Modbus プロバイダ

Modbus 通信

Version 1.0.0

ユーザーズ ガイド

Jan 10, 2017

【備考】
Modbus プロバイダは現在サポートされていません.Modbus.X プロバイダを使用してください.

【改版履歴】

バージョン	日付	内容
1.0.0.0	2010-04-26	初版
1.0.0	2012-07-17	ドキュメントのバージョンルールを変更

【対応機器】

機種	バージョン	注意事項

目次

1. はじめに

本書は Modbus プロバイダのユーザーズガイドです.

Modbus プロバイダはマスタとして動作し, Modbus スレーブに対して通信を行います.

この Modbus プロバイダを使用することで, CAO クライアントは Modbus コマンドの送受信を簡単に使用することができます.

本書は, この Modbus プロバイダの機能と実装されているメソッドについて説明します.

2. プロバイダの概要

2.1. 概要

Modbus プロバイダは, Modbus プロトコルを送受信するプロバイダです. 通信形態は, シリアル通信となります.

Modbus プロバイダのファイル形式は DLL(Dynamic Link Library)となっており, その詳細は表 2-1 のようになっています.

表 2-1 Modbus プロバイダ

ファイル名	CaoProvModbus.dll
ProgID	CaoProv.Modbus
レジストリ登録[1]	regsvr32 CaoProv Modbus.dll
レジストリ登録の抹消	regsvr32 /u CaoProv Modbus.dll

[1] ORiN SDK でインストールした場合は手動で登録／抹消する必要はありません.

2.2. メソッド・プロパティ

2.2.1. CaoWorkspace::AddController メソッド

RAC プロバイダでは，AddController で接続パラメータを設定し，通信の接続を行います．

このときオプションで通信形態，接続パラメータ，タイムアウトの設定，同期設定，エスケープ文字の指定を行います．

以下に AddController の引数仕様を示します．

```
AddController
(
    "<Controller 名>",      // コントローラ名
    "CaoProv.RAC",          // プロバイダ名．固定．
    "<マシン名>",           // プロバイダの実行マシン名．
    "<オプション>"          // オプション文字列
)
```

以下にオプション文字列に指定するリストを示します．ここで，通信デバイスの欄に“-”が入っているオプションは，そのデバイスを指定したときに無視されます．

表 2-2 CaoWorkspace::AddController のオプション文字列

オプション	説明
Conn =<接続パラメータ>	必須．通信形態と接続パラメータ． （参照 2.2.1.1）
Timeout [=<タイムアウト時間>]	送受信時のタイムアウト時間．(ミリ秒) （デフォルト：500）
Retry [=<リトライ回数>]	送受信時の通信リトライ回数． （デフォルト：0）

2.2.1.1. Conn オプション

以下に Conn オプションの接続パラメータ文字列を示します. ここで角括弧(“[ ]”)内は省略可能を示します. また, 各パラメータの解説中の下線部はオプションを指定しなかったときのデフォルト値になります.

RS232C デバイス

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

<COM Port>	:	COM ポート番号. ‘1’-COM1, ‘2’-COM2, …
<BaudRate>	:	通信速度. 4800, 9600, 19200, 38400, 57600, 115200.
<Parity>	:	パリティ. ‘N’-NONE, ‘E’-EVEN, ‘O’-ODD.
<DataBits>	:	データビット数. ‘7’-7bit, ‘8’-8bit.
<StopBits>	:	ストップビット数. ‘1’-1bit, ‘2’-2bit.
<Flow>	:	フロー制御. ‘1’-Xon/Xoff, ‘2’-ハードウェア制御. OR をとって指定できます.

2.2.2. CaoController::Execute メソッド

使用できるコマンド名と詳細は 3.1 を参考にしてください.

ブロードキャスト通信を行ったコマンドでは, 必ずタイムアウトエラーが発生します.

```
Execute
(
    "<コマンド名>",                 // コマンド名
    "<パラメータ>"                  // コマンドパラメータ
)
```

3. コマンドリファレンス

表 3-1 CaoController::Execute コマンド一覧

コマンド	FunctionID	機能	
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. コントローラクラス

Raw

構文 *object*.Raw(<Data>)

引数 <Slave> = VT_ARRAY | VT_UI1: 送信データ

戻り値 <Data> = VT_ARRAY | VT_UI1: 受信データ

説明 指定された送信データに CRC を付加して送信し，受信したデータから CRC を削除したパケットを返します.

ReadCoilStatus

構文 *object*.ReadCoilStatus(<Slave>, <Address>, <Points Count>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス
<Points Count> = VT_UI2: 読み取り点数

戻り値 <Data> = VT_ARRAY | VT_UI1: 読み出しデータ

説明 スレーブの DO（Discrete Output）の ON / OFF 状態を読み出します.
読み出したDO状態は，戻り値の<Data>に下位ビットから1ビットずつ割り当てられています.

