1. INTRODUCTION

Purpose

MODBUS is a request/reply protocol and offers services specified by **function Codes**. MODBUS function codes are elements of MODBUS request/reply PDUs. The objective of this document is to describe the function codes used within the framework of MODBUS transactions.

MODBUS is an application layer messaging protocol for client/server communication between devices connected on different types of buses or networks.

Scope

MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model, which provides client/server communication between devices connected on different types of buses or networks.

The industry's serial de facto standard since 1979, MODBUS continues to enable millions of automation devices to communicate. Today, support for the simple and elegant structure of MODBUS continues to grow. The Internet community can access MODBUS at a reserved system port 502 on the TCP/IP stack.

MODBUS is a request/reply protocol and offers services specified by **function codes**. MODBUS function codes are elements of MODBUS request/reply PDUs. The objective of this document is to describe the function codes used within the framework of MODBUS transactions.

MODBUS is an application layer messaging protocol for client/server communication between devices connected on different types of buses or networks.

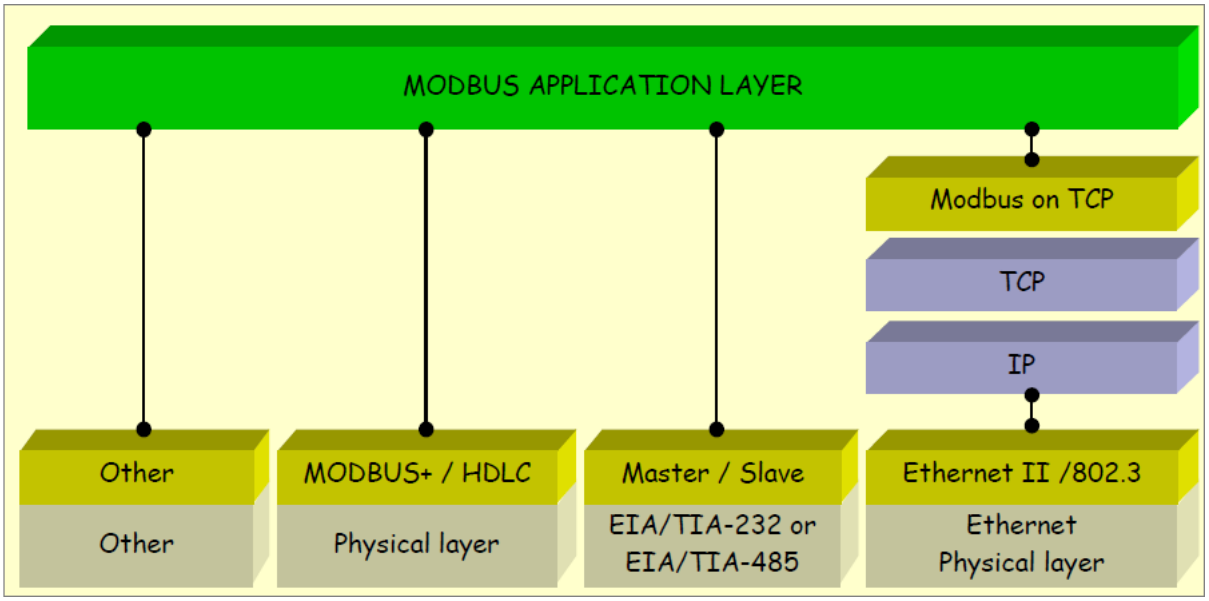
2. IMPLEMENTATION

It is currently implemented using: TCP/IP over Ethernet

Asynchronous serial transmission over a variety of media (wire: EIA/TIA-232-E, EIA-422, EIA/TIA-485-A; fibre, radio, etc.)

MODBUS PLUS, a high speed token passing network.

MODBUS communication stack



Abbreviations

ADU Application Data Unit

HDLC High level Data Link Control

HMI Human Machine Interface

IETF Internet Engineering Task Force

I/O Input /Output

IP Internet Protocol

MAC Media Access Control

MB MODBUS Protocol

MBAP MODBUS Application Protocol

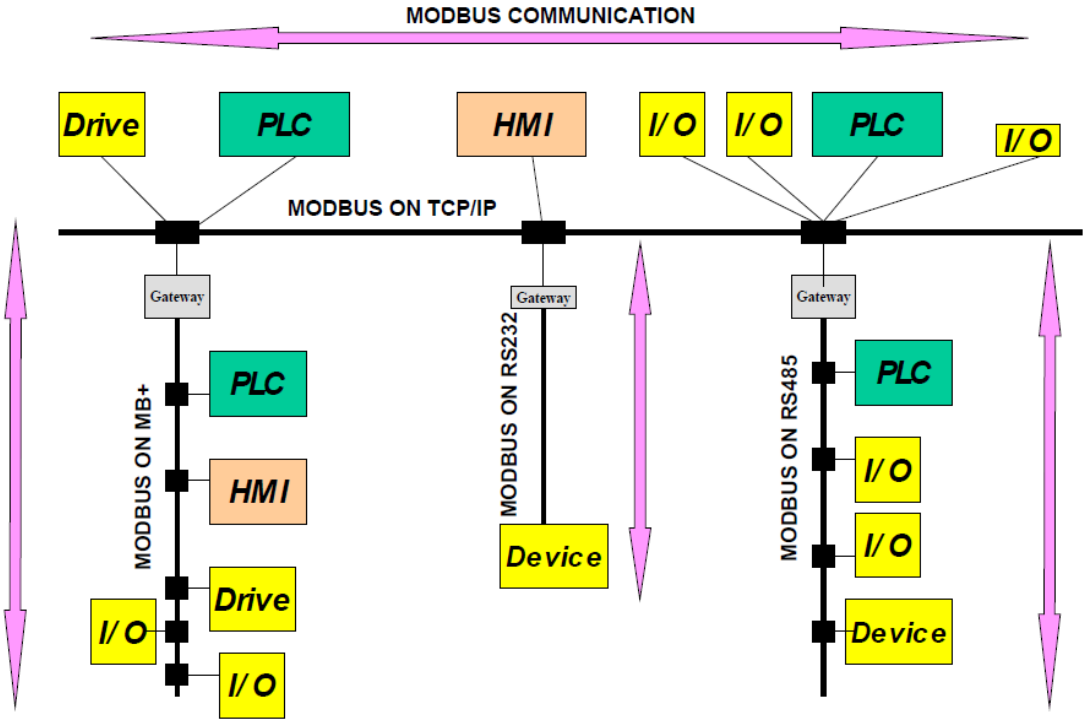
PDU Protocol Data Unit

PLC Programmable Logic Controller

TCP Transmission Control Protocol

Context

The MODBUS protocol allows an easy communication within all types of network architectures.

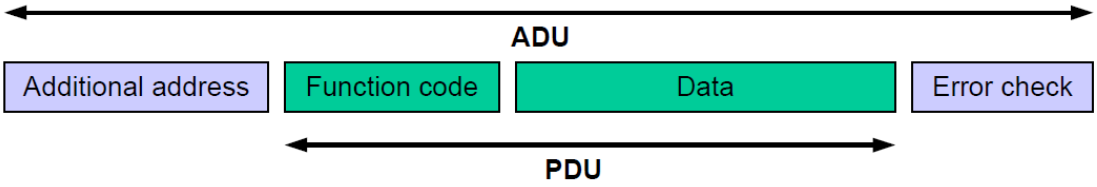


Every type of devices (PLC, HMI, Control Panel, Driver, Motion control, I/O Device…) can use MODBUS protocol to initiate a remote operation. The same communication can be done as well on serial line as on an Ethernet TCP/IP networks. Gateways allow a communication between several types of buses or network using the MODBUS protocol.

3. GENERAL DESCRIPTION

Protocol description

The MODBUS protocol defines a simple protocol data unit **(PDU)** independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or network can introduce some additional fields on the application data unit **(ADU)**.



The MODBUS application data unit is built by the client that initiates a MODBUS transaction. The function indicates to the server what kind of action to perform. The MODBUS application protocol establishes the format of a request initiated by a client.

The function code field of a MODBUS data unit is coded in one byte. Valid codes are in the range of 1 ... 255 decimal (the range 128 – 255 is reserved and used for exception responses). When a message is sent from a Client to a Server device the function code field tells the server what kind of action to perform. Function code "0" is not valid.

Sub-function codes are added to some function codes to define multiple actions.

The data field of messages sent from a client to server devices contains additional information that the server uses to take the action defined by the function code. This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.

The data field may be non-existent (of zero length) in certain kinds of requests, in this case the server does not require any additional information. The function code alone specifies the action.

If no error occurs related to the MODBUS function requested in a properly received MODBUS ADU the data field of a response from a server to a client contains the data requested. If an error related to the MODBUS function requested occurs, the field contains an exception code that the server application can use to determine the next action to be taken.

For example a client can read the ON / OFF states of a group of discrete outputs or inputs or it can read/write the data contents of a group of registers.

When the server responds to the client, it uses the function code field to indicate either a normal (error-free) response or that some kind of error occurred (called an exception response). For a normal response, the server simply echoes to the request the original function code.

