PROGRAMMING MANUAL

SR300-D1 | SR200-D1 | SR100-D1

Industrial pyranometers

Hukseflux
Thermal Sensors

Cautionary statements

Cautionary statements are subdivided into four categories: danger, warning, caution and notice according to the severity of the risk.

DANGER

Failure to comply with a danger statement will lead to death or serious physical injuries.

WARNING

Failure to comply with a warning statement may lead to risk of death or serious physical injuries.

CAUTION

Failure to comply with a caution statement may lead to risk of minor or moderate physical injuries.

NOTICE

Failure to comply with a notice may lead to damage to equipment or may compromise reliable operation of the instrument.

Contents

List of symbols

Quantities	Symbol	Unit
-	-	-

Subscripts

-

Acronyms

CRC	Cyclic Redundancy Check
LSW	Least-Significant Word
MSW	Most-Significant Word
PDU	Protocol Data Unit
SCADA	Supervisory Control And Data Acquisition
TIA	Telecommunications Industry Association
UART	Universal Asynchronous Receiver-Transmitter

Introduction

This document describes the specifics of the Modbus RTU interface of Hukseflux industrial pyranometers. This includes supported Modbus function codes, data types and behaviour specific to Hukseflux industrial series pyranometers. General information on the Modbus RTU protocol can be found in the “Modbus Protocol Specification” and the “Modbus Serial Line Protocol and Implementation Guide” found on http://www.modbus.org.

The registers of the individual instruments are listed in the respective instrument register lists.

1 Modbus over RS-485

Hukseflux Modbus instruments support communication over an RS-485 (formally TIA-485-A) network. Information of the implementation of Modbus and other relevant knowledge to support the use of a Hukseflux Modbus instrument is given in the following paragraphs.

1.1 Serial communication settings

Hukseflux Modbus instruments support a range of serial communication settings, as listed below.

SUPPORTED SERIAL COMMUNICATION SETTINGS	
Setting	**Valid configuration**
Baud rate	9600 – 115000, configurable in steps of 100
Parity bit	Even (E), odd (O), none (N)
Number of stop bits	1, 2

Configuration is denoted as `<number of data bits><parity bit><number of stop bits>`. For instance `8N2`, which means 8 data bits, no parity bit, 2 stop bits.

The default factory configuration of the instrument is 19200 baud, `8E1`, which means 8 data bits are used, an even number of ones in the data sets the parity bit to 1, and a single stop bit is used.

Baud rates are settable in steps of 100, but it is strongly recommended to use one of the standard baud rate settings of 9600, 19200, 38400 or 115000 to avoid misconfigured networks.

An example of a valid UART frame for sending one byte is given below. Note that the bits appended with an ‵ are optional bits; with parity set to none, or stop bits set to 1, those respective bits will not be included in the UART frame.

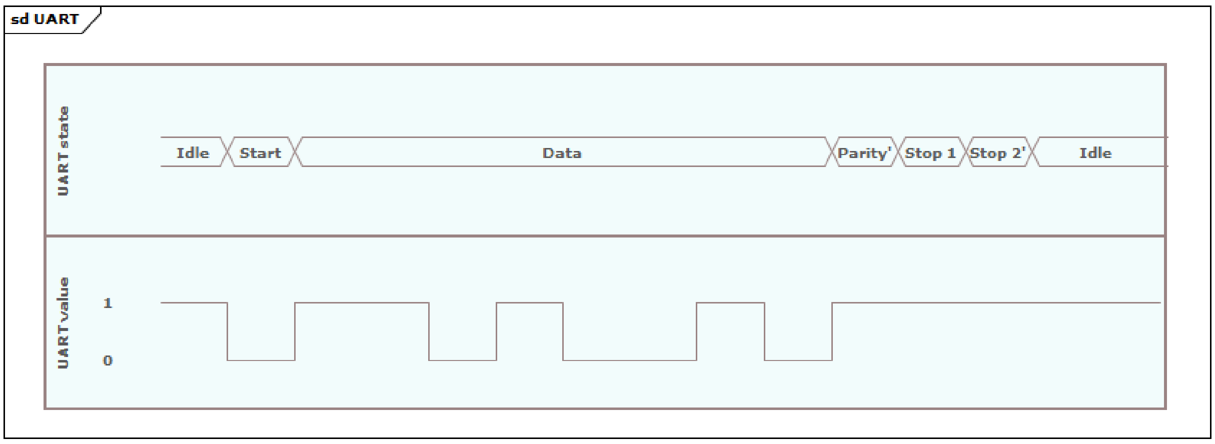

Figure 1.1.1: *An example UART frame.*

As can be seen, the start bit (always 0) signals the start of the frame, followed by eight data bits (0xD2 in this case). Parity is set to even, so the value is 1 for these data bits. Lastly, the stop bits are always high, before the line returns to an idle state.

1.2 Modbus communication settings

Modbus over RS-485 acts like a single-client, multiple-server network, where the client sends a request to a specific server and the server responds with the expected message. Hukseflux Modbus instruments always act as servers on the network.

Valid device addresses fall in the range of 1 – 247. Each device on the bus, both client and server(s), needs to be configured with the same serial communication settings.

The default factory address of a Hukseflux Modbus instrument is 1.

NOTICE

Each device on the RS-485 network should have the same serial communication settings.

NOTICE

Each Modbus server device on the RS-485 network should have a unique device address.

