SimpleModbusMaster

SimpleModbusMaster is a Modbus master library that allows Modbus/RTU communication via RS232 or RS485.

The library has 3 functions namely:

modbus_configure()

modbus_construct()

modbus_update()

To communicate with a slave you first need to create what's called a packet. In commercial PLC software terms this is usually called a block.

A packet is a struct that contains the relevant information linked to that specific packet. The Finite State Machine (FSM) running in the background scans each packet in a sequential manner continuously. All the information needed for the FSM is contained in a specific packet.

Each packet contains the following information:

unsigned char id;

unsigned char function;

unsigned int address;

unsigned int data;

unsigned int local_start_address;

unsigned int requests;

unsigned int successful_requests;

unsigned int failed_requests;

unsigned int exception_errors;

unsigned int retries;

unsigned char connection;

Using the modbus_construct() function you can initialize the packet.

E.g.

modbus_construct(&packets[PACKET1], 1, READ_HOLDING_REGISTERS, 0, 6, 0);

The modbus_construct() function accepts the following parameters:

Packet *_packet,
unsigned char id,
unsigned char function,
unsigned int address,
unsigned int data,
unsigned _local_start_address

The first parameter is the packet pointer and points to the address of the packet in the created array of packets.

The second parameter is the slaved device id range from 0-247 (247 is the standard but can go to 255). 0 is the broadcasting address for function 5, 6, 15 and 16.

The third parameter is the function code. Constants are provided for:

Function 1 - READ_COIL_STATUS 1 // Reads the ON/OFF status of discrete outputs (0X references, coils) in the slave.

Function 2 - READ_INPUT_STATUS 2 // Reads the ON/OFF status of discrete inputs (1X references) in the slave.

Function 3 - READ_HOLDING_REGISTERS 3 // Reads the binary contents of holding registers (4X references) in the slave.

Function 4 - READ_INPUT_REGISTERS 4 // Reads the binary contents of input registers (3X references) in the slave. Not writable.

Function 5 - FORCE_SINGLE_COIL 5 // Forces a single coil (0X reference) to either ON (0xFF00) or OFF (0x0000).

Function 6 - PRESET_SINGLE_REGISTER 6 // Presets a value into a single holding register (4X reference).

Function 15 - FORCE_MULTIPLE_COILS 15 // Forces each coil (0X reference) in a sequence of coils to either ON or OFF.

Function 16 - PRESET_MULTIPLE_REGISTERS 16 // Presets values into a sequence of holding registers (4X references).

Function 5 uses a whole unsigned int register to write a boolean value.

COIL_OFF 0x0000 // Function 5 OFF request is 0x0000

COIL_ON 0xFF00 // Function 5 ON request is 0xFF00

The fourth parameter is the address to read from or write to.

The fifth parameter is the index of the masters register map. More on this later...

The address, data and local_start_address parameter needs some further clarification.

The address parameter for function 1, 2, 5 and 15 is the number of boolean points. The address parameter for functions 3, 4, 6 and 16 is an unsigned int index from 0 – 65535. It must be noted however that some slaves accept the function as explicit or implicit which affects the address range. I.e. using function 3 as an example, READ_HOLDING_REGISTERS starts at address 40000 which is within the limits of an unsigned int so referring to address 40000 implicitly refers to function 3 and an address range of 0 – 9999, but when a slave refers to the function explicitly than the address can range from 0 – 65535.

The data parameter differs when using different functions namely:

For functions 1 & 2 data is the number of points. Points are stored in the masters unsigned int register map using shifting and padding.

For function 5 data is either ON (0xFF00) or OFF (0x0000)

For function 6 data is exactly that, one register's data

For functions 3, 4 & 16 data is the number of registers

For function 15 data is the number of coils

The last parameter is the index of the masters own address map which is basically an array of unsigned ints' with a finite size where data is read from and data is stored to from successful request from slaves.

As mentioned a packet contains all the information needed by the FSM. The information and various counters can be accessed by the user for each packet. The information counters can be used for diagnostic purposes. A brief explanation follows:

requests - contains the total requests to a slave

successful_requests - contains the total successful requests

failed_requests - general frame errors, checksum failures and buffer failures. These will result in a timeout.

retries - contains the number of retries exception_errors - contains the specific modbus exception response count. These are normally illegal function, illegal address, illegal data value or a miscellaneous error response.

