

AN-218 Implementing a RS-485 Multi-drop Network

Introduction

This application note describes how to interface ZX devices with a RS-485 multi-drop network. RS-485 networks are used by many industrial applications including process control, building automation and even the control of lights at a concert. A RS-485 header board and example hookup is provided to explain the simple hardware required to get started with networking ZX devices. The software included with this application note gives a sample application and a reusable set of APIs to send and receive messages on a RS-485 network. Here is a table of contents to help with document navigation.

Overview of RS-485.....	1
Hardware Interfacing.....	2
RS-485 Header Board.....	2
Example RS-485 Network.....	3
Software.....	4
Software Requirements and Protocol Stack.....	4
Serial Interface.....	5
RS485Init.....	5
RS485Term.....	5
RS485Start.....	6
RS485Stop.....	6
RS485Send.....	6
RS485FinishReceive.....	7
RS485Receive.....	8
Sample Application.....	10
Sample Output.....	10
Address Resolution.....	11
Implementation of Master.....	12
Implementation of each Slave.....	13
Development Considerations.....	14
Debugging Network Applications.....	14
Incremental Design and Coding.....	15
Network Performance.....	16
Summary.....	16

Overview of RS-485

RS-485 differs from the more familiar RS-232 standard in a number of ways. Below is summary table of the electrical differences.

	RS-232	RS-485
Number of Devices	2 (point to point)	Up to 32 (multi-drop)
Mode of Operation	Single-ended relative to common ground	Differential voltage on two wires A and B
Signaling	+5 to +15V is a digital 0 -5 to -15V is a digital 1	B > A is a digital 0 A > B is a digital 1
Simultaneous send and receive	Full duplex via TX and RX wires	Half duplex but full duplex is possible using two pairs of A and B wires
Maximum Cable Length	50 feet	4000 feet
Official Standard Name	EIA-232	EIA-485

In a RS-485 network if two devices try to talk at the same time, the signals will interfere with each other – this is termed bus contention. A similar network that also suffers from bus contention is Ethernet. Ethernet solves the

AN-218 Implementing a RS-485 Multi-drop Network

problem by first detecting the bus contention and then using a “back-off” algorithm to try again later. RS-485 has a much simpler approach to this problem by designating a master and the master orchestrates all network traffic. The other devices are termed slaves and slaves only talk in response to a request from the master. This ensures that only one device at a time is driving the network. There are algorithms for one device to “take over” mastery of the RS-485 network but this is outside the scope of this application note.

A balanced-line driver for RS-485 network has an enable signal. When disabled the line driver is not connected to the network and the A and B lines are tri-stated. There is also a balanced line receiver that also has an enable signal. With proper coordination between the various drops using the master-slave idea, it is possible for only one device to be driving the network at any one time and all the other devices are listening.

It is good practice to also include termination at each end of the transmission line especially for either high data rates or long wiring runs. In this case high transmission rates means > 19200 baud and long wiring runs means tens or even hundreds of meters. The common signal ground is also recommended to keep the signaling voltages on the A and B lines within tolerance.

When all the drops are in listen mode, there isn't an active driver and therefore the A and B lines tend to float. To avoid this condition it is common practice to add “bias” resistors to tie the A line to ground (pull-down) and the B line to positive voltage (pull-up).

Additional hardware considerations for surge suppression, isolation, transient protection and faults are outside the scope of this application note. A good resource more details is the “RS-422 and RS-485 Application Note” from B&B Electronics (<http://www.bb-elec.com/bb-elec/literature/tech/485appnote.pdf>).

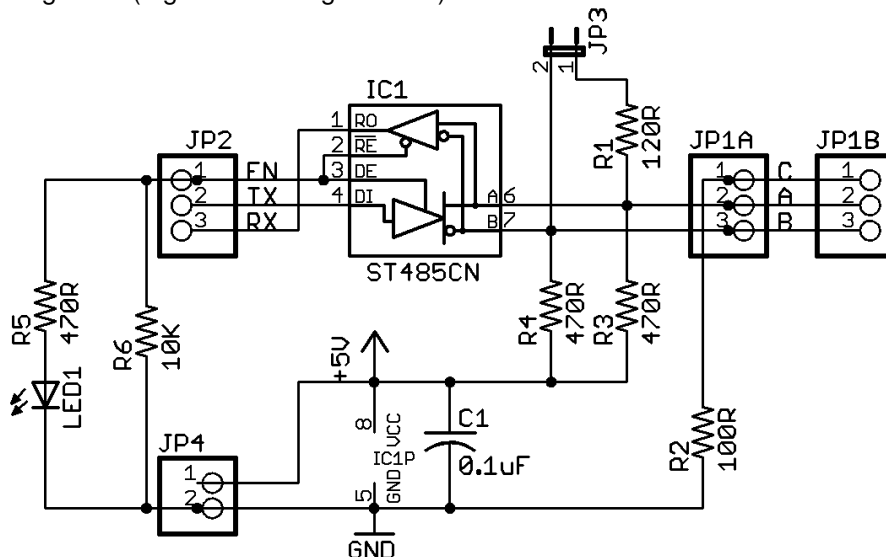
Hardware Interfacing

Many chip manufacturers make line driver/receiver (transceiver) chips for RS-485 networks. The author has successfully used the ST Microelectronics ST485. There are up to four connections required to a microcontroller that include the driver logic input, receiver logic output, enable line for the driver and enable line for the receiver. Because the two enable lines have opposite polarity, it is common practice to tie the two enable lines together so that one signal can control either driver enabled (logic 1) or receiver enabled (logic 0).

RS-485 Header Board

Below is an example RS-485 circuit that can either be used as a stand-alone “header” board or can be added to an existing microcontroller system. The schematic in EAGLE format is included in the associated ZIP file. The author can be contacted for the RS485 EAGLE part.

JP1A and JP1B are the connections to the RS-485 network. The author used 0.1” headers to aid with “daisy-chaining” of headers together but it is also common practice to use screw terminals. The RS-485 common ground (also called “C”) is connected into the local ground through R2 to limit ground currents. In some cases there may already be a common ground (e.g. same voltage source) and this line is not needed.



