Developer's Guide

WISE-PaaS/RMM 3.1

**Wireless IoT Sensing Embedded Agent
WISE-Agent Modbus Handler
Developer's Guide**

ADVANTECH

Enabling an Intelligent Planet

Change Log:

Date	Version	Description / Major change
2015/11/30	V0.1	Zach Chih, create draft document
2015/12/15	V1.0	Zach Chih, first formal release version
2016/2/17	V1.1	Zach Chih, add function description
2016/3/7	V1.2	Zach Chih, modify AI addresses matching modbus spec & fix func code 5,6
2016/3/14	V2.0	Zach Chih, 1. add support of different handlers' names with different INI files 2. change naming AI,AO,DI,DO to IR,R,IB,B & redefine sections in INI from modbus spec. 3. change SlaveIP & SlavePort to ClientIP &ClientPort

Table of Content

1   Introduction

WISE Agent – a software development framework to communicate between device and RMM Server

Advantech provides a software development framework to communicate and exchange information between a device and RMM Server called **Wireless IoT Solutions Embedded Agent** (WISE Agent).

WISE Agent framework provides a rich set of user-friendly, intelligent and integrated interfaces, which speeds development, enhances security and makes agent application easier and simpler to communicate with RMM Server.

Modbus handler is one of the handlers in the WISE Agent used to handle the communication between based on Modbus TCP protocol.

1.1   Framework Architecture

The architecture of a WISE Agent framework with Modbus handler is described as follows. The agent includes three different layers: Provisioning & Communication, Core Management and Handlers. The Modbus handler is in the Handlers layer and is loaded by Handler Loader.
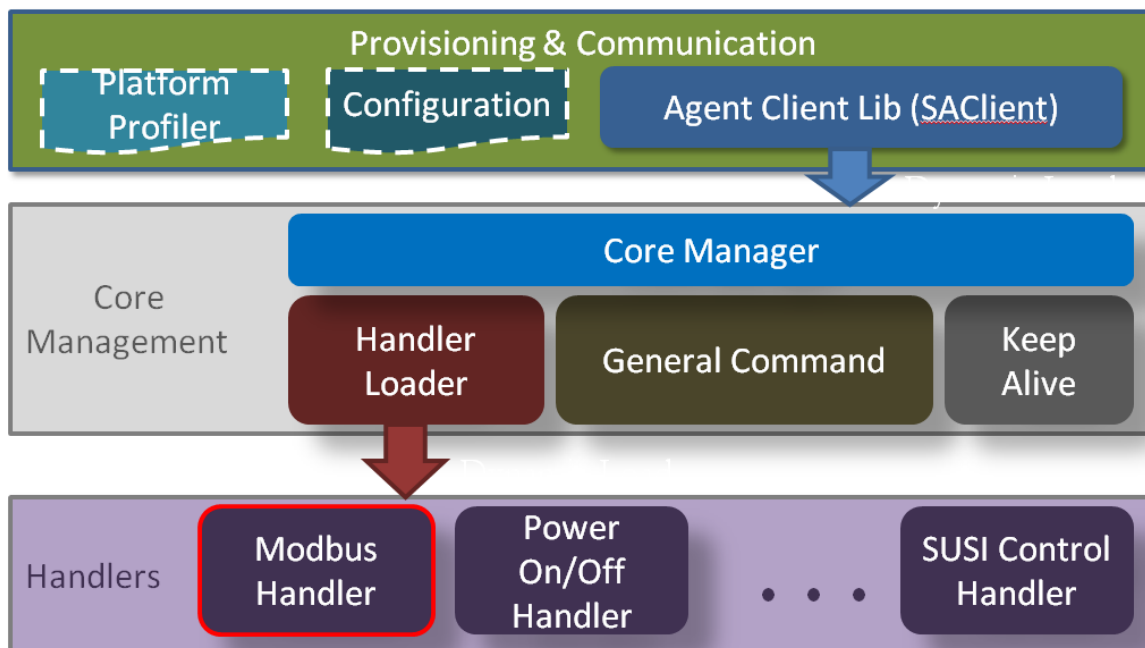
1.1.1 Provisioning & Communication

The Provisioning & Communication layer is the most important layer in this framework. The Provisioning & Communication layer include one library to connect to RMM Server and two structures defined the device information and server configuration.

