MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
*(A constituent unit of MAHE, Manipal)*

# Industrial Automation (ICE 3252)

## Industrial Communication Protocol- ModBus

Bipin Krishna
Assistant Professor (Sr.)
ICE Department
Manipal Institute of Technology
MAHE, Karnataka, India

# MODBUS

- The MODBUS standard defines an application layer messaging protocol, positioned at level 7 of the OSI model that provides "client/server" communications between devices connected on different types of buses or networks.
- It standardizes also a specific protocol on serial line to exchange MODBUS request between a master and one or several slaves.
- We will discuss the MODBUS over Serial Line protocol.
- **MODBUS Serial Line protocol is a Master-Slave protocol.**
- This protocol takes place at level 2 of the OSI model.

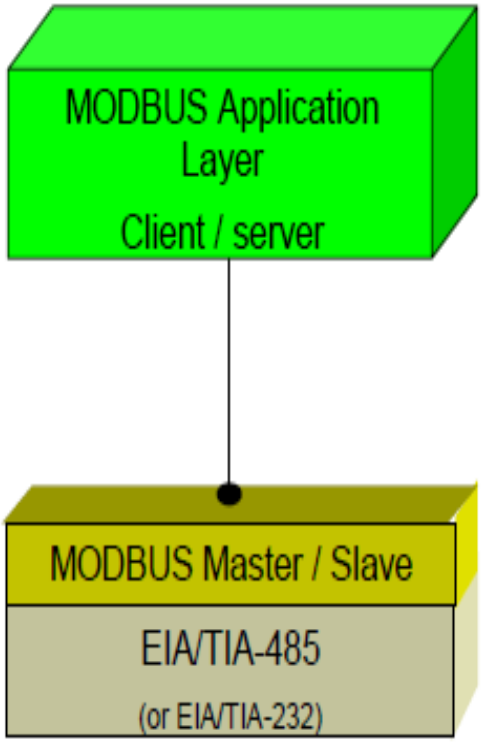
# MODBUS

- A master-slave type system has one node (the master node) that issues explicit commands to one of the "slave" nodes and processes responses.
- Slave nodes will not typically transmit data without a request from the master node, and do not communicate with other slaves.
- At the physical level, MODBUS over Serial Line systems may use different physical interfaces (RS485, RS232).
- TIA/EIA-485 (RS485) Two-Wire interface is the most common.
- As an add-on option, RS485 Four-Wire interface may also be implemented.
- A TIA/EIA-232- E (RS232) serial interface may also be used as an interface, when only short point to point communication is required.

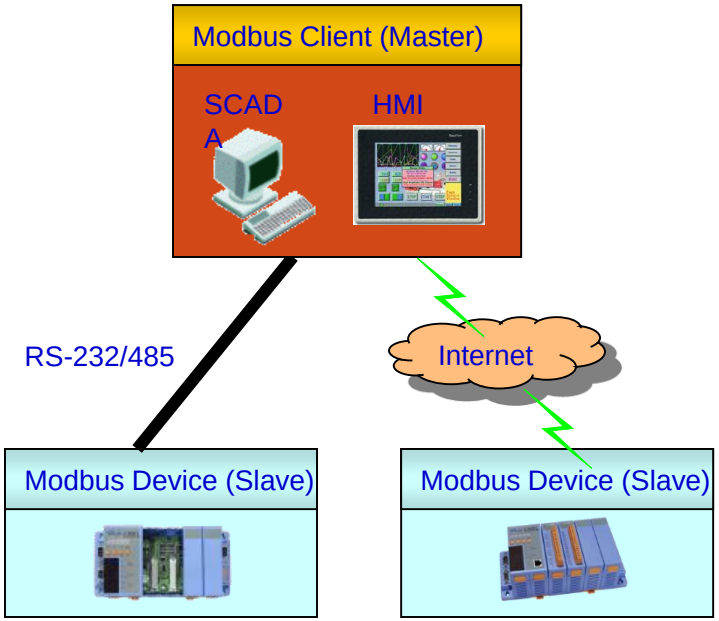
# MODBUS

- MODBUS application layer messaging protocol, positioned at level 7 of the OSI model, provides client/server communication between devices connected on buses or networks.
- On MODBUS serial line the client role is provided by the Master of the serial bus and the Slaves nodes act as servers.

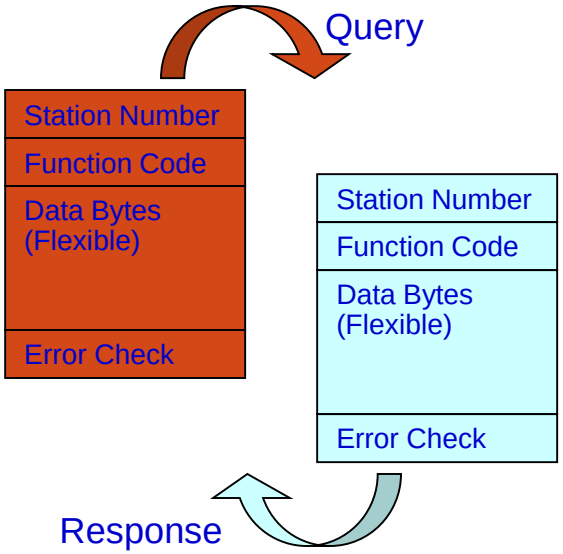
| Layer | ISO/OSI Model | |
|---|---|---|
| 7 | Application | MODBUS Application Protocol |
| 6 | Presentation | Empty |
| 5 | Session | Empty |
| 4 | Transport | Empty |
| 3 | Network | Empty |
| 2 | Data Link | MODBUS Serial Line Protocol |
| 1 | Physical | EIA/TIA-485 (or EIA/TIA-232) |