AN-218 Implementing a RS-485 Multi-drop Network

JP2 is the connection to the ZX microcontroller via 3 I/O pins one of which controls the enable/disable of the driver. A LED is used to indicate whenever the RS-485 line driver is enabled. The TX/RX signals should be connected to any pair of I/O pins that can provide logic level serial data. This could be a ZBasic logical serial port (COM3-6) or an USART port such as COM1.

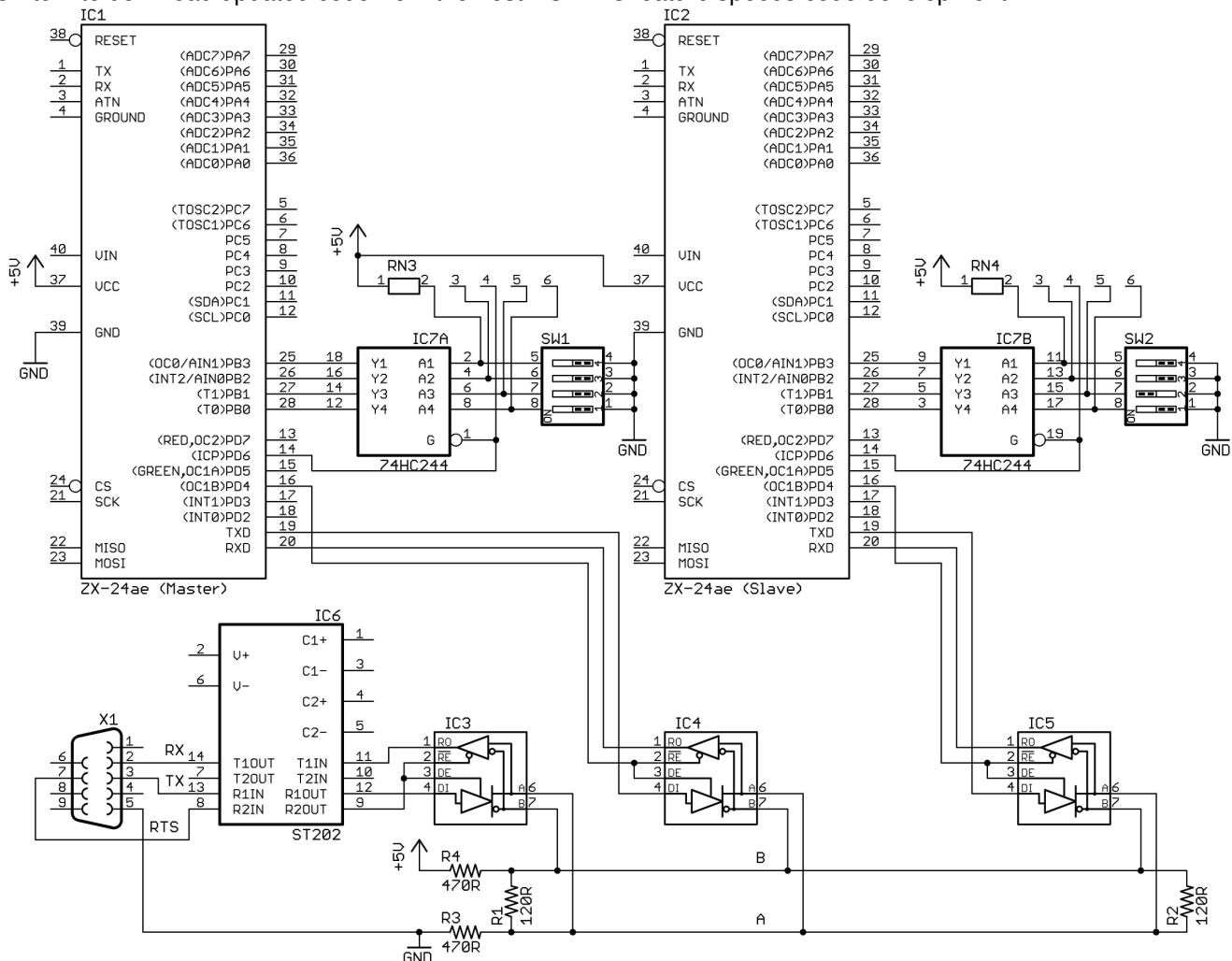
Jumper JP3 and its associated resistor R1 provide optional line termination. The optional resistor R6 is used to ensure that the default mode for the RS-485 transceiver is receiving. The two resistors R3 and R4 are the biasing resistors for the network and only need to be included on one of the header circuit boards. Because the Master is always needed for any RS-485 network, it is convenient to include the bias resistors at this drop. The header JP4 is used to provide power to the header board and the de-coupling capacitor C1 is optional.

Example RS-485 Network

The schematic below shows my test setup for a RS-485 network. It is not a complete circuit but shows the important details. Other variations are discussed later in this section.

There are 3 RS-485 transceivers (IC3-5) connected together into a RS-485 network. Two are connected to the COM1 ports of ZX-24ae devices (IC1-2) and the third to a standard MAX232 RS-232 dual transceiver chip (IC6). The RS-232 serial line is connected to the host computer and can be used to monitor the RS-485 network. In other applications it can be the master for the RS-485 network and the RTS line is often used to control the RS-485 driver enable. Line termination (R1-2) and bias resistors (R3-4) are also shown.

Of the two ZX-24ae devices, IC1 is configured to be the master and IC2 to be the slave. The switch on the ZX-24ae devices needs to be in the "logic level" position so that serial data is presented to the RS-485 transceiver via pins 19 and 20 of each ZX-24ae device. Pin D.4 is used as the RS-485 driver enable line. This example circuit uses ZX-24ae devices because they allow both high-speed serial transfers via COM1 and the ability to quickly "flip the switch" to download updated code from the host PC. This feature speeds code development.



