

MOTION Blinds WLAN Integration Guide

- Scope and Purpose..... 3
- Definition..... 3
- The integration process..... 6
- Interface..... 6
 - Get device list..... 6
 - Reference..... 6
 - Heartbeat..... 8
 - Reference..... 8
 - Child-device control..... 9
 - Reference..... 12
 - Child-device status report..... 21
 - Reference..... 23
 - Reference (for TDBU blinds only)..... 23
 - Child-device status query..... 24
 - Reference..... 26
- Encryption and decryption algorithm..... 27

Version Record

Date	Version	Description
2019-12-30	v0.99	First release
2020-11-04	v1.02	TDBU blinds feature added, fixed other wrong descriptions

Scope and Purpose

This guide introduces how to integrate motorized window covering products (For instance, Wi-Fi bridge & 433MHz Radio motors) into a 3rd party automation system via WLAN access.

Note: The 'Motion' APP must configure all of the setups include Bridge pairing and blinds adding. The 3rd party is unable to add/edit/delete the Wi-Fi bridge and the child devices/blinds. It only has child devices control permission via WLAN.

Definition

Client/Server

Client: The 3rd party automation system

Server: Wi-Fi bridge

Server UDP multicast address

Server receive: [238.0.0.18:32100]

Server sending: [238.0.0.18:32101]

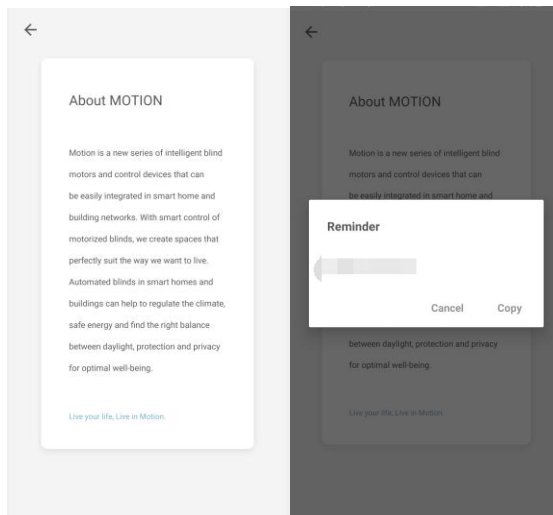
messageID/msgID

After each interface operation, messageID/msgID should be increased, otherwise **Server** will not respond.

KEY/token/AccessToken:

KEY is a 16-byte length string. It assigns by Motion APP. **KEY** and **token** use to create a 16-byte length **AccessToken**, **Server** responses only when it receives the correct **AccessToken**, the **AccessToken** algorithm reference is at the end of the document.

Please quickly tap the 'Motion APP About' page 5 times to get **KEY**.



token is a 16-byte length string. The **Client** can capture **token** in the interface ‘**Device discovering**’ or ‘**Heartbeat.**’

Accesstoken calculation logic & reference

Please use the URL below to verify your encryption.

<https://www.devglan.com/online-tools/aes-encryption-decryption>

Fill **token** in ‘Enter text to be Encrypted,’ fill **KEY** in ‘Enter Secret Key.’ (‘Select Mode’ == ‘ECB’ ; ‘Key Size in Bits’ == ‘128’; ‘Output Text Format’ == ‘Hex’)

For example

token : 37412C478E0FBEAB

KEY : 74ae544c-d16e-4c

AccessToken : 8570A96BC18ADB21D1FC155B24ECFD73

AES Online Encryption

Enter text to be Encrypted

OR

Select Mode

ECB

Key Size in Bits

128


Enter Secret Key

Output Text Format: ☐ Base64 ☒ Hex

Encrypt

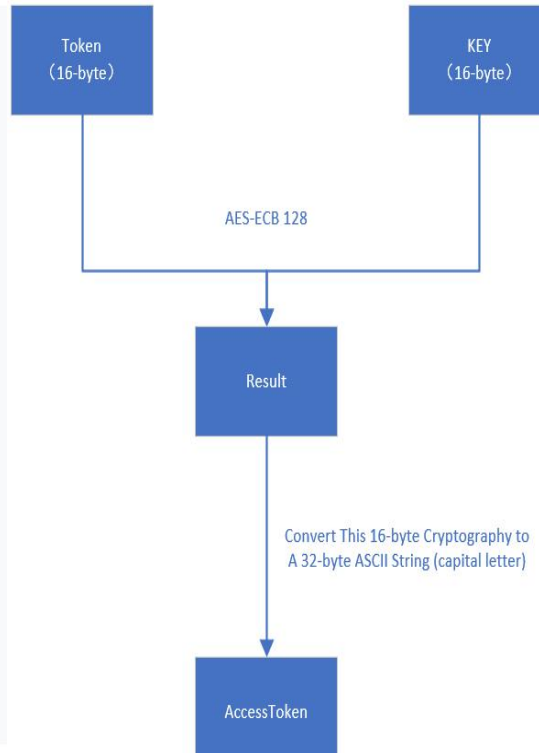
Sell your handmade goods.