Agent Client Library:

The library, named 'SAClient', is the main library user need to integrate into application, and it provides several simple API to communicate and exchange data with RMM Server.

Platform Profiler:

The Structure defined the Agent Profile include Agent, Platform and Custom Information. Agent Information carried the basic information about device ID, MAC Address, serial number, etc. Platform Information carried the OS version, BIOS Version, CPU name, etc. Custom Information carried the product name, manufacturer name and device type.

Configuration:

The Structure defined the Agent and Server Configuration. Agent Configuration configured the agent executing mode. Server Configuration configured the server IP, listen port, login ID and password.

1.1.2 Core Management

The Core Management layer in charge of load and manage the Handlers, bridge between SAClient and Handlers, handle the commands of agent control.

Core Manager:

The core module, dynamic loaded by SAClient, takes charge of functional modules (Handler Loader, General Command and Keep Alive) integration and interact with SAClient to communicate with RMM Server.

Handler Loader:

This module will read an XML file (module_config.xml) to get the information of handlers, including the name and path of handlers and how many handlers to load. This module also manage those loaded handlers.

General Command:

This module handles all the commands to control the Agent, such as agent update, rename host name. This module also handles the commands that need to delivered to all handlers, such as "GetCapability" to collect the capability of each handler, "AtuoReportStart" and "AutoReportStop" to control the sensor data report for every handler.

Keep Alive:

This module is the software watchdog to keep the threads of Handlers alive by calling Handler_Get_Status.

2   Modbus Handler

Modbus handler is one of handlers in WISE Agent framework and is used to collect devices' data based on Modbus TCP protocol. The library used in Modbus handler is libmodbus [2], which is a 3rd party free library to send/receive data according to the Modbus protocol. This library is written in C and supports RTU (serial) and TCP (Ethernet) communication. The license of libmodbus is LGPL v2.1v.

2.1   Working Flow

The flow of WISE Agent communicating with a device based on Modbus TCP by Modbus handler is described as follows. The WISE Agent uses the Modbus handler as the interface to handle all the communication with the device based on Modbus TCP protocol.

Steps of Modbus TCP protocol:

Step1 Modbus handler reads in INI file and is ready to set up connection with device according to the configuration in the INI file.

Step2 Modbus handler establishes a Modbus connection

Step3 WISE Agent communicates with the device.

Step4 If you no more need the communication, WISE Agent can disconnect with the device.
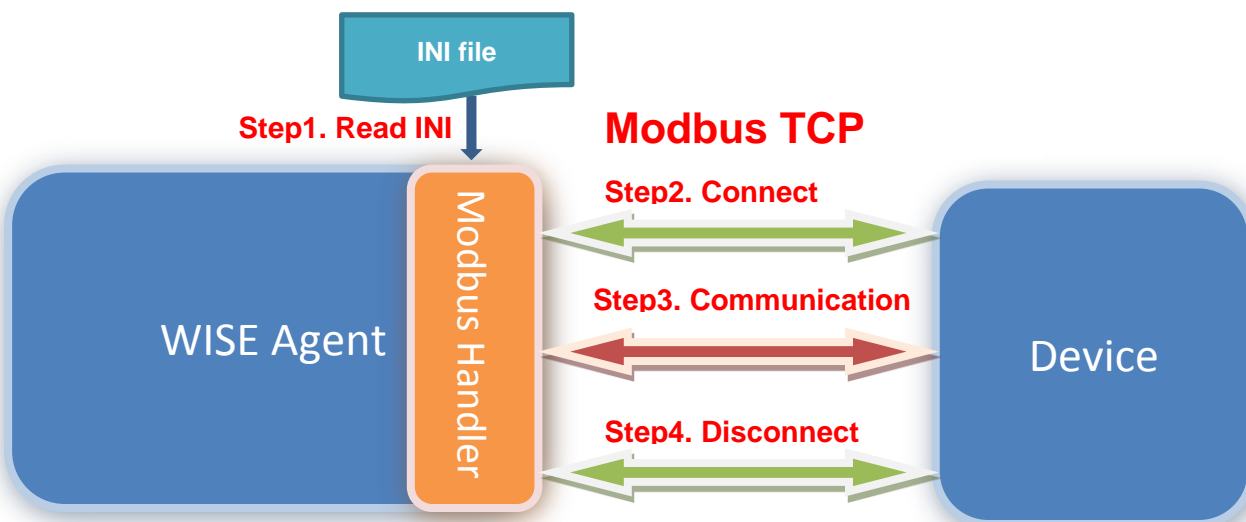
2.2 Functions and Format

Modbus handler has two major functions: Send Capability and Report Data. The Advantech sensor data format refers to IPSO Alliance Guide [3] and uses Sensor Markup Language (SENML) [4] to define media types for representing sensor measurements and device parameters in the SenML. Representations are defined in JavaScript Object Notation (JSON) [5].

