Overview

Modbus is an industry standard protocol that allows a variety of automation devices (such as Programmable Logic Controllers and Human Machine Interfaces) to communicate with each other. The Modbus protocol defines a simple protocol data unit (PDU) that is independent of the underlying communication layers.

Controllers communicate (via RS232/422/485) using a master–slave technique, in which only one device (the master) can initiate transactions (called ‘queries’). The other devices (the slaves) respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers.

The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (called a ‘response’) to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master.

The Modbus protocol establishes the format for the master’s query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error–checking field. The slave’s response message is also constructed using the Modbus protocol. It contains fields confirming the action taken, any data to be returned, and an error–checking field. If an error occurred in receipt of the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response.

A Modbus interpreter has been written in the Mint language. This not only demonstrates the power and flexibility of the Mint programming language, but also allows Modbus functions to be tailored to your application. There are two variants to Modbus, RTU and ASCII. This Mint-based interpreter offers support for the RTU variant with the Mint controller operating as a Slave device (refer to AN00130 for details of the Mint based Modbus ASCII interpreter).

Two sample Mint programs are included. One is written for use with the listed ‘Supported Controllers’ and assumes the data is restricted to 16 bit integer values. The second sample is aimed at NextMove controllers running MintMT and e100 drives running Mint Lite and provides support for both integer and floating point data. To illustrate this second example program further, two Allen Bradley PLC programs (suitable for a MicroLogix 1500 PLC) have also been provided.

The sample Mint programs allow a Modbus master to read/write the Mint NVLONG array when using integer data or NVFLOAT when using floating point data, to read digital inputs from the Mint controller and to write to digital outputs on the Mint controller. The interpreter can easily be updated to utilize COMMS locations instead of NVLONG registers (comments are included in the doDecode routine to indicate required changes to logic and simply replace NVLONG with COMMS).

For more detailed information about the Modbus RTU protocol visit [www.modbus.org](http://www.modbus.org)
Modbus RTU

Data Format

When controllers are setup to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, each 8-bit byte in a message contains two 4-bit hexadecimal characters. Each message must be transmitted in a continuous stream.

The Mint interpreter offers support for master devices with communication settings of:

- 8 data bits, 1 stop bit, No Parity

Baud rates are limited to those supported by the Mint controller (and of course must match the baud rate setting made on the Modbus master device).

The Mint Startup block defines the controller’s serial port setting (example program Modbus RTU Slave 2.x.mnt illustrates a setting of 57600 baud on terminal channel 1 – e.g. the serial port on a Nextmove ESB controller)... 

\[
\text{If SERIALBAUD(_TERM1) <> 57600 Then SERIALBAUD(_TERM1) = 57600}
\]

Message Framing

In RTU mode, messages start with a silent interval of at least 3.5 character times. This is most easily implemented as a multiple of character times at the baud rate that is being used on the network.

The first field then transmitted is the device address (in this case the node address of the Mint controller). Networked devices monitor the network bus continuously, including during the ‘silent’ intervals. When the first field (the address field) is received, each device decodes it to find out if it is the addressed device.

Following the last transmitted character, a similar interval of at least 3.5 character times marks the end of the message. A new message can begin after this interval. The entire message frame must be transmitted as a continuous stream. If a silent interval of more than 1.5 character times occurs before completion of the frame, the receiving device flushes the incomplete message and assumes that the next byte will be the address field of a new message.

A typical message frame is shown below.

<table>
<thead>
<tr>
<th>START</th>
<th>ADDRESS</th>
<th>FUNCTION</th>
<th>DATA</th>
<th>CRC CHECK</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>8 bits</td>
<td>n * 8 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{delay}</td>
<td>[STX]</td>
<td>[ETX]</td>
<td>[NUL][SOH][NUL][SOH]</td>
<td>Ŧ Ŧ</td>
<td>{delay}</td>
</tr>
</tbody>
</table>
**Address Field**

The address field of a RTU message frame contains one character (8 bits). Valid slave device addresses are in the range of 0 – 255 decimal.

