**Gammon Forum**

Username: 
Register forum user name
Password: 
Forgotten password?
Log on

See www.mushclient.com/spam for dealing with forum spam. Please read the MUSHclient FAQ!

# RS485 communications

**Postings by administrators only.**

Refresh page

**Posted by** **Nick Gammon** Australia (21,294 posts) bio *Forum Administrator*

**Date** Mon 14 Nov 2011 11:48 PM (UTC)

Amended on Mon 05 Oct 2015 04:08 AM (UTC) by Nick Gammon

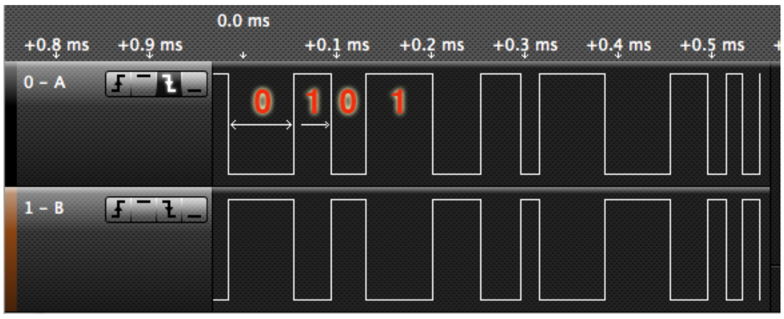
**Message** This post describes how you can connect multiple Arduinos together via an RS485 connection.

This is useful in situations where you need to connect devices together over longer distances than I2C or SPI can handle.

The RS485 protocol is described here:

http://en.wikipedia.org/wiki/Rs485

Basically it is an electrical protocol which allows you to communicate over long wires because it is "balanced". The graphic below shows th

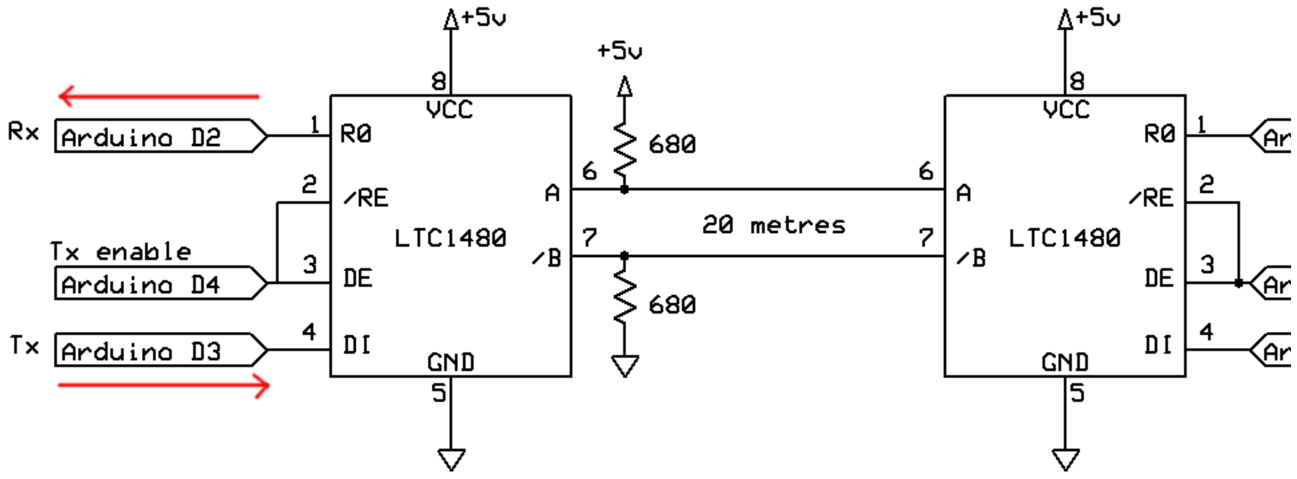


The "A" line shows a series of 0 and 1 bits, whilst the "B" line below is the inverse of the "A" line. This is more resistant to noise than simpl
ground, as small glitches of noise might be interpreted as data. This is because a "1" is when line A is higher than line B, and a "0" is when

To use RS485 with an Arduino we need an RS485 "transceiver" (transmitter/receiver) chip. These cost a couple of dollars and come in va
DIP-8 style, which was easy to breadboard with.

