte order (*big endian*). Users must configure a byte order by executing a swap of Words, which corresponds to the **b2** option on the **configuration by Strings.**, or select the **Swap Words** option on the configuration of operations on **numerical configuration** (please check topic **Operations Tab**).

- The application is trying to read PLC's inputs and outputs, but it is failing.

  - **Answer**: This device does not allow reading or writing input and output variables, and users must use PLC's internal variables to perform this reading, that is, create a mirror of inputs and outputs in an area where this Driver has access. Users must also be careful to create a PLC routine to check when an application changes an output value, so that it is effectively activated or deactivated in this PLC.

Known peculiarities of MPC 6006 devices by Atos - Schneider

- The application is trying to read a **DWord**-type value, but the correct value is not displayed. The application displays values different from the ones in this PLC.

- **Answer**: Please check the article *Using Modbus Master (ASC/RTU/TCP) drivers with ATOS's controllers* in Elipse Knowledgebase. If using the new **String** configuration (**Device** and **Item** fields), also check section **Byte Order** on topic **String Configuration**. If using the old numerical configuration (*N/B* parameters), also check topic **Operations Tab**, specially the **Byte Order** section.

How to join two Int16-type values (which are in this PLC) into an Int32-type value (in an application)?

- There is a 32-bit number stored in two 16-bit registers in this PLC. How to display this number on an application's screen as a single 32-bit register?

  - **Answer**: Users must create Tags using 32-bit data types, such as **Float**, **DWord**, or **Int32** data types. When configuring an I/O Tag, users must inform the first address of each PLC variable (please check topic **Configuring an I/O Tag**). This Driver then joins two 16-bit registers from the device into a single 32-bit value, which is returned on Tag's **Value** field or on Block Tag's Element. If using a configuration by **Strings** (**Device** and **Item** fields), inform a data type right after register's address (please check topic **String Configuration**). If using the old **numerical configuration** (*N/B* parameters), users must define operations with 32-bit data types. Notice that, on the configuration window (**Operations Tab**), 32-bit data types are always displayed with a four-byte size (the **Size** field, please check topic **Supported Data Types**).

- The application is already developed, but how to join values from two **Words** into a single Tag?

  - **Answer**: It is possible to execute that join operation using scripts, by creating an unsigned 32-bit integer. To do so, users must multiply the **Word** that contains the highest part of that word by 65536 and then sum the **Word** that contains the lowest part of that word. For example, **UInt32 = (HighWord × 65536) + LowWord**.

- The application wants to read **Float**-type values. A reading function **03** and a writing function **16** with a **Float** data type were configured. However, the application displays a value that is different from the one on this device.

  - Answer: The official Modbus protocol uses a big endian byte order, with the most significant bytes of every value coming first. If this Driver is reading nonsense values, even with its address correctly configured, probably the device uses a non-standard byte order. In this case, users must configure its swap options. If users are using a configuration by **Strings** (**Device** and **Item** fields) please check the **Byte Order** section on topic **String Configuration**. If users are using the old **numerical configuration** (*N/B* parameters), please check the **Byte Order** section on topic **Operations Tab** for more information on using these swap options.

How to communicate with more than one device on a Serial communication network?

- There is more than one device on a serial network, each one with a unique address. How to communicate with every one of them?

  - **Answer**: Users must be careful with the *Slave Id* of each **I/O Tag**, because this field indicates the device to communicate. On an RS485 serial network, all devices listen simultaneously to all driver requests (there is a single bus), although only the one with the corresponding *Slave Id* responds to the request (multiple devices with the same *Id* are not allowed). When configuring by **Strings**, this value can be provided in the **Device** field, or at the beginning of the **Item** field (please check for more information on topic **String Configuration**). If users are using the **numerical configuration**, this value is provided in the *N1/B1* parameter of each Tag. Users can use the same **operations** for Tags from several devices. A good reference for information and tips regarding the installation and maintenance of serial networks is the book *Serial Port Complete*, by Jan Axelson.

- There is more than one serial port on the computer. How to configure this Driver to communicate with devices connected to each one of these ports?

