Bare metal IR means that button presses from an infrared remote control will be processed by a python script running on an Orange Pi Zero where LIRC (Linux Infrared Remote Control) and ir-keytable are not installed. What is being used is the Linux kernel support for IR remotes and the python-evdev package by Georgi Valkov.

While this note is about handling IR remotes on an Orange Pi Zero running Armbian Stretch, much would probably be valid for the Raspberry Pi. I will probably look into that some time later.

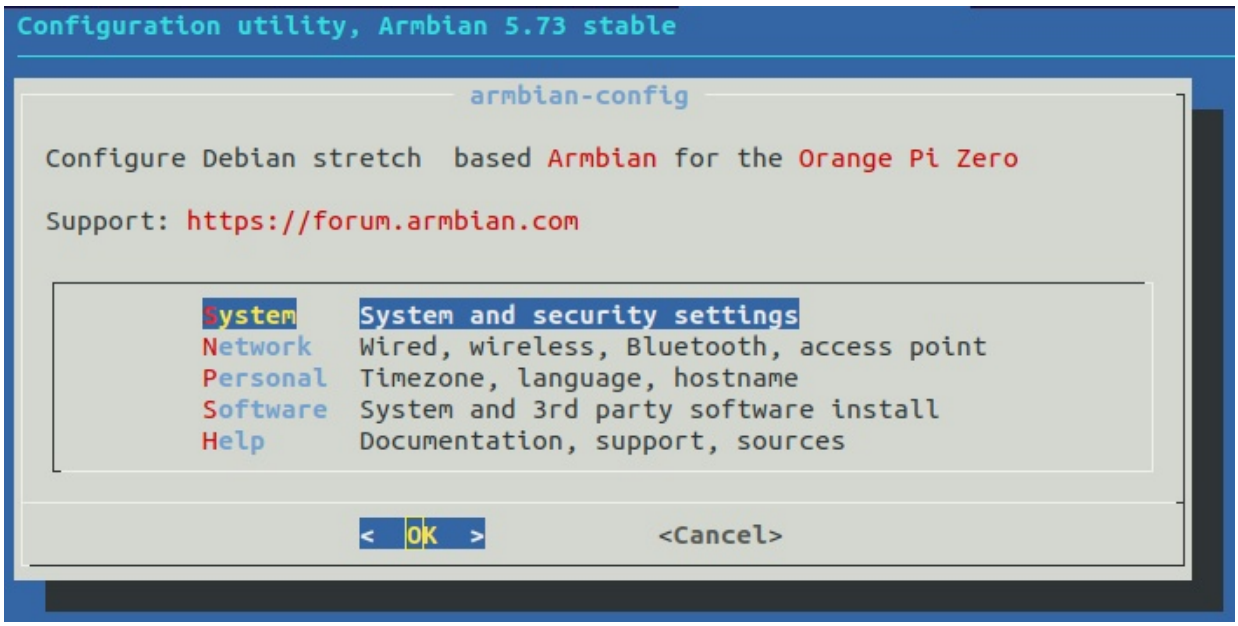
> If you ended up here because you want to control a program like Kodi with an IR remote and if is uses an IR protocol that is supported by the `Linux` kernel then there is probably no need for **LIRC** or Python scripts and so on. Just read the sections 1 and 2 below to verify that the remote can be used. Read sections 6 and 7 to set the IR protocol or protocols permanently. Then look into using `ir-keytable`. I have a few notes on that subject which can be found in the post entitled ir-keytable on the Orange Pi Zero.

