Si4735 Arduino Library

AUTHOR
Version 1.1.8
03/04/2020

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:

File Index

File List

Here is a list of all files with brief descriptions:

Module Documentation

Audio setup

Functions

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

Configures the digital audio output format.

void SI4735::digitalOutputSampleRate (uint16_t DOSR)

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

void SI4735::setVolume (uint8_t volume)

RESP8 - Returns the Chip Revision (ASCII).

void SI4735::setAudioMute (bool off)

Returns the current volume level.

uint8_t SI4735::getVolume ()

Gets the current volume level.

void SI4735::volumeUp ()

Set sound volume level Up

void SI4735::volumeDown ()

Set sound volume level Down

Detailed Description

Function Documentation

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

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).
uint8_t	OMONO Digital Output Mono Mode (0=Use mono/stereo blend).
uint8_t	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).
uint8_t	OFALL Digital Output DCLK Edge (0 = use DCLK rising edge, 1 = use DCLK falling edge)

```
00898 {
00899     si4735_digital_output_format df;
00900     df.refined.OSIZE = OSIZE;
00901     df.refined.OMONO = OMONO;
00902     df.refined.OMODE = OMODE;
00903     df.refined.OFALL = OFALL;
00904     sendProperty(DIGITAL_OUTPUT_FORMAT, df.raw);
00905 }
```

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

uint16_t	DOSR Digital Output Sample Rate(32–48 ksps .0 to disable digital audio output).

```
00917 {
00918     sendProperty(DIGITAL_OUTPUT_SAMPLE_RATE, DOSR);
00919 }
```

uint8_t SI4735::getVolume ()

Gets the current volume level.

See also

setVolume()

Returns

volume (domain: 0 - 63)

```
00961 {
00962     return this->volume;
00963 }
```

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.

```
00946 {
00947     uint16_t value = (off) ? 3 : 0; // 3 means mute; 0 means unmute
```

```
00948     sendProperty(RX_HARD_MUTE, value);
00949 }
```

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

uint8_t	volume (domain: 0 - 63)

```
00931 {
00932     sendProperty(RX_VOLUME, volume);
00933     this->volume = volume;
00934 }
```

void SI4735::volumeDown ()

Set sound volume level Down

See also

setVolume()

```
00987 {
00988     if (volume > 0)
00989         volume--;
00990     setVolume(volume);
00991 }
```

void SI4735::volumeUp ()

Set sound volume level Up

See also

setVolume()

```
00973 {
00974     if (volume < 63)
00975         volume++;
00976     setVolume(volume);
00977 }
```

Deal with Interrupt

Detailed Description

Deal with Interrupt

Deal with Interrupt and I2C bus

Data Structures

class SI4735

SI4735 Class. *More...*

Functions

SI4735::SI4735 ()

Clear RDS group type 0A buffer.

void SI4735::waitInterrupr (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 ()
Clear RDS group type 0A buffer.

void reset (void)
Reset the SI473X

void waitToSend (void)
Reset the Si47XX device.

void setup (uint8_t resetPin, uint8_t defaultFunction)
Wait for the Si47XX device is ready to receive a command.

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)
Configure the Si47XX to power it up.

void analogPowerUp (void)
Power the receiver up. Call setPowerUp before call this method.

void powerDown (void)
Deprecated. Use radioPowerUp.

void setFrequency (uint16_t)
Set the frequency to the corrent function of the Si4735 (FM, AM or SSB)

void getStatus ()

Tune the receiver.

void getStatus (uint8_t, uint8_t)
Gets the current status of the Si47XX device.

uint16_t getFrequency (void)
Gets the current frequency of the Si4735 (AM or FM)

uint16_t getCurrentFrequency ()
Gets the current frequency.

bool getSignalQualityInterrupt ()
Gets the current frequency stored in memory (it does not query the Si47XX device)

bool getRadioDataSystemInterrupt ()
Get the Radio Data System (RDS) Interrupt status.

bool getTuneCompleteTriggered ()
Get the Tune Complete status.

bool getStatusError ()
Get the Status Error.

bool getStatusCTS ()
Gets the Error flag Clear to Send.

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

bool getBandLimit ()
Returns true if a seek hit the band limit.

bool getStatusValid ()
Gets the channel status.

uint8_t getReceivedSignalStrengthIndicator ()
Returns the value of Received Signal Strength Indicator (dBÎ¼V).

uint8_t getStatusSNR ()
Gets the SNR metric when tune is complete (dB)

uint8_t getStatusMULT ()
Get the Status the M U L T.

uint8_t getAntennaTuningCapacitor ()
Get the Antenna Tuning Capacitor value.

void getAutomaticGainControl ()
Queries Automatic Gain Control STATUS.

void setAvcAmMaxGain ()
Queries Automatic Gain Control STATUS.

void setAvcAmMaxGain (uint8_t gain)
Sets the maximum gain for automatic volume control.

uint8_t getCurrentAvcAmMaxGain ()
Sets the maximum gain for automatic volume control.

void setAmSoftMuteMaxAttenuation (uint8_t smattn=0)
Sets the Am Soft Mute Max Attenuation.

void setSsbSoftMuteMaxAttenuation (uint8_t smattn=0)
Sets the SSB Soft Mute Max Attenuation object.

bool isAgcEnabled ()
Checks if the AGC is enabled.

uint8_t getAgcGainIndex ()
Gets the current AGC gain index.

void setAutomaticGainControl (uint8_t AGCDIS, uint8_t AGCIDX)
Automatic Gain Control setup.

void getCurrentReceivedSignalQuality (uint8_t INTACK)
Overrides the AGC setting.

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

uint8_t getCurrentRSSI ()
uint8_t getCurrentSNR ()
current receive signal strength (0â€“127 dBÎ¼V).