  - **Answer**: In this case, as there is more than one different physical layer (Serial 1, Serial 2, etc.), it is necessary as many I/O Drivers as the existing ports. Configurations for this Driver's **Tags** can be the same for all Driver objects (instances). The only difference is that one Driver must be configured to communicate via Serial 1 port, another Driver configured to communicate via Serial 2 port, and so on. Port configuration is performed on the Serial tab of Driver's configuration window (please check topic **Properties**).

How to communicate with more than one device on a serial network with an RS485 converter?

- There is an RS485 network with several devices communicating via an RS232-RS485 converter using a Serial port. Whenever an address switches (*Slave ID*), that is, when this Driver requests data from another device, a time-out occurs. After retrying the same message, this device answers normally. Is there a way of preventing this time-out during an address switching (*Slave ID*)?

  - Answer: Some RS232-RS485 converters require a time interval to switch, that is, commuting from transmission mode to reception mode, or vice versa. To circumvent this limitation, users can use the **Inter-frame delay** option on IOKit's **Serial** tab, available on the **configuration window**. This field defines a time interval between messages. The exact value of this interval depends on the converter in use but, if it is unknown, users are recommended to try values between 50 ms and 300 ms.

NOTE: IOKit's **Inter-frame delay** option may significantly degrade performance in some applications, and it must be used only when absolutely necessary. Please be sure that this converter is in good conditions, and if it effectively requires a delay. If needed, please check with manufacturer's technical support.

How to communicate with more than one device on an Ethernet network?

- There is more than one device connected to an Ethernet network, each one with a unique IP address. How to communicate with each one of them?

  - Answer: Currently, for each IP address, users need as many I/O Drivers as the number of IP addresses to communicate. The configuration referring to Driver **Tags** can be the same for all Drivers. The only difference is that one Driver must be configured to communicate with IP address 1, another Driver must be configured to communicate with IP address 2, and so on. The *Slave Id* parameter can be still used in **Modbus TCP** mode to differentiate devices connected to a Modbus Ethernet / RS485 gateway on the same IP address. Notice that this gateway not only must allow an interconnection among Ethernet and serial networks, but also convert ModbusTCP frames for the serial modes supported by devices (**ModbusRTU** or **ModbusASC**). The IP address must be configured on IOKit's **Ethernet** tab, on Driver's **configuration window**.

TIP: Avoid using protocol's **RTU** or **ASC** mode encapsulated in **TCP/IP** layer. If users want to encapsulate serial communication of devices using **Modbus RTU** in **TCP/IP**, there are gateways available in the market that not only encapsulate serial communication in **Ethernet TCP/IP** (physical, network, and transport layers), but also convert **Modbus RTU** into **Modbus TCP** (application layer). As a last option, if it is inevitable to use **Modbus RTU** in **Ethernet TCP/IP** layer, enable the **Reconnect after Timeout** option, previously described on topic **Modbus Tab**.

Modbus Simulator Software

- Is there any software that simulates Modbus protocol and can be used for testing with this Driver?

  - Answer: Yes, there are several alternatives. Elipse Software provides a free version (demo) of *Elipse Modbus Simulator* on its website, which allows simulating the most basic protocol features. There is also the possibility of using Elipse Software's Modbus Slave Driver as an emulator. Another possibility is Modsim, one of the oldest and well known alternatives to emulate a slave Modbus device. This simulator can be purchased at *http://www.win-tech.com/html/modsim32.htm*. In addition to it, there is also a free and open alternative named **Free Modbus PLC Simulator**, available at *www.plcsimulator.org*. There are still many other options and a list with other software can be found at *Modbus.org* website.

How to configure the N4/B4 parameter of I/O Tags?

- Which address to use in the *N4/B4* parameter of an I/O Tag?

  - **Answer**: This address varies from device to device. To know which is the exact address to use, please check device's documentation or contact its manufacturer's technical support. The topic **Addressing Tips** provides tips on common additional offset conventions used by many manufacturers.

When to use RTS and DTR controls (which appear on Serial tab of Driver's configuration window)?

- The application is communicating with a device directly connected to computer's RS232 serial port. How to configure **RTS** and **DTR** controls?

  - **Answer**: Please check device's documentation or manufacturer's technical support for the correct configuration.