Table of Contents

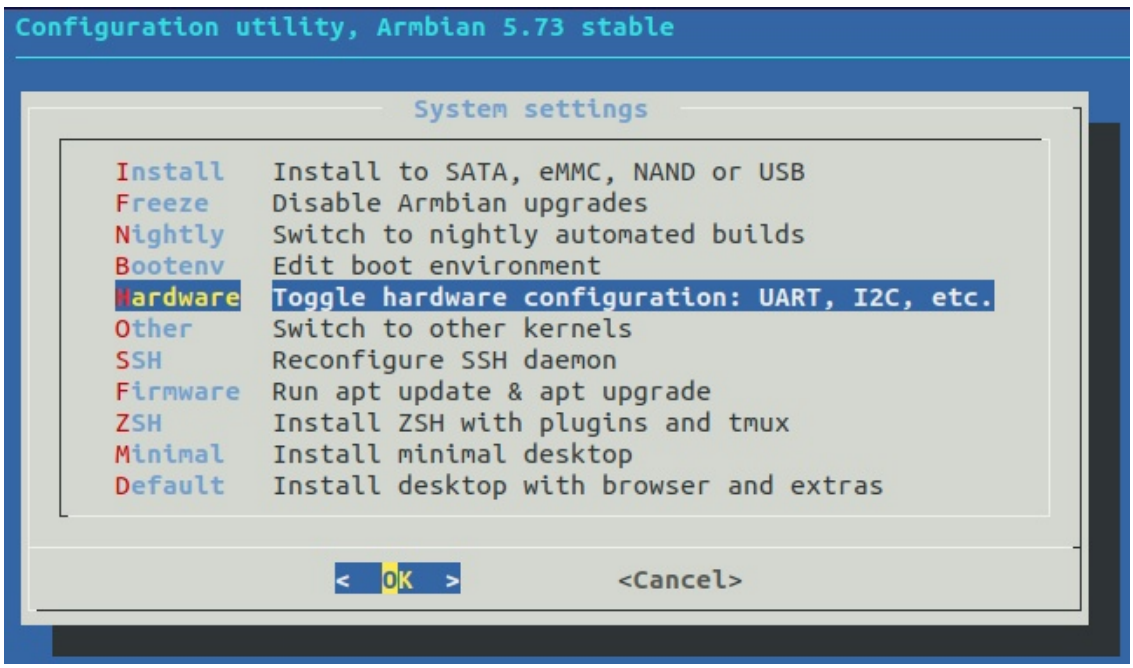
1. Installing the Kernel Module

By default the IR receiver driver, called `sunxi-cir`, is not installed. That makes sense as the Orange Pi Zero does not have an IR receiver on board. However the optional expansion card does contain one, and this is what I will be using. The simplest and indeed the only way I know of loading the kernel module when booting is to use the `armbian-config` utility. Enable the `cir` module in the **System/Hardware** settings. If you are not familiar with this procedure, here are the details starting with launching the configuration utility.

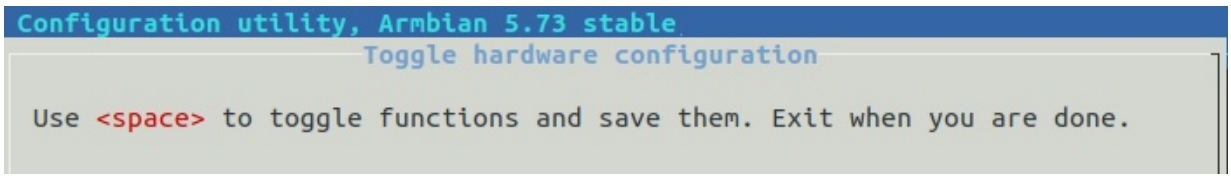
```
zero@opi:~$ sudo armbian-config
```

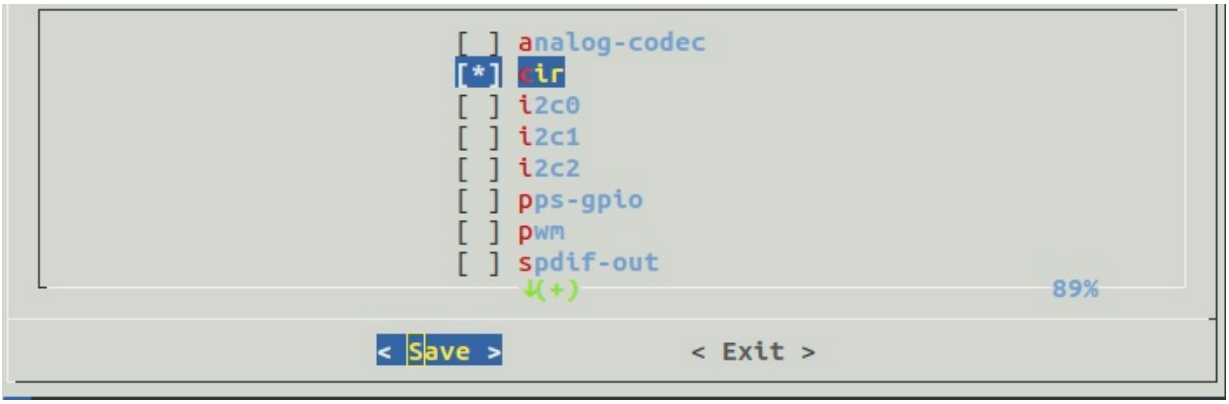


Make sure **System** is highlighted using the up and down cursor keys and then "clicking" on the **< OK >** button. That means pressing the space bar or Enter button with **< OK >** highlighted. If that button is not highlighted use the Tab or left and right cursor keys to make it so.

