
DVB-T2
Pad TV Tuner
HD809

Introduction

TV receiver cards, and later USB receiver sticks have been around since the late 90's. For the most part they have been targeted at the Windows platform and are usually bundled with driver software, and sometimes applications for playing and/or recording.

Very rarely have these devices been supported for other operating systems such as Linux - and in most cases the manufacturers do not release sufficient information to allow the creation of independent drivers. The possibility of using these devices with other platforms therefore often depends upon successful reverse engineering of the Windows drivers.

The HD809 is a small adapter which plugs directly into the micro USB connector on an Android tablet. Mine came with the recommendation to install the Cidana DVB-T2 app from the Google Play store. And apart from needing an external antenna connected, in my location the device worked pretty much as advertised. The only annoyance was that the antenna was too effective and the scanning process managed to find 4 copies of some multiplexes from different transmitters. This meant that the channel list also included four copies of many channels.

Plugging the adapter into most Windows or Linux PCs is a little difficult since they tend to only have full sized USB sockets and adapters are hard to come by. The Raspberry Pi zero on the other hand does have a microUSB connector although the adapter does then block the usual power connector and the performance is somewhat marginal since processor usage tends to 90% when streaming an HD channel.

More importantly though, with a standard build of Raspbian, the adaptor is not recognised. (also the same problem with Windows PCs). A search of the internet does reveal one attempt at writing a Linux driver (git clone git://git.linuxtv.org/anttip/media_tree.git/) although this needs to be compiled and loaded before it can be used. (My notes on doing this).

Once the driver had been compiled and loaded, I also installed TVHeadend as a client program to test the capabilities. (A first hiccup was because I had omitted to load a couple of firmware files but these were easily found on the driver author's github page). The good news at this point is that the adapter was recognised and showed up in TVHeadend, however an initial scan was only partially successful with only a couple of multiplexes (muxes) being found. It was however sometimes possible to stream tv channels although initially it was only SD muxes which could be persuaded to scan. Deleting the earlier results and re-plugging the adapter would often result in a couple of different muxes appearing – but always Standard Definition. A little investigation showed that the problem with HD muxes was mainly caused by errors in the TVHeadend database which omitted one mux for my local transmitter – and had also configured the remaining one as SD. After manually configuring the two HD muxes, they were no less likely to

scan successfully than the SD muxes. (However on the few occasions that HD channels would stream, there was frequent 'tearing' of the image).

Hardware

Clearly there had to be some difference between the commands being sent by the Linux driver and those sent by the Android driver. The problem was to find a way of snooping on the communications to identify the differences. The driver author had already identified that the adapter is made up of three functional chips – a lme2510c bridge chip which provides the interface to USB, an Si2157 tuner chip - which connects to the antenna and converts the signals to a low IF frequency, and an Si2168 demodulator chip which extracts the multiplexed TV signals from the analogue IF handed over by the tuner.

Control of the lme2510c is performed directly over the USB link whilst the other two chips are driven by I2C commands unpacked from the USB by the bridge chip and forwarded over two wires to the Si2168. (A small quirk here is that the I2C signal to the tuner chip is routed via the Si2168 and all commands to the tuner are 'bookended' by ‘open' and 'close' commands requesting the Si2168 to pass the signal through).

NB: There are many reports of various models of TV stick which change their chipset choice over time. My results are only valid for this particular combination of chips. Other similar adapters may use different chip combinations.

Driver Architecture

For a hardware and networks engineer, familiar primarily with Windows and bare-metal micro-controllers, the world of Linux, the Linux kernel and device drivers is (or was) almost totally unknown. I therefore had a steep learning curve. But with visits to the grandkids and friends out of bounds I had plenty of spare time to try some hacking.

The first point to note is that it is the bridge chip and its driver which control how a tv stick gets 'attached' to the Linux system. Often, a particular driver will support multiple models of TV stick, since they are all using the same type of bridge chip. Such a bridge driver contains a list of all the unique USB identifiers of the supported devices. This list is scanned and stored by the kernel during startup so that the correct driver can be launched when a USB device is plugged in. When the driver is launched, the ID is then also used by the driver to select the appropriate set of configuration options and to launch additional drivers for the tuner and demodulator chips. (A downside of this is that it makes the code quite large and complex and also means that full regression testing needs access to examples of all the devices covered - this may be why the author of lme2510c chose to write a stand-alone driver)

