Quick start guide

V0.1

2019 Mar 26th Michael Yi

Introduction.

The code is dedicated and runs on Linux platform. The build environment is Ubuntu and the target hardware platform is RASP PI. The intention for the code is to make the customer with Linux platform has an easy and quick start.

It has below feature.

- Simplify – It's light-weighted. It only contains the most common interface for China customer – boot up includes option for flash boot or host boot, radio basic operation and audio DSP common usage and functions.
- Easy use – It hides the details information of chip, less but necessary parameters for the APIs. For example, the boot routine is able to identify the GUIDs from the bin automatically, programmer does not need care about the GUIDs; for the audio block API, programmer does not need aware of the inner blocks of topology.
- Convenience – It has the interface to capture SPI logs, make use a program to burn and dump the flash, which is very helpful during the system bring up.

The sample code runs under user space. Why not deploy code on kernel space?

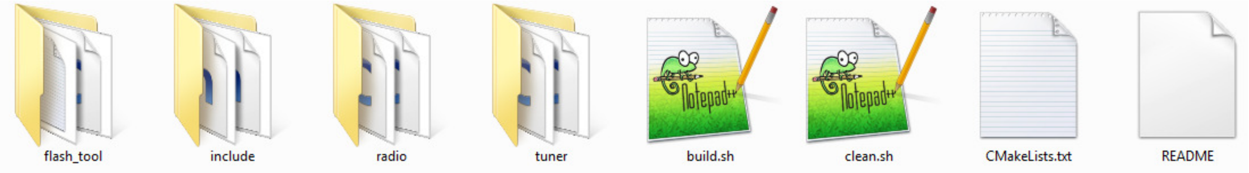
- Linux (Android) platform has different varieties of kernel. If the sample code depends on some kernel version, it might bring some extra effort from customer kernel version to sample dependent version.
- Si479x chips have radio and audio part, it contains a lot of commands inside. The code runs under driver should fast and simplify, but move too much logics to kernel space will make the driver has big size.
- Si479x has MCU inside to parse the commands sent to it. Application layer is able to take si479x as a standard SPI/I2C device. There is no necessary to move it to kernel space.
- Codes run in the application layer is easy to develop, modify and debug.

How to build

Prepare

Build environment (Ubuntu) + Linux GUN cross compiler for the target platform + CMake (sudo apt-get install cmake)

Code structure



FLASH_TOOL: Routine for dump and burn the SPI flash.

INCLUDE: The header files.

RADIO: The main routine for si479x_radio.

TUNER: The API for the si479x chips.

Build.sh: The scripts to trigger build.

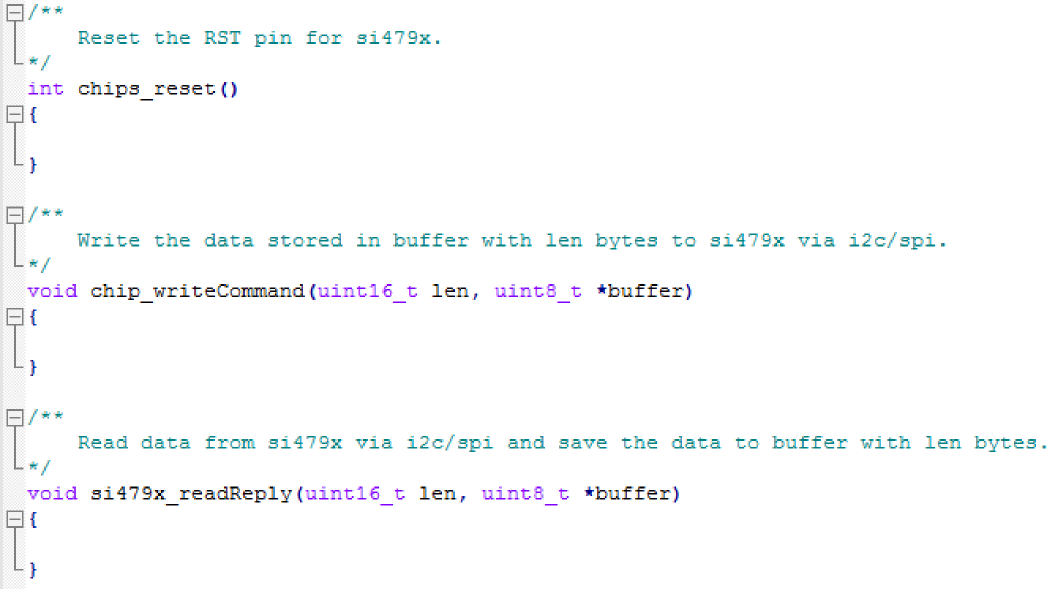
Clean.sh: The scripts to clean the output objects.