Start free trial



AES Encrypted Output:

8570A96BC18ADB21D1FC155B24ECFD7358B74E6E
CD4FE6306385732B841E5F28



The integration process

1. Device discovering using 'Get device list'.
2. Calculate 'Access**token**' using **KEY** and **token**.
3. Control child devices using 'Child device control', 'Child device status query', and 'Child device status report'.

Interface

Get device list

Get device list from Wi-Fi Bridge, includes Wi-Fi Bridge and child devices.

1. **Client** discovers Wi-Fi bridge & child devices using UDP unicast [Client_IP:32100] or UDP multicast [238.0.0.18:32100].
2. **Server** uploads lists by UDP unicast [Client_IP:32101].

Interface parameters:

Name	Type	Value	Description
msgType	String	GetDeviceList	Get device list
msgID	String		Message-ID (Timestamp)

Interface response

Name	Type	Value	Description
msgType	String	GetDeviceListAck	Upload device list
mac	String		Wi-Fi Bridge MAC address
deviceType	String		10000000: 433Mhz radio motor 02000001: Wi-Fi Bridge
ProtocolVersion	String		WLAN access protocol version
token	String		Token
data	JsonArray		Child device list

JsonArray

Name	Type	Value	Description
mac	String		Child device mac
deviceType	String		10000000: 433Mhz radio motor

Reference

Request data

```
//From Client_IP:PORT to 238.0.0.18:32100 or Form Client_IP:PORT to Server_IP:32100
{
  "msgType":"GetDeviceList",
  "msgID":"20200321134209916"
}
```

Response data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType":"GetDeviceListAck",
  "mac":"500291b691fd",
  "deviceType":"02000001",
  "ProtocolVersion":"0.9",
  "token":"37412C478E0FBEAB",
  "data":[
    {
      "mac":"500291b691fd",
      "deviceType":"02000001"
    },
    {
      "mac":"500291b691fd005f",
      "deviceType":"10000000"
    },
    {
      "mac":"500291b691fd0060",
      "deviceType":"10000000"
    }
  ]
}
```

Heartbeat

Keep alive, **Server** heartbeats per 30~60 seconds using UDP multicast [238.0.0.18:32101].

Interface parameters:

Name	Type	Value	Description
msgType	String	Heartbeat	
mac	String		Wi-Fi Bridge MAC address
deviceType	String		Device application type 02000001: Wi-Fi Bridge 10000000: 433Mhz radio motor
token	String		Token
data	JsonArray		Info

JsonArray

Name	Type	Value	Description
currentState	enum	1 2 3	1 : Working 2 : Pairing 3 : Updating
numberOfDevices	Int		Number of Child devices
RSSI	Int		Wi-Fi connection strength

Reference

Response data

```
// From Server_IP:32100 to 238.0.0.18:32101
{
  "msgType": "Heartbeat",
  "mac": "b4e62db27481",
  "deviceType": "02000001",
  "token": "37412C478E0FBEAB",
  "data": {
    "currentState": 1,
    "numberOfDevices": 3,
    "RSSI": -21
  }
}
```


Child-device control

Client controls child-devices using **'WriteDevice'** message. (UDP unicast [Client_IP:32100] or UDP multicast [238.0.0.18:32100])

Server response using **'WriteDeviceAck'**, and returns the child-device current status. (UDP unicast [Client_IP])

Interface parameters:

Name	Type	Value	Description
msgType	String	WriteDevice	Child-device control
mac	String		Message-ID (Timestamp)
deviceType	String		02000001: Wi-Fi Bridge 10000000: 433Mhz radio motor
AccessToken	String		
msgID	String		Timestamp
data	JsonArray		Control command

JsonArray

Name	Type	Value	Description
operation	enum	0 1 2 3 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
targetPosition	Int		0-100
targetAngle	Int		0-180

JsonArray (for TDBU Blinds only)

Name	Type	Value	Description
operation_T operation_B	enum	0 1 2 3 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
targetPosition_T targetPosition_B	Int		0-100

Interface response

Name	Type	Value	Description
msgType	String	WriteDeviceAck	
mac	String		Wi-Fi Bridge MAC address
deviceType	String		10000000: 433Mhz radio motor
msgID	String		Timestamp
data	JsonArray		

JsonArray

Name	Type	Value	Description
type	Int		1:Roller Blinds 2:Venetian Blinds 3:Roman Blinds 4:Honeycomb Blinds 5:Shangri-La Blinds 6:Roller Shutter 7:Roller Gate 8:Awning 10:Day&night Blinds 11:Dimming Blinds 12:Curtain 13:Curtain(Open Left) 14:Curtain(Open Right)
operation	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
currentPosition	Int		0-100
currentAngle	Int		0-180
currentState	enum	0 1 2 3 4	0: No limits 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor 1: DC Motor
batteryLevel	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

JSONArray (for TDBU blinds only)

Name	Type	Value	Description
type	Int		9:TDBU
operation_T operation_B	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
exist_subid			
currentPosition_T currentPosition_B	Int		0-100
currentState_T currentState_B	enum	0 1 2 3 4	0: No limits 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor 1: DC Motor
batteryLevel_T batteryLevel_B	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

Reference

Percentage control

Only bi-directional devices supported (wireless Mode == 1 required).

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT -> Server_IP:32100
{
  "msgType": "WriteDevice",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "AccessToken": "0D5D443049491C20988B46AC54323BA2",
  "msgID": "20200331103919505",
  "data": {
    "targetPosition": 44
  }
}
```