Trying to understand enough of the workings of the drivers to make progress takes some time. The documentation is pretty sparse and tends to assume detailed knowledge of Linux kernel and underlying Linux driver methodologies for Video, USB, I2C etc. The guidance in the Linux

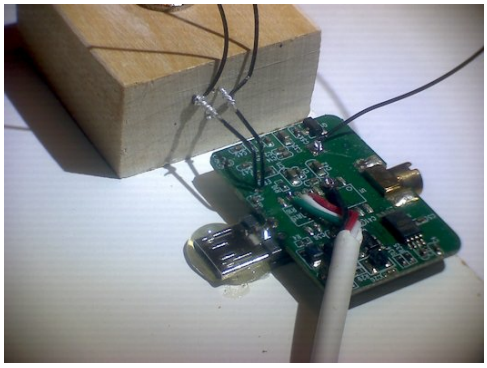
documentation is that where possible all adapters using the same bridge chip should use a common driver - and given that the lmedm04 driver already supports (presumably successfully) a range of other adapters using the lme2510c there would be a logic in extending that driver. After spending some time trying to do this however I gave up and reverted to trying to make lme2510.c work properly - along with some changes to si2157.c and si2168.c.

The Next Page outlines the diagnostic process used to resolve the initial problems

The Build Process to create a working driver for a Raspberry Pi is here.

Diagnosing The Problems

As mentioned previously, although the prototype driver achieved some level of functionality, the performance was sporadic and unpredictable. The Android application on the other hand performed very well. Ideally therefore we would like to compare the commands sent to the device by the different operating systems. However, although it is possible to record and examine the USB traffic using various tools within Linux, this does not seem to be possible with Android - at least without gaining root access which was not possible with the devices available to me.



Fortunately, quite a good level of access to the control messages can still be achieved by observing the I2C traffic being passed to the tuner and demodulator - as these receive most of the configuration commands. On this adapter, the chips have lead spacings of only 0.5mm which makes direct contact very difficult for an amateur however the I2C input connections to the Si2168 are found to be routed onto the underside of the board and appear as two pads 0.6mm in diameter and separated by about 0.8mm. The jig at the right was build to contact these two pads and then connect to a logic analyser.

In examining the circuit tracks of the adapter it also became apparent that all of the USB connections from the input connector are also routed to pads on the underside of the board - so it was easier for the investigation to use these to connect a standard USB cable rather than continue using the microUSB connector.

I had initially hoped to monitor the I2C bus by sampling using the GPIO pins of a Raspberry Pi - which is possible up to around 1 Msample/s, however it quickly became clear that the signals on this board use a 400kbit/s clock rate which requires sampling at something over 2 Msamples/s to avoid errors. I therefore bought a £10 dongle for the PC which could sample at up to 50Msamples/s. This allowed capture of many minutes of traffic at 3 Msamples/s. I then used a (very inefficient) piece of Python code to decode the 1's and 0's into I2C messages.

Startup

My first run was to compare the signals being sent after the adapter is first plugged in. In the case of Linux one can see an initial query to a small EPROM containing version information, both functional chips then receive queries for version status followed by initialisation commands - including the upload of firmware to the Si2168. The Android app uses very similar initialisation commands but uploads a more recent version of firmware - not only to the Si2168 but also to the Si2157. (It is worth noting that just using the updated firmware in the Linux system did not make

a noticeable difference to the behaviour).

Scanning

The scanning process in the Android app simply starts by sending a set of tuning instructions to the Si2157 with a frequency of 40MHz and then sending configuration instructions to the demodulator. If a signal is detected the app spends a second or two enumerating the channels found and then the process repeats with the next frequency.

My Linux TVHeadend installation on the other hand is already configured with a shortlist of 10 multiplexes and therefore attempts a similar process over fewer frequencies, sending the tune instruction, configuring the demodulator and then looking for channels. After a few runs, I managed to capture the I2C traffic for one of the failed scanning attempts and it was clear that there was a problem with the tune instruction being sent to the Si2157 - the first character or two were missing - in actual fact these were part of the instruction to open the I2C gate in the Si2168 - and was preventing the first instruction reaching the Si2157. There is clearly some issue within the lme2510c causing the loss of charactres under some undefined circumstances.