CMakeLists.txt: Used by CMake.

Modify HAL

Radio/platform/platform_hal.c

What customer need implement are below interfaces:



```
/**
    Reset the RST pin for si479x.
*/
int chips_reset()
{

}

/**
    Write the data stored in buffer with len bytes to si479x via i2c/spi.
*/
void chip_writeCommand(uint16_t len, uint8_t *buffer)
{

}

/**
    Read data from si479x via i2c/spi and save the data to buffer with len bytes.
*/
void si479x_readReply(uint16_t len, uint8_t *buffer)
{

}
```

Tips: There are available SPI codes in the samples. It needs little changes to make it work on customer platform. The sample code support 2 SPI devices for 2 chips application. It's ok to ignore unused SPI device for the single chip case.

Configure (Optional)

The sample codes are working for si4792x single chip, si4792x-si47904 dual chips and si4797x single chip application. The default configuration is for si47925. Please ignore this step for si47925 (with DTS) application; for other application, please modify configuration from dc_config.h. See configuration for details.

Build

Run ./build.sh to build the project and the output files are **bin/si479x_radio** and **bin/flash_tool**.

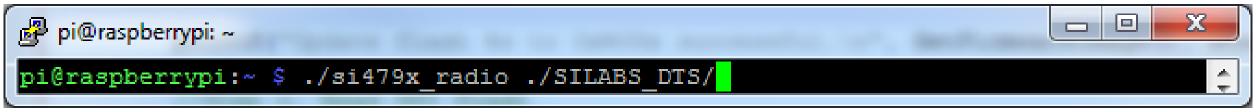
Run and Test

Host load

1. Copy si479x.bin (and key_exch.key - si47925 DTS application required) to the same destination folder, we marked it as $image for example.

Tips: Where to get the si479x.bin and flash_bl.bin. There are ones for software development under radio/firmware/si479xx. For si47925 requires DTS feature, please ask your dist.

2. Runs “./si479x_radio ./$image” to burn the si479x.bin to flash. It should be finished in 1~2 seconds.



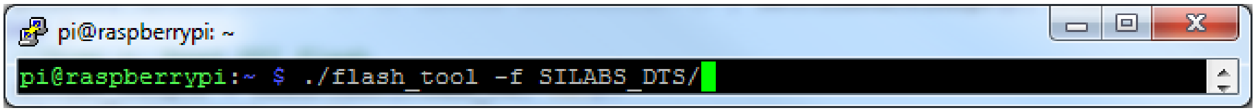
```
pi@raspberrypi:~ $ ./si479x_radio ./SILABS_DTS/
```

Tips: There are a lot of SPI logs during the loading, how to disable the SPI logs? Undefine SUPPORTS_LOGS in general_config.h.

Flash load

Update flash

1. Copy flash_bl.bin and si479x.bin (and key_exch.key - si47925 DTS application required) to some destination folder, we marked it as $image for example.
2. Runs “./flash_tool -f ./$image” to burn the si479x.bin to flash. It takes about 10 or more seconds.

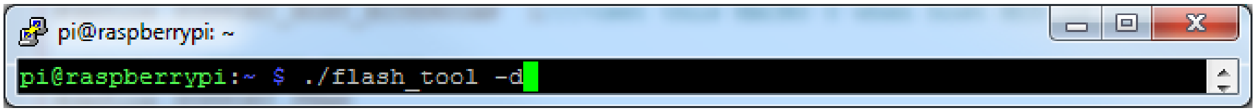


```
pi@raspberrypi:~ $ ./flash_tool -f SILABS_DTS/
```

Dump flash

Dump the si479x firmware from SPI is useful for debugging.

1. Copy flash_bl.bin to the same destination folder with flash_tool.
2. Runs “./flash_tool -d” and the firmware will be dumped to si479x.bin.