# Application Structure (general)



# Query-Response Cycle



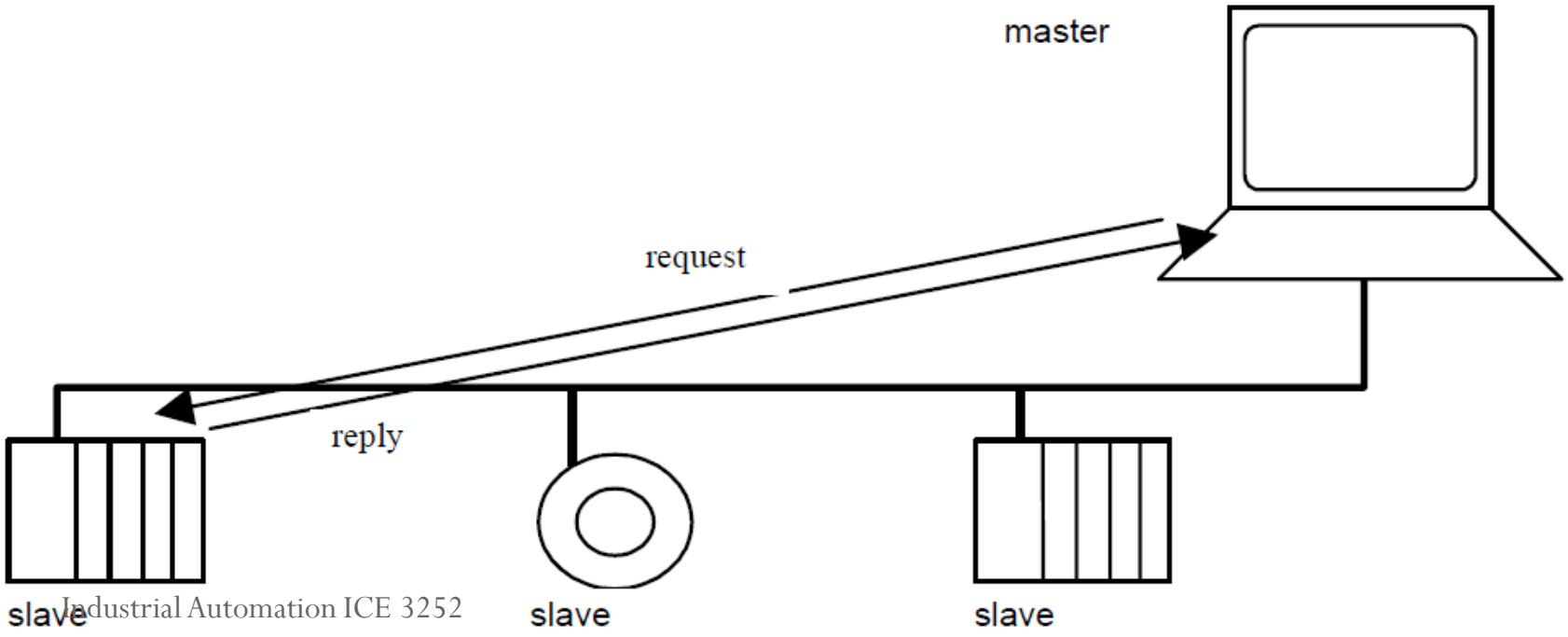
# MODBUS Data Link Layer

- The MODBUS Serial Line protocol is a Master-Slaves protocol.
- Only one master (at the same time) is connected to the bus, and one or several (247 maximum number) slaves nodes are also connected to the same serial bus.
- A MODBUS communication is always initiated by the master.
- The slave nodes will never transmit data without receiving a request from the master node.
- The slave nodes will never communicate with each other.
- The master node initiates only one MODBUS transaction at the same time.

