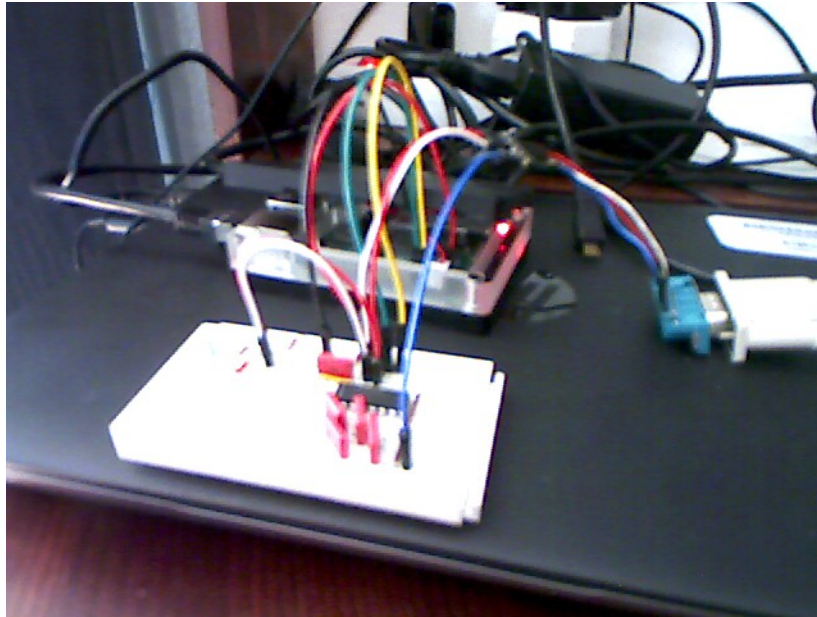















RPI

<https://www.raspberrypi.org/documentation/usage/gpio/>



<https://pi4j.com/1.2/pins/model-b-rev2.html>

RPI 2B

| Raspberry Pi Model A & B (P1 Header) | | | | | |
|--------------------------------------|---------------|----|---|------|----------------------|
| PIN # | NAME | | | NAME | PIN # |
| | 3.3 VDC Power | 1 |  | 2 | 5.0 VDC Power |
| 8 | SDA0 (I2C) | 3 |  | 4 | DNC |
| 9 | SCL0 (I2C) | 5 |  | 6 | 0V (Ground) |
| 7 | GPIO 7 | 7 |  | 8 | TxD (UART) 15 |
| | DNC | 9 |  | 10 | RxD (UART) 16 |
| 0 | GPIO 0 | 11 |  | 12 | GPIO1 1 |
| 2 | GPIO2 | 13 |  | 14 | DNC |
| 3 | GPIO3 | 15 |  | 16 | GPIO4 4 |
| | DNC | 17 |  | 18 | GPIO5 5 |
| 12 | MOSI | 19 |  | 20 | DNC |
| 13 | MISO | 21 |  | 22 | GPIO6 6 |
| 14 | SCLK | 23 |  | 24 | CE0 10 |
| | DNC | 25 |  | 26 | CE1 11 |

Attention! The GPIO pin numbering used in this diagram is intended for use with WiringPi / Pi4J. This pin numbering is not the raw Broadcom GPIO pin numbers.

<http://www.pi4j.com>

RPI 4B

<https://spellfoundry.com/2016/05/29/configuring-gpio-serial-port-raspbian-jessie-including-pi-3-4/>

Raspberry Pi 4 B J8 GPIO Header

| Pin# | NAME | | NAME | Pin# |
|------|---------------------------------|--|---------------------------------|------|
| 01 | 3.3v DC Power | | DC Power 5v | 02 |
| 03 | GPIO02 (SDA1, I ² C) | | DC Power 5v | 04 |
| 05 | GPIO03 (SCL1, I ² C) | | Ground | 06 |
| 07 | GPIO04 (GPCLK0) | | (TXD0, UART) GPIO14 | 08 |
| 09 | Ground | | (RXD0, UART) GPIO15 | 10 |
| 11 | GPIO17 | | (PWM0) GPIO18 | 12 |
| 13 | GPIO27 | | Ground | 14 |
| 15 | GPIO22 | | GPIO23 | 16 |
| 17 | 3.3v DC Power | | GPIO24 | 18 |
| 19 | GPIO10 (SPI0_MOSI) | | Ground | 20 |
| 21 | GPIO09 (SPI0_MISO) | | GPIO25 | 22 |
| 23 | GPIO11 (SPI0_CLK) | | (SPI0_CE0_N) GPIO08 | 24 |
| 25 | Ground | | (SPI0_CE1_N) GPIO07 | 26 |
| 27 | GPIO00 (SDA0, I ² C) | | (SCL0, I ² C) GPIO01 | 28 |
| 29 | GPIO05 | | Ground | 30 |
| 31 | GPIO06 | | (PWM0) GPIO12 | 32 |
| 33 | GPIO13 (PWM1) | | Ground | 34 |
| 35 | GPIO19 | | GPIO16 | 36 |
| 37 | GPIO26 | | GPIO20 | 38 |
| 39 | Ground | | GPIO21 | 40 |

Raspberry Pi 4 B J14 PoE Header

| | | | | |
|----|------|--|------|----|
| 01 | TR01 | | TR00 | 02 |
| 03 | TR03 | | TR02 | 04 |

Pinout Grouping Legend

| | | | |
|-------------------------------------|--|--|---|
| Inter-Integrated Circuit Serial Bus | | | Serial Peripheral Interface Bus |
| Ungrouped/Un-Allocated GPIO | | | Universal Asynchronous Receiver-Transmitter |
| Reserved for EEPROM | | | |