And finally there is a variable called "connection" that at any given moment contains the current connection status of the packet. If true then the connection is active. If false then communication will be stopped on this packet until the programmer sets the connection variable to true explicitly.

Note:

The Arduino serial ring buffer is 64 bytes or 32 registers. This means the amount of data requested form a slave in a single packet is limited to the amount of bytes available:

Function 1 & 2 – maximum number of points are limited to 472 (59 bytes)

Function 3 & 4 – maximum number of registers are limited to 29 (58 bytes)

Function 15 – maximum number of points are limited to 440 (55 bytes)

Function 16 – maximum number of registers are limited to 27 (54 bytes)

A different packet needs to be created when the id, function, address, data or local start address changes. The easiest to do this is to create an array of packets and initialize the array to the FSM.

E.g.

```
enum
{
  PACKET1,
  PACKET2,
  TOTAL_NO_OF_PACKETS // leave this last entry
};
// Create an array of Packets to be configured

Packet packets[TOTAL_NO_OF_PACKETS];

modbus_configure(&Serial, baud, SERIAL_8N2, timeout, polling, retry_count, TxEnablePin, packets,
TOTAL_NO_OF_PACKETS, regs);
```

The modbus_configure() function takes the following parameters:

The first parameter is the address of the Serial object you want to communicate on. This will change to &Serial, &Serial1 etc. Depending on how many serial ports your Arduino board supports. Bear in mind that you cannot use the Serial object once assign to the FSM as this will corrupt the internal buffer.

The second parameter is the baud rate. Have a look at the baud rates implemented in the Arduino Hardware Serial library.

The third parameter is the byte format. Valid modbus byte formats are:

SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits

SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit

SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit

You can obviously use SERIAL_8N1 but this does not adhere to the Modbus specifications. That said, I have tested the SERIAL_8N1 option on various commercial masters and slaves that were suppose to adhere to this specification and was always able to communicate... Go figure.

Note: The byte format option is removed from the function for the Arduino Due since it is to date not supported in V1.5.7.

The fourth parameter is the timeout value. This is the time allowed for a slave to respond in. Common values are 1000ms – 5000ms.

The fifth parameter, the polling delay, is sometimes the most confusing to explain to users. It is the resting period between requests from the master to allow a slave to enter its idle state. This is because a slave also runs on an FSM and can only start responding to a request once the idle state is reached. Some quick acting slaves will revert to the idle state within 10ms but the usual slave will take around 100ms – 200ms.

The polling delay can be reduced as the amount of slaves increase on the bus line due to the inherent delays when polling more than 2 or 3 slaves. The polling delay can also be used as a “scan rate” between packets. I.e. increasing the polling delay allows more time to pass before the next packet is fetched.

The sixth parameter is the retry count. Once a packet is assembled and the request is transmitted the master will either wait for the timeout, a successful request or a failed request. The failed request can either be a timeout or an exception error from the slave. A failed request will increment the packets internal retry count attribute until the retry count is reached. Once reached the packet is switched off. The packet is switched off by setting the connection attribute of the packet to 0. The connection attribute can also be switched off and on explicitly. You will have to enable the packet manually once the FSM has turned it off. The reason for the connection attribute is because of the time out involved in Modbus communication. Each faulty slave that's not communicating will slow down communication on the line with the time out value. E.g. using a time out of 1500ms, if you have 10 slaves and 9 of them stops communicating the latency burden placed on communication will be 1500ms * 9 = 13.5 seconds!

The seventh parameter, the TxEnablePin, is used in RS485 communication where the driver IC has to be toggled between receiving and transmitting mode. This is not used in RS232 but is still enabled and requires a pin value. You can used it for indication purposes to display when the driver is in Tx or Rx mode.

The eighth parameter is the address of the array of packets and the ninth is the size of the array.

The tenth parameter is the address of the masters own memory/register map. It is up to the user to make sure that the memory array is large enough to receive the data when a slave responds successfully or when writing from the memory array.

The last function namely modbus_update() is the only function used in loop. Its only function is to update the FSM. You can call modbus_update() more frequently if your sketch/program is large to enable the FSM to respond quicker.

This also brings me to the next point about using long delay() values in your sketch. Using delays longer than 100ms will affect the FSM negatively. It is not good practice to use large delays when coding. Rather use millis() to form some sort of crude multi-tasking.