1.3 Modbus request structure

1.3.1 Modbus frame

The standard Modbus frame consists of a Protocol Data Unit (PDU) that defines:
- A function code to indicate the type of Modbus request
- Data

For use in serial networks, the PDU is packed with additional fields into a Modbus Serial Line PDU that defines:
- A device address for addressing
- A cyclic redundancy check field for error detection

The structure of the Modbus Serial Line PDU is as follows:

Modbus request				
Device address	**Function code**	**Data**		**CRC**
Integer device address between 1 and 247	Indication of the action to perform	Any data needed to perform the Modbus function, *optional*		Error checking
1 byte	1 byte	N bytes		2 bytes

The CRC check uses the CRC-16 algorithm as described in Appendix B of the Modbus over Serial Line Specification and Implementation Guide v1.02

Example

For example, a valid Modbus request could look like this:

Modbus request				
Device address	**Function code**	**Data**		**CRC**
0x10	0x03	0x0A54	0x0004	0x4005
1 byte	1 byte	2 bytes		2 bytes

Note all values in the table are in hexadecimal notation

1.4 Supported Modbus function codes

This paragraph explains the purpose and use of the Modbus function codes that are supported by Hukseflux Modbus instruments. Each paragraph explains a function code and gives an example request and response. The possible error responses are also described.

MODBUS FUNCTION CODES	
FUNCTION CODE	**DESCRIPTION**
0x03	Read holding registers
0x04	Read input registers
0x06	Write single register
0x10	Write multiple registers
0x08	Diagnostics

When a request is sent, the response of the instrument will always start with the Modbus instrument's address, then the requested function code.

The instrument does not distinguish between input registers and holding registers, both function code 0x03 and 0x04 can be used to read the same 16-bit register, i.e. the input registers and holding registers share the same address space. Hukseflux Modbus instruments do not use coils (function code 0x01) or discrete inputs (function code 0x02).

Whenever an error response is generated, the most significant bit of the function code is set, e.g. for an error during function code 0x03, the function code in the response will be 0x83.

The exception codes supported by Hukseflux Modbus instruments are:

Supported exception codes		
Exception code	**Name**	**Description**
0x01	Illegal Function	The requested function code is not supported by the Hukseflux Modbus instrument.
0x02	Illegal Data Address	The data address received is not allowed by the Hukseflux Modbus instrument, for instance when requesting data from a non-existing data address, or when reading multiple registers and part of the address range does not exist.
0x03	Illegal Data Value	The supplied data fields are illegal in context of the request, for instance if the request supplies the wrong number of data bytes. Note: this exception is specifically NOT used to indicate values not supported by the Hukseflux Modbus instrument, for that, exception code 0x04 is used.
0x04	Server Device Failure	Executing the requested Modbus function failed, for instance because a data value not supported by the Hukseflux Modbus instrument was supplied in the request. Another case in which this exception code might be returned is in the case of a communication failure with one of the internal sensors.
0x06	Server Device Busy	The Hukseflux Modbus instrument is already processing another function. Repeat the request later.

1.4.1 0x03 - Read Holding Registers

This function code is used to read a the contents of a contiguous block of holding registers in a Hukseflux Modbus instrument. For Hukseflux Modbus instruments, holding registers and input registers are treated in the same way; this means that both function code 0x03 and 0x04 can be used to read the same group of registers.

Each register is packed as two bytes, the first byte contains the high order bits, the second byte contains the low order bits.

Modbus request				
Device address	**Function code**	**Read start address**	**No. of registers**	**CRC**
1 – 247	0x03	0x0000 – 0xFFFF	0x0000 – 0xFFFF	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Modbus response				
Device address	**Function code**	**Data**		**CRC**
1 – 247	0x03	0x0000 – 0xFFFF	…	Valid CRC16
1 byte	1 byte	N x 2 bytes	…	2 bytes

Error response			
Device address	**Function code**	**Exception code**	**CRC**
1 – 247	0x83	01, 02, 03, 04 or 06	Valid CRC16
1 byte	1 byte	1 byte	2 bytes

Examples

Example request, note all values in the tables below are in hexadecimal notation:

Example Modbus request				
Device address	**Function code**	**Read start address**	**No. of registers**	**CRC**
0x01	0x03	0x000A	0x0002	0x09E4
Request to device 0x01	Perform function code 0x03 (Read Holding Registers)	Start reading at address 0x000A	Read 2 registers	Valid CRC for this message

Example response:

Example Modbus response				
Device address	**Function code**	**Data**		**CRC**
0x01	0x03	0x010A	0x0402	0x35E7
Response from device 0x01	Performed function code 0x03 (Read Holding Registers)	Data read from register 0x000A	Data read from register 0x000B	Valid CRC for this message

Example error response:

Example Error response			
Device address	**Function code**	**Exception code**	**CRC**
0x01	0x83	0x01	0xF080
Response from device 0x01	Error performing function code 0x03 (Read Holding Registers)	Exception code 01	Valid CRC for this message

1.4.2 0x04 - Read Input Registers

This function code is used to read a the contents of a contiguous block of input registers in a Hukseflux Modbus instrument. For Hukseflux Modbus instruments, holding registers and input registers are treated in the same way; this means that both function code 0x03 and 0x04 can be used to read the same group of registers.
Each register is packed as two bytes, the first byte contains the high order bits, the second byte contains the low order bits.

