

# IS4320: I2C Modbus RTU Master Stack

Function Codes 1, 2, 3, 4, 15 and 16

## Applications

- PLCs
- Process Control
- IIoT (Industrial Internet of Things)
- Precision Agriculture
- Building Automation
- Communications between PCBs

## Modbus Stack Characteristics

- Implemented Function Codes:
  - FC 1 – Read Coils
  - FC 2 – Read Discrete Inputs
  - FC 3 – Read Holding Registers
  - FC 4 – Read Input Registers
  - FC 15 – Write Multiple Coils
  - FC 16 – Write Multiple Holding Registers
- Available baud rates: 1200, 2400, 9600, 19200, 57600, and 115200 bps.

## Main Advantages

- Eliminates engineering time and costs for protocol implementation and testing.
- Simple and easy to use solution.
- Reduces product time-to-market (TTM).
- Reduces microcontroller CPU load.
- Reduces impact on microcontroller peripherals (no need for timers or UARTs).
- Saves microcontroller pins with a shared I2C.
- Features a small, easy-to-solder SO8N package.
- Provides a low-cost solution.
- Makes the Modbus protocol transparent.
- I2C Speeds: 100kHz, 400kHz, and 1MHz.

## General Description

The IS4320 is an integrated circuit with a built-in Modbus RTU Master Stack, offering a standalone Modbus Master solution accessible through an I2C interface for simple microcontroller integration.

The IS4320 features two communication buses: a TTL UART for Modbus transceiver (RS485, RS422, RS232, etc.) and an I2C-Serial Interface for the microcontroller.

The aim of the IS4320 is to save engineering time and costs associated with implementing and testing the Modbus RTU communication protocol, providing a reliable solution that reduces the time-to-market (TTM) of your product.

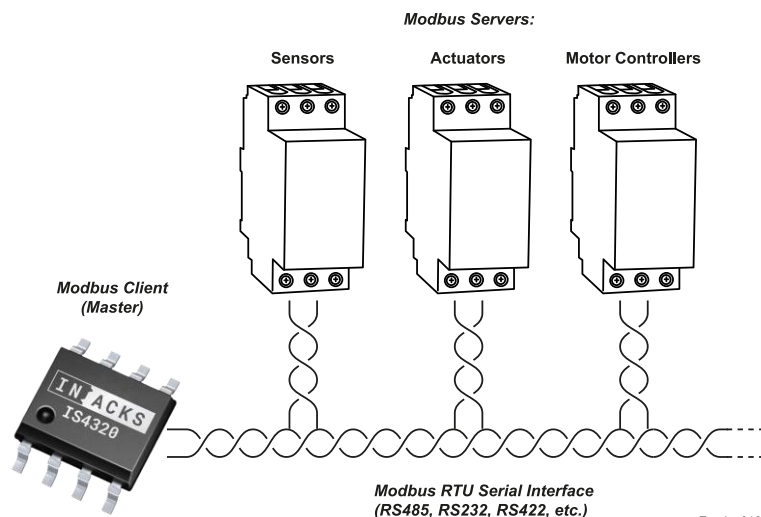
The IS4320 also brings benefits to your microcontroller: it utilizes I2C, eliminating the need for dedicated pins since I2C can be shared with other peripherals. Additionally, it eliminates the need for timers and decreases the CPU load on the microcontroller.

The device operates at 3.3V, and its I/O pins are 5V tolerant, allowing the use of either 3.3V or 5V transceivers. It is available in two temperature ranges: Industrial (-40°C to +85°C) and Extended (-40°C to +125°C) in an easy-to-solder SO8N package.

Part Number	Package	Op. Temperature
IS4320-S8-I	SO8N	-40°C to +85°C
IS4320-S8-E	SO8N	-40°C to +125°C




  

SDA	1	8	SCL
VDD	2	7	I2CSPD
VSS	3	6	DIR
TX	4	5	RX



Evadnrw0109

## Product Selection Guide

	Part Number	Form Factor	Physical Layer	Stack	Description
Stack Chip	IS4320-S8		SO8N	UART	Modbus RTU Master
					Modbus RTU Master Stack Chip. <a href="http://www.inacks.com/is4320">http://www.inacks.com/is4320</a> For smaller IC packages and product modifications, please contact us.
IS4320 Evaluation Boards	Kappa4320Ard		Arduino Compatible	RS485	Modbus RTU Master
	Kappa4320Rasp		Raspberry Pi Compatible	RS485	Modbus RTU Master

## 1. Electrical Specifications

### Absolute Maximum Ratings

Parameter			Min	Max	Unit
Input Voltage	VDD Pin		-0.3	4	V
	SCL, SDA, RX, TX, DIR Pins		-0.3	5.5	
	I2CSPD Pin		-0.3	4	
Current Sourced/Sunk by any I/O or Control Pin				±20	mA
Temperature	Operating Temperature	IS4320-S8-I	-40	+85	°C
		IS4320-S8-E	-40	+125	
	Storage Temperature		-65	+150	
Electrostatic Discharge (T <sub>A</sub> = 25°C)	Human-body model (HBM), Class 1C		-2000	+1500	V
	Charged-device model (CDM), Class C2a		-500	+500	

Exceeding the specifications outlined in the Absolute Maximum Ratings could potentially lead to irreversible harm to the device. It's important to note that these ratings solely indicate stress limits and don't guarantee the device's functionality under such conditions, or any others not specified in the Recommended Operating Conditions. Prolonged exposure to conditions at or beyond the absolute maximum ratings might compromise the reliability of the device.

### Recommended Operation Conditions

Parameter	Symbol	Min	Nom	Max	Unit
Supply Voltage	V <sub>DD</sub>	2.0	3.3	3.6	V
Input Voltage at SCL, SDA and RX, TX, DIR Pins	V <sub>I/O-IN</sub>	-0.3	3.3	5.5	
Input Voltage at I2CSPD Pin	V <sub>I2CSPD-IN</sub>	-0.3	1.8, 3.3	3.6	
Source/Sink Current at SCL, SDA and RX, TX, DIR Pins	I <sub>I/O-SS</sub>	-	-	±6	mA

### Electrical Characteristics

Parameter		Symbol	Min	Nom	Max	Unit
Current Consumption (T <sub>A</sub> = 25°C)		I <sub>OP</sub>	-	3.40	3.90	mA
Input Voltage	Logical High-Level	V <sub>IH</sub>	0.7xV <sub>DD</sub>	-	-	V
	Logical Low-Level	V <sub>IL</sub>	-	-	0.3xV <sub>DD</sub>	

Electrical Specifications Revision A



## 2.2. Organization

### Request and Response

The concepts of Request and Response are important to understand in Modbus. When your microcontroller uses the IS4320 to read from or write to a Modbus slave, the IS4320 sends a Request to the slave. The data or acknowledgment returned by the slave to the IS4320 is the Response. Even when the IS4320 writes data to a slave, the slave sends a Response to confirm that the write operation was received successfully.

### Modbus Memory Types

Modbus defines four types of memory for slaves: Discrete Inputs and Coils, which are organized as bits, and Input Registers and Holding Registers, which are organized as 16-bit words. Holding Registers is the most commonly used memory type.

Modbus slaves do not need to implement all memory types or registers. Often, only the Holding Registers memory type is used, with just the required number of registers. The memory and register count depend on the device's functionality. For example, a motor speed controller might implement a single Holding Register, allowing a Modbus Master to write a value between 0 and 100 to set its speed.

Memory Name	Organization	Read/Write	Access Method	Starting Address	Quantity of Registers	Accepts Broadcast?
Discrete Inputs	Bits	Read only	FC 2 – Read Discrete Input	0 to 0xFFFF	1 to 2000	No
Coils	Bits	Read/Write	FC 1 – Read Coils	0 to 0xFFFF	1 to 2000	No
			FC 15 – Write Multiple Coils	0 to 0xFFFF	1 to 1968	Yes
Input Registers	Words (16-bit)	Read only	FC 4 – Read Input Registers	0 to 0xFFFF	1 to 125	No
Holding Registers	Words (16-bit)	Read/Write	FC 3 – Read Holding Registers	0 to 0xFFFF	1 to 125	No
			FC 16 – Write Multiple Holding Registers	0 to 0xFFFF	1 to 123	Yes

### Holding Registers

A Modbus slave can have up to 65,536 Holding Registers, corresponding to addresses 0x0000 to 0xFFFF. You can access any register in this range and perform a read Request with Function Code 3 in quantities of 1 to 125 registers. To read more than 125 registers, you need to execute Function Code 3 multiple times. You can write to Holding Registers using Function Code 16 in quantities of 1 to 123 registers per execution. To write more than 123 registers, multiple executions of Function Code 16 are required.

Reading (FC 3) or writing (FC 16) a single register corresponds to accessing a full 16-bit word.

Function Code 16 (Write Multiple Registers) accepts broadcast. This means you can perform FC 16 to the special Modbus slave address 0, and the write Request will be sent to all Modbus slaves. Function Code 3 does not support broadcast.

### Input Registers

Input Registers work the same as Holding Registers but support read operations only, using Function Code 4.

### Coils

A Modbus slave can have up to 65,536 Coil status addresses, corresponding to addresses 0x0000 to 0xFFFF. Each address contains 16 Coil statuses, giving a total of 1,048,576 Coil statuses ( $65,536 \times 16 = 1,048,576$ ). You can access any Coil status in this range and perform a read Request in quantities of 1 to 2000 in a single Request using Function Code 1, or write in quantities of 1 to 1968 in a single Request using Function Code 15. More Coils can be accessed by executing the Request multiple times.

Reading (FC 1) or writing (FC 15) a single coil corresponds to accessing a 1-bit value.

Function Code 15 (Write Multiple Coils) supports broadcast. This means you can perform FC 15 to the special Modbus slave address 0, and the write Request will be sent to all Modbus slaves. Function Code 1 does not support broadcast.

### Discrete Inputs

Discrete Inputs work the same as coils but support read operations only, using Function Code 2.

## 2.3. IS4320 Advantages

The use of the IS3410 brings the following benefits:

1. Eliminates engineering time and costs for protocol implementation and testing.
2. Reduces product time-to-market (TTM).
3. Increases product reliability.
4. Saves microcontroller pins.
5. Reduces microcontroller CPU load.

The IS4320 significantly reduces engineering time by eliminating the need to manually implement and test the Modbus protocol. This time saving allows engineers to allocate resources more efficiently towards other critical aspects of product development. Additionally, this efficiency facilitates a faster time-to-market (TTM) and shortens the time to develop a minimum viable product (MVP). The streamlined development process enables companies to accelerate their product launch timelines, meeting market demands swiftly and effectively.

The use of a protocol stack chip eliminates the need for dedicated libraries on the microcontroller, both for the protocol and for the chip itself. Instead, it is accessed through a standard memory map using generic I2C functions, just like an I2C memory chip.

Using the IS4320 solution enhances customer application reliability, with the Modbus protocol already embedded, including its full protocol state machine and the T15 and T35 time constraints implemented.

Additionally, using the IS4320 can reduce Microcontroller pin requirements by saving three dedicated UART pins (Rx, Tx, and direction) and utilizing a shared bus like I2C.

Furthermore, offloading the Modbus protocol processing to the IS3410 saves Microcontroller CPU load, Flash, RAM memory, and Timer resources. This efficiency enhancement allows the Microcontroller to handle other tasks more effectively, contributing to overall system performance improvements and enabling the selection of a lower-end Microcontroller.

In conclusion, the usage of IS4320 not only streamlines development, enhances reliability, and accelerates time-to-market but also optimizes Microcontroller resources, making it a comprehensive solution for efficient product development and deployment.

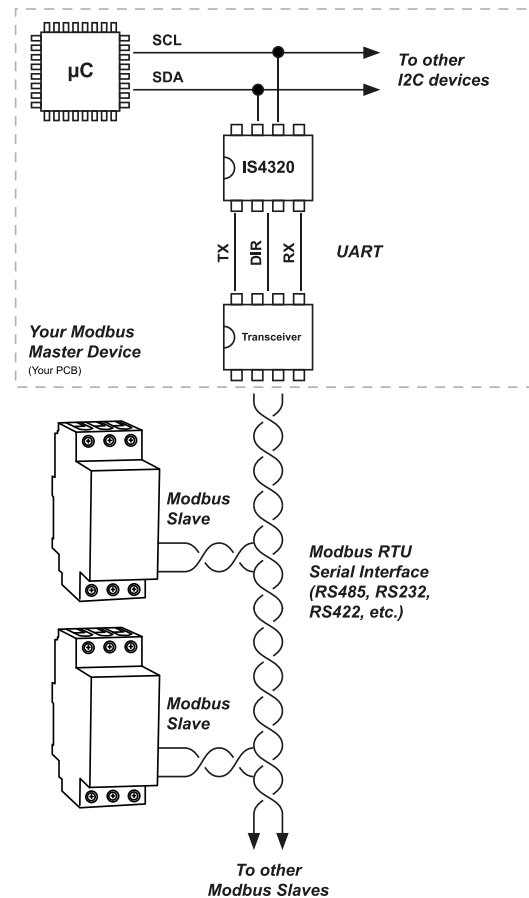
## 2.4. Modbus UART Port

The IS4320 is compatible with any Modbus RTU Serial Interface, including RS485, RS422, RS232, and others, thanks to its UART port. A transceiver matching the serial interface of the field bus (RS485, RS422, RS232, etc.) must be connected to the IS4320 UART port. This transceiver adapts the field bus voltage levels to 3.3V or 5V, ensuring proper operation with the IS4320.

For example, if the customer application connects to an RS485 field bus, an RS485 transceiver such as the THVD1500 should be used.

Refer to chapter Hardware Examples for hardware design examples.

Note: Connecting field buses like RS485 or others directly to the IS4320 will not work and will permanently damage the device.



Evadnw0111B

## 3. Usage

The IS4320 operates very simply via I2C in three steps. All operations are performed using I2C Word Write and I2C Word Read operations. These commands are the same ones typically used to access I2C EEPROMs and are already implemented in most microcontroller IDEs, such as STM32CubeIDE or Arduino.

Using the IS4320 can be summarized in three steps:

1. **Configure the IS4320 Modbus communications:**

Set the baud rate, parity, and stop bits to match the Modbus Slave device configuration by writing to the following registers: `CFG_MBBDR`, `CFG_MBPAR`, and `CFG_MBSTP`.

2. **Configure and execute the Modbus Request:**

Set the Slave ID, the Function Code, the Starting Address for the read or write operation, and the Quantity of registers. If performing a write operation, also set the data to be written. Finally, trigger the Request.

This is done using the following registers: `REQ_SLAVE`, `REQ_FC`, `REQ_STARTING`, `REQ_QTY` and `REQ_EXECUTE`. Use also `REQ_DATAx` for a write operation.

3. **Read the Modbus Response:**

After executing the Request, you can check the result of the operation by reading the `RES_STATUS` register. If you Requested to read some data, retrieve also the data by reading the `RES_DATAx` registers.



### 3.1. Example: Read Holding Registers

Suppose we have a Modbus temperature and humidity sensor. It has the Modbus Slave Address (Slave ID) 34, with temperature stored in Holding Register address 0 and humidity in Holding Register address 1. The sensor communicates over RS485 at 9600 bps, with even parity and 1 stop bit.

To read the temperature and humidity, the IS4320 must execute a Modbus Request using Function Code 3 (Read Holding Registers), with the starting address set to 0 and the quantity of registers set to 2 (to read register 0 and register 1).

The sequence involves three steps: configuring the Modbus communications, configuring and executing the Modbus Request, and reading the Modbus Response.

The IS4320 I2C register addresses and their contents are 16 bits long. Refer to chapter I2C Description for more information.

#### 1. First, configure the Modbus communications.

Register Type	I2C Register Address	Register Name	Register Description
Configuration Registers	0	CFG_MBBDR	Baud Rate Configuration
	1	CFG_MBPAR	Parity Bit Configuration
	2	CFG_MBSTP	Stop Bits Configuration
	3	CFG_MB_TIMEOUT	Modbus Response Timeout (ms)
	4	CFG_CHIP_ID	Chip Identification Number
	5	CFG_CHIP_REV	Chip Revision Number

**Figure 1: Configuration Registers of the IS4320 Memory Map**

#### 1.1. Configure the Baud Rate:

Perform an I2C Single Word Write to the I2C register address 0 (CFG\_MBBDR) with the value 112 to set the Baud Rate to 9600 bps.

