

## **Command Module Modbus protocol**

This document is covered by confidentiality agreement and reproduction of this document is strictly prohibited unless express written permission is obtained from AirSense Technology Ltd.

In line with continuous product improvement AirSense Technology Ltd reserves the right to modify or update specifications without notice.

Stratos-HSSD, Stratos-Quadra, Stratos-Micra, SenseNET, FastLearn and ClassiFire are trademarks of AirSense Technology Ltd.

Copyright © 2007 AirSense Technology Ltd.

|  |                    |
|--|--------------------|
| <a href="#">Introduction.....</a>                        | <a href="#">3</a>  |
| <a href="#">Modbus PDUs.....</a>                         | <a href="#">4</a>  |
| <a href="#">03 (0x03) Read Holding Registers.....</a>    | <a href="#">4</a>  |
| <a href="#">06 (0x06) Write Single Register.....</a>     | <a href="#">5</a>  |
| <a href="#">MODBUS Exception Responses.....</a>          | <a href="#">6</a>  |
| <a href="#">MODBUS Exception Codes.....</a>              | <a href="#">7</a>  |
| <a href="#">Modbus serial line protocol.....</a>         | <a href="#">8</a>  |
| <a href="#">UART data format.....</a>                    | <a href="#">8</a>  |
| <a href="#">CRC Generation.....</a>                      | <a href="#">9</a>  |
| <a href="#">Register mapping Command Module 1.7.....</a> | <a href="#">10</a> |
| <a href="#">Status and fault register format.....</a>    | <a href="#">10</a> |
| <a href="#">Register mapping Command Module 1.8.....</a> | <a href="#">11</a> |
| <a href="#">Status and fault register format.....</a>    | <a href="#">11</a> |

## Introduction

In process control industries there is a requirement to monitor and control AirSense detectors using the widely used Modbus protocol and to this end Modbus has been added as an additional BMS protocol on the Command Modules second serial port.

The Modbus protocol implementation allows monitoring of individual alarm and fault status on all detectors and the Command Module. In addition it allows the resetting of the Command Module and attached detectors and of the Isolating of the Command Module.

To implement Modbus on the Command Module the following two functions are supported:

**03 Read Holding registers**

**06 Write single register**

Note that multiple register writes is not implemented due to the fact that normal operation will consist of monitoring with the occasional reset.

These functions allow monitoring of the command module and detectors alarm/fault status and configuration of the programmable functions. Resetting of the Command Module is achieved by writing to a designated reset register. Writing to a designated isolate register toggles the Isolate status of the Command Module.

Separate registers are used to hold the command module and detectors status. The format is shown in the 'Register mapping' section later in this document.

# Modbus PDUs

The following pages detail the **Protocol Data Unit (PDU)** which is the transmission format independent part of the Modbus protocol. These conform to level 7 of the OSI specification of client/server communications.

The Command Module supports two functions described on the following pages. Attempted use of any other function will generate an **ILLEGAL FUNCTION** exception.

## 03 (0x03) Read Holding Registers

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15 in the PDU.

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

### Request

|                       |         |                  |
|-----------------------|---------|------------------|
| Function code         | 1 Byte  | <b>0x03</b>      |
| Starting Address      | 2 Bytes | 0x0000 to 0xFFFF |
| Quantity of Registers | 2 Bytes | 1 to 125 (0x7D)  |

### Response

|                |                      |                |
|----------------|----------------------|----------------|
| Function code  | 1 Byte               | <b>0x03</b>    |
| Byte count     | 1 Byte               | 2 x <b>N</b> * |
| Register value | <b>N</b> * x 2 Bytes |                |

\***N** = Quantity of Registers

### Error

|                |        |                       |
|----------------|--------|-----------------------|
| Error code     | 1 Byte | <b>0x83</b>           |
| Exception code | 1 Byte | 01 or 02 or 03 or 04* |

\*See 'Modbus exception codes for more information

Here is an example of a request to read registers 108 – 110:

| Request             |           | Response                |           |
|---------------------|-----------|-------------------------|-----------|
| Field Name          | (Hex)     | Field Name              | (Hex)     |
| Function            | <b>03</b> | Function                | <b>03</b> |
| Starting Address Hi | <b>00</b> | Byte Count              | <b>06</b> |
| Starting Address Lo | <b>6B</b> | Register value Hi (108) | <b>02</b> |
| No. of Registers Hi | <b>00</b> | Register value Lo (108) | <b>2B</b> |
| No. of Registers Lo | <b>03</b> | Register value Hi (109) | <b>00</b> |
|                     |           | Register value Lo (109) | <b>00</b> |
|                     |           | Register value Hi (110) | <b>00</b> |
|                     |           | Register value Lo (110) | <b>64</b> |

The contents of register 108 are shown as the two byte values of 02 2B hex, or 555 decimal. The contents of registers 109–110 are 00 00 and 00 64 hex, or 0 and 100 decimal, respectively.