ReadInputStatus

構文 *object*.ReadInputStatus(<Slave>, <Address>, <Points Count>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス
<Points Count> = VT_UI2: 読み取り点数

戻り値 <Data> = VT_ARRAY | VT_UI1: 読み出しデータ

説明 スレーブの DI（Discrete Input）の ON / OFF 状態を読み出します.
読み出した DI 状態は，戻り値の<Data>に下位ビットから1ビットずつ割り当てられています.

ReadHoldingRegister

構文 *object*.ReadHoldingRegister(<Slave>, <Address>, <Points Count>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス
<Points Count> = VT_UI2: 読み取り点数

戻り値 <Data> = VT_ARRAY | VT_UI2: 読み出しデータ

説明 スレーブの保持レジスタの内容を読み出します.

ReadInputRegister

構文 *object*.ReadInputRegister(<Slave>, <Address>, <Points Count>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス
<Points Count> = VT_UI2: 読み取り点数

戻り値 <Data> = VT_ARRAY | VT_UI2: 読み出しデータ

説明 スレーブの入力レジスタの内容を読出します.

ForceSingleCoil

構文 *object*.ForceSingleCoil(<Slave>, <Address>, <Data>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス
<Data> = VT_UI2: 変更データ

戻り値 <Address> = VT_UI2: アドレス
<Data> = VT_UI2: 変更データ

説明 スレーブの DO(Discrete Output)の状態を変更します.
戻り値の<Address>及び<Data>は引数で指定した変更データと同じ値になります.

PresetSingleRegister

構文 *object*.PresetSingleRegister(<Slave>, <Address>, <Data>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 書き込み開始アドレス
<Data> = VT_UI2: 変更データ

戻り値 <Address> = VT_UI2: アドレス
<Data> = VT_UI2: 変更データ

説明 スレーブの保持レジスタの内容を変更します.
戻り値の<Address>及び<Data>は引数で指定した変更データと同じ値になります.

ReadExceptionStatus

構文 *object*.ReadExceptionStatus(<Slave>)

引数 <Slave> = VT_UI1: スレーブアドレス

戻り値 <Data> = VT_UI1: 例外ステータス

説明 スレーブの例外ステータスの状態を読み出します.
読み出した例外ステータス, 読み出しデータの下位ビットから1ビットずつ割り当てられています.

Diagnostics

構文 *object*.Diagnostics(<Slave>, <Check Code>, <Data>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Check Code> = VT_UI2: 診断サブコード
<Data> = VT_UI2: データ

戻り値 <Check Code> = VT_UI2: 診断サブコード
<Data> = VT_UI2: データ

説明 マスターとスレーブ間の通信の診断やスレーブの機器の診断ファンクションです.
戻り値の<Check Code>及び<Data>は引数で指定した変更データと同じ値になります.

FetchCommunicationEventCounter

構文 *object*.FetchCommunicationEventCounter(<Slave>)

引数 <Slave> = VT_UI1: スレーブアドレス

戻り値 <Status> = VT_UI2: ステータス
<Event Counter> = VT_UI2: イベントカウンタ

説明 スレーブの通信イベントカウンタからステータスとイベントカウンタを読み出します.

FetchCommunicationEventLog

構文 *object*.FetchCommunicationEventLog(<Slave>)

引数 <Slave> = VT_UI1: スレーブアドレス

戻り値 <Status> = VT_UI2: ステータス
<Event Counter> = VT_UI2: イベントカウンタ
<Msg Counter> = VT_UI2: メッセージカウンタ
<Event> = VT_ARRAY | VT_UI2: イベント

説明 スレーブの通信イベントログ(ステータス, イベントカウンタ, メッセージカウントおよびイベント)を読み出します.

ForceMultipleCoils

構文 *object*.ForceMultipleCoils(<Slave>, <Address>, <Data Count>, <Data>)

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 書込み開始アドレス
<Data Count> = VT_UI2: データ数
<Data> = VT_ARRAY | VT_UI1: データ

戻り値 <Start> = VT_UI1: 書込み開始アドレス
<Data Count> = VT_UI1: データ数

説明 スレーブの連続した複数の DO(Discrete Output)の状態を書き込みます.