Respond data

```
// Form Server_IP:32100 to Client_IP:PORT
{
  "msgType": "WriteDeviceAck",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "msgID": "20200331103919505",
  "data": {
    "type": 13,
    "operation": 2,
    "currentPosition": 59,
    "currentAngle": 180,
    "currentState": 3,
    "voltageMode": 0,
    "batteryLevel": 811,
    "wirelessMode": 1,
    "RSSI": -73
  }
}
```

Percentage control (for TDBU blinds only)

Only bi-directional devices supported (wireless Mode == 1 required).

Request data

```
// Form Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType" : "WriteDevice",
  "mac" : "b4e62db27481005d",
  "deviceType" : "10000001"
  "AccessToken" : "0D5D443049491C20988B46AC54323BA2",
  "msgID" : "20201103151055138",
  "data" : {
    "targetPosition_B" : 75,
    "targetPosition_T" : 35
  },
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType":"WriteDeviceAck",
  "mac":"b4e62db27481005d",
  "deviceType":"10000001",
  "msgID":"20201103151055138",
  "data":{
    "type":9,
    "exist_subid":1,
    "operation_T":2,
    "operation_B":2,
    "currentPosition_T":35,
    "currentPosition_B":95,
    "currentState_T":3,
    "currentState_B":3,
    "voltageMode":1,
    "batteryLevel_T":836,
    "batteryLevel_B":836,
    "wirelessMode":1,
    "RSSI":-40
  }
}
```

Rotation/Angle control

Only bi-directional devices supported (wireless Mode == 1 required).

Request data

```
// Form Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType": "WriteDevice",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "AccessToken": "0D5D443049491C20988B46AC54323BA2",
  "msgID": "20200331105628663",
  "data": {
    "targetAngle": 78
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType": "WriteDeviceAck",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "msgID": "20200331105628663",
  "data": {
    "type": 13,
    "operation": 0,
    "currentPosition": 44,
    "currentAngle": 0,
    "currentState": 3,
    "voltageMode": 0,
    "batteryLevel": 811,
    "wirelessMode": 1,
    "RSSI": -68
  }
}
```

Open/Up

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType": "WriteDevice",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "AccessToken": "0D5D443049491C20988B46AC54323BA2",
  "msgID": "20200331105646317",
  "data": {
    "operation": 1
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType": "WriteDeviceAck",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "msgID": "20200331105646317",
  "data": {
    "type": 13,
    "operation": 2,
    "currentPosition": 47,
    "currentAngle": 77,
    "currentState": 3,
    "voltageMode": 0,
    "batteryLevel": 811,
    "wirelessMode": 1,
    "RSSI": -67
  }
}
```

Open/Up (for TDBU blinds only)

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType" : "WriteDevice",
  "mac" : "b4e62db27481005d",
  "deviceType" : "10000001",
  "AccessToken" : "0D5D443049491C20988B46AC54323BA2",
  "msgID" : "20201104095555138",
  "data" : {
    "operation_B" : 1,
    "operation_T" : 1
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType":"WriteDeviceAck",
  "mac":"b4e62db27481005d",
  "deviceType":"10000001",
  "msgID":"20201104095555138",
  "data":{
    "type":9,
    "exist_subid":1,
    "operation_T":2,
    "operation_B":2,
    "currentPosition_T":21,
    "currentPosition_B":53,
    "currentState_T":3,
    "currentState_B":3,
    "voltageMode":1,
    "batteryLevel_T":835,
    "batteryLevel_B":835,
    "wirelessMode":1,
    "RSSI":-42
  }
}
```