## 06 (0x06) Write Single Register

This function code is used to write a single holding register in a remote device. The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0 in the PDU. The normal response is an echo of the request, returned after the register contents have been written.

### Request

|                  |         |                  |
|------------------|---------|------------------|
| Function code    | 1 Byte  | <b>0x06</b>      |
| Register Address | 2 Bytes | 0x0000 to 0xFFFF |
| Register Value   | 2 Bytes | 0x0000 or 0xFFFF |

### Response

|                  |         |                  |
|------------------|---------|------------------|
| Function code    | 1 Byte  | <b>0x06</b>      |
| Register Address | 2 Bytes | 0x0000 to 0xFFFF |
| Register Value   | 2 Bytes | 0x0000 to 0xFFFF |

### Error

|                |        |                       |
|----------------|--------|-----------------------|
| Error code     | 1 Byte | <b>0x86</b>           |
| Exception code | 1 Byte | 01 or 02 or 03 or 04* |

\*See 'Modbus exception codes for more information

Here is an example of a request to write register 2 to 00 03 hex:

| Request             |           | Response            |           |
|---------------------|-----------|---------------------|-----------|
| Field Name          | (Hex)     | Field Name          | (Hex)     |
| Function            | <b>06</b> | Function            | <b>06</b> |
| Register Address Hi | <b>00</b> | Register Address Hi | <b>00</b> |
| Register Address Lo | <b>01</b> | Register Address Lo | <b>01</b> |
| Register Value Hi   | <b>00</b> | Register Value Hi   | <b>00</b> |
| Register Value Lo   | <b>03</b> | Register Value Lo   | <b>03</b> |

## MODBUS Exception Responses

When a client device sends a request to a server device it expects a normal response. One of four possible events can occur from the master's query:

If the server device receives the request without a communication error, and can handle the query normally, it returns a normal response.

If the server does not receive the request due to a communication error, no response is returned. The client program will eventually process a timeout condition for the request.

If the server receives the request, but detects a communication error (parity, LRC, CRC, ...), no response is returned. The client program will eventually process a timeout condition for the request.

If the server receives the request without a communication error, but cannot handle it (for example, if the request is to read a non-existent output or register), the server will return an exception response informing the client of the nature of the error.

The exception response message has two fields that differentiate it from a normal response:

**Function Code Field:** In a normal response, the server echoes the function code of the original request in the function code field of the response. All function codes have a most-significant bit (MSB) of 0 (their values are all below 80 hexadecimal). In an exception response, the server sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.

With the function code's MSB set, the client's application program can recognize the exception response and can examine the data field for the exception code.

**Data Field:** In a normal response, the server may return data or statistics in the data field (any information that was requested in the request). In an exception response, the server returns an exception code in the data field. This defines the server condition that caused the exception.

Example of a client request and server exception response

| Request                |       | Response       |       |
|------------------------|-------|----------------|-------|
| Field Name             | (Hex) | Field Name     | (Hex) |
| Function               | 03    | Function       | 83    |
| Register Address Hi    | 04    | Exception code | 02    |
| Register Address Lo    | A1    |                |       |
| Quantity of Outputs Hi | 00    |                |       |
| Quantity of Outputs Lo | 01    |                |       |

In this example, the client addresses a request to server device. The function code (03) is for a Read Holding Registers operation. It requests the status of the output at address 1245 (04A1 hex). Note that only that one register is specified to be read as quantity of outputs is 1 (0001 hex).

If the output address is non-existent in the server device, the server will return the exception response with the exception code shown (02). This specifies an illegal data address for the slave.

