Controlling Projector using Raspberry Pi

Chandrashekhar Choudhary

November 22, 2017

Objective

The objective of this project is to demonstrate automatic switching off of the projector using raspberry pi when the HDMI cable is plugged out. The project requires us to update the lirc driver code to use the new GPIO subsystem and sysfs attributes for controlling it.

Introduction

The lirc driver `lirc_rpi` uses the older driver model to transmit and receive the IR codes. The userspace utility irsend opens the char device /dev/lirc0 to communicate with the device. The irsend converts the hex code for the key to be transmitted into a sequence of pulses and spaces. It the opens the lirc0 device and writes this raw code to it. The `lirc_rpi` takes this sequence of pulse and space and transmits it. The status of hdmi can be checked with tvservice utility. The tvvservice utility also provides an option to print the entire edid data into a file which can be parsed using edidparser tool. In our module we are using gpio46 to monitor the status of hdmi and mailbox to get the edid data.

Methodology

To complete the project we decided to divide it into three sections:

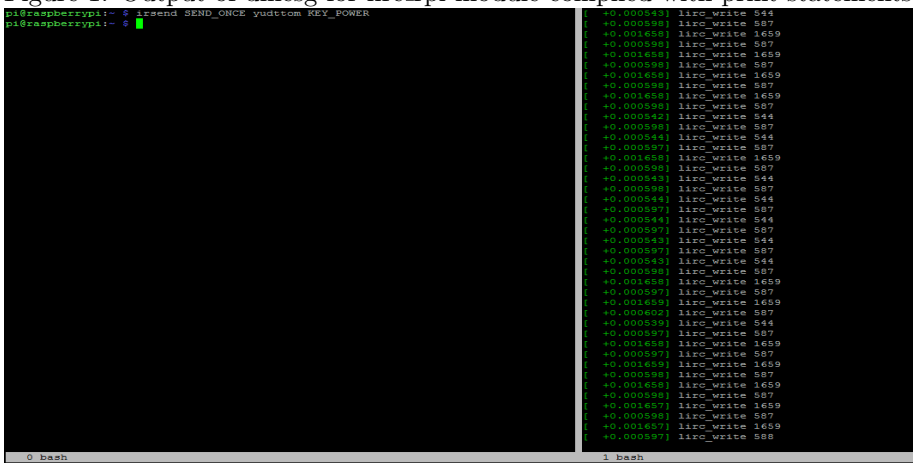
- Using the default lirc driver to receive and transmit the IR codes.
- Modifying the `lirc_rpi` driver to use the new GPIO subsystem and using sysfs attributes to control it.
- Monitoring HDMI events to detect disconnect event and then sending an IR pulse to turn off the projector.

The first part was relatively easier but was important step as we were able to test our hardware before proceeding to the next steps. We followed the steps mentioned in this[1] site to get our raspberry pi IR remote working. We had to make some changes in the `lircd_options.conf` file because by default it was set to use driver as devinput and device as auto, we changed it device /dev/lirc0 and driver as default. We used irrecord to make the config file for our remote. The irrecord can be used with –force option to get the raw code for the keys

and later we can use it with -a option to convert it into hex code. To transmit the hex code we used irsend utility and to receive it we used irw.

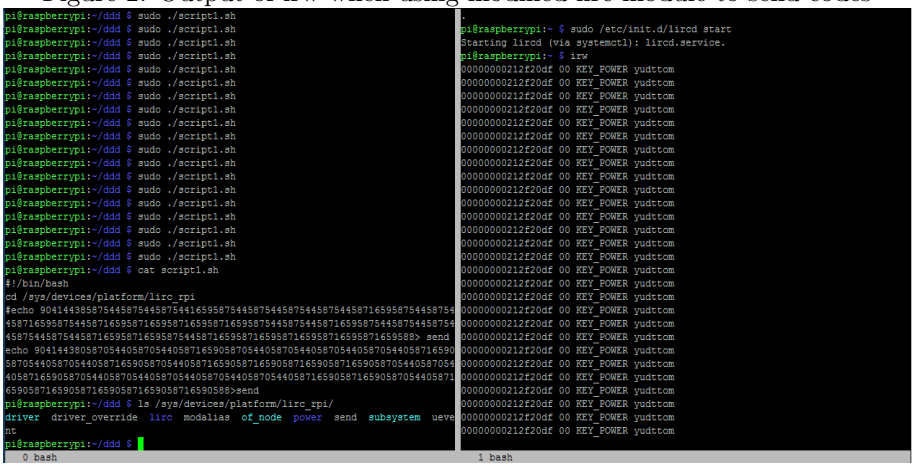
For the second part we started with reading the `lirc_rpi.c` file to understand the code and make appropriate changes. First thing that we did was adding some print statements in `lirc_rpi.c` code to see what data was being sent from userspace utility irsend to it. We noticed that the irsend utility was writing the entire raw code to the lirc device. In the figure 1 below we can see the output after adding the print statements in the `lirc_write` function. We can clearly observe that the entire raw code is being sent from the irsend command to the `lirc_write` function.