E.g.

```
long previousMillis = 0;
long interval = 1000;

unsigned long currentMillis = millis();
if (currentMillis - previousMillis >= interval)
{
   previousMillis = currentMillis;
   // do some stuff
}
else
{
   // do some other stuff
}
```

Tips:

One last thing; Modbus standard only makes provision for unsigned int (16 bit). When 32 bit information is requested as with floats, long etc. it becomes necessary for the user to shift and convert from multiple unsigned ints'. I have found the following code snippets useful:

Scenario:

regs[0] and regs[1] are the masters unsigned int register map. These two unsigned int registers contains the 32 bit long value requested from a slave. In order for the data to make sense the unsigned ints needs to be shifted and casted.

E.g. two unsigned int to long

```
long temp = (long)regs[0] << 16 | regs[1];
```

This is under the assumption of regs[0] = MSB and regs[1] = LSB. If the value seems to look like garbage try swapping the registers around.

E.g. two unsigned int to unsigned long

```
unsigned long temp = (unsigned long)regs[0] << 16 | regs[1];
```

This is under the assumption of regs[0] = MSB and regs[1] = LSB. If the value seems to look like garbage try swapping the registers around.

E.g. two unsigned int to float

```
float num;
unsigned long temp = (unsigned long)regs[0] << 16 | regs[1];
num = *(float*)&temp;
```

In order for the float value to be correctly casted the compiler needs to be directed to cast the address (not value) of the unsigned long to a float and then de-reference the temp variable address (obtaining the value) and assign it to the float variable. This is under the assumption of regs[0] = MSB and regs[1] = LSB. If the value seems to look like garbage try swapping the registers around.

To summarize:

1. Create an array of packets using the enum instruction:

```
enum
{
    PACKET1,
    PACKET2,
    TOTAL_NO_OF_PACKETS // leave this last entry
};
// Create an array of Packets to be configured

Packet packets[TOTAL_NO_OF_PACKETS];
```

2. Define the TOTAL_NO_OF_REGISTERS to initialize the size of your masters register map.

```
#define TOTAL_NO_OF_REGISTERS 10
// Masters register array
unsigned int regs[TOTAL_NO_OF_REGISTERS];
```

3. Initialize the packet with the correct information:
   E.g.

```
modbus_construct(&packets[PACKET1], 1, READ_COIL_STATUS, 0, 100, 3);

modbus_construct(&packets[PACKET1], 1, READ_INPUT_STATUS, 0, 1, 0);

modbus_construct(&packets[PACKET1], 1, READ_HOLDING_REGISTERS, 0, 6, 0);

modbus_construct(&packets[PACKET1], 1, READ_INPUT_REGISTERS, 0, 6, 0);

modbus_construct(&packets[PACKET1], 1, FORCE_SINGLE_COIL, 0, 1, 0);

modbus_construct(&packets[PACKET2], 1, PRESET_SINGLE_REGISTER, 1, 1, 9);

modbus_construct(&packets[PACKET2], 1, FORCE_MULTIPLE_COILS, 0, 16, 0);

modbus_construct(&packets[PACKET2], 1, PRESET_MULTIPLE_REGISTERS, 6, 6, 0);
```