| MODBUS Exception Codes |   |   |
|------------------------|---|---|
| Code                   | Name                                    | Meaning   |
| 01                     | ILLEGAL FUNCTION                        | The function code received in the query is not an allowable action for the server (or slave). This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server (or slave) is in the wrong state to process a request of this type, for example because it is not configured and is being asked to return register values.  |
| 02                     | ILLEGAL DATA ADDRESS                    | The data address received in the query is not an allowable address for the server (or slave). More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, a request with offset 96 and length 4 would succeed, a request with offset 96 and length 5 will generate exception 02.  |
| 03                     | ILLEGAL DATA VALUE                      | A value contained in the query data field is not an allowable value for server (or slave). This indicates a fault in the structure of the remainder of a complex request, such as that the implied length is incorrect. It specifically does NOT mean that a data item submitted for storage in a register has a value outside the expectation of the application program, since the MODBUS protocol is unaware of the significance of any particular value of any particular register. |
| 04                     | SLAVE DEVICE FAILURE                    | An unrecoverable error occurred while the server (or slave) was attempting to perform the requested action.   |
| 05                     | ACKNOWLEDGE                             | Specialized use in conjunction with programming commands. The server (or slave) has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to prevent a timeout error from occurring in the client (or master). The client (or master) can next issue a Poll Program Complete message to determine if processing is completed.   |
| 06                     | SLAVE DEVICE BUSY                       | Specialized use in conjunction with programming commands. The server (or slave) is engaged in processing a long-duration program command. The client (or master) should retransmit the message later when the server (or slave) is free.  |
| 08                     | MEMORY PARITY ERROR                     | Specialized use in conjunction with function codes 20 and 21 and reference type 6, to indicate that the extended file area failed to pass a consistency check. The server (or slave) attempted to read record file, but detected a parity error in the memory. The client (or master) can retry the request, but service may be required on the server (or slave) device.   |
| 0A                     | GATEWAY PATH UNAVAILABLE                | Specialized use in conjunction with gateways, indicates that the gateway was unable to allocate an internal communication path from the input port to the output port for processing the request. MODBUS Application Protocol Specification V1.1a Modbus-IDA Usually means that the gateway is incorrectly configured or overloaded.  |
| 0B                     | GATEWAY TARGET DEVICE FAILED TO RESPOND | Specialized use in conjunction with gateways, indicates that no response was obtained from the target device. Usually means that the device is not present on the network.  |

## ***Modbus serial line protocol***

The serial line data format is shown below:

| <b>Address field</b> | <b>PDU</b>             | <b>CRC</b> |
|----------------------|------------------------|------------|
| 1 byte               | Message dependent size | 2 bytes    |

Valid slave nodes addresses are in the range of 0 – 247 decimal. The individual slave devices are assigned addresses in the range of 1 – 247. A master addresses a slave by placing the slave address in the address field of the message. When the slave returns its response, it places its own address in the response address field to let the master know which slave is responding.

For the command module the address is currently fixed as 1. Address 0 is a broadcast address and any slave must receive data sent to this address.

RTU mode is used exclusively by the command module. In RTU mode all data is sent in binary format.

### **UART data format**

The UART data format is:

**Coding System:** 8–bit binary

**Bits per Byte:** 1 start bit

8 data bits, least significant bit sent first

no parity

1 stop bit

## **CRC Generation**

The Cyclical Redundancy Checking (CRC) field is two bytes, containing a 16-bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The device that receives recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16-bit register to all 1's. Then a process begins of applying successive 8-bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits and the parity bit, do not apply to the CRC.

During generation of the CRC, each 8-bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next 8-bit character is exclusive Ored with the register's current value, and the process repeats for eight more shifts as described above. The final content of the register, after all the characters of the message have been applied, is the CRC value.

A procedure for generating a CRC is:

1. Load a 16-bit register with FFFF hex (all 1's). Call this the CRC register.
2. Exclusive OR the first 8-bit byte of the message with the low-order byte of the 16-bit CRC register, putting the result in the CRC register.
3. Shift the CRC register one bit to the right (toward the LSB), zero-filling the MSB. Extract and examine the LSB.
4. (If the LSB was 0): Repeat Step 3 (another shift). (If the LSB was 1): Exclusive OR the CRC register with the polynomial value 0xA001 (1010 0000 0000 0001).
5. Repeat Steps 3 and 4 until 8 shifts have been performed. When this is done, a complete 8-bit byte will have been processed.
6. Repeat Steps 2 through 5 for the next 8-bit byte of the message. Continue doing this until all bytes have been processed.
7. The final content of the CRC register is the CRC value.
8. When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

### **Placing the CRC into the Message**

When the 16-bit CRC (two 8-bit bytes) is transmitted in the message, the low-order byte will be transmitted first, followed by the high-order byte.

## Register mapping Command Module 1.7

The following tables detail the register mapping to access various features of the unit. See the bus protocol reference for a description of the programmable functions.

| Register | Name            | Use   |
|----------|-----------------|---|
| 1        | STATUS_CM       | Command Module status   |
| 2        | STATUS_DET1     | Detector 1 status   |
| ..       | ..              | ..  |
| 128      | STATUS_DET127   | Detector 127 status   |
| 129      | FAULTS_CM       | Command Module faults   |
| 130      | FAULTS_DET1     | Detector 1 faults   |
| 256      | FAULTS_DET127   | Detector 127 faults   |
| -        | <i>unused</i>   | <i>unused</i>   |
| 300      | FN_1            | Programmable function 1   |
| ..       | ..              | ..  |
| 477      | FN_178          | Programmable function 178   |
| -        | <i>unused</i>   | <i>unused</i>   |
| 600      | CONTROL_RESET   | Reset – write any value to reset alarms or faults                                       |
| 601      | CONTROL_ISOLATE | Isolate – write any value to toggle isolate status. Read returns non-zero when isolated |

Accessing registers marked as unused or registers outside of the range of this table or registers marked *unused* will result in a ILLEGAL DATA VALUE exception being generated.