For more information check section CFG\_MBBDR Register.

#### 1.2. Configure the Parity Bit:

Perform an I2C Single Word Write to the I2C register address 1 (CFG\_MBPAR) with the value 122 to set the parity to Even.

For more information check section CFG\_MBPAR Register.

#### 1.3. Configure the Stop Bits:

Perform an I2C Single Word Write to the I2C register address 2 (CFG\_MBSTP) with the value 131 to set only one stop bit.

For more information check section CFG\_MBSTP Register.

## 2. Second, configure and execute the Modbus Request.

Register Type	I2C Register Address	Register Name	Register Description
Request Registers	6	REQ_EXECUTE	Execute the Request
	7	REQ_SLAVE	Modbus Slave Address for the Request
	8	REQ_FC	Function Code for the Request
	9	REQ_STARTING	Starting Address for the Request
	10	REQ_QTY	Quantity of Register/Coils to read or write for the Request
	11	REQ_DATA1	Data to Write for the Request 1
	12	REQ_DATA2	Data to Write for the Request 2
	13	REQ_DATA3	Data to Write for the Request 3
	...	REQ_DATA4 to ...	...
	135	REQ_DATA125	Data to Write for the Request 125
	136	REQ_DATA126	Data to Write for the Request 126
	137	REQ_DATA127	Data to Write for the Request 127

**Figure 2: Request Registers of the IS4320 Memory Map**

### 2.1. Configure the Slave Address:

Perform an I2C Single Word Write to the I2C register address 7 (REQ\_SLAVE) with the value 34, which is the Modbus Slave Address of the temperature and humidity sensor. For more information check section REQ\_SLAVE Register.

### 2.2. Configure the Function Code:

Perform an I2C Single Word Write to the I2C register address 8 (REQ\_FC) with the value 3, which is *Function Code 3 Read Holding Registers*. For more information check section REQ\_FC Register.

### 2.3. Configure the Starting Address:

Perform an I2C Single Word Write to the I2C register address 9 (REQ\_STARTING) with the value 0, indicating that reading should start at register address 0 of the temperature and humidity sensor. For more information check section REQ\_STARTING Register.

### 2.4. Configure the Quantity of Registers:

Perform an I2C Single Word Write to the I2C register address 10 (REQ\_QTY) with the value 2, to retrieve 2 registers (register 0 and register 1). For more information check section REQ\_QTY Register.

### 2.5. Execute the Request:

Perform an I2C Single Word Write to the I2C register address 6 (REQ\_EXECUTE) to execute the Modbus Request we just configured for the temperature and humidity sensor. For more information check section REQ\_EXECUTE Register.

## 3. Third, read the Modbus Response.

Register Type	I2C Register Address	Register Name	Register Description
Response Registers	138	RES_STATUS	Response Status
	139	RES_DATA1	Read Data from the Response 1
	140	RES_DATA2	Read Data from the Response 2
	141	RES_DATA3	Read Data from the Response 3
		RES_DATA4	
	...	to	...
		RES_DATA122)	
	261	RES_DATA123	Read Data from the Response 123
	262	RES_DATA124	Read Data from the Response 124
	263	RES_DATA125	Read Data from the Response 125

Figure 3: Response Registers of the IS4320 Memory Map

## 3.1. Check the Response:

Perform an I2C Single Word Read to the I2C register address 138 (RES\_STATUS) to check the result of the Request. You should expect a value of 2, indicating that the server has received the Request. Make sure to allow I2C clock stretching in your microcontroller, as the IS4320 will hold the I2C read operation until it receives the Response from the Modbus Slave or a timeout occurs. For more information check section RES\_STATUS Register.

## 3.2. Read the Response:

Perform an I2C Multiple Word Read from the I2C register address 139 (RES\_DATA1), reading the same number of words as specified in the Request Quantity register (REQ\_QTY). Since we are reading 2 registers, 2 words (4 bytes) should be read. For more information check section RES\_DATAx Register.

### 3.2. Example: Write Holding Registers

Suppose we have a Modbus motor speed controller. It has the Modbus Slave Address (Slave ID) 27, with the motor speed configuration in its Holding Register address 0. It communicates over RS485 at 115200 bps, with no parity and 1 stop bit.

To write the new motor speed setpoint, the IS4320 must execute a Modbus Request using Function Code 16 (Write Multiple Registers), with the starting address set to 0 and the quantity of registers set to 1.

The sequence of involve three steps: configuring the Modbus communications, configuring and executing the Modbus Request, and verifying the Response.

The IS4320 I2C register addresses and their contents are 16 bits long. Refer to chapter I2C Description for more information.

#### 1. First, configure the Modbus communications.

Register Type	I2C Register Address	Register Name	Register Description
Configuration Registers	0	CFG_MBBDR	Baud Rate Configuration
	1	CFG_MBPAR	Parity Bit Configuration
	2	CFG_MBSTP	Stop Bits Configuration
	3	CFG_MB_TIMEOUT	Modbus Response Timeout (ms)
	4	CFG_CHIP_ID	Chip Identification Number
	5	CFG_CHIP_REV	Chip Revision Number

**Figure 4: Configuration Registers of the IS4320 Memory Map**

#### 1.1. Configure the Baud Rate:

Perform an I2C Single Word Write to the I2C register address 0 (CFG\_MBBDR) with the value 115 to set the Baud Rate to 115200 bps.  
For more information check section CFG\_MBBDR Register.

#### 1.2. Configure the Parity Bit:

Perform an I2C Single Word Write to the I2C register address 1 (CFG\_MBPAR) with the value 120 to set the parity to No Parity.  
For more information check section CFG\_MBPAR Register.

#### 1.3. Configure the Stop Bits:

Perform an I2C Single Word Write to the I2C register address 2 (CFG\_MBSTP) with the value 131 to set only one stop bit.  
For more information check section CFG\_MBSTP Register.

## 2. Second, configure and execute the Modbus Request.

Register Type	I2C Register Address	Register Name	Register Description
Request Registers	6	REQ_EXECUTE	Execute the Request
	7	REQ_SLAVE	Modbus Slave Address for the Request
	8	REQ_FC	Function Code for the Request
	9	REQ_STARTING	Starting Address for the Request
	10	REQ_QTY	Quantity of Register/Coils to read or write for the Request
	11	REQ_DATA1	Data to Write for the Request 1
	12	REQ_DATA2	Data to Write for the Request 2
	13	REQ_DATA3	Data to Write for the Request 3
	...	REQ_DATA4 to ...	...
	135	REQ_DATA125	Data to Write for the Request 125
	136	REQ_DATA126	Data to Write for the Request 126
	137	REQ_DATA127	Data to Write for the Request 127

**Figure 5: Request Registers of the IS4320 Memory Map**

### 2.1 Configure the Slave Address:

Perform an I2C Single Word Write to the I2C register address 7 (REQ\_SLAVE) with the value 27, which is the Modbus Slave Address of the motor speed controller.  
For more information check section REQ\_SLAVE Register.

### 2.2 Configure the Function Code:

Perform an I2C Single Word Write to the I2C register address 8 (REQ\_FC) with the value 16, which is *Function Code 16 Write Multiple Registers*.  
For more information check section REQ\_FC Register.

### 2.3 Configure the Starting Address:

Perform an I2C Single Word Write to the I2C register address 9 (REQ\_STARTING) with the value 0, indicating that reading should start at register address 0 of the motor speed controller.  
For more information check section REQ\_STARTING Register.

### 2.4 Configure the Quantity of Registers:

Perform an I2C Single Word Write to the I2C register address 10 (REQ\_QTY) with the value 1, to write 1 register (register 0).  
For more information check section REQ\_QTY Register.

### 2.5 Execute the Request:

Perform an I2C Single Word Write to the I2C register address 6 (REQ\_EXECUTE) to execute the Modbus Request we just configured for the motor speed controller.  
For more information check section REQ\_EXECUTE Register.

### 3 Third, read the Modbus Response.

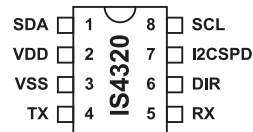
Register Type	I2C Register Address	Register Name	Register Description
Response Registers	138	RES_STATUS	Response Status
	139	RES_DATA1	Read Data from the Response 1
	140	RES_DATA2	Read Data from the Response 2
	141	RES_DATA3	Read Data from the Response 3
		RES_DATA4	
	...	to	...
		RES_DATA122)	
	261	RES_DATA123	Read Data from the Response 123
	262	RES_DATA124	Read Data from the Response 124
	263	RES_DATA125	Read Data from the Response 125

**Figure 6: Response Registers of the IS4320 Memory Map**

#### 3.1 Check the Response:

Perform an I2C Single Word Read to the I2C register address 138 (RES\_STATUS) to check the result of the Request. You should expect a value of 2, indicating that the server has received the Request. Make sure to allow I2C clock stretching in your microcontroller, as the IS4320 will hold the I2C read operation until it receives the Response from the Modbus Slave or a timeout occurs. For more information check section RES\_STATUS Register.

## 4. Pin Description



Pin	Name	Type	Description
1	SDA	Open Drain	I2C-compatible Data pin. Open drain, it requires pull-up. This is a 3.3 V pin and is 5 V tolerant.
2	VDD	Supply	3.3 V power supply pin. Bypass this pin to GND with a 100nF ceramic capacitor.
3	VSS	Ground	Ground reference pin.
4	TX	Digital Output Push-Pull	Modbus UART pins in TTL voltage levels. TX is the IS4320 transmit pin, RX the IS4320 receive pin. These are 3.3 V pins and are 5 V tolerant.
5	RX	Digital Input	
6	DIR	Digital Output Push-Pull	Direction pin for the transceivers, used to control the data flow direction on the bus. This pin goes high only when the IS4320 is transmitting data. It goes low while receiving data or waiting for data.  Example: In an RS485 transceiver, the Receiver Output Enable (RE) and Driver Output Enable (DE) pins are connected to this pin.
7	I2CSPD	Analog Input	I2C-Serial Interface Speed Selection pin. <ul style="list-style-type: none"> <li>For 100kHz pull to GND.</li> <li>For 400kHz make a voltage divider of VDD/2 (1.65V).</li> <li>For 1MHz pull to VDD (3.3V).</li> </ul>
8	SCL	Open Drain	I2C-compatible Clock pin. Open drain, it requires pull-up. This is a 3.3 V pin and is 5 V tolerant.

Pin Description Revision A

### 4.1. TX and RX Pins

Modbus UART Transmit and Receive Pins.

These pins handle UART transmit and receive functions for Modbus data and operate at TTL levels of 3.3V and they are 5V tolerant.

To interface with the field bus, these pins must connect to a suitable transceiver based on the field bus used: RS485, RS422, RS232, or others.

Please note that applying directly field bus (RS485, RS422, RS232, etc.) voltage levels to those pins will permanently damage the device.

For an RS485 fieldbus, use an RS485 transceiver, such as the THVD1500, to convert RS485 differential signaling to TTL/CMOS voltage levels. For an RS232 fieldbus, a transceiver like the MAX3221 can be used. Refer to the Hardware Examples chapter for more details.

### 4.2. DIR Pin

Modbus Direction Pin.

This pin is typically used in transceivers to control the data flow (sending or receiving). For RS485 transceivers, it connects to the DE and  $\overline{RE}$  pins of the transceiver.

Modbus Over Serial Line is usually implemented on “Two-Wire” RS485 electrical interface, which operates in a half-duplex topology. Therefore, a direction pin is needed to indicate whether the transceiver should send or receive data. By default, the DIR pin is in a low state, which sets the transceiver to receive mode.

## 4.3. SCL and SDA Pins

I2C-Compatible Bus Interface Pins.

**SCL (Serial Clock Line):** This pin is used to synchronize data transfer between the IS4320 device and the Microcontroller or other CPU.

**SDA (Serial Data Line):** This bidirectional pin is used for both sending and receiving data between the IS4320 and the Microcontroller or other CPU.

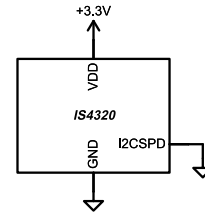
Both pins are open-drain and must be pulled up to 3.3 V or 5 V. The pull-up resistor value should be chosen based on the bus speed and capacitance. Typical values are 2 k $\Omega$  for Standard Mode (100 kHz) and 4.7 k $\Omega$  for Fast Mode (400 kHz) and Fast Mode Plus (1 MHz).

## 4.4. I2CSPD Pin

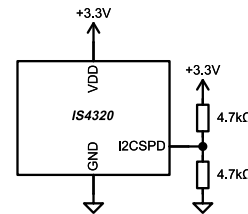
I2C-Serial Interface Speed Selection Pin.

This pin configures the IS4320 internal I2C-Serial Interface timings and filters to properly work with the selected bus speed.

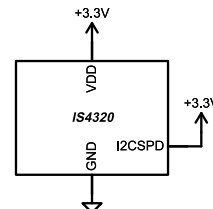
- For a **100 kHz** setting, set the I2CSPD pin to VSS.



- For a **400 kHz** setting, set the I2CSPD to 1.65 V ( $V_{DD}/2$ ) using a balanced voltage divider. This can be achieved by placing two 4.7 k $\Omega$  resistors from the I2CSPD pin: one to VDD and the other to VSS.



- For a **1000 MHz** setting, set the I2CSPD pin to VDD.



### Important Remark:

A mismatch between the configured I2C speed and the actual operating I2C speed (e.g., configuring the bus for 100 kHz but operating at 1 MHz) can lead to an inconsistent state where some I2C messages are processed while others are not.

Ensure a proper match between the actual operating speed and the configured speed at the I2CSPD pin: If your bus works at 100 kHz, ensure the I2CSPD pin is tied to VSS. If it works at 400 kHz ensure the pin is at 1.65 V. If it works at 1 MHz, ensure the pin is at 3.3 V.

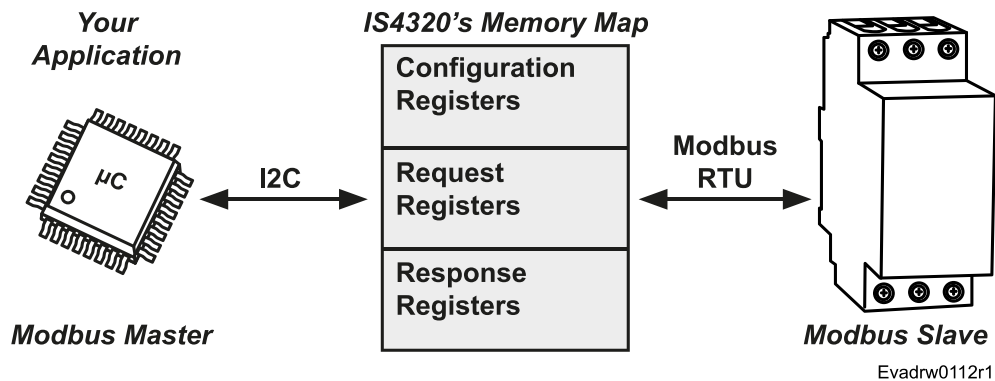


## 5. Memory Description

### 5.1. Memory Map Organization

The IS4320 is organized internally as a single page containing 264 registers, with addresses ranging from 0 to 263. These registers can be accessed individually or in blocks. The register addressing is 16-bit, and each register is 16-bit wide. There are

three types of registers: Configuration Registers, Request Registers, and Response Registers. All registers are readable, and most are writable, and can be accessed by the microcontroller via I2C.



#### Configuration Registers

The Configuration Registers range from address 0 to 5.

These registers are used to set the Modbus communication parameters: baud rate, parity bit, and stop bits. You can also set the Response timeout, which is the time (in milliseconds) the IS4320 waits for a Response after sending a Request.

Two special registers are also included: the Chip ID, which holds the constant number 20 and can be used to verify proper I2C communication, and the Chip Revision Number, which is used for production and product tracking.

The default configuration values are 19200 bauds, even parity, and one stop bit. You only need to set the configuration if your Modbus Slave differs from these settings. Once configured, you don't need to set it again as long as it matches your Modbus Slave communication parameters.

#### Request Registers

The Request Registers range from address 6 to 137.