| Pin | GPIO / Function | Default Resistor State | ALT0 | ALT1 | ALT2 | ALT3 | ALT4 | ALT5 |
|-----|-----------------|------------------------|------------|-------------|-----------|----------------|------------|------------|
| 1 | 3v3 | | | | | | | |
| 2 | 5v | | | | | | | |
| 3 | 2 | UP | SDA1 | SA3 | LCD_VSYNC | SPI3_MOSI | CTS2 | SDA3 |
| 4 | 5v | | | | | | | |
| 5 | 3 | UP | SCL1 | SA2 | LCD_HSYNC | SPI3_SCLK | RTS2 | SCL3 |
| 6 | GROUND | | | | | | | |
| 7 | 4 | UP | GPCLK0 | SA1 | DPI_D0 | SPI4_CE0_N | TXD3 | SDA3 |
| 8 | 14 | DOWN | TXD0 | SD6 | DPI_D10 | SPI5_MOSI | CTS5 | TXD1 |
| 9 | GROUND | | | | | | | |
| 10 | 15 | DOWN | RXD0 | SD7 | DPI_D11 | SPI5_SCLK | RTS5 | RXD1 |
| 11 | 17 | DOWN | FL1 | SD9 | DPI_D13 | RTS0 | SPI1_CE1_N | RTS1 |
| 12 | 18 | DOWN | PCM_CLK | SD10 | DPI_D14 | SPI6_CE0_N | SPI1_CE0_N | PWM0 |
| 13 | 27 | DOWN | SD0_DAT3 | TE1 | DPI_D23 | SD1_DAT3 | ARM_TMS | SPI6_CE1_N |
| 14 | GROUND | | | | | | | |
| 15 | 22 | DOWN | SD0_CLK | SD14 | DPI_D18 | SD1_CLK | ARM_TRST | SDA6 |
| 16 | 23 | DOWN | SD0_CMD | SD15 | DPI_D19 | SD1_CMD | ARM_RTCK | SCL6 |
| 17 | 3v3 | | | | | | | |
| 18 | 24 | DOWN | SD0_DAT0 | SD16 | DPI_D20 | SD1_DAT0 | ARM_TDO | SPI3_CE1_N |
| 19 | 10 | DOWN | SPI0_MOSI | SD2 | DPI_D6 | I2CSL_SDA_MOSI | CTS4 | SDA5 |
| 20 | GROUND | | | | | | | |
| 21 | 9 | DOWN | SPI0_MISO | SD1 | DPI_D5 | I2CSL_SDI_MISO | RXD4 | SCL4 |
| 22 | 25 | DOWN | SD0_DAT1 | SD17 | DPI_D21 | SD1_DAT1 | ARM_TCK | SPI4_CE1_N |
| 23 | 11 | DOWN | SPI0_SCLK | SD3 | DPI_D7 | I2CSL_SCL_SCLK | RTS4 | SCL5 |
| 24 | 8 | UP | SPI0_CE0_N | SD0 | DPI_D4 | I2CSL_CE_N | TXD4 | SDA4 |
| 25 | GROUND | | | | | | | |
| 26 | 7 | UP | SPI0_CE1_N | SWE_N_SRW_N | DPI_D3 | SPI4_SCLK | RTS3 | SCL4 |
| 27 | 0 | UP | SDA0 | SA5 | PCLK | SPI3_CE0_N | TXD2 | SDA6 |
| 28 | 1 | UP | SCL0 | SA4 | DE | SPI3_MISO | RXD2 | SCL6 |
| 29 | 5 | UP | GPCLK1 | SA0 | DPI_D1 | SPI4_MISO | RXD3 | SCL3 |
| 30 | GROUND | | | | | | | |
| 31 | 6 | UP | GPCLK2 | SOE_N_SE | DPI_D2 | SPI4_MOSI | CTS3 | SDA4 |
| 32 | 12 | DOWN | PWM0 | SD4 | DPI_D8 | SPI5_CE0_N | TXD5 | SDA5 |
| 33 | 13 | DOWN | PWM1 | SD5 | DPI_D9 | SPI5_MISO | RXD5 | SCL5 |
| 34 | GROUND | | | | | | | |
| 35 | 19 | DOWN | PCM_FS | SD11 | DPI_D15 | SPI6_MISO | SPI1_MISO | PWM1 |
| 36 | 16 | DOWN | FL0 | SD8 | DPI_D12 | CTS0 | SPI1_CE2_N | CTS1 |
| 37 | 26 | DOWN | SD0_DAT2 | TE0 | DPI_D22 | SD1_DAT2 | ARM_TDI | SPI5_CE1_N |
| 38 | 20 | DOWN | PCM_DIN | SD12 | DPI_D16 | SPI6_MOSI | SPI1_MOSI | GPCLK0 |
| 39 | GROUND | | | | | | | |
| 40 | 21 | DOWN | PCM_DOUT | SD13 | DPI_D17 | SPI6_SCLK | SPI1_SCLK | GPCLK1 |

<http://0pointer.de/blog/projects/serial-console.html>

```
root@raspberrypi:/home/pi# systemctl enable serial-getty@ttyAMA0.service
The unit files have no installation config (WantedBy=, RequiredBy=, Also=,
Alias= settings in the [Install] section, and DefaultInstance= for template
units). This means they are not meant to be enabled using systemctl.
```

Possible reasons for having this kind of units are:

- A unit may be statically enabled by being symlinked from another unit's .wants/ or .requires/ directory.
- A unit's purpose may be to act as a helper for some other unit which has a requirement dependency on it.
- A unit may be started when needed via activation (socket, path, timer, D-Bus, udev, scripted systemctl call, ...).
- In case of template units, the unit is meant to be enabled with some

instance name specified.

Systemd Enable Serial Port on RPI

```
root@raspberrypi:/home/pi# systemctl start serial-getty@ttyAMA0.service

root@raspberrypi:/home/pi# cat /boot/cmdline.txt
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 console=tty1 root=/dev/nfs
rootfstype=nfs nfsroot=192.168.40.40:/opt/remote/nfsroot/raspberry2,tcp,vers=4
smsc95xx.turbo_mode=N ip=dhcp elevator=deadline fsck.repair=no rootwait

# systemctl enable serial-getty@ttyAMA0.service
# systemctl start serial-getty@ttyAMA0.service

# cp /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/serial-
getty@ttyAMA0.service
# vi /etc/systemd/system/serial-getty@ttyAMA0.service
.... now make your changes to theagetty command line ...
# ln -s /etc/systemd/system/serial-getty@ttyAMA0.service
/etc/systemd/system/getty.target.wants/
# systemctl daemon-reload
# systemctl start serial-getty@ttyAMA0.service
```

```
pi@r+-----+
pi@r| A -      Serial Device          : /dev/ttyS1
pi@r| B - Lockfile Location           : /var/lock
pi@r| C -      Callin Program           :
pi@r| D -      Callout Program          :
pi@r| E -      Bps/Par/Bits              : 115200 8N1
pi@r| F - Hardware Flow Control         : No
pi@r| G - Software Flow Control         : No
pi@r|
pi@r|      Change which setting?
pi@r+-----+
```

<https://spellfoundry.com/2016/05/29/configuring-gpio-serial-port-raspbian-jessie-including-pi-3-4/>
https://elinux.org/RPi_Serial_Connection

<https://www.rogerirwin.co.nz/linux-open-source/enabling-a-serial-port-console/>

Using the serial port to login is a good way to get access to a device that doesn't have a network connection. This could be a small embedded Linux computer.

Raspberry Pi

```
enable_uart=1
```

On a Raspberry Pi you can edit `/boot/config.txt` and add this at the bottom. Then reboot. More details can be found [here](#).

Linux devices with systemd

```
sudo nano /lib/systemd/system/serial-getty@.service
```

First we need to edit the serial-getty service to set the correct baud rate.

```
# Edit this line
ExecStart=-/sbin/agetty --keep-baud 115200,38400,9600 %I $TERM
# To This
ExecStart=-/sbin/agetty 115200 %I $TERM
```

Set this to the baud rate that you wish to use. This should be the same as the computer is using at the other end.

```
systemctl daemon-reload
# For a USB serial adaptor
systemctl enable serial-getty@ttyUSB0.service
# For a built in serial port /dev/ttyS0
systemctl enable serial-getty@ttyS0.service
```

You can now log in using the serial console on `/dev/ttyUSB0` or `/dev/ttyS0`

1. Connect another computer to it using a NULL modem cable.
2. Then open a terminal emulator such as minicom.
3. Set the baud rate and port.
4. Press enter a few times and you should then see a login prompt.

[systemd for Administrators, Part XVI](#)

Gettys on Serial Consoles (and Elsewhere)

TL;DR: To make use of a serial console, just use `console=ttyS0` on the kernel command line, and systemd will automatically start a getty on it for you.

While physical [RS232](#) serial ports have become exotic in today's PCs they play an important role in modern servers and embedded hardware. They provide a relatively robust and minimalistic way to access the console of your device, that works even when the network is hosed, or the primary UI is unresponsive. VMs frequently emulate a serial port as well.

Of course, Linux has always had good support for serial consoles, but with [systemd](#) we tried to make serial console support even simpler to use. In the following text I'll try to give an overview how serial console [gettys](#) on systemd work, and how TTYs of any kind are handled.