### Status and fault register format

#### Detector Status.

| bit | use                |
|-----|--------------------|
| 1   | General fault flag |
| 2   | Aux Flag           |
| 3   | Pre Alarm flag     |
| 4   | Fire 1 flag        |
| 5   | Fire 2 flag        |
| 6   | Input 1            |
| 7   | Input 2            |
| 8   | <i>Reserved</i>    |

#### Command Module Status.

| bit | use                |
|-----|--------------------|
| 1   | General fault flag |
| 2   | Aux Flag           |
| 3   | Pre Alarm flag     |
| 4   | Fire 1 flag        |
| 5   | Fire 2 flag        |
| 6   | Input 1            |
| 7   | Input 2            |
| 8   | Loop break         |

#### Detector faults.

| bit | use             |
|-----|-----------------|
| 1   | Low flow*       |
| 2   | High flow*      |
| 3   | Head fault      |
| 4   | Mains fault     |
| 5   | Battery fault   |
| 6   | Isolated        |
| 7   | Separator fault |
| 8   | <i>Reserved</i> |

#### Command Module faults.

| bit | use             |
|-----|-----------------|
| 1   | <i>Reserved</i> |
| 2   | <i>Reserved</i> |
| 3   | Head fault      |
| 4   | Mains fault     |
| 5   | Battery fault   |
| 6   | Isolated        |
| 7   | <i>Reserved</i> |
| 8   | Bus loop break  |

\*Flow sensor fail is signalled by a simultaneous flow low and flow high fault.

## Register mapping Command Module 1.8

The following tables detail the register mapping to access various features of the unit. See the bus protocol reference for a description of the programmable functions.

| Register | Name            | Use   |
|----------|-----------------|---|
| 1        | STATUS_CM       | Command Module status   |
| 2        | STATUS_DET1     | Detector 1 status   |
| ..       | ..              | ..  |
| 128      | STATUS_DET127   | Detector 127 status   |
| 129      | FAULTS_CM       | Command Module faults   |
| 130      | FAULTS_DET1     | Detector 1 faults   |
| 256      | FAULTS_DET127   | Detector 127 faults   |
| -        | <i>unused</i>   | <i>unused</i>   |
| 300      | FN_1            | Programmable function 1   |
| ..       | ..              | ..  |
| 477      | FN_178          | Programmable function 178   |
| -        | <i>unused</i>   | <i>unused</i>   |
| 600      | CONTROL_RESET   | Reset – write any value to reset alarms or faults                                       |
| 601      | CONTROL_ISOLATE | Isolate – write any value to toggle isolate status. Read returns non-zero when isolated |
| -        | <i>unused</i>   | <i>unused</i>   |
| 701      | LEVEL_DET1      | Detector level detector 1   |
| 827      | LEVEL_DET127    | Detector level detector 127   |

Accessing registers marked as unused or registers outside of the range of this table or registers marked *unused* will result in a ILLEGAL DATA VALUE exception being generated.

Detector level information should not be used to generate alarms directly due to the relative scaling technology used by AirSense detectors. Level information is only valid when detectors are not signaling fault.

The value returned in this function varies between 0 and 255 and must be divided by 2.55 to get the detector percentage output as seen on the histogram viewer.

### Status and fault register format

#### Detector Status.

| bit | use                |
|-----|--------------------|
| 1   | General fault flag |
| 2   | Aux Flag           |
| 3   | Pre Alarm flag     |
| 4   | Fire 1 flag        |
| 5   | Fire 2 flag        |
| 6   | Input 1            |
| 7   | Input 2            |
| 8   | <i>Reserved</i>    |

#### Command Module Status.

| bit | use                |
|-----|--------------------|
| 1   | General fault flag |
| 2   | Aux Flag           |
| 3   | Pre Alarm flag     |
| 4   | Fire 1 flag        |
| 5   | Fire 2 flag        |
| 6   | Input 1            |
| 7   | Input 2            |
| 8   | Loop break         |

#### Detector faults.

| bit | use             |
|-----|-----------------|
| 1   | Low flow*       |
| 2   | High flow*      |
| 3   | Head fault      |
| 4   | Mains fault     |
| 5   | Battery fault   |
| 6   | Isolated        |
| 7   | Separator fault |
| 8   | <i>Reserved</i> |

#### Command Module faults.

| bit | use             |
|-----|-----------------|
| 1   | <i>Reserved</i> |
| 2   | <i>Reserved</i> |
| 3   | Head fault      |
| 4   | Mains fault     |
| 5   | Battery fault   |
| 6   | Isolated        |
| 7   | <i>Reserved</i> |
| 8   | Bus loop break  |

\*Flow sensor fail is signalled by a simultaneous flow low and flow high fault.