2.2.1 Send Capability

Modbus handler gets handler Capability and sends it to server according to an INI file configuration.

The following example showing format of reporting data is based on conditions below.

1) **Device**: WISE-4012E

2) **IP**: 192.168.1.1

3) **Port**: 502

4) **Connection**: true

5) **Item List:**

    1> **Platform**:

        1-Name

        2-ClientIP

        3-ClientPort

        4-Connection

    2> **Two Discrete Inputs**:

        1- Discrete Inputs: Two switches

    3> **Two Input Registers**:

        1- Input Registers: Two voltage inputs

    4> **Two Coils**:

        1- Coils: Two LEDs

```
{
  "Modbus_Handler": {
    "Platform": {
      "bn": "Platform",
      "e": [
        {
          "n": "Name",
```

```json
          "sv": "WISE-4012E",
          "asm": "r"
        },
        {
          "n": "ClientIP",
          "sv": "192.168.1.1",
          "asm": "r"
        },
        {
          "n": "ClientPort",
          "sv": "502",
          "asm": "r"
        },
        {
          "n": "Connection",
          "bv": true,
          "asm": "r"
        }
      ]
    },
    "Discrete Inputs": {
      "bn": "Discrete Inputs",
      "e": [
        {
          "n": "Switch0",
          "bv": true,
          "asm": "r"
        },
        {
          "n": "Switch1",
          "bv": true,
          "asm": "r"
        }
      ]
    },
    "Coils": {
      "bn": "Coils",
      "e": [
        {
```

```
              "n": "LED0",
              "bv": true,
              "asm": "rw"
          },
          {
              "n": "LED1",
              "bv": true,
              "asm": "rw"
          }
        ]
    },
    "Input Registers": {
        "bn": "Input Registers",
        "e": [
          {
              "n": "Voltage0",
              "v": 4.645000,
              "max": 5,
              "min": 0,
              "asm": "r",
              "u": "V"
          },
          {
              "n": "Voltage1",
              "v": 4.659000,
              "max": 5,
              "min": 0,
              "asm": "r",
              "u": "V"
          }
        ]
    }
  }
}
```

The details of semantics in the JSON are described in **Appendix B**.

2.2.2 Report Data

Modbus handler gets devices' information repeatedly and reports these data to server based on specific item list or report setting. And the data are uploaded according to an INI file configuration.

The following example showing format of reporting data is based on conditions below.

1) **Device**: WISE-4012E

2) **IP**: 192.168.1.1

3) **Port**: 502

4) **Connection**: true

5) **Item List:**

    1> **Platform**:

        1-Name

        2-ClientIP

        3-ClientPort

        4-Connection

    2> **Two Discrete Inputs**:

        1- Discrete Inputs: Two switches

    3> **Two Input Registers**:

        1- Input Registers: Two voltage inputs

    4> **Two Coils**:

        1- Coils: Two LEDs

```
{
  "Modbus_Handler": {
    "Platform": {
      "bn": "Platform",
      "e": [
        {
          "n": "Name",
          "sv": "WISE-4012E"
        },
        {
          "n": "ClientIP",
          "sv": "192.168.1.1"
        },
        {
          "n": "ClientPort",
          "sv": "502"
        },
```

```json
        {
          "n": "Connection",
          "bv": true
        }
      ]
    },
    "Discrete Inputs": {
      "bn": "Discrete Inputs",
      "e": [
        {
          "n": "Switch0",
          "bv": true
        },
        {
          "n": "Switch1",
          "bv": true
        }
      ]
    },
    "Coils": {
      "bn": "Coils",
      "e": [
        {
          "n": "LED0",
          "bv": true
        },
        {
          "n": "LED1",
          "bv": true
        }
      ]
    },
    "Input Registers": {
      "bn": "Input Registers",
      "e": [
        {
          "n": "Voltage0",
          "v": 4.634000
        },
```

```
        {
          "n": "Voltage1",
          "v": 4.669000
        }
      ]
    }
  }
}
```

The item list can be abbreviated as the item type or hander name, e.g., /Modbus_Handler/Discrete Inputs/ means all Discrete Inputs, /Modbus_Handler/ means all items. The details of semantics in the JSON are described in **Appendix B**.