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 ()
Gets the current volume level.

void volumeDown ()
Set sound volume level Down

void volumeUp ()
Set sound volume level Up

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)
Configures the digital audio output format.

void digitalOutputSampleRate (uint16_t DOSR)
Enables digital audio output and configures digital audio output sample rate in samples per second (sps).

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 intialFreq, 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)
Selects the bandwidth of the channel filter for AM reception.

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

void getFirmware (void)
Gets firmware information.

void setFunction (uint8_t FUNC)
void seekStation (uint8_t SEEKUP, uint8_t WRAP)
Look for a station (Automatic tune)

void seekStationUp ()
Search for the next station.

void seekStationDown ()
Search the previous station.

void 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.

void setSeekAmSpacing (uint16_t spacing)

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

void setSeekSrnThreshold (uint16_t value)

Sets the SNR threshold for a valid AM Seek/Tune.

void setSeekRssiThreshold (uint16_t value)

Sets the RSSI threshold for a valid AM Seek/Tune.

void setFmBlendStereoThreshold (uint8_t parameter)

Sets RSSI threshold for stereo blend (Full stereo above threshold, blend below threshold).

void setFmBlendMonoThreshold (uint8_t parameter)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold).

void setFmBlendRssiStereoThreshold (uint8_t parameter)

Sets RSSI threshold for stereo blend. (Full stereo above threshold, blend below threshold.)

void setFmBLendRssiMonoThreshold (uint8_t parameter)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold).

void setFmBlendSnrStereoThreshold (uint8_t parameter)

Sets SNR threshold for stereo blend (Full stereo above threshold, blend below threshold).

void setFmBLendSnrMonoThreshold (uint8_t parameter)

Sets SNR threshold for mono blend (Full mono below threshold, blend above threshold).

void setFmBlendMultiPathStereoThreshold (uint8_t parameter)

Sets multipath threshold for stereo blend (Full stereo below threshold, blend above threshold).

void setFmBlendMultiPathMonoThreshold (uint8_t parameter)

Sets Multipath threshold for mono blend (Full mono above threshold, blend below threshold).

void setFmStereoOn ()

Turn Off Stereo operation.

void setFmStereoOff ()

Turn Off Stereo operation.

void RdsInit ()

Starts the control member variables for RDS.

void setRdsIntSource (uint8_t RDSNEWBLOCKB, uint8_t RDSNEWBLOCKA, uint8_t RDSSYNCFOUND, uint8_t RDSSYNCLOST, uint8_t RDSRECV)

Configures interrupt related to RDS.

void 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.

void getRdsStatus ()
Gets RDS Status.

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 RDSEN, 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)
Returns the programa type.

uint8_t getRdsGroupType (void)
Returns the Group Type (extracted from the Block B)

uint8_t getRdsFlagAB (void)
Returns the current Text Flag A/B

uint8_t getRdsVersionCode (void)
Gets the version code (extracted from the Block B)

uint8_t getRdsProgramType (void)
Returns the Program Type (extracted from the Block B)

uint8_t getRdsTextSegmentAddress (void)
Returns the address of the text segment.

char * getRdsText (void)
Gets the RDS Text when the message is of the Group Type 2 version A.

char * getRdsText0A (void)
Gets the station name and other messages.

char * getRdsText2A (void)
Gets the Text processed for the 2A group.

char * getRdsText2B (void)
Gets the Text processed for the 2B group.

char * getRdsTime (void)
Gets the RDS time and date when the Group type is 4.

void getNext2Block (char *)
Process data received from group 2B.

void getNext4Block (char *)
Process data received from group 2A.

void ssbSetup ()
Starts the Si473X device on SSB (same AM Mode).

void setSSBBfo (int offset)
Sets the SSB Beat Frequency Offset (BFO).

void setSSBConfig (uint8_t AUDIOBW, uint8_t SBCUTFLT, uint8_t AVC_DIVIDER, uint8_t AVCEN, uint8_t SMUTESEL, uint8_t DSP_AFCDIS)
Sets the SSB receiver mode.

void setSSB (uint16_t fromFreq, uint16_t toFreq, uint16_t intialFreq, uint16_t step, uint8_t usblsb)
void setSSB (uint8_t usblsb)
Set the radio to AM function.

void setSSBAudioBandwidth (uint8_t AUDIOBW)
SSB Audio Bandwidth for SSB mode.

void setSSBAutomaticVolumeControl (uint8_t AVCEN)
Sets SSB Automatic Volume Control (AVC) for SSB mode.

void setSBBSidebandCutoffFilter (uint8_t SBCUTFLT)
Sets SBB Sideband Cutoff Filter for band pass and low pass filters.

void setSSBAvcDivider (uint8_t AVC_DIVIDER)
Sets AVC Divider.

void setSSBDspAfc (uint8_t DSP_AFCDIS)
Sets DSP AFC disable or enable.

void setSSBSoftMute (uint8_t SMUTESEL)
Sets SSB Soft-mute Based on RSSI or SNR Selection:

si47x_firmware_query_library queryLibraryId ()
Query the library information of the Si47XX device.

void patchPowerUp ()
This method can be used to prepare the device to apply SSBRX patch.

bool 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.

bool downloadPatch (int eeprom_i2c_address)
Transfers the content of a patch stored in a eeprom to the SI4735 device.

void ssbPowerUp ()
This function can be useful for debug and test.

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)
wait for interrupt (useful if you are using interrupt resource)

void sendSSBModeProperty ()
Sends the property command to the device.

void disableFmDebug ()
Sends SSB_MODE property to the device.

void clearRdsBuffer2A ()
disable some Si47XX debug resources implemented by the Silicon Labs

void clearRdsBuffer2B ()
Clear RDS group type 2A buffer.

void clearRdsBuffer0A ()
Clear RDS group type 2B buffer.