# Communication

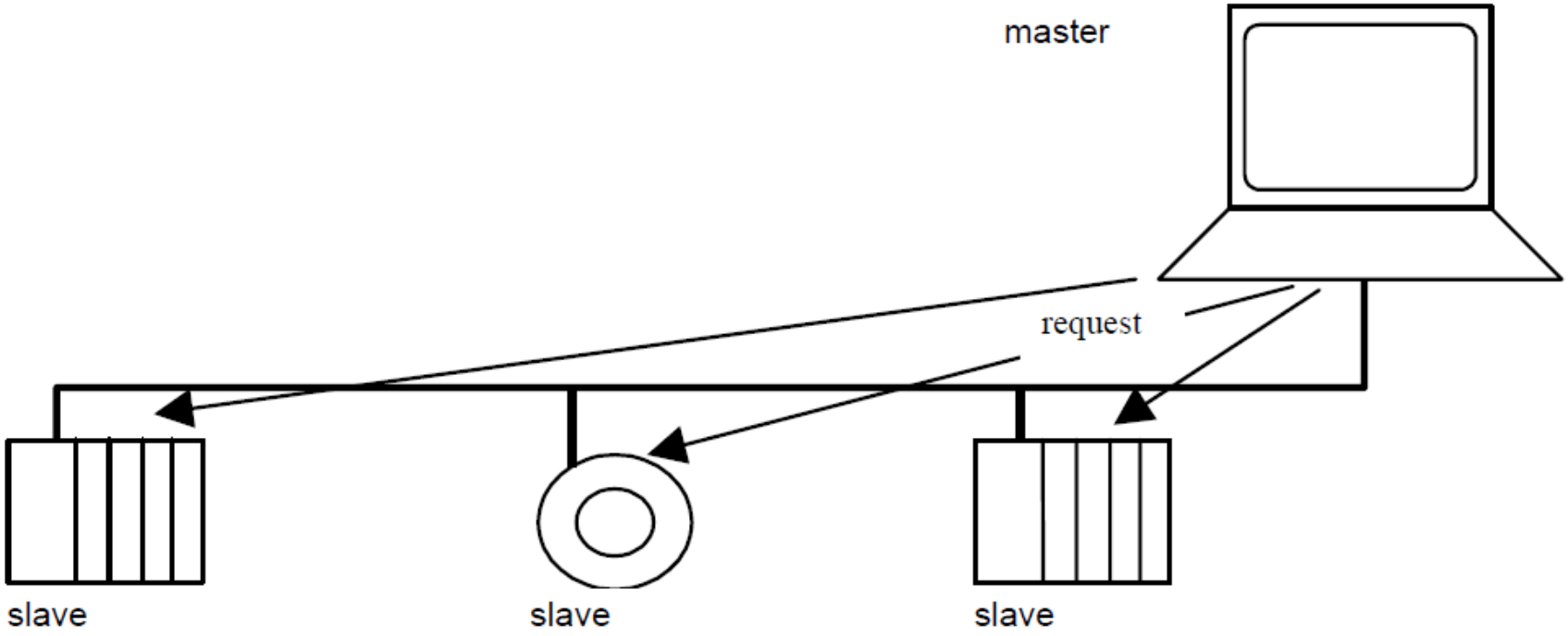
- The master node issues a MODBUS request to the slave nodes in two modes :**Unicast mode** and **broadcast mode**