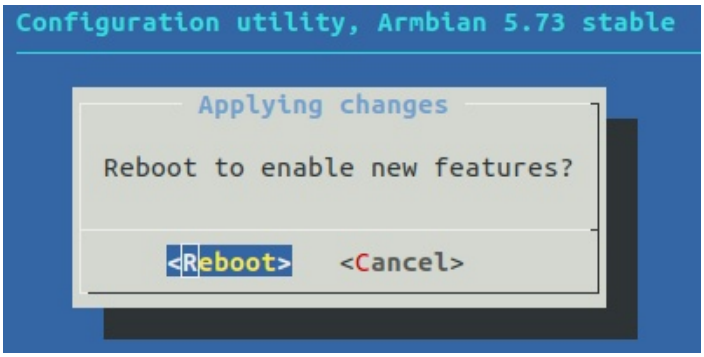


In the `System settings` screen, highlight the **Hardware** option and click on the **< OK >** button.





In the `Toggle hardware configuration` screen highlight the **cir** function and press the space bar so that a star '*' appears to the left of **cir**. Click on the **< Save >** button and then on the **< Exit >** button.



Finally, click on the **< Reboot >** button.

After a minute or so, start a new SSH session with the OPiZ and perform some checks to see if the module is loaded and initialized.

```
zero@opi:~$ lsmod | grep -i cir
sunxi_cir              16384  0

zero@opi:~$ dmesg | grep -i ir
...
[    8.767310] Registered IR keymap rc-empty
[    8.767518] rc rc0: sunxi-ir as /devices/platform/soc/1f02000.ir/rc/rc0
[    8.767835] input: sunxi-ir as /devices/platform/soc/1f02000.ir/rc/rc0/input0
[    8.771309] rc rc0: lirc_dev: driver sunxi-ir registered at minor = 0, raw IR rec
[    8.780154] sunxi-ir 1f02000.ir: initialized sunXi IR driver
...
zero@opi:~$ cat /proc/bus/input/devices
I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="sunxi-ir"
P: Phys=sunxi-ir/input0
S: Sysfs=/devices/platform/soc/1f02000.ir/rc/rc0/input0
U: Uniq=
H: Handlers=kbd event0
```

```
B: PROP=0
B: EV=100013
B: KEY=1000000 0 0 0 0
B: MSC=10
zero@opi:~$ ls /dev/input/event*
/dev/input/event0
```

Everything looks good. The module is loaded, it is initialized at boot time and IR remote control button presses show up on input event0. Time to try the remote control. Note if the `xxd` filter is not installed, just run the command without the pipe: `| xxd`. The output will look very weird, but who cares at this point.

```
zero@opi:~$ cat /dev/input/event0 | xxd
```

It looked so promising, but nothing happened as I pressed buttons on the remote IR control aimed at the receiver. Break out of the loop by pressing the Ctrl C combination. As can be seen, the remote is one of those very cheap remotes (about $1 US for the remote and a receiver) from the usual Chinese vendors. It seemed like a wise move, instead of trying this with the complex set box remote which I hope to use in the end. These cheap remotes are known to use the NEC protocol, so time to look at the supported protocol.


```
zero@opi:~$ cat /sys/class/rc/rc0/protocols
rc-5 nec rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
```

The output is the list of supported IR protocols, the one in square brackets is being used. I found that it was necessary to become the "superuser" to enable another protocol.

```
zero@opi:~$ sudo su
root@opi:/home/zero# echo nec > /sys/class/rc/rc0/protocols
root@opi:/home/zero# exit
exit
zero@opi:~$ cat /sys/class/rc/rc0/protocols
rc-5 [nec] rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
```