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 rdsTextAdress0A
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

bool SI4735::getACFIndicator () [inline]

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

Returns

true

```
01004                                   {
01005           return currentStatus.resp.AFCRL;
01006       };
```

uint8_t SI4735::getAgcGainIndex () [inline]

Gets the current AGC gain index.

Returns

uint8_t The current AGC gain index.

```
01135                                       {
01136           return currentAgcStatus.refined.AGCIDX;
01137       };
```

uint8_t SI4735::getAntennaTuningCapacitor () [inline]

Get the Antenna Tuning Capacitor value.

Returns the current antenna tuning capacitor value.

Returns

uint8_t capacitance

```
01069                                                   {
01070           return currentStatus.resp.READANTCAP;
01071       };
```

bool SI4735::getBandLimit () [inline]

Returns true if a seek hit the band limit.

(WRAP = 0 in FM_START_SEEK) or wrapped to the original frequency(WRAP = 1).

Returns

BLTF

```
01015                                   {
01016           return currentStatus.resp.BLTF;
01017       };
```

bool SI4735::getCurrentAfcRailIndicator () [inline]

Valid Channel.

```
01152 { return currentRqsStatus.resp.AFCRL; };
```

uint8_t SI4735::getCurrentAvcAmMaxGain () [inline]

Sets the maximum gain for automatic volume control.

Get the current Avc Am Max Gain

Returns

uint8_t Current AVC gain index value

```
01090                                                {
01091         return currentAvcAmMaxGain;
01092     };
```

bool SI4735::getCurrentBlendDetectInterrupt () [inline]

Multipath Detect High.

```
01161 { return currentRqsStatus.resp.BLENDINT; };
```

uint8_t SI4735::getCurrentMultipath () [inline]

Indicates stereo pilot presence.

```
01157 { return currentRqsStatus.resp.MULT; };
```

bool SI4735::getCurrentMultipathDetectHigh () [inline]

Multipath Detect Low.

```
01160 { return currentRqsStatus.resp.MULTHINT; };
```

bool SI4735::getCurrentMultipathDetectLow () [inline]

Signed frequency offset (kHz).

```
01159 { return currentRqsStatus.resp.MULTLINT; };
```

bool SI4735::getCurrentPilot () [inline]

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

```
01156 { return currentRqsStatus.resp.PILOT; };
```

uint8_t SI4735::getCurrentRSSI () [inline]

```
01145 { return currentRqsStatus.resp.RSSI; };
```

bool SI4735::getCurrentRssiDetectHigh () [inline]

RSSI Detect Low.

```
01148 { return currentRqsStatus.resp.RSSIHINT; };
```

bool SI4735::getCurrentRssiDetectLow () [inline]

current SNR metric (0–127 dB).

```
01147 { return currentRqsStatus.resp.RSSIILINT; };
```

uint8_t SI4735::getCurrentSignedFrequencyOffset () [inline]

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

```
01158 { return currentRqsStatus.resp.FREQOFF; };
```

uint8_t SI4735::getCurrentSNR () [inline]

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

```
01146 { return currentRqsStatus.resp.SNR; };
```

bool SI4735::getCurrentSnrDetectHigh () [inline]

SNR Detect Low.

```
01150 { return currentRqsStatus.resp.SNRHINT; };
```

bool SI4735::getCurrentSnrDetectLow () [inline]

RSSI Detect High.

```
01149 { return currentRqsStatus.resp.SNRLINT; };
```

bool SI4735::getCurrentSoftMuteIndicator () [inline]

AFC Rail Indicator.

```
01153 { return currentRqsStatus.resp.SMUTE; };
```

uint8_t SI4735::getCurrentStereoBlend () [inline]

Soft Mute Indicator. Indicates soft mute is engaged.

```
01155 { return currentRqsStatus.resp.STBLEND; };
```

bool SI4735::getCurrentValidChannel () [inline]

SNR Detect High.

```
01151 { return currentRqsStatus.resp.VALID; };
```

uint8_t SI4735::getCurrentVolume () [inline]

```
01183 { return volume; };
```

uint8_t SI4735::getFirmwareCHIPREV () [inline]

RESP7 - Returns the Component Minor Revision (ASCII).

```
01176 { return firmwareInfo.resp.CHIPREV; };
```

uint8_t SI4735::getFirmwareCMPMAJOR () [inline]

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

```
01174 { return firmwareInfo.resp.CMPMAJOR; };
```

uint8_t SI4735::getFirmwareCMPMINOR () [inline]

RESP6 - Returns the Component Major Revision (ASCII).

```
01175 { return firmwareInfo.resp.CMPMINOR; };
```

uint8_t SI4735::getFirmwareFWMAJOR () [inline]

RESP1 - Part Number (HEX)

```
01170 { return firmwareInfo.resp.FWMAJOR; };
```

uint8_t SI4735::getFirmwareFWMINOR () [inline]

RESP2 - Returns the Firmware Major Revision (ASCII).

```
01171 { return firmwareInfo.resp.FWMINOR; };
```

uint8_t SI4735::getFirmwarePATCHH () [inline]

RESP3 - Returns the Firmware Minor Revision (ASCII).

```
01172 { return firmwareInfo.resp.PATCHH; };
```

uint8_t SI4735::getFirmwarePATCHL () [inline]

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

```
01173 { return firmwareInfo.resp.PATCHL; };
```

uint8_t SI4735::getFirmwarePN () [inline]

Blend Detect Interrupt.

```
01169 { return firmwareInfo.resp.PN;};
```

bool SI4735::getGroupLost () [inline]

1 = RDS currently synchronized.

```
01242 { return currentRdsStatus.resp.GRPLOST; };
```

uint8_t SI4735::getNumRdsFifoUsed () [inline]

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

```
01243 { return currentRdsStatus.resp.RDSFIFOUSED; };
```

bool SI4735::getRadioDataSystemInterrupt () [inline]

Get the Radio Data System (RDS) Interrupt status.