## Electrical connection

Below is how I wired the transceiver chip up:

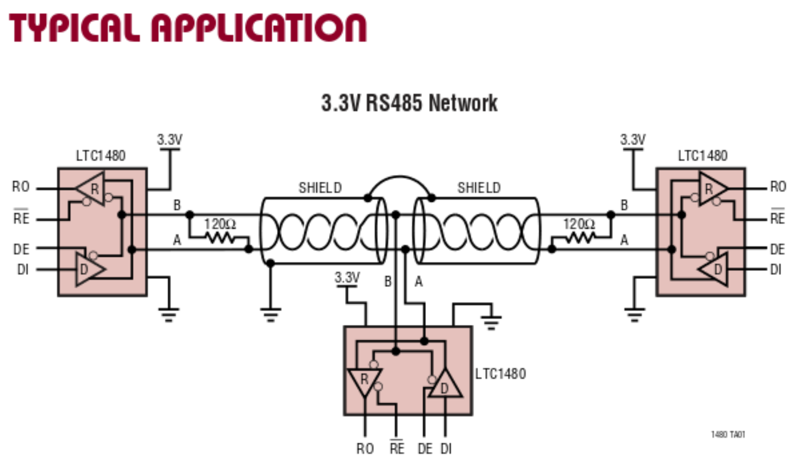


It needs power and ground, plus Rx/Tx connections. Pin 1 of the chip receives data from the A/B wires, and pin 4 is used to transmit data, high. To avoid contention only one device should be transmitting at once. The easiest way to arrange this is to design a single master / mu Then the master can make a request to a particular slave, and then wait for a response. An example of that is shown below.

The 680 ohm resistors are there to make sure that the A/B lines are in a "standard" state (A on, B off) if no tranceiver is configured for out thus avoiding noise from floating lines.

## Shield and ground return

The datasheet for the LTC1480 shows a shield over the wires, connected at **one end only**. Connecting at one end would reduce earth loo



Also an app note from TI: AN-1057 Ten Ways to bulletproof RS-485 Interfaces mentions that a ground wire should also be used, as follow

> **Quote:**
>
> Although the potential difference between the data-pair conductors determines the signal without officially involving ground, the bus need to provide a return path for induced common-mode noise and currents, such as the receivers' input current. A typical mistake is to connect only two wires. If you do this, the system may radiate high levels of EMI, because the common-mode return current finds its way back to th regardless of where the loop takes it. An intentional ground provides a low-impedance path in a known location, thus reducing emissions.

Also, Guidelines for Proper Wiring of an RS-485 (TIA/EIA-485-A) Network mentions:

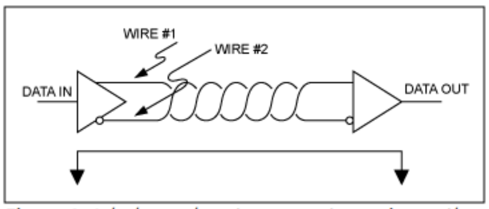


Figure 1. A balanced system uses two wires, other than ground, to transmit data.

> **Quote:**

A balanced system uses two wires, other than ground, to transmit data.

Also see Application Note 847 FAILSAFE Biasing of Differential Buses.

Also see below about termination resistors.

## Error checking protocol

The prudent designer would be worried about simply interpreting any incoming data as valid, without reasonable error checks. Noise on t
connected or disconnected half-way through a transmission, could be interpreted as valid data, when it isn't.

Hence I have written a small library that has the following features:

- Handles "packets" of between 1 and 255 bytes.
- Uses a "begin packet" character (Start of Text, STX, 0x02) to reliably indicate that a packet is starting.
- Uses an "end packet" character (End of Text, ETX, 0x03) to reliably indicate that a packet is ending.
- Each data byte (other than STX/ETX) is sent in a "doubled/inverted" form. That is, each nibble (4 bits) is sent twice, once normally,
  the only valid values for each nibble are:

  ```
  0F, 1E, 2D, 3C, 4B, 5A, 69, 78, 87, 96, A5, B4, C3, D2, E1, F0
  ```

  The inverse (ones complement) of 0 is F, hence 0 becomes 0F. The inverse of 1 is E, hence 1 becomes 1E. And so on.

  This guards somewhat against "bursts" of noise. A burst of either 0s or 1s is unlikely to corrupt a byte preserving this normal/invers
  are only 16/256 valid combinations, so noise has only a 6% chance of becoming a valid byte.

  Because of this, also, the STX and ETX characters cannot appear in ordinary data (they are not one of the 16 valid values).
- Each packet is followed by a CRC (cyclic redundancy check). This is a further test that the packet was received completely. It guards
  some bytes just becoming missing.

The library is available from:

http://www.gammon.com.au/Arduino/RS485_protocol.zip

Just unzip into your Arduino "libraries" folder (and restart the IDE).

## Callback functions

The library was written to allow for various hardware interfaces (eg. software serial, hardware serial, I2C). Thus when using it you supply
which have the job of doing the actual sending/receiving.