2.2.3 Upload Data

Modbus handler gets devices' information repeatedly and reports these data to server based on specific item list and two time parameters interval and timeout for upload setting. And the data are uploaded according to an INI file configuration.

The following example showing format of uploading data is based on conditions below.

1) **Device**: WISE-4012E

2) **IP**: 192.168.1.1

3) **Port**: 502

4) **Connection**: true

5) **Item List:**

    1> **Platform**:

        1- Name

        2- ClientIP

        3- ClientPort

        4- Connection

    2> **Two Discrete Inputs**:

        1- Discrete Inputs: Two switches

    3> **Two Input Registers**:

        1- Input Registers: Two voltage inputs

    4> **Two Coils**:

        1- Coils: Two LEDs

```
{
  "Modbus_Handler":{
    "Platform":{
      "bn":"Platform",
```

```
  "e":[
    {
      "n":"Name",
      "sv":"WISE-4012E"
    },
    {
      "n":"ClientIP",
      "sv":"192.168.1.1"
    },
    {
      "n":"ClientPort",
      "sv":"502"
    },
    {
      "n":"Connection",
      "bv":true
    }
  ]
},
"Discrete Inputs":{
  "bn":"Discrete Inputs",
  "e":[
    {
      "n":"Switch0",
      "bv":true
    },
    {
      "n":"Switch1",
      "bv":false
    }
  ]
},
"Coils":{
  "bn":"Coils",
  "e":[
    {
      "n":"LED0",
      "bv":true
    },
```

```
        {
          "n":"LED1",
          "bv":true
        }
      ]
    },
    "Input Registers":{
      "bn":"Input Registers",
      "e":[
        {
          "n":"Voltage0",
          "v":4.664000
        },
        {
          "n":"Voltage1",
          "v":4.677000
        }
      ]
    }
  }
}
```

The item list can be abbreviated as the item type or hander name, e.g., /Modbus_Handler/Discrete Inputs/ means all Discrete Inputs, /Modbus_Handler/ means all items. The details of semantics in the JSON are described in **Appendix B**.

2.2.4 Get Data

Modbus handler gets devices' information and reports these data with status codes to server based on specific sensor list. And the data are uploaded according to an INI file configuration.

The following example showing format of getting data is based on conditions below.

1) **Device**: WISE-4012E

2) **IP**: 192.168.1.1

3) **Port**: 502

4) **Connection**: true

5) **Sensor List:**

    1> **Two Discrete Inputs**:

        1- Discrete Inputs: Two switches

    2> **Two Input Registers**:

1- Input Registers: Two voltage inputs

```
{
  "sessionID":"3B579DA6D9E804D10316E5285586304F",
  "sensorInfoList":{
    "e":[
      {
        "n":"Modbus_Handler/Input Registers/Voltage1",
        "v":4.675000,
        "StatusCode":200
      },
      {
        "n":"Modbus_Handler/Input Registers/Voltage0",
        "v":4.653000,
        "StatusCode":200
      },
      {
        "n":"Modbus_Handler/Discrete Inputs/Switch1",
        "bv":false,
        "StatusCode":200
      },
      {
        "n":"Modbus_Handler/Discrete Inputs/Switch0",
        "bv":true,
        "StatusCode":200
      }
    ]
  }
}
```

The details of semantics in the JSON and StatusCode are described in **Appendix B** and **Appendix C**.

2.2.5 Set Data

Modbus handler sets a specific value to a sensor on the device and the status of setting is reported.

The following example showing format of setting data is based on conditions below.