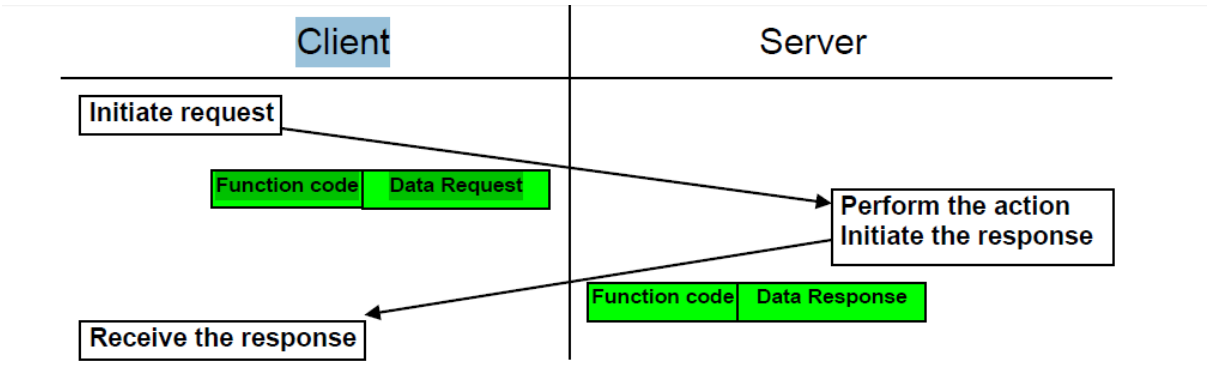


Figure 4: MODBUS transaction (error free)

PDU with its most for an exception response, the server returns a code that is equivalent to the original function code from the request significant bit set to logic 1.

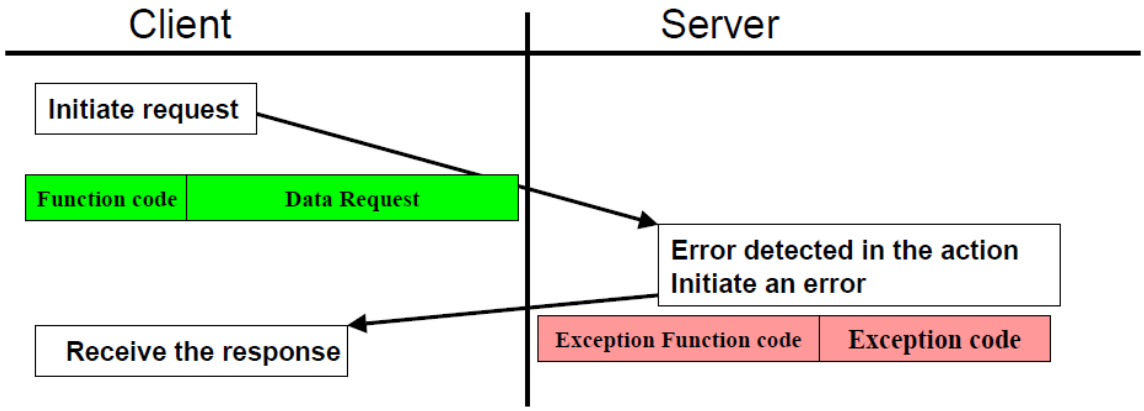


Figure 5: MODBUS transaction (exception response)

The size of the MODBUS PDU is limited by the size constraint inherited from the first MODBUS implementation on Serial Line network (max. RS485 ADU = 256 bytes).

Therefore:

MODBUS **PDU for serial line communication =** 256 - Server address (1 byte) - CRC (2 bytes) = **253 bytes**.

Consequently:

RS232 / RS485 **ADU** = 253 bytes + Server address (1 byte) + CRC (2 bytes) = **256 bytes**.

TCP MODBUS **ADU** = 253 bytes + MBAP (7 bytes) = **260 bytes**.

The MODBUS protocol defines three PDUs. They are:

• MODBUS Request PDU, mb_req_pdu

• MODBUS Response PDU, mb_rsp_pdu

• MODBUS Exception Response PDU, mb_excep_rsp_pdu

MODBUS Data model

MODBUS bases its data model on a series of tables that have distinguishing characteristics. The four primary tables are:

Primary tables	Object type	Type of	Comments
Discretes Input	Single bit	Read-Only	This type of data can be provided by an I/O system.
Coils	Single bit	Read-Write	This type of data can be alterable by an application program.
Input Registers	16-bit word	Read-Only	This type of data can be provided by an I/O system
Holding Registers	16-bit word	Read-Write	This type of data can be alterable by an application program.

The distinctions between inputs and outputs, and between bit-addressable and word-addressable data items, do not imply any application behaviour. It is perfectly acceptable, and very common, to regard all four tables as overlaying one another, if this is the most natural interpretation on the target machine in question.

For each of the primary tables, the protocol allows individual selection of 65536 data items, and the operations of read or write of those items are designed to span multiple consecutive data items up to a data size limit which is dependent on the transaction function code.

It's obvious that all the data handled via MODBUS (bits, registers) must be located in device application memory. But physical address in memory should not be confused with data reference. The only requirement is to link data reference with physical address.

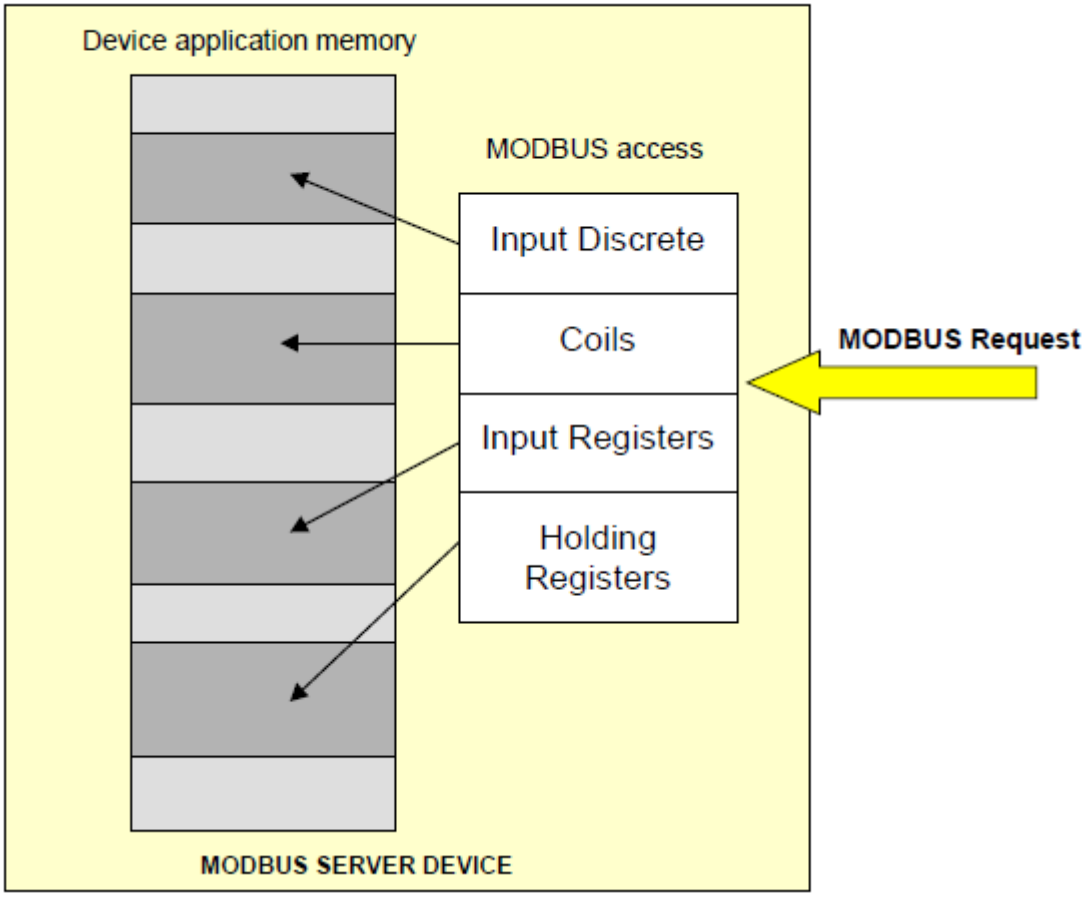
MODBUS logical reference numbers, which are used in MODBUS functions, are unsigned integer indices starting at zero.