For hardware serial, they might look like this:

```
void fWrite (const byte what)
  {
  Serial.write (what);
  }

int fAvailable ()
  {
  return Serial.available ();
  }

int fRead ()
  {
  return Serial.read ();
  }
```

For software serial, you might use:

```
  #include <SoftwareSerial.h>

  SoftwareSerial rs485 (2, 3);  // receive pin, transmit pin

  void fWrite (const byte what)
    {
    rs485.write (what);
    }

  int fAvailable ()
    {
    return rs485.available ();
    }

  int fRead ()
    {
    return rs485.read ();
    }
```

## Master

It is your responsibility to turn on the "write enable" pin before and after doing a "send". This configures the RS485 chip to allow writing t
master is:

```
#include "RS485_protocol.h"
#include <SoftwareSerial.h>

const byte ENABLE_PIN = 4;
const byte LED_PIN = 13;

SoftwareSerial rs485 (2, 3);  // receive pin, transmit pin

// callback routines

void fWrite (const byte what)
  {
  rs485.write (what);
  }

int fAvailable ()
  {
  return rs485.available ();
  }

int fRead ()
  {
  return rs485.read ();
  }

void setup()
{
  rs485.begin (28800);
  pinMode (ENABLE_PIN, OUTPUT);  // driver output enable
  pinMode (LED_PIN, OUTPUT);  // built-in LED
}  // end of setup

byte old_level = 0;

void loop()
{

  // read potentiometer
  byte level = analogRead (0) / 4;

  // no change? forget it
  if (level == old_level)
    return;

  // assemble message
  byte msg [] = {
     1,    // device 1
     2,    // turn light on
     level // to what level
  };

  // send to slave
  digitalWrite (ENABLE_PIN, HIGH);  // enable sending
  sendMsg (fWrite, msg, sizeof msg);
  digitalWrite (ENABLE_PIN, LOW);  // disable sending

  // receive response
  byte buf [10];
  byte received = recvMsg (fAvailable, fRead, buf, sizeof buf);

  digitalWrite (LED_PIN, received == 0);  // turn on LED if error

  // only send once per successful change
  if (received)
    old_level = level;

}  // end of loop
```

This example demonstates how you might command a light in some other part of the house to dim up/down. It reads a potentiometer con
other sides of the pot connected to +5V and Gnd). This gives an analog reading which is then sent to the slave.

We use a 3-byte message format:

- Address of slave (eg. 1 to 255)
- Command (eg. 2 = turn light on)
- Parameter (eg. 128 = half level)

Then we wait for a response from the slave to confirm it got the message. If not, we turn on an "error" LED.

## Slave

The code for the slave could be:

```
#include <SoftwareSerial.h>
#include "RS485_protocol.h"

SoftwareSerial rs485 (2, 3);  // receive pin, transmit pin
const byte ENABLE_PIN = 4;

void fWrite (const byte what)
  {
  rs485.write (what);
  }

int fAvailable ()
  {
  return rs485.available ();
  }

int fRead ()
  {
  return rs485.read ();
  }

void setup()
{
  rs485.begin (28800);
  pinMode (ENABLE_PIN, OUTPUT);  // driver output enable
}

void loop()
{
  byte buf [10];

  byte received = recvMsg (fAvailable, fRead, buf, sizeof (buf));

  if (received)
    {
    if (buf [0] != 1)
      return;  // not my device

    if (buf [1] != 2)
      return;  // unknown command

    byte msg [] = {
       0,  // device 0 (master)
       3,  // turn light on command received
    };

    delay (1);  // give the master a moment to prepare to receive
    digitalWrite (ENABLE_PIN, HIGH);  // enable sending
    sendMsg (fWrite, msg, sizeof msg);
    digitalWrite (ENABLE_PIN, LOW);  // disable sending

    analogWrite (11, buf [2]);  // set light level
   }  // end if something received

}  // end of loop
```

The slave simply loops looking for incoming data. The library returns a non-zero "received count" if a valid message is received. Then the s
(first byte of the message) to see if it is addressed to it (rather than a different slave). If not, it ignores the message.

Then if checks for a valid command (eg. 2 = turn light on). If not, it ignores it.

Finally if it passes these checks, it sends back a response. A small delay (of 1 mS) is inserted to give the master time to prepare for a respo
knows the slave is alive, and responding.

## Flushing the output

A small "gotcha" caught me when testing with hardware serial. The following code didn't work properly:

```
digitalWrite (ENABLE_PIN, HIGH);  // enable sending
sendMsg (fWrite, msg, sizeof msg);
digitalWrite (ENABLE_PIN, LOW);  // disable sending
```

Whilst it worked fine with software serial, the code above "turns off" the RS485 chip too quickly, because the last byte is still being sent fr
port.