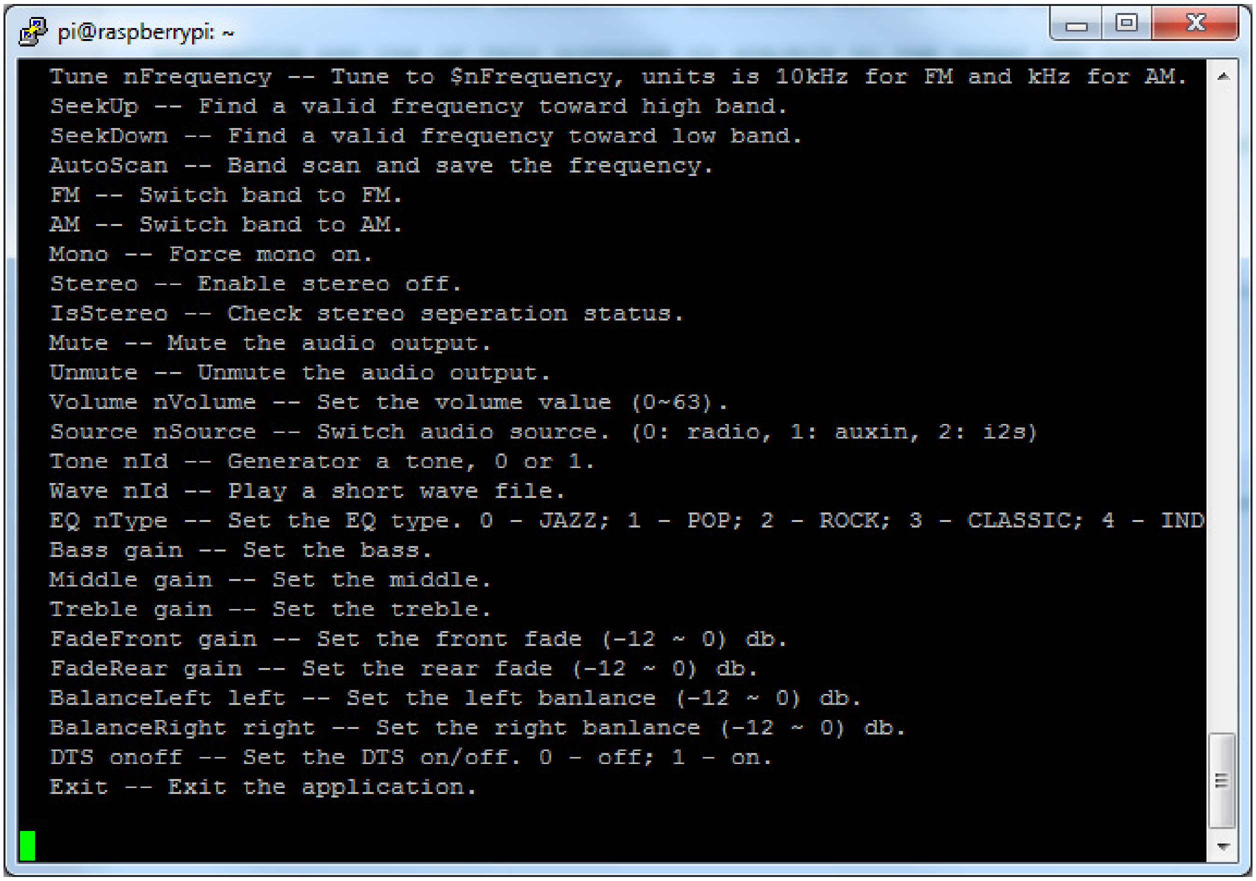


```
pi@raspberrypi:~ $ ./flash_tool -d
```

Tips: The si479x bin is updated to and dumped from address 0 on the SPI flash.

Test

In the si479x_radio main routine, it accepts and run some commands.



```
 Tune nFrequency -- Tune to $nFrequency, units is 10kHz for FM and kHz for AM.
 SeekUp -- Find a valid frequency toward high band.
 SeekDown -- Find a valid frequency toward low band.
 AutoScan -- Band scan and save the frequency.
 FM -- Switch band to FM.
 AM -- Switch band to AM.
 Mono -- Force mono on.
 Stereo -- Enable stereo off.
 IsStereo -- Check stereo seperation status.
 Mute -- Mute the audio output.
 Unmute -- Unmute the audio output.
 Volume nVolume -- Set the volume value (0~63).
 Source nSource -- Switch audio source. (0: radio, 1: auxin, 2: i2s)
 Tone nId -- Generator a tone, 0 or 1.
 Wave nId -- Play a short wave file.
 EQ nType -- Set the EQ type. 0 - JAZZ; 1 - POP; 2 - ROCK; 3 - CLASSIC; 4 - IND
 Bass gain -- Set the bass.
 Middle gain -- Set the middle.
 Treble gain -- Set the treble.
 FadeFront gain -- Set the front fade (-12 ~ 0) db.
 FadeRear gain -- Set the rear fade (-12 ~ 0) db.
 BalanceLeft left -- Set the left banlance (-12 ~ 0) db.
 BalanceRight right -- Set the right banlance (-12 ~ 0) db.
 DTS onoff -- Set the DTS on/off. 0 - off; 1 - on.
 Exit -- Exit the application.
```