These registers are used to set up and send Modbus Requests to the Modbus Slave. Here you set the Modbus Slave ID, the Function Code to be sent, the Starting Register for reading or writing data, and the Quantity of Registers to read or write. For write Requests, you also set the data to be sent.

#### Response Registers

The Response Registers range from address 138 to 263.

An important register stored here is `RES_STATUS`, which holds the result of the Modbus Request. This register indicates whether the Modbus Slave successfully received the Request, whether there was a timeout waiting for a Response, whether there was a Request configuration error, or whether the Slave reported an error.

The Response Registers also contain the data returned by the slave in reply to the master's Request.

## 5.2. Memory Map Table

Register Type	I2C Register Address	Register Name	Register Description
Configuration Registers	0	CFG_MBBDR	Baud Rate Configuration
	1	CFG_MBPAR	Parity Bit Configuration
	2	CFG_MBSTP	Stop Bits Configuration
	3	CFG_MB_TIMEOUT	Modbus Response Timeout (ms)
	4	CFG_CHIP_ID	Chip Identification Number
	5	CFG_CHIP_REV	Chip Revision Number
Request Registers	6	REQ_EXECUTE	Execute the Request
	7	REQ_SLAVE	Modbus Slave Address for the Request
	8	REQ_FC	Function Code for the Request
	9	REQ_STARTING	Starting Address for the Request
	10	REQ_QTY	Quantity of Register/Coils to read or write for the Request
	11	REQ_DATA1	Data to Write to the Modbus Slave
	12	REQ_DATA2	Data to Write to the Modbus Slave
	13	REQ_DATA3	Data to Write to the Modbus Slave
	...	REQ_DATA4 to ...	...
	135	REQ_DATA125	Data to Write to the Modbus Slave
	136	REQ_DATA126	Data to Write to the Modbus Slave
	137	REQ_DATA127	Data to Write to the Modbus Slave
Response Registers	138	RES_STATUS	Response Status
	139	RES_DATA1	Data Read from the Modbus Slave
	140	RES_DATA2	Data Read from the Modbus Slave
	141	RES_DATA3	Data Read from the Modbus Slave
	...	RES_DATA4 to ...	...
	261	RES_DATA123	Data Read from the Modbus Slave
	262	RES_DATA124	Data Read from the Modbus Slave
	263	RES_DATA125	Data Read from the Modbus Slave

### 5.3. CFG\_MBBDR Register

The `CFG_MBBDR` register stores the Modbus Baud Rate configuration. The default value is 113, representing 19200 bps, which is the default Modbus speed.

Allowed configuration values are 110 to 115. Any other value will be ignored.

- Value 110 sets a Modbus speed of 1200bps.
- Value 111 sets a Modbus speed of 2400bps.
- Value 112 sets a Modbus speed of 9600bps.
- Value 113 (default) sets a Modbus speed of 19200bps.
- Value 114 sets a Modbus speed of 57600bps.
- Value 115 sets a Modbus speed of 115200bps.

Any modifications to this register will take effect immediately after the I2C write operation.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** `CFG_MBBDR`  
**Description:** Baud Rate Configuration  
**Register Address:** 0 (0x000)  
**Default value:** 113 (0x0071)  
**Memory Type:** Volatile RAM  
**Allowed values:** 110 to 115 (0x006E to 0x0073)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	MBBDR [6 to 0]						

## 5.4. CFG\_MBPARG Register

The CFG\_MBPARG register stores the Modbus Parity Bit configuration. The default value is 122, representing Even Parity, which is the default Modbus Parity.

Allowed configuration values range from 120 to 122; attempting to write any other values will not have any effect.

- Value 120 represents No Parity.
- Value 121 represents Odd Parity.
- Value 122 (default) represents Even Parity.

Any modifications to this register will take effect immediately after the I2C write operation.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** CFG\_MBPARG  
**Description:** Parity Bit Configuration  
**Register Address:** 1 (0x001)  
**Default value:** 122 (0x007A)  
**Memory Type:** Volatile RAM  
**Allowed values:** 120 to 122 (0x0078 to 0x007A)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	CFG_MBPARG [6 to 0]						

## 5.5. CFG\_MBSTP Register

The `CFG_MBSTP` register contains the Modbus Stop Bits configuration. The default value is 131, representing One Stop Bit, which is the default Modbus Stop Bit.

Allowed configuration values range from 131 to 132; attempting to write any other values will not have any effect.

- Value 131 (default) One Stop bit (default).
- Value 132 Two Stop bit.

Any modifications to this register will take effect immediately after the I2C write operation.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** `CFG_MBSTP`  
**Description:** Stop Bits Configuration  
**Register Address:** 2 (0x002)  
**Default value:** 131 (0x0083)  
**Memory Type:** Volatile RAM  
**Allowed values:** 131 and 132 (0x0083 and 0x0084)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CFG_MBSTP [7 to 0]							

5.6. CFG\_MB\_TIMEOUT Register

Modbus Response timeout configuration register.

The CFG\_MB\_TIMEOUT register sets the Modbus Response timeout in milliseconds. Its default value is 1000 (1000 ms). This is the maximum time the IS4320 will wait for a Response from the Slave. If no Response is received within this period, the IS4320 sets the RES\_STATUS register to 3 – Sent and

Timeout. During this waiting period, the IS4320 will not accept any new Requests.

Allowed timeout milliseconds values range from 50 to 15000; attempting to write any other values will not be saved.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** CFG\_MB\_TIMEOUT  
**Description:** Modbus Response Timeout (ms)  
**Register Address:** 3 (0x003)  
**Default value:** 1000 (0x3E8)  
**Memory Type:** Volatile RAM  
**Allowed values:** 50 to 15000 (0x0032 and 0x3A98)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CFG_MB_TIMEOUT [14 to 0]							

## 5.7. CFG\_CHIP\_ID Register

The `CFG_CHIP_ID` register contains the chip identifier, which is a fixed value of 20. This value is used for production tracking. It is stored in ROM and will not change throughout the product's lifecycle.

Since this register value is constant, reading it during firmware development can help verify that I2C communications are working and that the chip's memory can be properly read.

This register is read-only.

**Name:** `CFG_CHIP_ID`  
**Description:** Chip Identification Number  
**Address:** 4 (0x004)  
**Memory Type:** ROM  
**Value:** 20 (0x0014)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
0	0	0	0	0	0	0	0

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	1	0	1	0	0

## 5.8. CFG\_CHIP\_REV Register

The CFG\_CHIP\_REV register indicates the chip revision.

This value is intended for production and product tracking. It is stored in ROM and may change throughout the product's lifecycle.

This register is read-only.

**Name:** CFG\_CHIP\_REV  
**Description:** Chip Revision Number  
**Address:** 5 (0x005)  
**Memory Type:** ROM  
**Value:** (Depends on the revision)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	-



## 5.9. REQ\_EXECUTE Register

Execute Request register.

Writing a 1 to this register sends the configured Modbus Request (set in the REQ\_x registers) to the Modbus Slave. Writing any value other than 1 has no effect. This register is automatically cleared to 0 immediately after being written with 1. The result of the Request is stored in the RES\_STATUS register.

Some Modbus slaves, especially those based on non-deterministic systems like PCs, may not respond well to high polling frequencies. It's good practice to leave a small delay between Requests if a high polling rate is not required. Polling once every second or every half-second are good starting points.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** REQ\_EXECUTE  
**Description:** Execute the Request  
**Register Address:** 6 (0x006)  
**Default value:** 0 (0x0000)  
**Memory Type:** Volatile RAM  
**Allowed values:** 0 or 1 (0x0000 or 0x0001)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	REQ_EXECUTE

## 5.10. REQ\_SLAVE Register

Modbus Slave Address register.

This register contains the Modbus Slave Address of the device you are going to send the Request.

The default value is 1. Each Modbus Slave must have a unique address, as two Slaves with the same address will cause both devices to become unresponsive.

Writing the special Modbus Slave address 0 broadcasts a message to all Slaves. The broadcast operation is only valid for write Requests, i.e., Function Codes 15 and 16.

Addresses 1 to 247 are used for unicast messages, that is, addressing a specific Slave. This is the most common usage.

Values higher than 247 are not allowed by the standard and will set the `RES_STATUS` register to error number 5 (REQ\_SLAVE Error).

This is a volatile RAM register. On each power-up, it loads its default value.

### Important Remark:

In a Modbus network, two slaves cannot have the same Address ID. Doing so will cause both devices to become unresponsive.

**Name:** REQ\_SLAVE  
**Description:** Modbus Slave Address for the Request  
**Register Address:** 7 (0x007)  
**Default value:** 1 (0x0001)  
**Memory Type:** Volatile RAM  
**Allowed values:** 1 to 247 (0x0000 to 0x00F7)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
REQ_SLAVE [7 to 0]							

### 5.11. REQ\_FC Register

Function Code Register for the Request.

This register contains the Function Code number that will be sent in the Request.

Valid Function Codes are: FC 1, FC 2, FC 3, FC 4, FC 15, FC 16.

Different values are not allowed and will set the `RES_STATUS` register to error number 6 (REQ\_FC Error).

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** REQ\_FC  
**Description:** Function Code for the Request  
**Register Address:** 8 (0x008)  
**Default value:** 3 (0x0003)  
**Memory Type:** Volatile RAM  
**Allowed values:** 1, 2, 3, 4, 15 and 16 (0x01, 0x02, 0x03, 0x04, 0x0F and 0x10)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	REQ_FC [4 to 0]				

5.12. REQ\_STARTING Register

Modbus Starting Address for the Request.

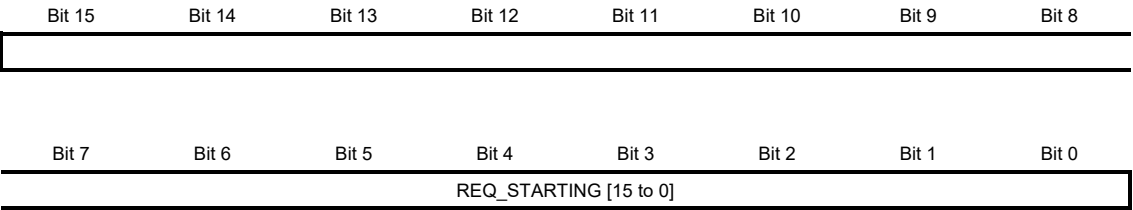
This register sets the Starting Address where the read or write Request will begin.

Rember that the Modbus memory range goes from address 0x0000 to 0xFFFF. Setting a combination of Starting Register + Quantity of Registers that exceeds 0xFFFF will result in an invalid memory

area and will set the RES\_STATUS register to value 202 (Modbus Exception Code 2 — ILLEGAL DATA ADDRESS). Modbus Slaves may also trigger this error if you attempt to access a Modbus memory address that is not implemented.

This is a volatile RAM register. On each power-up, it loads its default value.

**Name:** REQ\_STARTING  
**Description:** Starting Address for the Request  
**Register Address:** 9 (0x009)  
**Default value:** 0 (0x0000)  
**Memory Type:** Volatile RAM  
**Allowed values:** 0 to 65535 (0x0000 to 0xFFFF)



### 5.13. REQ\_QTY Register

This register sets how many Registers or Coils will be read or written in the Request.

The maximum Quantity depends on the Function Code:

- FC 1 allows reading between 1 and 2000 Coils
- FC 2 allows reading between 1 and 2000 Discrete Inputs
- FC 3 allows reading between 1 and 125 Holding Registers
- FC 4 allows reading between 1 and 125 Input Registers
- FC 15 allows writing between 1 and 1968 Coils
- FC 16 allows writing between 1 and 123 Holding Registers

Different values are not allowed and will set the RES\_STATUS register to error number 7 (REQ\_QTY Error).

Remember that the Modbus memory range goes from address 0x0000 to 0xFFFF. Setting a combination of Starting Register + Quantity of Registers that exceeds 0xFFFF will result in an invalid memory area and will set the RES\_STATUS register to value 202 (Modbus Exception Code 2 — ILLEGAL DATA ADDRESS). Modbus Slaves may also trigger this

error if you attempt to access a Modbus memory address that is not implemented.

This is a volatile RAM register. On each power-up, it loads its default value.

#### Holding Registers and Input Registers:

For the word (16-bit) Function Codes (FC 3, FC 4, FC 16), a Quantity of 1 register will read or write a full register (16-bit data size).

For a write Request (FC 16), the number of REQ\_DATAx registers sent corresponds to the REQ\_QTY register; for example, setting REQ\_QTY to 1 sends only REQ\_DATA1, while setting REQ\_QTY to 50 sends REQ\_DATA1 through REQ\_DATA50.

#### Coils and Discrete Inputs:

For the bit Function Codes (FC 1, FC 2, FC 15), the REQ\_QTY register defines how many bits are read or written. A Quantity of 1 affects bit 15 of the register REQ\_DATA1. A Quantity of 2 affects bits 15 and 14 of REQ\_DATA1. A Quantity of 17 bits will affect bits 15 to 0 of REQ\_DATA1 and bit 15 of REQ\_DATA2.

**Name:** REQ\_QTY  
**Description:** Quantity of Register/Coils to read or write for the Request  
**Register Address:** 10 (0x00A)  
**Default value:** 1 (0x0001)  
**Memory Type:** Volatile RAM  
**Allowed values:** 1 to 123/125/1968/2000 (0x0001 to 0x007B/0x007D/0x07B0/0x07D0)

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-			

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
REQ_QTY [10 to 0]							

## 5.14. REQ\_DATAx Register

REQ\_DATAx registers contain the data that will be written to the Modbus Slave when the Request is executed.

For the **16-bit write Function Code (FC 16 – Write Multiple Registers)**, the quantity of REQ\_DATAx registers to send is defined by the REQ\_QTY register.

For example, if REQ\_QTY = 1, only REQ\_DATA1 will be sent on the write Request. If REQ\_QTY = 100 REQ\_DATA1 to REQ\_DATA100 will be sent.

For **1-bit write Function Code (FC 15 – Write Multiple Coils)**, the number of REQ\_DATAx registers

to send is the result of REQ\_QTY divided by 16 (rounded up).

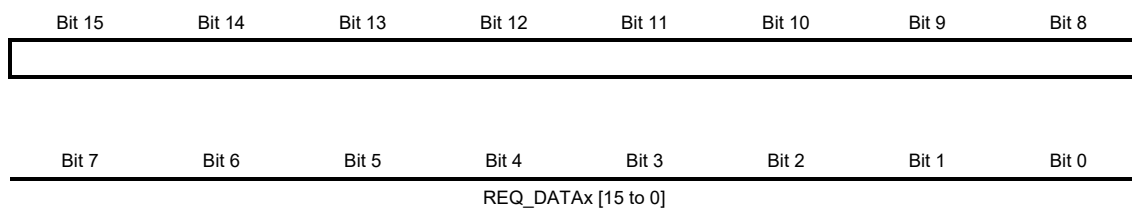
For example, a REQ\_QTY = 100 requires REQ\_DATA1 through REQ\_DATA7 (1600 coils ÷ 16 bits per register = 6.25 registers, rounded up to 7).

The following page shows a correlation between Modbus Coils Addresses and REQ\_DATAx registers.

REQ\_DATAx is set and cleared by the user. The IS4320 never changes this data.

These are volatile RAM registers. On each power-up, they are cleared to 0.

**Name:** REQ\_DATAx  
**Description:** Data to Write to the Modbus Slave  
**Register Address:** 11 to 137 (0x00B to 0x089)  
**Default value:** 0 (0x0000)  
**Memory Type:** Volatile RAM  
**Allowed values:** 0 to 65535 (0x0000 to 0xFFFF)



# IS4320 Modbus RTU Master

The table shows a correlation between Modbus Coils Addresses and REQ\_DATAx registers:

Coil Quantity	REQ_QTY=1	REQ_QTY=2	REQ_QTY=3	REQ_QTY=4	REQ_QTY=5	REQ_QTY=6	REQ_QTY=7	REQ_QTY=8	REQ_QTY=9	REQ_QTY=10	REQ_QTY=11	REQ_QTY=12	REQ_QTY=13	REQ_QTY=14	REQ_QTY=15	REQ_QTY=16	REQ_QTY=17	REQ_QTY=18	REQ_QTY=19	REQ_QTY=20	REQ_QTY=21	REQ_QTY=22	REQ_QTY=23	REQ_QTY=24	REQ_QTY=25	REQ_QTY=26	REQ_QTY=27	REQ_QTY=28	REQ_QTY=29	REQ_QTY=30	REQ_QTY=31	REQ_QTY=32	REQ_QTY=33	...
Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	...
REQ_DATAx Register	REQ_DATA1																REQ_DATA2																REQ...	

