iDevOS MODBUS RTU PROTOCOL

The Modbus RTU protocol is used with a single Master device and multiple Slave devices normally over an RS485 network. Single Slave networks can use RS232 or similar asynchronous connections.

The Master sends a request to a Slave which then responds with data and/or an acknowledgement.
The Master must continually poll slave devices which may have variable input data.

Modbus RTU uses 3.5 data bytes as intervals between each packet.
Up to 247 Slave devices can be connected to the network.
Each Slave can have up to 65,536 bytes of registers / memory.

Allocation of functionality for each register is subject to the specific design of the Slave device. Therefore the Master must have the flexibility to amend register location and codes in the event of future slave substitution.

iDevOS provides both 1 and 2 dimensional array solutions to reduce memory usage and allow simplified polling of slave devices. The programmer has total control over base address and offset values per slave.

The iDevOS operating system supports Modbus RTU as a Master or Slave device.
The implementation supports the following Modbus command functions to read and write.

01H = READ COILS
02H = READ DISCRETE INPUTS
03H = READ HOLDING REGISTER
04H = READ INPUT REGISTER
05H = WRITE SINGLE COIL
06H = WRITE REGISTER
0FH = WRITE MULTIPLE COILS
10H = WRITE MULTIPLE REGISTERS

REGISTERS are 16 bit values of 0000H to FFFFH
COILS and DISCRETE INPUTS are single bit Boolean values 0 or 1 with multiples of 8 packed into 1 byte U8.

Any of the 4 asynchronous serial interfaces AS1, AS2, RS2 and RS4 on iSMART TFT products can use Modbus as a protocol. Built In registers used for controlling Modbus are prefixed with MB_AS1, MB_AS2, MB_RS2 and as shown below for MB_RS4.

```
MB_RS4_MODE      //set the iSMART TFT as Master (0) or Slave (1)
MB_RS4_BASE      //set the base address from 0000H to FFFFH
MB_RS4_REGS      //point to a previously defined array of 1 or 2 dimensions
MB_RS4_COILS     //same as REGS
MB_RS4_SLAVEID   //define the identity of the slave device from 1 to 247
MB_RS4_FUNC      //set the Modbus command to be used
MB_RS4_REGOFF    //set the offset from the base address. Base+offset=actual
MB_RS4_REGCNT    //set how many 16bit registers to write/read, maximum 125
MB_RS4_COILOFF   //set the offset from the base address. Base+offset=actual
MB_RS4_COILCNT   //set the number of 8 bit coil registers to read/write
MB_RS4_STATUS    //read the current status
MB_RS4_ERROR     //contains the last error code
MB_RS4_TIMEOUT   //set the Modbus timeout period
MB_RS4_DATAOK    //name of a function used when data OK acknowledged
MB_RS4_DATAERR   //name of a function used when data ERROR acknowledged
MB_RS4_VALID_DATA  //point to an array containing valid slave data
MB_RS4_VALID_REG   //point to an array containing valid slave addresses
MB_RS4_DISABLE_FLG //used in slave mode to disable a range of registers
MB_RS4_DISABLE_REG //used in slave mode to define a range of registers
```

These registers can be written and read using the LOAD command, for example

```
LOAD(MB_RS4_MODE,0);          // set mode as Master
LOAD(status,MB_RS4_STATUS);   //read status code
```

Error Codes can be read from the MB_RS4_ERROR during a call to the DATAERR function.

00H = No Error
01H = Illegal Function – code received by the slave device is not recognized
02H = Illegal Data Address – the address does not exist in the slave
03H = Illegal Data Value – the value is not accepted by the slave
04H = Slave Device Failure – an unrecoverable error occurred during slave processing
05H = Acknowledge – the slave accepts the data but needs timeout extension
06H = Slave Busy – the master should try again later
07H = NACK – the slave cannot perform the command, master should request diagnostic

Status Codes can be read from MB_RS4_STATUS at any time and have the following meaning.