Close/Down

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType": "WriteDevice",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "AccessToken": "0D5D443049491C20988B46AC54323BA2",
  "msgID": "20200331105735705",
  "data": {
    "operation": 0
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType": "WriteDeviceAck",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "msgID": "20200331105735705",
  "data": {
    "type": 13,
    "operation": 2,
    "currentPosition": 0,
    "currentAngle": 0,
    "currentState": 3,
    "voltageMode": 0,
    "batteryLevel": 811,
    "wirelessMode": 1,
    "RSSI": -68
  }
}
```

Close/Down (for TDBU blinds only)

Request data

```
// Form Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType" : "WriteDevice",
  "mac" : "b4e62db27481005d",
  "deviceType" : "10000001",
  "AccessToken" : "0D5D443049491C20988B46AC54323BA2",
  "msgID" : "20201104100955213",
  "data" : {
    "operation_B" : 0,
    "operation_T" : 0
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType":"WriteDeviceAck",
  "mac":"b4e62db27481005d",
  "deviceType":"10000001",
  "msgID":"20201104100955213",
  "data":{
    "type":9,
    "exist_subid":1,
    "operation_T":2,
    "operation_B":2,
    "currentPosition_T":0,
    "currentPosition_B":0,
    "currentState_T":3,
    "currentState_B":3,
    "voltageMode":1,
    "batteryLevel_T":835,
    "batteryLevel_B":835,
    "wirelessMode":1,
    "RSSI":-37
  }
}
```

Stop

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType": "WriteDevice",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "AccessToken": "0D5D443049491C20988B46AC54323BA2",
  "msgID": "20200331105833122",
  "data": {
    "operation": 2
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType": "WriteDeviceAck",
  "mac": "b4e62db27481001f",
  "deviceType": "10000000",
  "msgID": "20200331105833122",
  "data": {
    "type": 13,
    "operation": 2,
    "currentPosition": 100,
    "currentAngle": 180,
    "currentState": 3,
    "voltageMode": 0,
    "batteryLevel": 811,
    "wirelessMode": 1,
    "RSSI": -70
  }
}
```

Stop (for TDBU blinds only)

Request data

```
// Form Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType" : "WriteDevice",
  "mac" : "b4e62db27481005d",
  "deviceType" : "10000001",
  "AccessToken" : "0D5D443049491C20988B46AC54323BA2",
  "msgID" : "20201104101726121",
  "data" : {
    "operation_B" : 2,
    "operation_T" : 2
  }
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType":"WriteDeviceAck",
  "mac":"b4e62db27481005d",
  "deviceType":"10000001",
  "msgID":"20201104101726121",
  "data":{
    "type":9,
    "exist_subid":1,
    "operation_T":2,
    "operation_B":2,
    "currentPosition_T":68,
    "currentPosition_B":100,
    "currentState_T":3,
    "currentState_B":3,
    "voltageMode":1,
    "batteryLevel_T":834,
    "batteryLevel_B":834,
    "wirelessMode":1,
    "RSSI":-43
  }
}
```

Child-device status report

The child-device reports status after it stops running(UPD multicast [238.0.0.18:32101]).

Only bi-directional motors supported(wireless Mode == 1 required).

Interface parameters:

Name	Type	Value	Description
msgType	String	Report	Status report
mac	String		Message-ID (Timestamp)
deviceType	String		
msgID	String		Timestamp
data	JsonArray		

JsonArray

Name	Type	Value	Description
type	Int		1:Roller Blinds 2:Venetian Blinds 3:Roman Blinds 4:Honeycomb Blinds 5:Shangri-La Blinds 6:Roller Shutter 7:Roller Gate 8:Awning 9:TDBU 10:Day&night Blinds 11:Dimming Blinds 12:Curtain 13:Curtain(Open Left) 14:Curtain(Open Right)
operation	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
currentPosition	Int		0-100
currentAngle	Int		0-180
currentState	enum	0 1 2 3 4	0: Not limit 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor

			1: DC Motor
batteryLevel	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

JSONArray (for TDBU blinds only)

Name	Type	Value	Description
type	Int		9:TDBU
operation	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
exist_subid			
currentPosition_T currentPosition_B	Int		0-100
currentAngle_T currentAngle_B	Int		0-180
currentState_T currentState_B	enum	0 1 2 3 4	0: Not limit 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor 1: DC Motor
batteryLevel_T batteryLevel_B	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