AN-218 Implementing a RS-485 Multi-drop Network

IC7 is a non-inverting tri-state buffer with a active low enable (e.g. 74HC244) that is used to connect a pair of 4-way DIP switches to each of the ZX-24ae devices. RN1 and RN2 are 10K resistor arrays which are used to pull up the lines to 5V. The enable for the buffer is connected to pin D.6 of the ZX-24ae devices and pins B.0 to B.3 are used to set the address of the device. The device address is the inverse of the value of these 4 bits. In the schematic below the master (IC1) is address 0 and the slave (IC2) is address 2.

The high-speed logical COM1 port is also available on other ZX devices such as the ZX-44, ZX-44a, ZX-40, ZX-40a, and ZX-24e. It is possible to use the COM1 port on a ZX-24 or ZX-24a device with the addition of RS-232 to RS-485 protocol converter or a RS232 transceiver. If network speed is less important then the ZBasic logical COM ports (COM3-6) can be used instead which is supported on all ZX devices.

The tri-state buffers are used to read in the state of the DIP switches at application startup and are not used again. Providing that the switch values do not affect anything else, these I/O pins can be reused in the application. These buffers could be removed entirely and the switch state read directly from pins B.0 to B.3. There are numerous variations here on how the address of the device can be given to it including reading it from EEPROM.

Software

A key part of this application note is the associated ZBasic software, which is split into three modules. The main module of interest is **RS485serial.bas** which implements a reusable API for sending and receiving messages over a RS-485 multi-drop network. **AN218.bas** contains a sample application that exercises the API and **DebugPrint.bas** implements a debugging interface over a RS-232 serial line that does not have to be COM1.

The software in the associated ZIP file requires ZBasic version 2.0.0 or greater and a VM version 2.0.0 or greater. The source code is distributed with a BSD license (<http://www.opensource.org/licenses/bsd-license.php>). Please contact the author if you need a commercial license for this code.

Software Requirements and Protocol Stack

An overview of hardware and electrical connections for a RS-485 network is described on page 1 and is termed the *physical layer* in the 7-layer ISO networking model (http://en.wikipedia.org/wiki/OSI_model). The first layer of software on top is the *datalink layer* and describes how data bytes are transmitted on the network. From the hardware description above it should be obvious that serial data is transmitted on a RS-485 network in a similar way to a RS-232 connection using a start bit, 7 or 8 data bits, an optional parity bit, and one or two stop bits.

In the next protocol layer up, which is also part of the *datalink layer*, the format of data packets is defined. There are four parts to this packet definition:

- Packet Size The maximum size of any data packet is 256 bytes.
- Packet Header The packet header is a single byte with a value from 1 to 255 that identifies the slave address. An address of 0 is a broadcast message to all slaves. A slave response contains the address of the slave.
- Packet Content The packet content (or message) is up to 253 bytes long and the format is defined by the application.
- Packet Tail The packet tail is 2 bytes long consisting of a 16-bit CRC value of the packet header and packet data. The CRC algorithm used is the same as the one used for Modbus.

Once a slave receives a data packet, it verifies the CRC and checks the packet header to determine if it should process the message. A slave should process packets addressed to it and send a response back to the master. The master in the meantime should wait for a response and timeout if no response is received within a certain time limit. A lack of response from any message except a broadcast message means that the slave is not connected to the network. A slave should still respond if it is busy processing a previous message and cannot respond to the new one. The situation with a bad CRC is more difficult because the packet header could be in doubt. For the sample application code, slaves assume that the packet header is correct and only send a response to the master when the CRC is correct.

The *application layer* defines the content of data packets and how responses and errors are sent back to the master. For the sample application, the message content and response are very simple. In a future application note, an implementation of a Modbus (<http://www.modbus.org>) slave will be presented. The [Modbus specification](#) defines a structure for packets of data and how commands, values and errors are formatted in data packets. The

AN-218 Implementing a RS-485 Multi-drop Network

packet definition in this application note is based on the [Modbus RDU serial protocol](#). This was done because it is both a reasonable packet definition and is needed for a Modbus slave implementation.

Serial Interface

The RS-485 serial interface in **RS485serial.bas** supports the following public routines:

- RS485Init
- RS485Term
- RS485Start
- RS485Stop
- RS485Send
- RS485Receive
- RS485FinishReceive

The following sections give the API signature and an abbreviated implementation code for each routine. Full source code can be found in the attached ZIP file.

RS485Init

This subroutine opens the RS-485 serial communication channel. It assumes that the caller has already defined the communications channel using the DefineCom or DefineCom3 library function. The output queue for the serial communication channel is hard-coded to 32 bytes not including the queue overhead. The programmer can define the size of the output queue and *RS485Init()* routine creates the output queue from dynamic memory. The reason is that the minimum size of the output queue is highly dependent on the data packet size and the baud rate. See page 16 for a discussion of network performance.

```
Public Sub RS485Init(ByVal serialChannel as Byte, ByVal baud as Long, _
    ByVal sendReceive as Byte, ByVal queueSize as UnsignedInteger)
    ' store configuration values for later use
    channel = serialChannel
    baudrate = baud
    sendReceivePin = sendReceive

    ' just to be sure, enable RS485 receiver and disable the driver
    Call PutPin(sendReceivePin, zxOutputLow)

    ' create storage for queue dynamically
    inQueueAddr = System.Alloc(queueSize+9)

    ' open the serial port
    Call OpenQueue(CByteArray(inQueueAddr), queueSize+9)
    Call OpenQueue(outQueue, sizeof(outQueue))
    Call OpenCom(channel, baudRate, CByteArray(inQueueAddr), outQueue)

    ' check that the serial channel correctly opened
    If StatusCom(channel) <> 3 Then
        Call DebugPrint("Could not open serial channel " & CStr(channel) & CRLF)
        channel = 0
    End If
End Sub
```

RS485Term

This subroutine closes the RS-485 serial communication channel and should be paired with *RS485Init()*.

```
Public Sub RS485Term()
    Call CloseCom(channel, CByteArray(inQueueAddr), outQueue)
    Call System.Free(inQueueAddr)
End Sub
```