Implementation examples of MODBUS model

The examples below show two ways of organizing the data in device. There are different organizations possible, but not all are described in this document. Each device can have its own organization of the data according to its application

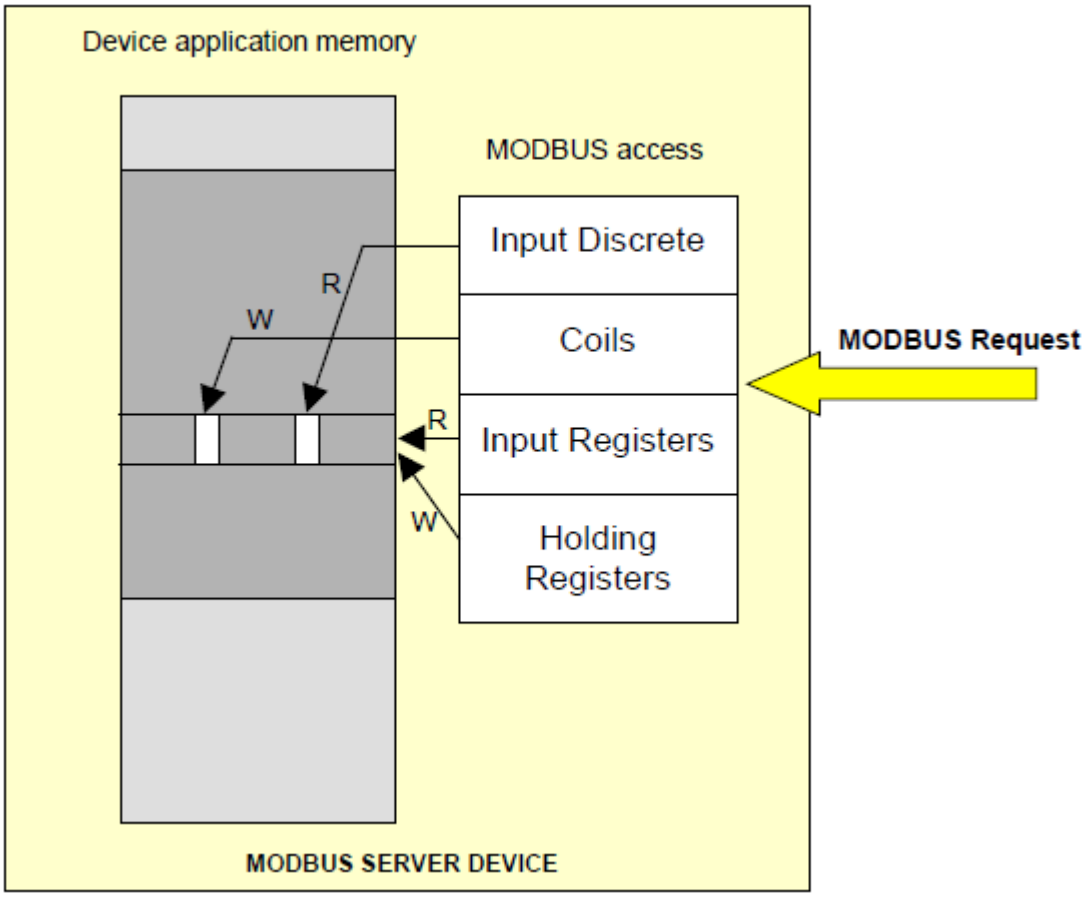
Example1: Device having 4 separate blocks

The example below shows data organization in a device having digital and analogue, inputs and outputs. Each block is separate because data from different blocks have no correlation. Each block is thus accessible with different MODBUS functions.



Example 2: Device having only 1 block

In this example, the device has only 1 data block. The same data can be reached via several MODBUS functions, either via a 16 bit access or via an access bit.



MODBUS Addressing model

The MODBUS application protocol defines precisely PDU addressing rules.

In a MODBUS PDU each data is addressed from 0 to 65535.

It also defines clearly a MODBUS data model composed of 4 blocks that comprises several elements numbered from 1 to n.

In the MODBUS data Model each element within a data block is numbered from 1 to n.

Afterwards the MODBUS data model has to be bound to the device application (IEC-61131 object, or other application model).

The pre-mapping between the MODBUS data model and the device application is totally vendor device specific.

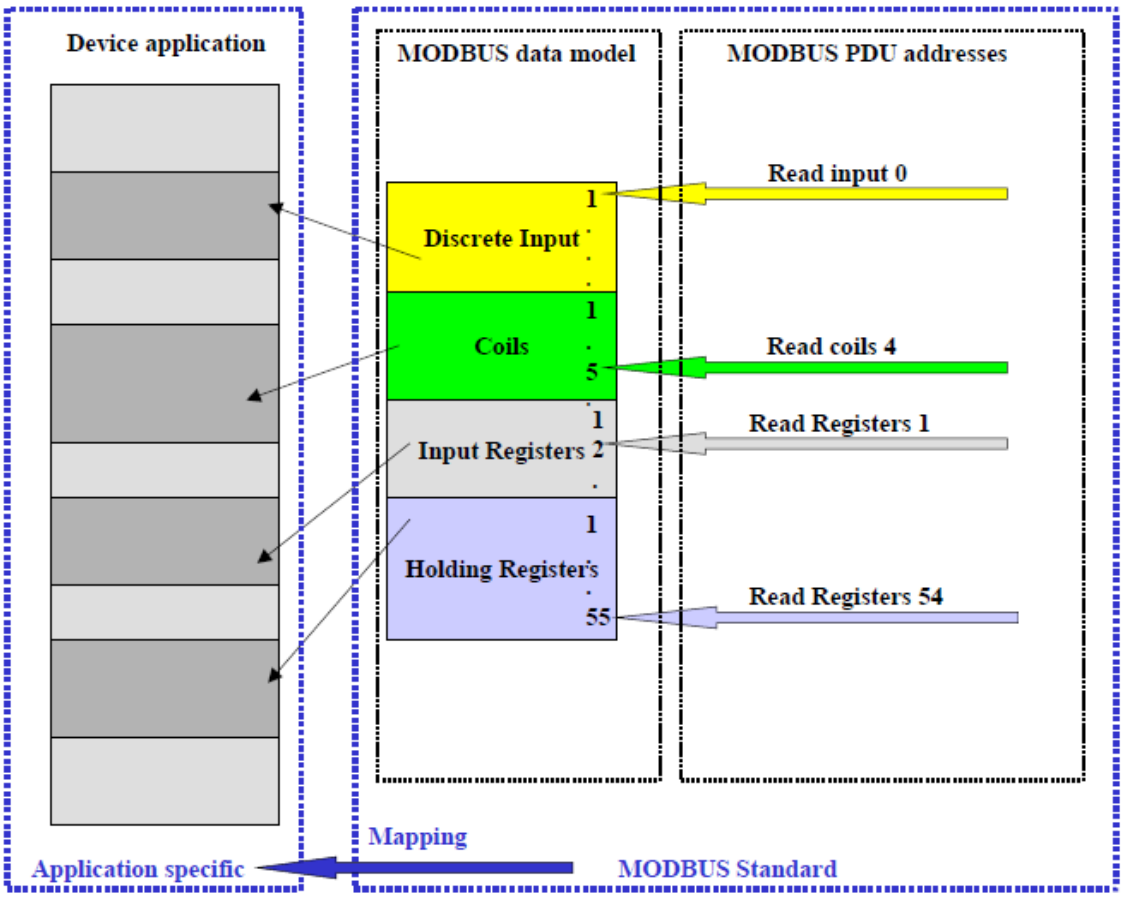


Figure 8 MODBUS Addressing model

The previous figure shows that a MODBUS data numbered X is addressed in the MODBUS PDU X-1.

Function Code Categories

There are three categories of MODBUS Functions codes. They are:

Public Function Codes

• Are well defined function codes.

• guaranteed to be unique,

• validated by the MODBUS.org community,

• Publicly documented

• have available conformance test,

• includes both defined public assigned function codes as well as unassigned function codes reserved for future use.

User-Defined Function Codes

• There are two ranges of user-defined function codes, i.e. 65 to 72 and from 100 to 110 decimal.

• User can select and implement a function code that is not supported by the specification.

• There is no guarantee that the use of the selected function code will be unique

• if the user wants to re-position the functionality as a public function code, he must initiate an RFC to introduce the change into the public category and to have a new public function code assigned.

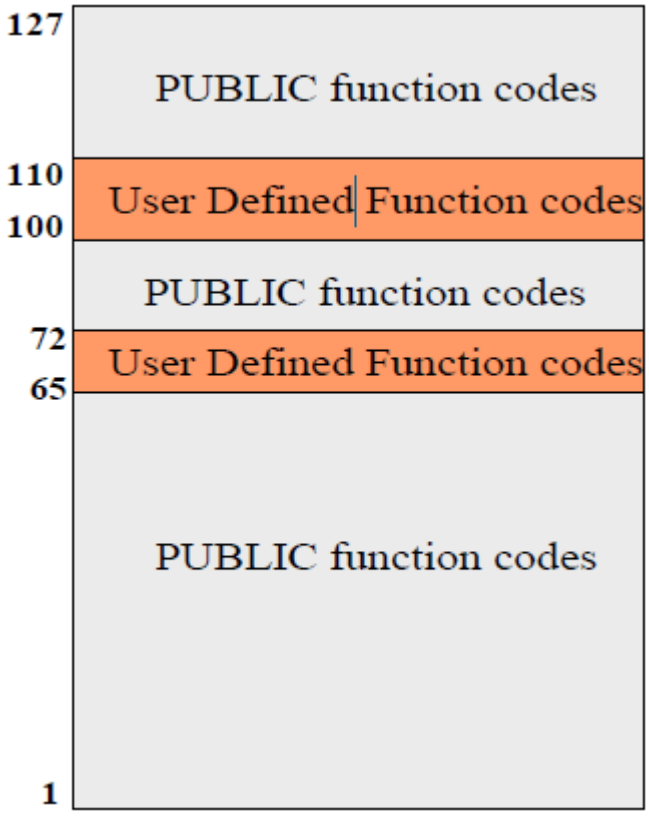
• MODBUS Organization, Inc. expressly reserves the right to develop the proposed RFC.

Reserved Function Codes

• Function Codes currently used by some companies for legacy products and that are not available for public use.

• Informative Note: The reader is asked refer to Annex A (Informative) MODBUS RESERVED FUNCTION CODES, SUBCODES AND MEI TYPES.

