Modbus provider

Modbus communication

Version 1.0.0

User's guide

Jan 10, 2017

[Remarks]

Modbus provider has not been supported recently. Please use Modbus.X provider instead.

[Revision history]

Version	Date	Contents
1.0.0.0	2010-04-26	First edition
1.0.0	2012-07-17	Changed the version rule of document.

[Supported models]

Model	Version	Note

Contents

1. Introduction

This is a user’s guide of Modbus provider.

Modbus provider runs as a Modbus master and it communicate with a Modbus slave.

With this provider, CAO client can send and receive Modbus commands easily.

This document describes functions and implemented method of Modbus provider.

2. Overview of provider

2.1. Overview

Modbus provider sends and receives Modbus protocol. Communication system is serial communication.

The file format of Modbus provider is DLL (Dynamic Link Library) and Table 2-1 shows the detail.

Table 2-1 Modbus provider

File name	CaoProvModbus.dll
ProgID	CaoProv.Modbus
Registration[1]	regsvr32 CaoProv Modbus.dll
Deregistration	regsvr32 /u CaoProv Modbus.dll

[1] You do not need to register/deregister manually if it is installed with ORiN2 SDK.

2.2. Method and Properties

2.2.1. CaoWorkspace::AddController method

Modbus provider establishes communication by specifying connection parameters at the timing of AddController.

You can set options (communication configuration, connection parameters, timeout, synchronous setting, escape character) by specifying option character strings.

The following shows argument specification of AddController.

```
AddController
(
    "<Controller name>",  // Controller name
    "CaoProv.Modbus",     // Provider name. Fixed.
    "<Computer name>",    // Computer name where provider runs.
    "<Option>"            // Option character string
)
```

The following table shows the list of option character strings.

Table 2-2 Option character string of CaoWorkspace::AddController

Option	Description
Conn =<Connection parameter>	Required. Communication configuration and connection parameters. (See 2.2.1.1)
Timeout [=<Timeout period>]	Timeout period for sending and receiving (milliseconds) (Default : 500)
Retry [=<Retry count>]	Communication retry count at the sending and receiving. (Default : 0)

2.2.1.1. Conn option

The following shows the connection parameter strings of Conn option. Parameters surrounded by the square brackets ([]) can be omitted. Underlined part shows the default value when the option is not specified.

RS232C device

“Conn=com:<COM Port>[:<BaudRate>[:<Parity>:<DataBits>:<StopBits>[:<Flow>]]]”

<COM Port>	:	COM port number. ‘1’-COM1,’2’-COM2,…
<BaudRate>	:	Communication speed. 4800,9600,19200,38400,57600,115200.
<Parity>	:	Parity. ’N’-NONE,’E’-EVEN,’O’-ODD.
<DataBits>	:	Data bit count. ‘7’-7bit,’8’-8bit.
<StopBits>	:	Stop bit. ‘1’-1bit,’2’-2bit.
<Flow>	:	Flow control.’1’-Xon/Xoff,’2’-Hardware control. You can specify with logical OR.

2.2.2. CaoController::Execute method

For about available command names and details,refer to3.1.

If this method is used for broadcasting command, a timeout error will occur.

```
Execute
(
    "<Command name>",              // Command name
    "<Parameter>"                  // Command parameter
)
```

3. Command reference

Table 3-1 CaoController::Execute command list

Command	Function ID	Description	
Raw	-	Raw Packet	
ReadCoilStatus	0x01	Read Coil Status	P. 9
ReadInputStatus	0x02	Read Input Status	P. 9
ReadHoldingRegister	0x03	Read Holding Registers	P. 10
ReadInputRegister	0x04	Read Input Registers	P. 10
ForceSingleCoil	0x05	Force Single Coil	P. 10
PresetSingleRegister	0x06	Preset Single Register	P. 11
ReadExceptionStatus	0x07	Read Exception Status	P. 11