- Unicast mode

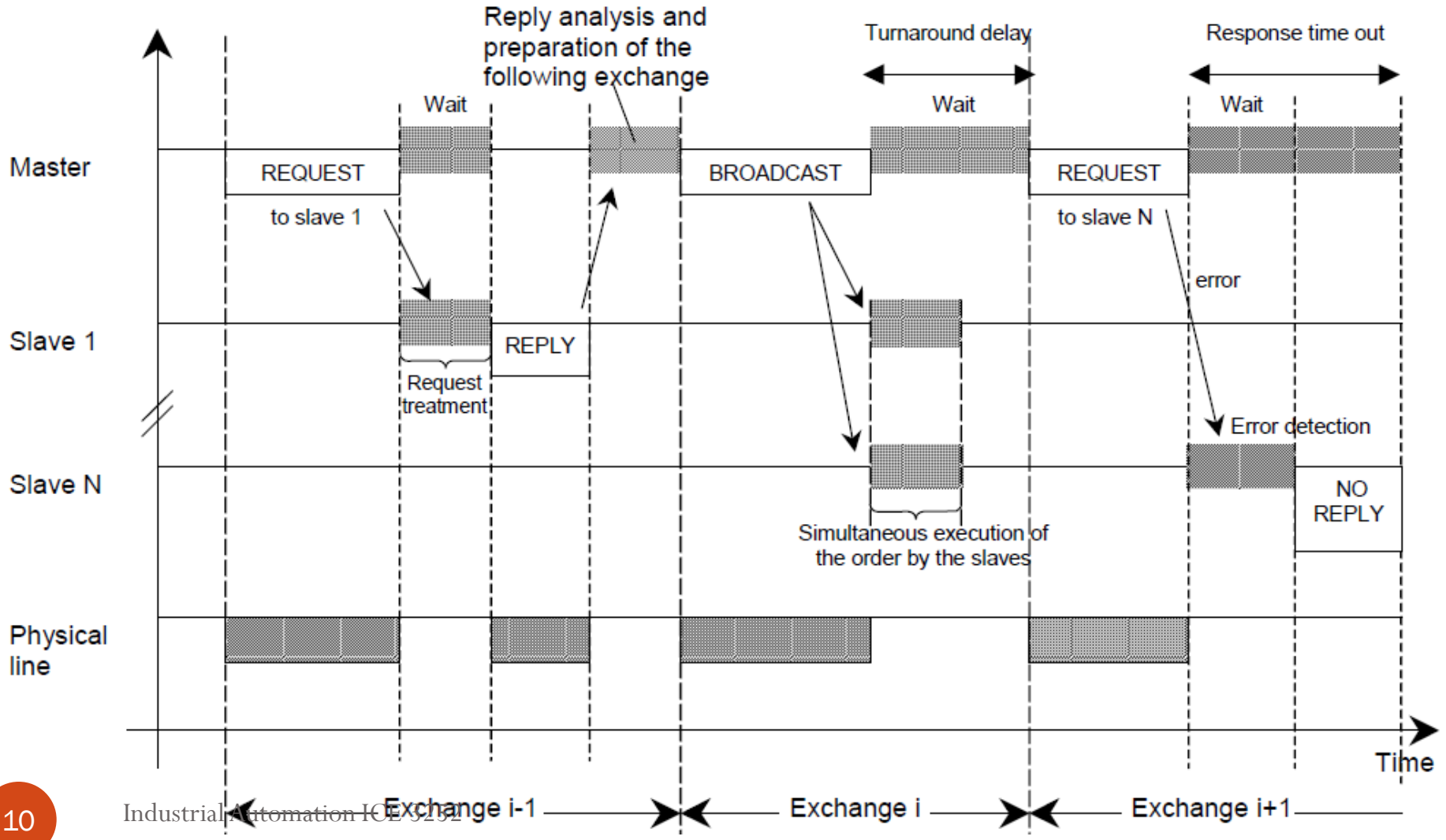


# Communication

- Broadcast mode



# Master/Slave timing diagram

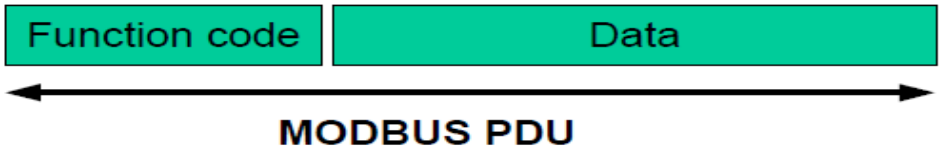


# Addressing in MODBUS

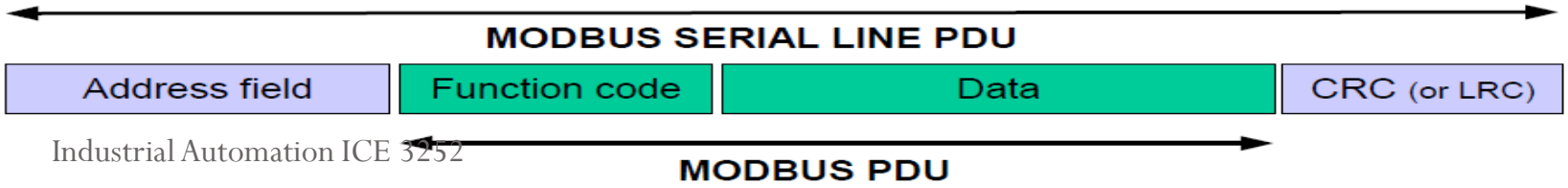
- MODBUS Addressing rules

| 0 | From 1 to 247 | From 248 to 255 |
|---|---|---|
| Broadcast address | Slave individual addresses | Reserved |

- MODBUS Frame:
  - The MODBUS application protocol defines a simple **P**rotocol **D**ata **U**nit (**PDU**) independent of the underlying communication layers:




- The mapping of MODBUS protocol on a specific bus or network introduces some additional fields on the **P**rotocol **D**ata **U**nit.
- The client that initiates a MODBUS transaction builds the MODBUS PDU, and then adds fields in order to build the appropriate



# Two Serial Transmission Modes

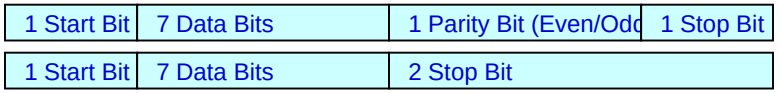
Two different serial transmission modes are defined : The RTU mode and the ASCII mode.

It defines the bit contents of message fields transmitted serially on the line.

It determines how information is packed into the message fields and decoded.