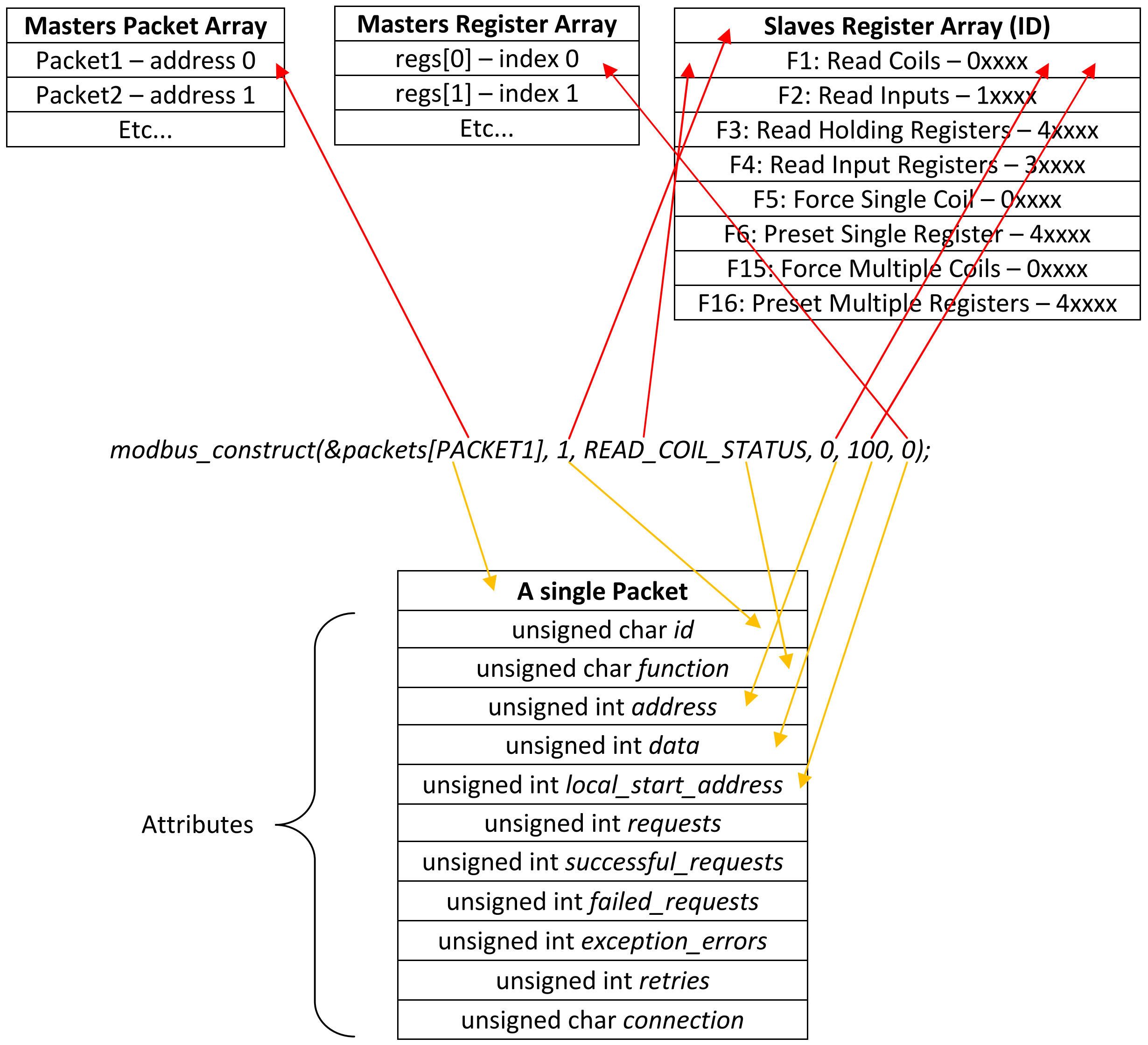
4. Initialize the Modbus FSM:

   E.g.

```
modbus_configure(&Serial, baud, SERIAL_8N2, timeout, polling, retry_count, TxEnablePin, packets,
TOTAL_NO_OF_PACKETS, regs);
```

5. Call modbus_update() in loop at least once.

Here is a pictorial view relating to the masters address space, modbus communication and the slaves address space:



Masters Packet Array
Packet1 – address 0
Packet2 – address 1
Etc...

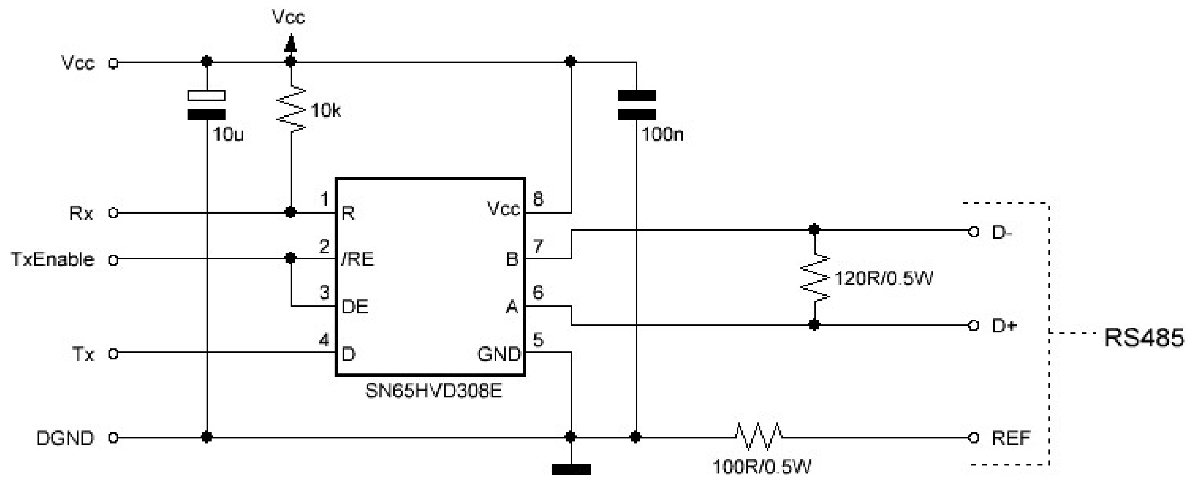
Masters Register Array
regs[0] – index 0
regs[1] – index 1
Etc...

