Pot-Switch XpressNet Throttle.

December 2 2022.

John A. Stewart

Introduction

This document describes the XpressNet hand-held wired throttle originally designed and produced in 2010.

This is a very simple throttle, designed with the following thoughts in mind;
- no batteries!
- Tactile - no need to look at it to control a locomotive;
- one throttle tied to one locomotive or consist - no issue with new operators pushing the wrong button and acquiring someone else's locomotive.

Code was placed on the Yahoo Group DIYDCC group, but when Yahoo closed its' group database, the code may have been removed or moved somewhere else.

This is simple code, that implements a fair amount of the XpressNet protocol. This protocol is available on-line; look at the https://JMRI.org website, they point to a valid, current location of the XpressNet Protocol Document.

Why the name? It contains a **potentiometer** and a **switch** as tactile controls, thus "Pot-Switch".

Now, while my example code only affects speed/direction, and headlight, you can add other switches, or automatic actions (like the on/off of headlight in this code), by doing some more Arduino programming to perform the action and to send it to the XpressNet Command Station.

Arduinos and XpressNet

This document assumes you understand the basics of Arduino hardware and programming, soldering and wiring, and the Lenz XpressNet connections.

If not, look at https://arduino.cc, and https://dccwiki.com/XpressNet.

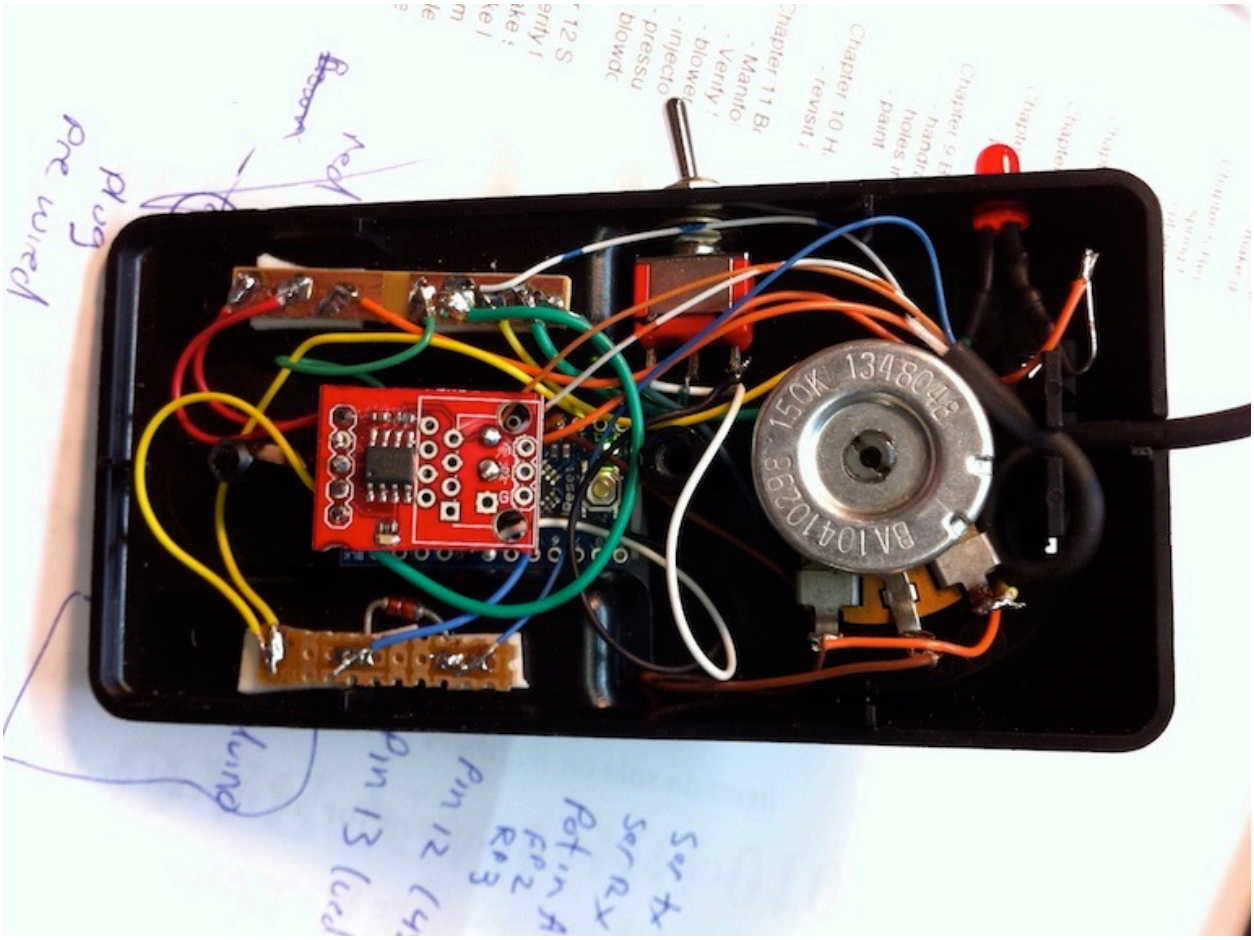
A simple Throttle - hardware.