This worked, something was being read from the event as keys were pressed on the remote.

```
```

```
zero@opi:~$ cat /dev/input/event0 | xxd
00000000: a988 545c 9970 0100 0400 0400 1600 0000  ..T\.p..........
00000010: a988 545c 9970 0100 0000 0000 0000 0000  ..T\.p..........
00000020: a988 545c f337 0200 0400 0400 1600 0000  ..T\.7..........
00000030: a988 545c f337 0200 0000 0000 0000 0000  ..T\.7..........
00000040: a988 545c b9dc 0300 0400 0400 1600 0000  ..T\............
00000050: a988 545c b9dc 0300 0000 0000 0000 0000  ..T\............
```

As before, press the Ctrl C keys in combination when tired of looking at this unintelligible output in response to button presses on the remote.

If the `nec` protocol does not work, open a second session and change the protocol as the super user as shown above (or see the last section below). The change is immediate so you can keep on clicking on the remote to see if `cat` reports an event in the first session.

One final remark at this juncture. The user, `zero` here, probably needs to be a member of the `input` group to read from the events queue. By default, this is true in **Armbian**.

```
zero@opi:~$ groups
zero dialout sudo audio video plugdev systemd-journal input netdev ssh
```

Wondering where `cir` comes from? I was. **linux-sunxi.org** had a reference to a Wikipedia page Consumer IR which I found informative. Among other things it mentioned that "CIR is the most common type of free-space optical communication" with a link to the latter. Here we go, down the rabbit hole again.

The reference for all this is the section entitled mainline kernel (4.x) and sunxi-cir on the IR page at **linux-sunxi.org**.

2. Getting the Remote Scan Codes

The binary output that was read from the `event0` input is not easily interpreted. Of course there is a program to make things easier.

```
zero@opi:~$ evtest /dev/input/event0
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "sunxi-ir"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 152 (KEY_SCREENLOCK)
  Event type 4 (EV_MSC)
    Event code 4 (MSC_SCAN)
```

```
Key repeat handling:
  Repeat type 20 (EV_REP)
    Repeat code 0 (REP_DELAY)
      Value    500
    Repeat code 1 (REP_PERIOD)
      Value    125
Properties:
Testing ... (interrupt to exit)
Event: time 1549043265.338647, type 4 (EV_MSC), code 4 (MSC_SCAN), value 18
Event: time 1549043265.338647, -------------- SYN_REPORT ------------
Event: time 1549043265.389764, type 4 (EV_MSC), code 4 (MSC_SCAN), value 18
Event: time 1549043265.389764, -------------- SYN_REPORT ------------
Event: time 1549043265.497432, type 4 (EV_MSC), code 4 (MSC_SCAN), value 18
Event: time 1549043265.497432, -------------- SYN_REPORT ------------
Event: time 1549043267.562994, type 4 (EV_MSC), code 4 (MSC_SCAN), value 19
Event: time 1549043267.562994, -------------- SYN_REPORT ------------
Event: time 1549043267.614064, type 4 (EV_MSC), code 4 (MSC_SCAN), value 19
```

Here is some information about the supported event types. As can be seen there are two types of events being reported:

EV_MSC
: Described as "miscellaneous input data" because these events do not fall in the other defined categories such as EV_KEY (keyboard, buttons...), EV_REL (mouse movements), EV_SW (switches).

EV_SYN
: Describe as "markers to separate events".

Each event type has a set of codes identifying events. SYN_REPORT is one of 4 codes associated with EV_SYN and it is "synchronize[s] and separate[s] events into packets of input data changes occurring at the same moment in time". This is not very meaningful here but it could be useful to differentiate between X and Y movements of the mouse that occurred simultaneously and two sequential movements along the X and Y axis.

The code MSC_SCAN is not defined in the documentation but it is obviously refering to a scan code generated by the input device. The scan code associated with each button on the IR remote is reported in the `value` field. The `time` field is probably the MSC_TIMESTAMP code which is the "number of microseconds elapsed since the last reset". It is supposed to be 32 bit unsigned integer value, so it is not immediately clear how the value should be read.

It did not take very long to create the following table to link the generated scan codes with the name of the buttons on the remote.