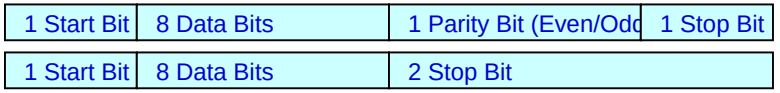
- ASCII Mode
  - Data system

    ASCII character, '0'~'9','A'~'F'

    
    
  - Bits per data unit
  - Error Check Field

    Longitudinal Redundancy Check (LRC)
- RTU Mode
  - Data system

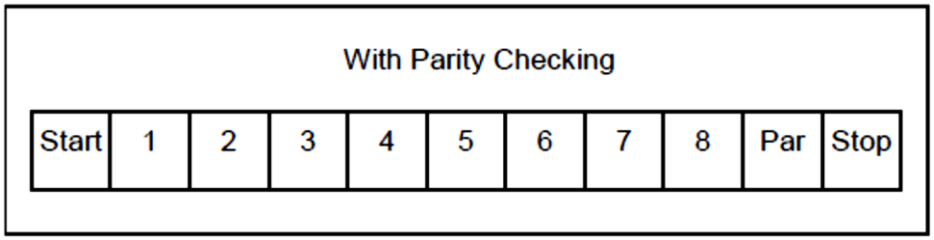
    8-bit Binary, 00~FF

    
    
  - Bits per data unit
  - Error Check Field

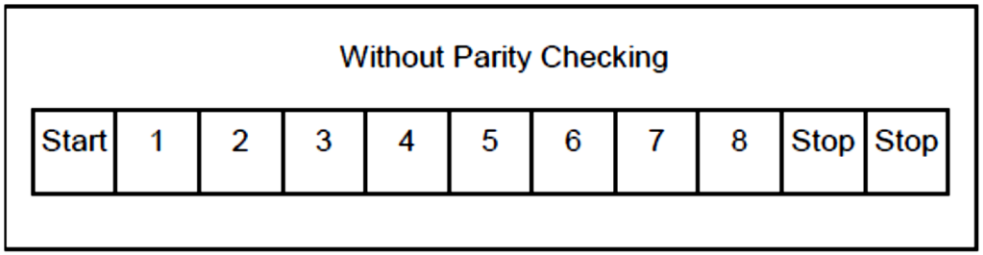
    Cyclical Redundancy Check (CRC)
- The default transmission mode is RTU mode. However user can choose ASCII mode based on requirement.

