Modbus in MAC400/800

The modbus implementation in MAC400/800 is a subset of the Modbus Specification V1.1b. This standard can be downloaded free of charge from the website www.modbus.org or http://modbus-ida.org.

Also you may want to download and read the Modbus Serial Line Protocol and Implementation Guide V1.02**,** that describes many aspects of the signals, and the details of using and inter-connecting RS-422, two-wire RS-485 and four-wire RS-485.

The serial communications lines normally used for communications between the basic motor and one of the intelligent MAC00-XX modules can be configured to use the Modbus protocol instead of the standard FastMac protocol.

The MAC400/800 firmware supports the two command types Read Holding Registers (3) and Write Multiple Register (0x10). All other commands will result in Exception replies (exception type 1, Illegal Function).
Use firmware v1.31 or later for MAC800 and firmware v1.03 or later for MAC400.

All registers can be read as well as written over Modbus, but the number of registers per transfer is limited to 16 16-bit registers or 8 32-bit registers. Contact JVL if more registers are needed in a single transfer.

All registers in the MAC400/800 motors are 32-bits. To comply with the clean 16-bit Modbus standard, a 32-bit register must be read or written as two consecutive 16-bit registers.
The register address mapping follows the normal documented register numbers but the address field, but must be multiplied by two, so to read or write register 3, P_SOLL, use the address 6.

The setup of the Modbus protocol is part of the general setup of the UART (serial port chip).
The motor default uses the FastMac protocol at 19200 baud, No parity, 8 data bits, 1 stop bit.
All fields in bits 8 to 31 depend on the chosen protocol, and currently are used only with Modbus.

Register 213, UART1_SETUP, supports the following bit-fields (default values marked with *):

Bits	Values	Description
3:0	0=9600, 1=19200, * 2=38400, 3=57600, 4=115200, 5=230400, 6=444444, 7=1000000 baud	Basic baud rate in bits per second. Note that values 6 and 7 are non-standard baud rates, that are intended to be used between two or more MAC400/800 motors only.
7:4	0=FastMac, * 1=Modbus (motoraddress), 2=Modbus (address 254)	Protocol to use – select 1 for Modbus. The option to use Modbus address 254 (instead of the motor address) is intended for use with the JVL MAC00-xx interface modules.
9:8	0=5,	Number of data bits in a byte. Modbus always uses 8 bits

	1=6, 2=7, 3=8 data bits	per byte.
10	Must be zero	
13:11	0=Even, 1=Odd, 2=Space, 3=Mark, 4 or 5=None 6 or 7=Multidrop	Parity scheme. Modbus should use either Even or Odd parity for maximum error checking. Multidrop parity is not supported by Modbus, but a non-standard multi-drop operation is supported, see bits 20 and 21.
15:14	0=1, 1=1.5, 2=2 stop bits	Number of stop bits to use.
19:16	0..15	Guard-time. Number of idle bit times between bytes during transmission. These can be seen as additional stop bits. Normally this value is set to zero, but with some UARTs that have trouble synchronizing on long telegrams, this value can be set to non-zero. Setting this value non-zero may help visually separating bytes on an oscilloscope.
20	0=Multi-drop 1=Point-to-point	Selects if the motor should drive differential transmitter line-pair A+ and A- at all times (=1) or only when it is actually transmitting bytes (=0). When only one (bus slave) motor is on the line, like regular RS422 4-wire operation, use the value of 1 for point-to-point. This can save pull-up and pull-down resistors on some systems. When more than one slave motor is on the line, in RS485 operation, use the value of 0 for multi-drop mode, so only one motor will drive the line at any time, and release the line up to 50 us after the last byte has been transmitted. You **must** use multi-drop mode if more than one slave motor is on the line – otherwise the electronics will be overloaded, and may fail permanently. **This will not be covered by the guarantee**.
21	0=Half duplex 1=Full duplex	Modbus is generally specified as a half-duplex protocol, so there will only be traffic in one direction between client and a server at a time. For very advanced operation, it is possible to use full-duplex operation to optimize communication speed, but at the cost of more difficult error checking. Full duplex can be used **only** with strict RS422 operation and four-wire RS485. For two-wire RS485 operation, use half duplex.
23:22	0..3	Reserved for future use
25:24	0=Passive server, 1=Active server with timeout monitoring.	For normal operation where a PC or PLC talks to one or more motors, set this to zero. If set to One, the motor will set a Communications Error

	2=Client (bus master) operation to transfer requested position and monitor errors.	and stop if it has not received a valid write request to P_SOLL within the timeout selected in bits 31:28. If set to two, the motor will write a position and read the error/status register of the client at address 254 once per sample period. If not both write acknowledge and error/status data is received within the timeout specified in bits 31:28, the motor will set a Communications error and stop.
27:26	Reserved	
31:28	Timeout in milliseconds.	

The firmware will use only the power-up value of register 213, so for any changes to take effect, do a Save in Flash operation.

Read Holding operation:

Request: <adr>, 0x03, RegHi, RegLo, CountHi, CountLo, CRC1, CRC2
Offset: [0] [1] [2] [3] [4] [5] [6] [7]
Reply: <adr>, 0x03, #Bytes, Reg0Hi, Reg0Lo, Reg1Hi, Reg1Lo, ... CRC1, CRC2

Example to read P_IST from motor with address 1, values in decimal:
1, 3, 0, 20, 0, 2, NN, MM (NN and MM are the CRC-16 bytes)

Write Multiple Register operation:

Request: <adr>, 0x10, RegHi, RegLo, CountHi, CountLo, NBytes, Val0Hi, Val0Lo, ..., CRC1, CRC2
Offset: [0] [1] [2] [3] [4] [5] [6] [7] [8]
Reply: <adr>, 0x10, RegHi, RegLo, CountHi, CountLo, CRC1, CRC2

Example to write P_SOLL to motor with address 1, values in decimal:
1, 16, 0, 6, 0, 2, 4, bb, aa, dd, cc, NN, MM (NN and MM are the CRC-16 bytes)

This would write a 32-bit hexadecimal value of ddccbbaa – note the byte-packing.