AN-218 Implementing a RS-485 Multi-drop Network

RS485Start

This subroutine starts the RS-485 serial communication channel and a task to receive data packets. An important part of this subroutine is that the caller creates the buffer which is used by the message receive code.

```
Public Sub RS485Start (ByRef buffer() as Byte, ByVal size as UnsignedInteger, _
                    ByVal addr as Byte)
    ' make sure channel exists
    If channel <> 0 Then
        ' save configuration data
        bufAddress = buffer.DataAddress
        bufSize = size
        deviceAddress = addr

        ' Calculate receive the message frame timeout in clock ticks.
        ' The timeout is at least 3.5 characters and is dependent
        ' on baud rate. Minimum value is at least 1 tick (1.95 ms)
        messageFrameInterval=CByte(3.5*11.0*CSng(Register.RTCTickFrequency)/CSng(baudrate))+1

        ' clear the semaphore for message receiving
        bufLocked = False

        ' start state for receive task
        status = Waiting
        error = NoData

        ' enable RS485 receiver
        Call PutPin(sendReceivePin, zxOutputLow)

        ' start the task to receive RS485 messages
        CallTask "receiveDataPacket", receiveTaskStack
    End If
End Sub
```

RS485Stop

This subroutine stops a RS-485 serial communication channel and should be paired with *RS485Start()*. At this point the receive task is halted and the buffer allocated by the caller is no longer used by the receive task.

```
Public Sub RS485Stop()
    ' make sure channel exists
    If channel <> 0 Then
        ' exit the RS485 task to receive RS485 messages and clear the semaphore
        ExitTask receiveTaskStack
        bufLocked = False
    End If
End Sub
```

RS485Send

This subroutine sends a message on the RS-485 serial channel. The CRC for the message is calculated and added before sending the message. There are two key parts to the implementation of this routine:

1. If the message buffer is larger than the output queue then it is broken up into "chunks" where each chunk is at most half of the output queue size. When the output queue is less than half full, another chunk is added to the queue. This keeps the output queue fed with data.
2. If the output queue is more than half full, then the send routine waits until it is less than half full. The sleep time is calculated based on the baud rate of the serial channel. If the output queue is still more than half full, then the routine continues waiting. There is no timeout on this wait as the RS-485 transceiver can send data even if no other device is listening. For the last chunk of data the routine waits until the output queue is empty and the final stop bit has been sent on the network. The internal subroutine *waitForSend()*, which is not listed here, performs the actual wait for a given buffer size.

AN-218 Implementing a RS-485 Multi-drop Network

```
Public Sub RS485Send(ByRef buffer() as Byte, ByVal length as UnsignedInteger)
    ' setup pointer to buffer
    Dim addr as UnsignedInteger
    Dim data as Byte Based addr

    ' calculate and add CRC as last two bytes of buffer
    Dim CRC as UnsignedInteger
    CRC = CRC16(buffer, length, &H8005, &Hffff, &H03)
    addr = buffer.DataAddress+length
    data = LoByte(CRC)
    addr = addr + 1
    data = HiByte(CRC)
    length = length + 2

    ' start sending of message by enabling RS485 driver
    Call PutPin(sendReceivePin, zxOutputHigh)

    ' send half a transmit buffer's worth of data at a time until everything is sent
    addr = buffer.DataAddress
    Do While length <> 0
        ' calculate how much more data to send
        Dim chunkSize as UnsignedInteger
        If length > OUTPUT_BUFSIZE \ 2 Then
            chunkSize = OUTPUT_BUFSIZE \ 2
        Else
            chunkSize = length
        End If
        length = length - chunkSize

        ' queue up next chunk of data
        Call PutQueue(outQueue, data, chunkSize)
        addr = addr + chunkSize

        ' wait until the transmit buffer is less than half full
        Call waitForSend(CINT(OUTPUT_BUFSIZE \ 2))
    Loop

    ' wait for remaining bytes to finish sending i.e. transmit buffer is empty
    Call waitForSend(0)
    Call Sleep(messageFrameInterval)

    ' disable the RS485 driver and re-enable the receiver
    Call PutPin(sendReceivePin, zxOutputLow)
End Sub
```

RS485FinishReceive

This subroutine must always be called after an *RS485Receive()* to free up the system resources and error status for the next receive.

```
Public Sub RS485FinishReceive()
    ' resets the state for the next receive
    status = Waiting
    error = NoData

    ' clears the semaphore
    bufLocked = False
End Sub
```

AN-218 Implementing a RS-485 Multi-drop Network

RS485Receive

This function attempts to receive some data from the RS-485 serial channel. It either returns the size of the data received or an error code (which is $> \&HFFF0$). As mentioned earlier the receive function depends heavily on a separate task to retrieve the data packet from the network.

The *RS485Receive()* function determines if the receive task is in the middle of receiving or has just received a message. In this case the function waits on a shared semaphore and then retrieves either the size of the message or an error code. Otherwise the function returns either the last error code that defaults to no data received.

```
Public Function RS485Receive() as UnsignedInteger
    ' make sure channel exists
    If channel = 0 Then
        RS485Receive = CUInt(RS485ErrorStatus.NoConnection)
        Exit Function
    End If

    ' If there is no current error then we are either in the
    ' middle of receiving data or have already received it
    If error = NoError Then

        ' Wait until the receiveDataPacket task releases the semaphore.
        Do While Not Semaphore(bufLocked)
            Call Sleep(SEM_WAIT_TIME)
        Loop

        ' if the data is ready then return size of message
        ' not including the final 2 bytes of the CRC
        If status = BufReady Then
            RS485Receive = dataSize - 2
        Else
            ' Otherwise return the error code and clear the semaphore.
            ' It is possible but very unlikely that no error code
            ' has been set by this time. This would be an internal logic error
            RS485Receive = CUInt(error)
            bufLocked = false
        End If
    Else
        ' Otherwise return the current error (defaults to NoData)
        RS485Receive = CUInt(error)
    End If
End Function
```

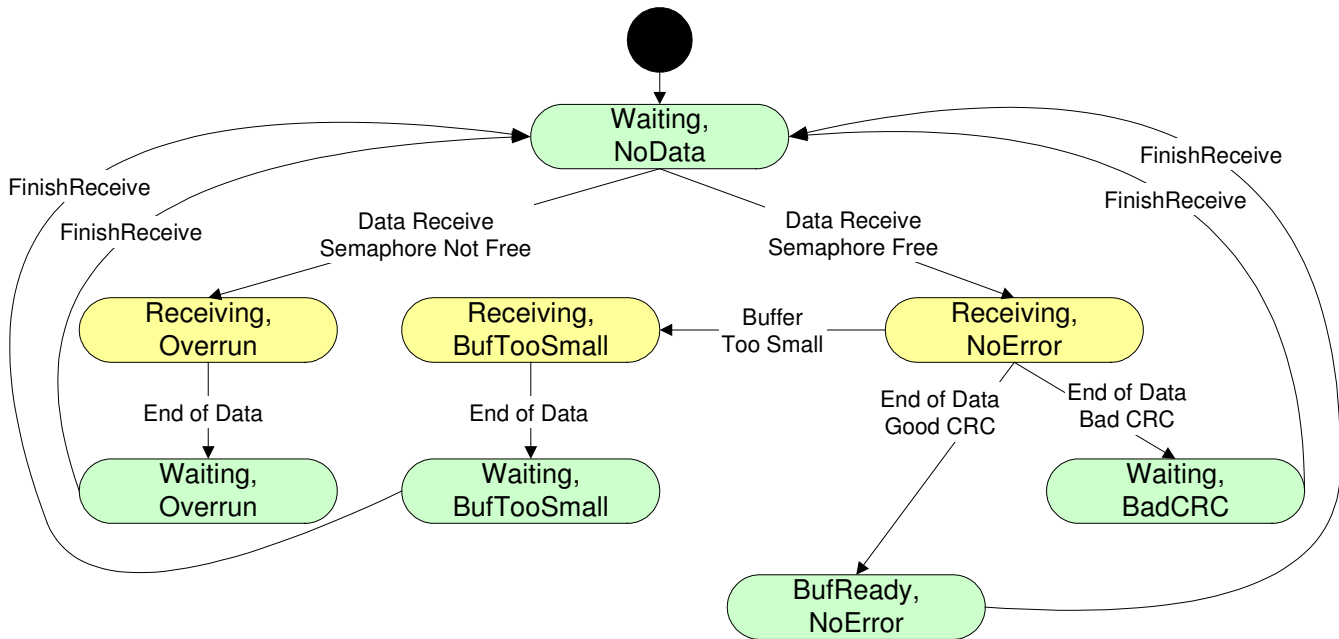
Note that if there is an error, it is cast from the ErrorStatus enum to an UnsignedInteger so that this function can return either a length or an error. Here is the definition of the various error status codes.

```
Public Enum RS485ErrorStatus
    NoError = &HFFF0    ' used if there is valid data i.e. no error
    NoConnection        ' used if serial channel did not open correctly
    NoData              ' default error if there is no data
    BufTooSmall        ' this is a programmer error that needs correcting
    BadCRC              ' the CRC is incorrect for the received message
    Overrun             ' used if a message is received while the
                        ' application task has the semaphore i.e. is busy
End Enum
```


AN-218 Implementing a RS-485 Multi-drop Network

The receive message task is a state machine that is a composition of two state variables called *status* and *error* that have corresponding enum types named *receiveStatus* and *errorStatus*. Receive status (or simply *status*) has two main states of either idle or *Receiving* data bytes from the input queue. The idle state has two sub-states that represent either the successful receipt of a message (*BufReady*) or waiting for the next message (*Waiting*). Error status (or simply *error*) has two main states or *NoError* and error. The error state has sub-states for each of the different error conditions including *BufTooSmall*, *BadCRC*, *Overrun*, and *NoData*. Although the *NoData* error status is not an error in application terms, it is considered one to simplify the state machine.

The diagram below shows the interaction of the *status* and *error* composite states. Each state rectangle shows the receive status and the error status. The color of the rectangle indicates if the task is at idle (green) or is actively receiving (yellow). Loops on states are not shown. The text on each transition indicates when the state machine moves to a new composite state.



A semaphore is used between the user task and the *receiveDataPacket()* task to manage sharing of the data buffer and the error status. Assuming the semaphore is not set, the *receiveDataPacket()* task tries to get the semaphore when data is first received in the input queue. If the user task has the semaphore then the receive status is set to *overrun*. Otherwise the semaphore is cleared after the *receiveDataPacket()* task has received the current packet. The semaphore cleared even if the data packet was received with an error. The user task now has an opportunity to get the semaphore, process the data or error status and release the semaphore by calling the *RS485FinishReceive()* routine.

Note that there are three conditions under which the receive message task takes bytes from the input queue and puts them into the receive buffer. In other cases or when there is an error, then the input queue is simply cleared. The three conditions are:

1. This device's address is zero, which means it is the master
2. This device's address matches the address in the message, which means the message is for this slave.
3. This address in the message is zero, which means this is a broadcast message

The code for the receive message task is not included and it is left as an exercise for the reader to relate the various parts of the *receiveDataPacket()* subroutine back to the description above.

Sample Application

The code for the sample application is in the **AN218.bas** module. The purpose of the application is for the master to send multiple messages to each of the slaves on a RS-485 network and note which slaves receive all of the messages. Then the master waits a while and starts over trying all of the slaves that did not successfully receive all of the messages. The sections below show some sample output and describe the algorithms used for address resolution, master, and slave.

Here is the common master and slave code from the *Main()* subroutine. The *getAddress2()* function which is described in the section on address resolution determines if the master() or slave() subroutine should be called.

```
Public Sub Main()  
    ' Sleep before getting things started  
    ' This is a must-have if using COM1 for RS-485  
    Call PutPin(Pin.GreenLED, zxOutputHigh)  
    Call PutPin(Pin.RedLED, zxOutputHigh)  
    Call Sleep(2.0)  
  
    ' Configure serial port for debug output, In this case  
    ' we are using COM1 and the pin number is not necessary  
    Call DefineDebugPort(DEBUG_CHANNEL, 19200, DEBUG_PIN)  
  
    ' retrieve the device id, 0 means master  
    Dim id as Byte  
    id = getBufferedAddress()  
  
    ' define the serial port and open a RS-485 serial channel with  
    ' 8 bits, no parity and one stop bit  
    Call CloseCom(RS485_CHANNEL, 0, 0)  
    Call DefineCom(RS485_CHANNEL, RX_PIN, TX_PIN, &H08)  
    Call RS485Init(RS485_CHANNEL, BAUDRATE, EN_PIN, QUEUESIZE)  
    Call RS485Start(buffer, BUFSIZE, id)  
  
    ' depending on the id, call the master or slave routine  
    If id = 0 Then  
        Call master()  
    Else  
        Call slave(id)  
    End If  
  
    ' we are done so close the RS-485 serial channel  
    Call RS485Stop()  
    Call RS485Term()  
End Sub
```

Sample Output

Here is some sample output from the Master. It first succeeds in sending data to slave #2 and then on the next round it succeeds with slave #4.

```
Starting master  
Status: 0001  
Sending data to slave 1...failed to receive data  
Sending data to slave 2...data received successfully  
Sending data to slave 3...failed to receive data  
Sending data to slave 4...failed to receive data  
Sending data to slave 5...failed to receive data  
:  
:  
Sending data to slave 14...failed to receive data  
Sending data to slave 15...failed to receive data  
Status: 0005  
Sending data to slave 1...failed to receive data  
Sending data to slave 3...failed to receive data
```

AN-218 Implementing a RS-485 Multi-drop Network

```
Sending data to slave 4...data received successfully
Sending data to slave 5...failed to receive data
.
.
Sending data to slave 14...failed to receive data
Sending data to slave 15...failed to receive data
Status: 0015
```

Here is the corresponding output for the slave with address 2. Slave 4 has similar output.

```
Starting slave 2
126 bytes in packet 1 Received
126 bytes in packet 2 Received
126 bytes in packet 3 Received
126 bytes in packet 4 Received
126 bytes in packet 5 Received
126 bytes in packet 6 Received
126 bytes in packet 7 Received
126 bytes in packet 8 Received
126 bytes in packet 9 Received
126 bytes in packet 10 Received
```

While the application is running, various LEDs are used to signal activity, success and failure. The “transmit” LEDs on the RS-485 header boards show when the corresponding ZX device has enabled the RS-485 driver. Except for very low baud rates it appears that the master and slave LEDs are on at the same time because the slave response message is very short.

The green LED on the slave is lit as each data packet is successfully received and stays lit after all the data packets have been received. The red LED on the slave is used to indicate when there is a receive error.

The green LED on the master is used to indicate that the status bits have changed from one cycle around the slaves. The LED is off until all the slaves have been sent at least one data packet, is lit during the wait time if there has been a change and is then turned off again for the next cycle. If all the slaves receive the data successfully then the green LED is permanently lit and the master stops sending data.

In order to fully appreciate the sequence of events between the master and a slave, a low-resolution video has been included in the associated ZIP file that shows a successful send of data to a single slave. The best way to view the video and the associated documentation is to open the html file in a browser.

Address Resolution

The same source module (AN218.bas) is used for both the master and slave code. Each slave is assigned an address in the range 1 to 15 and the master has an address of 0. The idea is that the same code can be loaded into any of the ZX devices on the network and they will use an external mechanism to determine their address. As an alternative the address could be “burnt” into EEPROM.

Because the range of addresses is 0 to 15, four I/O lines are required to determine a unique address. Connecting I/O pins to ground and using the built-in AVR chip pull-up resistors is a very easy way to achieve this. Here is code from the sample application that implements this function. It is assumed that if no inputs are connected to ground then this is the master so the bit pattern is complemented to retrieve the address.

```
Private Function getAddress() as Byte
Call PutPin(B.0, zxInputPullup)
Call PutPin(B.1, zxInputPullup)
Call PutPin(B.2, zxInputPullup)
Call PutPin(B.3, zxInputPullup)

getAddress = Not (Register.PINB) And MAX_SLAVES
End Function
```

An alternative mechanism, which only requires one I/O pin, is to use a tri-state buffer and pull in the value of the four I/O pins after briefly enabling the buffer. The RS-485 network example on page 3 uses DIP switches and a non-inverting tri-state buffer. These I/O pins can be used later in the application for other purposes either using a second tri-state buffer or by simply connecting directly to the I/O pins. Here is code from the sample application that implements this function. In this example the D.6 pin is used to enable and disable the tri-state input buffer.

AN-218 Implementing a RS-485 Multi-drop Network

```
Private Function getBufferedAddress () as Byte
    ' set 4 bits as input
    Call PutPin(B.0, zxInputTristate)
    Call PutPin(B.1, zxInputTristate)
    Call PutPin(B.2, zxInputTristate)
    Call PutPin(B.3, zxInputTristate)

    ' Enable tristate buffer and read value.
    ' Note that it uses inverse logic - 0V means a 1
    Call PutPin(TRISTATE_PIN, zxOutputLow)
    getBufferedAddress = Not (Register.PINB) And MAX_SLAVES

    ' disable tristate buffer
    Call PutPin(TRISTATE_PIN, zxOutputHigh)
End Function
```