Diagnostics	0x08	Diagnostics	P. 11
FetchCommunicationEventCounter	0x0B	Fetch Comm Event Counter	P. 12
FetchCommunicationEventLog	0x0C	Fetch Comm Event Log	P. 12
ForceMultipleCoils	0x0F	Force Multiple Coils	P. 12
PresetMultipleRegisters	0x10	Preset Multiple Registers	P. 13
ReportSlaveID	0x11	Report Slave ID	P. 13
ReadGeneralReference	0x14	Read General Reference	P. 14
WriteGeneralReference	0x15	Write General Reference	P. 14
MaskWrite4XRegister	0x16	Mask Write 4X Register	P. 15
ReadWrite4XRegisters	0x17	Read/Write 4X Registers	P. 15
ReadFIFOQueue	0x18	Read FIFO Queue	P. 16

3.1. Controller class

Raw

Syntax	*object*.Raw(<Data>)
Argument	<Slave> = VT_ARRAY | VT_UI1: Sending data
Return value	<Data> = VT_ARRAY | VT_UI1: Receiving data
Description	The specified sending data is add a CRC and then sent. Once the response packet arrives, CRC is deleted from the received packet and the result is returned.

ReadCoilStatus

Syntax	*object*.ReadCoilStatus(<Slave>, <Address>, <Points Count>)
Argument	<Slave> = VT_UI1: Slave address <Address> = VT_UI2:　Reading start address <Points Count> = VT_UI2: Reading data count
Return value	<Data> = VT_ARRAY | VT_UI1: Read data
Description	Read the ON/OFF state of DO (Discrete Output) of Slave address. The state of the read DO will be allocated to the return value <Data> in every one-bit from the lowest bit

ReadInputStatus

Syntax	*object*.ReadInputStatus(<Slave>, <Address>, <Points Count>)
Argument	<Slave> = VT_UI1: Slave address <Address> = VT_UI2: Reading start address <Points Count> = VT_UI2: Reading data count
Return value	<Data> = VT_ARRAY | VT_UI1: Read data
Description	Read the ON/OFF state of DI (Discrete Input) of the Modbus slave. The read DI state is allocated to the return value <Data> in every one-bit from the lowest

bit.

ReadHoldingRegister

Syntax	*object*.ReadHoldingRegister(<Slave>, <Address>, <Points Count>)
Argument	<Slave> = VT_UI1: Slave address <Address> = VT_UI2: Reading start address <Points Count> = VT_UI2: Reading data count
Return value	<Data> = VT_ARRAY | VT_UI2: Read data
Description	Read the content of the holding register of the Modbus slave.

ReadInputRegister

Syntax	*object*.ReadInputRegister(<Slave>, <Address>, <Points Count>)
Argument	<Slave> = VT_UI1: Slave address <Address> = VT_UI2: Reading start address <Points Count> = VT_UI2: Reading data count
Return value	<Data> = VT_ARRAY | VT_UI2: Read data
Description	Read the content of the Input register of the Modbus slave.

ForceSingleCoil

Syntax	*object*.ForceSingleCoil(<Slave>, <Address>, <Data>)
Argument	<Slave> = VT_UI1: Slave address <Address> = VT_UI2: Reading start address <Data> = VT_UI2: Force data
Return value	<Address> = VT_UI2: Address <Data> = VT_UI2: Force data
Description	Change the content of DO (Discrete Output) of the Modbus slave.

<Address> and <Data> of the return value will be the same as the ones specified by Argument.

PresetSingleRegister

Syntax	*object*.PresetSingleRegister(<Slave>, <Address>, <Data>)
Argument	**<Slave>** = VT_UI1: Slave address **<Address>** = VT_UI2: Writing start address **<Data>** = VT_UI2: Preset data
Return value	**<Address>** = VT_UI2: Address **<Data>** = VT_UI2: Preset data
Description	Change the content of the holding register of the Modbus slave. <Address> and <Data> of the return value will be the same as the ones specified by Argument.