Reference

```
// From Server_IP:32100 to 238.0.0.18:32101
{
  "msgType": "Report",
  "mac": "b4e62db274810049",
  "deviceType": "10000000",
  "msgID": "20201104110407805",
  "data": {
    "type": 2,
    "operation": 2,
    "currentPosition": 75,
    "currentAngle": 180,
    "currentState": 3,
    "voltageMode": 1,
    "batteryLevel": 782,
    "wirelessMode": 1,
    "RSSI": -50
  }
}
```

Reference (for TDBU blinds only)

```
// From Server_IP:32100 to 238.0.0.18:32101
{
  "msgType": "Report",
  "mac": "b4e62db27481005d",
  "deviceType": "10000001",
  "msgID": "20201104103250515",
  "data": {
    "type": 9,
    "exist_subid": 1,
    "operation_T": 2,
    "operation_B": 2,
    "currentPosition_T": 9,
    "currentPosition_B": 47,
    "currentState_T": 3,
    "currentState_B": 3,
    "voltageMode": 1,
    "batteryLevel_T": 834,
    "batteryLevel_B": 834,
    "wirelessMode": 1,
    "RSSI": -39
  }
}
```

Child-device status query

Client query using **'ReadDevice'** message. (UDP unicast [Server_IP:32100] or UDP multicast [238.0.0.18:32100])

Server response using **'ReadDeviceAck'**, and returns the child-device current status. (UDP unicast [Client_IP:32101])

Only bi-directional devices supported (wireless Mode == 1).

Interface parameters:

Name	Type	Value	Description
msgType	String	ReadDevice	Child-device control
mac	String		
deviceType	String		10000000: 433Mhz radio motor
msgID	String		Timestamp

Interface response

Name	Type	Value	Description
msgType	String	ReadDeviceAck	Status report
mac	String		
deviceType	String		10000000: 433Mhz radio motor
msgID	String		Timestamp
data	JSONArray		

JSONArray

Name	Type	Value	
type	Int		1:Roller Blinds 2:Venetian Blinds 3:Roman Blinds 4:Honeycomb Blinds 5:Shangri-La Blinds 6:Roller Shutter 7:Roller Gate 8:Awning 10:Day&night Blinds 11:Dimming Blinds 12:Curtain 13:Curtain(Open Left) 14:Curtain(Open Right)
operation	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop

			5: Status query
currentPosition	Int		0-100
currentAngle	Int		0-180
currentState	enum	0 1 2 3 4	0: Not limit 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor 1: DC Motor
batteryLevel	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

JSONArray (for TDBU blinds only)

Name	Type	Value	Description
type	Int		9:TDBU
operation_T operation_B	enum	0 1 2 5	0: Close/Down 1: Open/Up 2: Stop 5: Status query
exist_subid			
currentPosition_T currentPosition_B	Int		0-100
currentState_T currentState_B	enum	0 1 2 3 4	0: No limits 1: Top-limit detected 2: Bottom-limit detected 3: Limits detected 4: 3 rd -limit detected
voltageMode	enum	0 1	0: AC Motor 1: DC Motor
batteryLevel_T batteryLevel_B	Int		Power voltage (DC motor only)
wirelessMode	enum	0 1 2 3	0: Uni-direction 1: Bi-direction 2: Bi-direction (mechanical limits) 3:Others
RSSI	Int		Radio signal strength

Reference

Request data

```
// From Client_IP:PORT to 238.0.0.18:32100 or From Client_IP:PORT to Server_IP:32100
{
  "msgType": "ReadDevice",
  "mac": "b4e62db274810049",
  "deviceType": "10000000",
  "msgID": "20201104105826121"
}
```

Respond data

```
// From Server_IP:32100 to Client_IP:PORT
{
  "msgType": "ReadDeviceAck",
  "mac": "b4e62db274810049",
  "deviceType": "10000000",
  "msgID": "20201104105826121",
  "data": {
    "type": 2,
    "operation": 2,
    "currentPosition": 75,
    "currentAngle": 180,
    "currentState": 3,
    "voltageMode": 1,
    "batteryLevel": 782,
    "wirelessMode": 1,
    "RSSI": -48
  }
}
```

Encryption and decryption algorithm

Step1: Use **KEY**(From the 'Motion APP About' page) to make an AES-ECB 128 encryption for the received **token** (the 16-byte length string from 'Device discovering' or 'Heartbeat') and create a 16 byte-length cryptograph。
Step2: Convert this 16-byte cryptograph to a 32-byte ASCII string (capital letter)