Let's start with the key take-away: in most cases, to get a login prompt on your serial prompt you don't need to do anything. systemd checks the kernel configuration for the selected kernel console

and will simply spawn a serial getty on it. That way it is entirely sufficient to configure your kernel console properly (for example, by adding `console=ttyS0` to the kernel command line) and that's it. But let's have a look at the details:

In systemd, two template units are responsible for bringing up a login prompt on text consoles:

1. `getty@.service` is responsible for [virtual terminal](#) (VT) login prompts, i.e. those on your VGA screen as exposed in `/dev/tty1` and similar devices.
2. `serial-getty@.service` is responsible for all other terminals, including serial ports such as `/dev/ttyS0`. It differs in a couple of ways from `getty@.service`: among other things the `$TERM` environment variable is set to `vt102` (hopefully a good default for most serial terminals) rather than `linux` (which is the right choice for VTs only), and a special logic that clears the VT scrollback buffer (and only work on VTs) is skipped.

Virtual Terminals

Let's have a closer look how `getty@.service` is started, i.e. how login prompts on the virtual terminal (i.e. non-serial TTYs) work. Traditionally, the init system on Linux machines was configured to spawn a fixed number login prompts at boot. In most cases six instances of the getty program were spawned, on the first six VTs, `tty1` to `tty6`.

In a systemd world we made this more dynamic: in order to make things more efficient login prompts are now started on demand only. As you switch to the VTs the getty service is instantiated to `getty@tty2.service`, `getty@tty5.service` and so on. Since we don't have to unconditionally start the getty processes anymore this allows us to save a bit of resources, and makes start-up a bit faster. This behaviour is mostly transparent to the user: if the user activates a VT the getty is started right-away, so that the user will hardly notice that it wasn't running all the time. If he then logs in and types `ps` he'll notice however that getty instances are only running for the VTs he so far switched to.

By default this automatic spawning is done for the VTs up to VT6 only (in order to be close to the traditional default configuration of Linux systems)^[1]. Note that the auto-spawning of gettys is only attempted if no other subsystem took possession of the VTs yet. More specifically, if a user makes frequent use of [fast user switching](#) via GNOME he'll get his X sessions on the first six VTs, too, since the lowest available VT is allocated for each session.

Two VTs are handled specially by the auto-spawning logic: firstly `tty1` gets special treatment: if we boot into graphical mode the display manager takes possession of this VT. If we boot into multi-user (text) mode a getty is started on it -- unconditionally, without any on-demand logic^[2].

Secondly, `tty6` is especially reserved for auto-spawned gettys and unavailable to other subsystems such as X^[3]. This is done in order to ensure that there's always a way to get a text login, even if due to fast user switching X took possession of more than 5 VTs.

Serial Terminals

Handling of login prompts on serial terminals (and all other kind of non-VT terminals) is different from that of VTs. By default systemd will instantiate one `serial-getty@.service` on the main kernel^[4] console, if it is not a virtual terminal. The kernel console is where the kernel outputs its own log messages and is usually configured on the kernel command line in the boot loader via an argument such as `console=ttyS0`^[5]. This logic ensures that when the user asks the kernel to redirect its output onto a certain serial terminal, he will automatically also get a login prompt on it as the boot completes^[6]. systemd will also spawn a login prompt on the first special VM console

(that's `/dev/hvc0`, `/dev/xvc0`, `/dev/hvsi0`), if the system is run in a VM that provides these devices. This logic is implemented in a [generator](#) called `systemd-getty-generator` that is run early at boot and pulls in the necessary services depending on the execution environment.

In many cases, this automatic logic should already suffice to get you a login prompt when you need one, without any specific configuration of `systemd`. However, sometimes there's the need to manually configure a serial getty, for example, if more than one serial login prompt is needed or the kernel console should be redirected to a different terminal than the login prompt. To facilitate this it is sufficient to instantiate `serial-getty@.service` once for each serial port you want it to run on^[7]:

```
# systemctl enable serial-getty@ttyS2.service
# systemctl start serial-getty@ttyS2.service
```

And that's it. This will make sure you get the login prompt on the chosen port on all subsequent boots, and starts it right-away too.

Sometimes, there's the need to configure the login prompt in even more detail. For example, if the default baud rate configured by the kernel is not correct or other `agetty` parameters need to be changed. In such a case simply copy the default unit template to `/etc/systemd/system` and edit it there:

```
# cp /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/serial-
getty@ttyS2.service
# vi /etc/systemd/system/serial-getty@ttyS2.service
.... now make your changes to the agetty command line ...
# ln -s /etc/systemd/system/serial-getty@ttyS2.service
/etc/systemd/system/getty.target.wants/
# systemctl daemon-reload
# systemctl start serial-getty@ttyS2.service
```

This creates a unit file that is specific to serial port `ttyS2`, so that you can make specific changes to this port and this port only.

And this is pretty much all there's to say about serial ports, VTs and login prompts on them. I hope this was interesting, and please come back soon for the next installment of this series!

Footnotes

[1] You can easily modify this by changing `NAutoVTs=` in [logind.conf](#).

[2] Note that whether the getty on VT1 is started on-demand or not hardly makes a difference, since VT1 is the default active VT anyway, so the demand is there anyway at boot.

[3] You can easily change this special reserved VT by modifying `ReserveVT=` in [logind.conf](#).

[4] If multiple kernel consoles are used simultaneously, the *main* console is the one listed *first* in `/sys/class/tty/console/active`, which is the *last* one listed on the kernel command line.

[5] See [kernel-parameters.txt](#) for more information on this kernel command line option.

[6] Note that `agetty -s` is used here so that the baud rate configured at the kernel command line is not altered and continued to be used by the login prompt.

[7] Note that this `systemctl enable` syntax only works with `systemd` 188 and newer (i.e. F18). On older versions use `ln -s /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/getty.target.wants/serial-getty@ttyS2.service ; systemctl daemon-reload` instead.

LED Indicators on the UART RX/TX

<https://forums.parallax.com/discussion/153140/tx-rx-activity-light-circuit>

I see there are some basic requirements:

1. The circuit must not interfere with normal operation of the serial port.
2. I usually run at 115200 baud.
3. I want to at least be able to see the LED flash with the worst case character. In this case sending 8 1's. So one sees only the start bit. Of course the LED is brighter with characters that have more 0's.

1. The gain of my **2N3906** PNP transistors is about 180. With a base resistor of 10K Ω the current through the LED is self limited so the 330 Ω resistor is not required. Better yet any color LED from **RED** to **BLUE** work equally well.

2. The start bit is about 9 μ s wide. If the capacitor is to large it affects the fall time which may cause bit errors. I find that 10nF is about right.

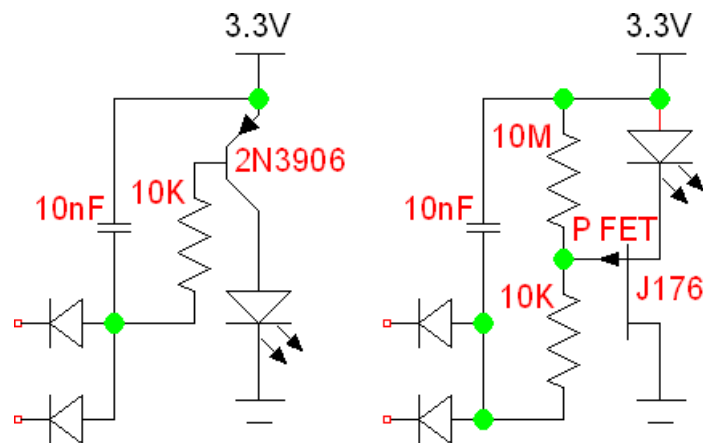
If you want to operate at a slower bit rate the capacitor can be proportionately larger.