A couple of solutions worked:

```
digitalWrite (ENABLE_PIN, HIGH);  // enable sending
sendMsg (fWrite, msg, sizeof msg);
delayMicroseconds (660);
digitalWrite (ENABLE_PIN, LOW);  // disable sending
```

I'm not very happy with hard-coded delays. Too low and it is still too quick, too high and the slave is responding before you turn the trans
has to be carefully tuned, and depends very much on the baud rate in use. The value required appears to be appromately two character tin
one character time is 1/2880 which is 347 uS. Doubling that gives about 690 uS.

Another approach is:

```
digitalWrite (ENABLE_PIN, HIGH);  // enable sending
sendMsg (fWrite, msg, sizeof msg);

while (!(UCSR0A & (1 << UDRE0)))  // Wait for empty transmit buffer
    UCSR0A |= 1 << TXC0;  // mark transmission not complete
while (!(UCSR0A & (1 << TXC0)));   // Wait for the transmission to complete

 digitalWrite (ENABLE_PIN, LOW);  // disable sending
```

This requires fiddling with hardware registers (and the exact ones depend on whether you are using Serial, Serial1, Serial2 and so on). The
hardware chip's buffer to empty, at the same time setting the "transmission not complete" flag. The second loop waits for the final byte to
hardware.

## Termination resistors

If the transceivers are not at the ends of the cable termination resistors are probably necessary. Something like 120 ohms, connected betw
each end only, stop the signal reflecting back along the cable.

## Conclusion

This setup seemed to work pretty well. Transmission was over ordinary "speaker cable". It wasn't twisted pair or shielded. At 28800 baud
the command and receive a response. Turning the knob resulted in light on the second Arduino smoothly changing value "instantly" (to h

**[EDIT]** Amended 7 August 2013 to change from NewSoftSerial to SoftwareSerial.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

**Posted by** Nick Gammon Australia (21,294 posts) bio *Forum Administrator*

**Date** Reply #1 on Mon 03 Dec 2012 11:12 PM (UTC)

Amended on Mon 03 Dec 2012 11:15 PM (UTC) by Nick Gammon

**Message**

## Non-blocking version

I have made a "non-blocking" version of this library. The protocol is the same, but it lets you handle an incoming packet a byte at a time. This could be handy where you have multiple incoming devices.

This library is available from:

http://www.gammon.com.au/Arduino/RS485_non_blocking.zip

## Sending sketch

```
#include <RS485_non_blocking.h>

size_t fWrite (const byte what)
{
  return Serial.write (what);
}

RS485 myChannel (NULL, NULL, fWrite, 0);

void setup ()
{
  Serial.begin (115200);
  myChannel.begin ();
}  // end of setup

const byte msg [] = "Hello world";

void loop ()
{
  myChannel.sendMsg (msg, sizeof (msg));
  delay (1000);
}  // end of loop
```

There is now a RS485 class (which I made an instance of called myChannel). You supply in the constructor a "read", "available" and "write" callback. If you are only writing you don't need the read or available callbacks, as illustrated above. You also supply a maximum buffer length (this only applies for reading).

## Receiving sketch

```
#include <RS485_non_blocking.h>

 int fAvailable ()
   {
   return Serial.available ();
   }

 int fRead ()
   {
   return Serial.read ();
   }


RS485 myChannel (fRead, fAvailable, NULL, 20);

void setup ()
  {
  Serial.begin (115200);
  myChannel.begin ();
  }  // end of setup

void loop ()
  {
  if (myChannel.update ())
    {
    Serial.write (myChannel.getData (), myChannel.getLength ());
    Serial.println ();
    }
  }  // end of loop
```

In this sketch the loop function calls myChannel.update () which gradually assembles the incoming packet into a buffer inside the class instance. You need to call myChannel.begin () in setup to actually allocate the required amount of memory for it.

Once myChannel.update () returns true, a packet is ready, and you can then access its data, and the length of its data.

There are some other functions in the class which tell you whether a packet is in progress, when it started, how many errors have occurred, and so on.

- Nick Gammon

top

**Posted by** Nick Gammon Australia (21,294 posts) bio *Forum Administrator*

**Date** Reply #2 on Mon 07 Sep 2015 10:04 PM (UTC)

**Message**

## Rolling-master system

Following on from a question on the Arduino StackExchange I developed a system for having a "rolling master".