MODBUS Function Code Categories



Public Function Code Definition

				Function Codes code	Sub code	(hex)	Section
Data Access	Bit access	Physical Discrete Inputs	Read Discrete Inputs	02		02	6.2
		Internal Bits Or Physical coils	Read Coils	01		01	6.1
			Write Single Coil	05		05	6.5
			Write Multiple Coils	15		0F	6.11
	16 bits access	Physical Input Registers	Read Input Register	04		04	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	03		03	6.3
			Write Single Register	06		06	6.6
			Write Multiple Registers	16		10	6.12
			Read/Write Multiple Registers	23		17	6.17
			Mask Write Register	22		16	6.16
			Read FIFO queue	24		18	6.18
	File record access		Read File record	20		14	6.14
			Write File record	21		15	6.15
Diagnostics			Read Exception status	07		07	6.7
			Diagnostic	08	00-18,20	08	6.8
			Get Com event counter	11		OB	6.9
			Get Com Event Log	12		0C	6.10
			Report Server ID	17		11	6.13
			Read device Identification	43	14	2B	6.21
Other			Encapsulated Interface Transport	43	13,14	2B	6.19
			CANopen General Reference	43	13	2B	6.20

Read Coils

This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. The Request PDU specifies the starting address, i.e. the address of the first coil specified, and the number of coils. In the PDU Coils are addressed starting at zero. Therefore coils numbered 1-16 are addressed as 0-15.

The coils in the response message are packed as one coil per bit of the data field. Status is indicated as 1= ON and 0= OFF. The LSB of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned output quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count Field specifies the quantity of complete bytes of data.

Request

Function code	1 Byte	**0x01**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of coils	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x01**
Byte count	1 Byte	**N***
Coil Status	**n** Byte	n = N or N+1

Read Holding Registers

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

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

Request

Function code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x **N***
Register value	**N*** x 2 Bytes	

*N = Quantity of Registers

Read Discrete Inputs

This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, i.e. the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.

The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count Field specifies the quantity of complete bytes of data.

Request

Function code	1 Byte	**0x02**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Inputs	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x02**
Byte count	1 Byte	**N***
Input Status	**N*** x 1 Byte	

***N** = Quantity of Inputs / 8 if the remainder is different of 0 ⇒ N = N+1

04 (0x04) Read Input Registers

This function code is used to read from 1 to 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

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

Request

Function code	1 Byte	**0x04**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	0x0001 to 0x007D

Response

Function code	1 Byte	**0x04**
Byte count	1 Byte	2 x **N***
Input Registers	**N*** x 2 Bytes	

***N** = Quantity of Input Registers

Data Encoding

• MODBUS uses a ‘big-Endian’ representation for addresses and data items. This means that when a numerical quantity larger than a single byte is transmitted, the most significant byte is sent first. So for example

Register size value
16 - Bits 0x1234 the first byte sent is 0x12 then 0x34

4. SOFTWARE REQUIREMENT SPECIFICATION

Software Requirements

Operating System	:	Windows XP/2003/8/8.1/10 or Linux /Solaris
User Interface	:	AWT, SOCKET IN JAVA
Programming Language	:	Java

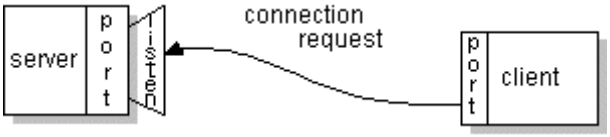
5. CODING & IMPLEMENTATION

5.1 Technologies Used

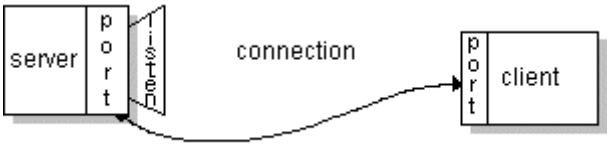
- **Socket programming**

What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java. Net .Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the Server Socket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.

If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation.

Reading from and Writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client,, that connects to an echo server. The echo server receives data from its client and echoes it back. The example EchoServer implements an echo server.

The EchoClient example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server

by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.

Note that the EchoClient example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

Let's walk through the program and investigate the interesting parts. The following statements in the try-with-resources statement in the EchoClient example are critical. These lines establish the socket connection between the client and the server and open a PrintWriter and a BufferedReader on the socket:

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);


try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
)
```

The first statement in the try-with resources statement creates a new Socket object and names it echoSocket. The Socket constructor used here requires the name of the computer and the port number to which you want to connect. The example program uses the first command-line argument as the name of the computer (the host name) and the second command line argument as the port number. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect. For example, if your echo server is running on the computer echoserver.example.com and it is listening on port number 7, first run the following command from the computerechoserver.example.com if you want to use the EchoServer example as your echo server:

java EchoServer 7

Afterward, run the EchoClient example with the following command:

java EchoClient echoserver.example.com 7

The second statement in the try-with resources statement gets the socket's output stream and opens a PrintWriter on it. Similarly, the third statement gets the socket's input stream and opens a BufferedReader on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, the EchoClient example needs to write to the PrintWriter. To get the server's response, EchoClient reads from the BufferedReaderobject stdIn, which is created in the fourth statement in the try-with resources statement. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to thePrintWriter connected to the socket:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
   out.println(userInput);
   System.out.println("echo: " + in.readLine());
}
```

The last statement in the while loop reads a line of information from the BufferedReader connected to the socket. The readLine method waits until the server echoes the information back to EchoClient. When readline returns, EchoClient prints the information to the standard output.

The while loop continues until the user types an end-of-input character. That is, the EchoClient example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. (You can type an end-of-input character by pressing **Ctrl-C**.) The while loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the try-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

AWT CONTROLS:

Components

Modern user interfaces are built around the idea of "components": reusable gadgets

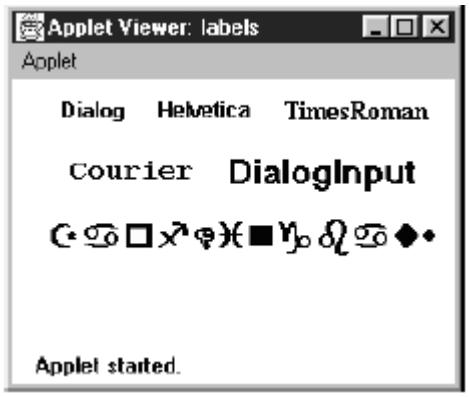
that implement a specific part of the interface. They don't need much introduction:

if you have used a computer since 1985 or so, you're already familiar with buttons, menus, windows, checkboxes, scrollbars, and many other similar items. AWT comes with a repertoire of basic user interface components, along with the machiner y for creating your own components (often combinations of the basic components) and for communicating between components and the rest of the program.
The next few sections summarize the components that are part of AWT. If you're new to AWT, you may find it helpful to familiarize yourself with what's available

before jumping into the more detailed discussions later in this book.

User Input

Java provides several different ways for a user to provide input to an application. The user can type the information or select it from a preset list of available choices. The choice depends primarily on the desired functionality of the program,the user-base, and the amount of back-end processing that you want to do.



The Text Field and Text Area classes

Two components are available for entering keyboard input: Text Field for single line input and Text Area for multi-line input. They provide the means to do things from character-level data validation to complex text editing.

The Button class

A button is little more than a label that you can click on. Selecting a button triggers an event telling the program to go to work.

The Java Management API includes a fancier button (ImageButton) with pictures rather than labels. For the time being, this is a standard extension of Java and not

in the Core API. If you don't want to use these extensions, you'll have to implement an image button yourself.

Layout

Layouts allow you to format components on the screen in a platform-independentway. Without layouts, you would be forced to place components at explicit locations on the screen, creating obvious problems for programs that need to run on multiple platforms. There's no guarantee that a TextArea or a Scrollbar or any other component will be the same size on each platform; in fact, you can bet they won't be. In an effort to make your Java creations portable across multiple platforms,Sun created a LayoutManager inter face that defines methods to reformat the screen based on the current layout and component sizes.

Layout managers try to give programs a consistent and reasonable appearance, regardless of the platform,the screen size, or actions the user might take.

The standard JDK provides five classes that implement the LayoutManager interface.They are FlowLayout, GridLayout, BorderLayout, CardLayout, and Grid-

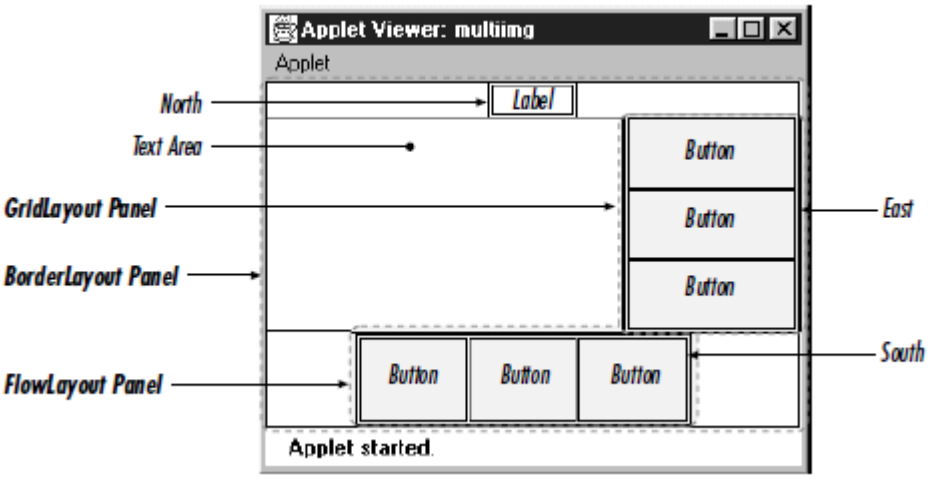
BagLayout. The Java 1.1 JDKincludes the LayoutManager2 inter face. This interface extends the LayoutManager.