Since it was clearly only a problem with the first character or two, and only appeared to happen at the beginning of a 'tune' instruction, the easiest workaround was to send some irrelevant information prior to the important stuff. I therefore inserted an empty command into the driver file at the beginning of the overall sequence. This still triggers the creation of a pair of open/close instructions for the I2C gate in the Si2168 and seems to 'oil the wheels' for the following instructions. Compiling this change to the Si 2157 driver and copying the resultant .ko file across to the target system resolved the problem and eliminated sporadic scan failures.

Making this change and achieving reliable operation of the tuner chip did however highlight an additional problem in that the whole driver chain would seize up after one or more tuning operations and no more I2C instructions would be visible leaving the lme2510c.

Chip Configuration

A more detailed examination of the two sets of configuration commands being sent to the tuner and demodulator over I2C showed very marked differences. We then face the same problem of reverse engineering that the original author of the driver would have had. Whilst we can simply replicate a particular sequence of commands as 'seen' it is very hard to know exactly what each one means - and equally what is the significance of the replies received from the chips. Replicating a set of commands to receive a particular channel is easy - translating that into a generic algorithm which can cope will every frequency and transmission configuration is very much harder. Within TVHeadend, there are options to pass detailed configuration information to the TV adapter such as whether a mux uses DVB-T or DVB-T2, the frequency, the bandwidth, the constellation, the guardband, the transmission mode, the type of error correcting coding etc. So in practice, a reverse-engineered driver is very unlikely to accurately map all of these to the commands required to set up the chips correctly. The fact that a driver has been released

probably means that it 'seems to work ok' for the formats used by the broadcasters in the country(s) of the developers but perhaps may not function correctly in a different country. I suspect that the 'tearing' which I observed on HD channels is due to a mismatch between the broadcast format used in the UK and the ones implicit in the released driver.

Demodulator

From analysing the commands being sent to the demodulator by the Android app, one of the crucial differences from the Linux approach is that when tuning each channel, it sends one particular instruction up to 250 times in succession until a certain reply is received. This suggests that the chip is cycling through a range of pre-programmed signal formats until it finds the right one. Copying this strategy into si2168.c, along with changes to some surrounding fixed instructions, has produced a driver which can reliably decode both HD and SD channels (in the UK at least).

It should be noted that the existing Si2168 driver is also used by several other DVB drivers, and the author has no means of testing whether any changes made here will cause collateral damage to the functioning of these other drivers. All changes are therefore targeted at just making the HD809 work and no more.

Tuner

The changes in instructions to the si2157 from the Android app are rather less marked, although they also rely on a different (smaller) firmware upload. Some individual instructions are different, but the main change perhaps is in handling the settling time required after a re-tune instruction. The strategy in the published Linux driver is to listen for the usual 80h acknowledge signal after the frequency-set instruction and then wait for ~ 100mS for the PLL to settle. The strategy used by Android using the new firmware, is to keep requesting replies from the chip until an 85h reply is received - presumably indicating that lock has been achieved. This change and other command changes have been incorporated into my Si2157 driver - although it is very possible that the previous driver and firmware would actually work adequately.

Bridge Driver

After modifying the tuner and demodulator drivers, the actual reception and tuning performance was very good, however there was still the issue that the system would lock up after some random number of tuning operations. The fact that this coincided with no I2C messages being forwarded from the bridge chip indicated that the issue must be in the lme2510.c (driver)

As a first step, turning on debug messages led to a point in the code where there was a comment indicating that the author had discovered that the chip is incapable of forwarding I2c messages whilst it is streaming video. An IF statement diverts the program flow away from the usual message processing code. Interestingly, the lmedm04 driver which uses the same chip does not mention this 'feature' but does allude to the fact that USB messages need to be treated as interrupt-driven rather than the usual polling.

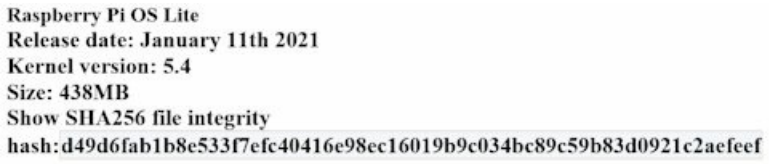
Both developers seem to have encountered some communication problems even if the conclusions are different. Looking in more detail at the IF statement, it checks the current status of the demodulator for a 'locked' condition - indicating that it has successfully demodulated a stream. However, to me it seems unlikely that any conflict of data flows in the lme2510c would depend upon the state of the demodulator. For example, unless the relevant instruction has been sent to the demodulator, the stream will not even appear on the video interface bus to the lme2510c. And even if the signal does arrive at the bridge chip it seem unlikely that just the presence of bytes at an input port would disrupt its ability to transfer messages between USB and I2C output.

