Part 10
-
Activating Serial

Version: 2020-03-02

Some basic information

Raspberry Pi 3's and newer are great little beasts, and have also Bluetooth. However, in order to use the Bluetooth correctly the `/dev/ttyAMA0`, the high-performance hardware serial port on previous Raspberry Pi models has been "stolen" from the GPIO header and an partly-software inferior second one, also called the miniUART, and known by `dev/ttyS0`, has been substituted in it's place. The miniUART calculates it's bit timing's from the CPU cores frequency and if the CPU is under heavy load it can corrupt the serial communications.
In order to work around this, many people "fix" the CPU core frequency so that the serial port is stable. This comes at a slight loss in performance though normally not noticeable.

So,
/dev/ttyAMA0 -> Bluetooth
/dev/ttyS0 -> GPIO serial port.

If you stick with these as is, your Bluetooth will work as nature intended AND you can use a serial port over the GPIO.

Pins to use for serial

Name		Pin Number						Name	
Alt	Base	wPi	BCM	Board	Board	BCM	wPi	Base	Alt
	+3V3			1	2			+5V	
I2C1_SDA	GPIO2	8	2	3	4			+5V	
I2C1_SCL	GPIO3	9	3	5	6			0V-GND	
	GPIO4	7	4	7	8	14	15	GPIO14	UART_TxD
	GND-0V			9	10	15	16	GPIO15	UART_RxD
	GPIO17	0	17	11	12	18	1	GPIO18	PWM
	GPIO27	2	27	13	14			0V-GND	
	GPIO22	3	22	15	16	23	4	GPIO23	
	+3V3			17	18	24	5	GPIO24	
SPI_MOSI	GPIO10	12	10	19	20			0V-GND	
SPI_MISO	GPIO9	13	9	21	22	25	6	GPIO25	
SPI_SCLK	GPIO11	14	11	23	24	8	10	GPIO8	SPI_CE0
	GND-0V			25	26	7	11	GPIO7	SPI_CE1

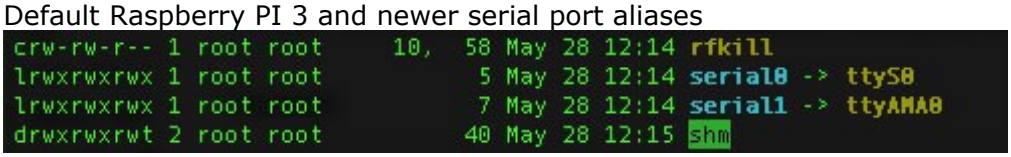
Serial Aliases

On the Raspberry Pi 3 and newer the second serial port is called `/dev/ttyS0` and is by default mapped to the GPIO pins 14 and 15. So immediately, if you have code that references `/dev/ttyAMA0` you're going to have problems and things aren't going to work. You could go through your code and replace `ttyAMA0` with `ttyS0` and that should work. However, if you find yourself use the same SD card on a Raspberry Pi 2 or older your code won't work again. In order to try and get around this the Foundation have introduced a *serial port alias*. Thus you have serial ports: `serial0` and `serial1` (RPi 3 and later Raspbian Jessie). The Raspberry Pi kernel sorts out where these point to depending on which Raspberry Pi you are on. Thus on a Raspberry Pi 3 and newer serial0 will point to GPIO pins 14 and 15 and use the "**mini-uart**" aka `/dev/ttyS0`. On Raspberry Pi 2 and older it will point to the hardware UART and `/dev/ttyAMA0`.

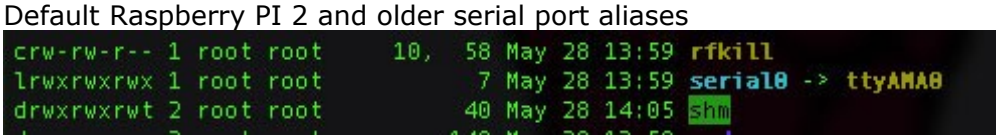
To find out where it is pointing you can use the command:

```
$ ls -l /dev | grep serial
```

Default Raspberry PI 3 and newer serial port aliases



Default Raspberry PI 2 and older serial port aliases



So where possible refer to the serial port via it's alias of "serial0" and your code should work on any Raspberry Pi's.