Request:

Modbus request				
Device address	**Function code**	**Read start address**	**No. of registers**	**CRC**
1 – 247	0x04	0x0000 – 0xFFFF	0x0000 – 0xFFFF	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Modbus response				
Device address	**Function code**	**Data**		**CRC**
1 – 247	0x04	0x0000 – 0xFFFF	…	Valid CRC16
1 byte	1 byte	N x 2 bytes	…	2 bytes

Error response			
Device address	**Function code**	**Exception code**	**CRC**
1 – 247	0x84	01, 02, 03, 04 or 06	Valid CRC16
1 byte	1 byte	1 byte	2 bytes

Examples

Example request, note all values in the tables below are in hexadecimal notation:

Example Modbus request				
Device address	**Function code**	**Read start address**	**No. of registers**	**CRC**
0x01	0x04	0x000A	0x0002	0xC951
Request to device 0x01	Perform function code 0x04 (Read Input Registers)	Start reading at address 0x000A	Read 2 registers	Valid CRC for this message

Example response:

Example Modbus response				
Device address	**Function code**	**Data**		**CRC**
0x01	0x04	0x010A	0x0402	0xF552
Response from device 0x01	Performed function code 0x04 (Read Input Registers)	Data read from register 0x000A	Data read from register 0x000B	Valid CRC for this message

Example error response:

Example Error response			
Device address	**Function code**	**Exception code**	**CRC**
0x01	0x84	0x01	0xC082
Response from device 0x01	Error performing function code 0x04 (Read Input Registers)	Exception code 01	Valid CRC for this message

1.4.3 0x06 - Write Single Register

This function code is used to write data to a single 16-bit register in a Hukseflux Modbus instrument.
The 16-bit data to write is packed as two bytes, the first byte contains the high order bits, the second byte contains the low order bits.

Modbus request				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
1 – 247	0x06	0x0000 – 0xFFFF	0x0000 – 0xFFFF	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Modbus response				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
1 – 247	0x06	0x0000 – 0xFFFF	0x0000 – 0xFFFF	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Error response			
Device address	**Function code**	**Exception code**	**CRC**
1 – 247	0x86	01, 02, 03, 04 or 06	Valid CRC16
1 byte	1 byte	1 byte	2 bytes

Examples

Example request, note all values in the tables below are in hexadecimal notation:

Modbus request				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
0x01	0x06	0x000A	0xA5A5	0xE312
Request to device 0x01	Perform function code 0x06 (Write Single Register)	Write data at address 0x000A	Write 0xA5A5	Valid CRC for this message

Example response, note the valid response to a function code 0x06 request is an echo of the request itself:

Modbus response				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
0x01	0x06	0x000A	0xA5A5	0xE312
Request to device 0x01	Performed function code 0x06 (Write Single Register)	Write data at address 0x000A	Write 0xA5A5	Valid CRC for this message

Example error response:

Error response			
Device address	**Function code**	**Exception code**	**CRC**
0x01	0x86	0x02	0xA1C3
Response from device 0x01	Error performing function code 0x06 (Write Single Register)	Exception code 02	Valid CRC for this message

1.4.4 0x10 - Write Multiple Registers

This function code is used to write a contiguous block of registers in a Hukseflux Modbus instrument. The maximum number of registers that can be written is 123.
Each register is packed as two bytes, the first byte contains the high order bits, the second byte contains the low order bits.

Modbus request						
Device address	**Function code**	**Write start address**	**No. of registers**	**Byte count**	**Data N**	**CRC**
1 – 247	0x10	0x0000 – 0xFFFF	1 – 123 (N registers)	Amount of bytes following this byte (2 x N)	Data	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	1 byte	N x 2 bytes	2 bytes

Modbus response				
Device address	**Function code**	**Write start address**	**No. of registers**	**CRC**
1 – 247	0x10	0x0000 – 0xFFFF	1 - 123	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Error response			
Device address	**Function code**	**Exception code**	**CRC**
1 – 247	0x90	01, 02, 03, 04 or 06	Valid CRC16
1 byte	1 byte	1 byte	2 bytes

Examples

Modbus request						
Device address	**Function code**	**Write start address**	**No. of registers**	**Byte count**	**Data N**	**CRC**
0x01	0x10	0x0040	0x0002	0x04	0x55AA	0xAA55
Request to device at 0x01	Perform function code 0x10 (Write Multiple Registers)	Start writing at register 0x0040	Write 2 registers	4 bytes follow this byte	Write 0x55AA	Write 0xAA55

Modbus response				
Device address	**Function code**	**Write start address**	**No. of registers**	**CRC**
0x01	0x10	0x0040	0x0002	0x1C40
Response from device at 0x01	Performed function code 0x10 (Write Multiple Registers)	Started writing at register 0x0040	Wrote 2 registers	Valid CRC for this message

Error response			
Device address	**Function code**	**Exception code**	**CRC**
0x01	0x90	0x04	0xC34D
Response from device at 0x01	Error performing function code 0x10 (Write Multiple Registers)	Exception code 04	Valid CRC for this message

1.4.5 0x08 - Diagnostics

This function code is used to access a series of tests between for checking the communication between devices on the bus, or for checking internal error conditions. The function code uses two-byte sub-functions to determine the type of test to perform. Each register is packed as two bytes, the first byte contains the high order bits, the second byte contains the low order bits.

The supported sub-function codes for Hukseflux Modbus instruments are:

Diagonostic sub-function codes		
Sub-function code	**Name**	**Description**
0x0A	Clear Counters and Diagnostics Register	Clears all diagnostic counters and the diagnostics register.
0x0B	Return Bus Message Count	Requests the number of Modbus requests the instrument has detected on the communications bus since its last restart, counter overflow, or clear counters request. This is the number of requests to all devices, not just this device.