Figure 1: Output of dmesg for lirc_rpi module compiled with print statements



Using the conclusions made from the above output we tried to make a sysfs attribute named send were we could echo the entire raw code and it would send it using the transmitter. After few tries we were able to get it working. So now we have an attribute named send where we can echo any raw data that we want to transmit and it will transmit it. The figure 2 below shows the output from raspberry pi where new modified module is inserted. The script1.sh has a command to echo the raw code for power key in the send file created under /sys/devices/platform/`lirc_rpi`/. As we can see from the figure 2 that whenever we execute the script1.sh we are able to receive it using irw.

Figure 2: Output of irw when using modified lirc module to send codes

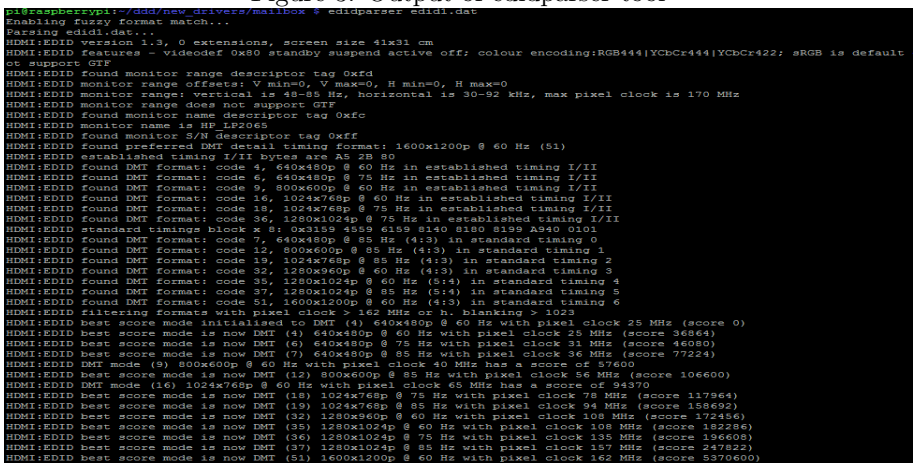


Next we realized that there was a pattern which was being followed in the raw data that was being sent. We studied the lircd.conf file created by us for the remote and realized that all the codes started with a header followed by data and a trailing bit. We got the values of header, pulse and space values for 0 and 1 and also the value for trailing bit from the lircd.conf file created and matched it with the raw code being sent with irsend. So we modified our send attribute to take hex codes for buttons instead of raw data. Now with this modification we were able to transmit codes using hex numbers, the decoding was being done in our send attribute. For now we have hardcoded the values for header, pulse and space values for 0 and 1 and trailing bits, so we need to recompile our module for different remotes, but we can create attributes for these values as well later to improve our module. This part was working properly when we were testing in lab with the projector there. So we decided to test it out with the projector in our classroom. When we tested it there it did not work which was something that we were not expecting, but still we thought of debugging the problem. It was then we realized that the code for our class projector was very big and it was out of our int range so we modified our send code again an this time instead of using int we converted the hex code into a char array of bits. We used that array to transmit the signal and were able to control the projector. While modifying the module we encountered some kernel crashes. Initially we were not able to realize what the problem was because all the terminals would just freeze whenever we tried writing anything to the send attribute. So we started a script session on our terminal were we had the dmesg output. Using that we were able to debug our problem which was quite careless on our part because we using a memory without allocating it.