ReadExceptionStatus

Syntax	*object*.ReadExceptionStatus(<Slave>)
Argument	**<Slave>** = VT_UI1: Slave address
Return value	**<Data>** = VT_UI1: Exception status
Description	Read the exception status of the Modbus slave. The slave returns the exception status every one bit from the lowest bit.

Diagnostics

Syntax	*object*.Diagnostics(<Slave>, <Check Code>, <Data>)
Argument	**<Slave>** = VT_UI1: Slave address **<Check Code>** = VT_UI2: Diagnosis sub code **<Data>** = VT_UI2: Data
Return	**<Check Code>** = VT_UI2: Diagnosis sub code

value	**<Data>** = VT_UI2: Data
Description	This diagnoses the communication between the Modbus master and slave and the Modbus slave device. <Check Code> and <Data> of Return value are the same ones specified by Argument.

FetchCommunicationEventCounter

Syntax	*object*.FetchCommunicationEventCounter(<Slave>)
Argument	**<Slave>** = VT_UI1: Slave address
Return value	**<Status>** = VT_UI2: Status **<Event Counter>** = VT_UI2: Event counter
Description	From the communication event counter of the Modbus slave, read the status and event counter.

FetchCommunicationEventLog

Syntax	*object*.FetchCommunicationEventLog(<Slave>)
Argument	**<Slave>** = VT_UI1: Slave address
Return value	**<Status>** = VT_UI2: Status **<Event Counter>** = VT_UI2: Event counter **<Msg Counter>** = VT_UI2: Message counter **<Event>** = VT_ARRAY | VT_UI2: Event
Description	Read the communication event log (status, event counter, message counter, and event) of the Modbus slave.

ForceMultipleCoils

Syntax	*object*.ForceMultipleCoils(<Slave>, <Address>, <Data Count>, <Data>)
Argument	**<Slave>** = VT_UI1: Slave address **<Address>** = VT_UI2: Writing start address

<Data Count> = VT_UI2: Data count
<Data> = VT_ARRAY | VT_UI1: Data

Return value <Start> = VT_UI1: Writing start address
<Data Count> = VT_UI1: Data count

Description Write the status of the consecutive DOs (Discrete Output) of the Modbus slave.
DO status to write are allocated from the lowest bit of <Data> argument.
<Start Address> and <Data Count> of the return value will be the same as the ones specified by Argument.

PresetMultipleRegisters

Syntax *object*.PresetMultipleRegisters(<Slave>, <Address>, <Data Count>, <Data>)

Argument <Slave> = VT_UI1: Slave address
<Address> = VT_UI2: Writing start address
<Data> = VT_ARRAY | VT_UI2: Data

Return value <Start Address> = VT_UI1: Writing start address
<Data Count> = VT_UI1: Writing data count

Description Write data into consecutive holding registers of the Modbus slave.
<Start Address> of the return value will be the same as the one specified by Argument.

ReportSlaveID

Syntax *object*.ReportSlaveID(<Slave>)

Argument <Slave> = VT_UI1: Slave address

Return value <Controller Type> = VT_UI1: Slave controller type.

0	Micro 84
1	484
2	184/384
3	584
8	884
9	984

<Run Indicator Status> = VT_UI1: Run indicator status

<Data> = VT_UI1: Additional information. Content differs depending on the device used.

Description Read the Modbus slave information.