0x0C	Return Communication Error Count	Requests the number of CRC, framing, overrun, and incomplete message errors the instrument has detected since its last restart, counter overflow, or clear counters request.
0x0D	Return Exception Error Count	Requests the number of exception responses returned by the instrument since its last restart, counter overflow, or clear counters request.
0x0E	Return Server Message Count	Requests the number of requests addressed to the instrument since its last restart, counter overflow, or clear counters request.
0x0F	Return Server No Response Count	Requests the number of requests the instrument has not responded to, either with a normal response or an error response, since its last restart, counter overflow, or clear counters request.
0x10	Return Server NAK Count	Requests the number of requests the instrument has responded to with a "not acknowledged" (NAK) response since its last restart, counter overflow, or clear counters request.
0x11	Return Server Busy Count	Requests the number of requests the instrument has responded to with a "Server Busy" exception response since its last restart, counter overflow, or clear counters request.
0x12	Return Bus Character Overrun Count	Requests the number of requests addressed to the instrument that the instrument could not handle because the instrument received characters faster than they could be stored.

The diagnostis function request are:

Modbus request				
Device address	**Function code**	**Sub-function code**	**Data**	**CRC**
1 - 247	0x08	0x000A - 0x0012	0x0000 (for all supported sub-function codes)	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Modbus response				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
1 - 247	0x08	0x000A - 0x0012	0x0000 - 0xFFFF	Valid CRC16
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Error response			
Device address	**Function code**	**Exception code**	**CRC**
1 - 247	0x88	01, 03, 04 or 06	Valid CRC16
1 byte	1 byte	1 byte	2 bytes

Examples

Modbus request				
Device address	**Function code**	**Sub-function code**	**Data**	**CRC**
0x01	0x08	0x000E	0x0000	0xC881
Request to device 0x01	Perform function code 0x08 (Diagnostics)	Perform sub-function 0x000E (Return Server Message Count)	Valid data field value for this sub-function code	Valid CRC for this message

Modbus response				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
0x01	0x08	0x000E	0x0040	0x3880
Response from device 0x01	Performed function code 0x08 (Diagnostics)	Performed sub-function 0x000E (Return Server Message Count)	64 messages since last restart / counter reset	Valid CRC for this message

Error response			
Device address	**Function code**	**Exception code**	**CRC**
0x01	0x88	0x03	Valid CRC16
Response from device 0x01	Error performing function code 0x08 (Diagnostics)	Exception code 03	2 bytes

2 Hukseflux Modbus registers

2.1 Register access

Register may be readable and/or writeable. The register access may be different depending on the instruments operating mode.

REGISTER ACCESS	
ACCESS SPECIFIER	**DESCRIPTION**
R-	Readable in measurement mode
-W	Writeable in measurement mode

2.2 Register addresses

Registers are grouped by functionality. Groups have a size of 0x0080 (128) register addresses or a multiple thereof. Not all addresses within a group are used.

The register address of a register follows from the group start address and the register offset. The upper 9 bits of the register address are determined by the group start address, the lower 7 bits of the are determined by the register offset (as illustrated in figure xxx). The register address may be determined by performing a bitwise or operation on the register group start address and the register offset.

```
0bgggg_gggg_grrr_rrrr

g: group address bit
r: register offset bit
```

Figure 2.2.1: *Register address constructed from group start address and register offset.*

2.3 Register data types

All datatypes are big endian unless stated otherwise.

REGISTER DATA TYPES

DATA TYPE	**REGISTER (BYTE) COUNT**	**DESCRIPTION**
bool	1 (2)	Boolean TRUE or FALSE value
u16	1 (2)	Unsigned 16-bit integer
i16	1 (2)	Signed 16-bit integer
u32	2 (4)	Unsigned 32-bit integer
i32	2 (4)	Signed 32-bit integer
u64	4 (8)	Unsigned 64-bit integer
string32	16 (32)	UTF-8 encoded, 32 character string with 2 characters per Modbus register. The byte order is ‘BADCFE’.
float	2 (4)	Single-precision floating-point format (IEEE 754 binary32).
statistic	8 (16)	A struct containing 4 floating point numbers corresponding to the minimum, maximum, average and standard deviation. Accumulation is reset upon reading.

NOTICE

32-bit parameters located in two 16-bit Modbus registers must be read in a single Modbus request to guarantee coherence of these two registers. Always use function code 0x10 when reading a data type consisting of multiple 16 bit registers.

In the examples, the pseudocode conventions as described in Appendix A – Pseudocode conventions will be used.

The Modbus specification does not define any data types, leaving the interpretation of the 16-bit values in the registers up to the user. While explaining the various data types defined by Hukseflux, `u16` and `u32` will be used for any raw data read from a Modbus register that needs to be processed to another data type, because the unsigned integer data types contain the raw bit values without any bits that have an additional function (e.g. a sign bit).

2.3.1 bool

The `bool` data type is used for simple TRUE or FALSE information.

bool datatype	
Register content	**Description**
0x0000	FALSE
0xFFFF	TRUE
Anything else	Invalid data

Read a `bool` by reading a 16-bit register and converting the received data as described in the table above.
Write a `bool` by writing a 16-bit register and write one of the valid values described in the table above.