The design objective is to have:

- Multiple devices (eg. Arduino Unos or equivalent in a smaller form factor, such as a Nano or Pro Mini) in your house.
- Each is connected to a "bus" of a pair of wires (preferably twisted-pair like is used for Ethernet cabling)
- Each one has one or more sensors (eg. light sensor, switches, movement sensors) which it tests
- Each one may also control things like lights, TV power, curtains opening, etc.
- The system should be tolerant of any device going offline (ie, failing or being powered off)
- Each device reports its status to the other devices, so that a switch on device A might turn on a light connected to device B, for exam

What I implemented is:

- Each device has its own address, which it gets from EEPROM. eg. 0, 1, 2, 3, 4, 5 ...
- You choose a range of addresses you are going to use (eg. maximum of 10)
- When the device powers up it first listens for other devices "talking" on the bus. Hopefully it will hear at least one other (if not, see b
- We decide on a fixed "message packet", say of 50 bytes including address, CRC, etc. At 9600 baud that would take 52 ms to send.
- Each device gets a "slot" of time, and waits its turn to talk to the bus.
- When its timeslot arrives, it goes into output mode, and broadcasts its packet which includes its own address. Therefore all the othe status (and act upon it if necessary). Eg. device 1 might report that switch 3 is closed, which means device 2 must turn a light on.
- Ideally, you know your timeslot has arrived because your device address is one greater than the packet you just listened to. Eg. You device 2 announce its status. Now it's your turn. Of course, you wrap around at the maximum number, so after device 9 you go back
- If a device is missing and does not respond, you give it a timeslot worth of time to respond, and then assume it is dead, and every de assumes the next timeslot has started. (eg. You heard device 2, device 3 should respond, after 52 ms of inactivity, device 4 can now r device 52 ms to respond, which should be plenty even if it is servicing an interrupt or something like that.
- If multiple devices in sequence are missing you count a 52 ms gap for each missing device, until it is your turn.
- Once you get at least one response the timing is resynchronized, so any drift in clocks would be cancelled.

The only difficulty here is that upon initial power-up (which might happen simultaneously if the power to the building is lost and then res currently broadcasting its status, and thus nothing to synchronize to.

In that case:

- If after listening long enough for all devices to be heard (eg. 500 ms) and hearing nothing, the device tentatively assumes it is the fir broadcast. However possibly two devices will do that at the same time, and thus never hear each other.
- If a device has not heard from another device, it staggers the time between broadcasts randomly (seeding the random-number gene number, to avoid all devices "randomly" staggering the broadcasts by the same amount).
- This random staggering by additional amounts of time won't matter, because there is no-one listening anyway.
- Sooner or later one device will get exclusive use of the bus and the other ones can then synchronize with it in the usual way.
- This random gap between attempts to communicate is similar to what Ethernet used to do when multiple devices shared one coaxial

## Setting the device addresses

First you need to set up the current device address, and the number of devices, in EEPROM, so run this sketch, changing **myAddress** for

```
#include <EEPROM.h>

const byte myAddress = 3;
const byte numberOfDevices = 4;

void setup ()
  {
  if (EEPROM.read (0) != myAddress)
    EEPROM.write (0, myAddress);
  if (EEPROM.read (1) != numberOfDevices)
    EEPROM.write (1, numberOfDevices);

  }  // end of setup

void loop () { }
```

## Code for rolling master

This code uses the RS485 library described earlier in this thread to ensure that a "packet" of data is received reliably.