- The application is communicating with a device using an RS232-RS485 converter connected to computer's RS232 serial port. How to configure **RTS** and **DTR** controls?

  - **Answer**: When communicating with devices using RS232-RS485 converters, such configurations depend on the converter. The device (*Slave*) does not influence it, as these signals only exist on the RS232 serial side, with no equivalent on the RS485 serial layer. The **RTS** control is commonly used by older converters to switch between transmission and reception modes (RS485 is half-duplex), and in these cases it must be configured in **Toggle** mode (there are some rare devices that require other configurations). On most recent converters, however, switching between transmission and reception is automatic, and these signals in general are not used, and they may be ignored. For more information, please check converter's documentation or manufacturer's technical support.

When to use Swap Bytes, Swap Words, and Swap DWords options?

These options must be used for 16-, 32-, or 64-bit data types, whose byte order of this device-provided value does not correspond to Modbus default byte order, where the most significant bytes always come first (*big endian*, also called *Motorola*). If this Driver is reading nonsense values, or values different from the ones stored on the PLC, it may use a byte order different from protocol's default. For more information, please check section **Byte Order** on topic **String Configuration** or, if using the old **numerical configuration** (*N/B* parameters), please check section **Byte Order** on topic **Operations Tab**. Users are also advised to check device's documentation.

- The application is trying to read a **Word** value, but this value appears different from the one configured in the PLC. If in the PLC it is configured as "1" (one), the application displays it as "256".

  - **Answer**: Value 1 (one) in hexadecimal is 0001H and value 256 in hexadecimal corresponds to 0100H. This device has a non-standard byte order. Users must enable the **Swap Bytes** option (the "b1" option on the **String Configuration**) to read the correct value.

- The application contains a Tag configured to read a **DWord** value, but the value read by the application is different from the value stored in the PLC. When setting the value "258", for example, to a PLC register, the application displays a nonsense value of "16908288".

  - **Answer**: The value 258 in hexadecimal is 00000102H and the value 16908288 in hexadecimal corresponds to 01020000H. This device has a byte order different from protocol's default, where the least significant **Word** comes first. In this case, users must enable the **Swap Words** option (the "b2" option on the **String Configuration**) to read the correct value.

How to correctly read Float data types from WEG TPW-03 PLCs?

- **Answer**: When configuring **I/O Tags**, users must enable the **Swap Words** option, which corresponds to the "b2" option on **String configuration**. If users are using the old numerical configuration (*N/B* parameters), please check section **Byte Order** on topic **Operations Tab**.

Known peculiarities of devices from the ABB Advant Controller 31 series 90 family (for example, ABB 07KT97 PLC)

- The E3 or Elipse Power application is trying to read registers or bits from a PLC, but there are always errors.

  - **Answer**: Devices from this series do not allow using E3 (or Elipse Power) Superblocks for two reasons:

    - There are interruptions on the address map of device's registers, with undefined address intervals.

    - Maximum **PDU** size is different from the one established by protocol's default, and it is defined as a size that supports 96 **Words** or **Bits**. As this protocol groups eight bits at each data byte, that results in different maximum **PDU** sizes for reading functions for **Bits** and **Words**, which prevents customizing the maximum **PDU** size allowed by this Driver, which does not allow configuring different limits for each protocol function.

- **Solution**: Follow these steps:

  - Disable Superblock reading, by configuring Driver's **EnableReadGrouping** property to False.

  - Prefer defining Block Tags, by grouping the largest possible number of variables in the smallest number of blocks, respecting device's limit of 96 **Words** or 96 **Bits** for each Block (for more information, please read the section about **Manual Grouping** on topic **Superblock Reading**).

NOTE: Users can also use automatic grouping (Superblocks) if there is no need to read **Words** and **Bits** on the same Driver object, obviously depending on the interval of addresses to read (more specifically, whether this interval contains interruptions or not). In this case, anyway, users must configure the **Customize Max. PDU Size** property on **Modbus Tab**, according to the limit of 96 **Words** (96 × 2 = 192 bytes) or 96 **Bits** (96 ÷ 8 = 12 bytes). Such possibility can be carefully evaluated, in a case-by-case basis, by the application's developer.

The application is trying to read Float data type values, and the following message appears on Driver's log: "Warning: denormalized float number! Returning zero". What to do?