00H = Idle
01H = Transmitting
02H = Receiving
03H = Wrong function code in slave response
04H = Wrong slave ID in slave response
05H = Wrong CRC
06H = Response timeout
07H = Invalid request – slave has requested to read with global slave ID (0)

The process of iDevOS initialisation is shown in the example RS485 interface below:

```
setup(rs4)             //set up rs485 interface
{
  baud = 9600;         //set baud rate
  parity = N;          //set parity to none
  flow = N;            //set hardware flow control off
  encode = sd;         //set idevos encoding to byte data
  duplex = H;          //set RS485 driver chip to half duplex operation
  rxi = Y;             //enable receive interrupt
  txi = Y;             //enable transmit interrupt
  proto = modbusRTU;   //use Modbus protocol for communications
}
```

Create Registers for Modbus control and storage

```
var(stage, 0, U8);     //define a stage counter with default value of 0
var(regs, 0, U16, 256); //create a 256 register array with 16 bit data
```

Setup the built in Modbus variables

```
load(MB_RS4_SLAVEID, 1);        //set the slave id.
load(MB_RS4_TIMEOUT, 500);      // response timeout in ms
load(MB_RS4_MODE, 0);           // 0 = master, 1 = slave
load(MB_RS4_BASE, 0);           //define the base register address
load(MB_RS4_REGS > "regs");     //point to the register array ‘regs’
load(MB_RS4_DATAOK > "f_comms_ok");    // the function used for data OK
load(MB_RS4_DATAERR > "f_comms_err"); //the function used for data error
```

Set up the screen content using the PAGE() and associated display and key commands according to the application user interface requirements.
This example has been restricted to show the main entity names for clarity.
It seeks to control 2 relays by sending a toggle command when the button is pressed.
The status of the relay is read back and the button style stDrawXX colour set to ON or OFF.

```
page(relayPG, stPage //draw a box for each relay with a button associated.
{
posn(200,200); draw(d_relay1,…..,stDrawOff); //stDrawOff colour = Blue
key(k_relay1,f_toggle_relay1,…….TOUCH);      //stDrawON colour = Red
posn(600,200); draw(d_relay2,…..,stDrawOff);
key(k_relay2,f_toggle_relay2,…….TOUCH);
posn(400,400); text(t_error, “”  , stTextRed24);
}
```

Please refer to the many user interface examples on the web for more information on how to create, define and control screen page entities.

Each button (key) press action generates a call to a function which WRITES data to the Slave according to it's command set. The example toggles 2 relays connected to a Modbus slave controller which uses 0300 Hex for the toggle command.

```
func(f_toggle_relay1)
{
 load(stage, 0);         // set stage for response
 load(MB_RS4_FUNC, 6);   // write register
 load(MB_RS4_REGOFF, 1); // register offset in slave for first relay
 load(MB_RS4_REGCNT, 1); // register count
 load(regs.1, \\0300);   // load toggle command to first relay register
 load(RS4, 1);           //initiate send
}

func(f_toggle_relay2)
{
 load(stage, 1);         // set stage to use when response received
 load(MB_RS4_FUNC, 6);   // write register
 load(MB_RS4_REGOFF, 2); // register offset in slave for second relay
 load(MB_RS4_REGCNT, 1); // register count
 load(regs.2, \\0300);   //load toggle command to second relay register
 load(RS4, 1);           //initiate send
}
```

The Slave responds to the Master confirming if the data was received OK or an error occurred. If the data was OK, function f_comms_ok is called and the resultant sub-function is processed according to the current value of 'stage'. The Master then sends a READ command to confirm the status of the relay which was toggled (0=OFF or 1=ON).

```
func(f_comms_ok)  //run the sub function according to value of 'stage'.
{
if(stage=0?[run(f_comms_ok_0);  exit(f_comms_ok);]); //first relay response
if(stage=1?[run(f_comms_ok_1);  exit(f_comms_ok);]); //second relay response
if(stage=2?[run(f_comms_ok_2);  exit(f_comms_ok);]); //first relay update
if(stage=3?[run(f_comms_ok_3);  exit(f_comms_ok);]); //second relay update
}
```