# Bit séquence in RTU mode

- With parity check



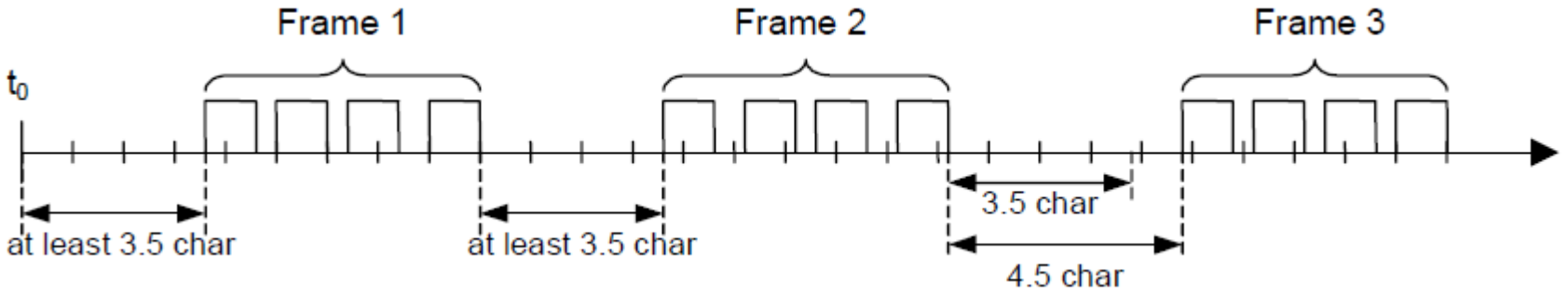

- Witho

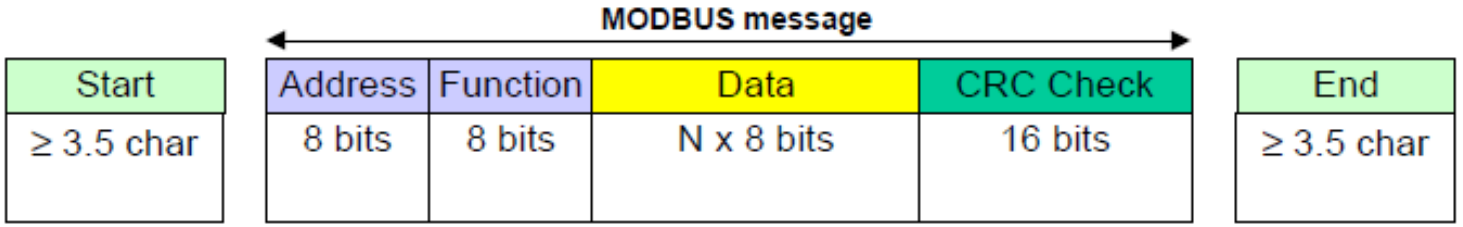


# RTU message frame

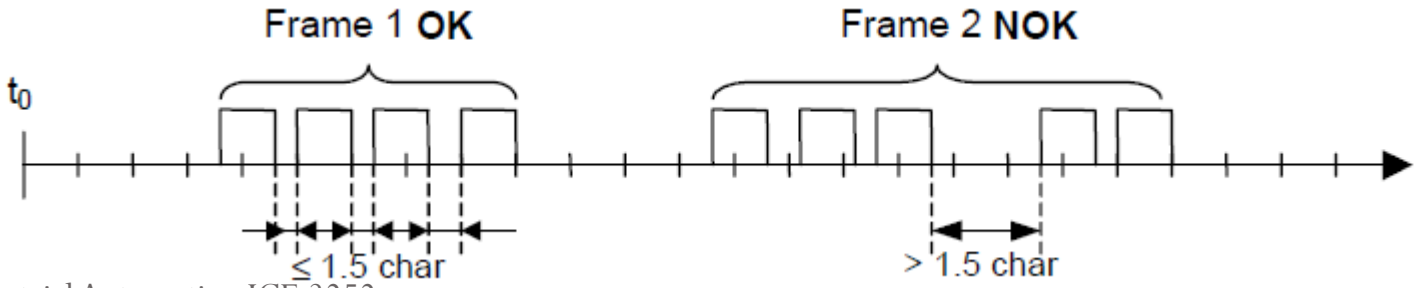
- A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point.
- This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed.
- Partial messages must be detected and errors must be set as a result.
- In RTU mode, message frames are separated by a silent interval of at least 3.5 character times.
- If a silent interval of more than 1.5 character times occurs between two characters, the message frame is declared incomplete and should be discarded by the receiver.

# RTU message frame



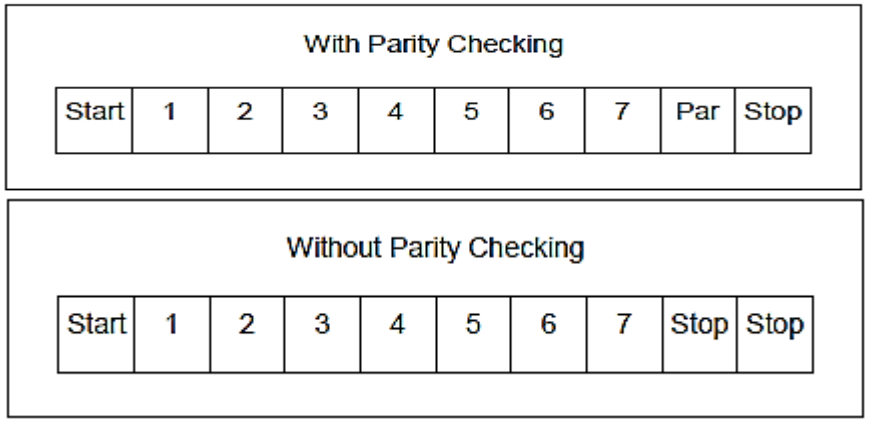




RTU Message Frame



# ASCII mode

- This mode is used when the physical communication link or the capabilities of the device does not allow the conformance with RTU mode requirements regarding timers management.
- Remark : this mode is less efficient than RTU since each byte needs two characters.




- ASCII message frame

| Start | Address | Function | Data | LRC | End |
|---|---|---|---|---|---|
| 1 char<br>: | 2 chars | 2 chars | 0 up to 2x252 char(s) | 2 chars | 2 chars<br>CR,LF |

- The CRLF pair is represented as ASCII 0D and 0A hex.

- The address field of a message frame contains two characters.
- In ASCII mode, a message is delimited by specific characters as Start-of-frames and End-of-frames.
- A message must start with **a 'colon' ( : )** character (ASCII 3A hex), and end with a **'carriage return – line feed'** (CRLF) pair (ASCII 0D and 0A hex).
- The devices monitor the bus continuously for the 'colon' character.
- When this character is received, each device decodes the next character until it detects the End-Of-Frame.
- Intervals of up to one second may elapse between characters within the message. Unless the user has configured a longer timeout, an interval greater than 1 second means an error has occurred.
- Some Wide-Area-Network application may require a timeout in the 4 to 5 second range.

# Modbus Message Packet

- ASCII Mode

| Start | Station Number | Function Code | Data | Error Check | End |
|---|---|---|---|---|---|
| 1 Char | 2 Chars | 2 Chars | n Chars | 2 Chars | 2 Chars |
| : | | | | LRC | CR,LF |

- RTU Mode

| Start | Station Number | Function Code | Data | Error Check | End |
|---|---|---|---|---|---|
| 3.5 Char | 1 Char | 1 Char | n Chars | 2 Chars | 3.5 Chars |
| Silence | | | | CRC | Silence |

- Modbus Plus network

| Prefixed Data | Station Number | Function Code | Data |
|---|---|---|---|
| 6 x 8 Bits | | | |

Byte 0, 1: transaction ID – usually 0
Byte 2, 3: protocol ID = 0
Byte 4, 5: number of bytes following

# Address

| **Modbus RTU Data Type** | **Common name** | **Starting address** |
|---|---|---|
| Modbus Coils | Bits, binary values, flags | 00001 |
| Digital Inputs | Binary inputs | 10001 |
| Analog Inputs | Binary inputs | 30001 |
| Analog outputs | Analog values, variables | 40001 |

## Modbus Function Code

- 01: read DOs (0xxxx)
- 02: read DIs (1xxxx)
- 03: read AOs (4xxxx)
- 04: read AIs (3xxxx)
- 05: write single DO (0xxxx)
- 06: write single AO (4xxxx)
- 15: wirte DOs (0xxxx)
- 16: write AOs (4xxxx)

# Comparison of ASCII with RTU mode of transmission

**Properties of Modbus/ASCII and Modbus/RTU**

| | Modbus/ASCII | | Modbus/RTU | |
|---|---|---|---|---|
| **Characters** | ASCII **0**...**9** and **A**..**F** | | Binary **0**...**255** | |
| **Error check** | **LRC** Longitudinal Redundancy Check | | **CRC** Cyclic Redundancy Check | |
| **Frame start** | character ':' | | 3.5 chars silence | |
| **Frame end** | characters **CR/LF** | | 3.5 chars silence | |
| **Gaps in message** | 1 sec | | 1.5 times char length | |
| **Start bit** | **1** | | **1** | |
| **Data bits** | **7** | | **8** | |
| **Parity** | even/odd | none | even/odd | none |
| **Stop bits** | **1** | **2** | **1** | **2** |

# Error Checking Methods

- The security of standard MODBUS Serial Line is based on two kinds of error checking :
- Parity checking (even or odd) should be applied to each character.
- Frame checking (LRC or CRC) must be applied to the entire message.
- Both the character checking and message frame checking are generated in the device (master or slave) that emits and applied to the message contents before transmission.
- The device (slave or master) checks each character and the entire message frame during receipt.

- The master is configured by the user to wait for a predetermined timeout interval ( Response time-out) before aborting the transaction.
- This interval is set to be long enough for any slave to respond normally ( unicast request).
- If the slave detects a transmission error, the message will not be acted upon.
- The slave will not construct a response to the master.
- Thus the timeout will expire and allow the master's program to handle the error.
- Note that a message addressed to a nonexistent slave device will also cause a timeout.

# Parity Checking

- Users may configure devices for Even ( required) or Odd Parity checking, or for No Parity checking ( recommended).
- This will determine how the parity bit will be set in each character.
- If either Even or Odd Parity is specified, the quantity of 1 bits will be counted in the data portion of each character (seven data bits for ASCII mode, or eight for RTU).
- The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bits.

# Example:

- For example, these eight data bits are contained in an RTU character frame: 1100 0101
- The total quantity of 1 bits in the frame is four.
- If Even Parity is used, the frame's parity bit will be a 0 making the total quantity of 1 bits still an even number (four).
- If Odd Parity is used, the parity bit will be a 1, making an odd quantity (five).

# Frame Checking

- Two kinds of frame checking is used depending on the transmission mode, RTU or ASCII.
- In RTU mode, messages include an error–checking field that is based on a Cyclical Redundancy Checking (CRC) method.
- The CRC field checks the contents of the entire message.
- It is applied regardless of any parity checking method used for the individual characters of the message.
- In ASCII mode, messages include an error–checking field that is based on a Longitudinal Redundancy Checking (LRC) method.
- The LRC field checks the contents of the message, exclusive of the beginning 'colon' and ending CRLF pair.
- It is applied regardless of any parity checking method used for the individual characters of the message.

# LRC Generation

- The Longitudinal Redundancy Check (LRC) field is one byte, containing an 8–bit binary value.
- The LRC is calculated by adding together successive 8–bit bytes in the message, discarding any carries, and then two's complementing the result.
- The LRC is an 8–bit field, therefore each new addition of a character that would result in a value higher than 255 decimal simply 'rolls over' the field's value through zero.
- Because there is no ninth bit, the carry is discarded automatically.
- LRC is used to detect burst errors.
- The main problem with LRC is that, it is not able to detect error if two bits in a data unit are damaged and two bits in exactly the same position in other data unit are also damaged.

# LRC generation

**A procedure for generating an LRC is:**

1. Add all bytes in the message, excluding the starting 'colon' and ending CRLF. Add them into an 8–bit field, so that carries will be discarded.

2. Subtract the final field value from FF hex (all 1's), to produce the ones–complement.

3. Add 1 to produce the twos–complement.

# LRC generation

- Placing the LRC into the Message:
  - For example, if the LRC value is 61 hex (0110 0001):

| Colon | Addr | Func | Data Count | Data | Data | Data | Data | LRC Hi | LRC Lo | CR | LF |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | "6" 0x36 | "1" 0x31 | | |

# Example:

- Consider the following 32 bit data to be transmitted,
- 11001010  10101010  11001100  11100011
- Step 1: Add all bytes in the message
- 11001010

  10101010

  11001100

  11100011

  ---

  01001111
- Step 2: find 1's compliment: 10110000
- Step 3: find 2's compliment: 10110001 or B1H
- LRC: 42H and 31H

# Example:

- Suppose 32 bit data plus LRC that was being transmitted is hit by a burst error and some bits are corrupted as shown below:
- 11001010 11101000 10001110 111000011
- Step 1: Add all bytes in the message
- 11001010