Slaves Register Array (ID)
F1: Read Coils – 0xxxx
F2: Read Inputs – 1xxxx
F3: Read Holding Registers – 4xxxx
F4: Read Input Registers – 3xxxx
F5: Force Single Coil – 0xxxx
F6: Preset Single Register – 4xxxx
F15: Force Multiple Coils – 0xxxx
F16: Preset Multiple Registers – 4xxxx

modbus_construct(&packets[PACKET1], 1, READ_COIL_STATUS, 0, 100, 0);

A single Packet
unsigned char *id*
unsigned char *function*
unsigned int *address*
unsigned int *data*
unsigned int *local_start_address*
unsigned int *requests*
unsigned int *successful_requests*
unsigned int *failed_requests*
unsigned int *exception_errors*
unsigned int *retries*
unsigned char *connection*

Hardware

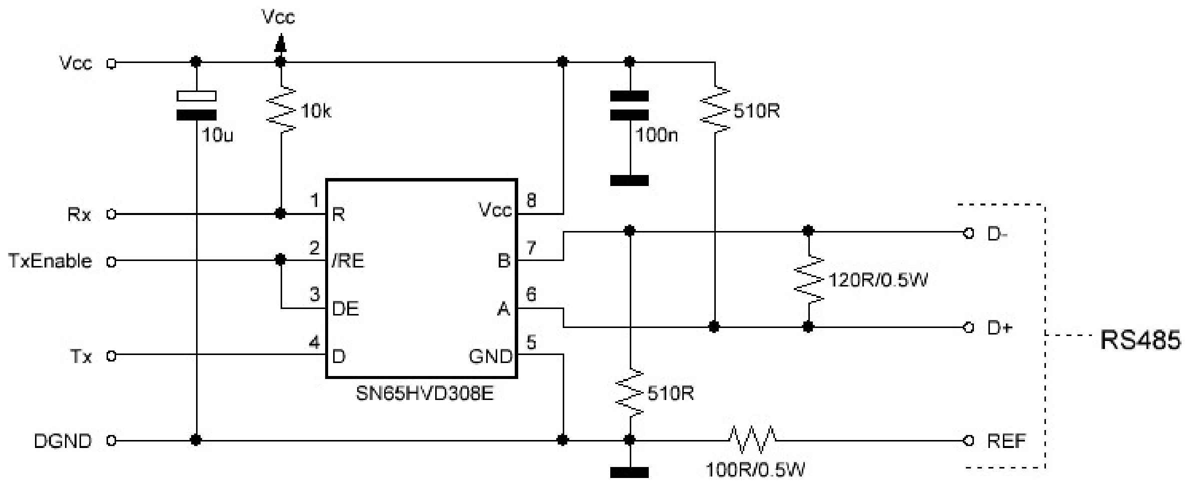


Above is the correct way to connect an RS485 driver IC to a micro-controller in a slave connection.

The 100 ohm resistor is there to minimize harmful ground loop currents between different ground potentials. I have found that this resistor is seldom used even in industrial units and as a result I have seen slave devices burst into flames! Some units do use fully isolated RS485 drivers but these are quite expensive. The capacitor is there to decouple supply spikes on every transmission. The 10k pull up is necessary to eliminate arbitrary switching on reception due to noise mostly due to bad lay out design. Beware that most manufacturers got the RS485 standard wrong on their datasheets with regards to the A and B lines. To avoid confusion manufacturers use D+ and D-.

Below is the correct way of terminating a driver IC in a master connection.

