

Si4735 Arduino Library

AUTHOR
Version 1.1.8
02/04/2020

Table of Contents

Table of contents

Deprecated List

Global [SI4735::analogPowerUp](#) (void)

Consider use radioPowerUp instead

Module Index

Modules

Here is a list of all modules:

Deal with Interrupt.....	1
Deal with Interrupt and I2C bus.....	1
Host and slave MCU setup.....	46
RDS Data types.....	49
Receiver Status and Setup.....	56
SI473X data types.....	59
SI47XX device Mode, Band and Frequency setup.....	65
SI47XX device information and start up.....	70

File Index

File List

Here is a list of all files with brief descriptions:

SI4735/SI4735.cpp	72
SI4735/SI4735.h	72

Module Documentation

Deal with Interrupt

Detailed Description

Deal with Interrupt

Deal with Interrupt and I2C bus

Data Structures

class [SI4735](#)

[SI4735](#) Class. [More...](#)

Functions

[SI4735::SI4735](#) ()

Construct a new [SI4735::SI4735](#) object.

void [SI4735::waitInterrupt](#) (void)

Interrupt handle.

int16_t [SI4735::getDeviceI2CAddress](#) (uint8_t [resetPin](#))

I2C bus address setup.

void [SI4735::setDeviceI2CAddress](#) (uint8_t [senPin](#))

Sets the I2C Bus Address.

void [SI4735::setDeviceOtherI2CAddress](#) (uint8_t [i2cAddr](#))

Sets the onther I2C Bus Address (for Si470X)

Detailed Description

This is a library for the [SI4735](#), BROADCAST AM/FM/SW RADIO RECEIVER, IC from Silicon Labs for the Arduino development environment. It works with I2C protocol. This library is intended to provide an easier interface for controlling the [SI4735](#).

See also

documentation on <https://github.com/pu2clr/SI4735>.

Si47XX PROGRAMMING GUIDE; AN332

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; AMENDMENT FOR SI4735-D60
SSB AND NBFM PATCHES

ATTENTION: According to Si47XX PROGRAMMING GUIDE; AN332; page 207, "For write operations, the system controller next sends a data byte on SDIO, which is captured by the device on rising edges of SCLK. The device acknowledges each data byte by driving SDIO low for one cycle on the next falling edge of SCLK. The system controller may write up to 8 data bytes in a single 2-wire transaction. The first byte is a command, and the next seven bytes are arguments. Writing more than 8 bytes results in unpredictable device behavior". So, If you are extending this library, consider that restriction presented earlier.

ATTENTION: Some methods were implemented usin inline resource. Inline methods are implemented in [SI4735.h](#)

Author

PU2CLR - Ricardo Lima Caratti

By Ricardo Lima Caratti, Nov 2019.

Data Structure Documentation

class SI4735

[SI4735](#) Class.

[SI4735](#) Class definition

This class implements all functions to help you to control the Si47XX devices. This library was built based on “Si47XX PROGRAMMING GUIDE; AN332 ”. It also can be used on all members of the SI473X family respecting, of course, the features available for each IC version. These functionalities can be seen in the comparison matrix shown in table 1 (Product Family Function); pages 2 and 3 of the programming guide.

Author

PU2CLR - Ricardo Lima Caratti

Public Member Functions

[SI4735](#) ()

Construct a new [SI4735::SI4735](#) object.

void [reset](#) (void)

Reset the SI473X

void [waitToSend](#) (void)

Wait for the si473x is ready (Clear to Send (CTS) status bit have to be 1).

void [setup](#) (uint8_t [resetPin](#), uint8_t defaultFunction)

Starts the SI473X device.

void [setup](#) (uint8_t [resetPin](#), int [interruptPin](#), uint8_t defaultFunction, uint8_t audioMode=[SI473X_ANALOG_AUDIO](#))

Starts the SI473X device.

void [setPowerUp](#) (uint8_t CTSIEN, uint8_t GPO2OEN, uint8_t PATCH, uint8_t XOSCEN, uint8_t FUNC, uint8_t OPMODE)

Set the Power Up parameters for si473X.

void [radioPowerUp](#) (void)

Powerup the Si47XX.

void [analogPowerUp](#) (void)

You have to call setPowerUp method before.

void [powerDown](#) (void)

Moves the device from powerup to powerdown mode.

void [setFrequency](#) (uint16_t)

Set the frequency to the corrent function of the Si4735 (FM, AM or SSB)

void [getStatus](#) ()

void [getStatus](#) (uint8_t, uint8_t)

uint16_t [getFrequency](#) (void)

uint16_t [getCurrentFrequency](#) ()
 bool [getSignalQualityInterrupt](#) ()
 bool [getRadioDataSystemInterrupt](#) ()
 Gets Received Signal Quality Interrupt(RSQINT)

bool [getTuneCompleteTriggered](#) ()
 Gets Radio Data System (RDS) Interrupt.

bool [getStatusError](#) ()
 Seek/Tune Complete Interrupt; 1 = Tune complete has been triggered.

bool [getStatusCTS](#) ()
 Return the Error flag (true or false) of status of the least Tune or Seek.

bool [getACFIndicator](#) ()
 Gets the Error flag of status response.

bool [getBandLimit](#) ()
 Returns true if the AFC rails (AFC Rail Indicator).

bool [getStatusValid](#) ()
 Returns true if a seek hit the band limit (WRAP = 0 in FM_START_SEEK) or wrapped to the original frequency(WRAP = 1).

uint8_t [getReceivedSignalStrengthIndicator](#) ()
 Returns true if the channel is currently valid as determined by the seek/tune properties (0x1403, 0x1404, 0x1108)

uint8_t [getStatusSNR](#) ()
 Returns integer Received Signal Strength Indicator ($dB\hat{I}^{1/4}V$).

uint8_t [getStatusMULT](#) ()
 Returns integer containing the SNR metric when tune is complete (dB).

uint8_t [getAntennaTuningCapacitor](#) ()
 Returns integer containing the multipath metric when tune is complete.

void [getAutomaticGainControl](#) ()
 Returns integer containing the current antenna tuning capacitor value.

void [setAvcAmMaxGain](#) (uint8_t gain)
 void [setAvcAmMaxGain](#) ()
 uint8_t [getCurrentAvcAmMaxGain](#) ()
 void [setAmSoftMuteMaxAttenuation](#) (uint8_t smattn)
 void [setAmSoftMuteMaxAttenuation](#) ()
 void [setSsbSoftMuteMaxAttenuation](#) (uint8_t smattn)
 void [setSsbSoftMuteMaxAttenuation](#) ()
 bool [isAgcEnabled](#) ()
 uint8_t [getAgcGainIndex](#) ()

void [setAutomaticGainControl](#) (uint8_t AGCDIS, uint8_t AGCIDX)
 void [getCurrentReceivedSignalQuality](#) (uint8_t INTACK)
 void [getCurrentReceivedSignalQuality](#) (void)
 uint8_t [getCurrentRSSI](#) ()
 uint8_t [getCurrentSNR](#) ()
current receive signal strength (0â€‘127 dB $\frac{1}{4}$ V).

bool [getCurrentRssiDetectLow](#) ()
current SNR metric (0–127 dB).

bool [getCurrentRssiDetectHigh](#) ()
RSSI Detect Low.

bool [getCurrentSnrDetectLow](#) ()
RSSI Detect High.

bool [getCurrentSnrDetectHigh](#) ()
SNR Detect Low.

bool [getCurrentValidChannel](#) ()
SNR Detect High.

bool [getCurrentAfcRailIndicator](#) ()
Valid Channel.

bool [getCurrentSoftMuteIndicator](#) ()
AFC Rail Indicator.

uint8_t [getCurrentStereoBlend](#) ()
Soft Mute Indicator. Indicates soft mute is engaged.

bool [getCurrentPilot](#) ()
Indicates amount of stereo blend in % (100 = full stereo, 0 = full mono).

uint8_t [getCurrentMultipath](#) ()
Indicates stereo pilot presence.

uint8_t [getCurrentSignedFrequencyOffset](#) ()
Contains the current multipath metric. (0 = no multipath; 100 = full multipath)

bool [getCurrentMultipathDetectLow](#) ()
Signed frequency offset (kHz).

bool [getCurrentMultipathDetectHigh](#) ()
Multipath Detect Low.

bool [getCurrentBlendDetectInterrupt](#) ()

Multipath Detect High.

uint8_t [getFirmwarePN](#) ()
Blend Detect Interrupt.

uint8_t [getFirmwareFWMAJOR](#) ()
RESP1 - Part Number (HEX)

uint8_t [getFirmwareFWMINOR](#) ()
RESP2 - Returns the Firmware Major Revision (ASCII).

uint8_t [getFirmwarePATCHH](#) ()
RESP3 - Returns the Firmware Minor Revision (ASCII).

uint8_t [getFirmwarePATCHL](#) ()
RESP4 - Returns the Patch ID High byte (HEX).

uint8_t [getFirmwareCMPMAJOR](#) ()
RESP5 - Returns the Patch ID Low byte (HEX).

uint8_t [getFirmwareCMPMINOR](#) ()
RESP6 - Returns the Component Major Revision (ASCII).

uint8_t [getFirmwareCHIPREV](#) ()
RESP7 - Returns the Component Minor Revision (ASCII).

void [setVolume](#) (uint8_t [volume](#))
RESP8 - Returns the Chip Revision (ASCII).

uint8_t [getVolume](#) ()
void [volumeDown](#) ()
void [volumeUp](#) ()
uint8_t [getCurrentVolume](#) ()
void [setAudioMute](#) (bool off)
Returns the current volume level.

void [digitalOutputFormat](#) (uint8_t OSIZE, uint8_t OMONO, uint8_t OMODE, uint8_t OFALL)
void [digitalOutputSampleRate](#) (uint16_t DOSR)
void [setAM](#) ()
Sets the radio to AM function. It means: LW MW and SW.

void [setFM](#) ()
Sets the radio to FM function.

void [setAM](#) (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)
Sets the radio to AM (LW/MW/SW) function.

void [setFM](#) (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)

Sets the radio to FM function.

void [setBandwidth](#) (uint8_t AMCHFLT, uint8_t AMPLFLT)

void [setFrequencyStep](#) (uint16_t step)

Sets the current step value.

uint8_t [getTuneFrequencyFast](#) ()

void [setTuneFrequencyFast](#) (uint8_t FAST)

Returns the FAST tuning status.

uint8_t [getTuneFrequencyFreeze](#) ()

FAST Tuning. If set, executes fast and invalidated tune. The tune status will not be accurate.

void [setTuneFrequencyFreeze](#) (uint8_t FREEZE)

Returns the FREEZE status.

void [setTuneFrequencyAntennaCapacitor](#) (uint16_t capacitor)

Only FM. Freeze Metrics During Alternate Frequency Jump.

void [frequencyUp](#) ()

Increments the current frequency on current band/function by using the current step.

void [frequencyDown](#) ()

Decrements the current frequency on current band/function by using the current step.

bool [isCurrentTuneFM](#) ()

void [getFirmware](#) (void)

Gets firmware information.

void [setFunction](#) (uint8_t FUNC)

void [seekStation](#) (uint8_t SEEKUP, uint8_t WRAP)

void [seekStationUp](#) ()

void [seekStationDown](#) ()

void [setSeekAmLimits](#) (uint16_t bottom, uint16_t top)

void [setSeekAmSpacing](#) (uint16_t spacing)

void [setSeekSrnThreshold](#) (uint16_t value)

void [setSeekRssiThreshold](#) (uint16_t value)

void [setFmBlendStereoThreshold](#) (uint8_t parameter)

void [setFmBlendMonoThreshold](#) (uint8_t parameter)

void [setFmBlendRssiStereoThreshold](#) (uint8_t parameter)

void [setFmBlendRssiMonoThreshold](#) (uint8_t parameter)

void [setFmBlendSnrStereoThreshold](#) (uint8_t parameter)

void [setFmBlendSnrMonoThreshold](#) (uint8_t parameter)

void [setFmBlendMultiPathStereoThreshold](#) (uint8_t parameter)

void [setFmBlendMultiPathMonoThreshold](#) (uint8_t parameter)

void [setFmStereoOn](#) ()

void [setFmStereoOff](#) ()

void [RdsInit](#) ()

void [setRdsIntSource](#) (uint8_t RDSNEWBLOCKB, uint8_t RDSNEWBLOCKA, uint8_t RDSSYNCFFOUND, uint8_t RDSSYNCLST, uint8_t RDSRECV)

void [getRdsStatus](#) (uint8_t INTACK, uint8_t MTFIFO, uint8_t STATUSONLY)

```

void getRdsStatus ()
bool getRdsReceived ()
bool getRdsSyncLost ()
    1 = FIFO filled to minimum number of groups

bool getRdsSyncFound ()
    1 = Lost RDS synchronization

bool getRdsNewBlockA ()
    1 = Found RDS synchronization

bool getRdsNewBlockB ()
    1 = Valid Block A data has been received.

bool getRdsSync ()
    1 = Valid Block B data has been received.

bool getGroupLost ()
    1 = RDS currently synchronized.

uint8_t getNumRdsFifoUsed ()
    1 = One or more RDS groups discarded due to FIFO overrun.

void setRdsConfig (uint8_t RDSSEN, uint8_t BLETHA, uint8_t BLETHB, uint8_t BLETHC, uint8_t
    BLETHD)
    RESP3 - RDS FIFO Used; Number of groups remaining in the RDS FIFO (0 if empty).

uint16_t getRdsPI (void)
uint8_t getRdsGroupType (void)
uint8_t getRdsFlagAB (void)
uint8_t getRdsVersionCode (void)
uint8_t getRdsProgramType (void)
uint8_t getRdsTextSegmentAddress (void)
char * getRdsText (void)
char * getRdsText0A (void)
char * getRdsText2A (void)
char * getRdsText2B (void)
char * getRdsTime (void)
void getNext2Block (char *)
void getNext4Block (char *)
void ssbSetup ()
void setSSBBfo (int offset)
void setSSBConfig (uint8_t AUDIOBW, uint8_t SBCUTFLT, uint8_t AVC_DIVIDER, uint8_t
    AVCEN, uint8_t SMUTESEL, uint8_t DSP_AFCDIS)
void setSSB (uint16_t fromFreq, uint16_t toFreq, uint16_t intialFreq, uint16_t step, uint8_t usblsb)
void setSSB (uint8_t usblsb)
void setSSBAudioBandwidth (uint8_t AUDIOBW)
void setSSBAutomaticVolumeControl (uint8_t AVCEN)
void setSSBSidebandCutoffFilter (uint8_t SBCUTFLT)
void setSSBAvcDivider (uint8_t AVC_DIVIDER)
void setSSBDspAfc (uint8_t DSP_AFCDIS)
void setSSBSoftMute (uint8_t SMUTESEL)
si47x\_firmware\_query\_library\_queryLibraryId ()

```

```

void patchPowerUp ()
bool downloadPatch (const uint8_t *ssb_patch_content, const uint16_t ssb_patch_content_size)
bool downloadPatch (int eeprom_i2c_address)
void ssbPowerUp ()
void setI2CLowSpeedMode (void)
void setI2CStandardMode (void)
    Sets I2C buss to 10KHz.

void setI2CFastMode (void)
    Sets I2C buss to 100KHz.

void setI2CFastModeCustom (long value=500000)
    Sets I2C buss to 400KHz.

void setDeviceI2CAddress (uint8_t senPin)
    Sets the I2C Bus Address.

int16_t getDeviceI2CAddress (uint8_t resetPin)
    I2C bus address setup.

void setDeviceOtherI2CAddress (uint8_t i2cAddr)
    Sets the onther I2C Bus Address (for Si470X)

```

Protected Member Functions

```

void waitInterrupr (void)
    Interrupt handle.

void sendProperty (uint16_t propertyValue, uint16_t param)
void sendSSBModeProperty ()
void disableFmDebug ()
void clearRdsBuffer2A ()
void clearRdsBuffer2B ()
void clearRdsBuffer0A ()

```

Protected Attributes

```

char rds\_buffer2A [65]
char rds\_buffer2B [33]
    RDS Radio Text buffer - Program Information.

char rds\_buffer0A [9]
    RDS Radio Text buffer - Station Informaation.

char rds\_time [20]
    RDS Basic tuning and switching information (Type 0 groups)

int rdsTextAdress2A
    RDS date time received information

int rdsTextAdress2B

```

rds_buffer2A current position

int [rdsTextAddress0A](#)

rds_buffer2B current position

int16_t [deviceAddress](#) = [SI473X_ADDR_SEN_LOW](#)

rds_buffer0A current position

uint8_t [lastTextFlagAB](#)

current I2C buss address

uint8_t [resetPin](#)

uint8_t [interruptPin](#)

pin used on Arduino Board to RESET the Si47XX device

uint8_t [currentTune](#)

pin used on Arduino Board to control interrupt. If -1, interrupt is no used.

uint16_t [currentMinimumFrequency](#)

tell the current tune (FM, AM or SSB)

uint16_t [currentMaximumFrequency](#)

minimum frequency of the current band

uint16_t [currentWorkFrequency](#)

maximum frequency of the current band

uint16_t [currentStep](#)

current frequency

uint8_t [lastMode](#) = -1

current steps

uint8_t [currentAvcAmMaxGain](#) = 48

Store the last mode used.

[si47x_frequency](#) [currentFrequency](#)

Automatic Volume Control Gain for AM - Default 48.

[si47x_set_frequency](#) [currentFrequencyParams](#)

data structure to get current frequency

[si47x_rqs_status](#) [currentRqsStatus](#)

[si47x_response_status](#) [currentStatus](#)

current Radio Signal Quality status

[si47x_firmware_information](#) [firmwareInfo](#)

current device status

[si47x_rds_status](#) [currentRdsStatus](#)

firmware information

[si47x_agc_status](#) [currentAgcStatus](#)

current RDS status

[si47x_ssb_mode](#) [currentSSBMode](#)

current AGC status

[si473x_powerup](#) [powerUp](#)

indicates if USB or LSB

uint8_t [volume](#) = 32

uint8_t [currentSsbStatus](#)

Member Function Documentation

void SI4735::clearRdsBuffer0A () [protected]

Clear RDS buffer 0A (text)

```
01308 {  
01309     for (int i = 0; i < 9; i++)  
01310         rds\_buffer0A[i] = ' '; // Station Name buffer  
01311 }
```

References [rds_buffer0A](#).

Referenced by [getRdsStatus\(\)](#), and [RdsInit\(\)](#).

void SI4735::clearRdsBuffer2A () [protected]

Clear RDS buffer 2A (text)

```
01289 {  
01290     for (int i = 0; i < 65; i++)  
01291         rds\_buffer2A[i] = ' '; // Radio Text buffer - Program Information  
01292 }
```

References [rds_buffer2A](#).

Referenced by [getRdsStatus\(\)](#), and [RdsInit\(\)](#).

void SI4735::clearRdsBuffer2B () [protected]

Clear RDS buffer 2B (text)

```
01299 {  
01300     for (int i = 0; i < 33; i++)  
01301         rds\_buffer2B[i] = ' '; // Radio Text buffer - Station Informaation  
01302 }
```

References [rds_buffer2B](#).

Referenced by [getRdsStatus\(\)](#), and [RdsInit\(\)](#).

void SI4735::digitalOutputFormat (uint8_t OSIZE, uint8_t OMONO, uint8_t OMODE, uint8_t OFALL)

Digital Audio Setup Configures the digital audio output format. Options: DCLK edge, data format, force mono, and sample precision.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 195.

Parameters

uint8_t	OSIZE Digital Output Audio Sample Precision (0=16 bits, 1=20 bits, 2=24
-------------------------	---

	bits, 3=8bits).
<i>uint8_t</i>	OMONO Digital Output Mono Mode (0=Use mono/stereo blend).
<i>uint8_t</i>	OMODE Digital Output Mode (0=I2S, 6 = Left-justified, 8 = MSB at second DCLK after DFS pulse, 12 = MSB at first DCLK after DFS pulse).
<i>uint8_t</i>	OFALL Digital Output DCLK Edge (0 = use DCLK rising edge, 1 = use DCLK falling edge)

```

00853 {
00854     si4735\_digital\_output\_format df;
00855     df.refined.OSIZE = OSIZE;
00856     df.refined.OMONO = OMONO;
00857     df.refined.OMODE = OMODE;
00858     df.refined.OFALL = OFALL;
00859     sendProperty(DIGITAL\_OUTPUT\_FORMAT, df.raw);
00860 }

```

void SI4735::digitalOutputSampleRate (uint16_t DOSR)

Enables digital audio output and configures digital audio output sample rate in samples per second (sps).

See also

Si47XX PROGRAMMING GUIDE; AN332; page 196.

Parameters

<i>uint16_t</i>	DOSR Digital Output Sample Rate(32–48 ksps .0 to disable digital audio output).
-----------------	---

```

00870 {
00871     sendProperty(DIGITAL\_OUTPUT\_SAMPLE\_RATE, DOSR);
00872 }

```

void SI4735::disableFmDebug () [protected]

There is a debug feature that remains active in Si4704/05/3x-D60 firmware which can create periodic noise in audio. Silicon Labs recommends you disable this feature by sending the following bytes (shown here in hexadecimal form): 0x12 0x00 0xFF 0x00 0x00 0x00.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 299.

```

00825 {
00826     Wire.beginTransaction(deviceAddress);
00827     Wire.write(0x12);
00828     Wire.write(0x00);
00829     Wire.write(0xFF);
00830     Wire.write(0x00);
00831     Wire.write(0x00);
00832     Wire.write(0x00);
00833     Wire.endTransmission();
00834     delayMicroseconds(2500);
00835 }

```

Referenced by setFM().

bool SI4735::downloadPatch (const uint8_t * *ssb_patch_content*, const uint16_t *ssb_patch_content_size*)

Transfers the content of a patch stored in a array of bytes to the [SI4735](#) device. You must mount an array as shown below and know the size of that array as well.

It is importante to say that patches to the [SI4735](#) are distributed in binary form and have to be transferred to the internal RAM of the device by the host MCU (in this case Arduino). Since the RAM is volatile memory, the patch stored into the device gets lost when you turn off the system. Consequently, the content of the patch has to be transferred again to the device each time after turn on the system or reset the device.

The disadvantage of this approach is the amount of memory used by the patch content. This may limit the use of other radio functions you want implemented in Arduino.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 64 and 215-220.

Example of content: `const PROGMEM uint8_t ssb_patch_content_full[] = { // SSB patch for whole SSBRX full download 0x15, 0x00, 0x0F, 0xE0, 0xF2, 0x73, 0x76, 0x2F, 0x16, 0x6F, 0x26, 0x1E, 0x00, 0x4B, 0x2C, 0x58, 0x16, 0xA3, 0x74, 0x0F, 0xE0, 0x4C, 0x36, 0xE4, 0x16, 0x3B, 0x1D, 0x4A, 0xEC, 0x36, 0x28, 0xB7, 0x16, 0x00, 0x3A, 0x47, 0x37, 0x00, 0x00, 0x00, 0x15, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9D, 0x29};`

`const int size_content_full = sizeof ssb_patch_content_full;`

Parameters

<code>ssb_patch_content</code>	point to array of bytes content patch.
<code>ssb_patch_content_size</code>	array size (number of bytes). The maximum size allowed for a patch is 15856 bytes

Returns

false if an error is found.

```
02245 {
02246     uint8_t content;
02247     register int i, offset;
02248     // Send patch to the SI4735 device
02249     for (offset = 0; offset < (int) ssb_patch_content_size; offset += 8)
02250     {
02251         Wire.beginTransaction(deviceAddress);
02252         for (i = 0; i < 8; i++)
02253         {
02254             content = pgm_read_byte_near(ssb_patch_content + (i + offset));
02255             Wire.write(content);
02256         }
02257         Wire.endTransmission();
02258
02259         // Testing download performance
02260         // approach 1 - Faster - less secure (it might crash in some
architectures)
02261         delayMicroseconds(MIN\_DELAY\_WAIT\_SEND\_LOOP); // Need check the
minimum value
02262
02263         // approach 2 - More control. A little more secure than approach 1
02264         /*
02265         do
02266         {
02267             delayMicroseconds(150); // Minimum delay founded (Need check the
minimum value)
02268             Wire.requestFrom(deviceAddress, 1);
02269             } while (!(Wire.read() & B10000000));
02270         */
02271
02272         // approach 3 - same approach 2
02273         // waitToSend();
02274
02275         // approach 4 - safer
02276         /*
02277         waitToSend();
02278         uint8_t cmd_status;
02279         Uncomment the lines below if you want to check erro.
02280         Wire.requestFrom(deviceAddress, 1);
02281         cmd_status = Wire.read();
02282         The SI4735 issues a status after each 8 byte transfered.
02283         Just the bit 7 (CTS) should be seted. if bit 6 (ERR) is seted, the
system halts.
02284         if (cmd_status != 0x80)
02285             return false;
02286         */
02287     }
02288     delayMicroseconds(250);
02289     return true;
02290 }
```


bool SI4735::downloadPatch (int eeprom_i2c_address)

Under construction... Transfers the content of a patch stored in a eeprom to the [SI4735](#) device.

TO USE THIS METHOD YOU HAVE TO HAVE A EEPROM WRITEN WITH THE PATCH CONTENT

See also

the sketch write_ssb_patch_eeprom.ino (TO DO)

Parameters

<i>eeprom_i2c_address</i>	
---------------------------	--

Returns

false if an error is found.

```
02304 {
02305     int ssb_patch_content_size;
02306     uint8_t cmd_status;
02307     int i, offset;
02308     uint8_t eepromPage[8];
02309
02310     union {
02311         struct
02312         {
02313             uint8_t lowByte;
02314             uint8_t highByte;
02315         } raw;
02316         uint16_t value;
02317     } eeprom;
02318
02319     // The first two bytes are the size of the patches
02320     // Set the position in the eeprom to read the size of the patch content
02321     Wire.beginTransaction(eeprom_i2c_address);
02322     Wire.write(0); // writes the most significant byte
02323     Wire.write(0); // writes the less significant byte
02324     Wire.endTransmission();
02325     Wire.requestFrom(eeprom_i2c_address, 2);
02326     eeprom.raw.highByte = Wire.read();
02327     eeprom.raw.lowByte = Wire.read();
02328
02329     ssb_patch_content_size = eeprom.value;
02330
02331     // the patch content starts on position 2 (the first two bytes are the
02332     size of the patch)
02333     for (offset = 2; offset < ssb_patch_content_size; offset += 8)
02334     {
02335         // Set the position in the eeprom to read next 8 bytes
02336         eeprom.value = offset;
02337         Wire.beginTransaction(eeprom_i2c_address);
02338         Wire.write(eeprom.raw.highByte); // writes the most significant byte
02339         Wire.write(eeprom.raw.lowByte); // writes the less significant byte
02340         Wire.endTransmission();
02341
02342         // Reads the next 8 bytes from eeprom
02343         Wire.requestFrom(eeprom_i2c_address, 8);
02344         for (i = 0; i < 8; i++)
02345             eepromPage[i] = Wire.read();
02346
02347         // sends the page (8 bytes) to the SI4735
02348         Wire.beginTransaction(deviceAddress);
02349         for (i = 0; i < 8; i++)
02350             Wire.write(eepromPage[i]);
02351         Wire.endTransmission();
02352
02353         waitToSend();
02354
02355         Wire.requestFrom(deviceAddress, 1);
02356         cmd_status = Wire.read();
02357         // The SI4735 issues a status after each 8 byte transfered.
02358         // Just the bit 7 (CTS) should be seted. if bit 6 (ERR) is seted,
02359         the system halts.
02359         if (cmd_status != 0x80)
```

```

02359             return false;
02360         }
02361         delayMicroseconds(250);
02362         return true;
02363     }

```

References waitToSend().

bool SI4735::getACFIndicator () [inline]

Gets the Error flag of status response.

```

00961 { return currentStatus.resp.AFCRL; };

```

uint8_t SI4735::getAgcGainIndex () [inline]

```

00983 { return currentAgcStatus.refined.AGCIDX; }; // Returns the current AGC gain
index.

```

uint8_t SI4735::getAntennaTuningCapacitor () [inline]

Returns integer containing the multipath metric when tune is complete.

```

00967 { return currentStatus.resp.READANTCAP; };

```

void SI4735::getAutomaticGainControl ()

Returns integer containing the current antenna tuning capacitor value.

Queries AGC STATUS

See also

Si47XX PROGRAMMING GUIDE; AN332; For FM page 80; for AM page 142.

AN332 REV 0.8 Universal Programming Guide Amendment for SI4735-D60 SSB and NBFM patches; page 18.

After call this method, you can call isAgcEnabled to know the AGC status and getAgcGainIndex to know the gain index value.

```

00961 {
00962     uint8_t cmd;
00963
00964     if (currentTune == FM\_TUNE\_FREQ)
00965     { // FM TUNE
00966         cmd = FM\_AGC\_STATUS;
00967     }
00968     else
00969     { // AM TUNE - SAME COMMAND used on SSB mode
00970         cmd = AM\_AGC\_STATUS;
00971     }
00972
00973     waitToSend();
00974
00975     Wire.beginTransaction(deviceAddress);
00976     Wire.write(cmd);
00977     Wire.endTransmission();
00978
00979     do
00980     {
00981         waitToSend();
00982         Wire.requestFrom(deviceAddress, 3);
00983         currentAgcStatus.raw[0] = Wire.read(); // STATUS response
00984         currentAgcStatus.raw[1] = Wire.read(); // RESP 1
00985         currentAgcStatus.raw[2] = Wire.read(); // RESP 2
00986     } while (currentAgcStatus.refined.ERR); // If error, try get AGC
status again.
00987 }

```

References waitToSend().

bool SI4735::getBandLimit () [inline]

Returns true if the AFC rails (AFC Rail Indicator).

```
00962 { return currentStatus.resp.BLTF; };
```

bool SI4735::getCurrentAfcRailIndicator () [inline]

Valid Channel.

```
00997 { return currentRgsStatus.resp.AFCRL; };
```

uint8_t SI4735::getCurrentAvcAmMaxGain () [inline]

```
00973 {return currentAvcAmMaxGain; };
```

bool SI4735::getCurrentBlendDetectInterrupt () [inline]

Multipath Detect High.

```
01006 { return currentRgsStatus.resp.BLENDINT; };
```

uint16_t SI4735::getCurrentFrequency ()

Gets the current frequency saved in memory. Unlike `getFrequency`, this method gets the current frequency recorded after the last `setFrequency` command. This method avoids bus traffic and CI processing. However, you can not get others status information like RSSI.

See also

[getFrequency\(\)](#)

```
00905 {
00906     return currentWorkFrequency;
00907 }
```

uint8_t SI4735::getCurrentMultipath () [inline]

Indicates stereo pilot presence.

```
01002 { return currentRgsStatus.resp.MULT; };
```

bool SI4735::getCurrentMultipathDetectHigh () [inline]

Multipath Detect Low.

```
01005 { return currentRgsStatus.resp.MULTHINT; };
```

bool SI4735::getCurrentMultipathDetectLow () [inline]

Signed frequency offset (kHz).

```
01004 { return currentRgsStatus.resp.MULTLINT; };
```

bool SI4735::getCurrentPilot () [inline]

Indicates amount of stereo blend in % (100 = full stereo, 0 = full mono).

```
01001 { return currentRgsStatus.resp.PILOT; };
```

void SI4735::getCurrentReceivedSignalQuality (uint8_t INTACK)

Queries the status of the Received Signal Quality (RSQ) of the current channel. This method should be called before call [getCurrentRSSI\(\)](#), [getCurrentSNR\(\)](#) etc. Command FM_RSQ_STATUS

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 75 and 141

Parameters

<i>INTACK</i>	Interrupt Acknowledge. 0 = Interrupt status preserved; 1 = Clears RSQINT, BLENDINT, SNRHINT, SNRLINT, RSSIHINT, RSSILINT, MULTHINT, MULTLINT.
---------------	---

```
01050 {
01051     uint8_t arg;
01052     uint8_t cmd;
```

```

01053         int sizeResponse;
01054
01055         if (currentTune == FM_TUNE_FREQ)
01056         { // FM TUNE
01057             cmd = FM_RSQ_STATUS;
01058             sizeResponse = 8; // Check it
01059         }
01060         else
01061         { // AM TUNE
01062             cmd = AM_RSQ_STATUS;
01063             sizeResponse = 6; // Check it
01064         }
01065
01066         waitToSend();
01067
01068         arg = INTACK;
01069         Wire.beginTransaction(deviceAddress);
01070         Wire.write(cmd);
01071         Wire.write(arg); // send B00000001
01072         Wire.endTransmission();
01073
01074         // Check it
01075         // do
01076         //{
01077             waitToSend();
01078             Wire.requestFrom(deviceAddress, sizeResponse);
01079             // Gets response information
01080             for (uint8_t i = 0; i < sizeResponse; i++)
01081                 currentRqsStatus.raw[i] = Wire.read();
01082             //} while (currentRqsStatus.resp.ERR); // Try again if error found
01083     }

```

References waitToSend().

void SI4735::getCurrentReceivedSignalQuality (void)

Queries the status of the Received Signal Quality (RSQ) of the current channel Command FM_RSQ_STATUS

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 75 and 141

Parameters

<i>INTACK</i>	Interrupt Acknowledge. 0 = Interrupt status preserved; 1 = Clears RSQINT, BLENDINT, SNRHINT, SNRLINT, RSSIHINT, RSSILINT, MULTHINT, MULTLINT.
---------------	---

```

01096 {
01097     getCurrentReceivedSignalQuality(0);
01098 }

```

uint8_t SI4735::getCurrentRSSI () [inline]

```
00990 { return currentRqsStatus.resp.RSSI; };
```

bool SI4735::getCurrentRssiDetectHigh () [inline]

RSSI Detect Low.

```
00993 { return currentRqsStatus.resp.RSSIHINT; };
```

bool SI4735::getCurrentRssiDetectLow () [inline]

current SNR metric (0–127 dB).

```
00992 { return currentRqsStatus.resp.RSSIILINT; };
```

uint8_t SI4735::getCurrentSignedFrequencyOffset () [inline]

Contains the current multipath metric. (0 = no multipath; 100 = full multipath)

```
01003 { return currentRqsStatus.resp.FREQOFF; };
```

uint8_t SI4735::getCurrentSNR () [inline]

current receive signal strength (0â€“127 dBÎ¼V).

```
00991 { return currentRgsStatus.resp.SNR; };
```

bool SI4735::getCurrentSnrDetectHigh () [inline]

SNR Detect Low.

```
00995 { return currentRgsStatus.resp.SNRHINT; };
```

bool SI4735::getCurrentSnrDetectLow () [inline]

RSSI Detect High.

```
00994 { return currentRgsStatus.resp.SNRLINT; };
```

bool SI4735::getCurrentSoftMuteIndicator () [inline]

AFC Rail Indicator.

```
00998 { return currentRgsStatus.resp.SMUTE; };
```

uint8_t SI4735::getCurrentStereoBlend () [inline]

Soft Mute Indicator. Indicates soft mute is engaged.

```
01000 { return currentRgsStatus.resp.STBLEND; };
```

bool SI4735::getCurrentValidChannel () [inline]

SNR Detect High.

```
00996 { return currentRgsStatus.resp.VALID; };
```

uint8_t SI4735::getCurrentVolume () [inline]

```
01028 { return volume; };
```

uint8_t SI4735::getFirmwareCHIPREV () [inline]

RESP7 - Returns the Component Minor Revision (ASCII).

```
01021 { return firmwareInfo.resp.CHIPREV; };
```

uint8_t SI4735::getFirmwareCMPMAJOR () [inline]

RESP5 - Returns the Patch ID Low byte (HEX).

```
01019 { return firmwareInfo.resp.CMPMAJOR; };
```

uint8_t SI4735::getFirmwareCMPMINOR () [inline]

RESP6 - Returns the Component Major Revision (ASCII).

```
01020 { return firmwareInfo.resp.CMPMINOR; };
```

uint8_t SI4735::getFirmwareFWMAJOR () [inline]

RESP1 - Part Number (HEX)

```
01015 { return firmwareInfo.resp.FWMAJOR; };
```

uint8_t SI4735::getFirmwareFWMINOR () [inline]

RESP2 - Returns the Firmware Major Revision (ASCII).

```
01016 { return firmwareInfo.resp.FWMINOR; };
```

uint8_t SI4735::getFirmwarePATCHH () [inline]

RESP3 - Returns the Firmware Minor Revision (ASCII).

```
01017 { return firmwareInfo.resp.PATCHH; };
```

uint8_t SI4735::getFirmwarePATCHL () [inline]

RESP4 - Returns the Patch ID High byte (HEX).

```
01018 { return firmwareInfo.resp.PATCHL; };
```

uint8_t SI4735::getFirmwarePN () [inline]

Blend Detect Interrupt.

```
01014 { return firmwareInfo.resp.PN; };
```

uint16_t SI4735::getFrequency (void)

Device Status Information Gets the current frequency of the Si4735 (AM or FM) The method status do it an more. See getStatus below.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 73 (FM) and 139 (AM)

```
00885 {
00886     si47x_frequency freq;
00887     getStatus(0, 1);
00888
00889     freq.raw.FREQ_L = currentStatus.resp.READFREQ_L;
00890     freq.raw.FREQ_H = currentStatus.resp.READFREQ_H;
00891
00892     currentWorkFrequency = freq.value;
00893     return freq.value;
00894 }
```

bool SI4735::getGroupLost () [inline]

1 = RDS currently synchronized.

```
01087 { return currentRdsStatus.resp.GRPLOST; };
```

void SI4735::getNext2Block (char * c)

Process data received from group 2B

Parameters

<i>c</i>	char array reference to the "group 2B" text
----------	---

```
01582 {
01583     char raw[2];
01584     int i, j;
01585
01586     raw[1] = currentRdsStatus.resp.BLOCKDL;
01587     raw[0] = currentRdsStatus.resp.BLOCKDH;
01588
01589     for (i = j = 0; i < 2; i++)
01590     {
01591         if (raw[i] == 0xD || raw[i] == 0xA)
01592         {
01593             c[j] = '\0';
01594             return;
01595         }
01596         if (raw[i] >= 32)
01597         {
01598             c[j] = raw[i];
01599             j++;
01600         }
01601         else
01602         {
01603             c[i] = ' ';
01604         }
01605     }
01606 }
```

Referenced by getRdsText0A(), and getRdsText2B().

void SI4735::getNext4Block (char * c)

Process data received from group 2A

Parameters

<i>c</i>	char array reference to the "group 2A" text
----------	---

```
01614 {
01615     char raw[4];
01616     int i, j;
01617
01618     raw[0] = currentRdsStatus.resp.BLOCKCH;
01619     raw[1] = currentRdsStatus.resp.BLOCKCL;
01620     raw[2] = currentRdsStatus.resp.BLOCKDH;
01621     raw[3] = currentRdsStatus.resp.BLOCKDL;
01622     for (i = j = 0; i < 4; i++)
01623     {
01624         if (raw[i] == 0xD || raw[i] == 0xA)
01625         {
01626             c[j] = '\0';
01627             return;
01628         }
01629         if (raw[i] >= 32)
01630         {
01631             c[j] = raw[i];
01632             j++;
01633         }
01634         else
01635         {
01636             c[i] = ' ';
01637         }
01638     }
01639 }
```

Referenced by `getRdsText()`, and `getRdsText2A()`.

uint8_t SI4735::getNumRdsFifoUsed () [inline]

1 = One or more RDS groups discarded due to FIFO overrun.

```
01088 { return currentRdsStatus.resp.RDSFIFOUSED; };
```

bool SI4735::getRadioDataSystemInterrupt () [inline]

Gets Received Signal Quality Interrupt(RSQINT)

```
00957 { return currentStatus.resp.RDSINT; };
```

uint8_t SI4735::getRdsFlagAB (void)

Returns the current Text Flag A/B

Returns

uint8_t

```
01516 {
01517     si47x\_rds\_blockb blkb;
01518
01519     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01520     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01521
01522     return blkb.refined.textABFlag;
01523 }
```

uint8_t SI4735::getRdsGroupType (void)

Returns the Group Type (extracted from the Block B)

```
01500 {
01501     si47x\_rds\_blockb blkb;
01502
01503     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01504     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01505
01506     return blkb.refined.groupType;
01507 }
```

bool SI4735::getRdsNewBlockA () [inline]

1 = Found RDS synchronization

```
01084 { return currentRdsStatus.resp.RDSNEWBLOCKA; };
```

Referenced by getRdsPI().

bool SI4735::getRdsNewBlockB () [inline]