...	...	REQ_QTY=1936	REQ_QTY=1937	REQ_QTY=1938	REQ_QTY=1939	REQ_QTY=1940	REQ_QTY=1941	REQ_QTY=1942	REQ_QTY=1943	REQ_QTY=1944	REQ_QTY=1945	REQ_QTY=1946	REQ_QTY=1947	REQ_QTY=1948	REQ_QTY=1949	REQ_QTY=1950	REQ_QTY=1951	REQ_QTY=1952	REQ_QTY=1953	REQ_QTY=1954	REQ_QTY=1955	REQ_QTY=1956	REQ_QTY=1957	REQ_QTY=1958	REQ_QTY=1959	REQ_QTY=1960	REQ_QTY=1961	REQ_QTY=1962	REQ_QTY=1963	REQ_QTY=1964	REQ_QTY=1965	REQ_QTY=1966	REQ_QTY=1967	REQ_QTY=1968
0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
REQ...	REQ_DATA122																REQ_DATA123																	

## 5.15. RES\_STATUS Register

When you execute a Modbus Request by writing 1 to the REQ\_EXECUTE register, the result of the operation is shown in RES\_STATUS.

If the Modbus Slave has received and accepted the Request, RES\_STATUS will be set to 2 (Sent and Received). If the Modbus Slaves did not respond to the Request, the register will be set to 3 (Sent and Timeout).

While you are configuring the Request registers (REQ\_SLAVE, REQ\_FC and REQ\_QTY), REQ\_STATUS is set to 0 (Ready to Send) if all the Request Registers have a valid configuration, or to 5, 6 or 7 if there is an error.

This is a Read Only register.

**Name:** RES\_STATUS  
**Description:** Response Status  
**Register Address:** 138 (0x08A)  
**Default value:** 0 (0x0000)  
**Memory Type:** Volatile RAM, Read Only

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	-	-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RES_STATUS [7 to 0]							

Value	Description	Details
0	Ready to Send	Request registers REQ_SLAVE, REQ_FC, and REQ_QTY are valid, so the Request can be sent to the Modbus Slave.
1	Sent and Waiting	The Request was sent to the Modbus Slave, and the IS4320 is waiting for the answer. The maximum waiting time is set in the CFG_MB_TIMEOUT register. You must wait for either a Response or a timeout. No other Request should be executed while waiting for the Response.
2	Sent and Received	The Modbus Slave has confirmed the reception of the Request.
3	Sent and Timeout	The Request was sent, but no Response was received from the Modbus slave. You are free to try again or perform a different Request. You should assume the Request was not received.
4	Broadcast Sent	A broadcast message has been sent. Modbus slaves do not confirm broadcast messages, so the IS4320 is ready for the next operation.
5	REQ_SLAVE Error	The value in the REQ_SLAVE register is not valid. This register contains the Modbus Slave ID, which identifies the device on the Modbus bus. Valid Slave IDs are 1 to 247. ID address 0 (Broadcast) is only valid with FC 15 and FC 16. The Request was not sent.
6	REQ_FC Error	The value in the REQ_FC register is not valid. Valid Function Codes are: FC 1, FC 2, FC 3, FC 4, FC 15, FC 16. The Request was not sent.
7	REQ_QTY Error	The value in the REQ_QTY register is not valid. The minimum must be 1, and the maximum depends on the Function Code: FC 1: Max read 2000 Coils FC 2: Max read 2000 Discrete Inputs FC 3: Max read 125 Holding Registers FC 4: Max read 125 Input Registers FC 15: Max write 1968 Coils FC 16: Max write 125 Holding Registers The Request was not sent.
8	Frame Error	An invalid Response has been received. Possible causes include: non-Modbus data, CRC error, or incorrectly built Modbus data frame.
201	Modbus Exception Code 1	ILLEGAL FUNCTION
202	Modbus Exception Code 2	ILLEGAL DATA ADDRESS
203	Modbus Exception Code 3	ILLEGAL DATA VALUE
204	Modbus Exception Code 4	SERVER DEVICE FAILURE



## 5.16. RES\_DATAx Register

RES\_DATAx registers contain the data received from the Modbus Slave when a read Request was executed.

For **16-bit Read Function Code (FC 3, FC 4)**, the quantity of RES\_DATAx registers received was defined by the REQ\_QTY register in the Request.

For example, Requesting 1 register (REQ\_QTY = 1) will fill only RES\_DATA1. Requesting 100 registers (REQ\_QTY = 100) will fill RES\_DATA1 through RES\_DATA100. The remaining RES\_DATAx registers will be cleared to 0.

For **1-bit Read Function Codes (FC 1, FC 2)**, the number of RES\_DATAx registers received is the result of REQ\_QTY divided by 16 (rounded up).

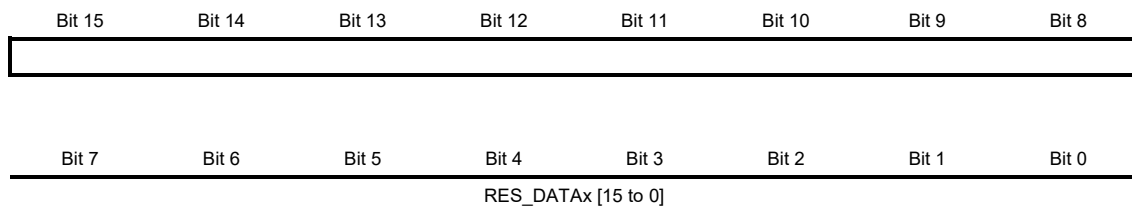
For example, Requesting 1 coil (REQ\_QTY = 1) will fill only bit 15 of RES\_DATA1. Requesting 100 coils (REQ\_QTY = 100) will fill RES\_DATA1 through RES\_DATA7 (100 coils ÷ 16 bits per register = 6.25, rounded up to 7). The remaining RES\_DATAx bits and registers will be cleared to 0.

The following page shows the correlation between Modbus Coils Addresses and REQ\_DATAx registers.

When executing a Write Function Code (FC 15, FC16), these registers are cleared to 0.

These are volatile RAM registers. On each power-up, they are cleared to 0.

**Name:** RES\_DATAx  
**Description:** Data Read from the Modbus Slave  
**Register Address:** 139 to 263 (0x08B to 0x107)  
**Default value:** 0 (0x0000)  
**Memory Type:** Volatile RAM, Read Only



# IS4320 Modbus RTU Master

The table shows a correlation between Modbus Coils Addresses and RES\_DATAx registers:

Coil Address	REQ_QTY=1	REQ_QTY=2	REQ_QTY=3	REQ_QTY=4	REQ_QTY=5	REQ_QTY=6	REQ_QTY=7	REQ_QTY=8	REQ_QTY=9	REQ_QTY=10	REQ_QTY=11	REQ_QTY=12	REQ_QTY=13	REQ_QTY=14	REQ_QTY=15	REQ_QTY=16	REQ_QTY=17	REQ_QTY=18	REQ_QTY=19	REQ_QTY=20	REQ_QTY=21	REQ_QTY=22	REQ_QTY=23	REQ_QTY=24	REQ_QTY=25	REQ_QTY=26	REQ_QTY=27	REQ_QTY=28	REQ_QTY=29	REQ_QTY=30	REQ_QTY=31	REQ_QTY=32	REQ_QTY=33	...
Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	...
RES_DATAx Register	RES_DATA1																RES_DATA2																RES...	

...	...	REQ_QTY=1968	REQ_QTY=1969	REQ_QTY=1970	REQ_QTY=1971	REQ_QTY=1972	REQ_QTY=1973	REQ_QTY=1974	REQ_QTY=1975	REQ_QTY=1976	REQ_QTY=1977	REQ_QTY=1978	REQ_QTY=1979	REQ_QTY=1980	REQ_QTY=1981	REQ_QTY=1982	REQ_QTY=1983	REQ_QTY=1984	REQ_QTY=1985	REQ_QTY=1986	REQ_QTY=1987	REQ_QTY=1988	REQ_QTY=1989	REQ_QTY=1990	REQ_QTY=1991	REQ_QTY=1992	REQ_QTY=1993	REQ_QTY=1994	REQ_QTY=1995	REQ_QTY=1996	REQ_QTY=1997	REQ_QTY=1998	REQ_QTY=1999	REQ_QTY=2000
0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
RES...	RES_DATA124																RES_DATA125																	

## 6. I2C Description

The IS4320 operates as a slave in the I2C-Serial Interface. It supports Standard Mode (100 kHz), Fast Mode (400 kHz), and Fast Mode Plus (1 MHz). The I2C-Master device, typically a microcontroller or a microprocessor, initiates and manages all read and write operations to the IS4320 (I2C-Slave device).

The IS4320 is represented on the bus by the I2C device address: 20 (0x14).

Pull-up resistors are required on the SCL and SDA lines for proper operation. The resistor values depend on the bus capacitance and operating speed. Typical values are 4.7 kΩ for Standard Mode (100 kHz) and 2 kΩ for Fast Mode and Fast Mode Plus (400 kHz and 1 MHz).

The IS4320's high state can be either 3.3V or 5V. A logical '0' is transmitted by pulling the line low, while a logical '1' is transmitted by releasing the line, allowing it to be pulled high by the pull-up resistor. The Master controls the Serial Clock (SCL) line, which generates the synchronous clock used by the Serial Data (SDA) line to transmit data.

A Start or Stop condition occurs when the SDA line changes during the High period of the SCL line. Data on the SDA line must be 8 bits long and is transmitted Most Significant Bit First and Most Significant Byte First. After the 8 data bits, the receiver must respond with either an acknowledge (ACK) or a no-acknowledge (NACK) bit during the ninth clock cycle, which is generated by the Master. To keep the bus in an idle state, both the SCL and SDA lines must be released to the High state.

The IS4320 memory map addressing (pointer register) is 16 bits wide, which means 2 bytes are required to set the memory address for any I2C read or write operation. The memory map itself contains 263 registers, each 16 bits wide. Therefore, 2 bytes of data are needed to read from or write to each IS4320 register.

The operability of the Read and Write commands of the IS4320 is very similar to an EEPROM memory. Thinking of the IS4320 as an EEPROM memory is a good analogy to quickly understand how to communicate with its memory map.

### 6.1. Highlights

- **I2C Device Address:** 20 (0x14)
- **I2C Memory Map Register Size and Addressing Size:** 16 bits, composed of two bytes — first the MSB, then the LSB.
- **Compatible I2C Speeds:**
  - Standard Mode (100kHz), recommended SCL and SDA pull-up value: 4.7 kΩ
  - Fast Mode (400kHz), recommended SCL and SDA pull-up value: 2 kΩ
  - Fast Mode Plus (1MHz), recommended SCL and SDA pull-up value: 2 kΩ
- **I2C Supported Operations:**
  - Single-Byte Write
  - Multiple-Byte Write (up to 264 registers)
  - Single-Byte Read
  - Multiple-Byte Read (up to 264 registers)
- **I2C Clock Stretching:** Required.
- **I2C Read Timeout:** Should be set to twice the value of `CFG_MB_TIMEOUT`. By default, configure your I2C timeout to 2000 ms.
- **Overreading and Overwriting the memory:**
  - If a write operation starts at a valid memory address (0 to 263) and continues past the last valid address, it will roll over to address 0.
  - Starting a write operation to an invalid memory address (greater than 263) will result in a NACK and data will be discarded.
  - If a read operation starts at a valid memory address (0 to 263) and continues past the last valid address, it will roll over to address 0.
  - Starting a read operation at an invalid memory address (greater than 263) will return a value of 0xFFFF.

## 6.2. Read Operations

### 6.2.1. Single Word Read

Reading a single word is an action performed by the Microcontroller (I2C-Master) to access any register within the IS4320 memory (I2C-Slave), regardless of the last read or written position. To perform this action, the microcontroller must first load the address of the IS4320 register to be read into the IS4320's internal Pointer Register. Once the address is set, the microcontroller can retrieve the data from the specified register.

To initiate the Single Word Read operation, the microcontroller begins by pulling down the SDA while the SCL is high to create a Start Condition. It then sends the IS4320 I2C Device Address 20 (0x14) with

the R/W bit set to '0' (write). Upon receiving the device address, the IS4320 acknowledges it. Subsequently, the microcontroller sends the two bytes of the Pointer Register address: the most significant byte first, followed by the less significant byte, each acknowledged by the IS4320. This sets the address of the next word to be read in the Pointer Register.

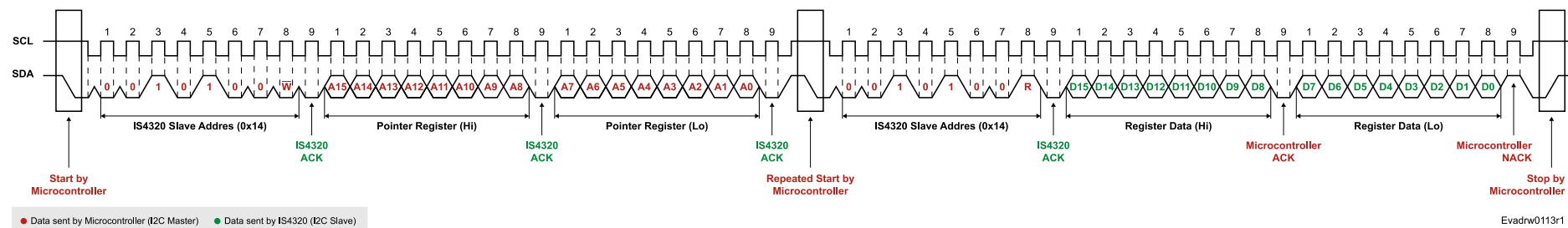
Next, the content of the Pointer Register, which is a word (two bytes), needs to be read.

The microcontroller generates a Repeated Start Condition, followed by the IS4320 I2C Device Address 20 (0x14) with the R/W bit set to '1' (read),

instructing the IS4320 to retrieve data. The IS4320 acknowledges and responds with the most significant byte, which the microcontroller acknowledges. Then, the IS4320 sends the less significant byte, which the microcontroller does not acknowledge (NACK). Finally, the microcontroller issues a Stop Condition by raising the SDA line while the SCL is high.

#### Invalid Memory Addressing

The valid memory range of the IS4320 goes from addresses 0 to 263. If a Read Operation is performed with a Pointer Register higher than 263, the read result will be 0xFFFF.



## 6.2.2. Multiple Word Read

Multiple Word Read functions similarly to Single Word Read but can read a block of up to 264 registers in a single operation. Remember, the registers are 16-bit words consisting of 2 bytes, so the number of registers retrieved should always be even.

To perform a Multiple Word Read, follow the same procedure as for a Single Word Read until the first data word is received. After receiving the first word, instead of generating a Not Acknowledge (NACK), the microcontroller should continue acknowledging

(ACK) each received data byte from the IS4320 for as many words as it intends to read. To conclude the read operation, after reading the last data word, the microcontroller should generate a Not Acknowledge (NACK) and a Stop Condition.

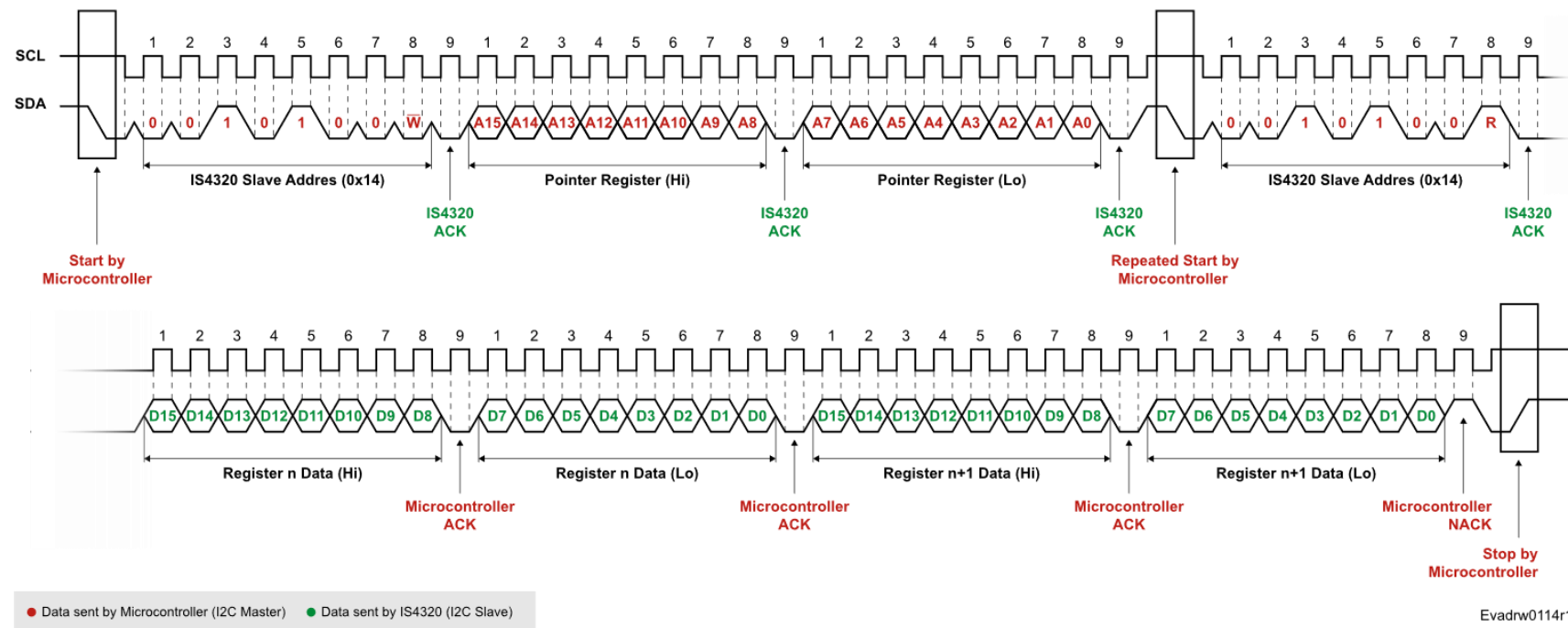
With each word read, the Pointer Register increments by one.

### Invalid Memory Addressing

The valid memory range of the IS4320 goes from addresses 0 to 263.

If the Read Operation is performed with a Pointer Register within the valid memory range (0 to 263), but the data retrieval extends beyond register 263, a rollover to position 0 will occur. For example, the value of register 264 will correspond to the content of register 0.

If a Read Operation is performed with a Pointer Register value higher than 263, the read result will be 0xFFFF.



## 6.3. Write Operations

### 6.3.1. Single Word Write

Writing a single word is an action performed by the Microcontroller (I2C-Master) to write data to any register within the IS4320 memory (I2C-Slave), regardless of the last read or written position. To perform this action, the Microcontroller must first load the address of the IS4320 register to be written into the IS4320's internal Pointer Register. Once the address is set, the Microcontroller can send the data to be stored.

To initiate the Single Word Write operation, the Microcontroller begins by pulling down the SDA line while the SCL line is high, creating a Start Condition.

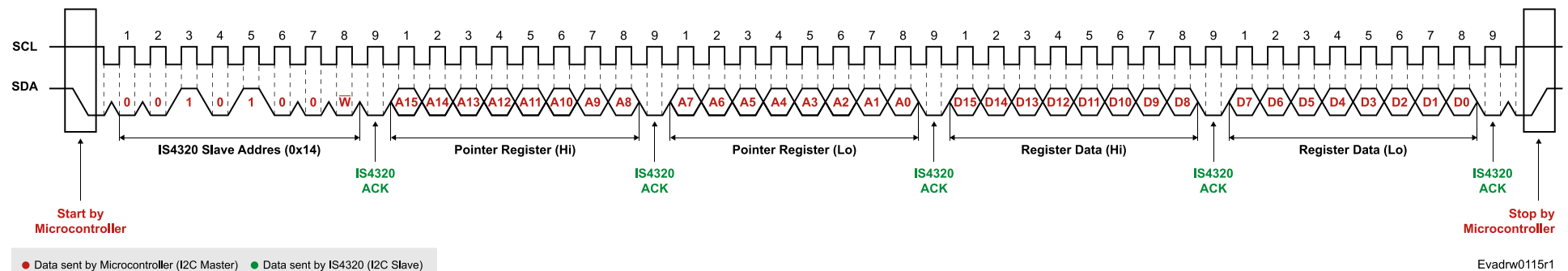
It then sends the IS4320 I2C Device Address 20 (0x14) with the R/W bit set to '0' (write). Upon receiving the device address, the IS4320 acknowledges it. Subsequently, the Microcontroller sends the two bytes of the Pointer Register address: the most significant byte first, followed by the least significant byte, each acknowledged by the IS4320. This sets the address of the next word to be written in the Pointer Register, preparing the device to receive the data.

The Microcontroller then sends the most significant byte of the word to be written first, which the IS4320

acknowledges. The Microcontroller follows by sending the least significant byte of the word, which the IS4320 also acknowledges. Finally, the Microcontroller issues a Stop Condition by raising the SDA line while the SCL line is high.

#### Invalid Memory Addressing

The valid memory range of the IS4320 goes from addresses 0 to 263. If a Write Operation is performed with a Pointer Register higher than 263, the IS4320 will answer with a NACK on the first received byte of the word.



## 6.3.2. Multiple Word Write

A Multiple Word Write performs a similar operation to a Single Word Write, but instead of writing to only one register, it can write to a block of up to 264 registers in a single operation.

To perform a Multiple Word Write, follow the same procedure as for a Single Word Write until the first data word is received. After receiving the first word, instead of generating a Stop Condition, the Microcontroller should continue sending data words. To conclude the write operation, after sending the last data word, the Microcontroller should generate a Stop Condition.

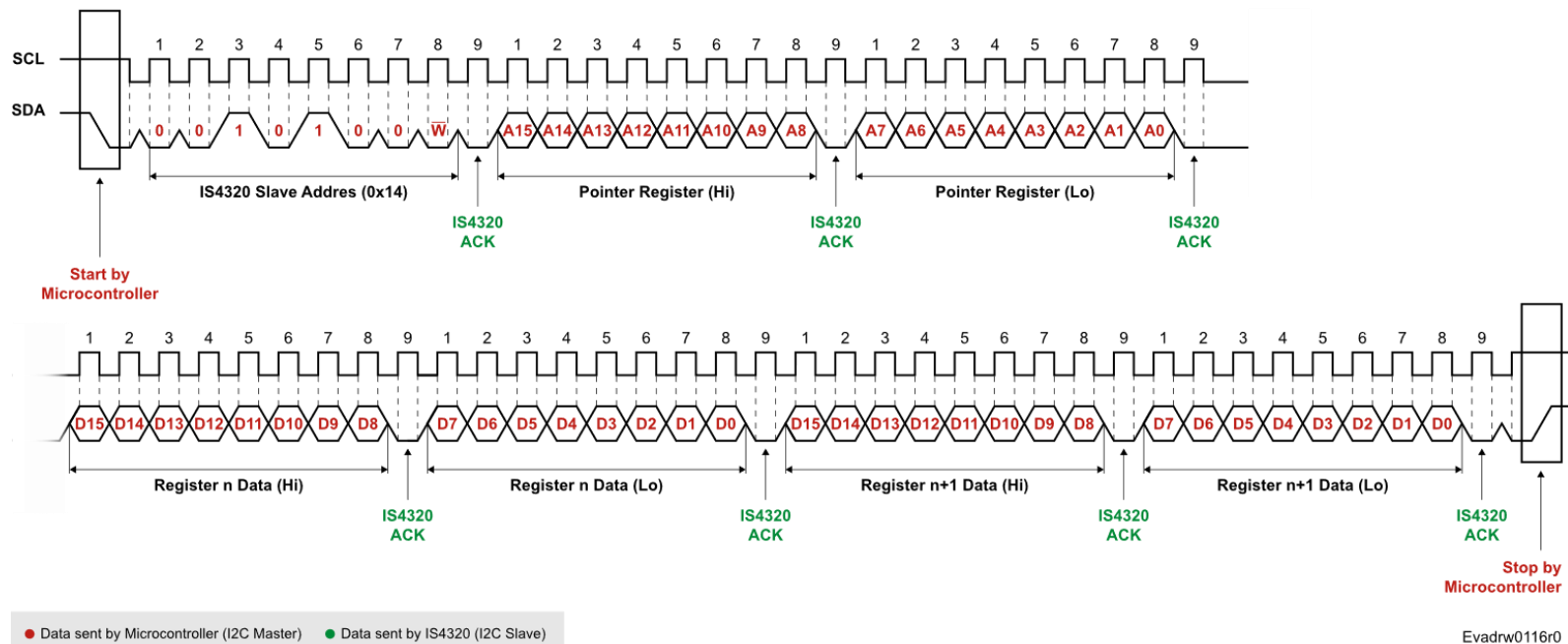
With each word written, the Pointer Register increments by one.

### Invalid Memory Addressing

The valid memory range of the IS4320 goes from addresses 0 to 263.

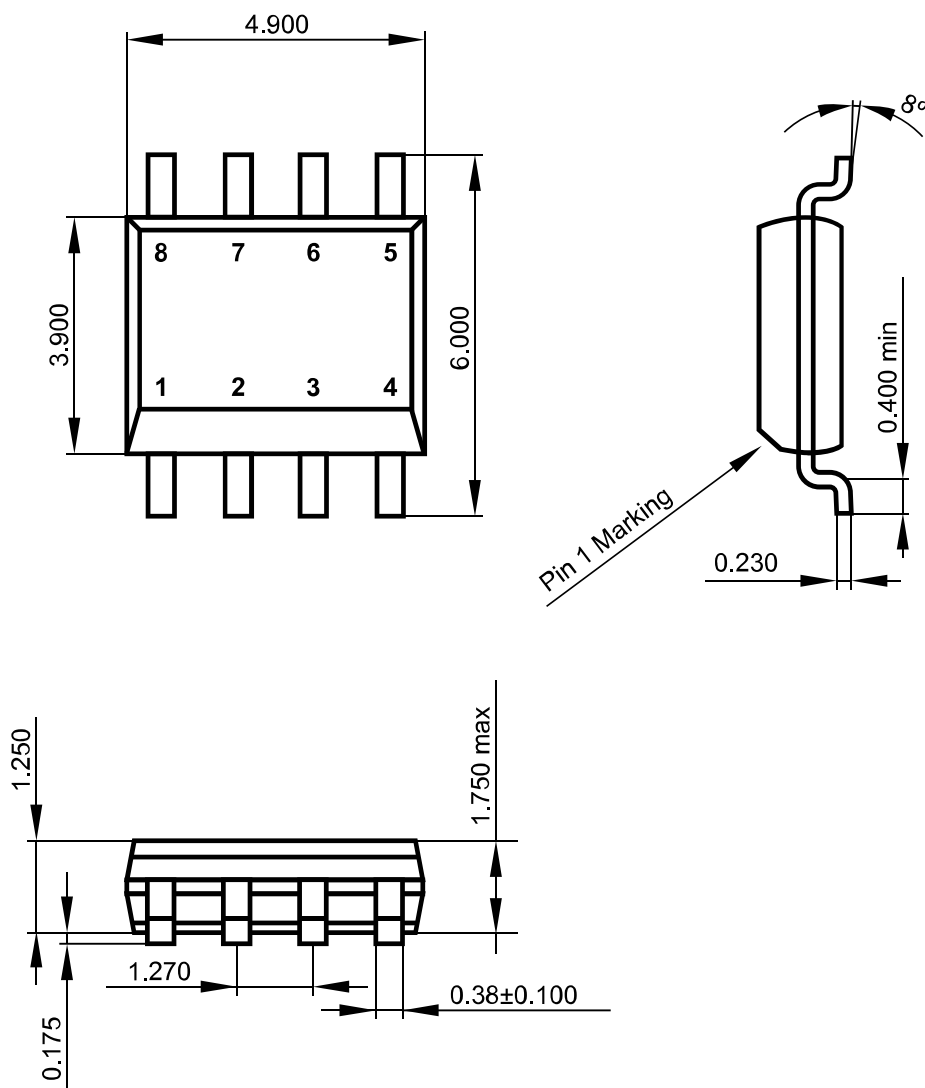
If a Write Operation is performed with a Pointer Register within the valid memory range (0 to 263) but exceeds the last memory register (263), a rollover to position 0 will occur. For example, writing a value to register 264 will result in writing the value to register 0.

If a Write Operation is performed with a Pointer Register higher than 263, the IS4320 will answer with a NACK on the first received byte of the word.



## 7. Mechanical

## SO8N Package



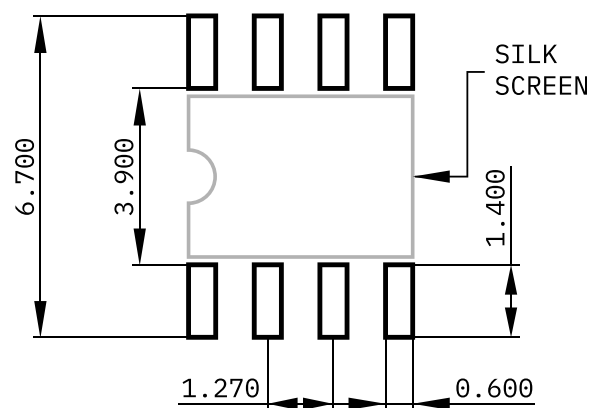
Units: millimeters

**Notes:**

This drawing is for general information only.  
Drawing not to scale.

Evadrw0033A



**S08N Recommended Footprint****Units: millimeters****Notes:**

This drawing is for general information only.  
Drawing not to scale.

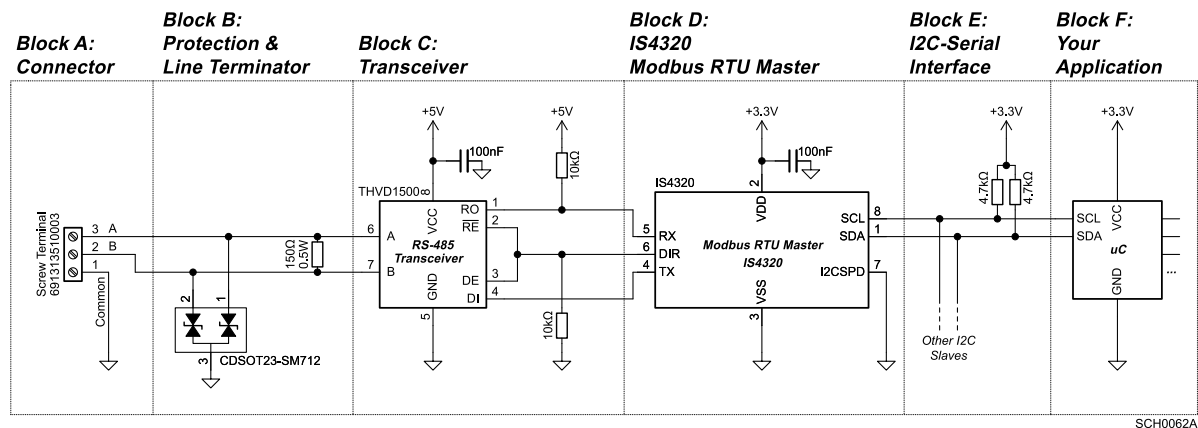
Evadrw0025r2

## 8. Hardware Examples

The following chapter represents an application design example for explanation proposals and is not part of the product standard. The customer must design his own solution, choose its most appropriate components and validate the final product according to the legislation and the Modbus specifications.

### 8.1. RS485 Example

This example shows the design of a Modbus over Serial Line working in RS485.



#### Block A: Connector

Typical Modbus Serial Line connectors include Screw Terminals, RJ45, and D-Sub 9-pin (commonly known as DB9), among others. The device-side connector must be female, while the cable-side connector must be male.

The recommended connector is RJ45, but in the schematic, a screw terminal is used for simplicity. When selecting a connector, always choose the shielded version if available. RJ45 and DB9 connectors typically come with shielded options, while terminal blocks usually do not.

On the cable-side connector, make sure to connect the cable shield to the connector shield to ensure proper electrical continuity across all cable shields on the bus.

All cable shields should be connected to Common and Protective Ground at a single point for the entire bus, ideally at the master device.

In the example, the connector has three positions: A, B, and Common. A and B are the differential lines for the transceiver, while Common serves as the reference point for the A and B signals. Common must be connected to the GND of your circuit.

Optionally, power can be supplied to your system through the Modbus connector. In this case, a four-position connector would be used for A, B, Common, and Power. In that case, the Common serves as the reference for A and B signals as well as the return path for Power. The voltage should be within the 5V to 24V range.

#### Block B: Protection & Line Polarization

##### Protection

The protection stage is influenced by several factors, including the intrinsic robustness and protection features of the transceiver, the potential harshness of the fieldbus environment, the product's budget, and its required reliability, among other considerations. Refer to your transceiver's documentation to determine the appropriate protection requirements.

In the schematic, a bidirectional 400-W transient suppressor diodes are used to protect against surge transients.

##### Line Terminator

Reflections on a transmission line occur whenever there's an impedance mismatch that a traveling wave encounters as it moves along the line. To reduce reflections at the ends of an RS485 cable, a line termination should be placed near each end of the bus. Terminating both ends is crucial because signals travel in both directions, but no more than two terminators should be used on the same bus. The line terminator connects across the balanced lines (cable A and B) and is typically a 150 Ω resistor rated at 0.5 W.

##### Line Polarization

Line polarization is not shown in the example, but it is explained because it is required for some transceivers.

Line Polarization is the process of biasing the RS485 bus to a known state by pulling signal A down and pulling signal B to 5V using resistors in the range of

450 to 650Ω. This ensures that the bus has a defined idle state.

When there is no data activity on an RS-485 balanced pair, the lines are not actively driven and are therefore susceptible to external noise or interference. To ensure that the transceiver remains in a stable state when no data signal is present, some transceivers require a biasing circuit. However, not all transceivers need this.

When selecting your transceiver, confirm in the datasheet whether line polarization is necessary or not. If it is necessary, you must document it in the product guide.

If polarization is needed, it should **ONLY** be implemented at one location on the bus, typically at the master device.

Bus polarization is a good technic to increase the resistance of the bus to external noise or interferences. However, it has the drawback of significantly reducing the number of devices that can support the bus.

## Block C: Transceiver

Modbus over Serial Line typically employs the RS485 electrical interface, which uses a transceiver to adapt RS485 fieldbus voltage levels to TTL voltage levels for the IS4320. Other electrical interfaces such as RS422 or RS232 can also be utilized. In the example, RS485 is being used.

A pull-down resistor on DE and RE will keep the transceiver in 'receiver' state by default, ensuring it does not disturb the fieldbus. Pull-up resistor on RO will keep the RX line clear for the microcontroller.

Using a 5V transceiver rather than a 3.3V is a good technic to increase the resistance of the bus to

external noise or interferences. 5V transceivers can be used directly with the IS4320 since TX, RX and DIR pins are 5V tolerant.

## Block D: IS4320 Modbus RTU Slave

The IS4320 is very simple to integrate into your design.

A decoupling capacitor should be placed on the power pins (VDD and VSS). It is recommended to use a 100 nF, 10-25 V low-ESR ceramic capacitor.

The I2CSPD pin defines the I2C speed. Connect this pin to GND for a speed of 100 kHz. For 400 kHz, it should be pulled to 1.65 V, which is half of 3.3 V. This can be achieved with a simple resistor voltage divider using 3.3 V and GND. For 1 MHz, the pin must be connected to 3.3 V. The maximum allowed voltage on this pin is 4 V.

## Block E: I2C-Serial Interface

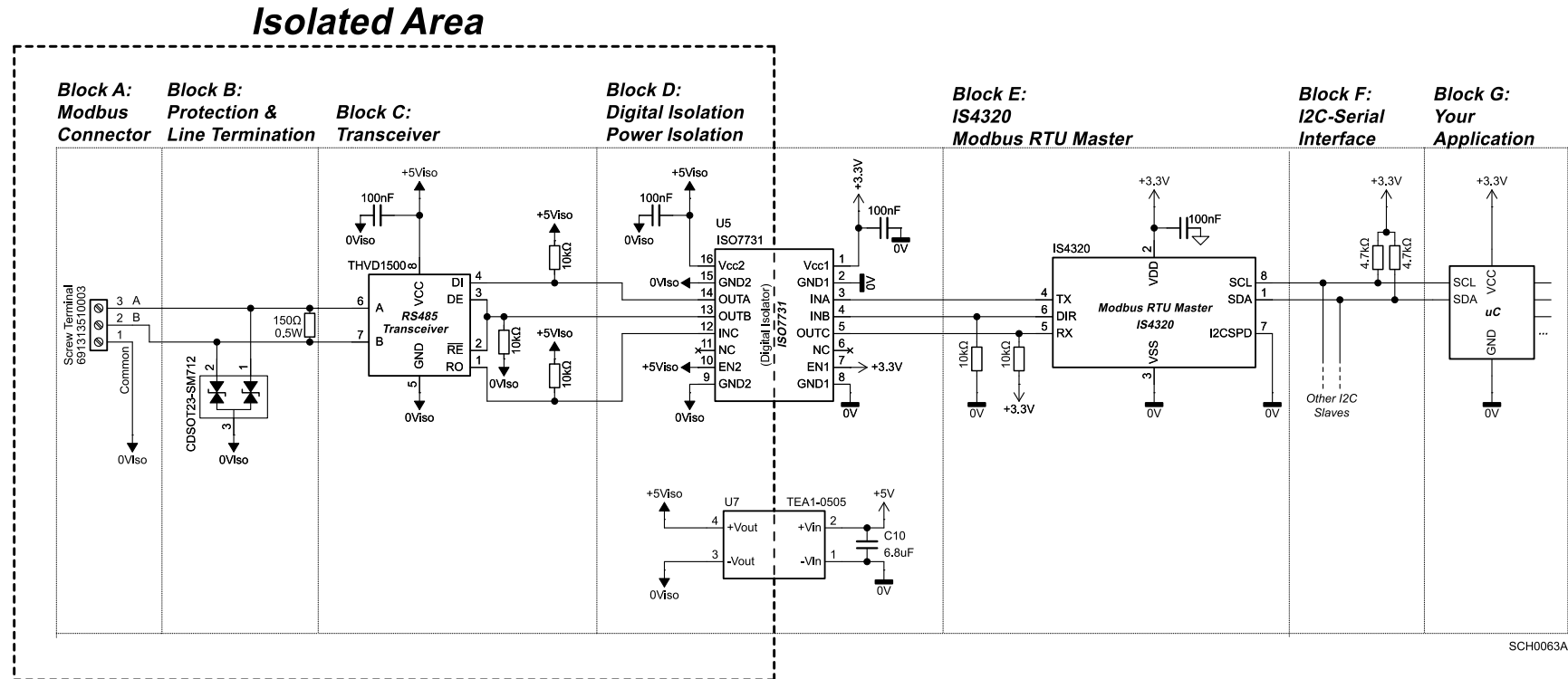
For proper operation of the I2C-Serial Interface, pull-up resistors to 3.3 V or 5V are necessary. Typical resistor values are 4.7 kΩ for Standard Mode (100 kHz) and 2 kΩ for both Fast Mode (400 kHz) and Fast Mode Plus (1 MHz).

## Block F: Your Application

Here is the rest of your product design. Typically, a microcontroller interfaces with the IS4320, but a microprocessor or a single-board computer, such as a Raspberry Pi, can also be used as long as they are equipped with an I2C-Serial Interface.

## 8.2. Isolated RS485 Example

This example shows the design of a Modbus over Serial Line working in isolated RS485.



## 8.3. RS232 Example

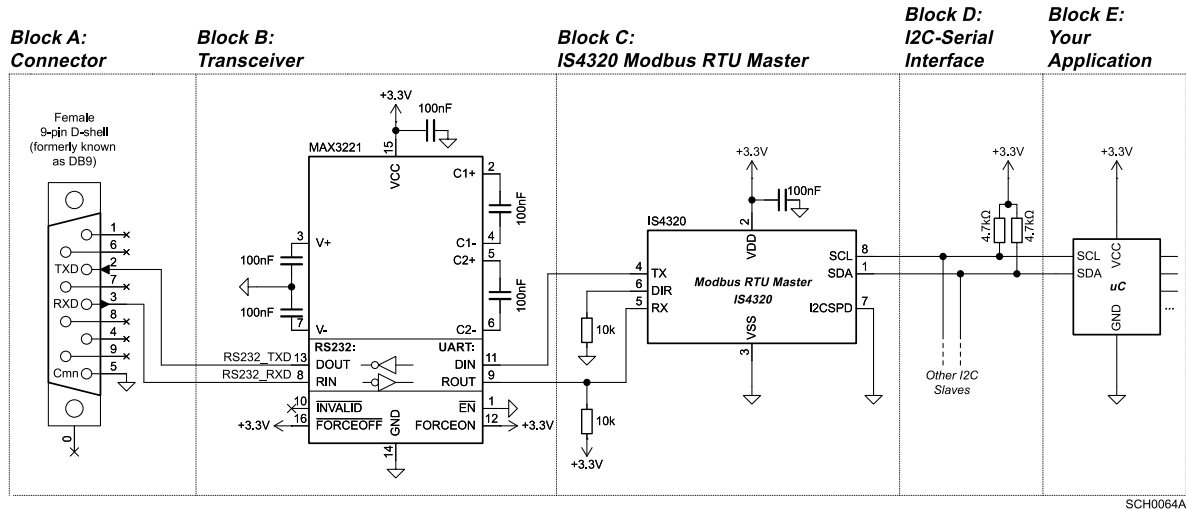
This example shows the design of a Modbus over Serial Line working in RS232.

You can use any RS232 transceiver together with the IS4320. In the figure below the MAX3221 is used.

The required signals on the connector are TXD, RXD and Common.

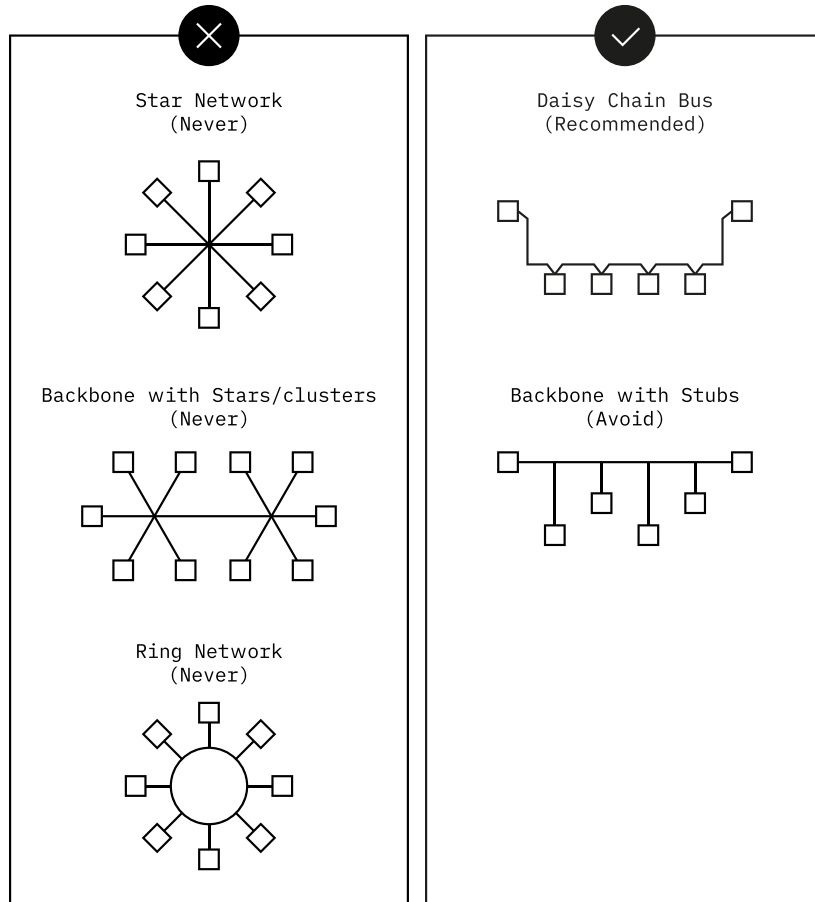
Use short cables, it's recommended to use less than 20 m.

Cable pinout must be direct pin to pin and never use crossed cables.

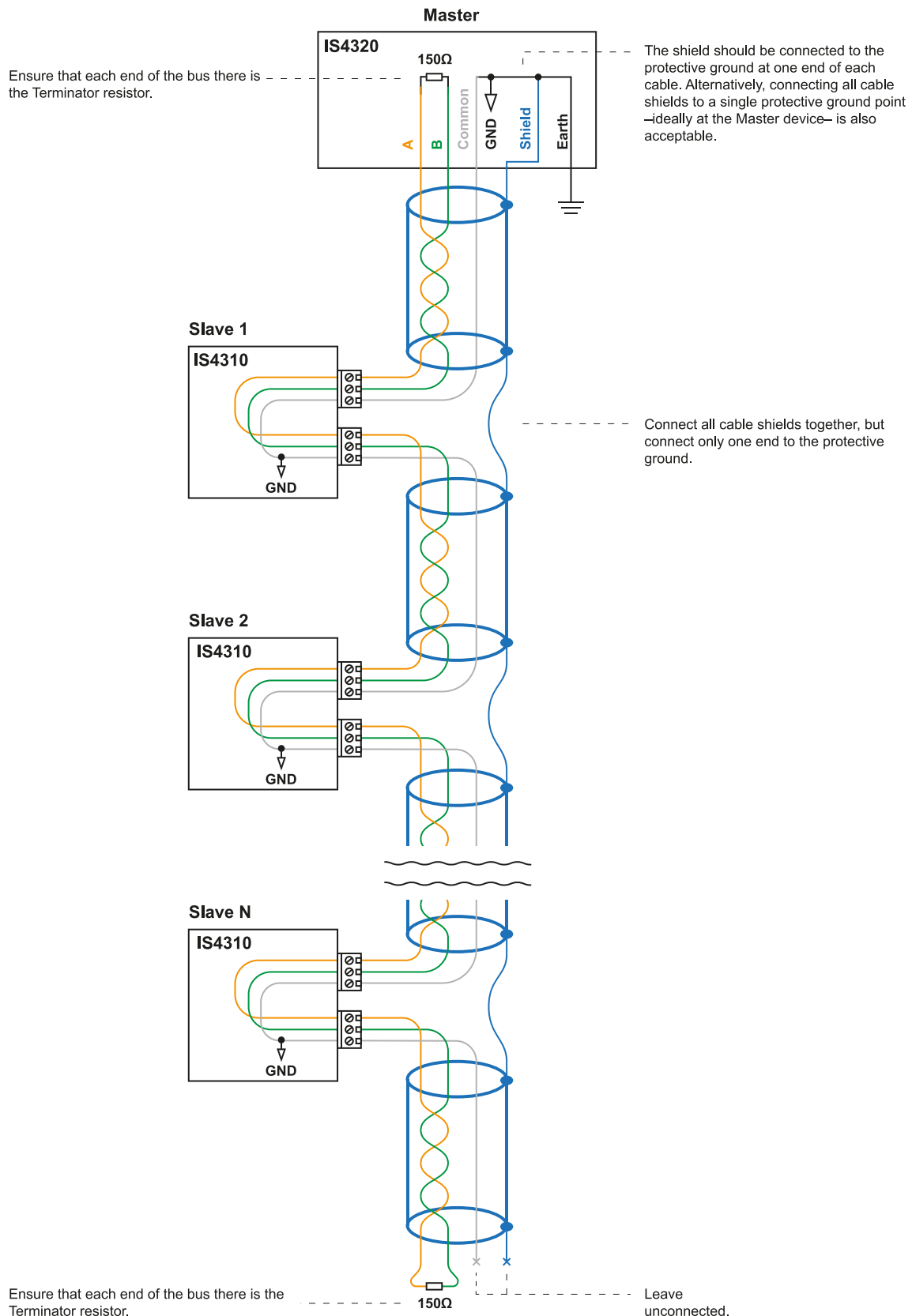


## 8.4. Bus Topology

In an RS485 setup without a repeater, a single trunk cable runs through the system, with devices connected in a daisy-chain manner. Short cables derivations (stubs) are also allowed but not recommended. Keep the derivation distance as short as possible. Other topologies are not allowed.