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 63

Returns

RDSINT status

```
00970                                                {
00971         return currentStatus.resp.RDSINT;
00972     };
```

bool SI4735::getRdsNewBlockA () [inline]

1 = Found RDS synchronization

```
01239 { return currentRdsStatus.resp.RDSNEWBLOCKA; };
```

Referenced by getRdsPI().

bool SI4735::getRdsNewBlockB () [inline]

1 = Valid Block A data has been received.

```
01240 { return currentRdsStatus.resp.RDSNEWBLOCKB; };
```

bool SI4735::getRdsReceived () [inline]

```
01236 { return currentRdsStatus.resp.RDSRECV; };
```

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

bool SI4735::getRdsSync () [inline]

1 = Valid Block B data has been received.

```
01241 { return currentRdsStatus.resp.RDSSYNC; };
```

bool SI4735::getRdsSyncFound () [inline]

1 = Lost RDS synchronization

```
01238 { return currentRdsStatus.resp.RDSSYNCFOUND; };
```

bool SI4735::getRdsSyncLost () [inline]

1 = FIFO filled to minimum number of groups

```
01237 { return currentRdsStatus.resp.RDSSYNCLOST; };
```

uint8_t SI4735::getReceivedSignalStrengthIndicator () [inline]

Returns the value of Received Signal Strength Indicator (dBÎ¼V).

Returns

uint8_t

```
01036                                                    {
01037         return currentStatus.resp.RSSI;
01038     };
```

bool SI4735::getSignalQualityInterrupt () [inline]

Gets the current frequency stored in memory (it does not query the Si47XX device)

STATUS RESPONSE Set of methods to get current status information. Call them after getStatus or getFrequency or seekStation

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 63

Get the Signal Quality Interrupt status

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 63

Returns

RDSINT status

```
00961                                          {
00962         return currentStatus.resp.RSQINT;
00963     };
```

bool SI4735::getStatusCTS () [inline]

Gets the Error flag Clear to Send.

Returns

CTS

```
00997 { return currentStatus.resp.CTS; };
```

bool SI4735::getStatusError () [inline]

Get the Status Error.

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

Returns

Error flag

```
00988                                   {
00989         return currentStatus.resp.ERR;
00990     };
```

uint8_t SI4735::getStatusMULT () [inline]

Get the Status the M U L T.

Returns the value containing the multipath metric when tune is complete.

Returns

uint8_t

```
01058                                    {
01059         return currentStatus.resp.MULT;
01060     };
```

uint8_t SI4735::getStatusSNR () [inline]

Gets the SNR metric when tune is complete (dB)

Returns the value of the SNR metric when tune is complete (dB).

Returns

uint8_t

```
01047                                   {
01048         return currentStatus.resp.SNR;
01049     };
```

bool SI4735::getStatusValid () [inline]

Gets the channel status.

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

Returns

true

false

```
01027                                  {
01028         return currentStatus.resp.VALID;
01029     };
```

bool SI4735::getTuneCompleteTriggered () [inline]

Get the Tune Complete status.

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

Returns

STCINT status

```
00979                                          {
00980         return currentStatus.resp.STCINT;
00981     };
```

uint8_t SI4735::getTuneFrequencyFast () [inline]

```
01200 { 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.

```
01202 { return currentFrequencyParams.arg.FREEZE; };
```

bool SI4735::isAgcEnabled () [inline]

Checks if the AGC is enabled.

Returns

true if the AGC is enabled

```
01126                                {
01127         return !currentAgcStatus.refined.AGCDIS;
01128     };
```

void SI4735::setAmSoftMuteMaxAttenuation (uint8_t *smattn* = 0) [inline]

Sets the Am Soft Mute Max Attenuation.

This function can be useful to disable Soft Mute. The value 0 disable soft mute.

Specified in units of dB. Default maximum attenuation is 8 dB.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 158.

Parameters

smattn	Maximum attenuation to apply when in soft mute

```
01104                                                                {
01105         sendProperty(AM_SOFT_MUTE_MAX_ATTENUATION, smattn);
01106     };
```

void SI4735::setAvcAmMaxGain () [inline]

Queries Automatic Gain Control STATUS.

Sets the Avc Am Max Gain to 48dB

```
01079                                  {
01080         sendProperty(AM_AUTOMATIC_VOLUME_CONTROL_MAX_GAIN,
((currentAvcAmMaxGain = 48) * 340));
01081     };
```

void SI4735::setFunction (uint8_t *FUNC*)

void SI4735::setI2CFastMode (void) [inline]

Sets I2C buss to 100KHz.

```
01296 { 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

value	in Hz. For example: The values 500000 sets the bus to 500KHz.

```
01305 { Wire.setClock(value); };
```

void SI4735::setI2CLowSpeedMode (void) [inline]

```
01294 { Wire.setClock(10000); };
```

void SI4735::setI2CStandardMode (void) [inline]

Sets I2C buss to 10KHz.

```
01295 { Wire.setClock(100000); };
```

void SI4735::setSsbSoftMuteMaxAttenuation (uint8_t *smattn* = 0)[inline]

Sets the SSB Soft Mute Max Attenuation object.

Sets maximum attenuation during soft mute (dB). Set to 0 to disable soft mute.

Specified in units of dB. Default maximum attenuation is 8 dB.

Parameters

smattn	Maximum attenuation to apply when in soft mute.

```
01117                                                                 {
01118         sendProperty(SSB_SOFT_MUTE_MAX_ATTENUATION, smattn);
01119     };
```

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

Returns the FAST tuning status.

```
01201 { currentFrequencyParams.arg.FAST = FAST; };
```

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

Returns the FREEZE status.

```
01203 { currentFrequencyParams.arg.FREEZE = FREEZE; };
```

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::rdsTextAdress0A [protected]

rds_buffer2B current position

Referenced by getRdsText0A().

int SI4735::rdsTextAdress2A [protected]

RDS date time received information

Referenced by getRdsText(), and getRdsText2A().

int SI4735::rdsTextAdress2B [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

uint8_t	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

senPin	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

uint8_t	i2cAddr (example 0x10)

```
00137                                                       {
00138     deviceAddress = i2cAddr;
00139 };
```

SI4735::SI4735 ()

Clear RDS group type 0A buffer.