ReadGeneralReference

Syntax *object*.ReadGeneralReference(<Slave>, <Sub Request>)

Argument

<Slave> = VT_UI1: Slave address

<Sub Request> = VT_ARRAY | VT_VARIANT: Sub request (<Req1>, <Req2>, …)

<Req n> = VT_ARRAY | VT_VARIANT: Request number (<Ref Type>, <File No.>, <Address>, <Data Count>)

<Ref Type> = VT_UI1:Reference type

<File No.> = VT_UI2: Extended memory file number

<Address> = VT_UI2:Reading start address

<Data Count> = VT_UI2:Data count

Return value

<Sub Response> = VT_ARRAY | VT_VARIANT: Sub response (<Res1>, <Res2>, …)

<Res n> = VT_ARRAY | VT_VARIANT: Response number (<Ref Type>, <Data>)

<Ref Type> = VT_UI1: Reference type

<Data> = VT_ARRAY | VT_UI2: Data

Description Read the content of the extended memory file register of the Modbus slave.

WriteGeneralReference

Syntax *object*.WriteGeneralReference(<Slave>, <Sub Request>)

Argument

<Slave> = VT_UI1: Slave address

<Sub Request> = VT_ARRAY | VT_VARIANT: Sub request (<Req1>, <Req2>, …)

<Req n> = VT_ARRAY | VT_VARIANT: Request number (<Ref Type>, <File No.>, <Address>, <Data>)

<Ref Type> = VT_UI1: Reference type

<File No.> = VT_UI2: Extended memory file number

<Address> = VT_UI2: Writing start address

<Data> = VT_ARRAY | VT_UI2: Data

Return value

<Sub Response> = VT_ARRAY | VT_VARIANT: Sub response (<Res1>, <Res2>, …)
  <Res n> = VT_ARRAY | VT_VARIANT: Response number
   (<Ref Type>, <File No.>, <Address>, <Data>)
   <Ref Type> = VT_UI1:Reference type
   <File No.> = VT_UI2: Extended memory file number
   <Address> = VT_UI2: Writing start address
   <Data> = VT_ARRAY | VT_UI2:Data

Description

Write data into the extended memory file register of the Modbus slave.
<Res n> of Return value will be the same as the one specified by Argument.

MaskWrite4XRegister

Syntax

`object.MaskWrite4XRegister(<Slave>, <Address>, <AND_Mask>, <OR_Mask>)`

Argument

<Slave> = VT_UI1: Slave address
<Address> = VT_UI2: Address
<AND_Mask> = VT_UI2: AND mask
<OR_Mask> = VT_UI2: OR mask

Return value

<Address> = VT_UI2: Address
<AND_Mask> = VT_UI2: AND mask
<OR_Mask> = VT_UI2: OR mask

Description

Modify the content of the specified Modbus slave's 4XXXX address using a combination of an AND mask, and OR mask, and the register's current values.
The function's algorithm is :
[Result] = ([Current value] AND <AND_Mask>) OR ($\overline{\text{<OR\_Mask>}}$ AND <AND_Mask>)

<Address>, <AND_Mask>, <OR_Mask> of Return value are the same as the ones specified in Argument.

ReadWrite4XRegisters

Syntax

`object.ReadWrite4XRegisters(<Slave>, <Read Address>, <Read Count>, <Write Address>, <Write Data>)`

Argument

<Slave> = VT_UI1: Slave address
<Read Address> = VT_UI2: Reading start address
<Read Count> = VT_UI2: Reading data count
<Write Address> = VT_UI2: Writing start address
<Write Data> = VT_ARRAY | VT_UI2: Writing data

Return value

<Data> = VT_ARRAY | VT_UI2: Read data

Description

Perform a combination of one read and one write operation of the 4XXXX register of the Modbus slave in a single Modbus transaction.
This writes <Write Data> to the address specified by <Write Start>, and then reads the contents of the address specified by <Read Start> by the number of <Read Count>.

ReadFIFOQueue

Syntax

object.ReadFIFOQueue(<Slave>, <Address>, <AND_Mask>, <OR_Mask>)

Argument

<Slave> = VT_UI1: Slave address
<Address> = VT_UI2: Reading start address

Return value

<Data> = VT_ARRAY | VT_UI2: Read data

Description

Read the content of FIFO queue in the 4XXXX register of the Modbus slave.