  11101000

  10001110

  11100011

  ---

  01001111
- Step 2: find 1's compliment: 10110000
- Step 3: find 2's compliment: 10110001 or B1H

# CRC generation

- A procedure for generating a CRC is:

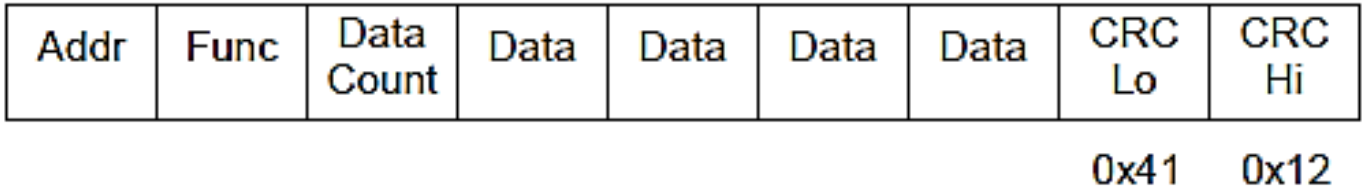
1. Load a 16–bit register with FFFF hex (all 1's). Call this the CRC register.
2. Exclusive OR the first 8–bit byte of the message with the low–order byte of the 16–bit CRC register, putting the result in the CRC register.
3. Shift the CRC register one bit to the right (toward the LSB), zero–filling the MSB. Extract and examine the LSB.
4. (If the LSB was 0): Repeat Step 3 (another shift).

   (If the LSB was 1): Exclusive OR the CRC register with the polynomial value 0xA001 (1010 0000 0000 0001).
5. Repeat Steps 3 and 4 until 8 shifts have been performed. When this is done, a complete 8–bit byte will have been processed.
6. Repeat Steps 2 through 5 for the next 8–bit byte of the message. Continue doing this until all bytes have been processed.
7. The final content of the CRC register is the CRC value.
8. When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

# CRC generation

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

For example, if the CRC value is 1241 hex (0001 0010 0100 0001):



# CRC generation

- Example of CRC calculation (frame 02 07)

| | | | | | |
|---|---|---|---|---|---|
| CRC register initialization | | 1111 | 1111 | 1111 | 1111 |
| XOR 1st character | | 0000 | 0000 | 0000 | 0010 |
| | | 1111 | 1111 | 1111 | 1101 |
| | Move 1 | 0111 | 1111 | 1111 | 1110\|1 |
| Flag to 1, XOR polynomial | | 1010 | 0000 | 0000 | 0001 |
| | | 1101 | 1111 | 1111 | 1111 |
| | Move 2 | 0110 | 1111 | 1111 | 1111\|1 |
| Flag to 1, XOR polynomial | | 1010 | 0000 | 0000 | 0001 |
| | | 1100 | 1111 | 1111 | 1110 |
| | Move 3 | 0110 | 0111 | 1111 | 1111\|0 |
| | Move 4 | 0011 | 0011 | 1111 | 1111\|1 |
| | | 1010 | 0000 | 0000 | 0001 |
| | | 1001 | 0011 | 1111 | 1110 |
| | Move 5 | 0100 | 1001 | 1111 | 1111\|0 |
| | Move 6 | 0010 | 0100 | 1111 | 1111\|1 |
| | | 1010 | 0000 | 0000 | 0001 |
| | | 1000 | 0100 | 1111 | 1110 |
| | Move 7 | 0100 | 0010 | 0111 | 1111\|0 |
| | Move 8 | 0010 | 0001 | 0011 | 1111\|1 |
| | | 1010 | 0000 | 0000 | 0001 |
| | | 1000 | 0001 | 0011 | 1110 |

# CRC generation

| | | | | |
|---|---|---|---|---|
| | 1000 | 0001 | 0011 | 1110 |
| XOR 2nd character | 0000 | 0000 | 0000 | 0111 |
| | 1000 | 0001 | 0011 | 1001 |
| Move 1 | 0100 | 0000 | 1001 | 1100 \| 1 |
| | 1010 | 0000 | 0000 | 0001 |
| | 1110 | 0000 | 1001 | 1101 |
| Move 2 | 0111 | 0000 | 0100 | 1110 \| 1 |
| | 1010 | 0000 | 0000 | 0001 |
| | 1101 | 0000 | 0100 | 1111 |
| Move 3 | 0110 | 1000 | 0010 | 0111 \| 1 |
| | 1010 | 0000 | 0000 | 0001 |
| | 1100 | 1000 | 0010 | 0110 |
| Move 4 | 0110 | 0100 | 0001 | 0011 \| 0 |
| Move 5 | 0011 | 0010 | 0000 | 1001 \| 1 |
| | 1010 | 0000 | 0000 | 0001 |
| | 1001 | 0010 | 0000 | 1000 |
| Move 6 | 0100 | 1001 | 0000 | 0100 \| 0 |
| Move 7 | 0010 | 0100 | 1000 | 0010 \| 0 |
| Move 8 | 0001 | 0010 | 0100 | 0001 \| 0 |
| | | | Most significant | least significant |



The CRC 16 of the frame is then: 4112