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 }
```

FM Mono Stereo audio setup

Functions

void SI4735::setFmBlendStereoThreshold (uint8_t parameter)

Sets RSSI threshold for stereo blend (Full stereo above threshold, blend below threshold).

void SI4735::setFmBlendMonoThreshold (uint8_t parameter)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold).

void SI4735::setFmBlendRssiStereoThreshold (uint8_t parameter)

Sets RSSI threshold for stereo blend. (Full stereo above threshold, blend below threshold.)

void SI4735::setFmBLendRssiMonoThreshold (uint8_t parameter)

Sets RSSI threshold for mono blend (Full mono below threshold, blend above threshold).

void SI4735::setFmBlendSnrStereoThreshold (uint8_t parameter)

Sets SNR threshold for stereo blend (Full stereo above threshold, blend below threshold).

void SI4735::setFmBLendSnrMonoThreshold (uint8_t parameter)

Sets SNR threshold for mono blend (Full mono below threshold, blend above threshold).

void SI4735::setFmBlendMultiPathStereoThreshold (uint8_t parameter)

Sets multipath threshold for stereo blend (Full stereo below threshold, blend above threshold).

void SI4735::setFmBlendMultiPathMonoThreshold (uint8_t parameter)

Sets Multipath threshold for mono blend (Full mono above threshold, blend below threshold).

void SI4735::setFmStereoOff ()

Turn Off Stereo operation.

void SI4735::setFmStereoOn ()

Turn Off Stereo operation.

void SI4735::disableFmDebug ()

Sends SSB_MODE property to the device.

Detailed Description

Function Documentation

void SI4735::disableFmDebug () [protected]

Sends SSB_MODE property to the device.

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.

```
00869 {
00870     Wire.beginTransmission(deviceAddress);
00871     Wire.write(0x12);
00872     Wire.write(0x00);
00873     Wire.write(0xFF);
00874     Wire.write(0x00);
00875     Wire.write(0x00);
00876     Wire.write(0x00);
00877     Wire.endTransmission();
00878     delayMicroseconds(2500);
00879 }
```

Referenced by SI4735::setFM().

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 dBÎ¼V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 56.

Parameters

parameter	valid values: 0 to 127

```
00738 {
00739     sendProperty(FM_BLEND_MONO_THRESHOLD, parameter);
00740 }
```

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

parameter	valid values: 0 to 100

```
00834 {
00835     sendProperty(FM_BLEND_MULTIPATH_MONO_THRESHOLD, parameter);
00836 }
```

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

parameter	valid values: 0 to 100

```
00818 {
00819     sendProperty(FM_BLEND_MULTIPATH_STEREO_THRESHOLD, parameter);
00820 }
```

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 dBÎ¼V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

parameter	valid values: 0 to 127

```
00770 {
00771     sendProperty(FM_BLEND_RSSI_MONO_THRESHOLD, parameter);
00772 }
```

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 dBÎ¼V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 59.

Parameters

parameter	valid values: 0 to 127

```
00754 {
00755     sendProperty(FM_BLEND_RSSI_STEREO_THRESHOLD, parameter);
00756 }
```

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

parameter	valid values: 0 to 127

```
00802 {
00803     sendProperty(FM_BLEND_SNR_MONO_THRESHOLD, parameter);
00804 }
```

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

parameter	valid values: 0 to 127

```
00786 {
00787     sendProperty(FM_BLEND_SNR_STEREO_THRESHOLD, parameter);
00788 }
```

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

parameter	valid values: 0 to 127

```
00722 {
00723     sendProperty(FM_BLEND_STEREO_THRESHOLD, parameter);
00724 }
```

void SI4735::setFmStereoOff ()

Turn Off Stereo operation.

TO DO

```
00844 {
00846 }
```

void SI4735::setFmStereoOn ()

Turn Off Stereo operation.

TO DO

```
00854 {
00856 }
```

FM RDS/DBDS

Functions

void SI4735::RdsInit ()
Starts the control member variables for RDS.

void SI4735::clearRdsBuffer2A ()
disable some Si47XX debug resources implemented by the Silicon Labs

void SI4735::clearRdsBuffer2B ()
Clear RDS group type 2A buffer.

void SI4735::clearRdsBuffer0A ()
Clear RDS group type 2B buffer.

void SI4735::setRdsConfig (uint8_t RDSEN, 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).

void SI4735::setRdsIntSource (uint8_t RDSNEWBLOCKB, uint8_t RDSNEWBLOCKA, uint8_t RDSSYNCFOUND, uint8_t RDSSYNCLOST, uint8_t RDSRECV)
Configures interrupt related to RDS.

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.

void SI4735::getRdsStatus ()
Gets RDS Status.

uint16_t SI4735::getRdsPI (void)
Returns the programa type.

uint8_t SI4735::getRdsGroupType (void)
Returns the Group Type (extracted from the Block B)

uint8_t SI4735::getRdsFlagAB (void)
Returns the current Text Flag A/B

uint8_t SI4735::getRdsTextSegmentAddress (void)
Returns the address of the text segment.

uint8_t SI4735::getRdsVersionCode (void)
Gets the version code (extracted from the Block B)

uint8_t SI4735::getRdsProgramType (void)
Returns the Program Type (extracted from the Block B)

void SI4735::getNext2Block (char *)
Process data received from group 2B.

void SI4735::getNext4Block (char *)
Process data received from group 2A.

char * SI4735::getRdsText (void)
Gets the RDS Text when the message is of the Group Type 2 version A.

char * SI4735::getRdsText0A (void)
Gets the station name and other messages.

char * SI4735::getRdsText2A (void)
Gets the Text processed for the 2A group.

char * SI4735::getRdsText2B (void)
Gets the Text processed for the 2B group.

char * SI4735::getRdsTime (void)
Gets the RDS time and date when the Group type is 4.

Detailed Description

Function Documentation

void SI4735::clearRdsBuffer0A () [protected]

Clear RDS group type 2B buffer.

Clear RDS buffer 0A (text)