Implementation of Master

The *master()* subroutine is responsible for keeping track of which slaves have received all of the data messages and looping around until all the slaves have received the data. The *dataSent* variable is a bit mask that tracks the slave status and the green LED is lit if there is a status change at the end of an iteration around the slaves.

```
Private Sub master ()
    Call DebugPrint ("Starting master" & CRLF)

    ' each bit represents slaves 1 to 15, bit 0 is the master
    Dim dataSent as UnsignedInteger, oldDataSent as UnsignedInteger
    dataSent = 0

    ' mark the master bit as set
    Call PutBit (dataSent, 0, 1)
    Call DebugPrint ("Status: " & CStrHex (dataSent) & CRLF)
    oldDataSent = dataSent

    ' keep sending until every slave has received the data
    Do While dataSent <> &HFFFF
        ' turn off LED
        Call PutPin (Pin.GreenLED, zxOutputHigh)

        ' Try to send data to each slave in turn
        Dim s as Byte
        For s = 1 to MAX_SLAVES
            If GetBit (dataSent, s) = 0 Then
                Call PutBit (dataSent, s, sendDataToSlave (s))
            End If
        Next s

        ' turn on green LED if something changed
        Call DebugPrint ("Status: " & CStrHex (dataSent) & CRLF)
        If dataSent <> oldDataSent Then
            oldDataSent = dataSent
            Call PutPin (Pin.GreenLED, zxOutputLow)
        End If

        ' sleep for a bit before trying the remaining slaves
        If dataSent <> &HFFFF Then
            Call Sleep (2.0)
        End If
    Loop
End Sub
```

AN-218 Implementing a RS-485 Multi-drop Network

The `sendDataToSlave()` function is responsible for attempting to send data to a slave and returning the status of that attempt. If the first data packet is received then the remaining packets are tried one at a time until either one fails or they are all successful. The actual content of each data packet is dummied data. The most crucial byte is the first byte after the packet header indicates the packet number. In more sophisticated application protocols such as Modbus this byte is usually a command code that is used by the slave to determine the format of the rest of the packet and the function/command that the slave should perform.

Within the `sendDataToSlave()` function, `RS485Receive()` is called every retry interval until either the timeout expires, an error occurs or a packet is received with a good CRC. If error or a data packet is received then the `RS485FinishReceive()` is called to complete the receive process. The source code for `sendDataToSlave()` is not presented here and it left as an exercise for the reader to determine how `RS485Receive()`, `RS485FinishReceive()` and error statuses are used to determine if the data packet was successfully received by the slave.

For the sample application the timeout for a slave response is set for 100 clock ticks with a retry interval of every 10 clock ticks. For very slow baud rates (i.e. less than 1200 baud) the timeout should be increased to 500 clock ticks. The timeout may also need to be increased if it is known that the slave may take a long time to respond because of internal processing. There is a tradeoff here of how long to wait before deciding a slave is not responding versus a slave taking a long time to respond. This is where good protocol design is essential.

Implementation of each Slave

The function of each slave is to receive data packets and send a response to the master. The slave subroutine never exits and simply waits for data packets from the master. A bit mask is used to keep track of which data packets have been received. A more sophisticated application may implement a mechanism to only resend the missing packets or start from where the last packet failed.

The processing for each data packet is very simple in this sample application. The slave sets the bit in the packet bit mask and then responds to the master. The slave response is a very short five byte packet that consists of the slave address, packet number, number of bytes received, and the CRC. The message receive buffer is reused for the response and then `FinishReceive()` is called to indicate that the buffer is now free for the receive task to use.

```
Private Sub slave(id as Byte)
    Call DebugPrint("Starting slave " & CStr(id) & CRLF)

    ' keep track of packets received
    Dim packets as UnsignedInteger
    packets = Not (CUInt(Pow(2.0, CSng(PACKETCOUNT+1))) - 1)

    ' packet numbering starts at 1 so set 0 bit
    Call PutBit(packets, 0, 1)

    ' keep going forever listening and processing messages from the master
    Do
        ' check if all packets received and set status LEDs
        If packets = &HFFFF Then
            Call PutPin(Pin.GreenLED, zxOutputLow)
        Else
            Call PutPin(Pin.GreenLED, zxOutputHigh)
        End If
        Call PutPin(Pin.RedLED, zxOutputHigh)

        ' try to receive some data
        Dim size as UnsignedInteger
        size = RS485Receive()

        ' check for an error code
        If size > CUInt(RS485ErrorStatus.NoError) Then
            ' if real error (not just no data), then turn on error LED
            ' and tell receive routine that we saw the error code
            If size <> CUInt(RS485ErrorStatus.NoData) Then
```

AN-218 Implementing a RS-485 Multi-drop Network

```
        Call PutPin(Pin.RedLED, zxOutputLow)
        Call RS485FinishReceive()
        Call DebugPrint("Received error " & CStrHex(size) & CRLF)
    End If
Else
    ' we received a valid transmission so turn on valid packet LED
    Call PutPin(Pin.GreenLED, zxOutputLow)
    Call DebugPrint(CStr(size)&" bytes in packet "&CSTR(buffer(2))&" Received"&CRLF)

    ' Now process this message by setting the packet bit
    Call PutBit(packets, buffer(2), 1)

    ' For packet 3 we are going to introduce a slight delay to simulate
    ' slave activity.
    If buffer(2) = 3 Then
        Call Sleep(20)
    End If

    ' Send a simple response to the master as follows:
    '   byte 1 is slave ID - already set
    '   byte 2 is packet number - already set
    '   byte 3 is number of bytes received
    buffer(3) = CByte(size)
    Call RS485Send(buffer, 3)
    Call RS485FinishReceive()
End If

    ' wait for next transmission
    Call Sleep(TRANSMISSION_GAP)
Loop
End Sub
```

Development Considerations

This section contains some additional material that might be useful for either modifying or developing your own network code.

Debugging Network Applications

During the development of this application note, several debugging “tricks” were developed to help with two major problems:

1. Multiple devices executing different code
2. The COM1 port used for RS-485 networking rather than debugging with Debug.Print.

The solutions to these problems should be useful for other application development and are presented here for reference purposes. It might also be helpful to develop an Application Note for ZBasic debugging.

A host PC with multiple serial ports is essential for developing this type of application. A multi-port serial card or USB to serial port adapters can be used to overcome the hardware serial port limitations of many modern PCs and laptops. One of the serial ports should be reserved for programming the ZBasic device via the IDE.