- **Answer**: This message does not mean a communication or configuration error. Users are advised to check PLC's programming why it is returning non-normalized values.

- **Additional Information**: Such message indicates that the device sent a floating point value (**Float**) to this Driver in **IEEE 754** format, but non-normalized. Such values may be a result of arithmetical operations with results that extrapolate all representation possibilities of this format, such as overflow, underflow, +∞, and −∞, etc. Non-normalized values are described in IEEE 754 standard, and they are not supposed to raise problems for this Driver or for an application. However, due to previous error detections related to specific hardware, this Driver now returns 0 (zero) to an application when receiving non-normalized values from a device, registering this message on log.

List of Devices that Communicate with Modbus

The next table contains a list of devices, separated by manufacturer, for which there is already some experience on communicating with Modbus protocol.

For a more complete list of devices already validated for this protocol, please check *Modbus Device Directory*, maintained by the Modbus Organization.

Devices that communicate with Modbus

MANUFACTURER	DEVICE
ABB	• ETE30 • MGE 144 • KT97 • KT98
Altus	• Almost all Altus devices have Modbus, except a few models from Piccolo series
Areva	• MiCOM P127 • P632 relay
Atos	ATOS PLCs support Modbus RTU protocol with small variations regarding the maximum size of frames and byte order. For more information about these variations, please check the article *Using Modbus Master (ASC/RTU/TCP) drivers with ATOS's controllers*, in Elipse Knowledgebase
BCM	• BCM1088 • BCM1086 • BCM-GP3000 • BCM2085

MANUFACTURER	DEVICE
Ciber Brasil	• Multivariable Meter for Electrical Quantities UDP200
	• Multivariable Meter for Electrical Quantities UDP600
Contemp	• CPM45
Deep Sea	• DSE5210
	• DSE5310
	• DSE5310M
	• DSE5320
	• DSE5510
	• DSE5510M
	• DSE5520
	• DSE7310
	• DSE7320
Embrasul	• MD4040 meter
Eurotherm	• 2500 Intelligent Data Acquisition and Precision Multi-Loop PID Control
Fatek	• FB - 14MCU
GE	• GE PAC RX7
	• GE GEDE UPS
Gefran	• Gefran Gilogk II PLC
	• Gefran 600RDR21
Hitachi	• Hitachi HDL17264
Honeywell	• HC-900 PLC
Horner APG	• Devices from series XLe/XLt all-in-one control devices
Koyo	• CPU 260 PLC
Kron	• MKM-120 meter
LG	• DMT40U
LG / LSIS	• LG Master K120 S
	• LG XGB - XBM
Moeller	• XC100 - Serial Port 232
	• XC200 - Serial Port 232/422/485 (XIO-SER communication module) and Ethernet Port
	• XV200 - Serial Port 232 and Ethernet Port
	• XVH300 - Serial Port 232 and Ethernet Port
	• XV400 - Serial Port 232 and Ethernet Port
Novus	• N1100
	• N1500
	• N2000
	• N3000

MANUFACTURER	DEVICE
Schneider	• Twido • A340 • M340 PLC • Premium Series • Frequency Converters and soft starter • MT and BT breakers • BT and MT protection relays • SEPAM Series 20, 40, and 80 relays
Telemecanique	• Zelio Logic controllers ending with "BD"
Unitronics	• V120
Weg	• TP 03 • Clic 02
Yaskawa	• V1000

List of Protocol's Default Exceptions

The next table contains a list of default exceptions, defined by the Modbus protocol specification (version 1.1b).

Exceptions are registered in Driver's log, whenever detected, and they can be read by an application by **Reading the Last Exception Code**.

Notice that, in addition to the exceptions listed here, a device may define other proprietary exceptions. In this case, they are supposed to be described on manufacturer's documentation of that device.

Exception codes standardized by the Modbus protocol

CODE (IN HEXADECIMAL)	NAME	MEANING
01	ILLEGAL FUNCTION	The function code received is not valid. This may indicate that this function is not implemented yet or the slave is in an inadequate status to process it.
02	ILLEGAL DATA ADDRESS	The data address received is not a valid address. More specifically, the combination of a reference address and the amount of data to transfer is invalid.
03	ILLEGAL DATA VALUE	The current value on Master's request is not valid. This indicates a failure on the remaining data structure of a complex request, such as when the informed size for a data block is not correct. This exception does not indicate that values submitted for writing are out of the expected scope by an application, as such information is not accessible to this protocol.