Code		Button		Code		Button
`0x52`	`82`	0		`0x46`	`70`	UP

0x16	22	1		0x40	64	OK
0x19	25	2		0x15	21	DOWN
0x0d	13	3		0x43	67	RIGHT
0x0c	12	4		0x44	68	LEFT
0x18	24	5		0x42	66	*
0x5e	94	6		0x4a	74	#
0x08	8	7				
0x1c	28	8				
0x5a	90	9				

3. Python Prerequisites

Python 3.5 is installed by default in **Armbian Strecth**. However this is not the case for `pip3`, the Python package installer, and its associated tools which will be used to install the needed library.

```
zero@opi:~$ sudo apt install -y python3-pip python3-dev python3-setuptools python3-
...
0 upgraded, 11 newly installed, 0 to remove and 0 not upgraded.
Need to get 40.7 MB of archives.
After this operation, 55.3 MB of additional disk space will be used.
...
Setting up python3-dev (3.5.3-1) ...
```

Now `pip3` (`pip` for Python 3) can be used to get the needed library.

```
zero@opi:~$ pip3 install evdev
Collecting evdev
  Using cached https://files.pythonhosted.org/packages/7e/53/374b82dd2ccec240b7388c6
...
Successfully installed evdev-1.1.2
```

That is the only library that will be used here.

4. Handling IR Events

Documentation for the `evdev` library can be found at Read the Docs. What follows is mostly a rehash of parts of the Reading events tutorial with slight modifications for the particulars in this context. I will begin with an interactive Python session to test everything.

```
zero@opi:~$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import evdev
>>> devices = [evdev.InputDevice(path) for path in evdev.list_devices()]
>>> for device in devices:
...   print(device.path, device.name, device.phys)
...       don't forget to press Enter twice!
/dev/input/event0 sunxi-ir sunxi-ir/input0

>>> device = evdev.InputDevice('/dev/input/event0')
>>> print(device)
device /dev/input/event0, name "sunxi-ir", phys "sunxi-ir/input0"
>>> for event in device.read_loop():
...   print(event)
...       don't forget to press Enter twice!
event at 1549134307.050319, code 04, type 04, val 25
event at 1549134307.050319, code 00, type 00, val 00
event at 1549134307.101433, code 04, type 04, val 25
event at 1549134307.101433, code 00, type 00, val 00
event at 1549134307.209101, code 04, type 04, val 25
event at 1549134307.209101, code 00, type 00, val 00
event at 1549134309.079462, code 04, type 04, val 24
event at 1549134309.079462, code 00, type 00, val 00
event at 1549134309.130551, code 04, type 04, val 24
event at 1549134309.130551, code 00, type 00, val 00
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/zero/.local/lib/python3.5/site-packages/evdev/eventio.py", line 45, ir
    r, w, x = select.select([self.fd], [], [])
KeyboardInterrupt
```

Press the keyboard combination Ctrl C to interrupt the loop. The first field displayed by `evdev` when a remote event is read is a time code. Each button press seems to come as two events, a button press with `code=type=04` and a synchronization event (`EV_SYN`) with `code=type=00` which is not of much use. It can easily be striped out.

```
>>> device = evdev.InputDevice('/dev/input/event0')
>>> for event in device.read_loop():
...   if event.type == 4:
...     print(event.value)
...       don't forget to press Enter twice!
25
25
```

```
25
24
24
 ^C Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/zero/.local/lib/python3.5/site-packages/evdev/eventio.py", line 45, ir
    r, w, x = select.select([self.fd], [], [])
KeyboardInterrupt
>>>  quit()
zero@opi:~$
```

5. A Short Python Script

This simple script will read all events from the input queue, and display the name of the buttons pressed on the IR remote based on their scan codes.

```python
#!/usr/bin/python3

from evdev import InputDevice

irr = InputDevice('/dev/input/event0')

print("Press remote IR buttons, Ctrl-C to quit")

for event in irr.read_loop():
   if event.type == 4:
      try:
         if event.value == 82:
            print("key: 0")
         elif event.value == 22:
            print("key: 1")
         elif event.value == 25:
            print("key: 2")
         elif event.value == 13:
            print("key: 3")
         elif event.value == 12:
            print("key: 4")
         elif event.value == 24:
            print("key: 5")
         elif event.value == 94:
            print("key: 6")
         elif event.value == 8:
            print("key: 7")
         elif event.value == 28:
            print("key: 8")
         elif event.value == 90:
            print("key: 9")
         elif event.value == 64:
            print("key: OK")
```

```
            elif event.value == 70:
               print("key: UP")
            elif event.value == 21:
               print("key: DOWN")
            elif event.value == 68:
               print("key: LEFT")
            elif event.value == 67:
               print("key: RIGHT")
            elif event.value == 66:
               print("key: *")
            elif event.value == 74:
               print("key: #")
            else:
               print(event)
         except:
            print("Problem key pressed...")
```

Download version: test_ir.py.