To try and narrow this down I loaded the usbmon driver which enables tools such as tcpdump to copy USB traffic to a file. I then captured a few examples of the lock-up behaviour. The surprising thing was that the stream on the USB was actually being stopped before the lme2510c driver arrived at the IF staement to potentially block I2c traffic. To understand this one needs to realise that the video packets on the USB link actually represent replies to request packets sent from the stream handler. So when a TV player requests a stop or a channel change, the first action (from TvHeadend at least) is to stop requesting video packets. Any new tune request necessarily then arrives when streaming has already stopped. On the face of it, the only reason why video packets and I2C messages might collide, is if a request for performance information or similar is sent during streaming - but there is already code in the draft driver to intercept any requests for error rate information and simply repeat figures which were obtained before streaming started.

It is questionable therefore whether the original IF statement is actually needed - seemingly not with the TVHeadend client, but for the more general case we either need to detect actual streaming activity on the USB, or perhaps issue a command to always stop streaming before processing I2c commands. It is not clear how the first of these options can be implemented since the request packets on the USB do not pass through the lme2150c driver. The second is also a little difficult since we do not have a full dictionary of commands for the chip. The command to start the streaming from the lme2510c USB port is '0600h' so a reasonable developer might have allocated 0601 or 06FF as a stop command - but that is speculation. In lmedm04.c they appear to use a function beginning with 03 which aims to stop all individual channels (PIDs) from being forwarded, however this driver does not try to implement the selective channel capability. (lmedm04 currently only deals with satellite adapters so it is possible that per-channel filtering is not possible with DVB-T). As a stopgap I have implented '0601' although streaming has already stopped at this point with TVHeadend and so it is difficult to tell if it is actually capable of doing anything....

Start from scratch

Download latest compressed image file – unzip and transfer to an SD card. I went with PiOS Lite but the GUI version should work also. At the time of writing this was the latest version:



Boot the image – run sudo raspi-config – enter wifi details, turn on ssh and spi interfaces

For development I find it easier to enable the root account so that system files can be transferred directly to the destination directory over sftp – also saves frequently having to type ‘sudo’. In addition to defining a root password it is also necessary to add a line in sshd_config.

sudo passwd root

sudo nano /etc/ssh/sshd_config

add line: PermitRootLogin yes

It is essential that any new drivers that we compile are based on the exact same source code as the downloaded image. The most reliable method is to extract the kernel HASH from the running system and then find the matching source in the archives. Finding the HASH though is not as simple as listing a file but below is a piece of magic which does the job!

Enter the following three commands:

FIRMWARE_HASH=$(zgrep " firmware as of" /usr/share/doc/raspberrypi-bootloader/changelog.Debian.gz | head -1 | awk '{ print $5 }')*

KERNEL_HASH=$(wget https://raw.github.com/raspberrypi/firmware/$FIRMWARE_HASH/extra/git_hash -O -)

$KERNEL_HASH

In my case, the answer is 76c49e60e742d0bebd798be972d67dd3fd007691

Then Google 76c49e60e742d0bebd798be972d67dd3fd007691.tar.gz - or your equivalent and you should find the source code corresponding to the image on the SD card. Download ready for system compilation.

Cross compiling.

(It is advisable to find a Linux host to perform the compilation. The first compilation of the source code typically takes a couple of hours on a modern PC – trying to do it on the RPi could take days!)

Expand the source code archive into your home folder and rename the expanded folder to 'linux'.

As a first step, check that it will basically compile according to the recipe at:
https://www.raspberrypi.org/documentation/linux/kernel/building.md#choosing_sources

(Choosing the appropriate options for cross-compiling for your model of Pi.)

Adding new drivers

Assuming that the standard compilation step has gone smoothly it is time to add the new drivers and create the modules to be copied across to the running system.

In the case of the lme2510c driver, since it is a new .c file, it is necessary to tell the system that it needs compiling so a couple of files need to be updated.

The lme2510.c file should first be copied to the linux/drivers/media/usb/dvb-usb-v2/ directory

The following text should also be added to the Kconfig file in the same directory – inserted just before the entry for "config DVB_USB_LME2510".

config DVB_USB_LME2510C

tristate "Leaguer MicroElectronics LME2510C"

depends on DVB_USB_V2

select DVB_SI2168 if MEDIA_SUBDRV_AUTOSELECT

select MEDIA_TUNER_SI2157 if MEDIA_SUBDRV_AUTOSELECT

help

Say Y here to support the Leaguer MicroElectronics LME2510C.