```
01426 {
01427     for (int i = 0; i < 9; i++)
01428         rds_buffer0A[i] = ' '; // Station Name buffer
01429 }
```

References SI4735::rds_buffer0A.

Referenced by SI4735::getRdsStatus(), and SI4735::RdsInit().

void SI4735::clearRdsBuffer2A () [protected]

disable some Si47XX debug resources implemented by the Silicon Labs

Clear RDS buffer 2A (text)

```
01403 {
01404     for (int i = 0; i < 65; i++)
01405         rds_buffer2A[i] = ' '; // Radio Text buffer - Program Information
01406 }
```

References SI4735::rds_buffer2A.

Referenced by SI4735::getRdsStatus(), and SI4735::RdsInit().

void SI4735::clearRdsBuffer2B () [protected]

Clear RDS group type 2A buffer.

Clear RDS buffer 2B (text)

```
01415 {
01416     for (int i = 0; i < 33; i++)
01417         rds_buffer2B[i] = ' '; // Radio Text buffer - Station Informaation
01418 }
```

References SI4735::rds_buffer2B.

Referenced by SI4735::getRdsStatus(), and SI4735::RdsInit().

void SI4735::getNext2Block (char * c)

Process data received from group 2B.

Parameters

c	char array reference to the "group 2B" text

```
01724 {
01725     char raw[2];
01726     int i, j;
01727
01728     raw[1] = currentRdsStatus.resp.BLOCKDL;
01729     raw[0] = currentRdsStatus.resp.BLOCKDH;
01730
01731     for (i = j = 0; i < 2; i++)
01732     {
01733         if (raw[i] == 0xD || raw[i] == 0xA)
01734         {
01735             c[j] = '\0';
01736             return;
01737         }
01738         if (raw[i] >= 32)
01739         {
01740             c[j] = raw[i];
01741             j++;
01742         }
01743         else
01744         {
01745             c[i] = ' ';
01746         }
01747     }
01748 }
```

Referenced by SI4735::getRdsText0A(), and SI4735::getRdsText2B().

void SI4735::getNext4Block (char * c)

Process data received from group 2A.

Parameters

c	char array reference to the "group 2A" text

```
01758 {
01759     char raw[4];
01760     int i, j;
```

```
01761
01762     raw[0] = currentRdsStatus.resp.BLOCKCH;
01763     raw[1] = currentRdsStatus.resp.BLOCKCL;
01764     raw[2] = currentRdsStatus.resp.BLOCKDH;
01765     raw[3] = currentRdsStatus.resp.BLOCKDL;
01766     for (i = j = 0; i < 4; i++)
01767     {
01768         if (raw[i] == 0xD || raw[i] == 0xA)
01769         {
01770             c[j] = '\0';
01771             return;
01772         }
01773         if (raw[i] >= 32)
01774         {
01775             c[j] = raw[i];
01776             j++;
01777         }
01778         else
01779         {
01780             c[i] = ' ';
01781         }
01782     }
01783 }
```

Referenced by SI4735::getRdsText(), and SI4735::getRdsText2A().

uint8_t SI4735::getRdsFlagAB (void)

Returns the current Text Flag A/B

Returns

uint8_t current Text Flag A/B

```
01650 {
01651     si47x_rds_blockb blkb;
01652
01653     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01654     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01655
01656     return blkb.refined.textABFlag;
01657 }
```

uint8_t SI4735::getRdsGroupType (void)

Returns the Group Type (extracted from the Block B)

Returns

BLOCKBL

```
01633 {
01634     si47x_rds_blockb blkb;
01635
01636     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01637     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01638
01639     return blkb.refined.groupType;
01640 }
```

uint16_t SI4735::getRdsPI (void)

Returns the programa type.

Read the Block A content

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 77 and 78

Returns

BLOCKAL

```
01617 {
01618     if (getRdsReceived() && getRdsNewBlockA())
01619     {
01620         return currentRdsStatus.resp.BLOCKAL;
01621     }
01622     return 0;
01623 }
```

References SI4735::getRdsNewBlockA(), and SI4735::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 betwenn 0 and 31)

```
01707 {
01708     si47x_rds_blockb blkb;
01709
01710     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01711     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01712
01713     return blkb.refined.programType;
01714 }
```

void SI4735::getRdsStatus ()

Gets RDS Status.

Same result of calling getRdsStatus(0,0,0).

Please, call getRdsStatus(uint8_t INTACK, uint8_t MTFIFO, uint8_t STATUSONLY) instead getRdsStatus() if you want other behaviour.

See also

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

```
01599 {
01600     getRdsStatus(0, 0, 0);
01601 }
```

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

INTACK	Interrupt Acknowledge; 0 = RDSINT status preserved. 1 = Clears RDSINT.
MTFIFO	0 = If FIFO not empty, read and remove oldest FIFO entry; 1 = Clear RDS

	Receive FIFO.
STATUSONLY	Determines if data should be removed from the RDS FIFO.