GridBagLayout

GridBagLayout is the most sophisticated and complex of the layouts provided in the development kit. With the GridBagLayout, you can organize components in multiple rows and columns, stretch specific rows or columns when space is available,and anchor objects in different corners. You provide all the details of eachcomponent through instances of the GridBagConstraints class.

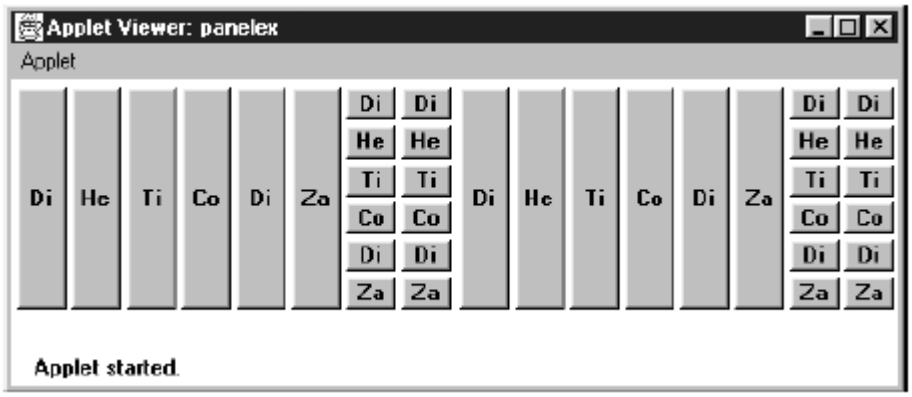
Containers

A Container is a type of component that provides a rectangular area within which other components can be organized by a LayoutManager. Because Container is a subclass of Component, a Container can go inside another Container, which can go inside another Container, and so on, like Russian nesting dolls. Subclassing Container allows you to encapsulate code for the components within it. This allows you to create reusable higher-level objects easily.



Panels

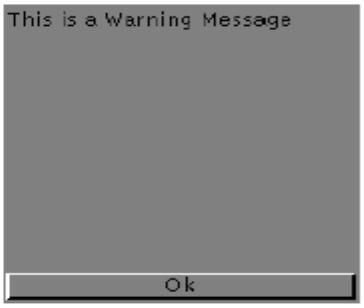
A Panel is the basic building block of an applet. It provides a container with no special features. The default layout for a Panel is FlowLayout.



Windows

A Window provides a top-level window on the screen, with no borders or menu bar. It provides a way to implement pop-up messages, among other things. The default layout for a Window is BorderLayout.

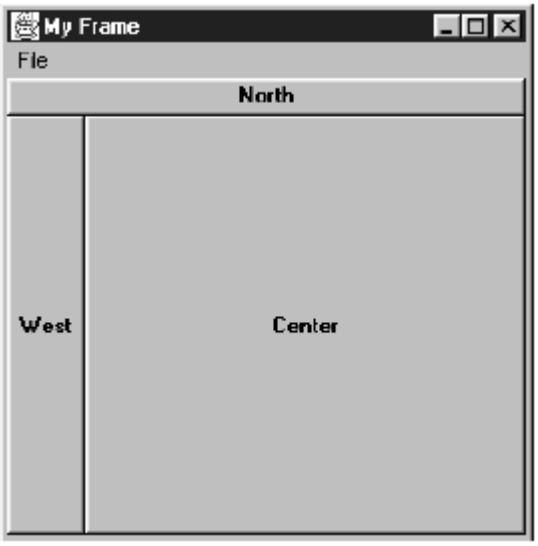
Windows



Motif

Frames

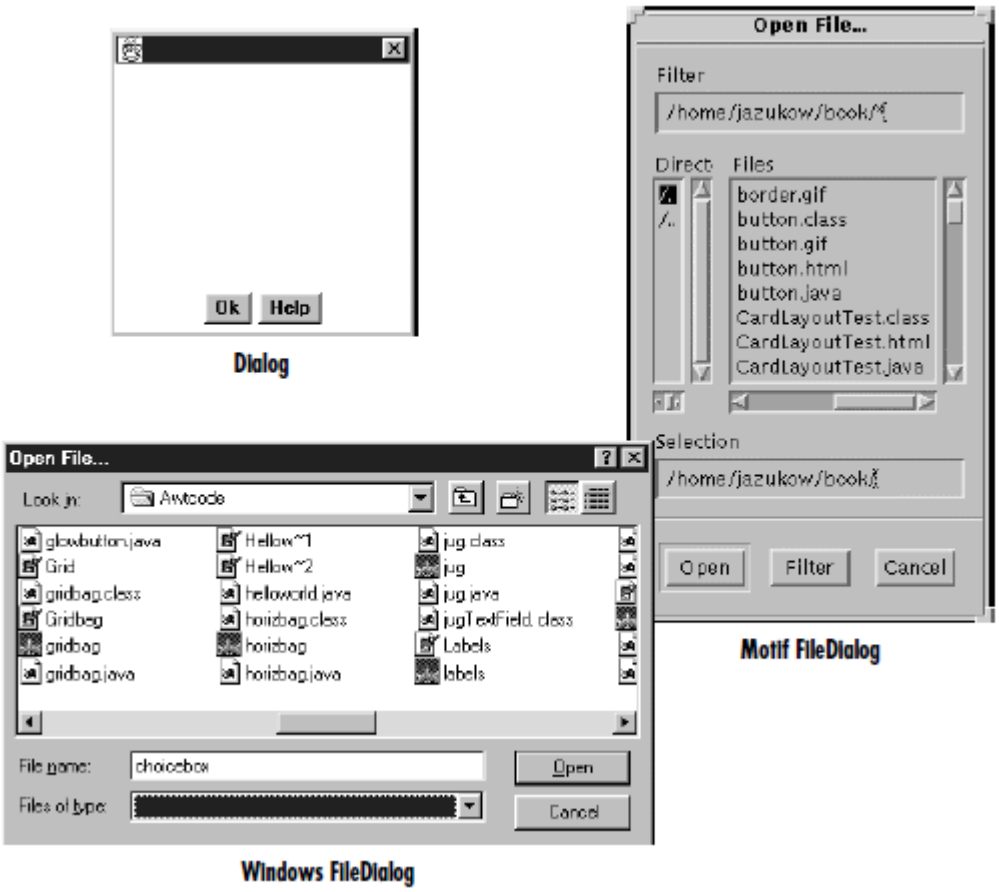
A Frame is a Window with all the window manager's adornments (window title, borders, window minimize/maximize/close functionality) added. It may also include a menu bar. Since Frame subclasses Window, its default layout is BorderLayout. Frame provides the basic building block for screen-oriented applications. Frame allows you to change the mouse cursor, set an icon image, and have menus.



Dialog and File Dialog

A Dialog is a Window that accepts input from the user. BorderLayout is the default layout of Dialog because it subclasses Window. A Dialog is a pop-up used for user interaction; it can be modal to prevent the user from doing anything with the application before responding. A File Dialog provides a prebuilt Dialog box that interacts with the file system. It implements the Open/Save dialog provided by the native windowing system. You will primarily use File Dialog with applications since

there is no guarantee that an applet can interact with the local file system.



EVENT HANDLING:

EVENT:

An event is an object that describes a state change in a source or component

Events can be generated by a person interacting with the elements in GUI. Some of the activities that cause events to be generated are pressing a button, entering a character via keyboard, selecting an item in a list

EVENT SOURCES:

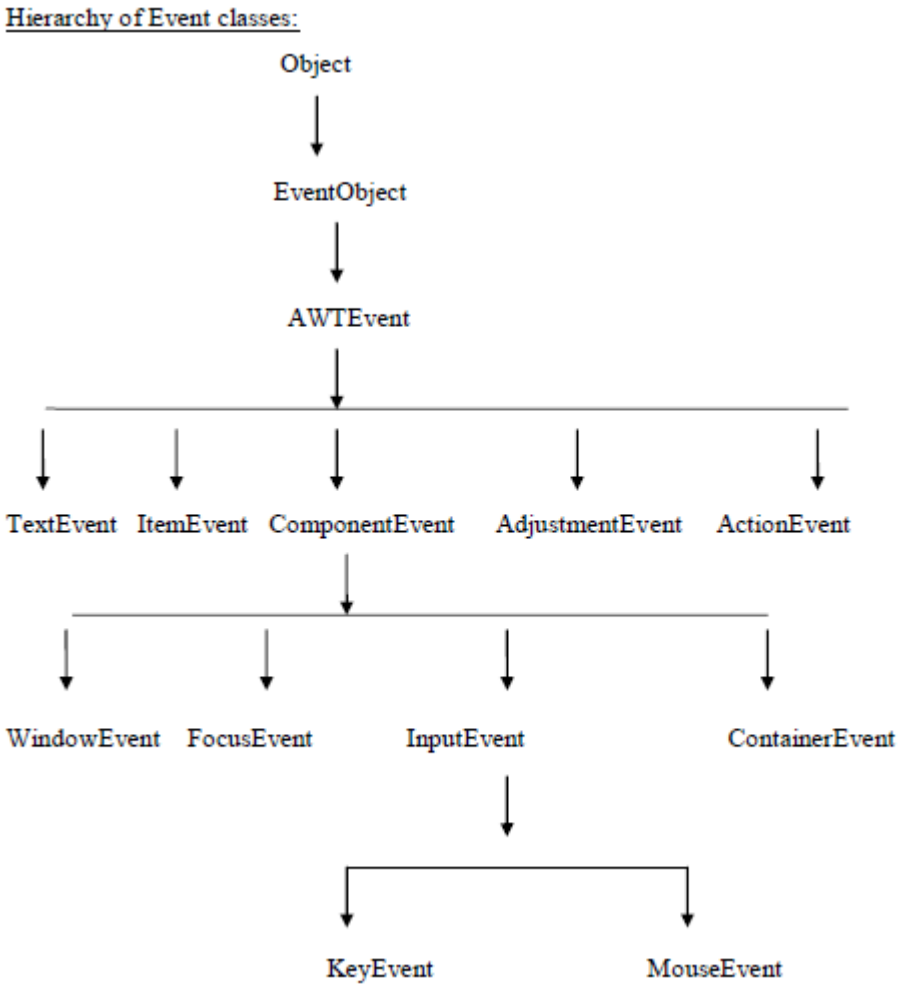
A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate than one type of event.

EVENT LISTENERS:

A listener is an object that is notified when an event occurs. It has two major requirements. First it must have been registered with one or more sources or components to receive notifications about specific types of events. Second it must implement methods to receive and process these notifications. All the listeners are interfaces that contain methods which are to be defined by us if we want handle events. Ex: Mouse Motion Listener, Key Listener …

EVENT CLASSES:

These are the classes which represent events.



Event Class	Listener of the event class	Methods in the Listener
ActionEvent	Action Listener	public void actionPerformed(ActionEvent ae)
MouseEvent	Mouse Listener, MouseWheelListener	public void mouseClicked(MouseEvent me) public void mouseEntered(MouseEvent me) public void mouseExited(MouseEvent me)

Like most windows programming environments, AWT is event driven. When an
Event occurs (for example, the user presses a key or moves the mouse), the environment generates an event and passes it along to a handler to process the event. If nobody wants to handle the event, the system ignores it. Unlike some windowing environments, you do not have to provide a main loop to catch and process all the events, or an infinite busy-wait loop. AWT does all the event management and passing for you.

5.2 CODE

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import javax.swing.JOptionPane;

class AppClient extends Frame implements ActionListener,Runnable
{
Frame f=new Frame("MODBUS PROTOCOL IMPLEMENTATION");
byte buff[]=new byte[12];

String str="";
Button b1,b2,b3,b4,b5,b6,b7;
```

```
TextField tf1,tf2,tf3,tf4,tf5,tf6,tf7,tf8,tf9;

TextArea ta;

Label n,l2,l3,l4,l5,l6,l7,l8,l9,l10,l11,l12;

BufferedReader br;
PrintWriter pw;
OutputStream outStream;

InputStream instream;
Socket S;
Thread th;

AppClient()
{
f.setBackground(new Color(120,100,200));
f.setLayout(new GridLayout(9,4,20,10));

b1=new Button("Connect");
Font myFont1 = new Font("Jokerman", Font.PLAIN,18);
b1.setFont(myFont1);
b1.setBackground(Color.green);

b2=new Button("Read1");
Font myFont2 = new Font("Jokerman", Font.PLAIN,15);
b2.setFont(myFont2);
b2.setBackground(Color.yellow);

b3=new Button("Read2");
Font myFont3 = new Font("Jokerman", Font.PLAIN,15);
b3.setFont(myFont3);
b3.setBackground(Color.yellow);
```

```
b4=new Button("Read3");
Font myFont4 = new Font("Jokerman", Font.PLAIN,15);
b4.setFont(myFont4);
b4.setBackground(Color.yellow);

b5=new Button("Read4");
Font myFont5 = new Font("Jokerman", Font.PLAIN,15);
b5.setFont(myFont5);
b5.setBackground(Color.yellow);

b6=new Button("Clear");
Font myFont6 = new Font("Jokerman", Font.PLAIN,18);
b6.setFont(myFont6);
b6.setBackground(Color.magenta);

b7=new Button("Exit");
Font myFont7 = new Font("Jokerman", Font.PLAIN,18);
b7.setFont(myFont7);
b7.setBackground(Color.red);

tf1=new TextField(20);
tf2=new TextField(20);
tf3=new TextField(20);
tf4=new TextField(20);
tf5=new TextField(20);
tf6=new TextField(20);
tf7=new TextField(20);
tf8=new TextField(20);
tf9=new TextField(20);

ta=new TextArea(20,49);

l2=new Label("Ip-Address:",Label.CENTER);
l2.setFont(new Font("Jokerman", Font.PLAIN, 22));
```

```
l3=new Label("Default Port:502",Label.CENTER);
l3.setFont(new Font("Jokerman", Font.BOLD, 22));
l3.setBackground(Color.cyan);

l4=new Label("READ",Label.CENTER);
l4.setFont(new Font("Jokerman", Font.BOLD, 22));
l4.setBackground(Color.orange);

l5=new Label("INPUT",Label.CENTER);
l5.setFont(new Font("Jokerman", Font.BOLD, 22));
l5.setBackground(Color.orange);

l6=new Label("START",Label.CENTER);
l6.setFont(new Font("Jokerman", Font.BOLD, 22));
l6.setBackground(Color.orange);

l7=new Label("LENGTH",Label.CENTER);
l7.setFont(new Font("Jokerman", Font.BOLD, 22));
l7.setBackground(Color.orange);

l8=new Label("Status Register:",Label.CENTER);
l8.setFont(new Font("Jokerman", Font.PLAIN, 22));

l9=new Label("Coil Register:",Label.CENTER);
l9.setFont(new Font("Jokerman", Font.PLAIN, 22));

l10=new Label("Input Register:",Label.CENTER);
l10.setFont(new Font("Jokerman", Font.PLAIN, 22));

l11=new Label("Holding Register:",Label.CENTER);
l11.setFont(new Font("Jokerman", Font.PLAIN, 22));

l12=new Label("TEXT AREA:",Label.CENTER);
```

```
l12.setFont(new Font("Jokerman", Font.BOLD, 22));
l12.setBackground(Color.orange);

DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd hh:mm:ss");
Date date = new Date();
n=new Label(dateFormat.format(date));
n.setFont(new Font("Jokerman", Font.BOLD,18));

b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
b4.addActionListener(this);
b5.addActionListener(this);
b6.addActionListener(this);
b7.addActionListener(this);

f.addWindowListener(new java.awt.event.WindowAdapter() {
   public void windowClosing(java.awt.event.WindowEvent windowEvent) {
      if (JOptionPane.showConfirmDialog(f,
         "Are you sure to close this window?", "Really Closing?",
         JOptionPane.YES_NO_OPTION,
         JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION){
         System.exit(0);
      }
   }
});

f.add(l2);
f.add(tf1);
f.add(l3);
f.add(b1);

f.add(l5);
f.add(l6);
```

```
f.add(l7);
f.add(l4);

f.add(l8);
f.add(tf2);
f.add(tf3);
f.add(b2);

f.add(l9);
f.add(tf4);
f.add(tf5);
f.add(b3);

f.add(l10);
f.add(tf6);
f.add(tf7);
f.add(b4);

f.add(l11);
f.add(tf8);
f.add(tf9);
f.add(b5);

f.add(l12);
f.add(ta);
f.add(b6);
f.add(b7);
f.add(n);

th=new Thread(this);
th.setDaemon(true);
th.start();
f.setSize(1000,400);
f.setLocation(100,100);
```

```
f.setVisible(true);
f.validate();
f.setResizable(false);//Removes Maximise Button
}
public void actionPerformed(ActionEvent ae)
{
DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
Date date = new Date();
n.setText(dateFormat.format(date));

String str = ae.getActionCommand();

if(str.equals("Connect"))
{
if(validateIPAddress(tf1.getText()))
{
try
{
        ta.setText("Connected");
        S=new Socket(InetAddress.getByName(tf1.getText()),502);
        instream = S.getInputStream();
        outStream = S.getOutputStream();
}
catch(Exception e)
{
}
b1.setLabel("Disconnect");
b1.setBackground(Color.red);
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Enter A Valid IP-Address",
        "Inane warning",
```

```
        JOptionPane.WARNING_MESSAGE);
}
}
else if(str.equals("Disconnect"))
{
try
{
ta.setText("Connection closed");
tf1.setText("");
S.close();
}
catch(Exception e)
{
}
b1.setLabel("Connect");
b1.setBackground(Color.green);
}
else if(str.equals("Read1"))
{
if(b1.getLabel().equals("Disconnect"))
{
if(validateTF(tf2.getText()) && validateTF(tf3.getText()) )
{
        for(int i=0;i<5;i++)
        buff[i]=0;
        buff[5]=6;
        buff[6]=0;
        buff[7]=2;
        buff[8]=(byte) MSB(tf2.getText());
        buff[9]=(byte) (LSB(tf2.getText())-1);
        buff[10]=(byte) MSB(tf3.getText());
        buff[11]=(byte) LSB(tf3.getText());

try
```

```
{
        outStream.write(buff);
}
catch(Exception ex)
{
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Enter A Valid Start/Length Value",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Connect To A Server",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
}
else if(str.equals("Read2"))
{
if(b1.getLabel().equals("Disconnect"))
{
if(validateTF(tf4.getText()) && validateTF(tf5.getText()) )
{
        for(int i=0;i<5;i++)
        buff[i]=0;
        buff[5]=6;
        buff[6]=0;
        buff[7]=1;
```

```
        buff[8]=(byte) MSB(tf4.getText());
        buff[9]=(byte) (LSB(tf4.getText())-1);
        buff[10]=(byte) MSB(tf5.getText());
        buff[11]=(byte) LSB(tf5.getText());
try
{
outStream.write(buff);
}
catch(Exception ex)
{
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Enter A Valid Start/Length Value",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Connect To A Server",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}

}
else if(str.equals("Read3"))
{
if(b1.getLabel().equals("Disconnect"))
{
if(validateTF(tf6.getText()) && validateTF(tf7.getText()) )
```

```
{
        for(int i=0;i<5;i++)
        buff[i]=0;
        buff[5]=6;
        buff[6]=0;
        buff[7]=4;
        buff[8]=(byte) MSB(tf6.getText());
        buff[9]=(byte) (LSB(tf6.getText())-1);
        buff[10]=(byte) MSB(tf7.getText());
        buff[11]=(byte) LSB(tf7.getText());
try
{
outStream.write(buff);
}
catch(Exception ex)
{
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Enter A Valid Start/Length Value",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}

}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Connect To A Server",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
```

```
}
else if(str.equals("Read4"))
{
if(b1.getLabel().equals("Disconnect"))
{
if(validateTF(tf8.getText()) && validateTF(tf9.getText()) )
{
for(int i=0;i<5;i++)
buff[i]=0;
buff[5]=6;
buff[6]=0;
buff[7]=3;
buff[8]=(byte) MSB(tf8.getText());
buff[9]=(byte) (LSB(tf8.getText())-1);
buff[10]=(byte) MSB(tf9.getText());
buff[11]=(byte) LSB(tf9.getText());
try
{
outStream.write(buff);
}
catch(Exception ex)
{
}
}
else
{
        JOptionPane.showMessageDialog(f,
        "Please Enter A Valid Start/Length Value",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
}
else
{
```

```
        JOptionPane.showMessageDialog(f,
        "Please Connect To A Server",
        "Inane warning",
        JOptionPane.WARNING_MESSAGE);
}
}
else if(str.equals("Clear"))
{
ta.setText(" ");
}
else if(str.equals("Exit"))
{
        if (JOptionPane.showConfirmDialog(f,
        "Are you sure to close this window?", "Really Closing?",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION)
        System.exit(0);
}
}
public void run()
{
while(true)
{
try
{
byte[] buffer = new byte[100];
int[] b1=new int[100];
int noofBytes=instream.read(buffer);
int[] b2=new int[100];
if((noofBytes>0))
{
ta.setText("");
for(int i=0;i<noofBytes;i++)
ta.append(buffer[i]+",");
```

```
int k=0;
if(buff[7]==2||buff[7]==1)
{
for(int buffindex=9;buffindex<noofBytes;buffindex++)
{
b1[k]=buffer[buffindex];
int x=b1[k];
int y=0,z=0;
int binary[]=new int[8];
if (x == 0)
{
for(y=0;y<8;y++)
binary[y]=0;
}
while (x > 0)
{
int rem = x % 2;
binary[y] = rem;
x = x / 2;
y++;
}
for(z=0;z<8;z++)
ta.append(binary[z]+",");
k++;
}
}
else if(buff[7]==4||buff[7]==3)
{
k=0;
int l=0;
for(int buffindex=9;buffindex<noofBytes;buffindex++)
{
b1[k]=buffer[buffindex];
k++;
```

```
}
int i=0;
while(i<k)
{
b2[l]=(b1[i]*256)+b1[i+1];
i=i+2;
ta.append(b2[l]+",");
l++;
}
}
}
}
catch(Exception e)
{
}
}
}
public boolean validateIPAddress(String ipAddress)
{
String tokens[]=ipAddress.split("\\.");
if(tokens.length!=4)
return false;
for(int i=0;i<4;i++)
{
for(char c:tokens[i].toCharArray())
{
int value = (int) c;
if ((value >= 65 && value <= 90) || (value >= 97 && value <= 122))
{
return false;
}
}
}
if(ipAddress==null)
```

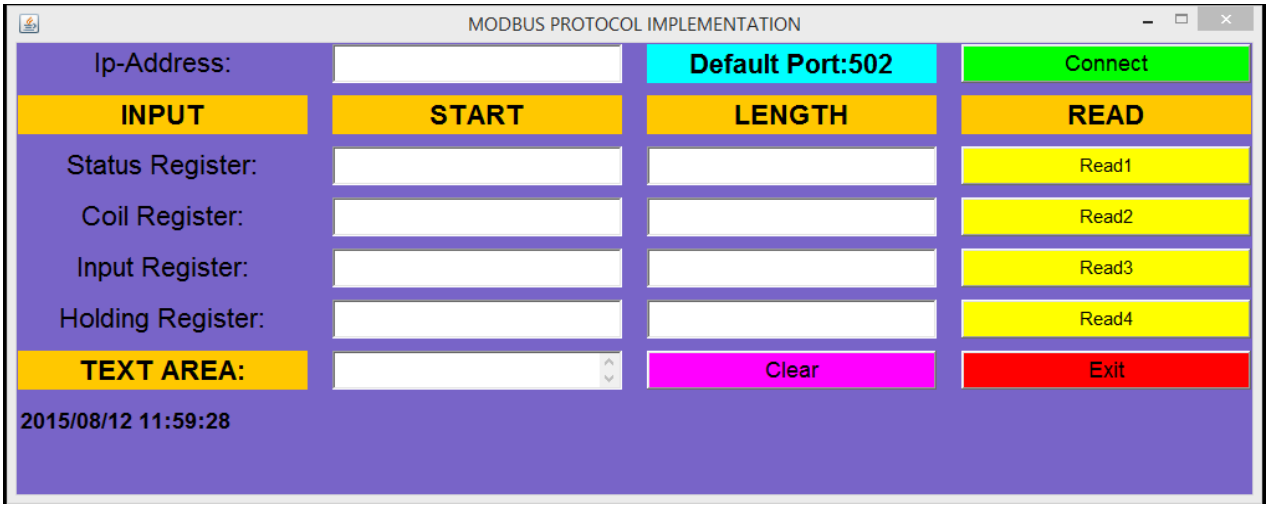
```
return false;
for(String str:tokens)
{
int i=Integer.parseInt(str);
if((i<0)||(i>255))
return false;
}
return true;
}
public boolean validateTF(String text)
{
for(char c:text.toCharArray())
{
int value = (int) c;
if ((value >= 65 && value <= 90) || (value >= 97 && value <= 122))
{
return false;
}
}
if(text.trim().isEmpty())
return false;
int i=Integer.parseInt(text);
if((i<1)||(i>256))
return false;
return true;
}
public int MSB(String tf)
{
int n =Integer.parseInt(tf);
int i,k=0,sum=0;
int binary[]=new int[16];
int msb[]=new int[8];
int val[]=new int[8];
if (n == 0)
```

```
{
for(i=0;i<16;i++)
binary[i]=0;
}
i=15;
while (n > 0)
{
int rem = n % 2;
binary[i] = rem;
n = n / 2;
i--;
}
for(i=0;i<8;i++)
msb[i]=binary[i];
for(int j=7;j>=0;j--)
{
val[k]=(int) Math.pow(2,j);
k++;
}
for(i=0;i<8;i++)
if(msb[i]==1)
sum=sum+val[i];
return sum;
}
public int LSB(String tf)
{
int n =Integer.parseInt(tf);
int i,k=0,sum=0;
int binary[]=new int[16];
int lsb[]=new int[8];
int val[]=new int[8];
if (n == 0)
{
for(i=0;i<16;i++)
```

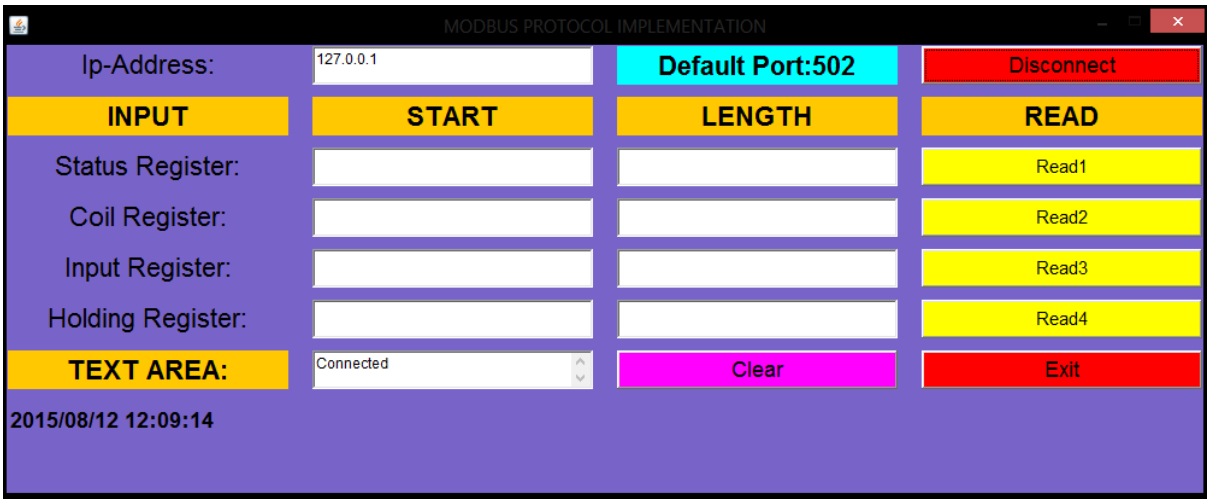
```
binary[i]=0;
}
i=15;
while (n > 0)
{
int rem = n % 2;
binary[i] = rem;
n = n / 2;
i--;
}
for(i=8;i<16;i++)
{
lsb[k]=binary[i];
k++;
}
k=0;
for(int j=7;j>=0;j--)
{
val[k]=(int) Math.pow(2,j);
k++;
}
for(i=0;i<8;i++)
if(lsb[i]==1)
sum=sum+val[i];
return sum;
}
public static void main(String args[])throws IOException
{
new AppClient();
}
}
```

6. SCREENSHOTS

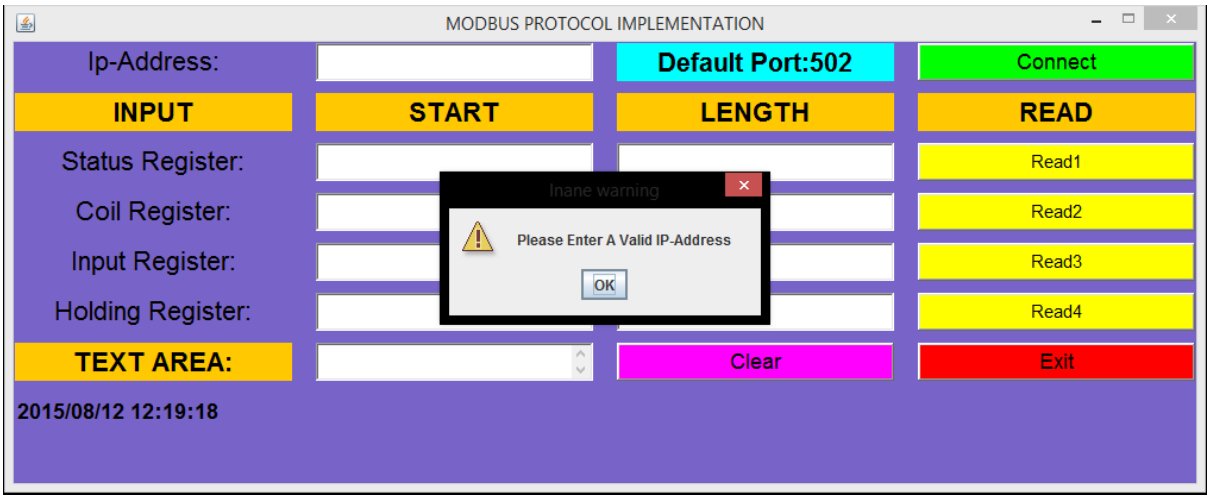
1. GUI

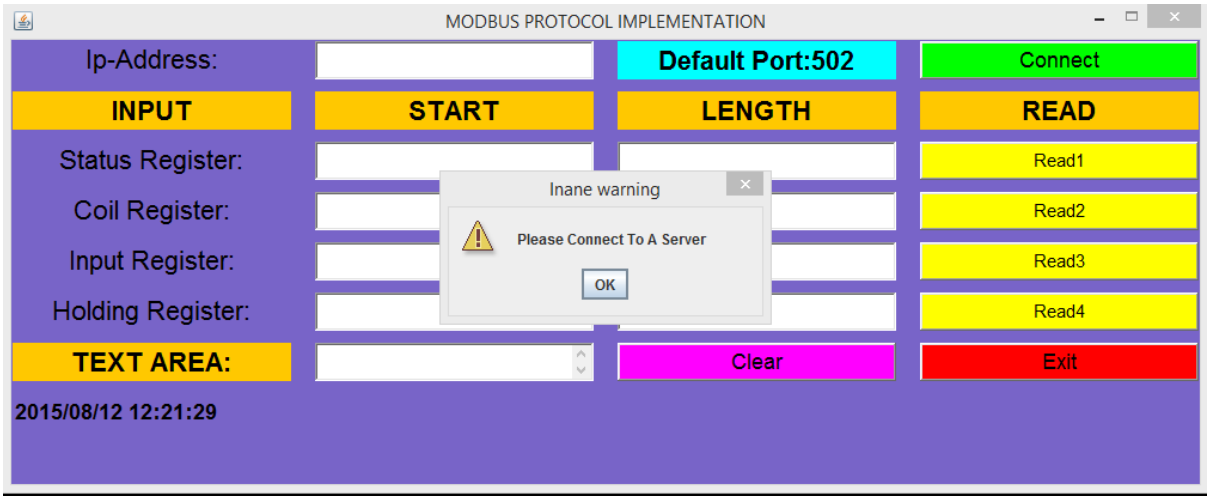


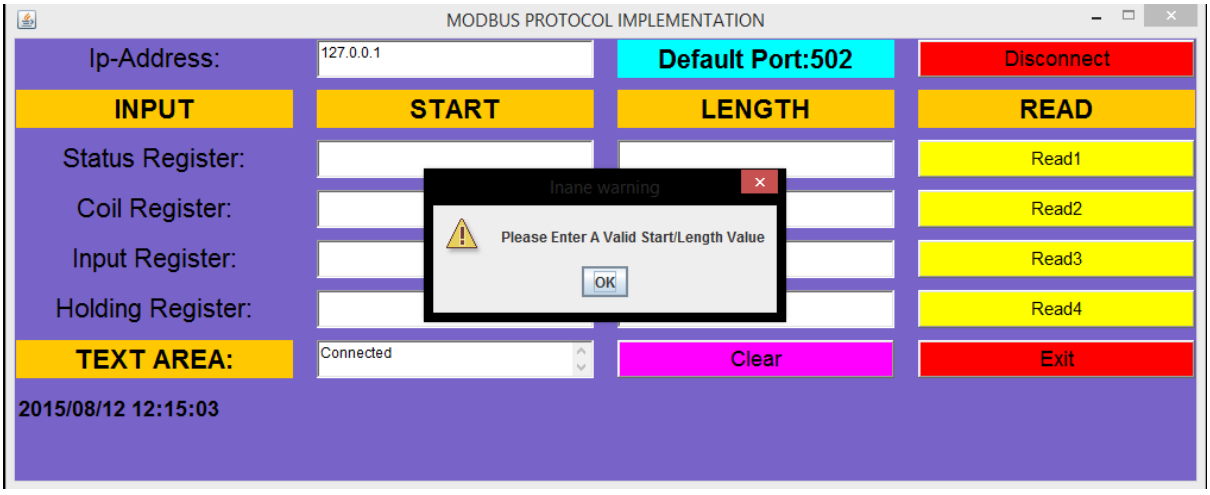
2. Connecting to the simulator

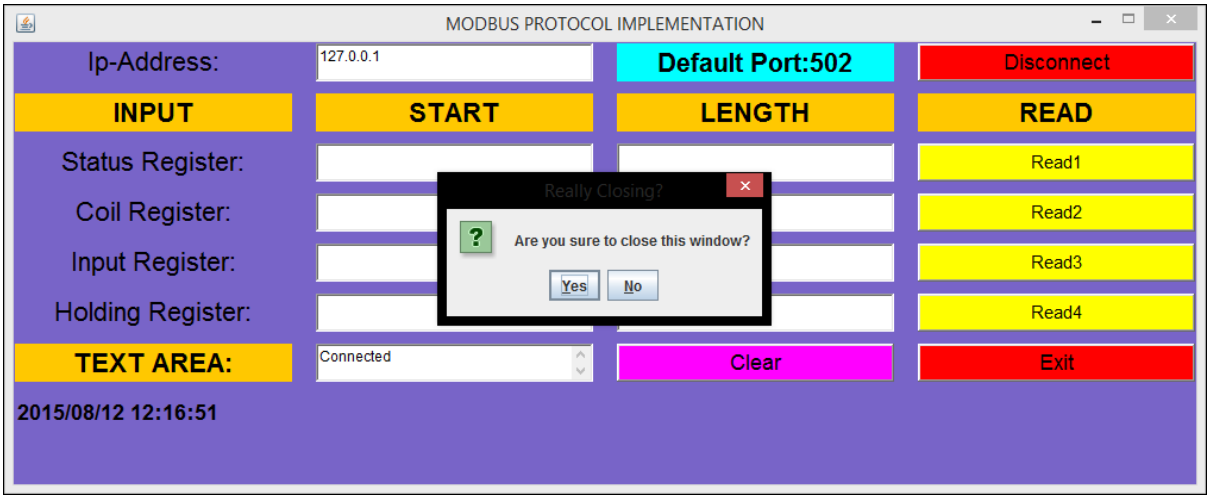


3. Dialog Boxes








MODBUS PROTOCOL IMPLEMENTATION
Ip-Address:
127.0.0.1
Default Port:502
Disconnect
INPUT
START
LENGTH
READ
Status Register:
Coil Register:
Input Register:
Holding Register:
Read1
Read2
Read3
Read4
Really Closing?
Are you sure to close this window?
Yes
No
TEXT AREA:
Connected
Clear
Exit
2015/08/12 12:16:51

7. REFERENCES

WEBSITES:

i) WWW.STACKOVERFLOW.COM
ii) WWW.W3SCHOOLS.COM
iii) FONTAWESOME.IO/ICON/COG/
iv) GETBOOTSTRAP.COM
v) WIKIPEDIA.ORG

BOOKS:

i) PROGRAMMING WORLD WIDE WEB
- ROBERT W. SEBESTA