1) **Device**: WISE-4012E

2) **IP**: 192.168.1.1

3) **Port**: 502

4) **Connection**: true

5) **Sensor:** Coils: LED0

```
{
  "sessionID":"3B579DA6D9E804D10316E5285586304F",
  "sensorInfoList":{
    "e":[
      {
        "n":"Modbus_Handler/Coils/LED0",
        "sv":"Success",
        "StatusCode":200
      }
    ]
  }
}
```

The details of semantics in the JSON and StatusCode are described in **Appendix B** and **Appendix C**.

3 Configuration

To set the configuration of which and where the values lie for reading, a Modbus Handler takes an INI file as input. Importantly, the INI file should have the file name same as the handler name, which means different handlers take different INI files as their inputs. The INI file format [6] is an informal standard for configuration files for some platforms and is a text file with a basic structure composed of sections, properties and values.

The following shows an example configuration of WISE-4012E. WISE-4012 has two Discrete Inputs for two switches, two Coils for two LEDs and two Input Registers for two voltage adjusters. The addresses of these values are given below.

WISE-4012E Wireless Modbus Mapping Table

Address 0X	Channel	Description	Attribute
10001	0	IB Value	Read
10002	1		Read
00017	0	B Value	R/W
00018	1		R/W
30001	0	IR Value (Value Range: 0~10000, Value Unit: mV)	Read
30002	1		Read
30003	Average Channel 0~1		Read

INI File for WISE-4012E

```
[Platform]
Name=WISE-4012E
ClientIP=192.168.1.1
ClientPort=502


[Discrete Inputs]
numberOfIB=2
IB0=0,Switch0
IB1=1,Switch1


[Coils]
numberOfB=2
B0=16,LED0
B1=17,LED1


[Input Registers]
numberOfIR=2
IR0=0,Voltage0,0,5,0.001,V
IR1=1,Voltage1,0,5,0.001,V
```

The details of WISE-4012E in INI file:

First Section:

     [Platform] describes a device name, IP and Port used by the device.

Second Section:

     [Discrete Inputs] describes the number of discrete inputs and a offset address of a value which is as "IB0=1", meaning the IB0 value is at offset 1 then the name of the discrete input follows.

Third Section:

     [Coils] describes the number of coils and a offset address of a value which is as "B0=16", meaning the B0 value is at offset 16 then the name of the coil follows.

Forth Section:

     [Input Registers] describes the number of input registers and a offset address of a value which is as "IR1=1", meaning the IR1 value is at offset 1 then the name of the input register follows. The range, precision and unit of the value are given too, as "0,5,0.001,V" means the minimum value is 0, the maximum value is 5,

the precision is 0.001 and the unit is voltage.

If the device has other attributes like holding registers, you should add another section named as [Holding Registers]. And if the device has less attributes, you should remove the corresponding section too. To set an appropriate configuration, please refer to the specification of the device.

4 Appendix

4.1 Appendix A

Modbus [1] is a serial communications protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs). Modbus is simple and robust to implement. And it is now commonly available means of connecting industrial electronic devices.

The versions of Modbus protocol exist for serial port and for Ethernet and other protocol that support the Internet protocol suite. And the Modbus Handler is based on one of these protocols, Modbus TCP/IP or Modbus TCP, which is for Ethernet.

4.1.1 Modbus TCP/IP

Modbus TCP/IP is a Modbus variant used for communications over TCP/IP protocols, and the port used is 502. The Modbus TCP/IP does not have a checksum calculation as lower layers already provide checksum protection.

4.1.2 Frame Format

A Modbus frame is composed of an Application Data Unit (ADU) which encloses a Protocol Data Unit (PDU.)

- ADU = Address + PDU + Error check
- PDU = Function code + Data
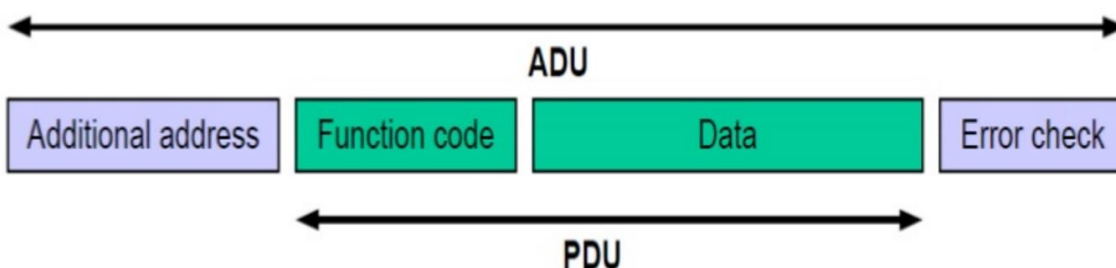
However, the Modbus TCP/IP does not have the Error check part because it already has the checksum protection mechanism.