It is up to you what data you broadcast in the "message" structure below. In the example there is the device address (needed so we know w
by an array of 10 bytes (eg. these could be 10 switch positions) plus another "status" integer, which might be the result of reading from a li

```
/*
 Multi-drop RS485 device control demo.

 Devised and written by Nick Gammon.
 Date: 7 September 2015
 Version: 1.0

 Licence: Released for public use.

 For RS485_non_blocking library see: http://www.gammon.com.au/forum/?id=11428
 For JKISS32 see: http://forum.arduino.cc/index.php?topic=263849.0
*/

#include <RS485_non_blocking.h>
#include <SoftwareSerial.h>
#include <EEPROM.h>

// the data we broadcast to each other device
struct
  {
  byte address;
  byte switches [10];
  int  status;
  }  message;

const unsigned long BAUD_RATE = 9600;
const float TIME_PER_BYTE = 1.0 / (BAUD_RATE / 10.0);  // seconds per sending one byte
const unsigned long PACKET_LENGTH = ((sizeof (message) * 2) + 6); // 2 bytes per payload byte plus STX/ETC/CRC
const unsigned long PACKET_TIME =  TIME_PER_BYTE * PACKET_LENGTH * 1000000;  // microseconds

// software serial pins
const byte RX_PIN = 2;
const byte TX_PIN = 3;
// transmit enable
const byte XMIT_ENABLE_PIN = 4;

// debugging pins
const byte OK_PIN = 6;
const byte TIMEOUT_PIN = 7;
const byte SEND_PIN = 8;
const byte SEARCHING_PIN = 9;
const byte ERROR_PIN = 10;

// action pins (demo)
const byte LED_PIN = 13;
const byte SWITCH_PIN = A0;


// times in microseconds
const unsigned long TIME_BETWEEN_MESSAGES = 3000;
unsigned long noMessagesTimeout;

byte nextAddress;
unsigned long lastMessageTime;
unsigned long lastCommsTime;
unsigned long randomTime;

SoftwareSerial rs485 (RX_PIN, TX_PIN);  // receive pin, transmit pin

// what state we are in
enum {
   STATE_NO_DEVICES,
   STATE_RECENT_RESPONSE,
   STATE_TIMED_OUT,
} state;

// callbacks for the non-blocking RS485 library
size_t fWrite (const byte what)
  {
  rs485.write (what);
  }

int fAvailable ()
  {
  return rs485.available ();
  }

int fRead ()
  {
  lastCommsTime = micros ();
  return rs485.read ();
  }

// RS485 library instance
RS485 myChannel (fRead, fAvailable, fWrite, 20);

// from EEPROM
byte myAddress;        // who we are
byte numberOfDevices;  // maximum devices on the bus

// Initial seed for JKISS32
static unsigned long x = 123456789,
                     y = 234567891,
                     z = 345678912,
                     w = 456789123,
                     c = 0;

// Simple Random Number Generator
unsigned long JKISS32 ()
  {
  long t;
  y ^= y << 5;
  y ^= y >> 7;
  y ^= y << 22;
  t = z + w + c;
  z = w;
  c = t < 0;
  w = t & 2147483647;
  x += 1411392427;
  return x + y + w;
  }  // end of JKISS32

void Seed_JKISS32 (const unsigned long newseed)
  {
  if (newseed != 0)
    {
    x = 123456789;
    y = newseed;
    z = 345678912;
    w = 456789123;
    c = 0;
    }
  }  // end of Seed_JKISS32

void setup ()
  {
  // debugging prints
  Serial.begin (115200);
  // software serial for talking to other devices
  rs485.begin (BAUD_RATE);
  // initialize the RS485 library
  myChannel.begin ();

  // debugging prints
  Serial.println ();
  Serial.println (F("Commencing"));
  myAddress = EEPROM.read (0);
  Serial.print (F("My address is "));
  Serial.println (int (myAddress));
  numberOfDevices = EEPROM.read (1);
  Serial.print (F("Max address is "));
  Serial.println (int (numberOfDevices));

  if (myAddress >= numberOfDevices)
    Serial.print (F("** WARNING ** - device number is out of range, will not be detected."));

  Serial.print (F("Packet length = "));
  Serial.print (PACKET_LENGTH);
  Serial.println (F(" bytes."));

  Serial.print (F("Packet time = "));
  Serial.print (PACKET_TIME);
  Serial.println (F(" microseconds."));

  // calculate how long to assume nothing is responding
  noMessagesTimeout = (PACKET_TIME + TIME_BETWEEN_MESSAGES) * numberOfDevices * 2;

  Serial.print (F("Timeout for no messages = "));
  Serial.print (noMessagesTimeout);
  Serial.println (F(" microseconds."));

  // set up various pins
  pinMode (XMIT_ENABLE_PIN, OUTPUT);

  // demo action pins
  pinMode (SWITCH_PIN, INPUT_PULLUP);
  pinMode (LED_PIN, OUTPUT);

  // debugging pins
  pinMode (OK_PIN, OUTPUT);
  pinMode (TIMEOUT_PIN, OUTPUT);
  pinMode (SEND_PIN, OUTPUT);
  pinMode (SEARCHING_PIN, OUTPUT);
  pinMode (ERROR_PIN, OUTPUT);

  // seed the PRNG
  Seed_JKISS32 (myAddress + 1000);

  state = STATE_NO_DEVICES;
  nextAddress = 0;

  randomTime = JKISS32 () % 500000;  // microseconds
  }  // end of setup

// set the next expected address, wrap around at the maximum
void setNextAddress (const byte current)
  {
  nextAddress = current;
  if (nextAddress >= numberOfDevices)
    nextAddress = 0;
  }  // end of setNextAddress


// Here to process an incoming message
void processMessage ()
  {

  // we cannot receive a message from ourself
  // someone must have given two devices the same address
  if (message.address == myAddress)
    {
    digitalWrite (ERROR_PIN, HIGH);
    while (true)
      { }  // give up
    }  // can't receive our address

  digitalWrite (OK_PIN, HIGH);

  // handle the incoming message, depending on who it is from and the data in it

  // make our LED match the switch of the previous device in sequence
  if (message.address == (myAddress - 1))
    digitalWrite (LED_PIN, message.switches [0]);

  digitalWrite (OK_PIN, LOW);
  } // end of processMessage

// Here to send our own message
void sendMessage ()
  {
  digitalWrite (SEND_PIN, HIGH);
  memset (&message, 0, sizeof message);
  message.address = myAddress;

  // fill in other stuff here (eg. switch positions, analog reads, etc.)

  message.switches [0] = digitalRead (SWITCH_PIN);

  // now send it
  digitalWrite (XMIT_ENABLE_PIN, HIGH);  // enable sending
  myChannel.sendMsg ((byte *) &message, sizeof message);
  digitalWrite (XMIT_ENABLE_PIN, LOW);  // disable sending
  setNextAddress (myAddress + 1);
  digitalWrite (SEND_PIN, LOW);

  lastCommsTime = micros ();   // we count our own send as activity
  randomTime = JKISS32 () % 500000;  // microseconds
  }  // end of sendMessage

void loop ()
  {
  // incoming message?
  if (myChannel.update ())
    {
    memset (&message, 0, sizeof message);
    int len = myChannel.getLength ();
    if (len > sizeof message)
      len = sizeof message;
    memcpy (&message, myChannel.getData (), len);
    lastMessageTime = micros ();
    setNextAddress (message.address + 1);
    processMessage ();
    state = STATE_RECENT_RESPONSE;
    }  // end of message completely received

  // switch states if too long a gap between messages
  if  (micros () - lastMessageTime > noMessagesTimeout)
    state = STATE_NO_DEVICES;
  else if  (micros () - lastCommsTime > PACKET_TIME)
    state = STATE_TIMED_OUT;

  switch (state)
    {
    // nothing heard for a long time? We'll take over then
    case STATE_NO_DEVICES:
      if (micros () - lastCommsTime >= (noMessagesTimeout + randomTime))
        {
        Serial.println (F("No devices."));
        digitalWrite (SEARCHING_PIN, HIGH);
        sendMessage ();
        digitalWrite (SEARCHING_PIN, LOW);
        }
      break;

    // we heard from another device recently
    // if it is our turn, respond
    case STATE_RECENT_RESPONSE:
      // we allow a small gap, and if it is our turn, we send our message
      if (micros () - lastCommsTime >= TIME_BETWEEN_MESSAGES && myAddress == nextAddress)
        sendMessage ();
      break;

    // a device did not respond in its slot time, move onto the next one
    case STATE_TIMED_OUT:
      digitalWrite (TIMEOUT_PIN, HIGH);
      setNextAddress (nextAddress + 1);
      lastCommsTime += PACKET_TIME;
      digitalWrite (TIMEOUT_PIN, LOW);
      state = STATE_RECENT_RESPONSE;  // pretend we got the missing response
      break;

    }  // end of switch on state

  }  // end of loop
```