## 8.5. Cable Wiring



## 9. Firmware Examples

The following chapter presents firmware examples for different platforms for demonstration purposes only and is not part of the product standard. Customers must develop their own firmware, perform all necessary tests, and validate the final product according to applicable regulations and Modbus specifications.

### 9.1. Arduino Example

Coding with Arduino for the IS4320 is very simple. It does not require any INACKS-specific library, just the standard Arduino Wire (I2C) functions: `Wire.write()`, `Wire.read()` and related.

This Arduino example is based on the Arduino UNO board, and the [Kappa4320Ard](#) Evaluation Board, which features the IS4320 chip. The Kappa4320Ard has an Arduino form factor and directly fits into the Arduino UNO board, so no additional connections between the Arduino and the IS4320 are required, making it very easy to test the example.

The project demonstrates how to use the Arduino to communicate with the IS4320 over I2C. In the example, the Arduino instructs the IS4320 (Modbus Master) to read Holding Register 0 using the Function Code 3 from a Modbus Slave, and prints the result via Serial Port.

To test the example, you will need a Modbus Slave. You can use the [pyModSlave](#) software to create a Modbus Slave. Configure the Slave with these values: Slave Address 1, 19200 baud, Even parity, and 1 Stop bit.

You can download this Arduino project at: <https://github.com/inacks/ISXMPL4320ex5>

```
#include <Wire.h>

// IS4320 Memory Map Addresses:
#define CFG_MBBDR      0
#define CFG_MBPAPAR    1
#define CFG_MBSTP      2
#define REQ_EXECUTE    6
#define REQ_SLAVE      7
#define REQ_FC         8
#define REQ_STARTING    9
#define REQ_QTY       10
#define RES_STATUS     138
#define RES_DATA1      139

void writeIS4320Register(uint16_t registerAddressToWrite, uint16_t value) {
    Wire.beginTransmission(0x14); // I2C address of the IS4320.

    // Send the 16-bit Holding Register address (2 bytes).
    Wire.write((registerAddressToWrite >> 8) & 0xFF); // High byte.
    Wire.write(registerAddressToWrite & 0xFF);        // Low byte.

    // Send the 16-bit data (2 bytes).
    Wire.write((value >> 8) & 0xFF); // High byte.
    Wire.write(value & 0xFF);        // Low byte.

    Wire.endTransmission();
}

uint16_t readIS4320Register(uint16_t holdingRegisterAddress) {
    uint16_t result; // Variable to store the read data.

    Wire.beginTransmission(0x14); // I2C address of the IS4320.

    // Send the 16-bit Holding Register address (2 bytes).
    Wire.write((holdingRegisterAddress >> 8) & 0xFF); // High byte.
    Wire.write(holdingRegisterAddress & 0xFF);        // Low byte.
    Wire.endTransmission(false); // Send a repeated start condition.

    // Request 2 bytes from the IS4320 (a full 16-bit Holding Register).
    Wire.requestFrom(0x14, 2);
    result = Wire.read(); // Read high byte.
    result <<= 8;         // Shift to make space for low byte.
    result |= Wire.read(); // Read and append low byte.

    return result; // Return the full 16-bit value.
}
```



```
}

void setup() {
  Wire.begin();           // Initialize the I2C interface.
  Serial.begin(9600);      // Initialize the serial port for debug printing.

  /* First, configure the Modbus communication parameters
   * to match the slave characteristics.
   * This only needs to be done once before communicating
   * with a Modbus device that has a different configuration. */

  // Set baud rate to 19200:
  uint16_t baudRate = 113;
  writeIS4320Register(CFG_MBBDR, baudRate);

  // Set parity to Even:
  uint16_t parityBit = 122;
  writeIS4320Register(CFG_MBPAP, parityBit);

  // Set stop bits to 1:
  uint16_t stopBits = 131;
  writeIS4320Register(CFG_MBSTP, stopBits);
}

void loop() {
  /* Example: Read Holding Register 0 and print its value
   * to the PC via the Serial port */

  // Set the Modbus Slave ID:
  uint16_t modbusSlaveId = 1;
  writeIS4320Register(REQ_SLAVE, modbusSlaveId);

  // Set the Function Code. For reading Holding Registers, use FC = 3:
  uint16_t functionCode = 3;
  writeIS4320Register(REQ_FC, functionCode);

  // Set the Starting Register address. Here we want to read Holding Register 0:
  uint16_t startingRegister = 0;
  writeIS4320Register(REQ_STARTING, startingRegister);

  // Set the number of Holding Registers to read (minimum is 1):
  uint16_t quantity = 1;
  writeIS4320Register(REQ_QTY, quantity);

  // Send the Request to the Modbus Slave:
  writeIS4320Register(REQ_EXECUTE, 1);

  // Read back the result/status of the operation:
  uint16_t status = readIS4320Register(RES_STATUS);

  if (status == 2) {
    // OK! The Request was sent and a Response was received
    // Read the data:
    uint16_t holdingRegisterData = readIS4320Register(RES_DATA1);
    Serial.print("Holding Register 0 content = ");
    Serial.println(holdingRegisterData);
  }
  else if (status == 3) {
    // Timeout: no Response from the Modbus Slave
    Serial.println("Timeout, the slave did not answer.");
    Serial.println("Did you start the pyModSlave? Did you set its configuration to: Slave Address 1, 19200 baud, Even parity, and 1 Stop bit?");
  }
  else if (status == 4) {
    Serial.println("Broadcast message sent.");
  }
  else if (status == 5) {
    Serial.println("You configured wrongly the REQ_SLAVE register.");
  }
  else if (status == 6) {
    Serial.println("You configured wrongly the REQ_FC register.");
  }
  else if (status == 7) {

```

```
        Serial.println("You configured wrongly the REQ_QTY register.");
    }
    else if (status == 8) {
        Serial.println("There was a Frame Error.");
    }
    else if (status == 201) {
        Serial.println("Modbus Exception Code 1: Illegal Function.");
    }
    else if (status == 202) {
        Serial.println("Modbus Exception Code 2: Illegal Data Address.");
        Serial.println("If using pyModSlave, make sure the 'Start Addr' parameter is set to 0.");
    }
    else if (status == 203) {
        Serial.println("Modbus Exception Code 3: Illegal Data Value.");
    }
    else if (status == 204) {
        Serial.println("Modbus Exception Code 4: Server Device Failure.");
    }
    else {
        Serial.println("Unkown Error");
    }

    // Add a delay to give the Modbus Slave time to respond and avoid stressing it:
    delay(1000);
}
```

## 9.2. STM32 Example

Coding with STM32 for the IS4320 is very simple. It does not require any INACKS-specific library, just the standard HAL I2C functions: `HAL_I2C_Mem_Read()` and `HAL_I2C_Mem_Write()`.

This STM32 CubeIDE project is based on the [Nucleo-C071](#) evaluation board from ST, which features an STM32C071RBT microcontroller, and the [Kappa4320Ard](#) Evaluation Board, which features the IS4320 chip. The Kappa4320Ard has an Arduino form factor and directly fits into the Nucleo-C071 board, so no additional connections between the Nucleo and the IS4320 are required, making it very easy to test the example.

The project demonstrates how to use the STM32 microcontroller to communicate with the IS4320 over I2C. In this example, the microcontroller instructs the IS4320 to read Holding Register 0 using the Function Code 3 from a Modbus Slave, and prints the result via the STLink Virtual COM Port. You can view the output using any serial terminal software, such as [CoolTerm](#).

To test the example, you will need a Modbus Slave. You can use the [pyModSlave](#) software to create a Modbus Slave. Configure the Slave with these values: Slave Address 1, 19200 baud, Even parity, and 1 Stop bit.

For clarity purposes, the code below excludes all extra HAL code and only shows the parts relevant to the IS4320.

You can download the full STM32 project example at: <https://github.com/inacks/ISXMPL4320ex6>

```
#include <stdio.h>

// IS4320 Memory Map Addresses:
#define CFG_MBBDR      0
#define CFG_MBPAR      1
#define CFG_MBSTP      2
#define REQ_EXECUTE    6
#define REQ_SLAVE      7
#define REQ_FC          8
#define REQ_STARTING    9
#define REQ_QTY        10
#define RES_STATUS     138
#define RES_DATA1      139

/**
 * @brief Reads a single register from the IS4320 memory map.
 * @param registerAddressToRead: Address in the IS4320 memory map to be read.
 * @retval Value stored at the registerAddressToRead register address.
 */
uint16_t readIS4320Register(uint16_t registerAddressToRead) {
    uint8_t IS4320_I2C_Chip_Address; // This variable stores the I2C chip address of the IS4320.
    IS4320_I2C_Chip_Address = 0x14; // The IS4320's I2C address is 0x14.
    // The STM32 HAL I2C library requires the I2C address to be shifted left by one bit.
    // Let's shift the IS4320 I2C address accordingly:
```

```
IS4320_I2C_Chip_Address = IS4320_I2C_Chip_Address << 1;

// The following array will store the read data.
// Since each holding register is 16 bits long, reading one register requires reading 2 bytes.
uint8_t readResultArray[2];

// This variable will contain the final result:
uint16_t readResult;

/*
 * This is the HAL function to read from an I2C memory device. The IS4320 is designed to operate as an I2C memory.
 *
 * HAL_I2C_Mem_Read parameters explained:
 * 1. &hi2c1: This is the name of the I2C that you're using. You set this in the CubeMX. Don't forget the '&'.
 * 2. IS4320_I2C_Chip_Address: The I2C address of the IS4320 (must be left-shifted).
 * 3. registerAddressToRead: The holding register address to read from the IS4320.
 * 4. I2C_MEMADD_SIZE_16BIT: You must indicate the memory addressing size. The IS4320 memory addressing is 16-bits.
 *    This keyword is an internal constant of HAL libraries. Just write it.
 * 5. readResultArray: An 8-bit array where the HAL stores the read data.
 * 6. 2: The number of bytes to read. Since one holding register is 16 bits, we need to read 2 bytes.
 * 7. 1500: Timeout in milliseconds. IMPORTANT, this timeout must be higher than the timeout specified in CFG_MB_TIMEOUT.
 *    1500 is a good default value.
 */
HAL_I2C_Mem_Read(&hi2c1, IS4320_I2C_Chip_Address, registerAddressToRead, I2C_MEMADD_SIZE_16BIT, readResultArray, 2, 1500);

// Combine two bytes into a 16-bit result:
readResult = readResultArray[0];
readResult = readResult << 8;
readResult = readResult | readResultArray[1];

return readResult;
}

/**
 * @brief Reads a single register from the IS4320 memory map.
 * @param registerAddressToRead: Address in the IS4320 memory map to be read.
 * @retval Value stored at the registerAddressToRead register address.
 */
void writeIS4320Register(uint16_t registerAddressToWrite, uint16_t value) {
    uint8_t IS4320_I2C_Chip_Address; // I2C address of IS4320 chip (7-bit).
    IS4320_I2C_Chip_Address = 0x14; // IS4320 I2C address is 0x14 (7-bit).
    // STM32 HAL expects 8-bit address, so shift left by 1:
    IS4320_I2C_Chip_Address = IS4320_I2C_Chip_Address << 1;

    // The HAL library to write I2C memories needs the data to be in a uint8_t array.
    // So, lets put our uint16_t data into a 2 registers uint8_t array.
    uint8_t writeValuesArray[2];
    writeValuesArray[0] = (uint8_t) (value >> 8);
    writeValuesArray[1] = (uint8_t) value;
}
```

```
/*
 * This is the HAL function to write to an I2C memory device. To be simple and easy to use, the IS4320 is designed to operate as an I2C
memory.
 *
 * HAL_I2C_Mem_Write parameters explained:
 * 1. &hi2c1: This is the name of the I2C that you're using. You set this in the CubeMX. Don't forget the '&'.
 * 2. IS4320_I2C_Chip_Address: The I2C address of the IS4320 (must be left-shifted).
 * 3. registerAdressToWrite: The holding register address of the IS4320 we want to write to.
 * 4. I2C_MEMADD_SIZE_16BIT: You must indicate the memory addressing size. The IS4320 memory addressing is 16-bits.
 * This keyword is an internal constant of HAL libraries. Just write it.
 * 5. writeValuesArray: An 8-bit array where we store the data to be written by the HAL function.
 * 6. 2: The number of bytes to write. Since one holding register is 16 bits, we need to write 2 bytes.
 * 7. 1500: IMPORTANT, this timeout must be higher than the timeout specified in CFG_MB_TIMEOUT.
 *    1500 is a good default value.
 */
HAL_I2C_Mem_Write(&hi2c1, IS4320_I2C_Chip_Address, registerAdressToWrite, I2C_MEMADD_SIZE_16BIT, writeValuesArray, 2, 1500);
}

// This function is required to make printf work over the Serial port.
// It is never called directly in the code - printf internally uses it
int _write(int file, char *ptr, int len) {
    HAL_UART_Transmit(&huart2, (uint8_t*)ptr, len, HAL_MAX_DELAY);
    return len;
}
```

Main and while(1):

```
int main(void) {
    /* First, configure the Modbus communication parameters
     * to match the slave characteristics.
     * This only needs to be done once before communicating
     * with a Modbus device that has a different configuration. */

    // Set baud rate to 19200:
    uint16_t baudRate = 113;
    writeIS4320Register(CFG_MBBDR, baudRate);

    // Set parity to Even:
    uint16_t parityBit = 122;
    writeIS4320Register(CFG_MBPAR, parityBit);

    // Set stop bits to 1:
    uint16_t stopBits = 131;
    writeIS4320Register(CFG_MBSTP, stopBits);

    while (1) {
        /* Example: Read Holding Register 0 and print its value
         * to the PC via the Serial port */

        // Set the Modbus Slave ID:
        uint16_t modbusSlaveId = 1;
        writeIS4320Register(REQ_SLAVE, modbusSlaveId);

        // Set the Function Code. For reading Holding Registers, use FC = 3:
        uint16_t functionCode = 3;
        writeIS4320Register(REQ_FC, functionCode);

        // Set the Starting Register address. Here we want to read Holding Register 0:
        uint16_t startingRegister = 0;
        writeIS4320Register(REQ_STARTING, startingRegister);

        // Set the number of Holding Registers to read (minimum is 1):
        uint16_t quantity = 1;
        writeIS4320Register(REQ_QTY, quantity);

        // Send the Request to the Modbus Slave:
        writeIS4320Register(REQ_EXECUTE, 1);

        // Read back the result/status of the operation:
        uint16_t status = readIS4320Register(RES_STATUS);

        if (status == 2) {
            // OK! The Request was sent and a Response was received
            // Read the data:
            uint16_t holdingRegisterData = readIS4320Register(RES_DATA1);
            printf("Holding Register 0 content = %d\r\n", holdingRegisterData);
        }
        else if (status == 3) {
            // Timeout: no Response from the Modbus Slave
            printf("Timeout, the slave did not answer.\n");
        }
        else if (status == 4) {
            printf("Broadcast message sent.\n");
        }
        else if (status == 5) {
            printf("You configured wrongly the REQ_SLAVE register.\n");
        }
        else if (status == 6) {
            printf("You configured wrongly the REQ_FC register.\n");
        }
        else if (status == 7) {
            printf("You configured wrongly the REQ_QTY register.\n");
        }
        else if (status == 8) {
            printf("There was a Frame Error.\n");
        }
        else if (status == 201) {
            printf("Modbus Exception Code 1: Illegal Function.\n");
        }
        else if (status == 202) {
            printf("Modbus Exception Code 2: Illegal Data Address.\n");
        }
    }
}
```

```
}  
else if (status == 203) {  
    printf("Modbus Exception Code 3: Illegal Data Value.\n");  
}  
else if (status == 204) {  
    printf("Modbus Exception Code 4: Server Device Failure.\n");  
}  
else {  
    printf("Unkown Error");  
}  
  
// Add a delay to give the Modbus Slave time to respond and avoid stressing it:  
HAL_Delay(1000);  
}
```

### 9.3. Raspberry Pi Example

Coding with Raspberry Pi for the IS4320 is very simple. It does not require any INACKS-specific library, just the standard Python `smbus2` library for I2C: `i2c_msg.write()`, `i2c_msg.read()` and related.