CODE (IN HEXADECIMAL)	NAME	MEANING
04	SLAVE DEVICE FAILURE	An irrecoverable error occurred during the processing of the requested function.
05	ACKNOWLEDGE	Used with programming commands. The Slave accepted this message and it is processing it. However, this processing demands a long time. This exception prevents a Master time-out. The end of this request must be tested by a polling process.
06	SLAVE DEVICE BUSY	Used with programming commands. Indicates that the Slave is processing another long-lasting command and that this request must be sent again later, when the Slave is available again.
08	MEMORY PARITY ERROR	Used with functions **20** and **21**, *reference type 6*, to indicate the extended file failed on a consistency test. The Slave device may need maintenance.
0A	GATEWAY PATH UNAVAILABLE	Used together with gateways, to indicate that a gateway was unable to allocate an internal path to process this request. It usually indicates that a gateway is not correctly configured or it is overloaded.
0B	GATEWAY TARGET DEVICE FAILED TO RESPOND	Used together with gateways, to indicate that no response was received from the destination device. It usually indicates that such device is not available on the network.

Maximum Limit for the Size of Blocks Supported by the Protocol

This topic presents the maximum limits for block sizes supported by the Modbus protocol, in its current version 1.1b (please check this specification at *protocol's official website*).

Notice that, due to **Superblock Reading** and **Automatic Block Partition** features, present in this Driver's current version, users hardly need to consider these limits in their applications, as this Driver already performs all optimizations automatically during communication.

However, as there are devices that do not support default limits established by this protocol, users may need to know these limits, and most of all they must know how to evaluate device's limits, in case they are forced to perform a manual grouping (please check topic **Superblock Reading**). In these cases, information on this topic can be very helpful.

Limits Supported by this Protocol

The Modbus protocol defines a simple data unit called **PDU** (*Protocol Data Unit*), which is kept unchanged through several protocol modes and through several communication layers.

A full communication layer, including a PDU and other additional header fields, is called **ADU** (*Application Data Unit*).

Limits supported by this protocol
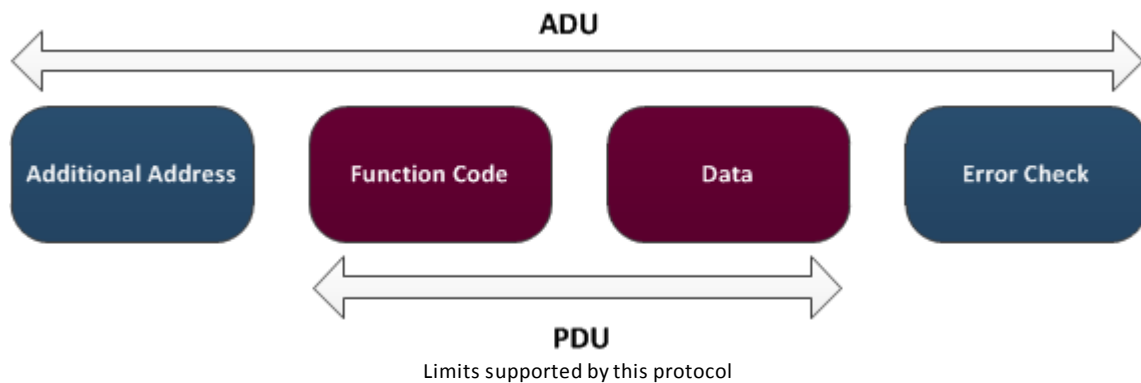
According to this protocol's specification, a full Modbus frame (ADU) can have a PDU with a maximum size of 253 bytes.

Thus, depending on the data type or Modbus function used in communication, this protocol imposes the limits for block elements at each communication described on the next table.