In order to capture the data transmitted on the RS-485 network, a good serial port monitor such as HHD Software’s freeware Serial Monitor should be utilized. The Serial Monitor needs to be started first and then any simple serial port application such as Microsoft Terminal is used to open the serial port. The Microsoft terminal output is not very helpful but the hex output from the Serial Monitor shows exactly what is being transmitted on the bus.

A ZBasic logical serial port (e.g. COM3) can be used for network speeds up to 19,200 baud. For faster speeds COM1 is needed for the RS-485 transceiver. In this case you may still want debug output from the application. The **DebugPrint.bas** module contains some simple routines and constants that facilitate debug output via a logical COM port. The subroutine *DefineDebugPort()* defines the COM port and speed to use. For compatibility, even COM1 can be defined as the debug port. The second subroutine *DebugPrint()* is named to be similar to the

AN-218 Implementing a RS-485 Multi-drop Network

Debug.Print library routine but is not quite as versatile. It requires a single string parameter with extra characters for the carriage return and line feed.

Note that if you are using COM1 for the RS-485 network, then you may experience difficulties in downloading a new ZBasic program to the master because of a conflict on the same COM1 port. The sample application includes a two second startup delay and delays between iterations when the COM1 port is idle so that there is time to get a request to the ZBasic bootloader to download a new program. If you get stuck and the EEPROM is socketed as on the ZX-24ae then it is always possible to reprogram the SPI EEPROM to facilitate a ZBasic program download.

Incremental Design and Coding

Because of the nature of this problem, it can only really be tested with multiple devices interacting at the same time. This means that the code needs to be developed incrementally. It is not productive to write all the code and then try to figure out why it doesn't work. The author's approach was to write small pieces of new function that whenever possible only affected either the slave or the master. For example a master listening for a slave response was implemented quite late and only after the slave listening to the master was completely tested. For instructional purposes here is the list of high-level increments that were implemented. The items that span both columns were done on the master and slave at the same time. Monitoring of the RS-485 network is also helpful in this case.

- | <u>Master</u> | <u>Slave</u> |
|--|--|
| 1. Used a Modbus test framework on the host PC as a temporary master. | 2. Basic message receive in the same task as the main slave at 9600 baud on COM3 |
| | 3. Added table-driven CalcCRC16() function to validate a complete message. Later CRC function was added to the ZBasic. |
| 4. Developed packet format and master code to send 10 dummy packets to a single slave. | |
| 5. Master was configured by grounding a single I/O pin | 6. Changed slave to receive new packet format and light green LED if successful. |
| | 7. Added support for configurable slave address. |
| | 8. Changed speed to 19,200 baud |
| 9. Added support for configurable master/slave address. | 10. Moved receive code into a separate task and added a semaphore between the main task (<i>RS485Receive</i>) and the receive task. This code was difficult to get right and there were several iterations on the correct design of the receive status and error status state machine. |
| | 11. Added slave response packet using existing code from master to send a packet (<i>RS485Send</i>). |
| 12. Added receive code to master to look for slave response and track received packets. | |
| 13. Loop to cycle around 15 slaves and use green LED to indicate success. | |
| 14. Added debugging to COM4 port in preparation for using COM1 port for faster networking | 15. Added COM4 debugging code to slave. |
| 16. Tested various speeds over 19,200 baud and spent some time debugging mainly the slave code. Some baud speed specific code was made more generalized. | |
| 17. Tried progressively lower speeds down to 300 baud. Only change required was timeout for master receive. | |
| | 18. Implemented complete tri-state buffer input mechanism rather than simply grounding I/O pins. |
| 19. Fine-tuning of code - mostly variable name changes and adding more comments to code. | |

AN-218 Implementing a RS-485 Multi-drop Network

Network Performance

The serial interface has been tested with baud rates between 300 and 460,800 and with different size data packets from 32 bytes to 512 bytes. The table below shows the input queue size needed to achieve "sustained" transfers at a given baud rate and packet size. Sustained because the master sends a packet, waits for a slave acknowledgement and then sends the next packet with very little wait time between each transfer. A transmission is deemed successful if ten data packets are sent and the slave acknowledges every packet.

Baud Rate	Packet Size			
	32	128	192	256
57,600	32	32	32	32
76,800	32	48	64	64
115,200	32	48	80	Fails
230,400	32	96	Fails	Fails
460,800	32	Fails	Fails	Fails

The default input queue size is 32 bytes. As the baud rate or packet size is increased the input queue size needs to be increased. At some point there is a failure mode and the data transfers are no longer successful - even when the input queue size is increased to a very large size. This is probably because the ZBasic code cannot keep up with the demand of receiving so much data so quickly.

The good news is that up to 460,800 baud is possible with a small enough packet size (32 bytes) and a fairly fast data rate of 57,600 baud can handle any packet size up to 256 bytes with only a 32 byte input queue. Even if the packet size is increased to 512 bytes, 57,600 baud still only requires a 48 byte input queue.

Summary

This application note addresses the complex problem of networking ZX devices so that they can work collaboratively. It pushes the envelope of what ZBasic can do and shows something more than just simple interfacing to another device. The **RS485Serial.bas** module is production level code that could be used in an industrial application.

This application note also lays the foundation for further work such as implementations of a Modbus master and slave. The author is also investigating how a master can replicate its ZBasic program to all the slaves on a network by sending multiple packets of data to each slave using a mechanism similar to the sample application.

Author

Mike Perks is a professional software engineer who became interested in microcontrollers a few years ago. Mike has written a number of articles, projects and application notes related to ZBasic, BasicX and AVR microcontrollers. Mike is also the owner of Oak Micros which specializes in AVR-based devices including his own ZX-based products. You may contact Mike at mikep@oakmicros.com or visit his website <http://oakmicros.com>.

e-mail: support@zbasic.net

Web Site: <http://www.zbasic.net>

Disclaimer: Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Copyright © Mike Perks 2007. All rights reserved. ZBasic, ZX-24, ZX-40, ZX-44 and combinations or variations thereof are trademarks of Elba Corp. or its subsidiaries. Other terms and product names may be trademarks of other parties.