My hardware of choice makes for a small, comfortable to hold throttle. I have attached two photos here of a completed throttle, and a description of the main bits and pieces.

Some of what I purchased way back when is not available but that does not matter - whatever you can get your hands on should work just fine.

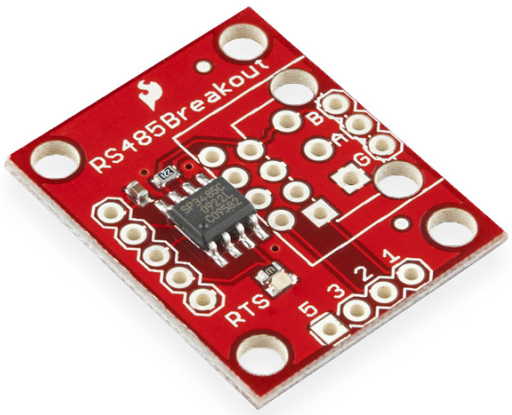
I wired mine up with bits of wire from the scrap box, and used bits of circuit boards that were produced for prototyping circuits “back in the day” for little bus-bars, but anything (such as some copper house wire scraps) could be used if the wires are wrapped around before soldering.

These throttles have lasted 12 years so far, no issues, so the design and build is good enough, I think.

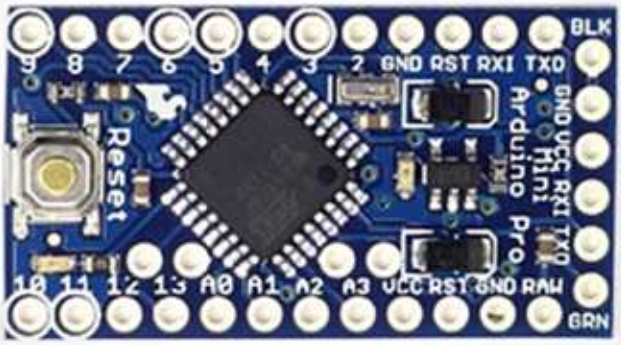


There is a little Arduino mounted inside. I used little “Arudino Pro Mini” 5 volt boards. At the time of writing, these are listed in the “Retired Products” in the arduino.cc website. However, they are still listed on the sparkfun.com website, part ID DEV-11113 for a 5V16MHz board.

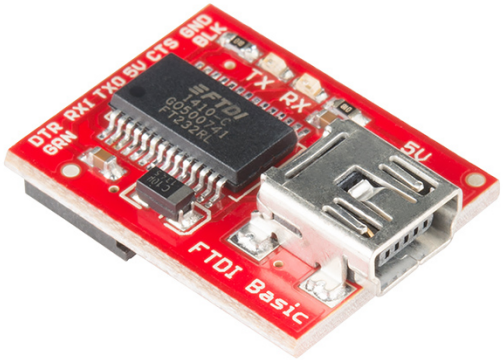
The little red board is a MAX485 interface board; it transfers data between the XpressNet RS-485 network, and the Arduino. The Sparkfun Part ID I used is BOB-10124



The Arduino Pro Mini does not have a USB connector. To program it, one is required.



I used a “FTDI Basic Breakout” board, that connects a USB cable to the pins on the Arduino Pro Mini. One of these boards (the MAX485 or the FTDI board) are used one at a time; one for programming, one for running.



How did I do this?

I put some typical “Arduino” male pins on the Arduino Pro Mini - one of the narrow ends is for serial connections. If you look at the right hand end of the Arduino in the picture here, you’ll see a row of pins on the right, with the label “BLK” on the top and “GRN” on the bottom. Same with the **FTDI Basic board**, and, although harder to see, with the **MAX485**.

FTDI Pin	Arduino Pin	MAX485 pin
GND	“BLK” (GND)	no pin
CTS	GND	wired elsewhere
5V VCC	VCC	wired elsewhere
TXO	RXI	TX0
RXI	TX0	RXI
DTR	“GRN” (DTR)	wired elsewhere

The **FTDI header** pinout works, if you look at the above table. You can plug in the FTDI into the Arduino, or the MAX485 into the Arduino. The FTDI has a male 6-pin socket on it, which matches the Arduino (which it is designed for), and once programmed, remove the FTDI and plug in the MAX485 with only 2 pin male socket soldered on, to match the TXO and RXI pins. The other wires on the MAX485 go elsewhere - more below.