Limits for Block Elements

MODBUS FUNCTION	DESCRIPTION	LIMIT
03, 04	Reading multiple 16-bit registers	125 registers (250 bytes)
16	Writing multiple 16-bit registers (*Holding Registers*)	123 registers (247 bytes)
01, 02	Reading multiple bits	2000 bits (250 bytes)
15	Writing multiple bits	1968 bits (247 bytes)
20	Reading file registers	124 registers (248 bytes)
21	Writing file registers	122 registers (244 bytes)

More information is available at *protocol's official website*.

The article *KB-23112: I/O Block's ideal size with Modbus driver* in Elipse Knowledgebase presents a summary of questions relative to Tag grouping and block resizing in this Driver, discussed here and on other topics.

BCD Encoding

BCD Encoding (*Binary-Coded Decimal*) was originally created to circumvent limitations regarding the maximum number of digits that can be represented in traditional formats for storing values. Formats such as the representation of real numbers in floating point are usually acceptable for math and scientific calculations. However, rounding errors caused by numbers that cannot be represented due to overflow or underflow problems may not be acceptable in certain applications, such as financial procedures. To overcome that limitation, a BCD encoding was developed to allow representing numbers up to the last digit.

In this representation, each decimal digit is represented only in a binary format, without limitations regarding its number of digits.

The next table shows decimal digits and their corresponding values in BCD (values in binary).

Decimal digits in BCD encoding

DECIMAL	BCD	DECIMAL	BCD
0	0000b	5	0101b
1	0001b	6	0110b
2	0010b	7	0111b
3	0011b	8	1000b
4	0100b	9	1001b

To improve efficiency of this encoding, it is common to represent two digits per byte. Notice that, on the previous table, each decimal digit requires only four bits, or a half byte, for its representation.

Such representation with two digits in each byte is called **Packed BCD**, and this the representation used by this Driver, that is, packets sent by this Driver with BCD values use a data byte for every two digits of the represented decimal value. That is why the **Size** field, for **BCD** data types, must be defined as half the maximum number of digits represented in values that are read or written.

Example

As an example, suppose that users want to send the value 84 in decimal (0x54 in hexadecimal format), using a packed BCD encoding in one byte, the format used by this Driver.

The first step is separate the two decimal digits that compose this value in its decimal representation:

- **Digit 1**: 8

- **Digit 2**: 4

If users want to send this value to a device without a BCD encoding, then the value sent to the protocol is 84, which is represented in hexadecimal format by 0x54, or else 01010100b in binary format.

By using a packed BCD format, however, users represent these two decimal digits separately in each half, or nibble, of the byte to send:

- **BCD**: 0x84 or 10000100b

Notice that, if this value 0x84 is mistakenly interpreted in BCD format as a value in hexadecimal format without this encoding, and this value is then converted to decimal, users get the value 132, which is meaningless.

The next table presents a few more examples of decimal values between 0 (zero) and 99 and their respective representations in Packed BCD format in one byte, presented in hexadecimal and binary formats.

Decimal digits in Packed BCD encoding

DECIMAL	HEXADECIMAL	BCD (HEXADECIMAL)	BCD (BINARY)
10	0x0A	0x10	00010000b
0	0x00	0x00	00000000b
99	0x63	0x99	10011001b
81	0x51	0x81	10000001b
45	0x2D	0x45	01000101b

Driver Revision History

VERSION	DATE	AUTHOR	COMMENTS
3.1.30	04/01/2016	C. Mello	• Fixed a problem with the reading of Tags configured by **Strings** when the Superblock service is activated (*Case 20754*).

VERSION	DATE	AUTHOR	COMMENTS
		A. Quites	• Added Tag configuration by **Strings** (*Case 19119*). • Added new Tag Browser with Tags configured by **Strings** (*Case 20460*). • Improvements on error checking when configuring user-defined data types (*Case 20415*). • Fixed a bug when reading and writing 8-digit **BCD** data types (*Case 19733*). • Fixed a vulnerability in which the Driver allowed users to configure structures with names of native data types, which may be confusing (*Case 19816*). • Fixed an error when reading **UTC32** data types in Blocks (*Case 19819*). • Fixed an error when writing **Double** data types in Blocks, that is, when writing more than one Element at the same time (*Case 20053*). • Fixed an error when generating operations' INI files using the old Driver's 1.0 version, a feature used for compatibility reasons (*Case 20203*). • Fixed an error in which a mistaken byte order configuration for **BCD** data types generated a change in values (*Case 20204*). • Fixed an error in which the Elipse SOE routine does not return event-reported values to native, non-structured data types (*Case 20364*).