1 = Valid Block A data has been received.

```
01085 { return currentRdsStatus.resp.RDSNEWBLOCKB; };
```

uint16_t SI4735::getRdsPI (void)

Returns the program type. Read the Block A content

See also

SI47XX PROGRAMMING GUIDE; AN332; pages 77 and 78

Returns

BLOCKAL

```
01488 {
01489     if (getRdsReceived() && getRdsNewBlockA())
01490     {
01491         return currentRdsStatus.resp.BLOCKAL;
01492     }
01493     return 0;
01494 }
```

References getRdsNewBlockA(), and getRdsReceived().

uint8_t SI4735::getRdsProgramType (void)

Returns the Program Type (extracted from the Block B)

See also

https://en.wikipedia.org/wiki/Radio_Data_System

Returns

program type (an integer between 0 and 31)

```
01567 {
01568     si47x_rds_blockb blk;
01569
01570     blk.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01571     blk.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01572
01573     return blk.refined.programType;
01574 }
```

bool SI4735::getRdsReceived () [inline]

```
01081 { return currentRdsStatus.resp.RDSRECV; };
```

Referenced by getRdsPI(), getRdsText0A(), and getRdsText2A().

void SI4735::getRdsStatus ()

Gets RDS Status. Same result of calling getRdsStatus(0,0,0);

See also

[SI4735::getRdsStatus\(uint8_t INTACK, uint8_t MTFIFO, uint8_t STATUSONLY\)](#)

Please, call [getRdsStatus\(uint8_t INTACK, uint8_t MTFIFO, uint8_t STATUSONLY\)](#) instead [getRdsStatus\(\)](#) if you want other behaviour

```
01473 {
01474     getRdsStatus(0, 0, 0);
01475 }
```


void SI4735::getRdsStatus (uint8_t INTACK, uint8_t MTFIFO, uint8_t STATUSONLY)

Gets the RDS status. Store the status in currentRdsStatus member. RDS COMMAND FM_RDS_STATUS

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 55 and 77

Parameters

<i>INTACK</i>	Interrupt Acknowledge; 0 = RDSINT status preserved. 1 = Clears RDSINT.
<i>MTFIFO</i>	0 = If FIFO not empty, read and remove oldest FIFO entry; 1 = Clear RDS Receive FIFO.
<i>STATUSONLY</i>	Determines if data should be removed from the RDS FIFO.

```

01426 {
01427     si47x\_rds\_command rds_cmd;
01428     static uint16_t lastFreq;
01429     // checking current FUNC (Am or FM)
01430     if (currentTune != FM\_TUNE\_FREQ)
01431         return;
01432
01433     if (lastFreq != currentWorkFrequency)
01434     {
01435         lastFreq = currentWorkFrequency;
01436         clearRdsBuffer2A();
01437         clearRdsBuffer2B();
01438         clearRdsBuffer0A();
01439     }
01440
01441     waitToSend();
01442
01443     rds_cmd.arg.INTACK = INTACK;
01444     rds_cmd.arg.MTFIFO = MTFIFO;
01445     rds_cmd.arg.STATUSONLY = STATUSONLY;
01446
01447     Wire.beginTransmission(deviceAddress);
01448     Wire.write(FM\_RDS\_STATUS);
01449     Wire.write(rds_cmd.raw);
01450     Wire.endTransmission();
01451
01452     do
01453     {
01454         waitToSend();
01455         // Gets response information
01456         Wire.requestFrom(deviceAddress, 13);
01457         for (uint8_t i = 0; i < 13; i++)
01458             currentRdsStatus.raw[i] = Wire.read();
01459     } while (currentRdsStatus.resp.ERR);
01460     delayMicroseconds(550);
01461 }

```

References [clearRdsBuffer0A\(\)](#), [clearRdsBuffer2A\(\)](#), [clearRdsBuffer2B\(\)](#), and [waitToSend\(\)](#).

bool SI4735::getRdsSync () [inline]

1 = Valid Block B data has been received.

```
01086 { return currentRdsStatus.resp.RDSSYNC; };
```

bool SI4735::getRdsSyncFound () [inline]

1 = Lost RDS synchronization

```
01083 { return currentRdsStatus.resp.RDSSYNCFFOUND; };
```

bool SI4735::getRdsSyncLost () [inline]

1 = FIFO filled to minimum number of groups

```
01082 { return currentRdsStatus.resp.RDSSYNCLOST; };
```

char * SI4735::getRdsText (void)

Gets the RDS Text when the message is of the Group Type 2 version A

Returns

char* The string (char array) with the content (Text) received from group 2A

```
01648 {
01649
01650     // Needs to get the "Text segment address code".
01651     // Each message should be ended by the code 0D (Hex)
01652
01653     if (rdsTextAdress2A >= 16)
01654         rdsTextAdress2A = 0;
01655
01656     getNext4Block(&rds_buffer2A[rdsTextAdress2A * 4]);
01657
01658     rdsTextAdress2A += 4;
01659
01660     return rds_buffer2A;
01661 }
```

References getNext4Block(), rds_buffer2A, and rdsTextAdress2A.

char * SI4735::getRdsText0A (void)

Gets the station name and other messages.

Returns

char* should return a string with the station name. However, some stations send other kind of messages

```
01670 {
01671     si47x_rds_blockb blkB;
01672
01673     // getRdsStatus();
01674
01675     if (getRdsReceived())
01676     {
01677         if (getRdsGroupType() == 0)
01678         {
01679             // Process group type 0
01680             blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01681             blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01682
01683             rdsTextAdress0A = blkB.group0.address;
01684             if (rdsTextAdress0A >= 0 && rdsTextAdress0A < 4)
01685             {
01686                 getNext2Block(&rds_buffer0A[rdsTextAdress0A * 2]);
01687                 rds_buffer0A[8] = '\0';
01688                 return rds_buffer0A;
01689             }
01690         }
01691     }
01692     return NULL;
01693 }
```

References getNext2Block(), getRdsReceived(), rds_buffer0A, and rdsTextAdress0A.

char * SI4735::getRdsText2A (void)

Gets the Text processed for the 2A group

Returns

char* string with the Text of the group A2

```
01701 {
01702     si47x_rds_blockb blkB;
01703
01704     // getRdsStatus();
01705     if (getRdsReceived())
01706     {
01707         if (getRdsGroupType() == 2 /* && getRdsVersionCode() == 0 */)
01708         {
01709             // Process group 2A
01710             // Decode B block information
01711             blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01712             blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01713             rdsTextAdress2A = blkB.group2.address;
01714 }
```

```

01715         if (rdsTextAdress2A >= 0 && rdsTextAdress2A < 16)
01716         {
01717             getNext4Block(&rds_buffer2A[rdsTextAdress2A * 4]);
01718             rds_buffer2A[63] = '\0';
01719             return rds_buffer2A;
01720         }
01721     }
01722 }
01723 return NULL;
01724 }

```

References getNext4Block(), getRdsReceived(), rds_buffer2A, and rdsTextAdress2A.

char * SI4735::getRdsText2B (void)

Gets the Text processed for the 2B group

Returns

char* string with the Text of the group AB

```

01733 {
01734     si47x_rds_blockb blkB;
01735
01736     // getRdsStatus();
01737     // if (getRdsReceived())
01738     // {
01739     //     if (getRdsNewBlockB())
01740     //     {
01741         if (getRdsGroupType() == 2 /* && getRdsVersionCode() == 1 */)
01742         {
01743             // Process group 2B
01744             blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01745             blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01746             rdsTextAdress2B = blkB.group2.address;
01747             if (rdsTextAdress2B >= 0 && rdsTextAdress2B < 16)
01748             {
01749                 getNext2Block(&rds_buffer2B[rdsTextAdress2B * 2]);
01750                 return rds_buffer2B;
01751             }
01752         }
01753     // }
01754     // }
01755     return NULL;
01756 }

```

References getNext2Block(), rds_buffer2B, and rdsTextAdress2B.

uint8_t SI4735::getRdsTextSegmentAddress (void)

Returns the address of the text segment. 2A - Each text segment in version 2A groups consists of four characters. A messages of this group can be have up to 64 characters. 2B - In version 2B groups, each text segment consists of only two characters. When the current RDS status is using this version, the maximum message length will be 32 characters.

Returns

uint8_t the address of the text segment.

```

01536 {
01537     si47x_rds_blockb blkb;
01538     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01539     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01540
01541     return blkb.refined.content;
01542 }

```

char * SI4735::getRdsTime (void)

Gets the RDS time and date when the Group type is 4

Returns

char* a string with hh:mm +/- offset

```

01764 {
01765     // Under Test and construction
01766     // Need to check the Group Type before.

```

```

01767     si47x_rds_date_time dt;
01768
01769     uint16_t minute;
01770     uint16_t hour;
01771
01772     if (getRdsGroupType() == 4)
01773     {
01774         char offset_sign;
01775         int offset_h;
01776         int offset_m;
01777
01778         // uint16_t y, m, d;
01779
01780         dt.raw[4] = currentRdsStatus.resp.BLOCKBL;
01781         dt.raw[5] = currentRdsStatus.resp.BLOCKBH;
01782         dt.raw[2] = currentRdsStatus.resp.BLOCKCL;
01783         dt.raw[3] = currentRdsStatus.resp.BLOCKCH;
01784         dt.raw[0] = currentRdsStatus.resp.BLOCKDL;
01785         dt.raw[1] = currentRdsStatus.resp.BLOCKDH;
01786
01787         // Unfortunately it was necessary to work well on the GCC compiler
01788         // on 32-bit platforms. See si47x_rds_date_time (typedef union) and CGG
01789         // "Crosses boundary" issue/features.
01790         // Now it is working on Atmega328, STM32, Arduino DUE, ESP32 and
01791         // more.
01792         minute = (dt.refined.minute2 << 2) | dt.refined.minute1;
01793         hour = (dt.refined.hour2 << 4) | dt.refined.hour1;
01794
01795         offset_sign = (dt.refined.offset_sense == 1) ? '+' : '-';
01796         offset_h = (dt.refined.offset * 30) / 60;
01797         offset_m = (dt.refined.offset * 30) - (offset_h * 60);
01798         // sprintf(rds_time, "%02u:%02u %c%02u:%02u", dt.refined.hour,
01799         dt.refined.minute, offset_sign, offset_h, offset_m);
01800         sprintf(rds_time, "%02u:%02u %c%02u:%02u", hour, minute,
01801         offset_sign, offset_h, offset_m);
01802         return rds_time;
01803     }
01804     return NULL;
01805 }

```

References rds_time.

uint8_t SI4735::getRdsVersionCode (void)

Gets the version code (extracted from the Block B)

Returns

0=A or 1=B

```

01550 {
01551     si47x_rds_blockb blkb;
01552
01553     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01554     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01555
01556     return blkb.refined.versionCode;
01557 }

```

uint8_t SI4735::getReceivedSignalStrengthIndicator () [inline]

Returns true if the channel is currently valid as determined by the seek/tune properties (0x1403, 0x1404, 0x1108)

```

00964 { return currentStatus.resp.RSSI; };

```

bool SI4735::getSignalQualityInterrupt () [inline]

STATUS RESPONSE Set of methods to get current status information. Call them after getStatus or getFrequency or seekStation See Si47XX PROGRAMMING GUIDE; AN332; pages 63

```

00956 { return currentStatus.resp.RSQINT; };

```

void SI4735::getStatus ()

Gets the current status of the Si4735 (AM or FM)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 73 (FM) and 139 (AM)

```
00948 {  
00949     getStatus(0, 1);  
00950 }
```

void SI4735::getStatus (uint8_t INTACK, uint8_t CANCEL)

Gets the current status of the Si4735 (AM or FM)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 73 (FM) and 139 (AM)

Parameters

<i>uint8_t</i>	INTACK Seek/Tune Interrupt Clear. If set, clears the seek/tune complete interrupt status indicator;
<i>uint8_t</i>	CANCEL Cancel seek. If set, aborts a seek currently in progress;

```
00917 {  
00918     si47x\_tune\_status status;  
00919     uint8_t cmd = (currentTune == FM\_TUNE\_FREQ) ? FM\_TUNE\_STATUS :  
AM\_TUNE\_STATUS;  
00920  
00921     waitToSend();  
00922  
00923     status.arg.INTACK = INTACK;  
00924     status.arg.CANCEL = CANCEL;  
00925  
00926     Wire.beginTransaction(deviceAddress);  
00927     Wire.write(cmd);  
00928     Wire.write(status.raw);  
00929     Wire.endTransmission();  
00930     // Reads the current status (including current frequency).  
00931     do  
00932     {  
00933         waitToSend();  
00934         Wire.requestFrom(deviceAddress, 8); // Check it  
00935         // Gets response information  
00936         for (uint8_t i = 0; i < 8; i++)  
00937             currentStatus.raw[i] = Wire.read();  
00938     } while (currentStatus.resp.ERR); // If error, try it again  
00939     waitToSend();  
00940 }
```

References [waitToSend](#)().

bool SI4735::getStatusCTS () [inline]

Return the Error flag (true or false) of status of the least Tune or Seek.

```
00960 { return currentStatus.resp.CTS; };
```

bool SI4735::getStatusError () [inline]

Seek/Tune Complete Interrupt; 1 = Tune complete has been triggered.

```
00959 { return currentStatus.resp.ERR; };
```

uint8_t SI4735::getStatusMULT () [inline]

Returns integer containing the SNR metric when tune is complete (dB).

```
00966 { return currentStatus.resp.MULT; };
```

uint8_t SI4735::getStatusSNR () [inline]

Returns integer Received Signal Strength Indicator (dB $\hat{1}/4$ V).

```
00965 { return currentStatus.resp.SNR; };
```

bool SI4735::getStatusValid () [inline]

Returns true if a seek hit the band limit (WRAP = 0 in FM_START_SEEK) or wrapped to the original frequency (WRAP = 1).

```
00963 { return currentStatus.resp.VALID; };
```

bool SI4735::getTuneCompleteTriggered () [inline]

Gets Radio Data System (RDS) Interrupt.

```
00958 { return currentStatus.resp.STCINT; };
```

uint8_t SI4735::getTuneFrequencyFast () [inline]

```
01045 { return currentFrequencyParams.arg.FAST; };
```

uint8_t SI4735::getTuneFrequencyFreeze () [inline]

FAST Tuning. If set, executes fast and invalidated tune. The tune status will not be accurate.

```
01047 { return currentFrequencyParams.arg.FREEZE; };
```

uint8_t SI4735::getVolume ()

Gets the current volume level.

See also

[setVolume\(\)](#)

Returns

volume (domain: 0 - 63)

```
01241 {  
01242     return this->volume;  
01243 }
```

bool SI4735::isAgcEnabled () [inline]

```
00982 { return !currentAgcStatus.refined.AGCDIS; }; // Returns true if the  
AGC is enabled
```

bool SI4735::isCurrentTuneFM ()

Returns true if the current function is FM (FM_TUNE_FREQ).

Returns

true if the current function is FM (FM_TUNE_FREQ).

```
00668 {  
00669     return (currentTune == FM\_TUNE\_FREQ);  
00670 }
```

void SI4735::patchPowerUp ()

This method can be used to prepare the device to apply SSBRX patch Call queryLibraryId before call this method. Powerup the device by issuing the POWER_UP command with FUNC = 1 (AM/SW/LW Receive)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 64 and 215-220 and

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE AMENDMENT FOR SI4735-D60
SSB AND NBFM PATCHES; page 7.

```
02166 {  
02167     waitToSend();  
02168     Wire.beginTransaction(deviceAddress);  
02169     Wire.write(POWER\_UP);  
02170     Wire.write(0b00110001); // Set to AM, Enable External Crystal  
Oscillator; Set patch enable; GP02 output disabled; CTS interrupt disabled.  
02171     Wire.write(SI473X\_ANALOG\_AUDIO); // Set to Analog Output  
02172     Wire.endTransmission();  
02173     delayMicroseconds(2500);
```

02174 }

References waitToSend().

si47x_firmware_query_library SI4735::queryLibraryId ()

SI47XX PATCH RESOURCES Call it first if you are applying a patch on [SI4735](#). Used to confirm if the patch is compatible with the internal device library revision. See Si47XX PROGRAMMING GUIDE; AN332; pages 64 and 215-220.

Returns

a struct [si47x_firmware_query_library](#) (see it in [SI4735.h](#)) Query the library information

You have to call this function if you are applying a patch on SI47XX (SI4735-D60)

The first command that is sent to the device is the POWER_UP command to confirm that the patch is compatible with the internal device library revision. The device moves into the powerup mode, returns the reply, and moves into the powerdown mode. The POWER_UP command is sent to the device again to configure the mode of the device and additionally is used to start the patching process. When applying the patch, the PATCH bit in ARG1 of the POWER_UP command must be set to 1 to begin the patching process. [AN332 page 219].

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 214, 215, 216, 219

[si47x_firmware_query_library](#) in [SI4735.h](#)

Returns

[si47x_firmware_query_library](#) Library Identification

```
02129 {
02130     si47x\_firmware\_query\_library libraryID;
02131
02132     powerDown(); // Is it necessary
02133
02134     // delay(500);
02135
02136     waitToSend();
02137     Wire.beginTransaction(deviceAddress);
02138     Wire.write(POWER\_UP);
02139     Wire.write(0b00011111); // Set to Read Library ID, disable
interrupt; disable GPO2OEN; boot normally; enable External Crystal Oscillator .
02140     Wire.write(SI473X\_ANALOG\_AUDIO); // Set to Analog Line Input.
02141     Wire.endTransmission();
02142
02143     do
02144     {
02145         waitToSend();
02146         Wire.requestFrom(deviceAddress, 8);
02147         for (int i = 0; i < 8; i++)
02148             libraryID.raw[i] = Wire.read();
02149     } while (libraryID.resp.ERR); // If error found, try it again.
02150
02151     delayMicroseconds(2500);
02152
02153     return libraryID;
02154 }
```

References [powerDown\(\)](#), and [waitToSend\(\)](#).

void SI4735::RdsInit ()

RDS implementation Starts the control variables for RDS.

```
01277 {
01278     clearRdsBuffer2A();
01279     clearRdsBuffer2B();
01280     clearRdsBuffer0A();
01281     rdsTextAddress2A = rdsTextAddress2B = lastTextFlagAB = rdsTextAddress0A =
0;
01282 }
```

References [clearRdsBuffer0A\(\)](#), [clearRdsBuffer2A\(\)](#), and [clearRdsBuffer2B\(\)](#).

Referenced by setRdsConfig().

void SI4735::seekStation (uint8_t SEEKUP, uint8_t WRAP)

Look for a station

See also

SI47XX PROGRAMMING GUIDE; AN332; pages 55, 72, 125 and 137

Parameters

<i>SEEKUP</i>	Seek Up/Down. Determines the direction of the search, either UP = 1, or DOWN = 0.
<i>Wrap/Halt.</i>	Determines whether the seek should Wrap = 1, or Halt = 0 when it hits the band limit.

```
01109 {
01110     si47x\_seek seek;
01111
01112     // Check which FUNCTION (AM or FM) is working now
01113     uint8_t seek_start = (currentTune == FM\_TUNE\_FREQ) ? FM\_SEEK\_START :
AM\_SEEK\_START;
01114
01115     waitToSend();
01116
01117     seek.arg.SEEKUP = SEEKUP;
01118     seek.arg.WRAP = WRAP;
01119
01120     Wire.beginTransaction(deviceAddress);
01121     Wire.write(seek_start);
01122     Wire.write(seek.raw);
01123
01124     if (seek_start == AM\_SEEK\_START)
01125     {
01126         Wire.write(0x00); // Always 0
01127         Wire.write(0x00); // Always 0
01128         Wire.write(0x00); // Tuning Capacitor: The tuning capacitor value
01129         Wire.write(0x00); // will be selected
01130     }
01131
01132     Wire.endTransmission();
01133     delay(100);
01134 }
```

References [waitToSend](#)().

void SI4735::seekStationDown ()

Search the previous station

See also

[seekStation\(uint8_t SEEKUP, uint8_t WRAP\)](#)

```
01154 {
01155     seekStation(0, 1);
01156     delay(50);
01157     getFrequency();
01158 }
```

void SI4735::seekStationUp ()

Search for the next station

See also

[seekStation\(uint8_t SEEKUP, uint8_t WRAP\)](#)

```
01142 {
01143     seekStation(1, 1);
01144     delay(50);
01145     getFrequency();
01146 }
```

void SI4735::sendProperty (uint16_t *propertyValue*, uint16_t *parameter*)[protected]

Sends (sets) property to the SI47XX This method is used for others to send generic properties and params to SI47XX

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 68, 124 and 133.

```
00679 {
00680     si47x\_property property;
00681     si47x\_property param;
00682
00683     property.value = propertyValue;
00684     param.value = parameter;
00685     waitToSend();
00686     Wire.beginTransaction\(deviceAddress\);
00687     Wire.write\(SET\_PROPERTY\);
00688     Wire.write\(0x00\);
00689     Wire.write\(property.raw.byteHigh\); // Send property - High byte - most
significant first
00690     Wire.write\(property.raw.byteLow\); // Send property - Low byte - less
significant after
00691     Wire.write\(param.raw.byteHigh\); // Send the arguments. High Byte -
Most significant first
00692     Wire.write\(param.raw.byteLow\); // Send the arguments. Low Byte - Less
significant after
00693     Wire.endTransmission\(\);
00694     delayMicroseconds\(550\);
00695 }
```

References [waitToSend\(\)](#).

void SI4735::sendSSBModeProperty () [protected]

Just send the property SSB_MOD to the device. Internal use (privete method).

```
02082 {
02083     si47x\_property property;
02084     property.value = SSB\_MODE;
02085     waitToSend();
02086     Wire.beginTransaction\(deviceAddress\);
02087     Wire.write\(SET\_PROPERTY\);
02088     Wire.write\(0x00\); // Always 0x00
02089     Wire.write\(property.raw.byteHigh\); // High byte first
02090     Wire.write\(property.raw.byteLow\); // Low byte after
02091     Wire.write\(currentSSBMode.raw\[1\]\); // SSB MODE params; freq. high byte
first
02092     Wire.write\(currentSSBMode.raw\[0\]\); // SSB MODE params; freq. low byte
after
02093
02094     Wire.endTransmission\(\);
02095     delayMicroseconds\(550\);
02096 }
```

References [waitToSend\(\)](#).

Referenced by [setSBBSidebandCutoffFilter\(\)](#), [setSSBAudioBandwidth\(\)](#),
[setSSBAutomaticVolumeControl\(\)](#), [setSSBAvcDivider\(\)](#), [setSSBConfig\(\)](#), [setSSBDspAfc\(\)](#), and
[setSSBSoftMute\(\)](#).

void SI4735::setAmSoftMuteMaxAttenuation () [inline]

```
00976 {sendProperty\(AM\_SOFT\_MUTE\_MAX\_ATTENUATION, 0\)};;
```

void SI4735::setAmSoftMuteMaxAttenuation (uint8_t smattn) [inline]

```
00975 {sendProperty\(AM\_SOFT\_MUTE\_MAX\_ATTENUATION, smattn\)};;
```

void SI4735::setAudioMute (bool off)

Returns the current volume level.

Sets the audio on or off

See also

See Si47XX PROGRAMMING GUIDE; AN332; pages 62, 123, 171

Parameters

value	if true, mute the audio; if false unmute the audio.
01228	{
01229	uint16_t value = (off)? 3:0; // 3 means mute; 0 means unmute
01230	sendProperty(RX_HARD_MUTE, value) ;

01231 }

void SI4735::setAutomaticGainControl (uint8_t AGCDIS, uint8_t AGCIDX)

If FM, overrides AGC setting by disabling the AGC and forcing the LNA to have a certain gain that ranges between 0 (minimum attenuation) and 26 (maximum attenuation); If AM/SSB, Overrides the AM AGC setting by disabling the AGC and forcing the gain index that ranges between 0 (minimum attenuation) and 37+ATTN_BACKUP (maximum attenuation);

See also

Si47XX PROGRAMMING GUIDE; AN332; For FM page 81; for AM page 143

Parameters

uint8_t	AGCDIS This param selects whether the AGC is enabled or disabled (0 = AGC enabled; 1 = AGC disabled);
uint8_t	AGCIDX AGC Index (0 = Minimum attenuation (max gain); 1 – 36 = Intermediate attenuation); if > greater than 36 - Maximum attenuation (min gain)).

```
01002 {
01003     si47x\_agc\_override agc;
01004
01005     uint8_t cmd;
01006
01007     cmd = (currentTune == FM\_TUNE\_FREQ) ? FM\_AGC\_OVERRIDE : AM\_AGC\_OVERRIDE;
01008
01009     agc.arg.AGCDIS = AGCDIS;
01010     agc.arg.AGCIDX = AGCIDX;
01011
01012     waitToSend();
01013
01014     Wire.beginTransaction(deviceAddress);
01015     Wire.write(cmd);
01016     Wire.write(agc.raw[0]);
01017     Wire.write(agc.raw[1]);
01018     Wire.endTransmission();
01019
01020     waitToSend();
01021 }
```

References [waitToSend](#)().

void SI4735::setAvcAmMaxGain () [inline]

```
00972 { sendProperty(AM\_AUTOMATIC\_VOLUME\_CONTROL\_MAX\_GAIN, ((currentAvcAmMaxGain =
48) * 340));};
```

void SI4735::setAvcAmMaxGain (uint8_t gain)

Sets the maximum gain for automatic volume control. If no parameter is sent, it will be consider 48dB.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 152

Parameters

uint8_t	gain Select a value between 12 and 192. Defaul value 48dB.
---------	--

```
01031 {
01032     uint16_t aux;
01033     aux = ( gain > 12 && gain < 193 )? (gain * 340) : (48 * 340);
01034     currentAvcAmMaxGain = gain;
01035     sendProperty(AM\_AUTOMATIC\_VOLUME\_CONTROL\_MAX\_GAIN, aux);
01036 }
```

void SI4735::setBandwidth (uint8_t AMCHFLT, uint8_t AMPLFLT)

Selects the bandwidth of the channel filter for AM reception. The choices are 6, 4, 3, 2, 2.5, 1.8, or 1 (kHz). The default bandwidth is 2 kHz. Works only in AM / SSB (LW/MW/SW)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 125, 151, 277, 181.

Parameters

<i>AMCHFLT</i>	the choices are: 0 = 6 kHz Bandwidth 1 = 4 kHz Bandwidth 2 = 3 kHz Bandwidth 3 = 2 kHz Bandwidth 4 = 1 kHz Bandwidth 5 = 1.8 kHz Bandwidth 6 = 2.5 kHz Bandwidth, gradual roll off 7–15 = Reserved (Do not use).
<i>AMPLFLT</i>	Enables the AM Power Line Noise Rejection Filter.

```

00633 {
00634     si47x\_bandwidth\_config filter;
00635     si47x\_property property;
00636
00637     if (currentTune == FM\_TUNE\_FREQ) // Only for AM/SSB mode
00638         return;
00639
00640     if (AMCHFLT > 6)
00641         return;
00642
00643     property.value = AM\_CHANNEL\_FILTER;
00644
00645     filter.param.AMCHFLT = AMCHFLT;
00646     filter.param.AMPLFLT = AMPLFLT;
00647
00648     waitToSend();
00649     this->volume = volume;
00650     Wire.beginTransaction(deviceAddress);
00651     Wire.write(SET\_PROPERTY);
00652     Wire.write(0x00); // Always 0x00
00653     Wire.write(property.raw.byteHigh); // High byte first
00654     Wire.write(property.raw.byteLow); // Low byte after
00655     Wire.write(filter.raw[1]); // Raw data for AMCHFLT and
00656     Wire.write(filter.raw[0]); // AMPLFLT
00657     Wire.endTransmission();
00658     waitToSend();
00659 }

```

References [waitToSend](#)().

void SI4735::setFmBlendMonoThreshold (uint8_t parameter)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold). To force stereo set this to 0. To force mono set this to 127. Default value is 30 dB $\frac{1}{4}$ V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 56.

Parameters

<i>parameter</i>	valid values: 0 to 127
------------------	------------------------

```

00720 {
00721     sendProperty(FM\_BLEND\_MONO\_THRESHOLD, parameter);
00722 }

```

void SI4735::setFmBlendMultiPathMonoThreshold (uint8_t parameter)

Sets Multipath threshold for mono blend (Full mono above threshold, blend below threshold). To force stereo, set to 100. To force mono, set to 0. The default is 60.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 60.

Parameters

<i>parameter</i>	valid values: 0 to 100
------------------	------------------------

```

00797 {
00798     sendProperty(FM\_BLEND\_MULTIPATH\_MONO\_THRESHOLD, parameter);
00799 }

```

void SI4735::setFmBlendMultiPathStereoThreshold (uint8_t parameter)

Sets multipath threshold for stereo blend (Full stereo below threshold, blend above threshold). To force stereo, set this to 100. To force mono, set this to 0. Default value is 20.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 60.

Parameters

<i>parameter</i>	valid values: 0 to 100
00784 {	
00785 sendProperty (FM_BLEND_MULTIPATH_STEREO_THRESHOLD , parameter);	
00786 }	

void SI4735::setFmBLendRssiMonoThreshold (uint8_t *parameter*)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold). To force stereo, set this to 0. To force mono, set this to 127. Default value is 30 dB \hat{I} $\frac{1}{4}$ V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

<i>parameter</i>	valid values: 0 to 127
00745 {	
00746 sendProperty (FM_BLEND_RSSI_MONO_THRESHOLD , parameter);	
00747 }	

void SI4735::setFmBlendRssiStereoThreshold (uint8_t *parameter*)

Sets RSSI threshold for stereo blend. (Full stereo above threshold, blend below threshold.) To force stereo, set this to 0. To force mono, set this to 127. Default value is 49 dB \hat{I} $\frac{1}{4}$ V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

<i>parameter</i>	valid values: 0 to 127
00732 {	
00733 sendProperty (FM_BLEND_RSSI_STEREO_THRESHOLD , parameter);	
00734 }	

void SI4735::setFmBLendSnrMonoThreshold (uint8_t *parameter*)

Sets SNR threshold for mono blend (Full mono below threshold, blend above threshold). To force stereo, set this to 0. To force mono, set this to 127. Default value is 14 dB.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

<i>parameter</i>	valid values: 0 to 127
00771 {	
00772 sendProperty (FM_BLEND_SNR_MONO_THRESHOLD , parameter);	
00773 }	

void SI4735::setFmBlendSnrStereoThreshold (uint8_t *parameter*)

Sets SNR threshold for stereo blend (Full stereo above threshold, blend below threshold). To force stereo, set this to 0. To force mono, set this to 127. Default value is 27 dB.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

<i>parameter</i>	valid values: 0 to 127
00758 {	
00759 sendProperty (FM_BLEND_SNR_STEREO_THRESHOLD , parameter);	
00760 }	

void SI4735::setFmBlendStereoThreshold (uint8_t *parameter*)

Sets RSSI threshold for stereo blend (Full stereo above threshold, blend below threshold). To force stereo, set this to 0. To force mono, set this to 127.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 90.

Parameters

<i>parameter</i>	valid values: 0 to 127
------------------	------------------------

```
00707 {  
00708     sendProperty\(FM\_BLEND\_STEREO\_THRESHOLD, parameter\);  
00709 }
```

void SI4735::setFmStereoOff ()

Turn Off Stereo operation.

```
00805 {  
00806     // TO DO  
00807 }
```

void SI4735::setFmStereoOn ()

Turn Off Stereo operation.

```
00813 {  
00814     // TO DO  
00815 }
```

void SI4735::setFunction (uint8_t *FUNC*)

void SI4735::setI2CFastMode (void) [inline]

Sets I2C buss to 100KHz.

```
01141 { Wire.setClock(400000); };
```

void SI4735::setI2CFastModeCustom (long *value* = 500000) [inline]

Sets I2C buss to 400KHz.

Sets the I2C bus to a given value.

ATTENTION: use this function with cation

Parameters

<i>value</i>	in Hz. For example: The values 500000 sets the bus to 500KHz.
--------------	---

```
01150 { Wire.setClock(value); };
```

void SI4735::setI2CLowSpeedMode (void) [inline]

```
01139 { Wire.setClock(10000); };
```

void SI4735::setI2CStandardMode (void) [inline]

Sets I2C buss to 10KHz.

```
01140 { Wire.setClock(100000); };
```

void SI4735::setRdsConfig (uint8_t *RDSSEN*, uint8_t *BLETHA*, uint8_t *BLETHB*, uint8_t *BLETHC*, uint8_t *BLETHD*)

RESP3 - RDS FIFO Used; Number of groups remaining in the RDS FIFO (0 if empty).

Sets RDS property (FM_RDS_CONFIG) Configures RDS settings to enable RDS processing (RDSSEN) and set RDS block error thresholds. When a RDS Group is received, all block errors must be less than or equal the associated block error threshold for the group to be stored in the RDS FIFO.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 104

IMPORTANT: All block errors must be less than or equal the associated block error threshold for the group to be stored in the RDS FIFO. 0 = No errors. 1 = 1–2 bit errors detected and corrected. 2 = 3–5 bit errors detected and corrected. 3 = Uncorrectable.

Recommended Block Error Threshold options: 2,2,2,2 = No group stored if any errors are uncorrected. 3,3,3,3 = Group stored regardless of errors. 0,0,0,0 = No group stored containing corrected or uncorrected errors. 3,2,3,3 = Group stored with corrected errors on B, regardless of errors on A, C, or D.

Parameters

<i>uint8_t</i>	RDSSEN RDS Processing Enable; 1 = RDS processing enabled.
<i>uint8_t</i>	BLETHA Block Error Threshold BLOCKA.
<i>uint8_t</i>	BLETHB Block Error Threshold BLOCKB.
<i>uint8_t</i>	BLETHC Block Error Threshold BLOCKC.
<i>uint8_t</i>	BLETHD Block Error Threshold BLOCKD.

```

01341 {
01342     si47x\_property property;
01343     si47x\_rds\_config config;
01344
01345     waitToSend();
01346
01347     // Set property value
01348     property.value = FM\_RDS\_CONFIG;
01349
01350     // Arguments
01351     config.arg.RDSSEN = RDSSEN;
01352     config.arg.BLETHA = BLETHA;
01353     config.arg.BLETHB = BLETHB;
01354     config.arg.BLETHC = BLETHC;
01355     config.arg.BLETHD = BLETHD;
01356     config.arg.DUMMY1 = 0;
01357
01358     Wire.beginTransaction(deviceAddress);
01359     Wire.write(SET\_PROPERTY);
01360     Wire.write(0x00); // Always 0x00 (I need to check it)
01361     Wire.write(property.raw.byteHigh); // Send property - High byte - most
significant first
01362     Wire.write(property.raw.byteLow); // Low byte
01363     Wire.write(config.raw[1]); // Send the arguments. Most
significant first
01364     Wire.write(config.raw[0]);
01365     Wire.endTransmission();
01366     delayMicroseconds(550);
01367
01368     RdsInit();
01369 }

```

References [RdsInit\(\)](#), and [waitToSend\(\)](#).

**void SI4735::setRdsIntSource (uint8_t *RDSNEWBLOCKB*, uint8_t *RDSNEWBLOCKA*,
uint8_t *RDSSYNCFOUND*, uint8_t *RDSSYNCLOST*, uint8_t *RDSRECV*)**

Configures interrupt related to RDS

Use this method if want to use interrupt

See also

Si47XX PROGRAMMING GUIDE; AN332; page 103

Parameters

<i>RDSRECV</i>	If set, generate RDSINT when RDS FIFO has at least FM_RDS_INT_FIFO_COUNT entries.
<i>RDSSYNCLOST</i>	If set, generate RDSINT when RDS loses synchronization.
<i>RDSSYNCFOUN D</i>	set, generate RDSINT when RDS gains synchronization.
<i>RDSNEWBLOCK A</i>	If set, generate an interrupt when Block A data is found or subsequently changed
<i>RDSNEWBLOCK B</i>	If set, generate an interrupt when Block B data is found or subsequently changed

```

01385 {

```

```

01386     si47x\_property property;
01387     si47x\_rds\_int\_source rds_int_source;
01388
01389     if (currentTune != FM\_TUNE\_FREQ)
01390         return;
01391
01392     rds_int_source.refined.RDSNEWBLOCKB = RDSNEWBLOCKB;
01393     rds_int_source.refined.RDSNEWBLOCKA = RDSNEWBLOCKA;
01394     rds_int_source.refined.RDSSYNCFFOUND = RDSSYNCFFOUND;
01395     rds_int_source.refined.RDSSYNCLOST = RDSSYNCLOST;
01396     rds_int_source.refined.RDSRECV = RDSRECV;
01397     rds_int_source.refined.DUMMY1 = 0;
01398     rds_int_source.refined.DUMMY2 = 0;
01399
01400     property.value = FM\_RDS\_INT\_SOURCE;
01401
01402     waitToSend();
01403
01404     Wire.beginTransaction(deviceAddress);
01405     Wire.write(SET\_PROPERTY);
01406     Wire.write(0x00); // Always 0x00 (I need to check it)
01407     Wire.write(property.raw.byteHigh); // Send property - High byte - most
significant first
01408     Wire.write(property.raw.byteLow); // Low byte
01409     Wire.write(rds_int_source.raw[1]); // Send the arguments. Most
significant first
01410     Wire.write(rds_int_source.raw[0]);
01411     Wire.endTransmission();
01412     waitToSend();
01413 }

```

References [waitToSend](#)().

void SI4735::setSBBSidebandCutoffFilter (uint8_t *SBCUTFLT*)

Sets SBB Sideband Cutoff Filter for band pass and low pass filters: 0 = Band pass filter to cutoff both the unwanted side band and high frequency components > 2.0 kHz of the wanted side band. (default) 1 = Low pass filter to cutoff the unwanted side band. Other values = not allowed.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>SBCUTFLT</i>	0 or 1; see above
-----------------	-------------------

```

01990 {
01991     currentSSBMode.param.SBCUTFLT = SBCUTFLT;
01992     sendSSBModeProperty();
01993 }

```

References [sendSSBModeProperty](#)().

void SI4735::setSeekAmLimits (uint16_t *bottom*, uint16_t *top*)

Sets the bottom frequency and top frequency of the AM band for seek. Default is 520 to 1710.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 127, 161, and 162

Parameters

<i>uint16_t</i>	bottom - the bottom of the AM band for seek
<i>uint16_t</i>	top - the top of the AM band for seek

```

01169 {
01170     sendProperty(AM\_SEEK\_BAND\_BOTTOM, bottom);
01171     sendProperty(AM\_SEEK\_BAND\_TOP, top);
01172 }

```

void SI4735::setSeekAmSpacing (uint16_t *spacing*)

Selects frequency spacing for AM seek. Default is 10 kHz spacing.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 163, 229 and 283

Parameters

<i>uint16_t</i>	spacing - step in KHz
-----------------	-----------------------

```
01182 {
01183     sendProperty(AM_SEEK_FREQ_SPACING, spacing);
01184 }
```

void SI4735::setSeekRssiThreshold (uint16_t value)

Sets the RSSI threshold for a valid AM Seek/Tune. If the value is zero then RSSI threshold is not considered when doing a seek. Default value is 25 dB¹/₄V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 127

```
01204 {
01205     sendProperty(AM_SEEK_RSSI_THRESHOLD, value);
01206 }
```

void SI4735::setSeekSrnThreshold (uint16_t value)

Sets the SNR threshold for a valid AM Seek/Tune. If the value is zero then SNR threshold is not considered when doing a seek. Default value is 5 dB.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 127

```
01193 {
01194     sendProperty(AM_SEEK_SNR_THRESHOLD, value);
01195 }
```

void SI4735::setSSB (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step, uint8_t usblsb)

```
02062 {
02063     currentMinimumFrequency = fromFreq;
02064     currentMaximumFrequency = toFreq;
02065     currentStep = step;
02066
02067     if (initialFreq < fromFreq || initialFreq > toFreq)
02068         initialFreq = fromFreq;
02069
02070     setSSB(usblsb);
02071
02072     currentWorkFrequency = initialFreq;
02073     setFrequency(currentWorkFrequency);
02074     delayMicroseconds(550);
02075 }
```

void SI4735::setSSB (uint8_t usblsb)

Set the radio to AM function. It means: LW MW and SW.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; pages 13 and 14

[setAM\(\)](#)

void [SI4735::setFrequency](#)(uint16_t freq)

Parameters

<i>usblsb</i>	upper or lower side band; 1 = LSB; 2 = USB
---------------	--

```
02036 {
02037     // Is it needed to load patch when switch to SSB?
02038     // powerDown();
02039     // It starts with the same AM parameters.
02040     setPowerUp(1, 1, 0, 1, 1, SI473X\_ANALOG\_AUDIO);
02041     radioPowerUp();
02042     // ssbPowerUp(); // Not used for regular operation
02043     setVolume(volume); // Set to previous configured volume
02044     currentSsbStatus = usblsb;
02045     lastMode = SSB\_CURRENT\_MODE;
02046 }
```


References radioPowerUp().

void SI4735::setSSBAudioBandwidth (uint8_t AUDIOBW)

SSB Audio Bandwidth for SSB mode

0 = 1.2 kHz low-pass filter* . (default) 1 = 2.2 kHz low-pass filter* . 2 = 3.0 kHz low-pass filter. 3 = 4.0 kHz low-pass filter. 4 = 500 Hz band-pass filter for receiving CW signal, i.e. [250 Hz, 750 Hz] with center frequency at 500 Hz when USB is selected or [-250 Hz, -750 1Hz] with center frequency at -500Hz when LSB is selected* . 5 = 1 kHz band-pass filter for receiving CW signal, i.e. [500 Hz, 1500 Hz] with center frequency at 1 kHz when USB is selected or [-500 Hz, -1500 1 Hz] with center frequency at -1kHz when LSB is selected* . Other values = reserved. Note: If audio bandwidth selected is about 2 kHz or below, it is recommended to set SBCUTFLT[3:0] to 0 to enable the band pass filter for better high- cut performance on the wanted side band. Otherwise, set it to 1.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>AUDIOBW</i>	the valid values are 0, 1, 2, 3, 4 or 5; see description above
----------------	--

```
02019 {  
02020     // Sets the audio filter property parameter  
02021     currentSSBMode.param.AUDIOBW = AUDIOBW;  
02022     sendSSBModeProperty\(\) ;  
02023 }
```

References sendSSBModeProperty().

void SI4735::setSSBAutomaticVolumeControl (uint8_t AVCEN)

Sets SSB Automatic Volume Control (AVC) for SSB mode

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>AVCEN</i>	0 = Disable AVC; 1 = Enable AVC (default).
--------------	--

```
01961 {  
01962     currentSSBMode.param.AVCEN = AVCEN;  
01963     sendSSBModeProperty\(\) ;  
01964 }
```

References sendSSBModeProperty().

void SI4735::setSSBAvcDivider (uint8_t AVC_DIVIDER)

Sets AVC Divider

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>AVC_DIVIDER</i>	SSB mode, set divider = 0; SYNC mode, set divider = 3; Other values = not allowed.
--------------------	--

```
01974 {  
01975     currentSSBMode.param.AVC_DIVIDER = AVC_DIVIDER;  
01976     sendSSBModeProperty\(\) ;  
01977 }
```

References sendSSBModeProperty().

void SI4735::setSSBBfo (int offset)

Single Side Band (SSB) implementation

This implementation was tested only on Si4735-D60 device.

SSB modulation is a refinement of amplitude modulation that one of the side band and the carrier are suppressed.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; pages 3 and 5

First of all, it is important to say that the SSB patch content is not part of this library. The patches used here were made available by Mr. Vadim Afonkin on his Dropbox repository. It is important to note that the author of this library does not encourage anyone to use the SSB patches content for commercial purposes. In other words, this library only supports SSB patches, the patches themselves are not part of this library.

What does SSB patch means? In this context, a patch is a piece of software used to change the behavior of the [SI4735](#) device. There is little information available about patching the [SI4735](#).

The following information is the understanding of the author of this project and it is not necessarily correct.

A patch is executed internally (run by internal MCU) of the device. Usually, patches are used to fixes bugs or add improvements and new features of the firmware installed in the internal ROM of the device. Patches to the [SI4735](#) are distributed in binary form and have to be transferred to the internal RAM of the device by the host MCU (in this case Arduino boards). Since the RAM is volatile memory, the patch stored into the device gets lost when you turn off the system. Consequently, the content of the patch has to be transferred again to the device each time after turn on the system or reset the device.

I would like to thank Mr Vadim Afonkin for making available the SSBRX patches for SI4735-D60 on his Dropbox repository. On this repository you have two files, `amrx_6_0_1_ssbrx_patch_full_0x9D29.csg` and `amrx_6_0_1_ssbrx_patch_init_0xA902.csg`. It is important to know that the patch content of the original files is constant hexadecimal representation used by the language C/C++. Actually, the original files are in ASCII format (not in binary format). If you are not using C/C++ or if you want to load the files directly to the [SI4735](#), you must convert the values to numeric value of the hexadecimal constants. For example: $0x15 = 21$ (00010101); $0x16 = 22$ (00010110); $0x01 = 1$ (00000001); $0xFF = 255$ (11111111);

ATTENTION: The author of this project does not guarantee that procedures shown here will work in your development environment. Given this, it is at your own risk to continue with the procedures suggested here. This library works with the I²C communication protocol and it is designed to apply a SSB extension PATCH to CI SI4735-D60. Once again, the author disclaims any liability for any damage this procedure may cause to your [SI4735](#) or other devices that you are using. Sets the SSB Beat Frequency Offset (BFO).

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; pages 5 and 23

Parameters

<code>offset</code>	16-bit signed value (unit in Hz). The valid range is -16383 to +16383 Hz.
01866 { 01867 01868 si47x_property property; 01869 si47x_frequency bfo_offset; 01870 01871 if (currentTune == FM_TUNE_FREQ) // Only for AM/SSB mode 01872 return; 01873 01874 waitToSend (); 01875 01876 property.value = SSB_BFO ; 01877 bfo_offset.value = offset; 01878 01879 Wire.beginTransaction(deviceAddress); 01880 Wire.write(SET_PROPERTY); 01881 Wire.write(0x00); // Always 0x00 01882 Wire.write(property. raw .byteHigh); // High byte first 01883 Wire.write(property. raw .byteLow); // Low byte after 01884 Wire.write(bfo_offset. raw .FREQH); // Offset freq. high byte first	

```

01885     Wire.write(bfo_offset.raw.FREQ_L); // Offset freq. low byte first
01886
01887     Wire.endTransmission();
01888     delayMicroseconds(550);
01889 }

```

References waitToSend().

void SI4735::setSSBConfig (uint8_t *AUDIOBW*, uint8_t *SBCUTFLT*, uint8_t *AVC_DIVIDER*, uint8_t *AVCEN*, uint8_t *SMUTESEL*, uint8_t *DSP_AFCDIS*)

Set the SSB receiver mode details: 1) Enable or disable AFC track to carrier function for receiving normal AM signals; 2) Set the audio bandwidth; 3) Set the side band cutoff filter; 4) Set soft-mute based on RSSI or SNR; 5) Enable or disable automatic volume control (AVC) function.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>AUDIOBW</i>	SSB Audio bandwidth; 0 = 1.2KHz (default); 1=2.2KHz; 2=3KHz; 3=4KHz; 4=500Hz; 5=1KHz.
<i>SBCUTFLT</i>	SSB side band cutoff filter for band pass and low pass filter if 0, the band pass filter to cutoff both the unwanted side band and high frequency component > 2KHz of the wanted side band (default).
<i>AVC_DIVIDER</i>	set 0 for SSB mode; set 3 for SYNC mode.
<i>AVCEN</i>	SSB Automatic Volume Control (AVC) enable; 0=disable; 1=enable (default).
<i>SMUTESEL</i>	SSB Soft-mute Based on RSSI or SNR.
<i>DSP_AFCDIS</i>	DSP AFC Disable or enable; 0=SYNC MODE, AFC enable; 1=SSB MODE, AFC disable.

```

01911 {
01912     if (currentTune == FM\_TUNE\_FREQ) // Only AM/SSB mode
01913         return;
01914
01915     currentSSBMode.param.AUDIOBW = AUDIOBW;
01916     currentSSBMode.param.SBCUTFLT = SBCUTFLT;
01917     currentSSBMode.param.AVC_DIVIDER = AVC_DIVIDER;
01918     currentSSBMode.param.AVCEN = AVCEN;
01919     currentSSBMode.param.SMUTESEL = SMUTESEL;
01920     currentSSBMode.param.DUMMY1 = 0;
01921     currentSSBMode.param.DSP_AFCDIS = DSP_AFCDIS;
01922
01923     sendSSBModeProperty();
01924 }

```

References sendSSBModeProperty().

void SI4735::setSSBDspAfc (uint8_t *DSP_AFCDIS*)

Sets DSP AFC disable or enable

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>DSP_AFCDIS</i>	0 = SYNC mode, AFC enable; 1 = SSB mode, AFC disable
-------------------	--

```

01934 {
01935     currentSSBMode.param.DSP_AFCDIS = DSP_AFCDIS;
01936     sendSSBModeProperty();
01937 }

```

References sendSSBModeProperty().

void SI4735::setSSBSoftMute (uint8_t *SMUTESEL*)

Sets SSB Soft-mute Based on RSSI or SNR Selection:

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

<i>SMUTESEL</i>	0 = Soft-mute based on RSSI (default); 1 = Soft-mute based on SNR.
-----------------	--

```
01948 {  
01949     currentSSBMode.param.SMUTESEL = SMUTESEL;  
01950     sendSSBModeProperty();  
01951 }
```

References [sendSSBModeProperty\(\)](#).

void SI4735::setSsbSoftMuteMaxAttenuation () [inline]

```
00979 { sendProperty(SSB_SOFT_MUTE_MAX_ATTENUATION, 0); };
```

void SI4735::setSsbSoftMuteMaxAttenuation (uint8_t smattn) [inline]

```
00978 { sendProperty(SSB_SOFT_MUTE_MAX_ATTENUATION, smattn); };
```

void SI4735::setTuneFrequencyFast (uint8_t FAST) [inline]

Returns the FAST tuning status.

```
01046 { currentFrequencyParams.arg.FAST = FAST; };
```

void SI4735::setTuneFrequencyFreeze (uint8_t FREEZE) [inline]

Returns the FREEZE status.

```
01048 { currentFrequencyParams.arg.FREEZE = FREEZE; };
```

void SI4735::setVolume (uint8_t volume)

RESP8 - Returns the Chip Revision (ASCII).

Sets volume level (0 to 63)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 62, 123, 170, 173 and 204

Parameters

<i>uint8_t</i>	volume (domain: 0 - 63)
----------------	-------------------------

```
01216 {  
01217     sendProperty(RX_VOLUME, volume);  
01218     this->volume = volume;  
01219 }
```

void SI4735::ssbPowerUp ()

This function can be useful for debug and teste.

```
02192 {  
02193     waitToSend();  
02194     Wire.beginTransaction(deviceAddress);  
02195     Wire.write(POWER_UP);  
02196     Wire.write(0b00010001); // Set to AM/SSB, disable interrupt; disable  
GPO2OEN; boot normaly; enable External Crystal Oscillator .  
02197     Wire.write(0b000000101); // Set to Analog Line Input.  
02198     Wire.endTransmission();  
02199     delayMicroseconds(2500);  
02200  
02201     powerUp.arg.CTSIEN = 0;           // 1 -> Interrupt anabled;  
02202     powerUp.arg.GPO2OEN = 0;          // 1 -> GPO2 Output Enable;  
02203     powerUp.arg.PATCH = 0;           // 0 -> Boot normally;  
02204     powerUp.arg.XOSCEN = 1;           // 1 -> Use external crystal  
oscillator;  
02205     powerUp.arg.FUNC = 1;              // 0 = FM Receive; 1 = AM/SSB  
(LW/MW/SW) Receiver.  
02206     powerUp.arg.OPMODE = 0b00000101; // 0x5 = 00000101 = Analog audio  
outputs (LOUT/ROUT).  
02207 }
```

References [waitToSend\(\)](#).

void SI4735::ssbSetup ()

Starts the Si473X device on SSB (same AM Mode). Same [SI4735::setup](#) optimized to improve loading patch performance

```

02181 {
02182     // setPowerUp(powerUp.arg.CTSIEN, 0, 0, 1, 1, SI473X_ANALOG_AUDIO);
02183     reset\(\);
02184     // radioPowerUp();
02185 }

```

References [reset\(\)](#).

void SI4735::volumeDown ()

Set sound volume level Down

See also

[setVolume\(\)](#)

```

01263 {
01264     if (volume > 0)
01265         volume--;
01266     setVolume\(volume\);
01267 }

```

void SI4735::volumeUp ()

Set sound volume level Up

See also

[setVolume\(\)](#)

```

01251 {
01252     if (volume < 63)
01253         volume++;
01254     setVolume\(volume\);
01255 }

```

Field Documentation

[si47x_agc_status](#) **SI4735::currentAgcStatus** [[protected](#)]

current RDS status

uint8_t SI4735::currentAvcAmMaxGain = 48 [[protected](#)]

Store the last mode used.

[si47x_frequency](#) **SI4735::currentFrequency** [[protected](#)]

Automatic Volume Control Gain for AM - Default 48.

[si47x_set_frequency](#) **SI4735::currentFrequencyParams** [[protected](#)]

data structure to get current frequency

uint16_t SI4735::currentMaximumFrequency [[protected](#)]

minimum frequency of the current band

uint16_t SI4735::currentMinimumFrequency [[protected](#)]

tell the current tune (FM, AM or SSB)

[si47x_rds_status](#) **SI4735::currentRdsStatus** [[protected](#)]

firmware information

[si47x_rqs_status](#) SI4735::currentRqsStatus [protected]

[si47x_ssb_mode](#) SI4735::currentSSBMode [protected]

current AGC status

uint8_t SI4735::currentSsbStatus [protected]

[si47x_response_status](#) SI4735::currentStatus [protected]

current Radio Signal Quality status

uint16_t SI4735::currentStep [protected]

current frequency

uint8_t SI4735::currentTune [protected]

pin used on Arduino Board to control interrupt. If -1, interrupt is no used.

uint16_t SI4735::currentWorkFrequency [protected]

maximum frequency of the current band

int16_t SI4735::deviceAddress = [SI473X_ADDR_SEN_LOW](#) [protected]

rds_buffer0A current position

[si47x_firmware_information](#) SI4735::firmwareInfo [protected]

current device status

uint8_t SI4735::interruptPin [protected]

pin used on Arduino Board to RESET the Si47XX device

uint8_t SI4735::lastMode = -1 [protected]

current steps

uint8_t SI4735::lastTextFlagAB [protected]

current I2C buss address

[si473x_powerup](#) SI4735::powerUp [protected]

indicates if USB or LSB

char SI4735::rds_buffer0A[9] [protected]

RDS Radio Text buffer - Station Informaation.

Referenced by clearRdsBuffer0A(), and getRdsText0A().

char SI4735::rds_buffer2A[65] [protected]

Referenced by clearRdsBuffer2A(), getRdsText(), and getRdsText2A().

char SI4735::rds_buffer2B[33] [protected]

RDS Radio Text buffer - Program Information.

Referenced by clearRdsBuffer2B(), and getRdsText2B().

char SI4735::rds_time[20] [protected]

RDS Basic tuning and switching information (Type 0 groups)

Referenced by getRdsTime().

int SI4735::rdsTextAddress0A [protected]

rds_buffer2B current position

Referenced by getRdsText0A().

int SI4735::rdsTextAddress2A [protected]

RDS date time received information

Referenced by getRdsText(), and getRdsText2A().

int SI4735::rdsTextAddress2B [protected]

rds_buffer2A current position

Referenced by getRdsText2B().

uint8_t SI4735::resetPin [protected]

uint8_t SI4735::volume = 32 [protected]

Function Documentation

int16_t SI4735::getDeviceI2CAddress (uint8_t resetPin)

I2C bus address setup.

Scans for two possible addresses for the Si47XX (0x11 or 0x63)

This function also sets the system to the found I2C bus address of Si47XX.

You do not need to use this function if the SEN PIN is configured to ground (GND). The default I2C address is 0x11. Use this function if you do not know how the SEN pin is configured.

Parameters

<i>uint8_t</i>	resetPin MCU Mater (Arduino) reset pin
----------------	--

Returns

int16_t 0x11 if the SEN pin of the Si47XX is low or 0x63 if the SEN pin of the Si47XX is HIGH or 0x0 if error.

```
00077                                     {
00078     int16_t error;
```

```

00079
00080     pinMode(resetPin, OUTPUT);
00081     delay(50);
00082     digitalWrite(resetPin, LOW);
00083     delay(50);
00084     digitalWrite(resetPin, HIGH);
00085
00086     Wire.begin();
00087     // check 0X11 I2C address
00088     Wire.beginTransmission(SI473X_ADDR_SEN_LOW);
00089     error = Wire.endTransmission();
00090     if ( error == 0 ) {
00091         setDeviceI2CAddress(0);
00092         return SI473X_ADDR_SEN_LOW;
00093     }
00094
00095     // check 0X63 I2C address
00096     Wire.beginTransmission(SI473X_ADDR_SEN_HIGH);
00097     error = Wire.endTransmission();
00098     if ( error == 0 ) {
00099         setDeviceI2CAddress(1);
00100         return SI473X_ADDR_SEN_HIGH;
00101     }
00102
00103     // Did find the device
00104     return 0;
00105 }

```

void SI4735::setDeviceI2CAddress (uint8_t senPin)

Sets the I2C Bus Address.

The parameter senPin is not the I2C bus address. It is the SEN pin setup of the schematic (eletronic circuit).

If it is connected to the ground, call this function with senPin = 0; else senPin = 1. You do not need to use this function if the SEN PIN configured to ground (GND).

The default value is 0x11 (senPin = 0). In this case you have to ground the pin SEN of the SI473X. If you want to change this address, call this function with senPin = 1

Parameters

<i>senPin</i>	0 - when the pin SEN (16 on SSOP version or pin 6 on QFN version) is set to low (GND - 0V) 1 - when the pin SEN (16 on SSOP version or pin 6 on QFN version) is set to high (+3.3V)
---------------	---

```

00124     {
00125         deviceAddress = (senPin)? SI473X_ADDR_SEN_HIGH : SI473X_ADDR_SEN_LOW;
00126     };

```

void SI4735::setDeviceOtherI2CAddress (uint8_t i2cAddr)

Sets the onther I2C Bus Address (for Si470X)

You can set another I2C address different of 0x11 and 0x63

Parameters

<i>uint8_t</i>	i2cAddr (example 0x10)
----------------	------------------------

```

00137     {
00138         deviceAddress = i2cAddr;
00139     };

```

SI4735::SI4735 ()

Construct a new [SI4735::SI4735](#) object.

```

00036 {
00037     // 1 = LSB and 2 = USB; 0 = AM, FM or WB
00038     currentSsbStatus = 0;

```



```
00039 }
```

void SI4735::waitInterrupr (void) [protected]

Interrupt handle.

If you setup interrupt, this function will be called whenever the Si4735 changes.

```
00055 {  
00056     while (!data_from_si4735)  
00057         ;  
00058 }
```

Host and slave MCU setup

Functions

void [SI4735::reset](#) (void)

Reset the SI473X

void [SI4735::waitToSend](#) (void)

Wait for the si473x is ready (Clear to Send (CTS) status bit have to be 1).

void [SI4735::setPowerUp](#) (uint8_t CTSIEN, uint8_t GPO2OEN, uint8_t PATCH, uint8_t XOSCEN,
uint8_t FUNC, uint8_t OPMODE)

Set the Power Up parameters for si473X.

void [SI4735::radioPowerUp](#) (void)

Powerup the Si47XX.

void [SI4735::analogPowerUp](#) (void)

You have to call setPowerUp method before.

void [SI4735::powerDown](#) (void)

Moves the device from powerup to powerdown mode.

Detailed Description

Function Documentation

void SI4735::analogPowerUp (void)

You have to call setPowerUp method before.

Deprecated:

Consider use `radioPowerUp` instead

See also

[`SI4735::setPowerUp\(\)`](#)

Si47XX PROGRAMMING GUIDE; AN332; pages 64, 129

```
00266 {  
00267     radioPowerUp\(\);  
00268 }
```

References `SI4735::radioPowerUp()`.

void SI4735::powerDown (void)

Moves the device from powerup to powerdown mode.

After Power Down command, only the Power Up command is accepted.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 67, 132

[`radioPowerUp\(\)`](#)

```
00281 {  
00282     waitToSend\(\);  
00283     Wire.beginTransaction(deviceAddress);  
00284     Wire.write(POWER\_DOWN);  
00285     Wire.endTransmission();  
00286     delayMicroseconds(2500);  
00287 }
```

References `SI4735::waitToSend()`.

Referenced by `SI4735::queryLibraryId()`, `SI4735::setAM()`, and `SI4735::setFM()`.

void SI4735::radioPowerUp (void)

Powerup the Si47XX.

Before call this function call the `setPowerUp` to set up the parameters.

Parameters you have to set up with `setPowerUp`

CTSIEN Interrupt anabled or disabled; GPO2OEN GPO2 Output Enable or disabled; PATCH Boot normally or patch; XOSCEN Use external crystal oscillator; FUNC defaultFunction = 0 = FM Receive; 1 = AM (LW/MW/SW) Receiver. OPMODE `SI473X_ANALOG_AUDIO` (B00000101) or `SI473X_DIGITAL_AUDIO` (B00001011)

See also

[`SI4735::setPowerUp\(\)`](#)

Si47XX PROGRAMMING GUIDE; AN332; pages 64, 129

```
00241     {  
00242         // delayMicroseconds(1000);  
00243         waitToSend\(\);  
00244         Wire.beginTransaction(deviceAddress);  
00245         Wire.write(POWER\_UP);  
00246         Wire.write(powerUp.raw[0]); // Content of ARG1  
00247         Wire.write(powerUp.raw[1]); // Content of ARG2  
00248         Wire.endTransmission();  
00249         // Delay at least 500 ms between powerup command and first tune command  
to wait for  
00250         // the oscillator to stabilize if XOSCEN is set and crystal is used as  
the RCLK.  
00251         waitToSend\(\);  
00252         delay(10);  
00253     }
```

References `SI4735::waitToSend()`.

Referenced by SI4735::analogPowerUp(), SI4735::setAM(), SI4735::setFM(), SI4735::setSSB(), and SI4735::setup().

void SI4735::reset (void)

Reset the SI473X

See also

Si47XX PROGRAMMING GUIDE; AN332;

```
00151 {
00152     pinMode(resetPin, OUTPUT);
00153     delay(10);
00154     digitalWrite(resetPin, LOW);
00155     delay(10);
00156     digitalWrite(resetPin, HIGH);
00157     delay(10);
00158 }
```

Referenced by SI4735::setup(), and SI4735::ssbSetup().

void SI4735::setPowerUp (uint8_t CTSIEN, uint8_t GPO2OEN, uint8_t PATCH, uint8_t XOSCEN, uint8_t FUNC, uint8_t OPMODE)

Set the Power Up parameters for si473X.

Use this method to change the default behavior of the Si473X. Use it before PowerUp()

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 65 and 129

Parameters

uint8_t	CTSIEN sets Interrupt anabled or disabled (1 = anabled and 0 = disabled)
uint8_t	GPO2OEN sets GP02 Si473X pin enabled (1 = anabled and 0 = disabled)
uint8_t	PATCH Used for firmware patch updates. Use it always 0 here.
uint8_t	XOSCEN sets external Crystal enabled or disabled
uint8_t	FUNC sets the receiver function have to be used [0 = FM Receive; 1 = AM (LW/MW/SW) and SSB (if SSB patch appllied)]
uint8_t	OPMODE set the kind of audio mode you want to use.

```
00195 {
00196     powerUp.arg.CTSIEN = CTSIEN;    // 1 -> Interrupt anabled;
00197     powerUp.arg.GPO2OEN = GPO2OEN;  // 1 -> GP02 Output Enable;
00198     powerUp.arg.PATCH = PATCH;      // 0 -> Boot normally;
00199     powerUp.arg.XOSCEN = XOSCEN;    // 1 -> Use external crystal oscillator;
00200     powerUp.arg.FUNC = FUNC;        // 0 = FM Receive; 1 = AM/SSB (LW/MW/SW)
Receiver.
00201     powerUp.arg.OPMODE = OPMODE;    // 0x5 = 00000101 = Analog audio outputs
(LOUT/ROUT).
00202
00203     // Set the current tuning frequency mode 0x20 = FM and 0x40 = AM (LW/MW/
SW)
00204     // See See Si47XX PROGRAMMING GUIDE; AN332; pages 55 and 124
00205
00206     if (FUNC == 0)
00207     {
00208         currentTune = FM_TUNE_FREQ;
00209         currentFrequencyParams.arg.FREEZE = 1;
00210     }
00211     else
00212     {
00213         currentTune = AM_TUNE_FREQ;
00214         currentFrequencyParams.arg.FREEZE = 0;
00215     }
00216     currentFrequencyParams.arg.FAST = 1;
```

```

00217     currentFrequencyParams.arg.DUMMY1 = 0;
00218     currentFrequencyParams.arg.ANTCAPH = 0;
00219     currentFrequencyParams.arg.ANTCAPL = 1;
00220 }

```

void SI4735::waitToSend (void)

Wait for the si473x is ready (Clear to Send (CTS) status bit have to be 1).

This function should be used before sending any command to a SI47XX device.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 63, 128

```

00170 {
00171     do
00172     {
00173         delayMicroseconds(MIN\_DELAY\_WAIT\_SEND\_LOOP); // Need check the
minimum value.
00174         Wire.requestFrom(deviceAddress, 1);
00175     } while (!(Wire.read() & B10000000));
00176 }

```

Referenced by SI4735::downloadPatch(), SI4735::getAutomaticGainControl(), SI4735::getCurrentReceivedSignalQuality(), SI4735::getFirmware(), SI4735::getRdsStatus(), SI4735::getStatus(), SI4735::patchPowerUp(), SI4735::powerDown(), SI4735::queryLibraryId(), SI4735::radioPowerUp(), SI4735::seekStation(), SI4735::sendProperty(), SI4735::sendSSBModeProperty(), SI4735::setAutomaticGainControl(), SI4735::setBandwidth(), SI4735::setFrequency(), SI4735::setRdsConfig(), SI4735::setRdsIntSource(), SI4735::setSSBBfo(), and SI4735::ssbPowerUp().

RDS Data types

Data Structures

union [si47x_rqs_status](#)

Radio Signal Quality data representation. [More...](#)

struct [si47x_rqs_status.resp](#)

union [si47x_rds_command](#)

Data type for RDS Status command and response information. [More...](#)

struct [si47x_rds_command.arg](#)

union [si47x_rds_status](#)

Response data type for current channel and reads an entry from the RDS FIFO. [More...](#)

struct [si47x_rds_status.resp](#)

union [si47x_rds_int_source](#)

FM_RDS_INT_SOURCE property data type. [More...](#)

struct [si47x_rds_int_source.refined](#)

union [si47x_rds_config](#)

Data type for FM_RDS_CONFIG Property. [More...](#)

struct [si47x_rds_config.arg](#)

union [si47x_rds_blocka](#)

Block A data type. [More...](#)

struct [si47x_rds_blocka.refined](#)

struct [si47x_rds_blocka.raw](#)

union [si47x_rds_blockb](#)

Block B data type. [More...](#)

struct [si47x_rds_blockb.group0](#)

struct [si47x_rds_blockb.group2](#)

struct [si47x_rds_blockb.refined](#)

struct [si47x_rds_blockb.raw](#)

union [si47x_rds_date_time](#)

struct [si47x_rds_date_time.refined](#)

Detailed Description

Data Structure Documentation

union si47x_rqs_status

Radio Signal Quality data representation.

Data type for status information about the received signal quality (FM_RSQ_STATUS and AM_RSQ_STATUS)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 75 and

Data Fields:

uint8_t	raw[8]	
struct	resp	
si47x_rqs_status		

struct si47x_rqs_status.resp

Data Fields:

uint8_t	AFCRL: 1	Valid Channel.
uint8_t	BLENDINT: 1	
uint8_t	CTS: 1	
uint8_t	DUMMY1: 1	
uint8_t	DUMMY2: 2	
uint8_t	DUMMY3: 1	Multipath Detect High.
uint8_t	DUMMY4: 1	AFC Rail Indicator.
uint8_t	DUMMY5: 4	Soft Mute Indicator. Indicates soft mute is engaged.
uint8_t	ERR: 1	
uint8_t	FREQOFF	RESP6 - Contains the current multipath metric. (0 = no multipath; 100 = full multipath)

uint8_t	MULT	RESP5 - Contains the current SNR metric (0–127 dB).
uint8_t	MULTHINT: 1	Multipath Detect Low.
uint8_t	MULTLINT: 1	SNR Detect High.
uint8_t	PILOT: 1	Indicates amount of stereo blend in% (100 = full stereo, 0 = full mono).
uint8_t	RDSINT: 1	
uint8_t	RSQINT: 1	
uint8_t	RSSI	Indicates stereo pilot presence.
uint8_t	RSSIHINT: 1	RSSI Detect Low.
uint8_t	RSSIILINT: 1	
uint8_t	SMUTE: 1	
uint8_t	SNR	RESP4 - Contains the current receive signal strength (0–127 dB $\frac{1}{4}$ V).
uint8_t	SNRHINT: 1	SNR Detect Low.
uint8_t	SNRLINT: 1	RSSI Detect High.
uint8_t	STBLEND: 7	
uint8_t	STCINT: 1	
uint8_t	VALID: 1	Blend Detect Interrupt.

union si47x_rds_command

Data type for RDS Status command and response information.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 77 and 78

Also https://en.wikipedia.org/wiki/Radio_Data_System

Data Fields:

struct si47x_rds_command	arg	
uint8_t	raw	

struct si47x_rds_command.arg

Data Fields:

uint8_t	dummy: 5	
uint8_t	INTACK: 1	
uint8_t	MTFIFO: 1	
uint8_t	STATUSONLY: 1	

union si47x_rds_status

Response data type for current channel and reads an entry from the RDS FIFO.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 77 and 78

Data Fields:

uint8_t	raw[13]	
struct	resp	
si47x_rds_status		

struct si47x_rds_status.resp**Data Fields:**

uint8_t	BLEA: 2	
uint8_t	BLEB: 2	
uint8_t	BLEC: 2	
uint8_t	BLED: 2	RESP11 - RDS Block D; LOW byte.
uint8_t	BLOCKAH	RESP3 - RDS FIFO Used; Number of groups remaining in the RDS FIFO (0 if empty).
uint8_t	BLOCKAL	RESP4 - RDS Block A; HIGH byte.
uint8_t	BLOCKBH	RESP5 - RDS Block A; LOW byte.
uint8_t	BLOCKBL	RESP6 - RDS Block B; HIGH byte.
uint8_t	BLOCKCH	RESP7 - RDS Block B; LOW byte.
uint8_t	BLOCKCL	RESP8 - RDS Block C; HIGH byte.
uint8_t	BLOCKDH	RESP9 - RDS Block C; LOW byte.
uint8_t	BLOCKDL	RESP10 - RDS Block D; HIGH byte.
uint8_t	CTS: 1	
uint8_t	DUMMY1: 1	
uint8_t	DUMMY2: 2	
uint8_t	DUMMY3: 1	RDS Sync Found; 1 = Found RDS synchronization.
uint8_t	DUMMY4: 2	RDS New Block B; 1 = Valid Block B data has been received.
uint8_t	DUMMY5: 1	RDS Sync; 1 = RDS currently synchronized.
uint8_t	DUMMY6: 5	Group Lost; 1 = One or more RDS groups discarded due to FIFO overrun.
uint8_t	ERR: 1	
uint8_t	GRPLOST: 1	
uint8_t	RDSFIFOUSED	
uint8_t	RDSINT: 1	
uint8_t	RDSNEWBLOCKA: 1	
uint8_t	RDSNEWBLOCKB: 1	RDS New Block A; 1 = Valid Block A data has been received.
uint8_t	RDSRECV: 1	
uint8_t	RDSSYNC: 1	
uint8_t	RDSSYNCFFOUND: 1	RDS Sync Lost; 1 = Lost RDS synchronization.
uint8_t	RDSSYNCLOST: 1	RDS Received; 1 = FIFO filled to minimum number of groups set by RDSFIFOCNT.
uint8_t	RSQINT: 1	
uint8_t	STCINT: 1	

union si47x_rds_int_source

FM_RDS_INT_SOURCE property data type.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 103

also https://en.wikipedia.org/wiki/Radio_Data_System

Data Fields:

uint8_t	raw[2]	
struct si47x_rds_int_source ce	refined	

struct si47x_rds_int_source.refined**Data Fields:**

uint8_t	DUMMY1: 1	f set, generate RDSINT when RDS gains synchronization.
uint8_t	DUMMY2: 5	If set, generate an interrupt when Block B data is found or subsequently changed.
uint8_t	DUMMY3: 5	Reserved - Always write to 0.
uint8_t	RDSNEWBLOCKA: 1	Always write to 0.
uint8_t	RDSNEWBLOCKB: 1	If set, generate an interrupt when Block A data is found or subsequently changed.
uint8_t	RDSRECV: 1	
uint8_t	RDSSYNCFOUND: 1	If set, generate RDSINT when RDS loses synchronization.
uint8_t	RDSSYNCLIST: 1	If set, generate RDSINT when RDS FIFO has at least FM_RDS_INT_FIFO_COUNT entries.

union si47x_rds_config

Data type for FM_RDS_CONFIG Property.

IMPORTANT: all block errors must be less than or equal the associated block error threshold for the group to be stored in the RDS FIFO. 0 = No errors; 1 = 1–2 bit errors detected and corrected; 2 = 3–5 bit errors detected and corrected; 3 = Uncorrectable. Recommended Block Error Threshold options: 2,2,2,2 = No group stored if any errors are uncorrected. 3,3,3,3 = Group stored regardless of errors. 0,0,0,0 = No group stored containing corrected or uncorrected errors. 3,2,3,3 = Group stored with corrected errors on B, regardless of errors on A, C, or D.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 58 and 104

Data Fields:

struct si47x_rds_config	arg	
uint8_t	raw[2]	

struct si47x_rds_config.arg

Data Fields:

uint8_t	BLETHA: 2	Block Error Threshold BLOCKB.
uint8_t	BLETHB: 2	Block Error Threshold BLOCKC.
uint8_t	BLETHC: 2	Block Error Threshold BLOCKD.
uint8_t	BLETHD: 2	
uint8_t	DUMMY1: 7	1 = RDS Processing Enable.
uint8_t	RDSEN: 1	

union si47x_rds_blocka

Block A data type.

Data Fields:

struct si47x_rds_blocka	raw	
struct si47x_rds_blocka	refined	

struct si47x_rds_blocka.refined**Data Fields:**

uint16_t	pi	
----------	----	--

struct si47x_rds_blocka.raw**Data Fields:**

uint8_t	highValue	
uint8_t	lowValue	

union si47x_rds_blockb

Block B data type.

For GCC on System-V ABI on 386-compatible (32-bit processors), the following stands:

1) Bit-fields are allocated from right to left (least to most significant). 2) A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus a bit-field never crosses its unit boundary. 3) Bit-fields may share a storage unit with other struct/union members, including members that are not bit-fields. Of course, struct members occupy different parts of the storage unit. 4) Unnamed bit-fields' types do not affect the alignment of a structure or union, although individual bit-fields' member offsets obey the alignment constraints.

See also

also Si47XX PROGRAMMING GUIDE; AN332; pages 78 and 79

also https://en.wikipedia.org/wiki/Radio_Data_System

Data Fields:

struct si47x_rds_blockb	group0	
struct si47x_rds_blockb	group2	
struct si47x_rds_blockb	raw	
struct si47x_rds_blockb	refined	

struct si47x_rds_blockb.group0

Data Fields:

uint16_t	address: 2	
uint16_t	DI: 1	
uint16_t	groupType: 4	
uint16_t	MS: 1	
uint16_t	programType: 5	
uint16_t	TA: 1	
uint16_t	trafficProgramCode: 1	
uint16_t	versionCode: 1	

struct si47x_rds_blockb.group2

Data Fields:

uint16_t	address: 4	
uint16_t	groupType: 4	
uint16_t	programType: 5	
uint16_t	textABFlag: 1	
uint16_t	trafficProgramCode: 1	
uint16_t	versionCode: 1	

struct si47x_rds_blockb.refined

Data Fields:

uint16_t	content: 4	
uint16_t	groupType: 4	
uint16_t	programType: 5	
uint16_t	textABFlag: 1	
uint16_t	trafficProgramCode: 1	
uint16_t	versionCode: 1	

struct si47x_rds_blockb.raw

Data Fields:

uint8_t	highValue	
uint8_t	lowValue	

union si47x_rds_date_time

Group type 4A (RDS Date and Time) When group type 4A is used by the station, it shall be transmitted every minute according to EN 50067. This Structure uses blocks 2,3 and 5 (B,C,D)

ATTENTION: To make it compatible with 8, 16 and 32 bits platforms and avoid Crosses boundary, it was necessary to split minute and hour representation.

Data Fields:

uint8_t	raw[6]	
struct si47x_rds_date_time	refined	

struct si47x_rds_date_time.refined

Data Fields:

uint8_t	hour1: 4	
uint8_t	hour2: 1	
uint8_t	minute1: 2	
uint8_t	minute2: 4	
uint32_t	mjd: 17	
uint8_t	offset: 5	
uint8_t	offset_sense: 1	

Receiver Status and Setup

Data Structuresunion [si47x_agc_status](#)struct [si47x_agc_status.refined](#)union [si47x_agc_override](#)struct [si47x_agc_override.arg](#)union [si47x_bandwidth_config](#)struct [si47x_bandwidth_config.param](#)union [si47x_ssb_mode](#)struct [si47x_ssb_mode.param](#)union [si4735_digital_output_format](#)

Digital audio output format data structure (Property 0x0102. DIGITAL_OUTPUT_FORMAT). [More...](#)

struct [si4735_digital_output_format.refined](#)struct [si4735_digital_output_sample_rate](#)

Digital audio output sample structure (Property 0x0104. DIGITAL_OUTPUT_SAMPLE_RATE). [More...](#)

Detailed Description

Data Structure Documentation**union si47x_agc_status**

AGC data types FM / AM and SSB structure to AGC

See also

Si47XX PROGRAMMING GUIDE; AN332; For FM page 80; for AM page 142

AN332 REV 0.8 Universal Programming Guide Amendment for SI4735-D60 SSB and NBFM patches; page 18.

Data Fields:

uint8_t	raw[3]	
struct si47x_agc_status	refined	

struct si47x_agc_status.refined

Data Fields:

uint8_t	AGCDIS: 1	
uint8_t	AGCIDX	
uint8_t	CTS: 1	
uint8_t	DUMMY: 7	
uint8_t	DUMMY1: 1	
uint8_t	DUMMY2: 2	
uint8_t	ERR: 1	
uint8_t	RDSINT: 1	
uint8_t	RSQINT: 1	
uint8_t	STCINT: 1	

union si47x_agc_override

If FM, Overrides AGC setting by disabling the AGC and forcing the LNA to have a certain gain that ranges between 0 (minimum attenuation) and 26 (maximum attenuation). If AM, overrides the AGC setting by disabling the AGC and forcing the gain index that ranges between 0

See also

Si47XX PROGRAMMING GUIDE; AN332; For FM page 81; for AM page 143

Data Fields:

struct si47x_agc_override	arg	
uint8_t	raw[2]	

struct si47x_agc_override.arg**Data Fields:**

uint8_t	AGCDIS: 1	
uint8_t	AGCIDX	
uint8_t	DUMMY: 7	

union si47x_bandwidth_config

The bandwidth of the AM channel filter data type AMCHFLT values: 0 = 6 kHz Bandwidth

1 = 4 kHz Bandwidth 2 = 3 kHz Bandwidth 3 = 2 kHz Bandwidth 4 = 1 kHz Bandwidth 5 = 1.8 kHz Bandwidth 6 = 2.5 kHz Bandwidth, gradual roll off 7–15 = Reserved (Do not use)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 125 and 151

Data Fields:

struct si47x_bandwidth_config	param	
uint8_t	raw[2]	

struct si47x_bandwidth_config.param**Data Fields:**

uint8_t	AMCHFLT: 4	
uint8_t	AMPLFLT: 1	
uint8_t	DUMMY1: 4	Selects the bandwidth of the AM channel filter.

uint8_t	DUMMY2: 7	Enables the AM Power Line Noise Rejection Filter.
---------	-----------	---

union si47x_ssb_mode

SSB - datatype for SSB_MODE (property 0x0101)

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Data Fields:

struct si47x_ssb_mode	param	
uint8_t	raw[2]	

struct si47x_ssb_mode.param

Data Fields:

uint8_t	AUDIOBW: 4	
uint8_t	AVC_DIVIDER: 4	SSB side band cutoff filter for band pass and low pass filter.
uint8_t	AVCEN: 1	set 0 for SSB mode; set 3 for SYNC mode;
uint8_t	DSP_AFCDIS: 1	Always write 0;.
uint8_t	DUMMY1: 1	SSB Soft-mute Based on RSSI or SNR.
uint8_t	SBCUTFLT: 4	0 = 1.2KHz (default); 1=2.2KHz; 2=3KHz; 3=4KHz; 4=500Hz; 5=1KHz
uint8_t	SMUTESEL: 1	SSB Automatic Volume Control (AVC) enable; 0=disable; 1=enable (default);.

union si4735_digital_output_format

Digital audio output format data structure (Property 0x0102. DIGITAL_OUTPUT_FORMAT).

Used to configure: DCLK edge, data format, force mono, and sample precision.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 195.

Data Fields:

uint16_t	raw	
struct si4735_digital_output_format	refined	

struct si4735_digital_output_format.refined

Data Fields:

uint8_t	dummy: 8	Digital Output DCLK Edge (0 = use DCLK rising edge, 1 = use DCLK falling edge)
uint8_t	OFALL: 1	Digital Output Mode (0000=I2S, 0110 = Left-justified, 1000 = MSB at second DCLK after DFS pulse, 1100 = MSB at first DCLK after

		DFS pulse).
uint8_t	OMODE: 4	Digital Output Mono Mode (0=Use mono/stereo blend).
uint8_t	OMONO: 1	Digital Output Audio Sample Precision (0=16 bits, 1=20 bits, 2=24 bits, 3=8bits).
uint8_t	OSIZE: 2	

struct si4735_digital_output_sample_rate

Digital audio output sample structure (Property 0x0104. DIGITAL_OUTPUT_SAMPLE_RATE).

Used to enable digital audio output and to configure the digital audio output sample rate in samples per second (sps).

See also

Si47XX PROGRAMMING GUIDE; AN332; page 196.

Data Fields:

uint16_t	DOSR	
----------	------	--

SI473X data types

SI473X data representation.

Data Structures

union [si473x_powerup](#)

Power Up arguments data type. [More...](#)

struct [si473x_powerup.arg](#)

union [si47x_frequency](#)

Represents how the frequency is stored in the si4735. [More...](#)

struct [si47x_frequency.raw](#)

union [si47x_antenna_capacitor](#)

Antenna Tuning Capacitor data type manipulation. [More...](#)

struct [si47x_antenna_capacitor.raw](#)

union [si47x_set_frequency](#)

AM Tune frequency data type command (AM_TUNE_FREQ command) [More...](#)

struct [si47x_set_frequency.arg](#)

union [si47x_seek](#)

Seek frequency (automatic tuning) [More...](#)

struct [si47x_seek.arg](#)

union [si47x_response_status](#)

Response status command. [More...](#)

struct [si47x_response_status.resp](#)

union [si47x_firmware_information](#)

Data representation for Firmware Information (GET_REV) [More...](#)

struct [si47x_firmware_information.resp](#)

union [si47x_firmware_query_library](#)

Firmware Query Library ID response. [More...](#)

struct [si47x_firmware_query_library.resp](#)

union [si47x_tune_status](#)

Seek station status. [More...](#)

struct [si47x_tune_status.arg](#)

union [si47x_property](#)

Data type to deal with SET_PROPERTY command. [More...](#)

struct [si47x_property.raw](#)

Detailed Description

SI473X data representation.

The goal here is separate data from code. The Si47XX family works with many internal data that can be represented by data structure or defined data type in C/C++. These C/C++ resources have been used widely here.

This approach made the library easier to build and maintain. Each data structure created here has its reference (name of the document and page on which it was based). In other words, to make the SI47XX device easier to deal, some defined data types were created to handle byte and bits to process commands, properties and responses. These data types will be usefull to deal with SI473X

Data Structure Documentation

union si473x_powerup

Power Up arguments data type.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 64 and 65

Data Fields:

struct	arg	
si473x_powerup		
uint8_t	raw[2]	

struct si473x_powerup.arg

Data Fields:

uint8_t	CTSIE: 1	GPO2 Output Enable (0 = GPO2 output disabled; 1 = GPO2 output enabled).
uint8_t	FUNC: 4	

uint8_t	GPO2OEN: 1	Patch Enable (0 = Boot normally; 1 = Copy non-volatile memory to RAM).
uint8_t	OPMODE	CTS Interrupt Enable (0 = CTS interrupt disabled; 1 = CTS interrupt enabled).
uint8_t	PATCH: 1	Crystal Oscillator Enable (0 = crystal oscillator disabled; 1 = Use crystal oscillator and OPMODE=ANALOG AUDIO) .
uint8_t	XOSCEN: 1	Function (0 = FM Receive; 1–14 = Reserved; 15 = Query Library ID)

union si47x_frequency

Represents how the frequency is stored in the si4735.

It helps to convert frequency in uint16_t to two bytes (uint8_t) (FREQL and FREQH)

Data Fields:

struct si47x_frequency	raw	
uint16_t	value	

struct si47x_frequency.raw

Data Fields:

uint8_t	FREQH	Tune Frequency High byte.
uint8_t	FREQL	

union si47x_antenna_capacitor

Antenna Tuning Capacitor data type manipulation.

Data Fields:

struct si47x_antenna_capacitor	raw	
uint16_t	value	

struct si47x_antenna_capacitor.raw

Data Fields:

uint8_t	ANTCAPH	Antenna Tuning Capacitor High byte.
uint8_t	ANTCAPL	

union si47x_set_frequency

AM Tune frequency data type command (AM_TUNE_FREQ command)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 135

Data Fields:

struct si47x_set_frequency	arg	
uint8_t	raw[5]	

struct si47x_set_frequency.arg**Data Fields:**

uint8_t	ANTCAPH	ARG3 - Tune Frequency Low byte.
uint8_t	ANTCAPL	ARG4 - Antenna Tuning Capacitor High byte.
uint8_t	DUMMY1: 4	Valid only for FM (Must be 0 to AM)
uint8_t	FAST: 1	
uint8_t	FREEZE: 1	ARG1 - FAST Tuning. If set, executes fast and invalidated tune. The tune status will not be accurate.
uint8_t	FREQH	SSB Upper Side Band (USB) and Lower Side Band (LSB) Selection. 10 = USB is selected; 01 = LSB is selected.
uint8_t	FREQL	ARG2 - Tune Frequency High byte.
uint8_t	USBLSB: 2	Always set 0.

union si47x_seek

Seek frequency (automatic tuning)

Represents searching for a valid frequency data type.

Data Fields:

struct si47x_seek	arg	
uint8_t	raw	

struct si47x_seek.arg**Data Fields:**

uint8_t	RESERVED1: 2	
uint8_t	RESERVED2: 4	Determines the direction of the search, either UP = 1, or DOWN = 0.
uint8_t	SEEKUP: 1	Determines whether the seek should Wrap = 1, or Halt = 0 when it hits the band limit.
uint8_t	WRAP: 1	

union si47x_response_status

Response status command.

Response data from a query status command

See also

Si47XX PROGRAMMING GUIDE; pages 73 and

Data Fields:

uint8_t	raw[8]	
---------	--------	--

struct si47x_response_status	resp	
---	------	--

struct si47x_response_status.resp

Data Fields:

uint8_t	AFCRL: 1	Valid Channel.
uint8_t	BLTF: 1	
uint8_t	CTS: 1	Error. 0 = No error 1 = Error.
uint8_t	DUMMY1: 1	Seek/Tune Complete Interrupt; 1 = Tune complete has been triggered.
uint8_t	DUMMY2: 2	Received Signal Quality Interrupt; 0 = interrupt has not been triggered.
uint8_t	DUMMY3: 5	AFC Rail Indicator.
uint8_t	ERR: 1	
uint8_t	MULT	This byte contains the SNR metric when tune is complete (dB).
uint8_t	RDSINT: 1	
uint8_t	READANTCAP	Contains the multipath metric when tune is complete.
uint8_t	READFREQH	Reports if a seek hit the band limit.
uint8_t	READFREQL	Read Frequency High byte.
uint8_t	RSQINT: 1	Radio Data System (RDS) Interrupt; 0 = interrupt has not been triggered.
uint8_t	RSSI	Read Frequency Low byte.
uint8_t	SNR	Received Signal Strength Indicator (dB $\frac{1}{4}$ V)
uint8_t	STCINT: 1	
uint8_t	VALID: 1	Clear to Send.

union si47x_firmware_information

Data representation for Firmware Information (GET_REV)

The part number, chip revision, firmware revision, patch revision and component revision numbers.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 66 and 131

Data Fields:

uint8_t	raw[9]	
struct si47x_firmware_information	resp	

struct si47x_firmware_information.resp

Data Fields:

uint8_t	CHIPREV	RESP7 - Component Minor Revision (ASCII).
---------	---------	---

uint8_t	CMPMAJOR	RESP5 - Patch ID Low byte (HEX).
uint8_t	CMPMINOR	RESP6 - Component Major Revision (ASCII).
uint8_t	CTS: 1	
uint8_t	DUMMY1: 1	
uint8_t	DUMMY2: 2	
uint8_t	ERR: 1	
uint8_t	FWMAJOR	RESP1 - Final 2 digits of Part Number (HEX).
uint8_t	FWMINOR	RESP2 - Firmware Major Revision (ASCII).
uint8_t	PATCHH	RESP3 - Firmware Minor Revision (ASCII).
uint8_t	PATCHL	RESP4 - Patch ID High byte (HEX).
uint8_t	PN	
uint8_t	RDSINT: 1	
uint8_t	RSQINT: 1	
uint8_t	STCINT: 1	

union si47x_firmware_query_library

Firmware Query Library ID response.

Used to represent the response of a power up command with FUNC = 15 (patch)

To confirm that the patch is compatible with the internal device library revision, the library revision should be confirmed by issuing the POWER_UP command with Function = 15 (query library ID)

See also

Si47XX PROGRAMMING GUIDE; AN332; page 12

Data Fields:

uint8_t	raw[8]	
struct	resp	
si47x_firmware_query_library		

struct si47x_firmware_query_library.resp

Data Fields:

uint8_t	CHIPREV	RESP5 - Reserved, various values.
uint8_t	CTS: 1	
uint8_t	DUMMY1: 1	
uint8_t	DUMMY2: 2	
uint8_t	ERR: 1	
uint8_t	FWMAJOR	RESP1 - Final 2 digits of Part Number (HEX).
uint8_t	FWMINOR	RESP2 - Firmware Major Revision (ASCII).
uint8_t	LIBRARYID	RESP6 - Chip Revision (ASCII).
uint8_t	PN	
uint8_t	RDSINT: 1	
uint8_t	RESERVED1	RESP3 - Firmware Minor Revision (ASCII).
uint8_t	RESERVED2	RESP4 - Reserved, various values.
uint8_t	RSQINT: 1	
uint8_t	STCINT: 1	

union si47x_tune_status

Seek station status.

Status of FM_TUNE_FREQ or FM_SEEK_START commands or Status of AM_TUNE_FREQ or AM_SEEK_START commands.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 73 and 139

Data Fields:

struct si47x_tune_status	arg	
uint8_t	raw	

struct si47x_tune_status.arg

Data Fields:

uint8_t	CANCEL: 1	If set, clears the seek/tune complete interrupt status indicator.
uint8_t	INTACK: 1	
uint8_t	RESERVED2: 6	If set, aborts a seek currently in progress.

union si47x_property

Data type to deal with SET_PROPERTY command.

Property Data type (help to deal with SET_PROPERTY command on si473X)

Data Fields:

struct si47x_property	raw	
uint16_t	value	

struct si47x_property.raw

Data Fields:

uint8_t	byteHigh	
uint8_t	byteLow	

Si47XX device Mode, Band and Frequency setup

Functions

void [SI4735::setTuneFrequencyAntennaCapacitor](#) (uint16_t capacitor)

Only FM. Freeze Metrics During Alternate Frequency Jump.

void [SI4735::setFrequency](#) (uint16_t)

Set the frequency to the current function of the Si4735 (FM, AM or SSB)

void [SI4735::setFrequencyStep](#) (uint16_t step)

Sets the current step value.

void [SI4735::frequencyUp](#) ()

Increments the current frequency on current band/function by using the current step.

void [SI4735::frequencyDown](#) ()

Decrements the current frequency on current band/function by using the current step.

void [SI4735::setAM](#) ()

Sets the radio to AM function. It means: LW MW and SW.

void [SI4735::setFM](#) ()

Sets the radio to FM function.

void [SI4735::setAM](#) (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)

Sets the radio to AM (LW/MW/SW) function.

void [SI4735::setFM](#) (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)

Sets the radio to FM function.

Detailed Description

Function Documentation

void SI4735::frequencyDown ()

Decrements the current frequency on current band/function by using the current step.

See also

[setFrequencyStep\(\)](#)

```
00506 {
00507
00508     if (currentWorkFrequency <= currentMinimumFrequency)
00509         currentWorkFrequency = currentMaximumFrequency;
00510     else
00511         currentWorkFrequency -= currentStep;
00512
00513     setFrequency(currentWorkFrequency) ;
00514 }
```

void SI4735::frequencyUp ()

Increments the current frequency on current band/function by using the current step.

See also

[setFrequencyStep\(\)](#)

```
00489 {
00490     if (currentWorkFrequency >= currentMaximumFrequency)
00491         currentWorkFrequency = currentMinimumFrequency;
00492     else
```

```

00493         currentWorkFrequency += currentStep;
00494
00495         setFrequency(currentWorkFrequency);
00496     }

```

void SI4735::setAM ()

Sets the radio to AM function. It means: LW MW and SW.

Define the band range you want to use for the AM mode.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 129.

```

00526 {
00527     // If you're already using AM mode, it is not necessary to call
powerDown and radioPowerUp.
00528     // The other properties also should have the same value as the previous
status.
00529     if ( lastMode != AM\_CURRENT\_MODE ) {
00530         powerDown();
00531         setPowerUp(1, 1, 0, 1, 1, SI473X\_ANALOG\_AUDIO);
00532         radioPowerUp();
00533         setAvcAmMaxGain(currentAvcAmMaxGain); // Set AM Automatic Volume
Gain to 48
00534         setVolume(volume); // Set to previous configured volume
00535     }
00536     currentSsbStatus = 0;
00537     lastMode = AM\_CURRENT\_MODE;
00538 }

```

References SI4735::powerDown(), and SI4735::radioPowerUp().

Referenced by SI4735::setAM().

void SI4735::setAM (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)

Sets the radio to AM (LW/MW/SW) function.

See also

[setAM\(\)](#)

Parameters

<i>fromFreq</i>	minimum frequency for the band
<i>toFreq</i>	maximum frequency for the band
<i>initialFreq</i>	initial frequency
<i>step</i>	step used to go to the next channel

```

00571 {
00572
00573     currentMinimumFrequency = fromFreq;
00574     currentMaximumFrequency = toFreq;
00575     currentStep = step;
00576
00577     if (initialFreq < fromFreq || initialFreq > toFreq)
00578         initialFreq = fromFreq;
00579
00580     setAM();
00581     currentWorkFrequency = initialFreq;
00582     setFrequency(currentWorkFrequency);
00583 }

```

References SI4735::setAM().

void SI4735::setFM ()

Sets the radio to FM function.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 64.

```
00548 {
00549     powerDown();
00550     setPowerUp(1, 1, 0, 1, 0, SI473X\_ANALOG\_AUDIO);
00551     radioPowerUp();
00552     setVolume(volume); // Set to previous configured volume
00553     currentSsbStatus = 0;
00554     disableFmDebug();
00555     lastMode = FM\_CURRENT\_MODE;
00556 }
```

References [SI4735::disableFmDebug\(\)](#), [SI4735::powerDown\(\)](#), and [SI4735::radioPowerUp\(\)](#).

Referenced by [SI4735::setFM\(\)](#).

void SI4735::setFM (uint16_t fromFreq, uint16_t toFreq, uint16_t initialFreq, uint16_t step)

Sets the radio to FM function.

Defines the band range you want to use for the FM mode.

See also

[setFM\(\)](#)

Parameters

<i>fromFreq</i>	minimum frequency for the band
<i>toFreq</i>	maximum frequency for the band
<i>initialFreq</i>	initial frequency (default frequency)
<i>step</i>	step used to go to the next channel

```
00600 {
00601     currentMinimumFrequency = fromFreq;
00602     currentMaximumFrequency = toFreq;
00603     currentStep = step;
00604
00605     if (initialFreq < fromFreq || initialFreq > toFreq)
00606         initialFreq = fromFreq;
00607
00608     setFM();
00609
00610     currentWorkFrequency = initialFreq;
00611     setFrequency(currentWorkFrequency);
00612 }
00613 }
```

References [SI4735::setFM\(\)](#).

void SI4735::setFrequency (uint16_t freq)

Set the frequency to the current function of the Si4735 (FM, AM or SSB)

You have to call [setup](#) or [setPowerUp](#) before call [setFrequency](#).

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 70, 135

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 13

Parameters

<i>uint16_t</i>	freq Is the frequency to change. For example, FM => 10390 = 103.9 MHz; AM => 810 = 810 KHz.
-----------------	---

```

00435 {
00436     waitToSend\(\); // Wait for the si473x is ready.
00437     currentFrequency.value = freq;
00438     currentFrequencyParams.arg.FREQH = currentFrequency.raw.FREQH;
00439     currentFrequencyParams.arg.FREQL = currentFrequency.raw.FREQL;
00440
00441     if (currentSsbStatus != 0)
00442     {
00443         currentFrequencyParams.arg.DUMMY1 = 0;
00444         currentFrequencyParams.arg.USBLSB = currentSsbStatus; // Set to LSB
00445     or USB
00446         currentFrequencyParams.arg.FAST = 1; // Used just
00447 on AM and FM
00448         currentFrequencyParams.arg.FREEZE = 0; // Used just
00449 on FM
00450     }
00451     Wire.beginTransaction(deviceAddress);
00452     Wire.write(currentTune);
00453     Wire.write(currentFrequencyParams.raw[0]); // Send a byte with FAST and
00454 FREEZE information; if not FM must be 0;
00455     Wire.write(currentFrequencyParams.arg.FREQH);
00456     Wire.write(currentFrequencyParams.arg.FREQL);
00457     Wire.write(currentFrequencyParams.arg.ANTCAPH);
00458 // If current tune is not FM sent one more byte
00459 if (currentTune != FM_TUNE_FREQ)
00460     Wire.write(currentFrequencyParams.arg.ANTCAPL);
00461 Wire.endTransmission();
00462 waitToSend\(\); // Wait for the si473x is ready.
00463 currentWorkFrequency = freq; // check it
00464 delay(MAX\_DELAY\_AFTER\_SET\_FREQUENCY); // For some reason I need to delay
00465 here.
00466 }
00467
References SI4735::waitToSend().

```

void SI4735::setFrequencyStep (uint16_t step)

Sets the current step value.

This function does not check the limits of the current band. Please, don't take a step bigger than your legs.

Parameters

<i>step</i>	if you are using FM, 10 means 100KHz. If you are using AM 10 means 10KHz For AM, 1 (1KHz) to 1000 (1MHz) are valid values. For FM 5 (50KHz) and 10 (100KHz) are valid values.
-------------	--

```

00477 {
00478     currentStep = step;
00479 }

```

void SI4735::setTuneFrequencyAntennaCapacitor (uint16_t capacitor)

Only FM. Freeze Metrics During Alternate Frequency Jump.

Selects the tuning capacitor value.

For FM, Antenna Tuning Capacitor is valid only when using TXO/LPI pin as the antenna input.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 71 and 136

Parameters

<i>capacitor</i>	If zero, the tuning capacitor value is selected automatically. If the value is set to anything other than 0: AM - the tuning capacitance is manually set as 95 fF
------------------	---

	<p>x ANTCAP + 7 pF. ANTCAP manual range is 1–6143; FM - the valid range is 0 to 191.</p> <p>According to Silicon Labs, automatic capacitor tuning is recommended (value 0).</p>
--	---

```

00399 {
00400     si47x\_antenna\_capacitor cap;
00401
00402     cap.value = capacitor;
00403
00404     currentFrequencyParams.arg.DUMMY1 = 0;
00405
00406     if (currentTune == FM\_TUNE\_FREQ)
00407     {
00408         // For FM, the capacitor value has just one byte
00409         currentFrequencyParams.arg.ANTCAPH = (capacitor <= 191) ?
cap.raw.ANTCAPL : 0;
00410     }
00411     else
00412     {
00413         if (capacitor <= 6143)
00414         {
00415             currentFrequencyParams.arg.FREEZE = 0; // This parameter is not
used for AM
00416             currentFrequencyParams.arg.ANTCAPH = cap.raw.ANTCAPH;
00417             currentFrequencyParams.arg.ANTCAPL = cap.raw.ANTCAPL;
00418         }
00419     }
00420 }

```

Si47XX device information and start up

Functions

void [SI4735::getFirmware](#) (void)
Gets firmware information.

void [SI4735::setup](#) (uint8_t [resetPin](#), int [interruptPin](#), uint8_t defaultFunction, uint8_t
audioMode=[SI473X_ANALOG_AUDIO](#))
Starts the Si473X device.

void [SI4735::setup](#) (uint8_t [resetPin](#), uint8_t defaultFunction)
Starts the Si473X device.