The reference code

```
#include "stdint.h"
#include "stdio.h"
#include "mgOS.h"

// Optional length 128,192,256
#define AES_KEY_LENGTH 128

// Encryption & decryption mode
#define AES_MODE_ECB 0 // Electronic codebook
#define AES_MODE_CBC 1 // Cipher-block chaining
#define AES_MODE AES_MODE_ECB

#define Nk (AES_KEY_LENGTH / 32) // Nk = 4, Key string length, 4 bytes (1 word)
#define Nb 4 // Nb = 4,

// Nr:Encryption rounds
#if AES_KEY_LENGTH == 128
#define Nr 10
#elif AES_KEY_LENGTH == 192
#define Nr 12
#elif AES_KEY_LENGTH == 256
#define Nr 14
#else
#error AES_KEY_LENGTH must be 128, 192 or 256 bools!
#endif

// GF(28) polynomial
#define BPOLY 0x1B // Lower 8 bools of (x^8 + x^4 + x^3 + x + 1), ie. (x^4 + x^3 + x + 1).

// AES subkey table, it requires a 176 bytes space when using a 128-byte length key.
static uint8_t g_roundKeyTable[4*Nb*(Nr+1)];

// Encryption SBox
static const const uint8_t SBox[256] =
{
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};

// Decryption SBox
```

```

static const const uint8_t InvSBox[256] =
{
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};

//=====
// Function:    RotationWord
// Description:  Cyclic right shift of "word" data
// Input:       pWord -- 4-byte data to be shifted to the right
// Output:      pWord -- 4-byte data after right shift
// Return:      Null
//=====
static void RotationWord(uint8_t *pWord)
{
    uint8_t temp = pWord[0];
    pWord[0] = pWord[1];
    pWord[1] = pWord[2];
    pWord[2] = pWord[3];
    pWord[3] = temp;
}

//=====
// Function:    XorBytes
// Description:  Batch XOR two sets of data
// Input:       pData1 -- The first set of data to be XORed
//              pData2 -- The second set of data to be XORed
//              nCount -- Data length to XOR
// Output:      pData1 -- The result after XOR
// Return:      Null
//=====
static void XorBytes(uint8_t *pData1, const uint8_t *pData2, uint8_t nCount)
{
    uint8_t i;

    for (i = 0; i < nCount; i++)
    {
        pData1[i] ^= pData2[i];
    }
}

//=====
// Function:    AddRoundKey
// Description:  Add (exclusive OR) subkey to the intermediate state data, the data length is 16 bytes
// Input:       pState -- Status data
//              pRoundKey -- subkey data
// Output:      pState -- State data after subkey
// Return:      Null
//=====
static void AddRoundKey(uint8_t *pState, const uint8_t *pRoundKey)
{
    XorBytes(pState, pRoundKey, 4*Nb);
}

```

```

//=====
// Function:    SubBytes
// Description:  Replace state data with S box
// Input:    pState -- Status data
//           nCount -- subkey data
//           blnvert -- Whether to use the reverse S box (used when decrypting)
// Output:    pState -- Status data after replacement.
// Return:    Null
//=====
static void SubBytes(uint8_t *pState, uint8_t nCount, bool blnvert)
{
    uint8_t i;
    const uint8_t const *pSBox = blnvert ? InvSBox : SBox;

    for (i = 0; i < nCount; i++)
    {
        pState[i] = pSBox[pState[i]];
    }
}

//=====
// Function:    ShiftRows
// Description:  Row migration status data
// Input:    pState -- Status data
//           blnvert -- Whether to move backwards (used when decrypting).
// Output:    pState -- Status data after row migration.
// Return:    Null. Null
//=====
static void ShiftRows(uint8_t *pState, bool blnvert)
{
    // Note: Status data stored as 'row'
    uint8_t r;    // row
    uint8_t c;    // column
    uint8_t temp;
    uint8_t rowData[4];

    for (r = 1; r < 4; r++)
    {
        // Back up one row of data
        for (c = 0; c < 4; c++)
        {
            rowData[c] = pState[r + 4*c];
        }

        temp = blnvert ? (4 - r) : r;
        for (c = 0; c < 4; c++)
        {
            pState[r + 4*c] = rowData[(c + temp) % 4];
        }
    }
}

//=====
// Function:    GfMultBy02
// Description:  Multiplication by 2 in GF (28)
// Input:    num -- multiplier
// Output:    Null
// Return:    The result of num multiplied by 2
//=====
static uint8_t GfMultBy02(uint8_t num)
{
    if ((num & 0x80) == 0)
    {
        num = num << 1;
    }
    else
    {
        num = (num << 1) ^ BPOLY;
    }
}