Convert a `bool` register to a boolean data type as follows:

Pseudocode – convert register value to bool

```
convert_bool (U16 register_value):
     IF register_value == 0xFFFF THEN
          RETURN true
     ELSE IF register_value == 0x0000 THEN
          RETURN false
     ELSE
          RETURN error
     ENDIF
```

2.3.2 u16

The `u16` data type is used to encode unsigned integers in the range of 0 to 2^{16}-1.

Read a `u16` by reading one 16-bit register. Write a `u16` by writing a single 16-bit register and write a value between 0 and 2^{16}-1.

2.3.3 i16

The `i16` data type is used to encode unsigned integers in the range of -2^{15} to 2^{15}-1.

The most significant bit gives the sign of the value, where a `0` signifies a positive number and a `1` signifies a negative number.

Read an `i16` by reading one 16-bit register.
Write an `i16` by writing a single 16-bit register and write a value between -2^{15} and 2^{15}-1.

2.3.4 u32

The `u32` data type is used to encode unsigned integers in the range of 0 to 2^{32}-1.

Read a `u32` by reading two 16-bit registers and combining the bytes to a single 32-bit value.
Write a `u32` by splitting a 32-bit value between 0 and 2^{32}-1 into two 16-bit parts, then writing two 16-bit registers.

2.3.5 i32

The `i32` data type is used to encode unsigned integers in the range of -2^{31} to 2^{31}-1.

The most significant bit gives the sign of the value, where a `0` signifies a positive number and a `1` signifies a negative number.

Read an `i32` by reading two 16-bit registers and combining the bytes to a single 32-bit value.
Write an `i32` by splitting a 32-bit value between -2^{31} and 2^{31}-1 into two 16-bit parts, then writing two 16-bit registers.

2.3.6 u64

The `u64` data type is used to encode unsigned integers in the range of 0 to 2^{64}-1.

Read a `u64` by reading four 16-bit registers and combining the bytes to a single 64-bit value.
Write a `u64` by splitting a 64-bit value between 0 and 2^{64}-1into four 16-bit parts, then writing four 16-bit registers.

u64 as a timestamp

One of the applications of the `u64` data type in Hukseflux Modbus instruments is as a timestamp. The decimal format of this timestamp is `00yyyymmdd`. This part is used as the 32 most significant bits in the `u64` value. The 32 least significant are set to zero. Convert a `u64` timestamp to a date as follows:

Pseudocode – convert register value to date

```
convert_date (U64 timestamp):
     year := FLOOR(timestamp{63:32} / 10000)
     month := FLOOR((timestamp{63:32} - year * 10000) / 100)
     day := timestamp{63:32} - month * 100 - year * 10000

     return year, month, day
```

Convert a date to a timestamp as follows:

Pseudocode – convert values to timestamp

```
convert_timestamp (year, month, day):
     U32 combined_date := 10000 * year + 100 * month + day
     U64 timestamp := combined_date << 32

     return timestamp
```

Examples:

Timestamp examples

Year	Month	Day	Combined date (decimal)	Timestamp (64-bit hexadecimal)
2024	2	27	20240227	0x0134d76300000000
1994	10	14	19941014	0x0130469600000000

2.3.7 string32

The `string32` data type is used to encode a string of 32 ASCII characters. Each ASCII character has the size of a byte, meaning two ASCII characters can fit in a single 16-bit Modbus register.

Read a `string32` by reading sixteen 16-bit registers and combining the bytes to a 32 ASCII character string.
Write a `string32` by splitting a 32 ASCII character string into sixteen 16-bit parts, then writing sixteen 16-bit registers.

The order of appearance for the ASCII characters in the string matches to the 16-bit registers as follows:
- The first and second characters ASCII are written in the first 16-bit register, the third and the fourth in the second 16-bit register, etc.
- The first appearing ASCII character in the string is placed in the least significant byte of the 16-bit register and the second appearing character is placed in the most significant byte.

To ensure that old values from `string32` register are fully cleared, strings that have less then 32 ASCII characters should be appended/padded with ASCII NULL characters (0x00 in hexadecimal notation).

Examples:

string32 examples		
ASCII string	**Reordered ASCII string**	**Reordered ASCII string as 16-bit values with NULL padding**
Hello Hukseflux!	eHll ouHskfeul!x	0x6548, 0x6c6c, 0x206f, 0x7548, 0x736b, 0x6665, 0x756c, 0x0078, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
Solar radiation measurement	oSal raridtaoi nemsarumenet	0x6f53, 0x616c, 0x2072, 0x6172, 0x6964, 0x7461, 0x6f69, 0x206e, 0x656d, 0x7361, 0x7275, 0x6d65, 0x6e65, 0x0074, 0x0000, 0x0000

Convert a string of ASCII characters into 16-bit parts of Modbus register data as follows:

Pseudocode – convert string to register data

```
convert_register_data (string):
     U16 data[16] := 0x00

     FOR i IN CEIL(string.length() / 2):
          data[i] := (string[(2 * i) + 1] << 8) OR (string[2 * i])

     return data
```

Convert 16-bit parts of Modbus register data into a string of ASCII characters as follows:

Pseudocode – convert register data to string

```
convert_string32 (U16 register_data[N]):
     STRING32 string

     FOR i IN register_data.length():
          string[2 * i] := register_data{7:0}
          string[(2 * i) + 1] := register_data{15:8}

     return string
```

2.3.8 float

The `float` data type is used to encode single-precision decimal numbers and follows the IEEE 754 standard. A floating point number is represented by a 32-bit value as follows:

float format		
Sign (`s`)	**Exponent (`x`)**	**Fraction (`f`)**
bit 31, 1 bit	bits 30 - 23, 8 bits	bits 22 - 0, 23 bits
Determines the sign of the number, `0` is a positive number, `1` is a negative number	The number to which 2 needs to be raised	Decimal part that is added to one, starting at a value of 0.5 for the most significant bit, the dividing by 2 for each additional bit; so 0.5, 0.25, 0.125, etc.