There are some debugging LED flashes, to help see if the system is alive or not. You can omit those when you are happy it works. Basically

```
// debugging pins
const byte OK_PIN = 6;
const byte TIMEOUT_PIN = 7;
const byte SEND_PIN = 8;
const byte SEARCHING_PIN = 9;
const byte ERROR_PIN = 10;
```

And then remove any references to those pins (setting them to output, high or low).

## What the demo does

As it currently stands, if you close a switch on A0 (short to ground) it will turn off a LED (pin 13) on the next highest device in sequence. T
are talking to each other. Of course in practice you would have something more sophisticated in the packet which is being broadcast.

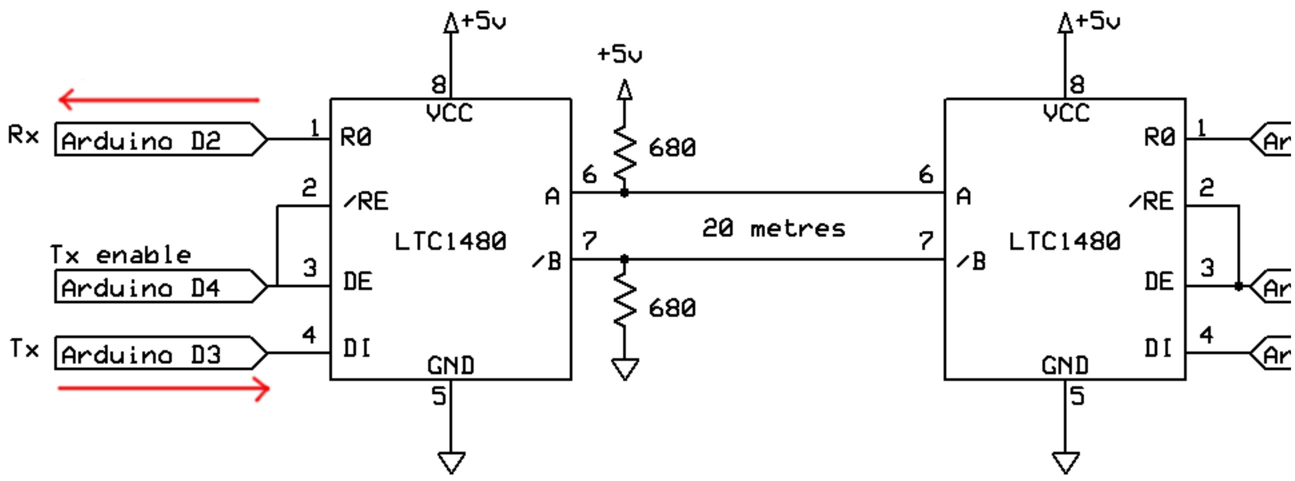
I found in testing that the LED appeared to come on and off instantly.