```
01550 {
01551     si47x_rds_command rds_cmd;
01552     static uint16_t lastFreq;
01553     // checking current FUNC (Am or FM)
01554     if (currentTune != FM_TUNE_FREQ)
01555         return;
01556 
01557     if (lastFreq != currentWorkFrequency)
01558     {
01559         lastFreq = currentWorkFrequency;
01560         clearRdsBuffer2A();
01561         clearRdsBuffer2B();
01562         clearRdsBuffer0A();
01563     }
01564 
01565     waitToSend();
01566 
01567     rds_cmd.arg.INTACK = INTACK;
01568     rds_cmd.arg.MTFIFO = MTFIFO;
01569     rds_cmd.arg.STATUSONLY = STATUSONLY;
01570 
01571     Wire.beginTransmission(deviceAddress);
01572     Wire.write(FM_RDS_STATUS);
01573     Wire.write(rds_cmd.raw);
01574     Wire.endTransmission();
01575 
01576     do
01577     {
01578         waitToSend();
01579         // Gets response information
01580         Wire.requestFrom(deviceAddress, 13);
01581         for (uint8_t i = 0; i < 13; i++)
01582             currentRdsStatus.raw[i] = Wire.read();
01583     } while (currentRdsStatus.resp.ERR);
01584     delayMicroseconds(550);
01585 }
```

References SI4735::clearRdsBuffer0A(), SI4735::clearRdsBuffer2A(), SI4735::clearRdsBuffer2B(), and SI4735::waitToSend().

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

```
01793 {
01794 
01795     // Needs to get the "Text segment address code".
01796     // Each message should be ended by the code 0D (Hex)
01797 
01798     if (rdsTextAdress2A >= 16)
01799         rdsTextAdress2A = 0;
01800 
01801     getNext4Block(&rds_buffer2A[rdsTextAdress2A * 4]);
01802 
01803     rdsTextAdress2A += 4;
01804 
01805     return rds_buffer2A;
01806 }
```

References SI4735::getNext4Block(), SI4735::rds_buffer2A, and SI4735::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

```
01817 {
01818     si47x_rds_blockb blkB;
01819
01820     // getRdsStatus();
01821
01822     if (getRdsReceived())
01823     {
01824         if (getRdsGroupType() == 0)
01825         {
01826             // Process group type 0
01827             blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01828             blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01829
01830             rdsTextAdress0A = blkB.group0.address;
01831             if (rdsTextAdress0A >= 0 && rdsTextAdress0A < 4)
01832             {
01833                 getNext2Block(&rds_buffer0A[rdsTextAdress0A * 2]);
01834                 rds_buffer0A[8] = '\0';
01835                 return rds_buffer0A;
01836             }
01837         }
01838     }
01839     return NULL;
01840 }
```

References SI4735::getNext2Block(), SI4735::getRdsReceived(), SI4735::rds_buffer0A, and SI4735::rdsTextAdress0A.

char * SI4735::getRdsText2A (void)

Gets the Text processed for the 2A group.

Returns

char* string with the Text of the group A2

```
01850 {
01851     si47x_rds_blockb blkB;
01852
01853     // getRdsStatus();
01854     if (getRdsReceived())
01855     {
01856         if (getRdsGroupType() == 2 /* && getRdsVersionCode() == 0 */)
01857         {
01858             // Process group 2A
01859             // Decode B block information
01860             blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01861             blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01862             rdsTextAdress2A = blkB.group2.address;
01863
01864             if (rdsTextAdress2A >= 0 && rdsTextAdress2A < 16)
01865             {
01866                 getNext4Block(&rds_buffer2A[rdsTextAdress2A * 4]);
01867                 rds_buffer2A[63] = '\0';
01868                 return rds_buffer2A;
01869             }
01870         }
01871     }
01872     return NULL;
01873 }
```

References SI4735::getNext4Block(), SI4735::getRdsReceived(), SI4735::rds_buffer2A, and SI4735::rdsTextAdress2A.

char * SI4735::getRdsText2B (void)

Gets the Text processed for the 2B group.

Returns

char* string with the Text of the group AB

```
01883 {
01884     si47x_rds_blockb blkB;
01885
01886     // getRdsStatus();
01887     // if (getRdsReceived())
01888     // {
01889     // if (getRdsNewBlockB())
01890     // {
01891     if (getRdsGroupType() == 2 /* && getRdsVersionCode() == 1 */)
01892     {
01893         // Process group 2B
01894         blkB.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01895         blkB.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01896         rdsTextAdress2B = blkB.group2.address;
01897         if (rdsTextAdress2B >= 0 && rdsTextAdress2B < 16)
01898         {
01899             getNext2Block(&rds_buffer2B[rdsTextAdress2B * 2]);
01900             return rds_buffer2B;
01901         }
01902     }
01903     //  }
01904     // }
01905     return NULL;
01906 }
```

References SI4735::getNext2Block(), SI4735::rds_buffer2B, and SI4735::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.

```
01672 {
01673     si47x_rds_blockb blkb;
01674     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01675     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01676
01677     return blkb.refined.content;
01678 }
```

char * SI4735::getRdsTime (void)

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

Returns

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