The value of the encoded number is calculated as follows:

$$value = -1^s \cdot 2^x \cdot (1 + f)$$

Special case: a value of "all 1s" for the exponent signals a `NaN` (Not a Number) value, i.e. a value that is undefined as a number, such as the outcome of `0/0`. Hukseflux Modbus instruments do not return `NaN` values as a valid register values, instead error response code 4 is used.

Examples:

float examples				
Number	**s**	**x**	**f**	**Result**
-1.25	1	01111111	01000000000000000000000	
	Negative number	127 - 127 = **0**	**0.25**	$-1^1 \cdot 2^0 \cdot (1 + 0.25) = -1 \cdot 1 \cdot 1.25 = -1.25$
103.625	0	10000101	10011110100000000000000	
	Positive number	133-127 = **6**	0.5 + 0.0625 + 0.03125 + 0.015625 + 0.0078125 + 0.001953125 = **0.619140625**	$-1^0 \cdot 2^6 \cdot (1 + 0.619140625) = 1 \cdot 64 \cdot 1.619140625 = 103.625$

Read a `float` by reading two 16-bit registers and concatenating the register contents into a single number, all float registers in Hukseflux Modbus instruments are encoded high-to-low, so that the first received 16-bit value contains the high part, and the second received 16-bit value contains the low part of the final 32-bit value.
Write a `float` by writing two 16-bit registers with a 32-bit float split over 2 16-bit registers (4 bytes).

Most programming languages support the casting of an unsigned 32-bits integer to float. In the case the programming language of choice does not support this, convert a 32-bit integer value to float as follows:

Pseudocode – convert register value to float

```
convert_float (U32 register_content):
      sign := register_content{31}
      exponent := register_content{30:23}
      fraction := register_content{22:0}

      register_value := -1^sign * 2^exponent * (1 + fraction)

      return register_value
```

2.3.9 statistic

The `statistic` data type is used to encode the minimum, maximum, average and standard deviation for a measurement.

Read a `statistic` object by reading eight 16-bit registers and combining the bytes into four `float` values, each representing one of the statistics (minimum, maximum, average or standard deviation) for the corresponding measurement. It is also possible to read only a part of a `statistic` object by only reading the 16-bit registers corresponding to the desired part of the `statistic` object.
Every read (full or partial) of the `statistic` data type results in all corresponding statistics being cleared. The consequence of this behaviour is that in order to read an entire `statistic` object without any of the statistics being cleared between reads, a single Modbus request to read multiple registers should be used.

The following paragraphs describe how each of the statistical values within a `statistic` data type are calculated.

Minimum
The minimum value statistic is determined by comparing each new measurement to the minimum value that is currently stored in the `statistic` object. In case the new measurement has a lower value than the value stored in the `statistic` object, the `statistic` object will be updated with the new measurement as the new minimum value.

Examples:

Minimum example

Current minimum	**New measurement**	**New minimum**
-10.0	5.0	5.0
-15.0	-17.0	-17.0
20	30.0	20.0

Maximum
The maximum value statistic is determined by comparing each new measurement to the maximum value that is currently stored in the `statistic` object. In case the new

measurement has a higher value than the value stored in the `statistic` object, the `statistic` object will be updated with the new measurement as the new maximum value.

Examples:

Maximum example		
Current maximum	**New measurement**	**New maximum**
5.0	10.0	10.0
-17.0	-15.0	-15.0
30.0	20.0	30.0

Average
The average value statistic is determined by dividing the sum of each measurement since the `statistic` object was last read by the amount of measurement that were performed during this time. In case a measurement has not yet been performed since the last read to this statistics value, the average will be 0.

Examples:

Average example		
Sum since last read	**Measurements since last read**	**Average value**
1536.19	57	$\frac{1536.19}{57} = 26.9507$
-256.8	10	$\frac{-256.8}{10} = -25.68$
X	0	0

Standard deviation
The standard deviation statistic is determined in a couple of steps:

Step 1
The average value is determined after a new measurement has been performed. This value will be referred to as the *mean* value in the following steps.

Step 2
The mean value is used to calculate the squared deviation from the mean value (S_{SDM}):

$$S_{SDM} = S_{SDM,prev} + (V - M)^2$$

Here, $S_{SDM,prev}$ is the previously calculated value ofS_{SDM}, V is the new measurement for which the statistics are being updated and M is the calculated mean value from Step 1.

Step 3
The standard deviation (σ) is calculated:

$$\sigma = \sqrt{\frac{S_{SDM}}{N}}$$

Here, N is the number of measurements since the last read to this statistics value. In case a measurement has not yet been performed since the last read to this statistics value, the standard deviation will be 0.

Examples:

float examples					
M	**V**	**$S_{SDM,prev}$**	**N**	**S_{SDM}**	**σ**
1536.2	1471.78	12.5	87	$12.5 + (1471.78 - 1536.2)^2 = 4162.44$	$\sqrt{\frac{4162.44}{87}} = 6.92$
-461.01	10.57	2314.78	263	$2314.78 + (10.57 - (-461.01))^2 = 224702.47$	$\sqrt{\frac{224702.47}{263}} = 29.23$
X	X	X	0	X	0

2.4 Register list

This paragraph explains how the Modbus register list supplied with the instrument should be interpreted and used. Each sub-paragraph explains a column of the Modbus register list and provides examples where appropriate. Note that all values in the tables below are in hexadecimal notation.