書き込む DO 状態は，引数の<Data>に下位ビットから1ビットずつ割り当てます．
戻り値の<Start Address>及び<Data Count>は引数で指定した変更データと同じ値になります．

PresetMultipleRegisters

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

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 書込み開始アドレス
<Data> = VT_ARRAY | VT_UI2: データ

戻り値 <Start Address> = VT_UI1: 書込み開始アドレス
<Data Count> = VT_UI1: 書き込みデータ数

説明 スレーブの連続した複数の保持レジスタにデータを書き込みます．
戻り値の<Start Address>は引数で指定した変更データと同じ値になります．

ReportSlaveID

構文 *object*.ReportSlaveID(<Slave>)

引数 <Slave> = VT_UI1: スレーブアドレス

戻り値 <Controller Type> = VT_UI1: スレーブのコントローラタイプ

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

<Run Indicator Status> = VT_UI1: 実行インジケータ状態
<Data> = VT_UI1: 追加情報，内容はデバイスに依存します．

説明 スレーブの情報を読み出します．

ReadGeneralReference

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

引数 <Slave> = VT_UI1: スレーブアドレス
<Sub Request> = VT_ARRAY | VT_VARIANT:検出位置リスト (<Req1>, <Req2>, …)
<Req n> = VT_ARRAY | VT_VARIANT:検出位置
(<Ref Type>, <File No.>, <Address>, <Data Count>)
<Ref Type> = VT_UI1:参照タイプ
<File No.> = VT_UI2:拡張メモリファイル番号
<Address> = VT_UI2:読み取り開始アドレス
<Data Count> = VT_UI2:データ数

戻り値 <Sub Response> = VT_ARRAY | VT_VARIANT:検出位置リスト (<Res1>, <Res2>, …)
<Res n> = VT_ARRAY | VT_VARIANT:検出位置 (<Ref Type>, <Data>)
<Ref Type> = VT_UI1:参照タイプ
<Data> = VT_ARRAY | VT_UI2:データ

説明 スレーブの拡張メモリレファイルジスタの内容を読み出します.

WriteGeneralReference

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

引数 <Slave> = VT_UI1: スレーブアドレス
<Sub Request> = VT_ARRAY | VT_VARIANT:検出位置リスト (<Req1>, <Req2>, …)
<Req n> = VT_ARRAY | VT_VARIANT:検出位置
(<Ref Type>, <File No.>, <Address>, <Data>)
<Ref Type> = VT_UI1:参照タイプ
<File No.> = VT_UI2:拡張メモリファイル番号
<Address> = VT_UI2:書き込み開始アドレス
<Data> = VT_ARRAY | VT_UI2:データ

戻り値 <Sub Response> = VT_ARRAY | VT_VARIANT:検出位置リスト (<Res1>, <Res2>, …)
<Res n> = VT_ARRAY | VT_VARIANT:検出位置
(<Ref Type>, <File No.>, <Address>, <Data>)
<Ref Type> = VT_UI1:参照タイプ
<File No.> = VT_UI2:拡張メモリファイル番号

<Address> = VT_UI2:書き込み開始アドレス
<Data> = VT_ARRAY | VT_UI2:データ

説明 スレーブの拡張メモリレファイルジスタにデータを書き込みます.
戻り値の<Res n>は引数と指定した値と同じになります.

MaskWrite4XRegister

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

引数
<Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: アドレス
<AND_Mask> = VT_UI2: AND マスク
<OR_Mask> = VT_UI2: OR マスク

戻り値
<Address> = VT_UI2: アドレス
<AND_Mask> = VT_UI2: AND マスク
<OR_Mask> = VT_UI2: OR マスク

説明 現在値, AND マスク, OR マスクの組み合わせて, 指定したスレーブの4XXXX アドレスの値を更新します.
更新する値は以下の式を使用して計算されます.
[結果] = ([現在地] AND <AND_Mask>) OR (<OR_Mask> AND $\overline{\text{<AND\_Mask>}}$)

戻り値の<Address>, <AND_Mask>, <OR_Mask>は引数と指定した値と同じになります.

ReadWrite4XRegisters

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

引数
<Slave> = VT_UI1: スレーブアドレス
<Read Address> = VT_UI2: 読み取り開始アドレス
<Read Count> = VT_UI2: 読み取り点数
<Write Address> = VT_UI2: 書き込み開始アドレス
<Write Data> = VT_ARRAY | VT_UI2: 書き込みデータ

戻り値 <Data> = VT_ARRAY | VT_UI2: 読み出しデータ

説明 スレーブの 4XXXX レジスタの読み込みと書き込みを実行します.
<Write Start>で指定したアドレスに<Write Data>を書き込み, <Read Start>で指定したアドレスから<Read Count>の数だけデータを読み出します.

ReadFIFOQueue

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

引数 <Slave> = VT_UI1: スレーブアドレス
<Address> = VT_UI2: 読み取り開始アドレス

戻り値 <Data> = VT_ARRAY | VT_UI2: 読み出しデータ

説明 スレーブの 4XXXX レジスタにある FIFO キューの内容を読み込みます.