When the slave sends its response, it places its own address in the address field of the response to let the master know which slave is responding.

Address 0 is used for the broadcast address, which all slave devices recognize. When configuring a Mint controller for use as a Modbus Slave it is therefore advisable to avoid setting this controller up as Mint Node 0 (note that the master can only use the broadcast address for write transactions).

**Function Field**

The function code field of a RTU message frame contains one character (8 bits). Valid codes (from the master) are in the range of 1 – 127 decimal.

When a message is sent from a master to a slave device the function code field tells the slave what kind of action to perform. Examples are to read the ON/OFF states of a group of discrete inputs; to read the data contents of a group of registers; to read the diagnostic status of the slave or to write to designated registers.

When the slave responds to the master, 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 slave simply echoes the original function code. For an exception response, the slave returns a code that is equivalent to the original function code with its most–significant bit set to a logic 1.

The Mint Modbus RTU interpreter supports a subset of the full Modbus function set:-

**01 (decimal) – Read Outputs**

This function reads the ON/OFF status of discrete digital outputs on the slave. Broadcast is not supported (i.e. the function request will be ignored).

The query message from the master specifies the starting input and quantity of outputs to be read from the Mint controller. Outputs are addressed starting at zero (as they are in Mint).

The output status in the response message is packed as one output per bit of the data field. Status is indicated as: 1 = ON; 0 = OFF. The LSB of the first data byte contains the output addressed in the query. The other outputs 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.

**02 (decimal) – Read Inputs**

This function reads the ON/OFF status of discrete digital inputs in the slave. Broadcast is not supported (i.e. the function request will be ignored).
The query message from the master specifies the starting input and quantity of inputs to be read from the Mint controller. Inputs are addressed starting at zero (as they are in Mint).

The input status in the response message is 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.

03 (decimal) – Read Holding Registers

This function reads the binary contents of holding registers (Mint NVLONG locations in this case) in the slave. Broadcast is not supported.

The query message specifies the starting register and quantity of registers to be read. Registers are addressed starting at zero. However Mint Comms locations are addressed from 1 – 99 (see also Application Note AN00110) so care should be taken not to try and read from register 0 on the Mint controller if using Comms instead of NVLONG.

The data is processed as 16–bit values (range +/- 32767) when using integer data or pairs of Modbus registers are used to read a 32-bit floating point value.

05 (decimal) – Force Single Coil

This function forces a single coil (Mint controller digital output) to either ON or OFF. When broadcast by the master (i.e. sent to Node 0), the function forces the same digital output in all attached slaves.

The query message specifies the digital output to be forced. Outputs are addressed starting at zero (as they are in Mint).

The requested ON/OFF state is specified by a constant in the query data field:

- A value of FF00 hex requests the coil to be ON
- A value of 0000 requests it to be OFF

All other values are illegal and will not affect the coil.

The normal response is an echo of the query, returned after the output has been forced.

06 (decimal) – Preset Single Register

This function presets (writes) a value into a single holding register (Mint NVLONG location in this case). When broadcast, the function presets the same register reference in all attached slaves.

The query message specifies the register reference to be preset. Registers are addressed starting at zero. However Mint Comms locations are addressed from 1 – 99 (see also Application Note AN00110) so care should be taken not to try and write to register 0 on the Mint controller if using Comms instead of NVLONG.

The data is processed as 16–bit values (range +/- 32767) when using integer data or pairs of Modbus registers are used to write a 32-bit floating point value.
16 (decimal) – Preset Multiple Registers

This function presets (writes) values into a sequence of holding registers (Mint NVLONG locations in this case). When broadcast, the function presets the same register references in all attached slaves.

The query message specifies the register references to be preset. Registers are addressed starting at zero. However Mint Comms locations are addressed from 1 – 99 (see also Application Note AN00110) so care should be taken not to try and write to register 0 on the Mint controller if using Comms instead of NVLONG.