Process stage 0 or 1 to request the relay ON or OFF state from the slave.

```
func(f_comms_ok_0)                 //first relay response
{
 load(stage, 2);                   // set stage for final relay value response
 load(MB_RS4_FUNC, 3);             // read register command
 load(MB_RS4_REGOFF, 1);           // register offset
 load(MB_RS4_REGCNT, 1);           // register count
 load(RS4, 1);                     // initiate send
}

func(f_comms_ok_1)                 //second relay response
{
  load(stage, 3);                  // set stage for final relay value response
  load(MB_RS4_FUNC, 3);            // read register command
  load(MB_RS4_REGOFF, 2);          // register offset
  load(MB_RS4_REGCNT, 1);          // register count
  load(RS4, 1);                    // initiate send
}
```

The READ data from the Slave is put into the ‘regs’ array. The following sub functions check the respective array register for 0 = OFF and the draw entity colour on the screen is then updated.

```
func(f_comms_ok_2)   //update the style of the relay 1 draw entity
{
 if(regs.1==0?[load(d_relay1.style, stDrawOff);]:[load(d_relay1.style,stDrawOn);]);;
}

func(f_comms_ok_3)  //update the style of the relay 1 draw entity
{
 if(regs.2==0?[load(d_relay2.style,stDrawOff);]:[load(d_relay2.style,stDrawOn);]);;
}
```

If communication timeout or a read / write error occurred, the function f_comms_err is called and the error text message on the screen displayed for 500ms.

```
func(f_comms_err)
{
  text(t_error, "ERROR");;             //show the word ERROR
  wait(500);                           //for 500ms
  text(t_error, "");;                  //clear the error message
}
```

<u>Read/Write Coil Example</u>
To simplify iDevOS operation, each coil is allocated a U8 array element

```
var(regs, 0, U8, 8); //create an 8 register array with 8 bit data

func(f_set_coils)
{
 load(MB_RS4_FUNC, 15);    // write coils command
 load(MB_RS4_COILOFF, 10); // coil offset in the slave is 10
 load(MB_RS4_COILCNT, 8);  // register count in regs array
 load(regs.1,\\00,\\FF,\\00,\\FF,\\00,\\FF,\\00,\\FF); //alternate coils ON
 load(RS4, 1);            //initiate send
}
```

The iDevOS firmware compacts the 8 register elements into a single byte and sends it to the slave. When reading coil data from the slave, the coil bits are distributed to the 8 array elements.

<u>Holding Registers Example</u>
A typical application may use a Slave with Holding Registers starting at register address 40001 and have several concurrent 16 bit registers containing alarm, temperature, humidity and counter information. This data is to be displayed on the display screen of the Master controller which requests updates every 250ms.

Setup the built in Modbus variables

```
load(MB_RS4_SLAVEID, 1);      //set the slave id.
load(MB_RS4_TIMEOUT, 150);    //response timeout in ms. Typical 70% of update
load(MB_RS4_MODE, 0);         // 0 = master, 1 = slave
load(MB_RS4_BASE, 40001);     //define the base register address
load(MB_RS4_REGS > "regs");   //point to the register array ‘regs’
load(MB_RS4_DATAOK > "f_comms_ok");      // the function used for data OK
load(MB_RS4_DATAERR > "f_comms_err");    //the function used for data error
```

Using a TIMER set to 250ms or a PAGE LOOP command, call a function (read_slave) to send a READ registers request to the Slave device

```
func(read_slave)
{
 load(stage,0);
 load(MB_RS4_FUNC, 3);   // read registers
 load(MB_RS4_REGOFF, 0); // register offset in slave for data from base 40001
 load(MB_RS4_REGCNT, 4); // register count of 4 for values in 40001, 2, 3 & 4
 load(RS4, 1);           //initiate send
}
```

Depending on the command sent , process the result in the receive function f_comms_ok

```
func(f_comms_ok)
{
if(stage=0 ? [run(f_read_data);  exit(f_comms_ok);]);   //read
if(stage=1 ? [run(f_write_data); exit(f_comms_ok);]);   //write
}
```

If reading data, update the entity values on the screen.

```
func(f_read_data)
{
text(t_alarm_data, regs.0);     //update alarm status
text(t_temp_data,regs.1);       //update temperature
text(t_humid_data,regs.2);      //update humidity
text(t_count_data,regs.3);;     //update counter and refresh display
}
```

The previous example shown is simplistic as the alarm codes may need converting to a message using pointers and the temperature and humidity values may need conversion from a voltage value plus a calibration offset using CALC commands prior to update on the screen.