Porting guide

Boot up

With below boot up sequence, it’s able to get the radio from speakers.

```
//Step a. Platform hal init, GPIO, spi/i2c init, etc.
platform_hal_init();

// Step b. Setup the firmware image location, if not set, the default directory is “./”
SetFirmwareImageFolder(argv[1]);

//Step c. reset chipsets
chips_reset();

//Step d. host load or flash load
if (bootMode == 0)
{
        //host load
        dc_hostload_bootup();
}
else
{
        //flash load
        dc_flashload_bootup();
}

//Step e. configurations post boot and tune
if (band == BAND_FM)
{
        //configure audio settings
        dc_post_bootup(BAND_FM);

        //tune some frequency
        dc_fm_tune(10430);
}
```

```
else if(band == BAND_AM)
{
        //configure audio settings
        dc_post_bootup(BAND_AM);

        //tune some frequency
        dc_am_tune(999);
}

//Step f. Initial audio manager
Audio_mgr_init();

//Step g. Initial DTS for DTS case.
#ifdef SUPPORT_DTS_CS
DTS_Init();
#endif

//Step h. switch to radio source
Audio_switch(RADIO_AUDIO);
```

API

Radio function.

*RET_CODE **dc_set_mode**(uint8_t band);*

Switch the band. 0 – FM, 1 – AM.

*RET_CODE **dc_fm_tune**(uint16_t Freq);*

Tune a FM frequency, the unit is 10 kHz.

*RET_CODE **dc_am_tune**(uint16_t Freq);*

Tune an AM frequency, the unit is 100 Hz.

*RET_CODE **dc_fm_seek**(uint8_t seekUp, uint8_t seekMode, _SEEK_PROCESS seek_callback);*

FM seek up or down, warp band limit or not, seek_callback used to notify current validated frequency.

*uint8_t **dc_fm_autoseek**(_SEEK_PROCESS seek_callback, _SEEK_FOUND found_callback);*

Auto scan the whole band. seek_callback is used to notify current validated frequency, found_callback notify the good frequency.

*void **Radio_ForceMono**();*

Force the radio to mono, disable stereo.

*void **Radio_EnableStereo**();*

Enable stereo.

*uint8_t **Radio_CheckStereo**();*

Check the stereo separation value. Non-zero value indicate it’s stereo.

Audio

General

*void **Audio_mute**(uint8_t mute);*

Mute (set mute = 1) or unmute (set mute = 0) the audio output.

*void **Audio_volume**(uint8_t volume);*

The volume range is 0 ~ 40. There is a table gains matched with the volume. Customer is able to modify it per their requirement.

*void **Audio_tonegen**(uint8_t tone_id);*

Play the tone. Customer is able to change the tone descriptors. Tone_id 0, 1 used for TONE_GEN; Tone id 2 used for WAVE_PLAYER.

*void **Audio_EQ** (uint8_t id);*

There are 5 preset EQ. Customer is able to extend it.

*void **Audio_Treble**(double gain);*

Set the treble gain for all speakers. The range is -12 to 12 by default. It's defined by AUDIO_TREBLE_MIN and AUDIO_TREBLE_MAX.

*void **Audio_Middle**(double gain);*

Set the mid gain for all speakers. The range is -12 to 12 by default. It's defined by AUDIO_MID_MIN and AUDIO_MID_MAX.

*void **Audio_Bass**(double gain);*

Set the mid gain for all speakers. The range is -12 to 12 by default. It's defined by AUDIO_BASS_MIN and AUDIO_BASS_MAX.

*void **Audio_FadeFront**(double front);*

Set the gain for LF, RF. The range is -12 to 0 by default. It's defined by AUDIO_FADE_MIN and AUDIO_FADE_MAX.

*void **Audio_FadeRear**(double rear);*

Set the gain for LR, RR. The range is -12 to 0 by default. It's defined by AUDIO_FADE_MIN and AUDIO_FADE_MAX.

*void **Audio_BalanceLeft**(double left);*

Set the gain for LF, LR. The range is -12 to 0 by default. It's defined by AUDIO_BALANCE_MIN and AUDIO_BALANCE_MAX.

*void **Audio_BalanceRight**(double right);*

Set the gain for RF, RR. The range is -12 to 0 by default. It's defined by AUDIO_BALANCE_MIN and AUDIO_BALANCE_MAX.

*void **Audio_Bypass_Delay**(int8_t onoff);*

For the audio algorithm (for example DTS) application, it will tune the delay module. It may impact bench test, use this API to disable the delays.

*void **Audio_Bypass_Cabin_EQ**(int8_t onoff);*

For the audio algorithm (for example DTS) application, it will tune the cabin EQ module for each speaker. It may impact bench test, use this API to disable the cabin EQ module.

DTS

*void **DTS_Init**();*

Initialize the DTS module.

*void **DTS_Bypass**(uint16_t onoff);*

Bypass the DTS module.

*void **DTS_Set_PROCESS_MODE**(uint16_t value);*

*void **DTS_Set_PHANTOM_CENTER**(uint16_t value);*

*void **DTS_Set_FOCUS_CENTER**(uint16_t value);*

*void **DTS_Set_FOCUS_FRONT**(uint16_t value);*

*void **DTS_Set_FOCUS_REAR**(uint16_t value);*

*void **DTS_Set_TB_FRONT**(uint16_t value);*

*void **DTS_Set_TB_SUB**(uint16_t value);*

*void **DTS_Set_TB_REAR**(uint16_t value);*

*void **DTS_Set_TB_FRONT_SPKSZ**(uint16_t value);*

*void **DTS_Set_TB_SUB_SPKSZ**(uint16_t value);*

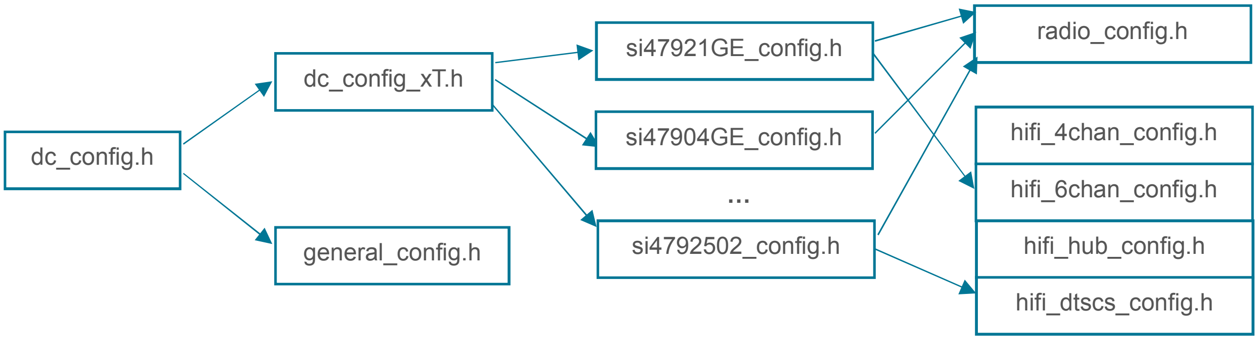
*void **DTS_Set_TB_REAR_SPKSZ**(uint16_t value);*

Set the DTS module parameters.

Configuration

Why need configurations?

Although settings are variable and can be set through parameter, but for each project, a lot of settings are fixed. So for these setting, we define it as MACRO and hide it inside the API. All the MACRO are in the folder include/config.



The configuration tree is as above chart. Some useful and more frequently used MACRO as listed as below.

general_config.h

#define ***SUPPORTS_LOGS***

Enable this to open print SPI logs, disable it when it's not necessary.

Radio_config.h

#define ***FM_SPACE*** *10*

#define ***FM_BOT_FREQ*** *8750*

#define ***FM_TOP_FREQ*** *10800*

Hifi_xxx_config.h

#define ***AUDIO_BASS_MAX*** *12*

#define ***AUDIO_BASS_MAX*** *-12*

#define ***AUDIO_TREBLE_MAX*** *12*

#define ***AUDIO_TREBLE_MIN*** *-12*

#define ***AUDIO_MID_MAX*** *12*

#define ***AUDIO_MID_MIN*** *-12*

The audio gain limitation for bass, treble, middle.

There are also a lot of MACROs for the module id for the topology in hifi_xxx_config.h. Different topologies have their own hifi_xxx_config.h.

Bring up skills.

1. Make sure the power and hardware connection is good – checks VA, VD, VIO1/2 and crystal is connected well.
2. Check the RSTB pin works as expected.
3. Check SPI and i2c bus driver. For i2c bus, use the correct i2c address. After reset the chip, the chip should reply [0x80, 0x00, 0x00, 0x00]. If not, there may be some hardware or driver itself issue.
4. Capture the SPI logs and send to Silicon Labs for some help if necessary.