The 510 ohm pull up and pull down resistors are usually mounted on the master side. Theoretically it would be best to install it in the middle of your bus line. The 120 ohm termination resistor must always be used on distances greater than 1 meter and/or baud rates higher than 9600. It's best to install it as a norm as noise and reflections will cause havoc on your communication once implemented in the field. The termination resistors should be mounted at both ends of the bus line. Please note that adding bias resistors will load the driver IC output. With the indicated values the max number of units on the bus line is limited to eight 12kΩ, sixteen 24kΩ, or thirty-two 48kΩ units.



For further protection transient suppressors can be installed across the differential lines, from the Vcc to D+ and from GND to D-.

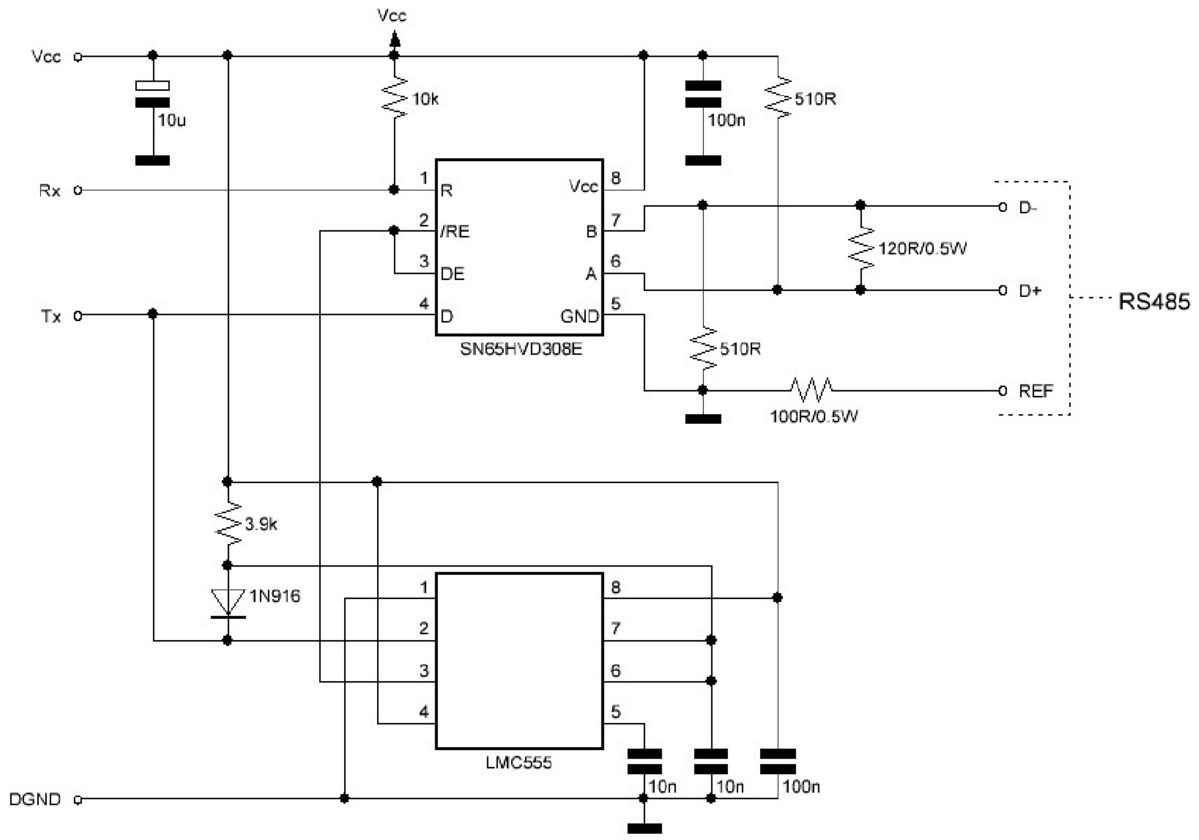
Note:

The new fail safe RS485 driver IC's like the MAX3440E or similar do not require bias resistors which allows you to drive the maximum number of slaves according to their input resistance.

Below is a very clever circuit I have been using and is suitable for industrial use.

I got the idea from Circuit Cellar. It does auto baud selection and auto Tx enable switching. The micro-controller only needs to interface with its Rx and Tx pins. Because the circuit switches automatically from reception to transmission without any controller logic it is perfectly suited for a standalone repeater circuit to extend the number of slaves on the bus line or to restore the drive current on long transmission lines. It works flawlessly at baud rates from 9600 to 115200. Faster baud rates require a decrease in the 3.9k timing resistor.