Makefile in the same directory also needs the following text adding just before the entry for lmedm04.

dvb-usb-lme2510c-objs := lme2510c.o

obj-$(CONFIG_DVB_USB_LME2510C) += lme2510c.o

The si2157.c file in directory linux/drivers/media/tuners needs to be replaced by our new

version.

Also the file si2157_priv.h should be replaced by the updated version.

The si2168.c file in directory linux/drivers/media/dvb-frontends needs to be replaced by our new version.

Also the file si2158_priv.h should be replaced by the updated version.

The .config file in the root linux directory also needs to be updated to ensure that the new file is compiled.

Just before the line:

CONFIG_DVB_USB_LME2510=m

Add the line:

CONFIG_DVB_USB_LME2510C=m

We are now ready to repeat the system compilation which should only take a minute or two since only the new files will be compiled. The console should indicate that three .ko files have been created.

Inserting into the target system

This is strangely complicated because there are usually multiple copies of modules in a Raspbian system and it is not immediately obvious which one is the default. One way around this is to delete all copies of the module we wish to replace and then get the system to report where it would like the new one to be installed.

So, for example: (this is shown logged in as root – if not put sudo before all commands)

root@raspberrypi:~# find / -iname si2157.ko

/usr/lib/modules/5.4.83-v7l+/kernel/drivers/media/tuners/si2157.ko

/usr/lib/modules/5.4.83+/kernel/drivers/media/tuners/si2157.ko

/usr/lib/modules/5.4.83-v8+/kernel/drivers/media/tuners/si2157.ko

/usr/lib/modules/5.4.83-v7+/kernel/drivers/media/tuners/si2157.ko

We can then remove each copy:

root@raspberrypi:~# rm /usr/lib/modules/5.4.83-v7l+/kernel/drivers/media/tuners/si2157.ko

root@raspberrypi:~# rm /usr/lib/modules/5.4.83+/kernel/drivers/media/tuners/si2157.ko

root@raspberrypi:~# rm /usr/lib/modules/5.4.83-v8+/kernel/drivers/media/tuners/si2157.ko

root@raspberrypi:~# rm /usr/lib/modules/5.4.83-v7+/kernel/drivers/media/tuners/si2157.ko

Get the system to update its database:

root@raspberrypi:~# depmod

Try to load the removed module:

root@raspberrypi:~# modprobe si2157

modprobe: FATAL: Module si2157 not found in directory /lib/modules/5.4.83+

We now know where the system will be looking for the new modules.

We then need to delete all copies of si2168.ko

The new copies of the three .ko files can then be placed in the relevant directories - I tend to use sftp access but this does need a root password to have been created. Otherwise copy the files to the home directory and use sudo cp to copy them to the target directory.

Type ‘depmod’ to register the changes to the available modules

There are three .fw (firmware) files also to be transferred into /usr/lib/firmware/

Reboot with the adapter connected and check dmesg to confirm that the adapter has been recognised.

TVHeadend

Run apt-get update

Run apt-get install tvheadend

Install will ask for an admin id and password before installing the majority of the package

Once complete, use a browser on another computer to finish configuration at:

{IP Address of Rpi}:9981

When the Wizard starts, enter language preferences but then cancel the remainder as it is easier to define networks etc from the main menu.

From Configuration/DVB Inputs/TV Adapters a Silicon Labs si2168 device should be visible

Move to Networks Tab and click ‘Add’ and select DVB-T

Choose a name for the network – eg: Freeview for the Uk

Use the drop down menu to identify your local transmitter to populate the mux list

Click Create

(Make sure you have an antenna connected before continuing)

Return to TV Adapters tab and select the Si2168 DVB-T adapter – click on ‘enable’ and select your network from the drop-down menu. Click on save – the adapter status should change colour to green.

Moving to the MUX tab should show a scan underway. If not, by returning to the networks tab and selecting your network, you can click on ‘force scan’.

NB: the TVHeadend database of local transmitters contains some errors so if some multiplexes are showing a ‘fail’ it is worth checking elsewhere for the details e.g the Linux reference list at https://git.linuxtv.org/dtv-scan-tables.git/tree/dvb-t/