Detailed Description

Function Documentation

void SI4735::getFirmware (void)

Gets firmware information.

See also

SI47XX PROGRAMMING GUIDE; AN332; pages 66, 131

```
00299 {
00300     waitToSend\(\);
00301
00302     Wire.beginTransaction(deviceAddress);
00303     Wire.write(GET\_REV);
00304     Wire.endTransmission();
00305
00306     do
00307     {
00308         waitToSend\(\);
00309         // Request for 9 bytes response
00310         Wire.requestFrom(deviceAddress, 9);
00311         for (int i = 0; i < 9; i++)
00312             firmwareInfo.raw[i] = Wire.read();
00313     } while (firmwareInfo.resp.ERR);
00314 }
```

References SI4735::waitToSend().

Referenced by SI4735::setup().

void SI4735::setup (uint8_t resetPin, int interruptPin, uint8_t defaultFunction, uint8_t audioMode = [SI473X_ANALOG_AUDIO](#))

Starts the SI473X device.

If the audio mode parameter is not entered, analog mode will be considered.

Parameters

<i>uint8_t</i>	resetPin Digital Arduino Pin used to RESET command
<i>uint8_t</i>	interruptPin interrupt Arduino Pin (see your Arduino pinout). If less than 0, interrupt disabled
<i>uint8_t</i>	defaultFunction
<i>uint8_t</i>	audioMode default SI473X_ANALOG_AUDIO (Analog Audio). Use SI473X_ANALOG_AUDIO or SI473X_DIGITAL_AUDIO

```
00329 {
00330     uint8_t interruptEnable = 0;
00331     Wire.begin();
00332
00333     this->resetPin = resetPin;
00334     this->interruptPin = interruptPin;
00335
00336     // Arduino interrupt setup (you have to know which Arduino Pins can deal
with interrupt).
00337     if (interruptPin >= 0)
00338     {
00339         pinMode(interruptPin, INPUT);
00340         attachInterrupt(digitalPinToInterrupt(interruptPin),
interrupt_hundler, RISING);
00341         interruptEnable = 1;
00342     }
00343
00344     pinMode(resetPin, OUTPUT);
00345     digitalWrite(resetPin, HIGH);
00346
00347     data_from_si4735 = false;
00348
00349     // Set the initial SI473X behavior
00350     // CTSIEN  1 -> Interrupt enabled or disabled;
00351     // GPO2OEN 1 -> GPO2 Output Enable;
00352     // PATCH   0 -> Boot normally;
00353     // XOSCEN  1 -> Use external crystal oscillator;
00354     // FUNC     defaultFunction = 0 = FM Receive; 1 = AM (LW/MW/SW)
Receiver.
00355     // OPMODE  SI473X_ANALOG_AUDIO or SI473X_DIGITAL_AUDIO.
00356     setPowerUp(interruptEnable, 0, 0, 1, defaultFunction, audioMode);
00357
00358     reset\(\);
00359     radioPowerUp\(\);
```

```

00360     setVolume(30); // Default volume level.
00361     getFirmware();
00362 }

```

References `SI4735::getFirmware()`, `SI4735::radioPowerUp()`, and `SI4735::reset()`.

`void SI4735::setup (uint8_t resetPin, uint8_t defaultFunction)`

Starts the Si473X device.

Use this setup if you are not using interrupt resource

Parameters

<code>uint8_t</code>	resetPin Digital Arduino Pin used to RESET command
<code>uint8_t</code>	defaultFunction

```

00375 {
00376     setup(resetPin, -1, defaultFunction);
00377     delay(250);
00378 }

```

File Documentation

SI4735/SI4735.cpp File Reference

```
#include <SI4735.h>
```

SI4735/SI4735.h File Reference

```
#include <Arduino.h>
#include <Wire.h>
```

Data Structures

union [si473x_powerup](#)

Power Up arguments data type. [More...](#)

union [si47x_frequency](#)

Represents how the frequency is stored in the si4735. [More...](#)

union [si47x_antenna_capacitor](#)

Antenna Tuning Capacitor data type manipulation. [More...](#)

union [si47x_set_frequency](#)

AM Tune frequency data type command (AM_TUNE_FREQ command) [More...](#)

union [si47x_seek](#)

Seek frequency (automatic tuning) [More...](#)

union [si47x_response_status](#)

Response status command. [More...](#)

union [si47x_firmware_information](#)
Data representation for Firmware Information (GET_REV) [More...](#)

union [si47x_firmware_query_library](#)
Firmware Query Library ID response. [More...](#)

union [si47x_tune_status](#)
Seek station status. [More...](#)

union [si47x_property](#)
Data type to deal with SET_PROPERTY command. [More...](#)

union [si47x_rqs_status](#)
Radio Signal Quality data representation. [More...](#)

union [si47x_rds_command](#)
Data type for RDS Status command and response information. [More...](#)

union [si47x_rds_status](#)
Response data type for current channel and reads an entry from the RDS FIFO. [More...](#)

union [si47x_rds_int_source](#)
FM_RDS_INT_SOURCE property data type. [More...](#)

union [si47x_rds_config](#)
Data type for FM_RDS_CONFIG Property. [More...](#)

union [si47x_rds_blocka](#)
Block A data type. [More...](#)

union [si47x_rds_blockb](#)
Block B data type. [More...](#)

union [si47x_rds_date_time](#)
union [si47x_agc_status](#)
union [si47x_agc_override](#)
union [si47x_bandwidth_config](#)
union [si47x_ssb_mode](#)
union [si4735_digital_output_format](#)
Digital audio output format data structure (Property 0x0102. DIGITAL_OUTPUT_FORMAT). [More...](#)

struct [si4735_digital_output_sample_rate](#)
Digital audio output sample structure (Property 0x0104. DIGITAL_OUTPUT_SAMPLE_RATE). [More...](#)

class [SI4735](#)
[SI4735](#) Class. [More...](#)

```

struct si473x\_powerup.arg
struct si47x\_frequency.raw
struct si47x\_antenna\_capacitor.raw
struct si47x\_set\_frequency.arg
struct si47x\_seek.arg
struct si47x\_response\_status.resp
struct si47x\_firmware\_information.resp
struct si47x\_firmware\_query\_library.resp
struct si47x\_tune\_status.arg
struct si47x\_property.raw
struct si47x\_rqs\_status.resp
struct si47x\_rds\_command.arg
struct si47x\_rds\_status.resp
struct si47x\_rds\_int\_source.refined
struct si47x\_rds\_config.arg
struct si47x\_rds\_blocka.refined
struct si47x\_rds\_blocka.raw
struct si47x\_rds\_blockb.group0
struct si47x\_rds\_blockb.group2
struct si47x\_rds\_blockb.refined
struct si47x\_rds\_blockb.raw
struct si47x\_rds\_date\_time.refined
struct si47x\_agc\_status.refined
struct si47x\_agc\_override.arg
struct si47x\_bandwidth\_config.param
struct si47x\_ssb\_mode.param
struct si4735\_digital\_output\_format.refined

```

Macros

```

#define POWER\_UP\_FM 0
#define POWER\_UP\_AM 1
#define POWER\_UP\_WB 3
#define POWER\_PATCH 15
#define SI473X\_ADDR\_SEN\_LOW 0x11
#define SI473X\_ADDR\_SEN\_HIGH 0x63
#define POWER\_UP 0x01
#define GET\_REV 0x10
#define POWER\_DOWN 0x11
#define SET\_PROPERTY 0x12
#define GET\_PROPERTY 0x13
#define GET\_INT\_STATUS 0x14
#define FM\_TUNE\_FREQ 0x20
#define FM\_SEEK\_START 0x21
#define FM\_TUNE\_STATUS 0x22
#define FM\_AGC\_STATUS 0x27
#define FM\_AGC\_OVERRIDE 0x28
#define FM\_RSQ\_STATUS 0x23
#define FM\_RDS\_STATUS 0x24
#define FM\_RDS\_INT\_SOURCE 0x1500
#define FM\_RDS\_INT\_FIFO\_COUNT 0x1501
#define FM\_RDS\_CONFIG 0x1502
#define FM\_RDS\_CONFIDENCE 0x1503
#define FM\_BLEND\_STEREO\_THRESHOLD 0x1105
#define FM\_BLEND\_MONO\_THRESHOLD 0x1106
#define FM\_BLEND\_RSSI\_STEREO\_THRESHOLD 0x1800
#define FM\_BLEND\_RSSI\_MONO\_THRESHOLD 0x1801
#define FM\_BLEND\_SNR\_STEREO\_THRESHOLD 0x1804
#define FM\_BLEND\_SNR\_MONO\_THRESHOLD 0x1805

```

```

#define FM_BLEND_MULTIPATH_STEREO_THRESHOLD 0x1808
#define FM_BLEND_MULTIPATH_MONO_THRESHOLD 0x1809
#define AM_TUNE_FREQ 0x40
#define AM_SEEK_START 0x41
#define AM_TUNE_STATUS 0x42
#define AM_RSQ_STATUS 0x43
#define AM_AGC_STATUS 0x47
#define AM_AGC_OVERRIDE 0x48
#define GPIO_CTL 0x80
#define GPIO_SET 0x81
#define SSB_TUNE_FREQ 0x40
#define SSB_TUNE_STATUS 0x42
#define SSB_RSQ_STATUS 0x43
#define SSB_AGC_STATUS 0x47
#define SSB_AGC_OVERRIDE 0x48
#define DIGITAL_OUTPUT_FORMAT 0x0102
#define DIGITAL_OUTPUT_SAMPLE_RATE 0x0104
#define REFCLK_FREQ 0x0201
#define REFCLK_PRESCALE 0x0202
#define AM_DEEMPHASIS 0x3100
#define AM_CHANNEL_FILTER 0x3102
#define AM_AUTOMATIC_VOLUME_CONTROL_MAX_GAIN 0x3103
#define AM_MODE_AFC_SW_PULL_IN_RANGE 0x3104
#define AM_MODE_AFC_SW_LOCK_IN_RANGE 0x3105
#define AM_RSQ_INTERRUPTS 0x3200
#define AM_RSQ_SNR_HIGH_THRESHOLD 0x3201
#define AM_RSQ_SNR_LOW_THRESHOLD 0x3202
#define AM_RSQ_RSSI_HIGH_THRESHOLD 0x3203
#define AM_RSQ_RSSI_LOW_THRESHOLD 0x3204
#define AM_SOFT_MUTE_RATE 0x3300
#define AM_SOFT_MUTE_SLOPE 0x3301
#define AM_SOFT_MUTE_MAX_ATTENUATION 0x3302
#define AM_SOFT_MUTE_SNR_THRESHOLD 0x3303
#define AM_SOFT_MUTE_RELEASE_RATE 0x3304
#define AM_SOFT_MUTE_ATTACK_RATE 0x3305
#define AM_SEEK_BAND_BOTTOM 0x3400
#define AM_SEEK_BAND_TOP 0x3401
#define AM_SEEK_FREQ_SPACING 0x3402
#define AM_SEEK_SNR_THRESHOLD 0x3403
#define AM_SEEK_RSSI_THRESHOLD 0x3404
#define AM_AGC_ATTACK_RATE 0x3702
#define AM_AGC_RELEASE_RATE 0x3703
#define AM_FRONTEND_AGC_CONTROL 0x3705
#define AM_NB_DETECT_THRESHOLD 0x3900
#define AM_NB_INTERVAL 0x3901
#define AM_NB_RATE 0x3902
#define AM_NB_IIR_FILTER 0x3903
#define AM_NB_DELAY 0x3904
#define RX_VOLUME 0x4000
#define RX_HARD_MUTE 0x4001
#define GPO_IEN 0x0001
#define SSB_BFO 0x0100
#define SSB_MODE 0x0101
#define SSB_RSQ_INTERRUPTS 0x3200
#define SSB_RSQ_SNR_HI_THRESHOLD 0x3201
#define SSB_RSQ_SNR_LO_THRESHOLD 0x3202
#define SSB_RSQ_RSSI_HI_THRESHOLD 0x3203
#define SSB_RSQ_RSSI_LO_THRESHOLD 0x3204
#define SSB_SOFT_MUTE_RATE 0x3300
#define SSB_SOFT_MUTE_MAX_ATTENUATION 0x3302

```

```
#define SSB\_SOFT\_MUTE\_SNR\_THRESHOLD 0x3303
#define SSB\_RF\_AGC\_ATTACK\_RATE 0x3700
#define SSB\_RF\_AGC\_RELEASE\_RATE 0x3701
#define SSB\_RF\_IF\_AGC\_ATTACK\_RATE 0x3702
#define SSB\_RF\_IF\_AGC\_RELEASE\_RATE 0x3703
#define LSB\_MODE 1
#define USB\_MODE 2
#define SI473X\_ANALOG\_AUDIO 0b00000101
#define SI473X\_DIGITAL\_AUDIO1 0b00001011
#define SI473X\_DIGITAL\_AUDIO2 0b10110000
#define SI473X\_DIGITAL\_AUDIO3 0b10110101
#define FM\_CURRENT\_MODE 0
#define AM\_CURRENT\_MODE 1
#define SSB\_CURRENT\_MODE 2
#define MAX\_DELAY\_AFTER\_SET\_FREQUENCY 30
#define MIN\_DELAY\_WAIT\_SEND\_LOOP 300
```

Macro Definition Documentation

```
#define AM_AGC_ATTACK_RATE 0x3702

#define AM_AGC_OVERRIDE 0x48

#define AM_AGC_RELEASE_RATE 0x3703

#define AM_AGC_STATUS 0x47

#define AM_AUTOMATIC_VOLUME_CONTROL_MAX_GAIN 0x3103

#define AM_CHANNEL_FILTER 0x3102

#define AM_CURRENT_MODE 1

#define AM_DEEMPHASIS 0x3100

#define AM_FRONTEND_AGC_CONTROL 0x3705

#define AM_MODE_AFC_SW_LOCK_IN_RANGE 0x3105

#define AM_MODE_AFC_SW_PULL_IN_RANGE 0x3104

#define AM_NB_DELAY 0x3904

#define AM_NB_DETECT_THRESHOLD 0x3900

#define AM_NB_IIR_FILTER 0x3903

#define AM_NB_INTERVAL 0x3901

#define AM_NB_RATE 0x3902

#define AM_RSQ_INTERRUPTS 0x3200

#define AM_RSQ_RSSI_HIGH_THRESHOLD 0x3203

#define AM_RSQ_RSSI_LOW_THRESHOLD 0x3204

#define AM_RSQ_SNR_HIGH_THRESHOLD 0x3201

#define AM_RSQ_SNR_LOW_THRESHOLD 0x3202

#define AM_RSQ_STATUS 0x43

#define AM_SEEK_BAND_BOTTOM 0x3400

#define AM_SEEK_BAND_TOP 0x3401

#define AM_SEEK_FREQ_SPACING 0x3402
```



```
#define AM_SEEK_RSSI_THRESHOLD 0x3404

#define AM_SEEK_SNR_THRESHOLD 0x3403

#define AM_SEEK_START 0x41

#define AM_SOFT_MUTE_ATTACK_RATE 0x3305

#define AM_SOFT_MUTE_MAX_ATTENUATION 0x3302

#define AM_SOFT_MUTE_RATE 0x3300

#define AM_SOFT_MUTE_RELEASE_RATE 0x3304

#define AM_SOFT_MUTE_SLOPE 0x3301

#define AM_SOFT_MUTE_SNR_THRESHOLD 0x3303

#define AM_TUNE_FREQ 0x40

#define AM_TUNE_STATUS 0x42

#define DIGITAL_OUTPUT_FORMAT 0x0102

#define DIGITAL_OUTPUT_SAMPLE_RATE 0x0104

#define FM_AGC_OVERRIDE 0x28

#define FM_AGC_STATUS 0x27

#define FM_BLEND_MONO_THRESHOLD 0x1106

#define FM_BLEND_MULTIPATH_MONO_THRESHOLD 0x1809

#define FM_BLEND_MULTIPATH_STEREO_THRESHOLD 0x1808

#define FM_BLEND_RSSI_MONO_THRESHOLD 0x1801

#define FM_BLEND_RSSI_STEREO_THRESHOLD 0x1800

#define FM_BLEND_SNR_MONO_THRESHOLD 0x1805

#define FM_BLEND_SNR_STEREO_THRESHOLD 0x1804

#define FM_BLEND_STEREO_THRESHOLD 0x1105

#define FM_CURRENT_MODE 0

#define FM_RDS_CONFIDENCE 0x1503

#define FM_RDS_CONFIG 0x1502
```

```

#define FM_RDS_INT_FIFO_COUNT 0x1501

#define FM_RDS_INT_SOURCE 0x1500

#define FM_RDS_STATUS 0x24

#define FM_RSQ_STATUS 0x23

#define FM_SEEK_START 0x21

#define FM_TUNE_FREQ 0x20

#define FM_TUNE_STATUS 0x22

#define GET_INT_STATUS 0x14

#define GET_PROPERTY 0x13

#define GET_REV 0x10

#define GPIO_CTL 0x80

#define GPIO_SET 0x81

#define GPO_IEN 0x0001

#define LSB_MODE 1

#define MAX_DELAY_AFTER_SET_FREQUENCY 30

#define MIN_DELAY_WAIT_SEND_LOOP 300

#define POWER_DOWN 0x11

#define POWER_PATCH 15

#define POWER_UP 0x01

#define POWER_UP_AM 1

#define POWER_UP_FM 0

```

[SI4735](#) ARDUINO LIBRARY

Const, Data type and Methods definitions References: Si47XX PROGRAMMING GUIDE AN332 AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE

See also

documentation on <https://github.com/pu2clr/SI4735>

Author

PU2CLR - Ricardo Lima Caratti

By Ricardo Lima Caratti, Nov 2019

```

#define POWER_UP_WB 3

#define REFCLK_FREQ 0x0201

#define REFCLK_PRESCALE 0x0202

#define RX_HARD_MUTE 0x4001

#define RX_VOLUME 0x4000

#define SET_PROPERTY 0x12

#define SI473X_ADDR_SEN_HIGH 0x63

#define SI473X_ADDR_SEN_LOW 0x11

#define SI473X_ANALOG_AUDIO 0b00000101

#define SI473X_DIGITAL_AUDIO1 0b00001011

#define SI473X_DIGITAL_AUDIO2 0b10110000

#define SI473X_DIGITAL_AUDIO3 0b10110101

#define SSB_AGC_OVERRIDE 0x48

#define SSB_AGC_STATUS 0x47

#define SSB_BFO 0x0100

#define SSB_CURRENT_MODE 2

#define SSB_MODE 0x0101

#define SSB_RF_AGC_ATTACK_RATE 0x3700

#define SSB_RF_AGC_RELEASE_RATE 0x3701

#define SSB_RF_IF_AGC_ATTACK_RATE 0x3702

#define SSB_RF_IF_AGC_RELEASE_RATE 0x3703

#define SSB_RSQ_INTERRUPTS 0x3200

#define SSB_RSQ_RSSI_HI_THRESHOLD 0x3203

#define SSB_RSQ_RSSI_LO_THRESHOLD 0x3204

#define SSB_RSQ_SNR_HI_THRESHOLD 0x3201

#define SSB_RSQ_SNR_LO_THRESHOLD 0x3202

```

```
#define SSB_RSQ_STATUS 0x43

#define SSB_SOFT_MUTE_MAX_ATTENUATION 0x3302

#define SSB_SOFT_MUTE_RATE 0x3300

#define SSB_SOFT_MUTE_SNR_THRESHOLD 0x3303

#define SSB_TUNE_FREQ 0x40

#define SSB_TUNE_STATUS 0x42

#define USB_MODE 2
```

Index

INDE