```

```

    return num;
}

//=====================================================
// Function:    MixColumns
// Description:  Mixed state column data
// Input:       pState -- Status data
//              blInvert -- Whether to mix backwards (used when decrypting).
// Output:      pState -- State data after mixing columns
// Return:      Null
//=====================================================
static void MixColumns(uint8_t *pState, bool blInvert)
{
    uint8_t i;
    uint8_t temp;
    uint8_t a0Pa2_M4; // 4(a0 + a2)
    uint8_t a1Pa3_M4; // 4(a1 + a3)
    uint8_t result[4];

    for (i = 0; i < 4; i++, pState += 4)
    {
        temp = pState[0] ^ pState[1] ^ pState[2] ^ pState[3];
        result[0] = temp ^ pState[0] ^ GfMultBy02((uint8_t) (pState[0] ^ pState[1]));
        result[1] = temp ^ pState[1] ^ GfMultBy02((uint8_t) (pState[1] ^ pState[2]));
        result[2] = temp ^ pState[2] ^ GfMultBy02((uint8_t) (pState[2] ^ pState[3]));
        result[3] = temp ^ pState[3] ^ GfMultBy02((uint8_t) (pState[3] ^ pState[0]));

        if (blInvert)
        {
            a0Pa2_M4 = GfMultBy02(GfMultBy02((uint8_t) (pState[0] ^ pState[2])));
            a1Pa3_M4 = GfMultBy02(GfMultBy02((uint8_t) (pState[1] ^ pState[3])));
            temp = GfMultBy02((uint8_t) (a0Pa2_M4 ^ a1Pa3_M4));
            result[0] ^= temp ^ a0Pa2_M4;
            result[1] ^= temp ^ a1Pa3_M4;
            result[2] ^= temp ^ a0Pa2_M4;
            result[3] ^= temp ^ a1Pa3_M4;
        }

        memcpy(pState, result, 4);
    }
}

//=====================================================
// Function:    BlockEncrypt
// Description:  Encrypt single block data
// Input:       pState -- Status data
// Output:      pState -- Encrypted status data
// Return:      Null
//=====================================================
static void BlockEncrypt(uint8_t *pState)
{
    uint8_t i;

    AddRoundKey(pState, g_roundKeyTable);

    for (i = 1; i <= Nr; i++)    // i = [1, Nr]
    {
        SubBytes(pState, 4*Nb, 0);
        ShiftRows(pState, 0);

        if (i != Nr)
        {
            MixColumns(pState, 0);
        }

        AddRoundKey(pState, &g_roundKeyTable[4*Nb*i]);
    }
}

```

```

//=====
// Function:   BlockDecrypt
// Description: Decrypt single block data
// Input:    pState -- Status data
// Output:    pState -- Decrypted single block data
// Return:    Null
//=====
static void BlockDecrypt(uint8_t *pState)
{
    uint8_t i;

    AddRoundKey(pState, &g_roundKeyTable[4*Nb*Nr]);

    for (i = Nr; i > 0; i--) // i = [Nr, 1]
    {
        ShiftRows(pState, 1);
        SubBytes(pState, 4*Nb, 1);
        AddRoundKey(pState, &g_roundKeyTable[4*Nb*(i-1)]);

        if (i != 1)
        {
            MixColumns(pState, 1);
        }
    }
}

//=====
// Function:   AES_Init
// Description: Initialize, perform extended key operations here
// Input:    pKey -- The original key, it must be AES_KEY_LENGTH/8 bytes long.
// Output:    Null
// Return:    Null
//=====
void AES_Init(const void *pKey)
{
    // Extended key
    uint8_t i;
    uint8_t *pRoundKey;
    uint8_t Rcon[4] = {0x01, 0x00, 0x00, 0x00};

    memcpy(g_roundKeyTable, pKey, 4*Nk);

    pRoundKey = &g_roundKeyTable[4*Nk];

    for (i = Nk; i < Nb*(Nr+1); pRoundKey += 4, i++)
    {
        memcpy(pRoundKey, pRoundKey - 4, 4);

        if (i % Nk == 0)
        {
            RotationWord(pRoundKey);
            SubBytes(pRoundKey, 4, 0);
            XorBytes(pRoundKey, Rcon, 4);

            Rcon[0] = GfMultBy02(Rcon[0]);
        }
        else if (Nk > 6 && i % Nk == Nb)
        {
            SubBytes(pRoundKey, 4, 0);
        }

        XorBytes(pRoundKey, pRoundKey - 4*Nk, 4);
    }
}

//=====
// Function:   AES_Encrypt