3. There is no need for the 100 Ω input resistor. In fact it just limits the charge on the capacitor dimming the LED no matter what value I tried.

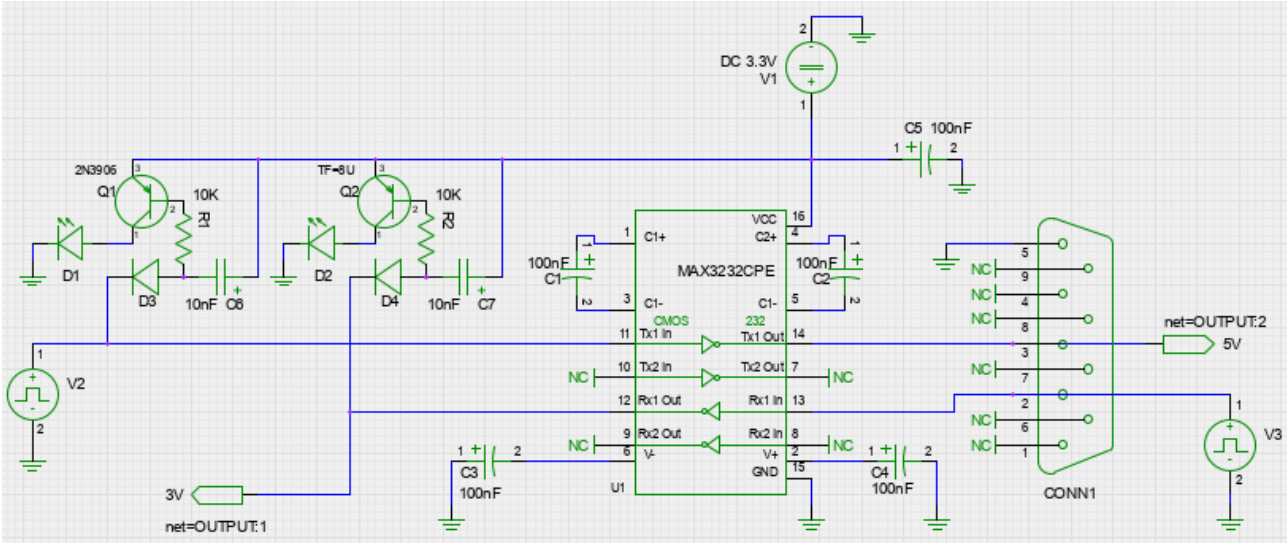
So my conclusion is to just use the 4 parts:

2N3906, 10nF, 10K Ω , and the **1N4148** signal diode.

Have fun with this.



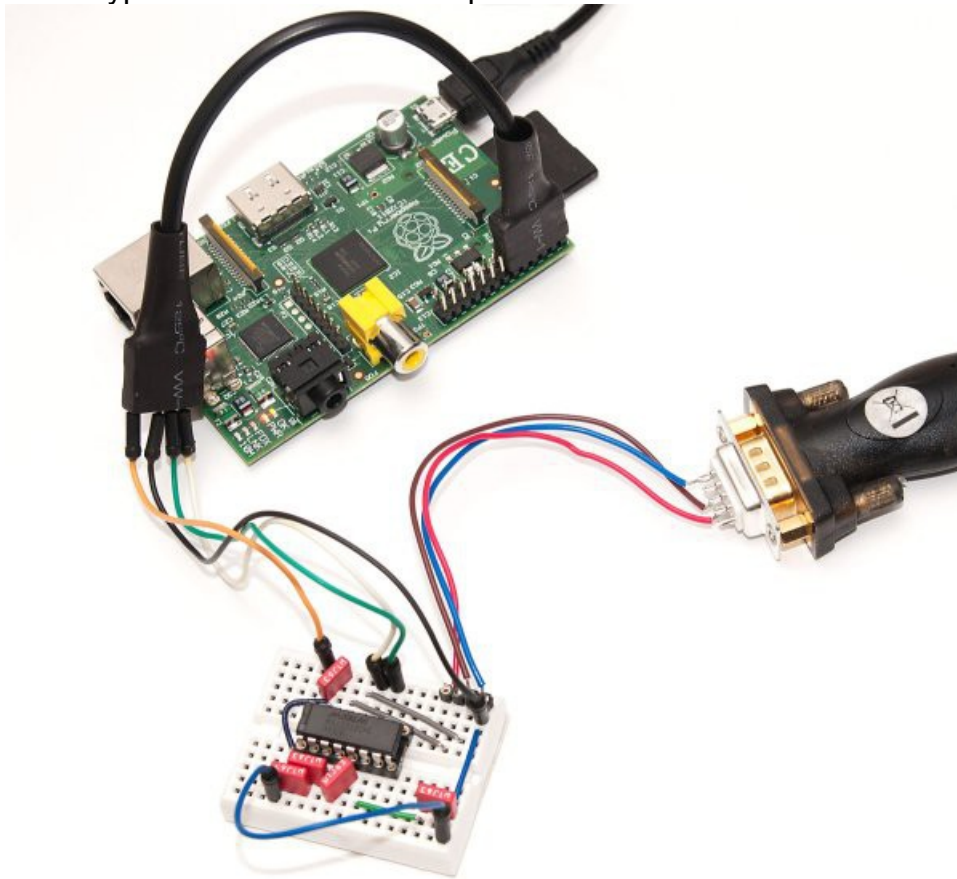
Final



<https://projects-raspberry.com/raspberry-pi-serial-console-with-max3232cpe/>

So in this short tutorial, I'll show you how to use a MAX3232CPE transceiver to safely convert the normal UART voltage levels to 3.3V accepted by Raspberry Pi, and connect to the Pi using Putty. This is what you'll need:

- Raspberry Pi unit
- Serial port in your PC or USB to serial -adapter
- MAX3232CPE or similar RS-232 to 3.3V logic level transceiver
- 5 x 0.1 uF capacitors (I used plastic ones)
- Jumper wires and breadboard
- Some type of female-female adapter



The last item is needed to connect male-male jumper wires to RaspPi GPIO pins. I had a short 2x6 pin extension cable available and used that, but an IDE cable and other types ribbon cable work fine as well. Just make sure it doesn't internally short any of the connections – use a multimeter if in doubt!

The connections on Pi side are rather straightforward. We'll use the 3.3V pin for power – the draw should not exceed 50 mA, but this should not be an issue, since MAX3232CPE draws less than 1 mA and the capacitors are rather small. GND is also needed, and the two UART pins, TXD and RXD.