VERSION	DATE	AUTHOR	COMMENTS
			• Fixed an error when reading real-time event Tags from GE PAC RX7 controllers, when used with callbacks (*Case 20374*).
3.0.11	05/29/2015	A. Quites	• Driver ported to IOKit 2.0 (*Case 13891*).
			• Fixed an error on operations' configuration window, when removing an operation from the table could fail (*Case 14874*).
			• Fixed an error in which this Driver could behave erratically when the **Reconnect after timeout** option was enabled, when reading SOE Tags using callbacks. This error could, in rare situations, allow this Driver to try a disconnection several consecutive times, and also try reconnections several consecutive times as well (*Case 14775*).
			• Fixed a handle leak when downloading events from a GE PAC RX7 controller (*Case 16404*).
			• Fixed an error in **Use Bit Mask** feature when used with Superblocks enabled (*Case 18340*).
			• Fixed an error when reading **Strings** with odd sizes, where the last characters could be truncated if swap options were enabled (*Case 16744*).

VERSION	DATE	AUTHOR	COMMENTS
2.8.17	10/19/2012	A. Quites	• Implemented Elipse Modbus SOE, a feature that standardizes SOE readings for most PLCs (*Case 12038*).
			• Added user-defined types or structures, as part of implementing Elipse Modbus SOE feature (*Case 12038*).
			• Implemented SOE readings for Schneider Electric Series SEPAM 20, 40, and 80 relays (*Case 12106*).
			• Added a new **Sp_time** type to represent timestamps for Schneider Electric SEPAM Series relays (*Case 12106*).
			• Added date and time types **GenTime**, **UTC64d**, and **UTC32**, which can be used to represent timestamps (*Case 12038*).
			• Added support for callback readings in SOE Tags (*Case 12464*).
			• Added an option to reconnect in case of timeout when receiving frames, on Ethernet physical layer (*Case 12537*).
			• Redesigned the configuration window to allow adding new configuration options, by creating a new **Operations** tab (*Case 12038*).
			• Removed the **Swap Address Delay** option from Driver's configuration window. This option, turned obsolete by IOKit's **Inter-frame delay** option, is still available as an offline configuration, thus keeping compatibility with legacy applications (*Case 13285*).

VERSION	DATE	AUTHOR	COMMENTS
			• Removed the **Reverse Frame** option from Driver's configuration window for operations, because it is now obsolete. This option is still supported for legacy applications (*Case 12443*).
			• Added information about configurations for **Float** types in WEG TPW-03 controllers (*Case 12546*).
			• Added information about Driver's configuration for ABB 07KT97 controllers and all controllers from ABB Advant Controller 31 series 90 family (*Case 12667*).
			• Added ATOS PLC to the list of devices that support Modbus protocol (*Case 13247*).
			• Added information about automatic block partition feature, available since version 2.00 (*Case 11594*).
			• Added WEG Clic 02 PLC to the list of supported devices (*Case 11602*).
			• Revised text of introductory topic to clearly state that this Driver works as Master on a Modbus network, allowing communication with Slave devices (*Case 12035*).
			• Added a topic about Superblocks (the **EnableReadGrouping** property) (*Case 13287*).
			• Added a topic about automatic partitioning of blocks (*Case 13288*).
			• Added a topic with optimization tips (*Case 13287*).

VERSION	DATE	AUTHOR	COMMENTS
			• Added to topic **Frequently Asked Questions** a question about a message referring to reception of non-normalized **Float** values (*Case 13295*).
			• Updated the list of devices that communicate using Modbus protocol, considering devices registered during recent support calls (*Case 13298*).
			• Added topic **Quick Configuration Guide**, with a step-by-step procedure to configure this Driver for the most common usage cases (*Case 13301*).
			• Fixed an error when loading a configuration file on Windows CE ARM HPC2000 platform (*Case 12352*).
			• Fixed an error when writing **float_GE** types (*Case 12298*).
			• Fixed an error in which the last character of a **String** with an odd number of characters could not be returned to an application (*Case 12466*).
2.7.1	06/30/2010	A. Quites	• Adjustments in data reception to discard possible invalid bytes (*Case 11394*).
		C. Mello	• Updated topic **Reading the Last Exception Code** (*Case 11233*). • Updated topic **Frequently Asked Questions** (*Case 11316*).
2.6.3	11/26/2009	A. Quites	• Removed a false error message in the log file when reading memory data with address zero (*Case 10654*). • Optimized reading of **Bit** types via Superblocks (*Case 10971*).