Handling Floating Point (Decimal) Values

Two 16 bit registers (U16) can combine to form a U32 and the CALC command “CFLT” will convert this to a Floating Point value. The reverse process is possible when converting a Float back to two 16 bit registers.

```
CALC( u32, regs.1.4, 65536, regs.1.5, “*+”);
CALC( flt, u32, "CFLT" );  // Converts a float to IEEE 754 binary 32 formatted float


CALC( u32, flt, "CFLT" );  // Converts an IEEE 754 binary 32 formatfloat to a float
CALC( regs.1.4, u32, 65536, “/”);
CALC( regs.1.5, u32, 65535, “&”);  //logical AND to mask lower 16 bits
```

Using 2 Dimension Arrays

A 2 dimensional 16 bit array can be used to hold address in dim .0 and data in dim .1.
The example shows Master Write operation with absolute values for clarity.

```
VAR(regs, 0, U16, 2, 10);     //create a 2 dimension array with 10 addresses
LOAD(regs.0,4,5,6,7,9,1000,1001,30000,30004,63023);  //load default addresses
LOAD(regs.1.2, 102);          //load value 102 into data dim.1 at address 6
LOAD(regs.1.3, 29123);        //load value 29123 into data dim.1 at address 7
LOAD(MB_RS4_REGS > "array");  //point to array
LOAD(MB_RS4_REGOFF, 2);       //offset into mapping table
LOAD(MB_RS4_REGCNT, 2);       // number of registers in mapping table
LOAD(RS4, 1);                 //initiate send
```

This read example uses the same array as in the previous write example
Master Read (read regs 1000 and 1001)

```
LOAD(MB_RS4_REGOFF, 5); offset into mapping table
LOAD(MB_RS4_REGCNT, 2); number of registers in mapping table
LOAD(RS4, 1);
```

After read....

```
LOAD(var1, array.1.5);
LOAD(var2, array.1.6);
```

The array could be modified to `VAR( regs, 0, U16, 4, 10);` to hold Slave Id in dim.2, RegOffset in dim.3 and Reg Count set to 1. This allows further flexibility and automation.

```
LOAD(MB_RS4_SLAVEID, regs.2.num);
LOAD(MB_RS4_REGOFF, regs.3.num );
LOAD(MB_RS4_REGCNT, 1 );
```

If placed in a LOOP, the variable ‘num’ can be sequenced from 0 to 9 to scan all Slaves.

Slave Mode with 2 Dimensional Arrays

When using a 2 dim array in Slave mode the values in the address dimension gives all possible address values.

Slave Write (write regs 6 and 7 with 102 and 29123)

```
VAR(regs, 0, U16, 2, 10);
LOAD(array.0, 4, 5, 6, 7, 9, 1000, 1001, 30000, 30004, 63023);
LOAD(MB_RS4_REGS > "regs");
```

Within application runtime

```
LOAD(regs.1.2,102);         //update reg 6
LOAD(regs.1.3,29123);       //update reg 7
```

When the module receives a request from the master to read registers the registers are read immediately (asynchronous to the running of the iDev code). Therefore the iDev code needs to ensure the registers are updated frequently in a separate loop or using a timer depending on the application requirements.

Slave Read (read regs 1000 and 1001)

```
VAR(regs, 0, U16, 2, 10);
LOAD(regs.0, 4, 5, 6, 7, 9, 1000, 1001, 30000, 30004, 63023);
LOAD(MB_RS4_REGS > "regs");
```

Within application runtime

```
LOAD(varA, regs.1.5);
LOAD(varB, regs.1.5);
```

Advanced Register Restrictions when Operating in Slave Mode

It is possible to configure specific registers to be ‘active’ and accessible by the master.
The slave can also define ranges of allowable values which can be written by the master.

The **MB_RS4_VALID_DATA** built in variable is used to define valid data value ranges for registers and must point to a pointer array. This array points to any number of U16 array variables that each define a range of registers along with the valid values.