Note: the timer IC is of the CMOS type which is far superior to their bipolar counterpart.



I personally prefer toggling the TxEnable pin using the micro-controller as I have access to the bit timing if needed.

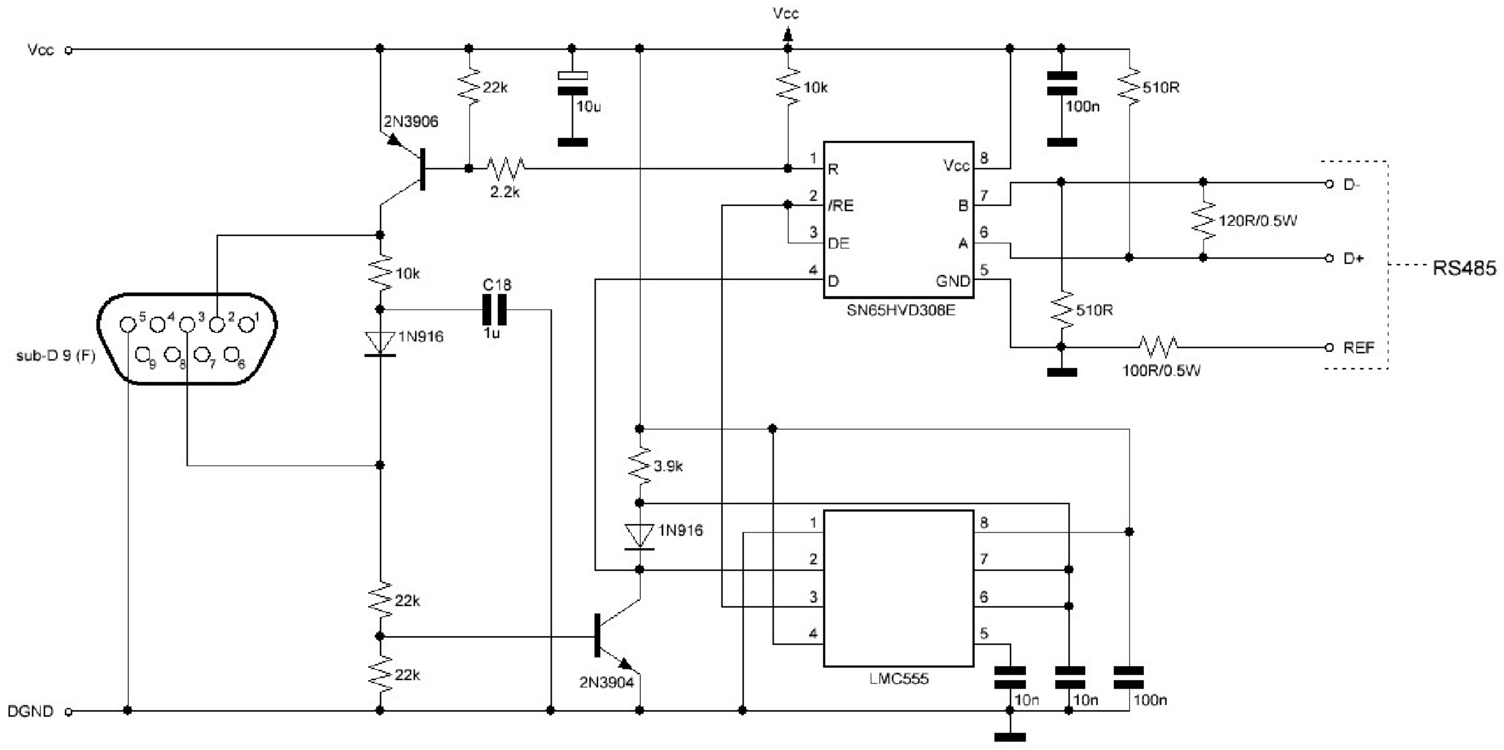
More information with regards to Modbus and RS485 can be found on my Google drive.

https://drive.google.com/folderview?id=0B0B286tJkafVY18xYVhmWG1qbkk&usp=sharing

RS232

The Tx and Rx pins are TTL232 (0v and 5V) compatible and not RS232 (±15V). To use RS232 communications a level shifter IC like the MAX232A or similar must be used. There is another very simple circuit that allows translation between the two levels.

Below is such a circuit. It uses a charge pump to “steel’ the negative voltage required for the RS232 standard.



Note:

This circuit has not been tested extensively.