Enabling

First things first: Let's begin by ensuring the Raspberry Pi is up to date by running the following two commands.

```
$ sudo apt update
$ sudo apt upgrade
```

As of August 2019 the GPIO serial port is disabled by default. In order to enable it, edit config.txt:

```
$ sudo nano /boot/config.txt
```

and add the line at the bottom:

```
enable_uart=1
```

This will also lock the cpu core frequency for you so there's nothing else you need to do. If you aren't convinced, the command to add in `/boot/config.txt` is

```
core_freq=250
```

Reboot for the changes to take effect.
This should get you good serial communications for most uses.

Disabling the Console

If you are using the serial port for anything other than the console you need to disable it. This will be slightly different depending on whether you are running a Raspberry Pi 3 and newer or not. For Raspberry Pi 2 or older , remember it's `/dev/ttyAMA0` that is linked to the getty (console) service. So you need to perform this command from a terminal window:

Raspberry Pi 3 and later

```
$ sudo systemctl stop serial-getty@ttyS0.service
$ sudo systemctl disable serial-getty@ttyS0.service
$ sudo systemctl mask serial-getty@ttyS0.service
```

Raspberry 2 and older

```
$ sudo systemctl stop serial-getty@ttyAMA0.service
$ sudo systemctl disable serial-getty@ttyAMA0.service
$ sudo systemctl mask serial-getty@ttyAMA0.service
```

Note:
Disabling the service deletes the symlink, so the unit file itself is not affected, but the service is not loaded at the next boot, when systemd reads /etc/systemd/system.
However, a disabled service can be loaded, and will be started if a service that depends on it is started. Enable and disable only configure auto-start behaviour for units, and the state is easily overridden.
A masked service is one whose unit file is a symlink to /dev/null. This makes it "impossible" to load the service, even if it is required by another, enabled service.
When you mask a service, a symlink is created from /etc/systemd/system to /dev/null, leaving the original unit file elsewhere untouched. When you unmask a service the symlink is deleted. However, I have noticed that these commands are not always honoured.
You also need to remove the console from the cmdline.txt. If you edit this with:

```
$ sudo nano /boot/cmdline.txt
```

you will see something like

```
dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline fsck.repair=yes root wait
```

→ remove the line: `console=serial0,115200` , save and reboot for changes to take effect.

Or something like this

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

→ Remove all references to ttyAMA0 (which is the name of the serial port). The file will now look like this

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
elevator=deadline rootwait
```

When you have finished press [Ctrl] + X. This will ask if you want to save the modified files. Press 'Y' and then hit [Return] to save the file with the same name.

Edit the `/boot/config.txt`

```
$ sudo nano /boot/config.txt
```

and add look for `console=serial0,115200 or console=ttyAMA0,115200`. If found, comment it out

```
#console=serial0,115200
#console=ttyAMA0,115200
```

Save and reboot

```
$ sudo reboot
```

Swapping the Serial Ports on Raspberry Pi 3 and newer

What if you don't want to use the Bluetooth and you want that high performance `/dev/ttyAMA0` back on the GPIO? Well you can do this and the way you do this is via a device overlay called "`pi3-miniuart-bt`" i.e. use the mini-uart, `/dev/ttyS0`, for Bluetooth. Note that you may get some loss of performance on your Bluetooth though.
You can also just disable the Bluetooth all together by using another overlay "`pi3-disable-bt`".

Swapping
To use add the following line to the /boot/config.txt

```
$ sudo nano /boot/config.txt
```

and add:

```
enable_uart=1
dtoverlay=pi3-miniuart-bt
```

Save and reboot for changes to take effect.

You can check that it has worked by:

```
$ ls -l /dev
```

and you'll see something like this:

Before the changes, default Raspberry Pi 3 and newer serial port aliases