Modbus TCP frame format

Name	Length (bytes)	Function
Transaction identifier	2	For synchronization between messages of server & client
Protocol identifier	2	Zero for Modbus/TCP
Length field	2	Number of remaining bytes in this frame
Unit identifier	1	Slave address (255 if not used)
Function code	1	Function codes as in other variants
Data bytes	n	Data as response or commands

Unit identifier is used with Modbus/TCP devices that are composites of several Modbus devices, e.g. on Modbus/TCP to Modbus RTU gateways. In such case, the unit identifier tells the Slave Address of the device behind the gateway. Natively Modbus/TCP-capable devices usually ignore the Unit Identifier.

4.1.3 Entities

Modbus has four entities for processing different kinds of input/output data. Each of entities has its own address range and function as below.

(Entity address is a 16-bit value in the data part of the Modbus frame and its range goes from 0 to 65535.)

Entity	Address Range	Type	Mode	Function
Coils	00001~09999	Single bit	Read/Write	Read/Write discrete output or coils
Discrete Input	10001~19999	Single bit	Read	Read discrete inputs
Input Registers	30001~39999	16-bit value	Read	Read 16-bit input registers such as analog inputs
Holding Registers	40001~49999	16-bit value	Read/Write	Read/Write 16-bit holding registers such as I/O

4.1.4 Function codes

The various reading, writing and other operations are categorized as follows. The most primitive reads and writes are shown in bold. Requests and responses follow frame formats described in the following sections. This Chapter gives details of data formats of most used function codes.

Modbus function codes

Function type			Function name	Function code
Data Access	Bit access	Physical Discrete Inputs	**Read Discrete Inputs**	2
		Internal Bits or Physical Coils	**Read Coils**	1
			Write Single Coil	5
			Write Multiple Coils	15
	16-bit access	Physical Input Registers	**Read Input Registers**	4
		Internal Registers or Physical Output Registers	**Read Multiple Holding Registers**	3
			Write Single Holding Register	6
			Write Multiple Holding Registers	16
			Read/Write Multiple Registers	23
			Mask Write Register	22
			Read FIFO Queue	24
	File Record Access		Read File Record	20
			Write File Record	21
Diagnostics			Read Exception Status	7
			Diagnostic	8
			Get Com Event Counter	11
			Get Com Event Log	12
			Report Slave ID	17
			Read Device Identification	43
Other			Encapsulated Interface Transport	43

4.1.4.1 Function code 1 (read coils) and function code 2 (read discrete inputs)

Request	Address of first coil/discrete input to read (16 bits)
	Number of coils/discrete inputs to read (16 bits)
Normal response	Number of bytes of coil/discrete input values to follow (8-bit)
	Coil/discrete input values (8 coils/discrete inputs per byte)

Value of each coil/discrete input is binary (0 for off, 1 for on.)

4.1.4.2 Function code 5 (force/write single coil)

Request	Address of coil (16-bit)
	Value to force/write: 0 for off and 65,280 (FF00 in hexadecimal) for on
Normal response	Address of coil (16-bit)
	Value to force/write: 0 for off and 65,280 (FF00 in hexadecimal) for on

4.1.4.3 Function code 15 (force/write multiple coils)

Request	Address of first coil to force/write (16-bit)
	Number of coils to force/write (16-bit)
	Number of bytes of coil values to follow (8-bit)
	Coil values (8 coil values per byte)
Normal response	Address of first coil (16-bit)
	number of coils (16-bit)

Value of each coil is binary (0 for off, 1 for on).