We started reading up on hdmi in order to complete the third part of our project. The initial idea was to somehow detect if the hdmi was attached or not and register and unregister a dummy platform device based on its status. We thought that this will generate ADD and REMOVE uevents for which we could write a udev rule to fire up when the hdmi was disconnected to switch off the projector. While searching for how to detect if hdmi is connected we saw the device tree source for our raspberry pi, where we found that gpio 46 was being used for hdmi[2]. After that we thought of monitoring the gpio46 to see if there

was any change when hdmi was connected or disconnected. We exported the gpio46 using the file /sys/class/gpio/export and checked for any change in the value of gpio46. We noticed that whenever hdmi was connected its value changed from 1 to 0, so we thought that we will add an interrupt on gpio46 to detect rising and falling edge and based on the type of interrupt we will register or unregister our platform device. When we tried adding interrupt on that pin our interrupt handler was not being called. We tried the same code after changing the gpio46 to some other pin to check if something was wrong with our interrupt handler, but for the other gpio it worked so we concluded that we cannot add interrupt on that gpio. When we were testing our interrupt handler for other gpio our kernel crashed a few times. In our interrupt handler we were trying to register and unregister a dummy device to generate uevents but the kernel kept crashing every time we changed the pin level. Since we were using a jumper to generate interrupts on our pin we were getting many interrupts between short time period something similar to debounce on switch press. To remove multiple interrupts we added a code to check the time difference between our interrupts and take action only if it was greater than some threshold. Even after removing multiple interrupts we were not able to remove the kernel crashes. We noticed that the kernel was crashing after registering the device second time, it was giving an error that kobj already exists something is seriously wrong. We were not able to remove this error so we thought of generating uevents using the kobject as used in this code[3]. Since we were not able to add interrupt on gpio46 we thought that we will add a timer with period 1 second in our module and poll the gpio46 after every timeout and detect if there is any change between the previous and next state of the signal level of gpio46. We wrote our module with a timer which will generate uevents whenever hdmi was connected or disconnected. Now the only task that remained was to differentiate between monitor and projector. To do that we started reading about mailbox communication mechanism. We found that there are 2 mailboxes mailbox 0 and mailbox 1. The mailbox 0 is used by arm to read data from gpu and the mailbox 1 is used by arm to send data to gpu. Since we wanted to get hdmi data from the gpu[4] we had to use only mailbox 0. We read the documentation for mailbox[5] in the linux source code. From that documentation we realized that to communicate with mailbox we had to use `linux/mailbox_client.h` file to read the edid property. After more research we found this file[6] which was extending `mailbox_client` to communicate with gpu. So to get the edid property we used these APIs. We referred this site[7] to create the packet for receiving the edid data. We created a module which will just print the edid data to check if we were able to access the edid data through mailbox. After few tries we were able to get the edid data. Before creating the module we had used tvservice to monitor the hdmi events. We used -d option with the tvservice to get to edid data from the gpu in a file. The tool edidparser can be used to convert this raw data into human readable format. We compared our data from the module to this data and it was the same. Figure 3 below shows the output of the edidparser tool.

Figure 3: Output of edidparser tool



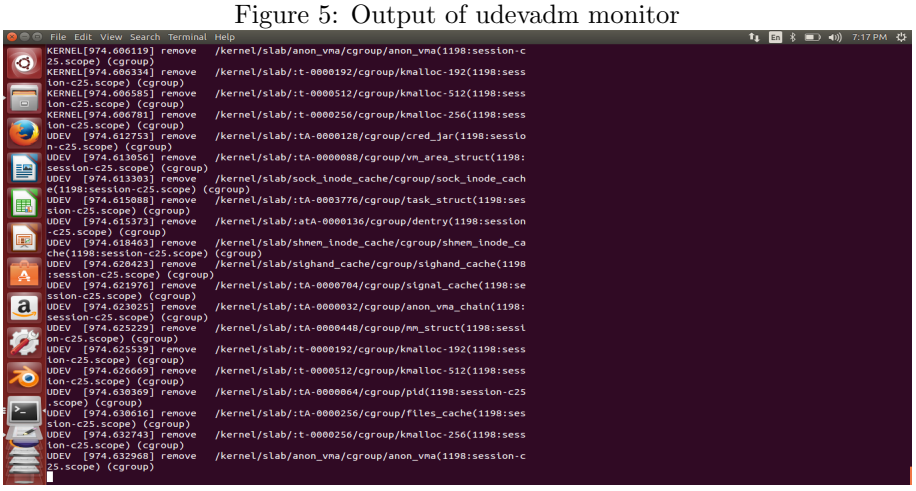
Next we saw that there was a tag 0xfc after which the monitor name was being printed. We were not able to find any way of differentiating between monitor and printer except for the name tag. One more thing that we observed that initially if we do not connect the hdmi and check this value everything was zero. Once we connected the hdmi cable we could see the data for the monitor so we thought that it would again become zero after disconnecting the hdmi, but it was not so we were getting the same data even after disconnecting. We were not able to see any changes in the data to detect the hdmi status. Therefore we thought that we will use our old module to detect any changes in gpio46 and whenever there is any change in hdmi status we will get the edid data from the mailbox. From the data received we our checking for the name tag and extracting the name of the device connected on the hdmi. If this device matches the name of the projector then only we will send out the change event.