Using MAX3232CPE for 3.3V UART

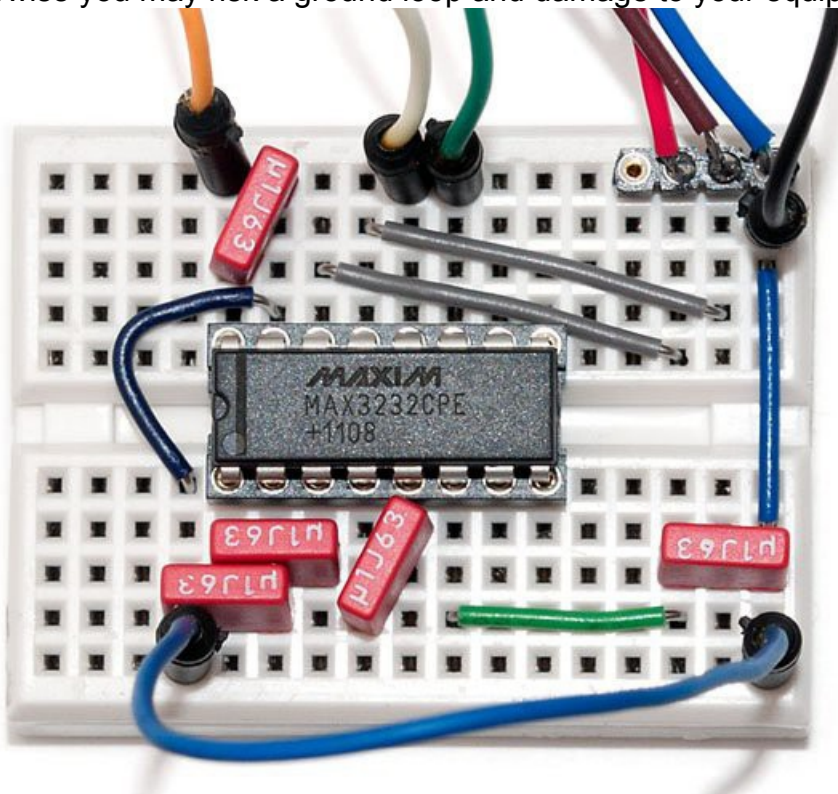
The MAX3232CPE is very much like it's 5V sister model, MAX232. It uses a few capacitors to deliver true +-12V RS-232 signalling on one end, and 3.3V signalling on the

other. I'm not going to cover the internals in detail this time, please either refer to [the datasheet](#) or my [previous tutorial](#) discussing this same chip.

Above you can see one rather compact way to wire the MAX3232. The orange, white, green and black wires come from Raspberry Pi and provide power and data lines. The red, brown and blue wires go to the RS-232 port – see the illustration on right for connections on this side.

Update: The RS-232 connection diagram is from the side you'd solder the wires from ("back side" of the connector), and wired so you can connect a PC USB serial adapter to Raspberry Pi. If you'd like to talk to serial peripherals from Pi instead, RX/TX wires need to be reversed.

Before you connect the Pi, check with a voltmeter that GND from Raspberry Pi and GND from RS-232 do not differ from each other too much (a few millivolts is usually OK), otherwise you may risk a ground loop and damage to your equipment!



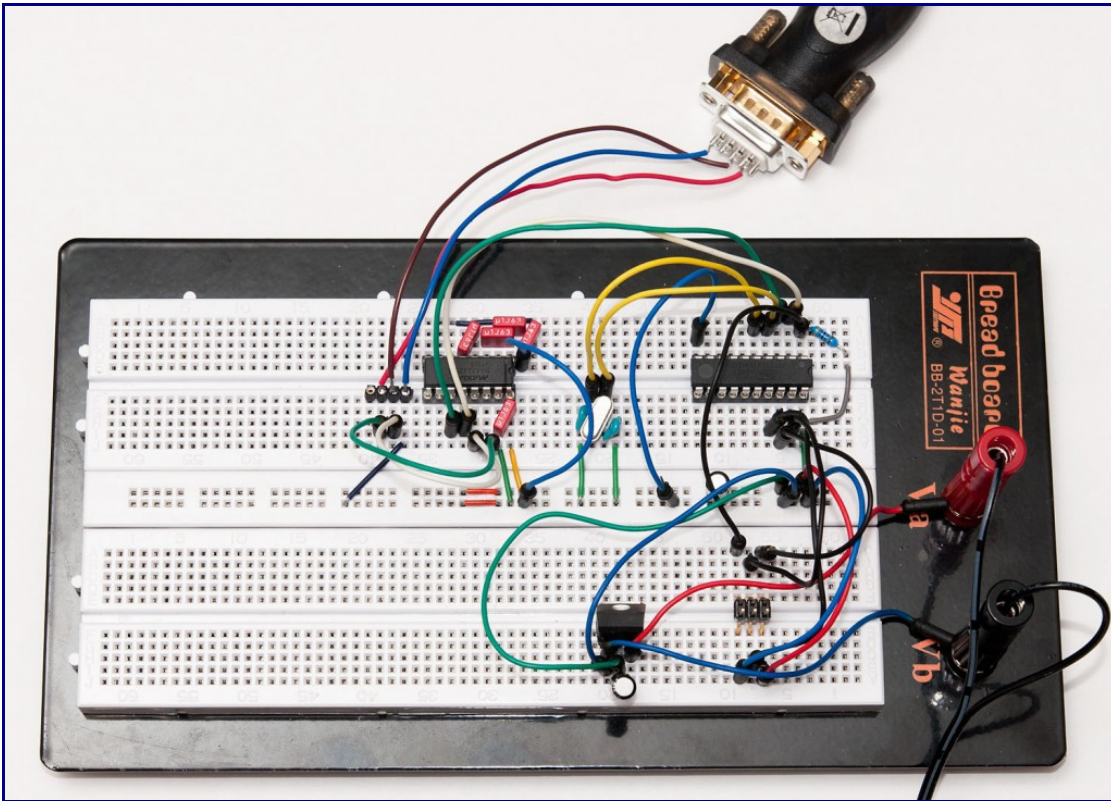
Using Putty to connect to Raspberry Pi

After you've made all the connections, double check the connections once more, maybe even check the [this full sized photo](#). If everything looks OK, power up the Pi and plug in the RS-232 cable.

Now all you need to do is to fire up [Putty](#), change connection type to "Serial", enter your serial COM port, and then access the "Serial" settings (red highlight on the right image).

<https://codeandlife.com/2012/04/12/3-3v-uart-with-max3232cpe/>

Before diving right into SPI communications for my [SD tutorial](#), I wanted to have a 3.3V development platform that could output some meaningful status information, not just light a LED if something goes wrong. In this post, I will outline the basic testing platform that will be used in the upcoming part 3 of that tutorial, and discuss a little about UART on AVR in the progress. Here's what we'll build:

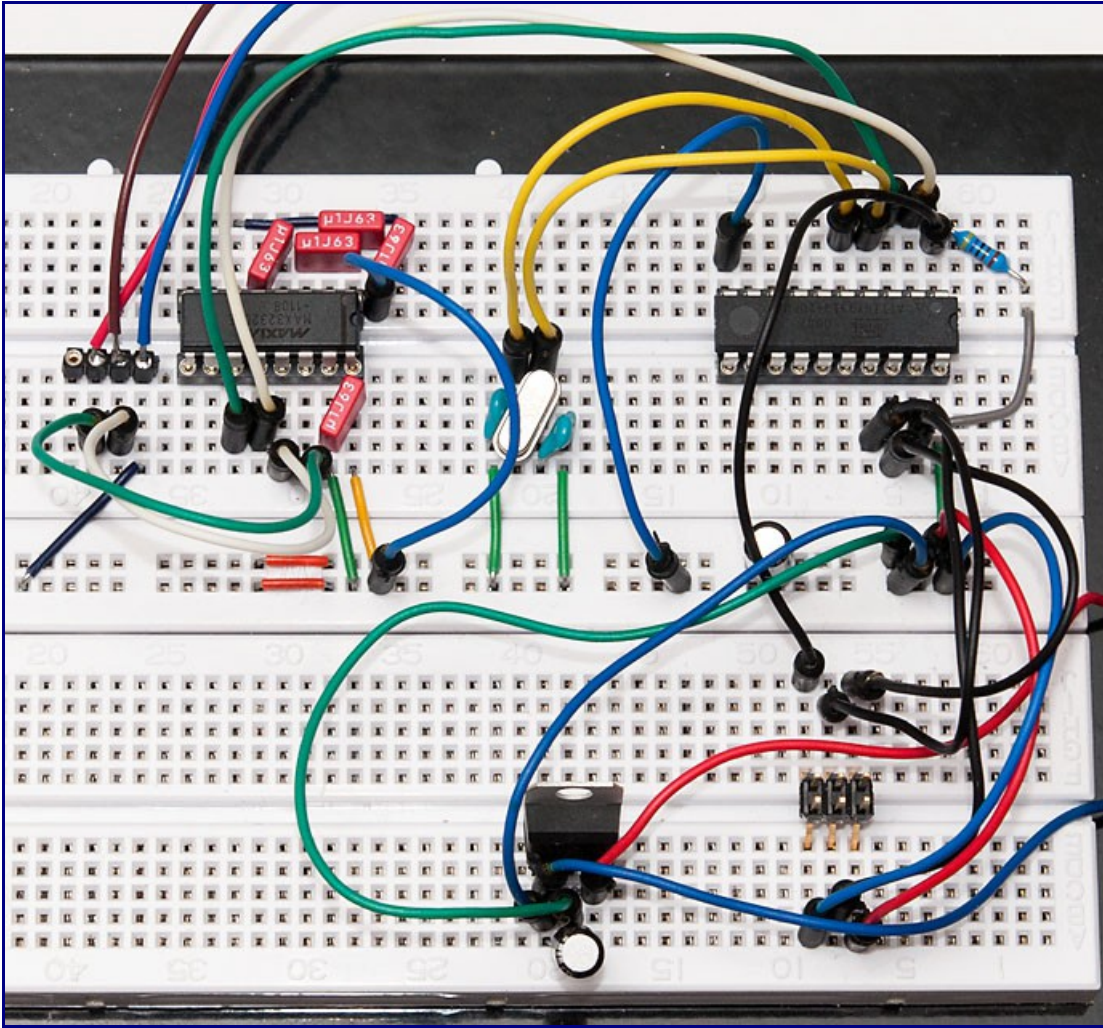


If you want to build it yourself, you'll need:

- ATtiny2313 or other AVR chip with UART pins (RX/TX) separate from SPI pins (MOSI/MISO/SCK)
- 20 MHz crystal (other speeds will work, too) and ~27 pF capacitors
- 4k7 pullup resistor for ATtiny2313 RESET pin
- 3.3V regulator such as LD1086V33 or some other 3.3V voltage source
- 2 filtering caps for the regulator input/output sides, 10 uF
- [MAX3232CPE](#) or similar RS-232 transceiver that works on a 3.3V voltage
- RS-232 port on your computer or a USB to RS-232 dongle
- RS-232 to breadboard connector (home-soldered example seen above)

Crystal can be left out if the internal oscillator is calibrated accurately enough for RS-232 timings (less than 1 % error). Also, you can build this setup with a 5V voltage source and normal MAX232, but that combination cannot communicate directly with an SD card using level conversion.

A Word on Wiring



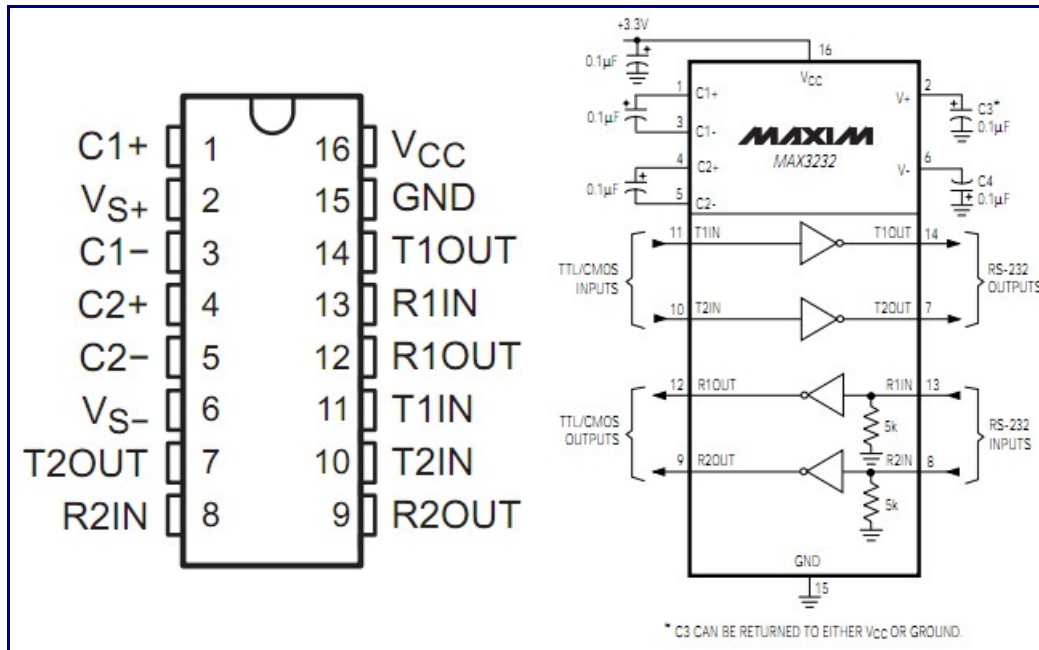
I assume that you have prior experience on wiring a 6-pin programming header to a ATtiny2313 and know the basics of regulators. If in doubt, you can refer to my [USB tutorial part 1](#) which covers the 3.3V regulator covered here in more detail. Here's one good order to do this:

1. Start by wiring >5V voltage source to regulator inputs. Add a filtering cap between the VIN and GND.
2. Connect regulator output (green wire in my setup) to breadboard VCC power rail, and GND to GND. Add another filtering cap there between VCC and GND.
3. Turn on power for a moment to confirm with a multimeter that you have 3.3V at power rails
4. Wire the ATtiny2313 with power and RESET pullup, then wire the 6-pin programming header
5. Connect a programmer and check that the chip responds
6. Add a crystal and 27 pF caps on both sides connected to ground. Wire the crystal to ATtiny2313 XTAL pins
7. Update fuses – for ATtiny2313 the 20 MHz crystal the low fuse should be FE
8. You could flash a simple LED blinker at this point to see if the crystal is working OK

RS-232 with MAX3232CPE

The MAX232 and it's pin-compatible 3V version, MAX3232 is basically a nice level shifting

circuit designed to convert the +-12V lines from RS-232 into TTL-compatible 5V (232) or 3.0-5.5V (3232) levels. To reach the higher (and lower) voltages, it uses charge pumps and four 0.1 uF (standard MAX232 needs 1.0 uF) capacitors, plus one more for VCC stabilization. Here's the pinout and logical diagram:



So basically we put a capacitor between C1+ and C1-, another between C2+ and C2-, third between VS+ and ground, fourth between VS- and ground, and finally the fifth between VCC and ground. We will only be using one of the two transceivers, so pins 7, 8, 9 and 10 are not connected. The TX and RX are wired as follows:

- ATtiny TXD (pin 3, green jumper wire) is wired to T1IN (pin 11)
- ATtiny RXD (pin 2, white jumper wire) is wired to R1OUT (pin 12)
- RS-232 "Receive" (pin 2, the red wire) is wired to T1OUT (with a green jumper wire)
- RS-232 "Transmit" (pin 3, the brown wire) is wired to R1IN (with a white jumper wire)

This is consistent with the following logic: ATtiny is wired to "TTL/CMOS" side of MAX3232, i.e. to T1IN and R1OUT (right schematic above), and RS-232 to the other side, T1OUT and R1IN, respectively. The ATtiny "TX" and "RX" correspond to MAX3232 T/R lettering, but the RS-232 does not – you can think the RS-232 "Receive" pin (pin 2) as computer's way of saying "I will receive data from this pin" and "Transmit" pin as "I will transmit data from this pin". In other words, "Receive" means "Transmit into this" and "Transmit" means "Receive from here". Confusing, isn't it?

In any case, after you've read the above two times, consult the [big picture above](#) again to verify that you got it correctly, and proceed to the code part. :)

UART code in AVR

The ATtiny2313 datasheet section on UART (or "USART", as they like to call it) is surprisingly clear – it even contains example code on using it. Here's a simple test program that just echoes back everything that is sent over the RS-232:

```
#include <avr/io.h>

void USARTInit(unsigned int ubrr_value) { // is UBRR>255 supported?
    //Set Baud rate
```

```

    UBRRH = (unsigned char)(ubrr_value >> 8);
    UBRRL = (unsigned char)(ubrr_value & 255);
    // Frame Format: asynchronous, no parity, 1 stop bit, char size 8
    UCSRC = (1 << UCSZ1) | (1 << UCSZ0);
    //Enable The receiver and transmitter
    UCSRB = (1 << RXEN) | (1 << TXEN);
}

char USARTReadChar() { // blocking
    while(!(UCSRA & (1<<RXC))) {}
    return UDR;
}

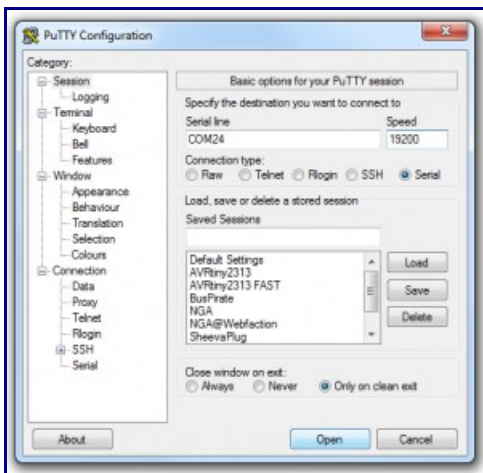
void USARTWriteChar(char data) { // blocking
    while(!(UCSRA & (1<<UDRE))) {}
    UDR=data;
}

int main() {
    USARTInit(64); // 20 MHz / (16 * 19200 baud) - 1 = 64.104x

    while(1)
        USARTWriteChar(USARTReadChar()); // echo

    return 1;
}

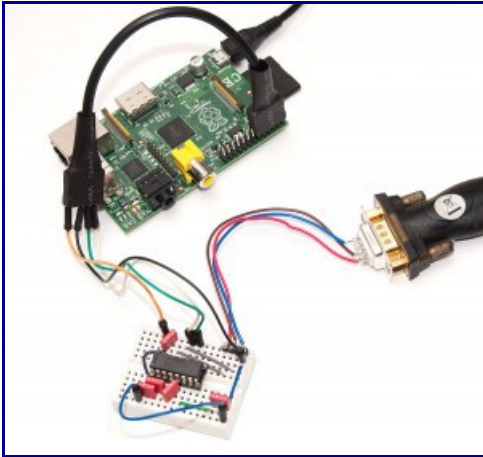
```



Note that the “UBRR magic constant” 64 is correct for a 20 MHz crystal and baud rate of 19 200. Use the formula $(\text{int})(F_{\text{CPU}} / (16 * \text{baudrate}))$ to calculate other alternatives. Note that the rounding error should not exceed 1-2 % or it will not work, so all [bit rates](#) might not work well with your particular crystal.

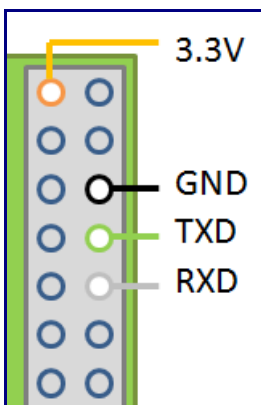
Compile and flash it to chip, and provided that you have installed your RS-232 drivers on PC, you should be able to use Putty or some other terminal to connect. Note that pressing <enter> will only send a carriage return, not a newline, so you’ll likely be stuck on the first line of your terminal when typing any text.

Now that we’ve got everything working, it would be easy to modify the code to not just echo everything back, but perhaps rot-13 encrypt everything, or do some other creative thing. But that’s it for me today, it’s past midnight and I’m getting some sleep now.



In addition to the audio, video, network and USB connectors, the Raspberry Pi also has 26 GPIO pins. These pins also include an UART serial console, which can be used to log in to the Pi, and many other things. However, normal UART devices communicate with -12V (logical “1”) and +12V (logical “0”), which may just fry something in the 3.3V Pi. Even “TTL level” serial at 5V runs the same risk.

So in this short tutorial, I’ll show you how to use a MAX3232CPE transceiver to safely convert the normal UART voltage levels to 3.3V accepted by Raspberry Pi, and connect to the Pi using Putty. This is what you’ll need:



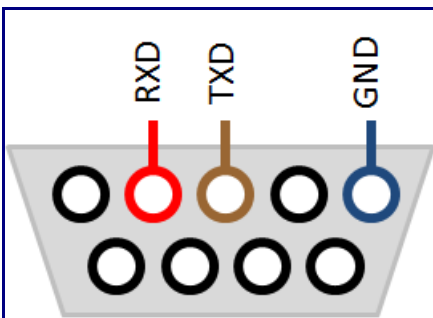
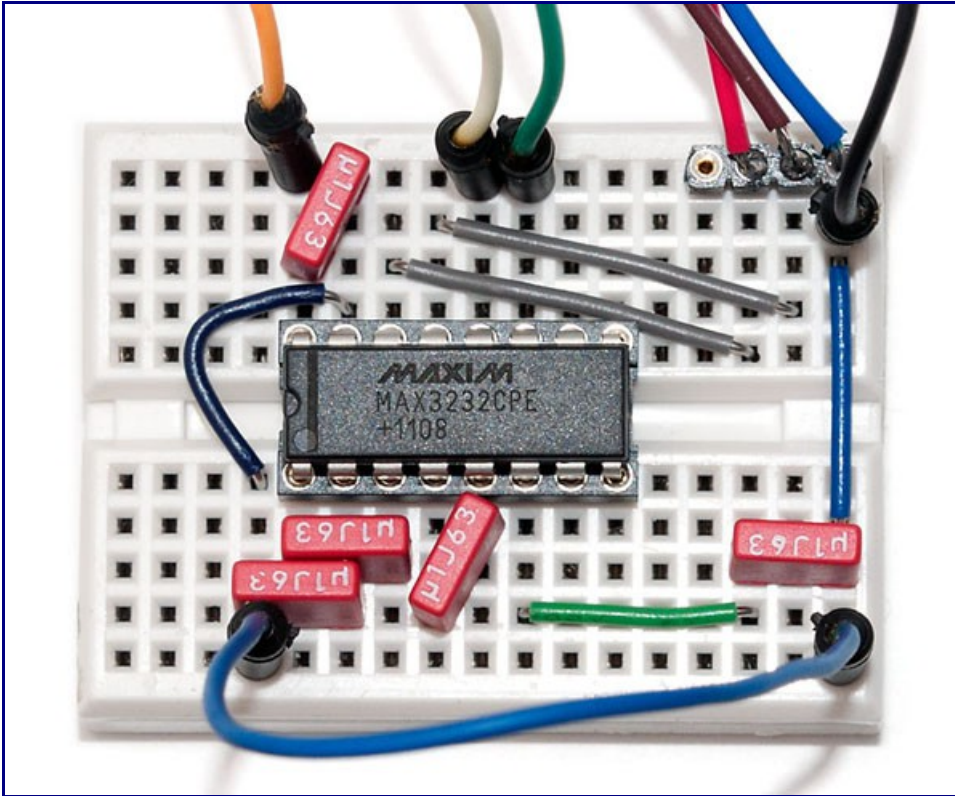
- Raspberry Pi unit
- Serial port in your PC or USB to serial -adapter
- MAX3232CPE or similar RS-232 to 3.3V logic level transceiver
- 5 x 0.1 uF capacitors (I used plastic ones)
- Jumper wires and breadboard
- Some type of female-female adapter

The last item is needed to connect male-male jumper wires to RaspPi GPIO pins. I had a short 2×6 pin extension cable available and used that, but an IDE cable and other types ribbon cable work fine as well. Just make sure it doesn’t internally short any of the connections – use a multimeter if in doubt!

The connections on Pi side are rather straightforward. We’ll use the 3.3V pin for power – the draw should not exceed 50 mA, but this should not be an issue, since MAX3232CPE draws less than 1 mA and the capacitors are rather small. GND is also needed, and the two UART pins, TXD and RXD.

Using MAX3232CPE for 3.3V UART

The MAX3232CPE is very much like it's 5V sister model, MAX232. It uses a few capacitors to deliver true +-12V RS-232 signalling on one end, and 3.3V signalling on the other. I'm not going to cover the internals in detail this time, please either refer to [the datasheet](#) or my [previous tutorial](#) discussing this same chip.

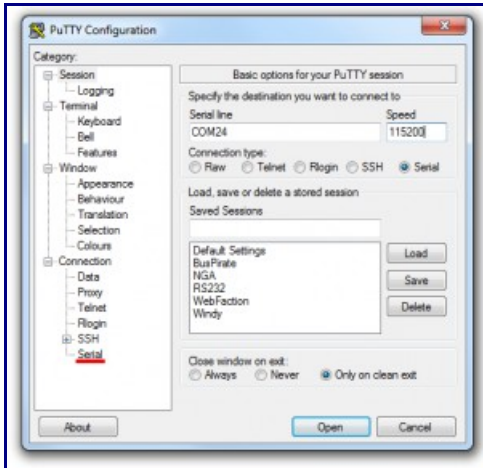


Above you can see one rather compact way to wire the MAX3232. The orange, white, green and black wires come from Raspberry Pi and provide power and data lines. The red, brown and blue wires go to the RS-232 port – see the illustration on right for connections on this side.

Update: The RS-232 connection diagram is from the side you'd solder the wires from ("back side" of the connector), and wired so you can connect a PC USB serial adapter to Raspberry Pi. If you'd like to talk to serial peripherals from Pi instead, RX/TX wires need to be reversed.

Before you connect the Pi, check with a voltmeter that GND from Raspberry Pi and GND from RS-232 do not differ from each other too much (a few millivolts is usually OK), otherwise you may risk a ground loop and damage to your equipment!

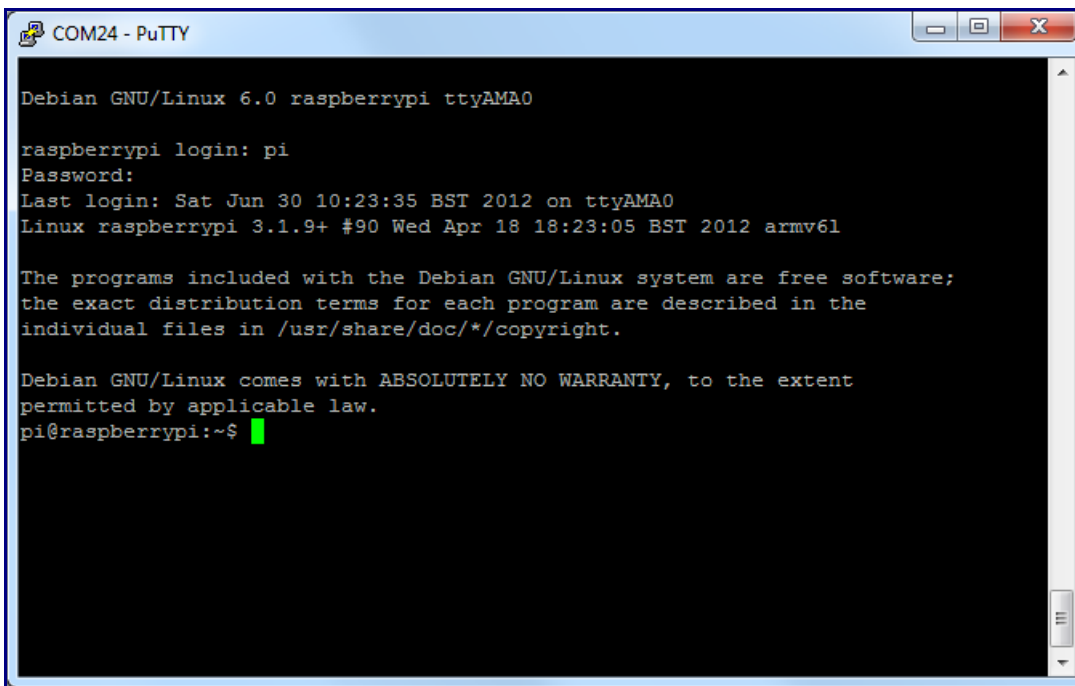
Using Putty to connect to Raspberry Pi



After you've made all the connections, double check the connections once more, maybe even check the [this full sized photo](#). If everything looks OK, power up the Pi and plug in the RS-232 cable.

Now all you need to do is to fire up [Putty](#), change connection type to "Serial", enter your serial COM port, and then access the "Serial" settings (red highlight on the right image).

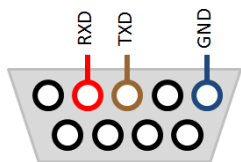
In the serial settings, make sure you have [8 bits, 1 stop bit, no parity and no flow control](#) before you connect. I initially had XON/XOFF flow control and got nothing but garbage, so don't forget this step!



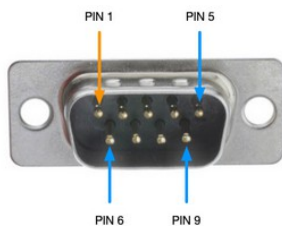
Once the wiring and the settings are correct, you should be welcomed by the Raspberry Pi login screen (note that chances are the login text is already gone when you first connect, you'll need to press enter a few times to get the login text again). Quite nice! Also, if you disable the login (Google for details), you can use the ttyAMA0 serial device for anything you like (for example, outputting logs or connecting to another device with RS-232 connection).

RS232 – 10 PIN CONNECTOR

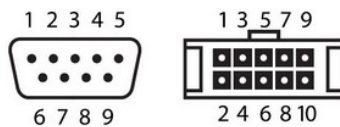
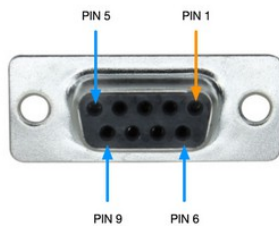
From the back side
MALE



MALE



FEMALE



| DB-9 | IDC-10 |
|-------|--------|
| pin 1 | pin 1 |
| pin 2 | pin 2 |
| pin 3 | pin 3 |
| pin 4 | pin 4 |
| pin 5 | pin 5 |
| pin 6 | pin 6 |
| pin 7 | pin 7 |
| pin 8 | pin 8 |
| pin 9 | pin 9 |
| NC | pin 10 |

*NC = Not Connected