4.1.4.4 Function code 4 (read input registers) and function code 3 (read holding registers)

Request	Address of first register to read (16-bit)
	Number of registers to read (16-bit)
Normal response	Number of bytes of register values to follow (8-bit)
	Register values (16 bits per register)

4.1.4.5 Function code 6 (preset/write single holding register)

Request	Address of holding register to preset/write (16-bit)
	New value of the holding register (16-bit)
Normal response	Address of holding register to preset/write (16-bit)
	New value of the holding register (16-bit)

4.1.4.6 Function code 16 (preset/write multiple holding registers)

Request	Address of first holding register to preset/write (16-bit)
	Number of holding registers to preset/write (16-bit)
	Number of bytes of register values to follow (8-bit)
	New values of holding registers (16 bits per register)
Normal response	Address of first preset/written holding register (16-bit)
	number of preset/written holding registers (16-bit)

4.1.5 Exception responses

For a normal response, slave repeats the function code. If a slave want to report an error, it will reply with the requested function code plus 128 (3 becomes 131), and will only include one byte of data, known as the exception code.

Main Modbus exception codes

Code	Text	Details
1	Illegal Function	Function code received in the query is not recognized or allowed by slave
2	Illegal Data Address	Data address of some or all the required entities are not allowed or do not exist in slave
3	Illegal Data Value	Value is not accepted by slave
4	Slave Device Failure	Unrecoverable error occurred while slave was attempting to perform requested action
5	Acknowledge	Slave has accepted request and is processing it, but a long duration of time is required. This response is returned to prevent a timeout error from occurring in the master. Master can next issue a *Poll Program Complete* message to determine if processing is completed
6	Slave Device Busy	Slave is engaged in processing a long-duration command. Master should retry later
7	Negative Acknowledge	Slave cannot perform the programming functions. Master should request diagnostic or error information from slave
8	Memory Parity Error	Slave detected a parity error in memory. Master can retry the request, but service may be required on the slave device
10	Gateway Path Unavailable	Specialized for Modbus gateways. Indicates a misconfigured gateway
11	Gateway Target Device Failed to Respond	Specialized for Modbus gateways. Sent when slave fails to respond

4.2   Appendix B

Media Types for Sensor Markup Language (SENML)

Semantics

SenML	JSON	Type	Description
Base Name	bn	String	This is a string that is prepended to the names found in the entries
Base Time	bt	Integer	A base time that is added to the time found in an entry
Base Units	bu	String	A base unit that is assumed for all entries, unless otherwise indicated
Version	ver	Number	Version number of media type format
Measurement or Parameters	e	Array	Array of values for sensor measurements or other generic parameters

Name	n	String	Name of the sensor or parameter
Units	u	String	Units for a measurement value
Value	v	Float	Value of the entry
String Value	sv	String	
Boolean Value	bv	Boolean	
Value Sum	s	Float	Integrated sum of the values over time
Time	t	Integer	Time when value was recorded
Update Time	ut	Integer	Update time. A time in seconds that represents the maximum time before this sensor will provide an updated reading for a measurement.

The Data Type of Sensor Data Type

Data Type (type)	SenML Field
b (boolean)	bv
s (string)	s
e (enum)	e
i (integer)	v
d (decimal)	v
h(hexadecimal)	s
o(octet-stream)	s

Advantech Sensor Semantics

SenML	JSON	Type	Description
Min Range Value	min	Float	The minimum value that can be measured by the sensor
Max Range Value	max	Float	The maximum value that can be measured by the sensor
Access Mode	asm	String	The access mode of the resource. Ex: read (r), write (w), read/write (rw)
Standard Format	st	String	The sensor format is which standard format
Health Status	Health	Integer	The health status of network or device. Range: -1 ~ 100 Good: > 80, Average: 60 ~ 80, Below Average: 40~60, Bad:0~40, -1: Off line or Fault

4.3   Appendix C

The status code is used to describe the status of setting or getting data in a session.

Status Code

StatusCode	Meaning
200	Success
404	Not Found
405	Read/Write Only
415	Format Error
416	Out Of Range
500	Fail
503	System Busy

5   Reference

[1] "Modbus," [online]. Available: https://en.wikipedia.org/wiki/Modbus.

[2] "libmodbus," [online]. Available: http://libmodbus.org/documentation/.

[3] "IPSO," [online]. Available: http://www.ipso-alliance.org/.

[4] "SENML," [online]. Available: https://datatracker.ietf.org/doc/draft-jennings-senml/.

[5] "JSON," [online]. Available: http://json.org/.

[6] "INI File," [online]. Available: https://en.wikipedia.org/wiki/INI_file.