```
crw-rw-r-- 1 root root     10,  58 May 28 12:14 rfkill
lrwxrwxrwx 1 root root           5 May 28 12:14 serial0 -> ttyS0
lrwxrwxrwx 1 root root           7 May 28 12:14 serial1 -> ttyAMA0
drwxrwxrwt 2 root root          40 May 28 12:15 shm
```

After the changes

```
crw-rw-r-- 1 root root     10,  58 May 27 22:04 rfkill
lrwxrwxrwx 1 root root           7 May 27 22:04 serial0 -> ttyAMA0
lrwxrwxrwx 1 root root           5 May 27 22:04 serial1 -> ttyS0
drwxrwxrwt 2 root root          40 May 27 22:04 shm
```

Disabling Bluetooth
Add the following line to the /boot/config.txt

```
$ sudo nano /boot/config.txt
```

and add:

```
enable_uart=1
dtoverlay=pi3-disable-bt
```

Save and reboot for changes to take effect.

You can check that it has worked by:

```
$ ls -l /dev
```

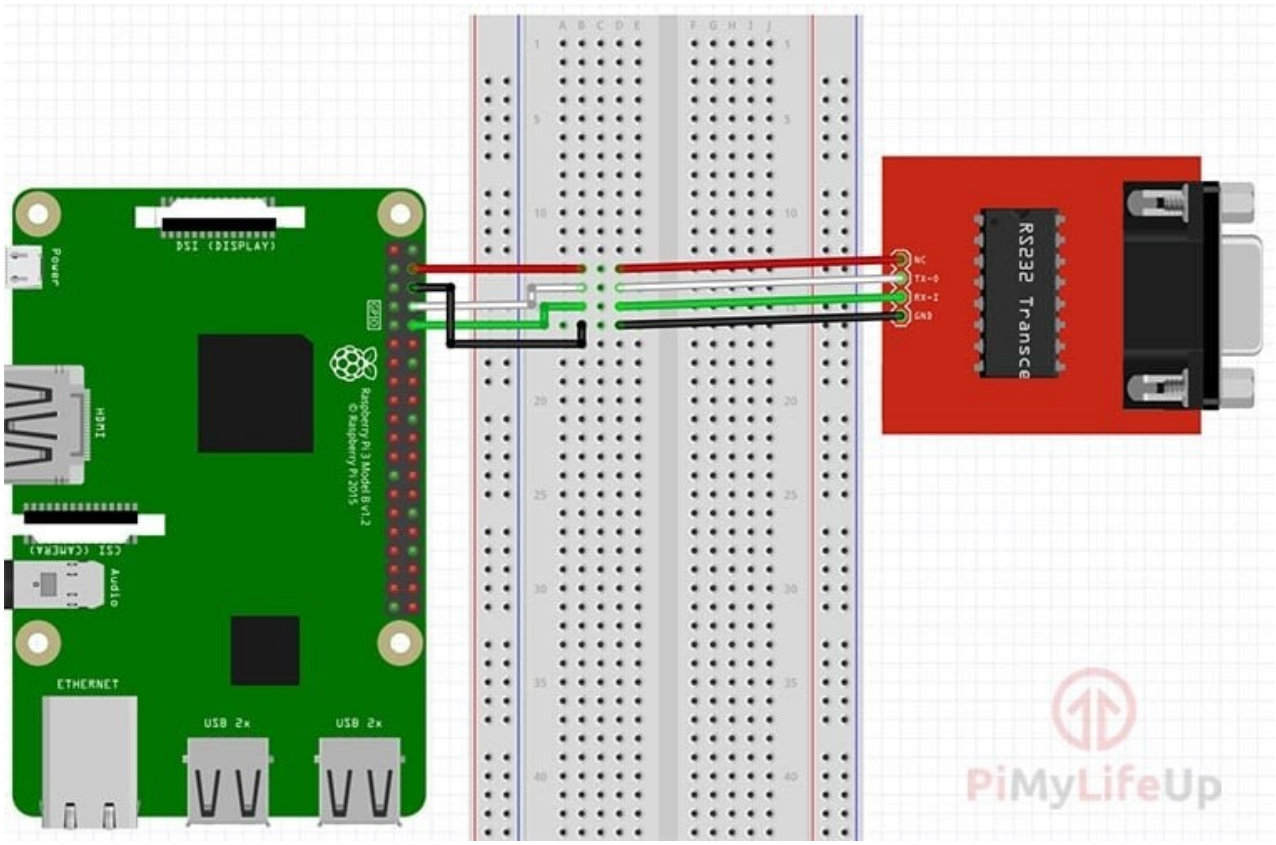
Do not forget to disable the console, if not needed (see above)

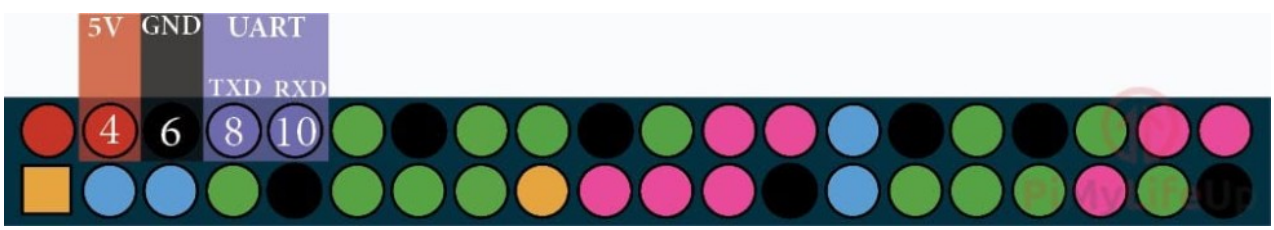
Wiring your Raspberry Pi for Serial Pi

You need at least this extra component : RS232 to TTL converter board

On your RS232 to TTL adapter you should find at least 4 connections, some circuits do come with more connections but the only four you need is: VCC (IC Power-supply pin), TX (Transmitted Data), RX (Received Data) and GND (Ground power-supply pin)
You can connect the wires directly to the GPIO Pins or use the breadboard as a middleman like we did in this tutorial. We mainly did this as we didn't have any female to female breadboard wire available to us.
Wiring your RS232 to TTL adapter to your Raspberry Pi is a simple process, with it requiring only 4 of the GPIO connecting to be wired to the serial connector, even better all 4 GPIO pins needed are in a row so it is easy to follow. Make use of our table and guide below to connect your serial connector to your Raspberry Pi

- VCC connects to Pin 4.
- TX connects to Pin 8.
- RX connects to Pin 10.
- GND connects to Pin 6.





Test the Serial Port

A great way to test out the serial port is to use the `minicom` program. If you don't have this installed, run

```
$ sudo apt-get –y install minicom
```

Connect your PC to the Raspberry Pi serial port using an appropriate serial port adapter and wiring, then open Putty or a similar serial terminal program on PC side. Setup a connection using the serial port at 9600 baud.

Now run up minicom on the Raspberry Pi using

```
$ sudo minicom –b 9600 –o –D /dev/serial0

or

$ sudo minicom –b 9600 –o –D /dev/serial1
```

What you type into the minicom terminal screen should appear on the serial PC terminal and vice versa.

Setting up the Raspberry Pi for Serial Read and Write

Let's now check to make sure that everything has been changed correctly by running the following command on your Raspberry Pi.

```
$ dmesg | grep tty
```

Here you want to make sure the following message is not displayed in the output, if it is not there then you can skip onto the next section. Otherwise start over from step 2. These messages basically indicate that Serial Login is still enabled for that interface.

- Raspberry Pi 3 and older and Raspberry Pi Zero W

  ```
  [ttyS0] enabled
  ```

- Raspberry Pi 2 and older and also the Raspberry Pi Zero

  ```
  [ttyAMA0] enabled
  ```

Programming the Raspberry Pi for Serial Writing

To start off lets begin writing our serial_write.py script, this will basically write data over the serial port. Run the following two commands on your Raspberry Pi to begin writing the file.