If I had written this in Pascal, I would have created an array of records, with the scan code as the first field and the button name as the second field. The array would be sorted by the scan code and a binary search would be used to find the button name given the scan code. It remains for me to find the idiomatic way of doing this in Python.

The output of the script can be seen below.

```
zero@opi:~$ python3 test_ir.py
Press remote IR buttons, Ctrl-C to quit
key: 6
key: 6
key: 6
key: 5
key: 5
key: OK
key: OK
key: OK
^C Traceback (most recent call last):
  File "test_ir.py", line 9, in <module>
    for event in irr.read_loop():
  File "/home/zero/.local/lib/python3.5/site-packages/evdev/eventio.py", line 45, ir
    r, w, x = select.select([self.fd], [], [])
KeyboardInterrupt
zero@opi:~$
```

The script can be made to behave as if it is a program. That is the purpose of the *shebang* line stating with `#!` at the top of the file. However in **Linux** the file has to be marked as executable. In **Windows** the `.py` extension should be sufficient as it

should be associated with the Python interpreter in the registry. The extension plays no role in **Linux**.

```
zero@opi:~$ chmod +x test_ir.py
zero@opi:~$ ./test_ir.py
Press remote IR buttons, Ctrl-C to quit
...
```

There is an asynchronous version of the script. Using a second session running `htop` to monitor each script in turn, I was surprised to find that the asynchronous version seemed a little slower in handling button presses. However neither version consumed any appreciable processor time when no remote button was pressed and processing a sequence of events related to a button press took barely 1% of the processor time with the slower asynchronous implementation.

This is just verification that it is possible to handle at least some types of IR remotes without **LIRC**. The script is not really usable as it is because the repeated codes from the remote accumulate in the event queue. Either time codes need to be used or the event queue needs to be read one event at a time, flushing the queue as needed to ignore spurious repeated codes. This could be the subject of a future post perhaps.

6. Setting the IR Protocol Permanently

I glossed over the problem of setting the IR protocol to be used by the `sunxi-cir` module at the beginning of this post. Why was it necessary to run a shell as `root` using the `su` command? In other words, why does the `sudo` command not work? It has to do with file permissions and persistence of the `sudo` command.

```
zero@opi:~$ sudo echo rc-5 > /sys/class/rc/rc0/protocols
-bash: /sys/class/rc/rc0/protocols: Permission denied
zero@opi:~$ ls -l /sys/class/rc/rc0/protocols
-rw-r--r-- 1 root root 4096 Feb  3 12:21 /sys/class/rc/rc0/protocols
```

As can be seen, writing to the `protocols` file can only be done by the owner of the file which is `root`. The `echo` command is performed as user `root` because of the `sudo` prefix, but writing the output of `echo` to the file is being done as user `zero` who does not have that privilege. This reminds me of an error I sometimes make. The combined command

```
sudo apt update && apt upgrade
```

will not work because only the first one is done as user `root`. It has to be written as follows.

```
sudo apt update && sudo apt upgrade
```

Something akin to that has to be done. This is a well-known problem discussed at length in a stackoverflow exchange since 2008. Here is one of two generally agreed upon solutions to the problem.

```
zero@opi:~$ echo nec | sudo tee /sys/class/rc/rc0/protocols
nec
zero@opi:~$ cat /sys/class/rc/rc0/protocols
rc-5 [nec] rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
```

This is understandable. The output of `echo` is fed into the input of `tee`. That command is a filter that writes its input to the specified file. The `sudo` command means that `root` executes `tee` which will have the needed write privilege. Getting that straightened out was important because the Orange Pi Zero is to be run as a headless server. It will not be possible to manually set the IR protocol. Instead that is done as a `cron` task on each reboot.

```
zero@opi:~$ crontab -e
```

Add the following line at the bottom of the file.

```
@reboot echo nec | sudo tee /sys/class/rc/rc0/protocols
```

Luckily, `tee` was available on **Armbian** but what can be done on a **Linux** without that filter? The other proposed solution has `root` create a shell to which is passed the `echo` command that worked when run by `root`.

```
zero@opi:~$ sudo sh -c "echo jvc > /sys/class/rc/rc0/protocols"
zero@opi:~$ cat /sys/class/rc/rc0/protocols
rc-5 nec rc-6 [jvc] sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
```

Many prefer this solution as it does not involve a third program. There is a little *caveat* when multiple protocols need to be enabled.

```
zero@opi:~$ sudo sh -c "echo 'nec +sony' > /sys/class/rc/rc0/protocols"
zero@opi:~$ cat /sys/class/rc/rc0/protocols
IR protocols: rc-5 [nec] rc-6 jvc [sony] rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
```

Just make sure that single quotes are used around all the protocols to be echoed to

avoid confusion with the double quotes around the command passed on to the shell. I suppose this version could just as easily used in the `cron` task.

Clearly there is some "magic" surrounding the `protocols` file. Let's create a file named `prots` with the same content and same permissions as the original `protocols`. When the same `echo` command is used to enable a protocol, the content of `prots` is just replaced.

```
zero@opi:~$ sudo echo "rc-5 nec rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon
zero@opi:~$ ls -l prots
-rw-r--r-- 1 zero zero 67 Feb  3 17:02 prots
zero@opi:~$ cat prots
rc-5 nec rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc]
zero@opi:~$ echo nec | sudo tee prots
nec
zero@opi:~$ cat prots
nec
```

Clearly access to the `protocols` file is not direct. But how is this done? I clearly have a lot to learn about **Linux**.

7. My Very Own IR Protocol Setter

I was inspired by hololeap's answer on the **stackoverflow** exchange and decided to create a bash function that combined setting and displaying IR protocols in one command. Edit the shell configuration file `.bashrc` in the home directory.

```
zero@opi:~$ nano .bashrc
```

Just add the following lines and save the modified file.

```
# function to set IR remote protocol
irp() {
    [[ "$#" -ge 2 ]] && echo "Usage: irp [[+|-]<protocol>]" && return 1
    [[ "$#" -eq 1 ]] && echo "$1" | sudo tee /sys/class/rc/rc0/protocols > /dev/null
    echo -n "IR protocols: " && cat /sys/class/rc/rc0/protocols
}
```

The session has to be stopped and restarted for the command to be available or run the `exec bash` command. Instead of a "man page" explaining how to use it, here is an exhaustive list of examples.

Display the supported and enabled IR protocols

```
```

```
zero@opi:~$ irp
IR protocols: rc-5 nec rc-6 [jvc] sony rc-5-sz sanyo sharp mce_kbd xmp imon [li
```

Enabled protocols are surrounded by [] brackets. `lirc` is always enabled.

Enable a single IR protocol

```
zero@opi:~$ irp nec
IR protocols: rc-5 [nec] rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [li
```

Enable a supplementary IR protocol

```
zero@opi:~$ irp +jvc
IR protocols: rc-5 [nec] rc-6 [jvc] sony rc-5-sz sanyo sharp mce_kbd xmp imon [
```

Disable a single IR protocol

```
zero@opi:~$ irp
IR protocols: rc-5 [nec] rc-6 [jvc] sony rc-5-sz sanyo sharp mce_kbd xmp imon [
zero@opi:~$ zero@opi:~$ irp -jvc
IR protocols: rc-5 [nec] rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [li
```

Combine multiple operations

```
zero@opi:~$ irp
IR protocols: rc-5 [nec] rc-6 [jvc] sony rc-5-sz sanyo sharp [mce_kbd] xmp imor
zero@opi:~$ zero@opi:~$ irp "-jvc +sony +sanyo"
IR protocols: rc-5 [nec] rc-6 jvc [sony] rc-5-sz [sanyo] sharp [mce_kbd] xmp im
```

Disable all IR protocols

```
zero@opi:~$ irp lirc
IR protocols: rc-5 nec rc-6 jvc sony rc-5-sz sanyo sharp mce_kbd xmp imon [lirc
```

ir-keytable on the Orange Pi Zero
LIRC on the Orange Pi Zero

Last Update: 2019-02-06. Original version: February 2, 2019.