```

```

// Description:    Encrypted data
// Input:   pPlainText -- Plain text, the data to be encrypted, whose length is nDataLen bytes.
//          nDataLen   -- The data length, in bytes, must be an integral multiple of AES_KEY_LENGTH/8.
//          pIV        -- Initialization vector, if using ECB mode, can be set to NULL.
// Output:   pCipherText -- The ciphertext, which is encrypted by plaintext, can be the same as pPlainText.
// Return:   Null。 Null
//=====
#if AES_MODE == AES_MODE_CBC
void AES_Encrypt(const uint8_t *pPlainText, uint8_t *pCipherText, uint16_t nDataLen, const uint8_t *pIV)
{
    uint16_t i;

    if (pPlainText != pCipherText)
    {
        memcpy(pCipherText, pPlainText, nDataLen);
    }

    for (i = nDataLen/(4*Nb); i > 0; i--, pCipherText += 4*Nb)
    {
        XorBytes(pCipherText, pIV, 4*Nb);
        BlockEncrypt(pCipherText);
        pIV = pCipherText;
    }
}
#else
void AES_Encrypt(const uint8_t *pPlainText, uint8_t *pCipherText, uint16_t nDataLen)
{
    uint16_t i;

    if (pPlainText != pCipherText)
    {
        memcpy(pCipherText, pPlainText, nDataLen);
    }

    for (i = nDataLen/(4*Nb); i > 0; i--, pCipherText += 4*Nb)
    {
        BlockEncrypt(pCipherText);
    }
}
#endif

//=====
// Function:    AES_Decrypt
// Description:    Decrypt data
// Input:   pCipherText -- Ciphertext, the data to be decrypted, is nDataLen bytes in length.
//          nDataLen   -- The data length, in bytes, must be an integral multiple of AES_KEY_LENGTH/8.
//          pIV        -- Initialization vector, if using ECB mode, can be set to NULL.
// Output:   pPlainText -- Plain text, that is, the data decrypted by ciphertext, can be the same as pCipherText.
// Return:   Null
//=====
#if AES_MODE == AES_MODE_CBC
void AES_Decrypt(uint8_t *pPlainText, const uint8_t *pCipherText, uint16_t nDataLen, const uint8_t *pIV)
{
    uint16_t i;

    if (pPlainText != pCipherText)
    {
        memcpy(pPlainText, pCipherText, nDataLen);
    }

    // Decrypt from the last piece of data, so there is no need to open up space to save IV
    pPlainText += nDataLen - 4*Nb;
    for (i = nDataLen/(4*Nb); i > 0; i--, pPlainText -= 4*Nb)
    {
        BlockDecrypt(pPlainText);

        if (i == 1)
        {
            // Last block of data

```



```

        XorBytes(pPlainText, pIV, 4*Nb);
    }
    else
    {
        XorBytes(pPlainText, pPlainText - 4*Nb, 4*Nb);
    }
}
}
#else
void AES_Decrypt(uint8_t *pPlainText, const uint8_t *pCipherText, uint16_t nDataLen)
{
    uint16_t i;

    if (pPlainText != pCipherText)
    {
        memcpy(pPlainText, pCipherText, nDataLen);
    }

    // Decrypt from the last piece of data, so there is no need to open up space to save IV
    pPlainText += nDataLen - 4*Nb;
    for (i = nDataLen/(4*Nb); i > 0; i--, pPlainText -= 4*Nb)
    {
        BlockDecrypt(pPlainText);
    }
}
#endif

//Encryption
void mgAesEncrypt(uint8_t *in, uint8_t *out, uint8_t *key, int length)
{
    AES_Init(key);
    AES_Encrypt(in, out, length);
}

//Decryption
void mgAesDecrypt(uint8_t *in, uint8_t *out, uint8_t *key, int length)
{
    AES_Init(key);
    AES_Decrypt(out, in, length);
}

static void test()
{
    // Test AES algorithm
    char KEY[] = "Q2W3E4R5T6Y7U8I9";
    char token[] = "5Axy9la9kR0tFTFs";
    char AccessToken[17] = {0};

    uint8_t O[16];

    mgAesEncrypt((uint8_t *)token, O, (uint8_t *)KEY, 16);

    sprintf(
        AccessToken,
        "%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X%02X",
        O[0], O[1], O[2], O[3], O[4], O[5], O[6], O[7], O[8], O[9], O[10], O[11], O[12], O[13], O[14], O[15]);

    printf("AccessToken:%s", AccessToken);
}

```