Each U16 must be structured to match one of the following layouts to define a range of registers + a range of values or a list of values as follows:

Range of Values

Byte 0	0
Byte 1	Reg start
Byte 2	Reg end
Byte 3	Value low
Byte 4	Value high

List of Values

Byte 0	1
Byte 1	Reg start
Byte 2	Reg end
Byte 3	Value 1
Byte 4	Value 2

Byte n	Value n

This is an example that allows values 100 – 200 in registers 10 – 15 and values 1,2 and 3 in register 1000.

```
// define variables
VAR(valid_data > ””, PTR, 2);
VAR(valid_data_A, 0, U16, 5);
VAR(valid_data_B, 0, U16, 6);

// populate data
LOAD(valid_data_A, 0, 10, 15, 100, 200);
LOAD(valid_data_B, 1, 1000, 1000, 1, 2, 3);

// assign pointers
LOAD(valid_data.0 > valid_data _A);
LOAD(valid_data.1 > valid_data _B);
LOAD(MB_RS4_VALID_DATA > valid_data);
```

If a write request is received with an out of range value then the ILLEGAL DATA VALUE (0x03) Modbus code is returned.

If a write request is received with an out of range register then the ILLEGAL DATA ADDRESS (0x02) Modbus code is returned.

The **MB_RS4_VALID_REG** variable is used to define valid ranges of registers and must point to a pointer array. This in turn points an any number of U16 array variables (with 2 elements) to define valid register ranges.

This example defines 2 ranges of valid registers (10 – 90 and 120 – 190) :-

```
// define variables
VAR(valid_regs > ””, PTR, 2);
VAR(valid_regs_A, 0, U16, 2);
VAR(valid_regs_B, 0, U16, 2);

// load register ranges
LOAD(valid_regs_A, 10, 90);
LOAD(valid_regs_B, 120, 190);

// assign pointers
LOAD(valid_regs.0 > valid_regs_A);
LOAD(valid_regs.1 > valid_regs_B);
LOAD(MB_RS4_VALID_REG > valid_regs);
```

The **MB_RS4_DISABLE_FLG** and **MB_RS4_DISABLE_REG** pair of variables are used to define if registers are accessible by the slave and can be enabled and disabled at run time. These variables must point to a pointer array.

The **MB_RS4_DISABLE_FLG** array points to any number of U16 array variables to define the access enabled / disabled flag values.

The **MB_RS4_DISABLE_REG** array points to any number of U16 array variables with 2 elements to define the register range associated with the flags.

This example uses 3 ranges of registers that need to be enabled or disabled in the application.

```
// define variables
VAR(disable_flags > ””, PTR, 3);
VAR(disable_regs > ””, PTR, 3);

VAR(disable_flags_A, 0, U16);
VAR(disable_flags_B, 0, U16);
VAR(disable_flags_C, 0, U16);

VAR(disable_regs_A, 0, U16, 2);
VAR(disable_regs_B, 0, U16, 2);
VAR(disable_regs_C, 0, U16, 2);
```

```
// load register ranges
LOAD(disable_regs_A.0, 10);
LOAD(disable_regs_A.1, 20);

LOAD(disable_regs_B.0, 109);
LOAD(disable_regs_B.1, 130);

LOAD(disable_regs_C.0, 191);
LOAD(disable_regs_C.1, 199);

// assign pointers
LOAD(disable_flags.0 > disable_flags_A);
LOAD(disable_flags.1 > disable_flags_B);
LOAD(disable_flags.2 > disable_flags_C);

LOAD(disable_regs.0 > disable_regs_A);
LOAD(disable_regs.1 > disable_regs_B);
LOAD(disable_regs.2 > disable_regs_C);

LOAD(MB_RS4_DISABLE_FLG > disable_flags);
LOAD(MB_RS4_DISABLE_REG > disable_regs);
```

When access needs to be disabled for one of the banks, for example regs 10 – 20 use LOAD(disable_flags_A, 1); then enable again with LOAD(disable_flags_A, 1);

If a write register access (codes 06h / 10h) is received from the slave to one of the registers defined (10 – 20, 109 – 130, 191 – 199) and the corresponding flag = 1 then a NACK (0x07) Modbus response is returned to the slave.