Register list columns
Column name
Register name
Register address
Register type
Register description
Enumeration
Default value
Persistent
Unit

2.4.1 register_name

All registers in the Modbus register list have a defined register name. These names give summarized explanation on the data behind the register or the action can be started by reading or writing to the register. For a more detailed explanation of the register, the register description column is used.

The register name column has no further function and is not relevant when composing a Modbus request.

2.4.2 register_address

All registers in the Modbus register list have a defined register address. This value should be used as the read or write start address when composing a Modbus request to a register from the Modbus register list. The values in the register column are presented in 16-bit hexadecimal notation.

Example

An example is given below on how to compose a Modbus request using the register address value from the Modbus register list. Consider a Modbus register with register address 0x1234 for a device with device address 1.
The following example shows how to compose a Modbus read request using function code 0x03 (Read Holding Registers) to read the data for the above described register. This example assumes that the data is 32-bit, so the number of registers field in the request will be set to 2.

Read request				
Device address	**Function code**	**Read start address**	**No. registers**	**CRC**
0x01	0x03	0x1234	0x0002	0xC0BC

2.4.3 register_type

All registers in the Modbus register list have a type. This defines the datatype of the data or configuration option that the register represents. Since datatypes can have different sizes, the register type also defines the amount of 16-bit Modbus registers that have to be interacted with to fully read or write the value that the register represents. The following table gives the full name for all datatypes as they are presented in the register_type column as well as the size in 16-bit Modbus registers.

REGISTER DATA TYPES			
REGISTER TYPE	**DATA TYPE**	**REGISTER (BYTE) COUNT**	**DESCRIPTION**
bool	bool	1 (2)	A single value signaling TRUE or FALSE
u16	u16	1 (2)	Unsigned 16-bit integer
i16	i16	1 (2)	Signed 16-bit integer
u32	u32	2 (4)	Unsigned 32-bit integer
i32	i32	2 (4)	Signed 32-bit integer
u64	u64	4 (8)	Unsigned 64-bit integer
string32	string32	16 (32)	UTF-8 encoded, 32 character string with 2 characters per Modbus register. The byte order is 'BADCFE'.
float	float	2 (4)	Single-precision floating-point format (IEEE 754 binary32).
statistic	statistic	8 (16)	A struct containing 4 floating point numbers corresponding to the minimum, maximum, average and standard deviation. Accumulation is reset upon reading.

2.4.4 register_access

All registers in the Modbus register list have a defined register access field. This field defines if the register data can be read and/or written. The format of this field is `<ReadAccess><WriteAccess>`.

`<ReadAccess>` indicates if a register accepts a Modbus read request by being defined as either an `R`, indicating that the register does accept a Modbus read request, or a `-`, indicating that the register does not accept a Modbus read request.

`<WriteAccess>` indicates if a register accept a Modbus write request by being defined as either a `W`, indicating that the register does accept a Modbus write request, or a `-`, indicating that the register does not accept a Modbus write request.

In practice, Modbus registers will always accept a read request. This leads to the set of supported access combinations defined in the following table.

Register access options	
Register access	**Description**
`R-`	The register accepts Modbus read requests. The register does not accept Modbus write requests.
`RW`	The register accepts Modbus read requests. The register accepts Modbus write requests.

Not adhering to the register_access field, i.e. sending a Modbus write request to a register that only accepts Modbus read requests, will result in the device responding with exception code 0x04.

2.4.5 register_description

All registers in the Modbus register list have a register description. This field contains a short explanation about the corresponding register. This description can contain information such as the unit of data that can be retrieved using the register, i.e. `Instrument temperature in °C`. Other type of registers might have a description containing the action that will be taken when a Modbus request is send to that register, i.e. `Soft restart`.

The register description column has no further function and is not relevant when composing a Modbus request.

2.4.6 enumeration

Registers in the Modbus register list can have a defined enumeration in the enumeration column. An enumeration is a collection of named elements in which each element corresponds to a unique number in that collection.

The format used in the Modbus register list for a named element and corresponding number within an enumeration is `<number>: <named_element>`. The number will always be presented in decimal notation.

A register in the Modbus register list with a defined enumeration will always be of the type u16.

Registers in the Modbus register list with a defined enumeration will only accept Modbus write requests containing a value from this enumeration as the write data. A write requests to these registers containing a value not specified in the enumeration will result in the device responding with Modbus exception code 0x04.

Similarly, registers in the Modbus register list with a defined enumeration will always respond to a Modbus read request with a value from this enumeration.

Examples

A couple examples are given below on how to use a defined enumeration from the Modbus register list to compose the data for a write request and to decode the data from a response to a read request to such a Modbus register.

Consider a register with address 0x1234 and with the following defined enumeration:

```
0:   OptionA
1:   OptionB
5:   OptionC
10:  OptionD
```

The following example shows how to compose a Modbus write request to write `OptionC` from the above defined enumeration to the corresponding register using function 0x06 (Write Single Register) to a device with device address 1. The above defined enumeration shows that element `OptionC` corresponds to the number 5, or 0x0005 in 16-bit hexadecimal notation.