VERSION	DATE	AUTHOR	COMMENTS
		C. Mello	• Changed the **Wait Silence** procedure to be performed on any errors occurring in this Driver. Users can also execute the **Wait Silence** procedure manually by using the new Special Tag *N2* = 9001 (*Case 10850*). • Added compatibility to Windows CE platform (*Case 10914*).
2.5.12	06/30/2009	A. Quites	• Fixed an error in the **Swap Address Delay** option (*Case 10425*). • Updated the default value for PDU's maximum size, according to the most recent Modbus standard (*Case 10274*). • Fixed an error in special Tags that read the last exception, which could not report some exceptions (*Case 10337*). • Fixed an error when returning the year in SOE registers with the current day for GE PAC RX7 controllers (*Case 10382*). • Fixed an error in function **20** (*Read File Record*) (*Case 10312*).
		M. Ludwig	• Added a new Tag for reading SOE events from specific points for GE PAC RX7 controllers (*Case 10370*).
2.4.10	02/17/2009	A. Quites	• Fixed an error when reading **Bit** types with Superblocks (*Case 10100*). • Implemented reading of the most recent GE SOE event (*Case 10178*). • Created default operations for the most common function types (*Case 9185*).
2.3.22	09/02/2008	C. Mello	• Added a new **Float_GE** data type (*Case 9427*).

VERSION	DATE	AUTHOR	COMMENTS
		A. Quites	• Fixed an error on offline configuration of the *ModiconModbus.Modbus Mode* parameter (*Case 9831*). • Implemented reading of bit masks for registers (*Case 9682*). • Implemented reading of an event buffer on GE PAC RX7 controllers (*Case 9523*). • Implemented the possibility of configuring a maximum limit for PDU (*Case 9154*). • Fixed an error in byte swapping features (**Swap Byte**, **Swap Word**, **Swap DWord**, and **Rev. Frame**) when reading **Word** types with Superblocks enabled (*Case 9220*). • Implemented the **Enable CMS Addressing** option (*Case 8665*). • Revisions and tests in rarely used Modbus functions (*Case 8730*). • Improved consistency on data written using Modbus function **06** (*Write Single Register*) (*Case 8663*).
2.2	05/11/2007	A. Quites	• Fixed an error when reading blocks of non-**Word** types with Superblocks enabled (*Case 8243*).

VERSION	DATE	AUTHOR	COMMENTS
2.1	01/23/2007	A. Quites	• Added support for Superblocks (*Case 6185*). • Improved consistency of *N2/B2* parameters for Tags accessing registers (*Case 7714*). • Fixed an error when reading **BCD**-type data blocks with size four (*Case 7728*). • Fixed an error when reading blocks of **Strings** (*Case 7804*).
2.0	11/10/2006	A. Quites	• Original version with IOKit (*Case 3339*).
1.0		R. Farina	• Every release published before revision control.

Headquarters
Rua 24 de Outubro, 353 - 10º andar
90510-002 Porto Alegre
Phone: (+55 51) 3346-4699
Fax: (+55 51) 3222-6226
E-mail: elipse-rs@elipse.com.br

Taiwan
9F., No.12, Beiping 2nd St., Sanmin Dist.
807 Kaohsiung City - Taiwan
Phone: (+886 7) 323-8468
Fax: (+886 7) 323-9656
E-mail: evan@elipse.com.br

Check our website for information about a representative in your country.

www.elipse.com.br

kb.elipse.com.br

forum.elipse.com.br

www.youtube.com/elipsesoftware

elipse@elipse.com.br

Microsoft Partner
Gold Independent Software Vendor (ISV)