Finally we created a module to poll the gpio46 and if any state change is detected we are getting the name of the device and comparing it with the projector name. If it matches the name of the projector we are sending the uevent change. In our udev rule we are checking for our device name and its state attribute. In the Figure 4 we can see the udev rule used to call script2.sh. If the state attribute is disconnec we our running a script which will echo the hex value of the key power to the send attribute of the modified lirc driver. The Figure 5 below shows the output from udevadm monitor, we can see the uevents from the kernel.

Figure 4: Figure showing udev rule and script to be called

```
pi@raspberrypi:/etc/udev/rules.d $ cat 90-my.rules
KERNEL=="my_hdmi", SUBSYSTEM=="platform",ACTION=="change" ATTR{state}=="disconne
c",RUN+="/home/pi/ddd/script2.sh"
pi@raspberrypi:/etc/udev/rules.d $ cd ~/ddd/
pi@raspberrypi:~/ddd $ cat script2.sh
#!/bin/bash
cd /sys/devices/platform/lirc_rpi/
sudo su
echo 0x40040011240BCEF > send
pi@raspberrypi:~/ddd $
```

Figure 5: Output of udevadm monitor



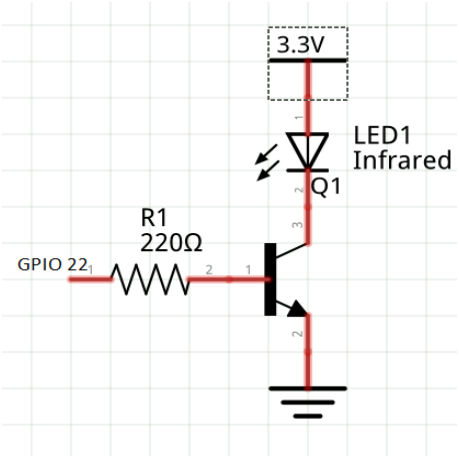
Hardware used

- Raspberry PI
- IR Transmitter
- TSOP
- Resistance 220ohms
- Transistor 2N3904

Circuit Diagram

The Figure 6 below shows the circuit used for the transmitter part.

Figure 6: Transmitter Circuit Diagram



Conclusion

We were able to achieve our objective even after facing lots of challenges. We were able to complete the project using the various things that we had learned in the device driver course. Overall the project exposed us to a variety of new things allowing us to learn them.

Future Scope

For now we have hardcoded the values of header pulse, space, pulses and spaces for 0's and 1's in our lirc module but later we can add attributes to show and store the values from sysfs so that our module will much more flexible. Moreover for the second module we are reading the name of the device connected and comparing it with the name of the class projector, we can add an attribute to take the name of the projector from sysfs itself. Also we were not able to get the status of the hdmi nor the type of device connected from the edid data, like monitor or projector, if somehow we can get this information we wont have to compare the name of the projector.

References

[1] Setting up lirc on the raspberrypi. `http://alexba.in/blog/2013/01/06/setting-up-lirc-on-the-raspberrypi/`.

[2] `https://elixir.free-electrons.com/linux/v4.9.44/source/arch/arm/boot/dts/bcm2835-rpi-b-plus.dts`.

[3] `http://elixir.free-electrons.com/linux/v4.9.44/source/drivers/leds/led-triggers.c`.

[4] `https://msreekan.com/2016/09/13/raspberry-pi2-arm-gpu-ipc/`.

[5] https://elixir.free-electrons.com/linux/v4.9.44/source/Documentation/mailbox.txt.

[6] https://elixir.free-electrons.com/linux/v4.6/source/drivers/firmware/raspberrypi.c.

[7] https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface.