Atamate Level 2 Programming Test

C

1. Introduction

The test problem is to write an emulator of an embedded system that will listen for UDP messages on a given port and interpret and act on signals received in a proprietary protocol. The signals are commands from a client.

2. Business Rules

1. Each message contains a code to identify the type of message. The message code must be one of 0x11, 0x21, 0x31, 0x41 or 0x51.
2. Each message contains a sequence number. Each message must be acknowledged with the next number in the sequence. Any message received with an unexpected sequence number should be ignored.
3. The protocol has four types of message.

   Ack Signal

   This signal is sent by the emulator to acknowledge receipt of a Relay or Get State signal or by the client to acknowledge receipt of a Relay State signal

code	1 byte 0x11	Required
sequence nr	1 byte, the next message sequence number	Required

   Relay Signal

   This signal is sent to the emulator to instruct it to set the state of one of two relays.

code	1 byte 0x21	Required
sequence nr	1 byte, the message sequence number	Required
count	1 byte, number of relay settings in the message	Required
relay number	1 byte, the relay to operate: 0 or 1	Required
relay value	1 byte, the state to set: 0 (open) or 1 (closed)	Required
...	Repeated for each relay state in the message	

   Get State Signal

   This signal is sent by the client to request a Relay State signal..

code	1 byte, 0x31	Required
sequence nr	1 byte, the message sequence number	Required

   Relay State Signal

   This signal is sent by the emulator to report the state of its relays.

code	1 byte, 0x41	Required
sequence nr	1 byte, the message sequence number	Required
relay state	1 byte, a bitmap of the relay state. Bit 0 gives the state	Required

of relay 0, bit 1 gives the state of relay 1

Quit Signal

This signal is sent by the client to stop the emulator.

code	1 byte, 0x51	Required
sequence nr	1 byte, the message sequence number	Required

2.1. Examples

Relay signal sent from client

```
Client                        Emulator
  |                             |
  |------Relay 0: Close-------->|
  |       21:99:01:00:01        |
  |                             |
  |<-----Ack--------------------|
  |       11:9a                 |
```

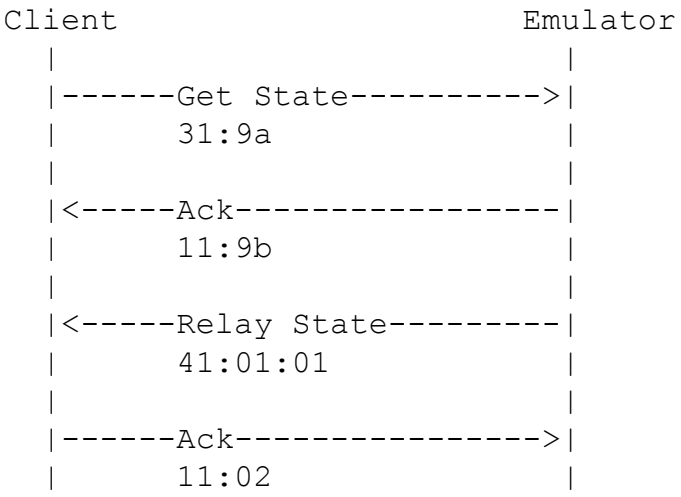
The client sends the Relay signal

0x21	The message code
0x99	The message sequence number (153, for example)
0x01	Number of relay settings in message (1, for example)
0x00	First relay to set (0, for example)
0x01	Relay state to set (1->Closed, for example)

The emulator replies with the Ack signal

0x11	The message code
0x9a	The next message sequence number (154, for example)

Get State signal sent from client



The client sends the Get State signal

0x31	The message code
0x9a	The message sequence number (154, for example)

The emulator replies with the Ack signal

0x11	The message code
0x9b	The next message sequence number (155, for example)

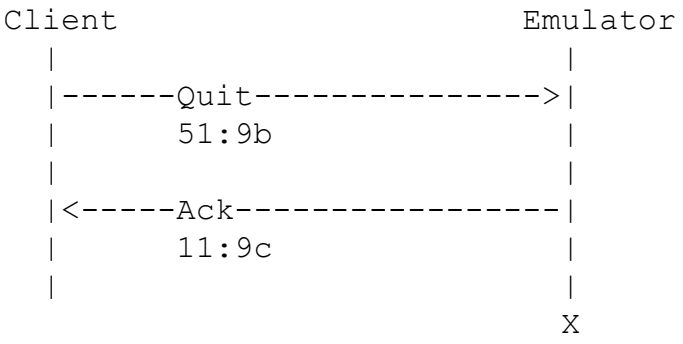
Next, the emulator sends the Relay State signal

0x41	The message code
0x01	The message sequence number (1, for example. Note, there is a different sequence number for the two directions of message flow)
0x01	The bitmap of relay states (0: Closed, 1: Open, for example)

The client replies with the Ack signal

0x11	The message code
0x02	The next message sequence number (2, for example)

Quit signal sent from client



The client sends the Quit signal

0x51	The message code
0x9b	The message sequence number (155, for example)

The emulator replies with the Ack signal

0x11	The message code
0x9c	The next message sequence number (156, for example)

The emulator quits

2.2. Output

As this is an emulator, it doesn't have real relays to open and close. Instead, the emulator should print out diagnostics as it receives messages. In the examples above the messages should be:

```
Emulator Started
...Relay 0: Close
...Received Get State
...Quit
```

3. Coding Style

I am looking for the following good coding practices.

1. Working code.
2. Good clear naming of structures, functions and variables.
3. Comments.
4. Error checking and a good strategy for returning error codes.
5. Unit tests. I used the CMocka framework.
6. Defensive coding. Don't assume that messages are in the right format or that methods are called with valid parameters.
7. Flexible programming. Avoid duplication and allow for possible future changes.