The data is processed as 16-bit values (range +/- 32767) when using integer data or pairs of Modbus registers are used to write 32-bit floating point values.

23 (decimal) – Read/Write Multiple Registers

This function presets (writes) values into a sequence of holding registers (Mint NVLONG locations in this case) and read back a sequence of holding registers (Mint NVLONG locations in this case) all in a single transaction (this can be useful where a PLC needs to confirm that the written data has been received/processed). Broadcast is not supported.

The query message specifies the register references to be preset and the register references to be read. Registers are addressed starting at zero. However Mint Comms locations are addressed from 1 – 99 (see also Application Note AN00110) so care should be taken not to try to read or write register 0 on the Mint controller if using Comms instead of NVLONG.

The data is processed as 16-bit values (range +/- 32767). This function is not supported if using floating point data.

Data Field

The data field is constructed using sets of two hexadecimal digits, in the range of 00 to FF hexadecimal that are then packed into a single ASCII character. The data field of messages sent from a master to slave devices contains additional information which the slave must use 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.

For example, if the master requests a slave to read a group of holding registers (function code 03), the data field specifies the starting register and how many registers are to be read. If the master writes to a group of registers in the slave (function code 16 decimal), the data field specifies the starting register, how many registers to write, the count of data bytes to follow in the data field, and the data to be written into the registers.

If no error occurs, the data field of a response from a slave to a master contains the data requested. If an error occurs, the field contains an exception code that the master application can use to determine the next action to be taken.

Error Checking Field

When RTU mode is used for character framing, the error checking field contains two ASCII characters. The error check characters are the result of a Cyclic Redundancy Check (CRC) calculation that is performed on the message contents.
The CRC characters are appended to the message as the last field.

**Exception Responses**

Except for broadcast messages, when a master device sends a query to a slave device it expects a normal response. One of four possible events can occur from the master’s query:

- If the slave device receives the query without a communication error, and can handle the query normally, it returns a normal response.
- If the slave does not receive the query due to a communication error, no response is returned. The master program will eventually process a timeout condition for the query.
- If the slave receives the query, but detects a communication error (incorrect CRC), no response is returned. The master program will eventually process a timeout condition for the query.
- If the slave receives the query without a communication error, but cannot handle it (for example, if the request is to read a non-existent register), the slave will return an exception response informing the master of the nature of the error.

The exception response message has two fields that differentiate it from a normal response:

**Function Code Field:** In a normal response, the slave echoes the function code of the original query in the function code field of the response. All function codes have a most-significant bit (MSB) of 0 (their values are all below 80 hexadecimal).

In an exception response, the slave sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.

With the function code’s MSB set, the master’s application program can recognize the exception response and can examine the data field for the exception code.

**Data Field:** In a normal response, the slave may return data or statistics in the data field (any information that was requested in the query). In an exception response, the slave returns an exception code in the data field. This defines the slave condition that caused the exception.

Only three exception responses are provided by the Mint interpreter program...

```vbnet
Const _nIllegalFunction As Integer = 1
Const _nIllegalAddress As Integer = 2
Const _nIllegalValue As Integer = 3
```

An illegal function response is generated if the master requests a function other than those supported by the interpreter.

An illegal address response is generated if the master tries to access a NVLONG register outside the valid range (or a Comms location outside of the range 1 – 99 if Comms locations are used), tries to read an input outside of the controller’s input range for Bank 0 or tries to write to an output outside of the controller’s output range for Bank 0. The illegal address response validation...
can be updated if the user wishes to access Comms locations instead of NVLONG registers (see the comments in subroutine doDecode).

An illegal value response is generated if the master attempts to write an illegal value to a digital output (i.e. something other than 0xff00 or 0x0000).

**Debugging**

The variable bDebug has been defined to allow some simple debugging. If this is set to _TRUE, various messages will be displayed via terminal channel 2 (e.g. the USB connection on a NextMove ESB) during the execution of the interpreter. For Mint controllers that only support a single terminal channel (e.g. MintDrive II which only has a single serial interface) the example program should be edited to remove references to terminal channel 2 (#2).

**Controller Specifics**

The sample Mint code that accompanies this Application Note is specifically targeted at non Ethernet Powerlink controllers. Please refer to comments within the code for minor keyword differences when using an e100 controller (e.g. BUSNODE(6) replaces NODE).

If using a MintDriveII with version 2.x of the code then to disable the native serial protocols for Host Comms Protocol (HCP) and Baldor Binary Protocol (BBP) the commands....

```
COMMSMODE = _cmBBP_DISABLE
SERIALPROTOCOL(_TERM1) = _spNONE
```

...must be used instead of the single COMMSMODE = _cmHCP_DISABLE + _cmBBP_DISABLE.

This is detailed in the Startup block of the sample program.

**Also note that pins 7 and 8 must be linked together at the MintDriveII end of the serial cable if using RS232.**

Sample Mint code ‘AB Modbus RTU Slave x.x.mnt’ is targeted at NextMove motion controllers running MintMT and e100 drives running Mint Lite and is likely to be used if interfacing these controllers to a PLC (e.g. an Allen Bradley MicroLogix 1500 LRP Series C which has support for Modbus RTU master operation as standard). This code includes support for transfer of 32 bit floating point data as well as 16 bit integer data. The baud rate is set to 38400 by default in this program (to match the maximum rate of a MicroLogix PLC).

By default the program is coded to utilize 32 bit floating point data for all Modbus functions via this line of Mint code (in the ModbusRTUSlave task)...

```
Const _bFloat As Integer    = _true
```

This would allow, for example, an Allen Bradley PLC to read/write data between a floating point data area (e.g. F8) and NVFLOAT on the Baldor controller.

Change this line to read...

```
Const _bFloat As Integer    = _false
```
...should you wish to use 16 bit integer data (e.g. to transfer data between an Allen Bradley integer data area such as N7 and NVLONG on the Baldor controller).

Support for Modbus function 23 (Read/Write registers) is not included in this second example program as it is not typically provided by PLCs capable of transferring floating point data (function 23 only supports 16 bit integer values).

**Sample PLC Programs**

Two Allen Bradley sample programs suitable for use with a MicroLogix 1500 LRP Series C PLC (one for transfer of integer data between the N7 data area on the PLC and NVLONG on the Baldor controller, and one for transfer of floating point data between the F8 data area on the PLC and NVFLOAT on the Baldor controller) are included with this application note.

Note that the MicroLogix 1500 must be Series C or later in order to support operation as a Modbus RTU master device.

Custom Data Monitors (CDMs) are provided with each to allow the user to test the operation. The target node and data address (i.e. the Baldor NVFLOAT/NVLONG address) can be modified via the CDM.

**Timing**

Operating the example program ‘Modbus RTU Slave x.x.mnt’ at 57600 baud with only the ModbusRTUSlave task running on a Nextmove ESB-2, timing for functions 01, 02, 03, 05, 06, 16 and 23 (decimal) was found to be as follows:

**01 – Read Outputs**
- 1 Output : 18ms
- 16 Outputs : 20ms

**02 – Read Inputs**
- 1 Input : 18ms
- 16 Inputs : 20ms

**03 – Read Register**
- 1 location : 20ms
- 99 locations : 295ms

**05 – Write Output**
- 1 Output : 18ms

**06 – Write Register**
- 1 location : 10ms

**16 – Write Register**
- 1 location : 34ms
- 99 locations : 285ms

**23 – Read/Write Registers**
- 1 location : 73ms
This information may be useful if configuration of timeout settings is possible on the master device. It should be noted that these timings will increase as other Mint tasks/events are included in the application.

Note that when using RS422 it is necessary to include a delay in the Mint program before responding to allow time for the transmitter on the controller to be enabled (see comments in the subroutine ‘doFinishMsg’. This will affect the timings listed above accordingly.