COMMENT: Each Arduino board design is different. The Arduino used above was the smallest one available when I wrote this code, and the use of the FTDI board made the design and build quite small.

I do use Arduino **MAX 2560** boards for other XpressNet projects, and these have 4 serial ports and a built in USB connector, so do your research and see how you can program an Arduino, then use a serial port to send data over the XpressNet bus.

Throttle - Component Inter-connect Wiring

This section talks about the wiring of components, whether using the physical design above, or of the design from your own hands.

In the picture of the inside of the throttle, you'll see the **SPDT switch** and the **LED** on one side. The LED has a 10k Ohm resistor soldered to a lead, and black heat-shrink tubing applied. These go to digital pins on the Arduino. There are many examples in the Arduino documentation showing how to wire switches and LEDs, so if you need assistance, have a look at the examples.

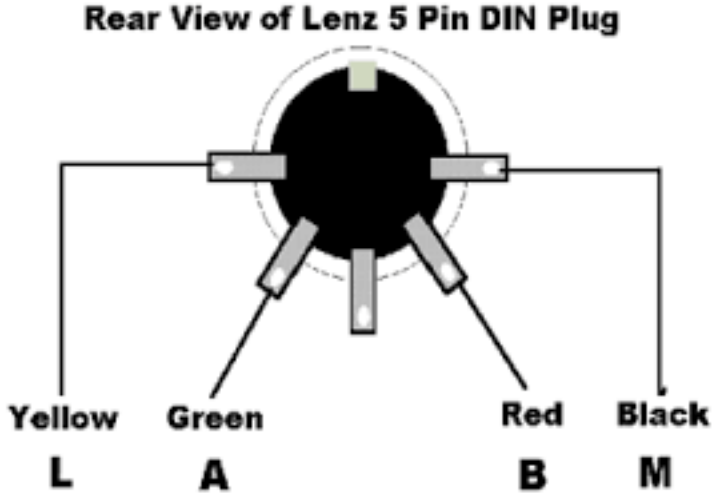
The **Potentiometer** is wired in typical Arduino fashion; one lead to ground, one to +5v, and the middle lead (the "wiper") to an Analogue input pin.

The **12V** from **XpressNet** comes in on the 4-wire cable, the power wire (blue in the picture) goes to the right side of the bit of PC board on the bottom, then through a diode (visible in the picture) to an "XpressNet Positive" bus on the other half of that bit of PC board. This then goes to the Arduino, which has a voltage regulator on-board to reduce this **12V down to 5V** for use.

The **Ground** from the XpressNet goes to the other bit of PC board (the right half). The other (left) half is a **+5V bus**, from the output of the Arduino voltage regulator.

XpressNet Cabling - the Lenz documentation for their command stations has pinouts for different format of plugs for their XpressNet bus. I use the **5-pin DIN** plugs as they are strong and easy to plug in. If you are looking for pre-wired cables, 5-pin **MIDI** cables are the same, and it may be cheaper to pick up a 5-pin MIDI cable from a music store and cut one connector off rather than purchasing connectors and wire.

Here's a picture of the Lenz 5-Pin DIN plug wiring, found on-line, likely from a scan of Lenz documentation.



Computer Code

The Arduino computer code is meant to be fairly easy to read. There are some sections that can be explained here in more detail than in the code itself.

Looping based on waiting for Serial Data

We wait for Commands coming in from the XpressNet Bus, and then if a Command is one we can process, we act on it, and, if required, send data to the XpressNet Bus in a timely manner.

Thus:

```
//////////////////////////////////////////////////////////////////
void loop() {

  // get the byte from the XPressnet bus, if there is one.
  int rxdata = USART_Receive();
```

we wait for data from the XpressNet bus. As long as valid data (parity, etc) is received from the XpressNet bus, we then process it.

9-bit Serial Data

XpressNet is 9-bit data; it's an old way of sending data serially on multi-drop implementations (e.g. RS-485), where a "command bit" is desired to show timing or some form of superiority. You'll see this in older products, such as "Raymarine Seatalk V1" on boats, or on Lab equipment, or… It works very well in certain circumstances, and XpressNet is one.

Setting 9-bit serial is interesting. I ended up going through the Arduino CPU device manual and coding this in. I did a quick read, and did not see an easier way using standard Arduino calls, but I may have missed something.

No matter what, this code, although written back in 2009-2010, still works today.

The easy part:

```
//////////////////////////////////////////////////////////////////
void USART_Transmit (unsigned char data8) {
  /* wait for empty transmit buffer */
  while (!(UCSRA_PORT & (1<<UDRE_PORT))) {}
  /* put the data into buffer, and send */

 UDR_PORT = data8;
}
```

The code in upper case (e.g. “UDR_PORT”) works directly with the serial data hardware in the Arduino. The ATMega data sheet shows how to do this, but it is a long and technical document, so read it only if you can’t fall asleep, and just use this code.