```
$ mkdir ~/serial
$ cd ~/serial
$ nano serial_write.py
```

Within this file write the following lines of code:

```
#!/usr/bin/env python
import time
import serial
```

The first line of code is there to tell the operating system what it should try running the file with. Otherwise it will likely attempt to run it as a normal bash script.
The first import is, **time**. We use this library to temporary sleep the script every now and then for our test counter, you don't need this package to be able to do serial writes.
The second import is, **serial**. This library contains all the functionality to deal with serial connections, this allows reading and writing through the serial ports.

```
ser = serial.Serial(
    port = '/dev/ttyS0', #Replace ttyS0 with ttyAM0 for Pi1,Pi2,Pi0
    baudrate = 9600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout=1
)
counter=0
```

This section of code primarily instantiates the serial class, setting it up with all the various bits of information that it needs to make the connection with.

- port – This defines the serial port that the object should try and do read and writes over. For Pi 3 and Pi Zero W this should be tty0. If you are using a Pi 2 and older, or the base Pi Zero then you should be using ttyAM0.
- baudrate – This is the rate at which information is transferred over a communication channel.
- parity – Sets whether we should be doing parity checking, this is for ensuring accurate data transmission between nodes during communication.
- stopbits – This is the pattern of bits to expect which indicates the end of a character or of the data transmission.
- bytesize – This is the number of data bits.
- timeout – This is the amount of time that serial commands should wait for before timing out.

```
while 1:
    ser.write('Write counter: %d \n'%(counter))
    time.sleep(1)
    counter += 1
```

This code is rather simple, it loops forever continually writing the text "Write Counter: 1" (where 1 is replaced with the current counter number) to the serial port. This means that any script or device listening on the other side will continually receive that text.
On each loop, we use the time library to sleep the script for 1 second before increasing the counter, this is to try and not spam the serial port.

Once you have finished writing the serial_write.py script it should look somewhat like what is displayed below:

```
#!/usr/bin/env python
import time
import serial

ser = serial.Serial(
    port = ' /dev/ttyS0', #Replace ttyS0 with ttyAM0 for Pi1,Pi2,Pi0
    baudrate = 9600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout = 1
)
counter=0

while 1:
    ser.write("Write counter: %d \n"%(counter))
    time.sleep(1)
    counter += 1
```

Once you are sure you have entered the code correctly you can save the file by pressing Ctrl + X then pressing Y and then finally hitting Enter.

Now that we have completed writing the serial_write.py script we can't test it just yet. First, we need to write the serial_read.py script. The reason for this is to tell if serial writes are actually being written through the serial, we need something to actually be receiving them.

Programming the Raspberry Pi for Serial Reading

To start off let's begin writing the serial_read.py script, this will basically write data over the serial port. Run the following two commands on your Raspberry Pi to begin writing the file.

```
$ mkdir ~/serial
$ cd ~/serial
$ nano serial_read.py
```

Within this file write the following lines of code:

```
#!/usr/bin/env python
import time
import serial

ser = serial.Serial(
    port = '/dev/ttyUSB0',
    baudrate = 9600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout = 1
)
```

Since we have already gone over large amounts of this code we won't bother going over it again. The only difference between this block of code and the one we used in our serial write script is that for the port we are using our USB device. In our case this was ttyUSB0, remember to change this if you got a different result earlier on in this Raspberry Pi serial read and write tutorial.

```
while 1:
    x = ser.readline()
    print x
```

This piece of code is very simple. Basically, it utilizes a function from a serial object that we setup earlier in the code. This function reads a terminated line, basically this means it reads until it hits a line that ends in "\n". Anything after that will be rejected. Once it reads the value it stores it into our *x* variable.
Finally we print the value that we obtain using the ser.readline() function.
Once you have finished writing the serial_read.py script it should look somewhat like what is displayed below:

```
#!/usr/bin/env python
import time
import serial

ser = serial.Serial(
    port = '/dev/ttyUSB0',
    baudrate = 9600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout = 1
)

while 1:
    x = ser.readline()
    print x
```

Once you are sure you have entered the code correctly you can save the file by pressing Ctrl + X then pressing Y and then finally hitting Enter.