Write request				
Device address	**Function code**	**Write address**	**Write data**	**CRC**
0x01	0x06	0x1234	0x0005	0x0D7F

The following example shows how to decode a Modbus response to a read request to this register using function 0x04 (Read Input Register) and a device with device address 1. This request can result in the following response:

Response to read request			
Device address	**Function code**	**Read data**	**CRC**
0x01	0x04	0x000A	0xC01E

The device responded with 0x000A, or 10 in decimal notation, as data. Looking at the above defined enumeration, 10 corresponds to `OptionD`.

2.4.7 Default value

This column lists the factory default value of the register, where applicable.

2.4.8 Persistent

If a programmed value is saved across instrument power cycles and soft-reboots, the value in this column is `True`. Otherwise, if the value is not saved across power cycles and reboots, the value is `False`.

2.4.9 Unit

If the value in a register has a physical unit, the unit is given in this column. If a value does not have a specific unit, for instance when a value is just a number, or a string, the unit is listed as `Undefined`.

Hukseflux
Thermal Sensors

Appendix A – Pseudocode conventions

Throughout the explanation of the various register data types, pseudocode is used when explaining how to work with the data types, or how to convert the received Modbus data to the intended value. The conventions used throughout this chapter are listed below.

Pseudocode

```
pseudocode is placed in blocks formatted like this
```

`IF`, `THEN`, `ELSE`, `OR`, `AND`, `WHILE`, `RETURN` are keywords used to signify common programming language functions. These keywords are always capitalised.
Common operators are `+`, `-`, `*`, `/`, `^` (raise to the power of), `<<` (bitwise left-shift), and `>>` (bitwise right-shift). Comparison is made with `<` (smaller than), `>` (larger than), `<=` (smaller than or equal), `>=` (larger than or equal), `==` (exact same value). Assignment is made by `:=` to avoid confusion between `=` and `==`. `FLOOR` is used to indicate a value should be rounded down to the nearest integer value. `CEIL` is used to indicate a value should be rounded up to the nearest integer value.

Explanatory comment in the code are placed between `/*` and `*/`, like this: `/* this is a comment*/`

A value prepended with `0x` is in hexadecimal notation, a value prepended with `0b` is in binary notation. Arrays of a data type are denoted as `DATA_TYPE[N]` where `DATA_TYPE` is the data type and `N` is the number of items in the array. Positions in the array are counted from 0, meaning that the first item in the array is found at position `0` and the last item in the array at position `N-1`. For example:

Array example

Notation	Meaning	Last item at position
`U16[4]`	4 items of data type `u16`	3
`FLOAT[16]`	16 items of data type `float`	15

The notation `content{23:14}` denotes that bit 23 downto (and including) bit 14 from a value are used to construct a value as if the least significant bit (bit 14 in this case) were bit 0. This means that `content` would be bit-masked and right-shifted to obtain the correct value, as shown here for `content{23:14}`:

Pseudocode

```
/* using a range of bits from a value */

result := (content AND 0x00FFC000) >> 13
```

The `AND` operation with `0x00FFC000` (which is `0b00000000111111111100000000000000` in binary, note that bits 23 downto 14 are `1`) removes all bit content outside of the desired bit range. Right-shifting with `>>` places the least significant bit of the desired range in the position that represents a value of 1. The `result` of this combined operation then is the desired value.

Modbus requests return the content of 16-bit registers, most often received as a series of bytes. The received bytes need to be combined to either 16-bit or larger (in the case of for instance a 32-bit value) values. Examples how to combine received bytes to 16-bit and 32-bit values are given below. Note how bytes are left-shifted to the correct bit position, then combined by using a bitwise `OR`.

Pseudocode

```
/* combining bytes to multi-byte values */

/* 16-bit */
result_16 := (received_byte_0 << 8) OR received_byte_1

/* 32-bit */
result_32 := (received_byte_0 << 24) OR (received_byte_1 << 16) OR
(received_byte_2 << 8) OR received_byte_3

/* 64-bit */
result_64 := (received_byte_0 << 56) OR (received_byte_1 << 48) OR
(received_byte_2 << 40) OR (received_byte_3 << 32) OR (received_byte_4 <<
24) OR (received_byte_5 << 16) OR (received_byte_6 << 8) OR
received_byte_7
```

Conversely, creating byte values from a single 16-bit or 32-bit value to ready the value for transmission is done as shown below. Note that the most significant part of the value is stored in the byte with the lowest number (i.e. that is transmitted first).

Pseudocode

```
/* splitting values into multiple bytes */

/* 16-bit */
byte_0 := (value AND 0x0000FF00) >> 8
byte_1 := (value AND 0x000000FF)

/* 32-bit */
byte_0 := (value AND 0xFF000000) >> 24
byte_1 := (value AND 0x00FF0000) >> 16
byte_2 := (value AND 0x0000FF00) >> 8
byte_3 := (value AND 0x000000FF)

/* 64-bit */
byte_0 := (value AND 0xFF00000000000000) >> 56
byte_1 := (value AND 0x00FF000000000000) >> 48
byte_2 := (value AND 0x0000FF0000000000) >> 40
byte_3 := (value AND 0x000000FF00000000) >> 32
byte_4 := (value AND 0x00000000FF000000) >> 24
byte_5 := (value AND 0x0000000000FF0000) >> 16
byte_6 := (value AND 0x000000000000FF00) >> 8
byte_7 := (value AND 0x00000000000000FF)
```

Hukseflux
Thermal Sensors