With all devices disconnected the first one connected will "hunt" for other devices. If you have debugging LEDs connected as I did you car
come on at random intervals as it broadcasts its packet with randomly-varying gaps. Once you have a second one connected they settle do
information. I tested with three connected at once.

It would probably be more reliable with HardwareSerial - I used SoftwareSerial to help with debugging. A few small changes would accom

## Hardware setup

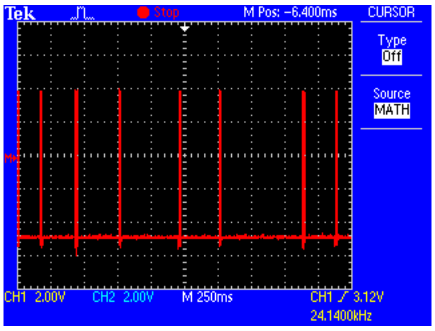
Here is an example of two devices connected together. Since they are symmetric they all need the read and write-enable pins to be connec



For additional ones, they just join the A/B lines at some appropriate point. We assume each Arduino is powered locally from a nearby pov

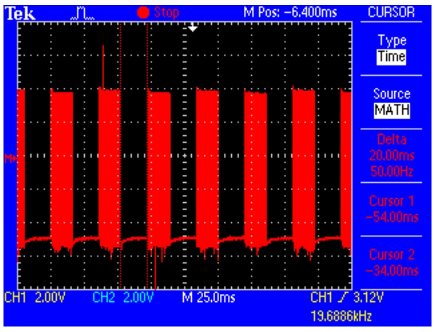
## Screen shots of timing

These images show the code working. First, with only one device connected:



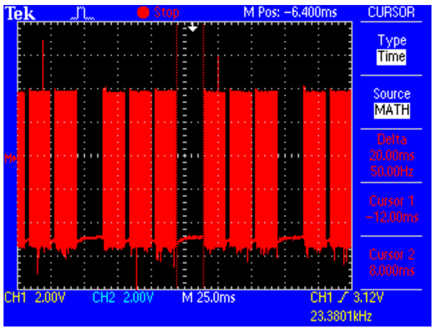
You can see from the pulses there that the device is broadcasting its data at randomly-varying intervals, to avoid continuing to clash with a
powered up at the same moment.

---



Now we see blocks of data from two devices, with roughly the same size gaps in the middle. I configured it for four devices, but only two ar
blocks of data, and two gaps.

---



Now with three devices online, we see three blocks of data, and one gap, as the missing device is bypassed.

## Cable run test

For a proper hardware test, I connected the devices up to my in-house UTP cabling. I have Cat-5 cable running from various rooms to a ce
from one end of the house to another (a reasonable length run) it still works fine. There is 5 m cable between the Arduino and the wall soc
another 5 m cable at the other end. Then there are about 2 x 15 m runs from the rooms to the switch room, and inside that there is a short
them together.

This was with the boards programmed to run at 19200 baud, so you could experiment to see if they were reliable at 19200 baud.

---

- Nick Gammon

www.gammon.com.au, www.mushclient.com

---

**Posted by** Nick Gammon Australia (21,294 posts) bio *Forum Administrator*

**Date** Reply #3 on Sat 20 Aug 2016 09:35 PM (UTC)

**Message** I got an email from a user who said that using the SN75176 transceiver chip caused problems with his circuit that did not occur with other types of transceivers, in particular if he used more than 3 slaves.

If you are having problems, and you are using that chip, you may want to try a more modern transceiver.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

The dates and times for posts above are shown in Universal Co-ordinated Time (UTC).

To show them in your local time you can join the forum, and then set the 'time correction' field in your profile to the number of hours difference between your location and UTC time.

124,022 views.

**Postings by administrators only.**

Refresh page

Go to topic: (Choose topic) Go Search the forum

top

*Quick links:* **MUSHclient**. MUSHclient **help**. Forum **shortcuts**. Posting **templates**. Lua **modules**. Lua **documentation**.

Home

Designed & written by Nick Gammon

Nick Gammon 27k ●6 ●51 ●159

Comments to: Gammon Software support
XML Forum RSS feed ( https://gammon.com.au/rss/forum.xml )

Best Viewed With AnyBrowser FutureQuest