This Raspberry Pi example is written in Python and based on the Raspberry Pi Model B and the [Kappa4320Rasp](#) Evaluation Board, which features the IS4320 chip. The Kappa4320Rasp has a Raspberry Pi form factor and directly fits into it, requiring no additional connections, which makes testing the example very easy.

The example demonstrates how to use the Raspberry Pi to communicate with the IS4320 over I2C. In this example, the Raspberry Pi instructs the IS4320 to read Holding Register 0 using Function Code 3 from a Modbus Slave, and prints the result to the console.

To test the example, you will need a Modbus Slave. You can use the [pyModSlave](#) software to create a Modbus Slave. Configure the Slave with these values: Slave Address 1, 19200 baud, Even parity, and 1 Stop bit.

Attention: The IS4320 stretches the I2C signal while waiting for the Modbus Slave Response or its timeout. Since clock stretching is not officially supported, your code must handle the I2C exception that occurs when the IS4320 stretches the signal. Continue reading and catching the I2C exception while the IS4320 is stretching the bus.

You can download this Python project at: <https://github.com/inacks/ISXMPL4320ex4>

```
from smbus2 import SMBus, i2c_msg
import time

I2C_BUS = 1 #
DEVICE_ADDRESS = 0x14 # 7-bit I2C address of the IS4320

# IS4320 register map
CFG_MBBDR = 0
CFG_MBPAR = 1
CFG_MBSTP = 2
CFG_CHIP_ID = 4
CFG_CHIP_REV = 5
REQ_EXECUTE = 6
REQ_SLAVE = 7
REQ_FC = 8
REQ_STARTING = 9
REQ_QTY = 10
RES_STATUS = 138
RES_DATA1 = 139

def write_IS4320_register(start_register, data_word):
    """
    Write a 16-bit register to the IS4320 memory map.

    :param start_register: The 16-bit register address to start writing to.
    :param data_bytes: A list of bytes to write.
    """
    high_addr = (start_register >> 8) & 0xFF
    low_addr = start_register & 0xFF
    data_word_high = (data_word >> 8) & 0xFF
    data_word_low = data_word & 0xFF
    with SMBus(I2C_BUS) as bus:
        msg = i2c_msg.write(DEVICE_ADDRESS, [high_addr, low_addr, data_word_high,
        data_word_low])
        bus.i2c_rdwr(msg)

def read_IS4320_register(start_register):
    """
    Read a 16-bit value from the IS4320 memory map.

    :param start_register: 16-bit register address to read from
    :return: 16-bit integer value read (big-endian)
    """
    high_addr = (start_register >> 8) & 0xFF # High byte of register address
    low_addr = start_register & 0xFF # Low byte of register address
    try:
        with SMBus(I2C_BUS) as bus:
            # Write register address first to set internal pointer
            write_msg = i2c_msg.write(DEVICE_ADDRESS, [high_addr, low_addr])
            # Prepare to read 2 bytes from the device
```



```
        read_msg = i2c_msg.read(DEVICE_ADDRESS, 2)
        bus.i2c_rdwr(write_msg, read_msg)

        data = list(read_msg) # Read bytes as list of ints
        # Combine high and low bytes into 16-bit integer (big-endian)
        value = (data[0] << 8) | data[1]
        return value
    except IOError as e:
        return None # return None on failure

chipID = None
chipRev = None

# Detect the chip
chipID = None
chipRev = None

# Detect the chip
while True:
    chipID = read_IS4320_register(CFG_CHIP_ID)
    chipRev = read_IS4320_register(CFG_CHIP_REV)

    if chipID == 20:
        print("IS4320 Chip detected on I2C!")
        print("Chip ID:", chipID)
        print("Chip Rev:", chipRev)
        break

    print("ERROR: IS4320 Chip NOT detected on I2C!")
    time.sleep(1)

# First, configure the Modbus communication parameters
# to match the slave characteristics.
# This only needs to be done once before communicating
# with a Modbus device that has a different configuration.

# Set baud rate to 19200:
baudRate = 113
write_IS4320_register(CFG_MBBDR, baudRate)

# Set parity to Even:
parityBit = 122
write_IS4320_register(CFG_MBPAP, parityBit)

# Set stop bits to 1:
stopBits = 131
write_IS4320_register(CFG_MBSTP, stopBits)

while True:
    """Example: Read Holding Register 0 and print its value."""

    # Set the Modbus Slave ID:
    modbusSlaveId = 1
    write_IS4320_register(REQ_SLAVE, modbusSlaveId)

    # Set the Function Code. For reading Holding Registers, use FC = 3:
    functionCode = 3
    write_IS4320_register(REQ_FC, functionCode)

    # Set the Starting Register address. Here we want to read Holding Register 0:
    startingRegister = 0
    write_IS4320_register(REQ_STARTING, startingRegister)

    # Set the number of Holding Registers to read (minimum is 1):
    quantity = 1
    write_IS4320_register(REQ_QTY, quantity)

    # Send the Request to the Modbus Slave:
    write_IS4320_register(REQ_EXECUTE, 1)

    # Read back the result/status of the operation:
    status = None
    while (status == None):
        status = read_IS4320_register(RES_STATUS)

    if status == 2:
```

```
# OK! The Request was sent and a Response was received
holdingRegisterData = read_IS4320_register(RES_DATA1)
print("Holding Register 0 Content =", holdingRegisterData)

elif status == 3:
    # Timeout: no Response from the Modbus Slave
    print("Timeout, the slave did not answer.")
    print("Did you start the pyModSlave? Did you set its configuration to: "
          "Slave Address 1, 19200 baud, Even parity, and 1 Stop bit?")

elif status == 4:
    print("Broadcast message sent.")

elif status == 5:
    print("You configured wrongly the REQ_SLAVE register.")

elif status == 6:
    print("You configured wrongly the REQ_FC register.")

elif status == 7:
    print("You configured wrongly the REQ_QTY register.")

elif status == 8:
    print("There was a Frame Error.")

elif status == 201:
    print("Modbus Exception Code 1: Illegal Function.")

elif status == 202:
    print("Modbus Exception Code 2: Illegal Data Address.")
    print("If using pyModSlave, make sure the 'Start Addr' parameter is set to 0.")

elif status == 203:
    print("Modbus Exception Code 3: Illegal Data Value.")

elif status == 204:
    print("Modbus Exception Code 4: Server Device Failure.")

else:
    print("Unknown Error")

# Add a delay to give the Modbus Slave time to respond and avoid stressing it:
time.sleep(1)
```

## 10. PC/Mac Tools

The following third-party software options are provided for reference only. These applications are not developed, maintained, or endorsed by INACKS. We do not guarantee their functionality, compatibility, or compliance with the Modbus standard. Users should evaluate and choose software based on their specific needs.

### 10.1. Modbus Tools

#### USB Isolated Modbus Device



**Title:** DamnModIsoUsb

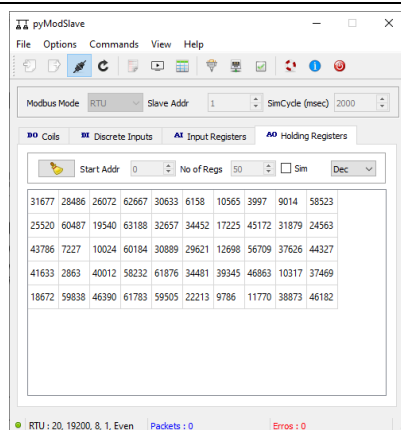
**Description:**

The DamnModIsoUsb is a USB to Modbus RTU interface that allows a PC or Mac to operate as either a Modbus RTU Master or Slave. It includes a robust USB connector and galvanic isolation to ensure safe and reliable operation. The device is compatible with Python, pyModSlave, QModMaster, and other software or scripts that support serial communication.

**Web:**

[www.inacks.com/damnmodisousb](http://www.inacks.com/damnmodisousb)

#### Modbus Slave Software



**Title:** pyModSlave

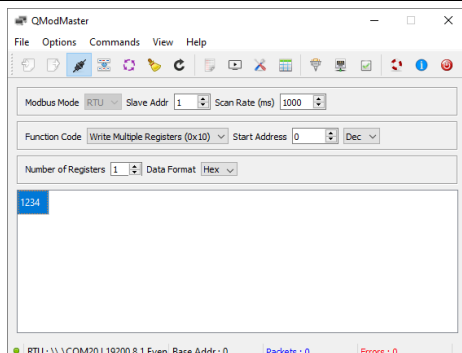
**Description:**

pyModSlave is a free python-based implementation of a Modbus slave application for simulation purposes. You can install the python module or use the precompiled (for Windows only) stand alone GUI (Qt based) utility (unzip and run). pyModSlave also includes a bus monitor for examining all traffic on the bus.

**Web:**

<https://sourceforge.net/projects/pymodslave/>

#### Modbus Master Software



**Title:** QModMaster

**Description:**

QModMaster is a free Qt-based implementation of a Modbus master application. A graphical user interface allows easy communication with Modbus RTU and TCP slaves. QModMaster also includes a bus monitor for examining all traffic on the bus.

**Web:**

<https://sourceforge.net/projects/qmodmaster/>

**Important:**

Go to menu *Options* → *Settings*, and set the parameter *Base Addr* to 0 to avoid confusions with the Modbus Registers Addresses.

## 10.2. I2C Tools

### USB I2C Master Device

---



**Title:** DamnI2cUsb

**Description:**

The DamnI2cUsb is a USB to I2C-Master interface that allows a PC or Mac to operate as an I2C Master Device, similar to a microcontroller.

Communication is handled over a serial link, making it compatible with Python and other scripting or programming languages. The tool allows reading from and writing to the IS4320 memory map, making it especially useful during the debugging stage.

**Web:**

[www.inacks.com/damni2cusb](http://www.inacks.com/damni2cusb)

## Content

IS4320: I2C Modbus RTU Master Stack .....	1	5.16. RES_DATAx Register .....	33
Product Selection Guide .....	2	6. I2C Description .....	35
1. Electrical Specifications .....	3	6.1. Highlights .....	35
2. Detailed Description .....	4	6.2. Read Operations .....	36
2.1. IS4320 Description .....	4	6.2.1. Single Word Read .....	36
2.2. Organization .....	5	6.2.2. Multiple Word Read .....	37
2.3. IS4320 Advantages .....	6	6.3. Write Operations .....	38
2.4. Modbus UART Port .....	7	6.3.1. Single Word Write .....	38
3. Usage .....	8	6.3.2. Multiple Word Write .....	39
3.1. Example: Read Holding Registers .....	9	7. Mechanical .....	40
3.2. Example: Write Holding Registers .....	12	8. Hardware Examples .....	42
4. Pin Description .....	15	8.1. RS485 Example .....	42
4.1. TX and RX Pins .....	15	8.2. Isolated RS485 Example .....	44
4.2. DIR Pin .....	15	8.3. RS232 Example .....	45
4.3. SCL and SDA Pins .....	16	8.4. Bus Topology .....	46
4.4. I2CSPD Pin .....	16	8.5. Cable Wiring .....	47
5. Memory Description .....	17	9. Firmware Examples .....	48
5.1. Memory Map Organization .....	17	9.1. Arduino Example .....	48
5.2. Memory Map Table .....	18	9.2. STM32 Example .....	51
5.3. CFG_MBBDR Register .....	19	9.3. Raspberry Pi Example .....	56
5.4. CFG_MBPARG Register .....	20	10. PC/Mac Tools .....	59
5.5. CFG_MBSTP Register .....	21	10.1. Modbus Tools .....	59
5.6. CFG_MB_TIMEOUT Register .....	22	10.2. I2C Tools .....	60
5.7. CFG_CHIP_ID Register .....	23	Content .....	61
5.8. CFG_CHIP_REV Register .....	24	Appendix .....	62
5.9. REQ_EXECUTE Register .....	25	Revision History .....	62
5.10. REQ_SLAVE Register .....	26	Documentation Feedback .....	62
5.11. REQ_FC Register .....	27	Sales Contact .....	62
5.12. REQ_STARTING Register .....	28	Customization .....	62
5.13. REQ_QTY Register .....	29	Trademarks .....	62
5.14. REQ_DATAx Register .....	30	Disclaimer .....	63
5.15. RES_STATUS Register .....	32		

## Appendix

### Revision History

---

Date	Revision Code	Description
September 2025	ISDOC141A	- Initial Release

### Documentation Feedback

---

Feedback and error reporting on this document are very much appreciated. Please indicate the code or title of the document.

[feedback@inacks.com](mailto:feedback@inacks.com)

### Sales Contact

---

For special order requirements, large volume orders, or scheduled orders, please contact our sales department at:

[sales@inacks.com](mailto:sales@inacks.com)

### Customization

---

INACKS can develop new products or customize existing ones to meet specific client needs. Please contact our engineering department at:

[engineering@inacks.com](mailto:engineering@inacks.com)

### Trademarks

---

This company and its products are developed independently and are not affiliated with, endorsed by, or associated with any official protocol or standardization entity. All trademarks, names, and references to specific protocols remain the property of their respective owners.

## Disclaimer

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, INACKS does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. INACKS takes no responsibility for the content in this document if provided by an information source outside of INACKS.

In no event shall INACKS be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, INACKS's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of INACKS.

**Right to make changes** — INACKS reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — INACKS products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an INACKS product can reasonably be expected to result in personal injury, death or severe property or environmental damage. INACKS and its suppliers accept no liability for inclusion and/or use of INACKS products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Quick reference data** — The Quick reference data is an extract of the product data given in the Limiting values and Characteristics sections of this document, and as such is not complete, exhaustive or legally binding.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. INACKS makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using INACKS products, and INACKS accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the INACKS product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

INACKS does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using INACKS products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). INACKS does not accept any liability in this respect.

**Limiting values** — Stress above one or more limiting values (as defined in the Absolute Maximum Ratings System of IEC 60134) will cause permanent damage to the device. Limiting values are stress ratings only and (proper) operation of the device at these or any other conditions above those given in the Recommended operating conditions section (if present) or the Characteristics sections of this document is not warranted. Constant or repeated exposure to limiting values will permanently and irreversibly affect the quality and reliability of the device.

**Terms and conditions of commercial sale** — INACKS products are sold subject to the general terms and conditions of commercial sale, as published at <http://www.inacks.com/commercialsaleterms>, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. INACKS hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of INACKS products by customer.

**No offer to sell or license** — Nothing in this document may be interpreted or construed as an offer to sell products that is open for acceptance or the grant, conveyance or implication of any license under any copyrights, patents or other industrial or intellectual property rights.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Non-automotive qualified products** — This INACKS product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. INACKS accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

**Protocol Guidance Disclaimer:** The information provided herein regarding the protocol is intended for guidance purposes only. While INACKS strive to provide accurate and up-to-date information, this content should not be considered a substitute for official protocol documentation. It is the responsibility of the client to consult and adhere to the official protocol documentation when designing or implementing systems based on this protocol.

INACKS make no representations or warranties, either expressed or implied, as to the accuracy, completeness, or reliability of the information contained in this document. INACKS shall not be held liable for any errors, omissions, or inaccuracies in the information or for any user's reliance on the information.

The client is solely responsible for verifying the suitability and compliance of the provided information with the official protocol standards and for ensuring that their implementation or usage of the protocol meets all required specifications and regulations. Any reliance on the information provided is strictly at the user's own risk.

**Certification and Compliance Disclaimer:** Please be advised that the product described herein has not been certified by any competent authority or organization responsible for protocol standards. INACKS do not guarantee that the chip meets any specific protocol compliance or certification standards.

It is the responsibility of the client to ensure that the final product incorporating this product is tested and certified according to the relevant protocol standards before use or commercialization. The certification process may result in the product passing or failing to meet these standards, and the outcome of such certification tests is beyond our control.

INACKS disclaim any liability for non-compliance with protocol standards and certification failures. The client acknowledges and agrees that they bear sole responsibility for any legal, compliance, or technical issues that arise due to the use of this product in their products, including but not limited to the acquisition of necessary protocol certifications.