The harder part:

Setting the data port speed/parity/ and defining the macros.

Some of the Arduino chips use one form, others (with maybe more than 1 serial port) use another.

```
// Which serial port is used, if we have more than one on the chip?
//
// ATmega8, maybe others will NOT have UCSR0A defined, so we just use
// this define to see what the serial port defines are.
//
#ifndef UCSR0A
  // definitely only one serial port here.

  // and, do the bit-twiddling ports for this one-serial-port Arduino.
  #define UCSRA_PORT UCSRA
…
```

looks at compile time for a “compiler define” for one style, if not, it uses the other style.

```
  //
  // ATMega368, ATMega1280, ATMega2560...
  // we can put the Xpressnet bus config on another serial port

  #define UCSRA_PORT UCSR## 0 ##A
```

In the first example above, `UCSRA_PORT` is defined to `UCSRA`, in the second example, it is defined as `UCSR0A`.

In the second example, the serial port used is port 0; in the first example, it is the only port available on the board - also port 0, but the actual hardware commands are different.

Note that you can also use a different serial port - say port 3 - if you want to use Serial 0 to send info out to the Arduino monitor - in this case, change all of the #defines to use the port you wish.

Commands from the Command Station

The Lenz command station gets the attention of throttles by setting the 9th bit to 1. This makes the serial input data to be greater than 255, or 0xFF in Hexadecimal.

```
//////////////////////////////////////////////////////////
// XPressnet Call Bytes.
//
#define CALL_BYTE 0x100
```

```
// broadcast to everyone, we save the incoming data and process it
later.
#define GENERAL_BROADCAST 0x160
```

then in the code, at the top of the `loop()` function, you'll see:

```
//////////////////////////////////////////////////////////////////
void loop() {

  // get the byte from the XPressnet bus, if there is one.
  int rxdata = USART_Receive();

  // is the data ok?
  if (rxdata != -1) {

     // This IS a Call Byte
     if (rxdata >= CALL_BYTE) {
```

Anything greater than 256 is a command from on high, and we look at it in that frame of mind.

Some of these commands may be broadcasts to everyone (e.g. "*emergency stop button pushed*"),

Some may directed to one throttle in particular (e.g. "*what's your speed??*" or "*Any lights changed??*")

Finally, a call may be "you still there?"

The XpressNet technical documentation has the details of these interactions, if you want to delve deeper into it.

Reading Pot and switch, setting LED

As mentioned above, the sequencing of the throttle is determined by the received data. No data, no processing.

If we do have a valid XpressNet connection, and data is flowing, then we can read the Potentiometer and the SPDT switch, and set the LED status.

For instance, if the track is in emergency stop, the following code will be executed:

```
 if (currentState != STATE_NORMAL_OPS) {
   // let the command station give us our clock; we count
   // "ticks" here.
   currentFlashCounter++;
   if (currentFlashCounter >= flashInterval) {
     currentFlashCounter = 0;

     // flip the state.
     displayLEDState = !displayLEDState;
```

```
    if (displayLEDState) {LED_ON;} else {LED_OFF;}
```

which blinks the LED, the flash rate comes from the number of XpressNet Command Station commands received; after a certain number ("`flashInterval`") the LED will be toggled.

Same with reading the Potentiometer and SPDT switch - currently the code uses every 20 "ticks"

```
} else if (analogueReadCounter == 30) {
   // read the speed for speed changes
   analogueReadCounter = 0;
   potValue = map(analogRead(POT_PIN),0,1023,0,decoderSpeedSteps);
  …
```

Handling Errors from XpressNet

Errors happen - sometimes they are intermittent, other times, they are "fatal" and need to be resolved before rebooting this throttle.

Rebooting happens every time you unplug and plug in the throttle.

The function:

```
//////////////////////////////////////////////////////////////////
// on a catastrophic error, loop until reset, flashing...
void loopErrorToLED(int pattern) {
  int i;

  for (;;) {
    for (i=0; i<pattern; i++) {
      LED_ON;
      delay (200);
      LED_OFF;
      delay (200);
    }
    delay (2000);
  }
}
```

is called by the code, with an integer parameter, which is the "flash" code. Here is what the LED tells us:

Solid on: Everything ok
Slow 50/50 flashing: Track power off
1 quick flash: Locomotive part of a double header
2 quick flashes: Locomotive not free - MU, not single locomotive
3 quick flashes: Locomotive not free - part of a MU.
4 quick flashes: Speed steps not supported - not 14,27,28,128.
5 quick flashes: Command station not XPressNet 3.0 or higher

6 quick flashes: internal error - Speed step not supported, but should have been caught before here.
7 quick flashes: locomotive used by someone else

Conclusion

I hope this code, and this documentation is useful. Feedback welcome.