```
01916 {
```

```
01917     // Under Test and construction
01918     // Need to check the Group Type before.
01919     si47x_rds_date_time dt;
01920
01921     uint16_t minute;
01922     uint16_t hour;
01923
01924     if (getRdsGroupType() == 4)
01925     {
01926         char offset_sign;
01927         int offset_h;
01928         int offset_m;
01929
01930         // uint16_t y, m, d;
01931
01932         dt.raw[4] = currentRdsStatus.resp.BLOCKBL;
01933         dt.raw[5] = currentRdsStatus.resp.BLOCKBH;
01934         dt.raw[2] = currentRdsStatus.resp.BLOCKCL;
01935         dt.raw[3] = currentRdsStatus.resp.BLOCKCH;
01936         dt.raw[0] = currentRdsStatus.resp.BLOCKDL;
01937         dt.raw[1] = currentRdsStatus.resp.BLOCKDH;
01938
01939         // Unfortunately it was necessary to wotk well on the GCC compiler
on 32-bit
01940         // platforms. See si47x_rds_date_time (typedef union) and CGG
"Crosses boundary" issue/features.
01941         // Now it is working on Atmega328, STM32, Arduino DUE, ESP32 and
more.
01942         minute = (dt.refined.minute2 << 2) | dt.refined.minute1;
01943         hour = (dt.refined.hour2 << 4) | dt.refined.hour1;
01944
01945         offset_sign = (dt.refined.offset_sense == 1) ? '+' : '-';
01946         offset_h = (dt.refined.offset * 30) / 60;
01947         offset_m = (dt.refined.offset * 30) - (offset_h * 60);
01948         // sprintf(rds_time, "%02u:%02u %c%02u:%02u", dt.refined.hour,
dt.refined.minute, offset_sign, offset_h, offset_m);
01949         sprintf(rds_time, "%02u:%02u %c%02u:%02u", hour, minute,
offset_sign, offset_h, offset_m);
01950
01951         return rds_time;
01952     }
01953
01954     return NULL;
01955 }
```

References SI4735::rds_time.

uint8_t SI4735::getRdsVersionCode (void)

Gets the version code (extracted from the Block B)

Returns

0=A or 1=B

```
01688 {
01689     si47x_rds_blockb blkb;
01690
01691     blkb.raw.lowValue = currentRdsStatus.resp.BLOCKBL;
01692     blkb.raw.highValue = currentRdsStatus.resp.BLOCKBH;
01693
01694     return blkb.refined.versionCode;
01695 }
```

void SI4735::RdsInit ()

Starts the control member variables for RDS.

RDS implementation

This method is called by setRdsConfig()

See also

setRdsConfig()

```
01389 {
01390     clearRdsBuffer2A();
01391     clearRdsBuffer2B();
01392     clearRdsBuffer0A();
01393     rdsTextAdress2A = rdsTextAdress2B = lastTextFlagAB = rdsTextAdress0A =
0;
01394 }
```

References SI4735::clearRdsBuffer0A(), SI4735::clearRdsBuffer2A(), and SI4735::clearRdsBuffer2B().

Referenced by SI4735::setRdsConfig().

void SI4735::setRdsConfig (uint8_t *RDSEN*, 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 (RDSEN) 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

uint8_t	RDSEN RDS Processing Enable; 1 = RDS processing enabled.
uint8_t	BLETHA Block Error Threshold BLOCKA.
uint8_t	BLETHB Block Error Threshold BLOCKB.
uint8_t	BLETHC Block Error Threshold BLOCKC.
uint8_t	BLETHD Block Error Threshold BLOCKD.

```
01462 {
01463     si47x_property property;
01464     si47x_rds_config config;
01465
01466     waitToSend();
01467
01468     // Set property value
01469     property.value = FM_RDS_CONFIG;
01470
01471     // Arguments
01472     config.arg.RDSEN = RDSEN;
01473     config.arg.BLETHA = BLETHA;
01474     config.arg.BLETHB = BLETHB;
01475     config.arg.BLETHC = BLETHC;
01476     config.arg.BLETHD = BLETHD;
01477     config.arg.DUMMY1 = 0;
01478
```

```
01479     Wire.beginTransmission(deviceAddress);
01480     Wire.write(SET_PROPERTY);
01481     Wire.write(0x00);                    // Always 0x00 (I need to check it)
01482     Wire.write(property.raw.byteHigh); // Send property - High byte - most
significant first
01483     Wire.write(property.raw.byteLow);  // Low byte
01484     Wire.write(config.raw[1]);         // Send the argments. Most
significant first
01485     Wire.write(config.raw[0]);
01486     Wire.endTransmission();
01487     delayMicroseconds(550);
01488
01489     RdsInit();
01490 }
```

References SI4735::RdsInit(), and SI4735::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

RDSRECV	If set, generate RDSINT when RDS FIFO has at least FM_RDS_INT_FIFO_COUNT entries.
RDSSYNCLOST	If set, generate RDSINT when RDS loses synchronization.
RDSSYNCFOUND	set, generate RDSINT when RDS gains synchronization.
RDSNEWBLOCKA	If set, generate an interrupt when Block A data is found or subsequently changed
RDSNEWBLOCKB	If set, generate an interrupt when Block B data is found or subsequently changed

```
01508 {
01509     si47x_property property;
01510     si47x_rds_int_source rds_int_source;
01511
01512     if (currentTune != FM_TUNE_FREQ)
01513         return;
01514
01515     rds_int_source.refined.RDSNEWBLOCKB = RDSNEWBLOCKB;
01516     rds_int_source.refined.RDSNEWBLOCKA = RDSNEWBLOCKA;
01517     rds_int_source.refined.RDSSYNCFOUND = RDSSYNCFOUND;
01518     rds_int_source.refined.RDSSYNCLOST = RDSSYNCLOST;
01519     rds_int_source.refined.RDSRECV = RDSRECV;
01520     rds_int_source.refined.DUMMY1 = 0;
01521     rds_int_source.refined.DUMMY2 = 0;
01522
01523     property.value = FM_RDS_INT_SOURCE;
01524
01525     waitToSend();
01526
01527     Wire.beginTransmission(deviceAddress);
01528     Wire.write(SET_PROPERTY);
01529     Wire.write(0x00);                    // Always 0x00 (I need to check it)
01530     Wire.write(property.raw.byteHigh); // Send property - High byte - most
significant first
01531     Wire.write(property.raw.byteLow);  // Low byte
01532     Wire.write(rds_int_source.raw[1]); // Send the argments. Most
significant first
01533     Wire.write(rds_int_source.raw[0]);
01534     Wire.endTransmission();
01535     waitToSend();
01536 }
```

References SI4735::waitToSend().

Frequency and Si47XX device status

Functions

uint16_t SI4735::getFrequency (void)
Gets the current frequency of the Si4735 (AM or FM)

uint16_t SI4735::getCurrentFrequency ()
Gets the current frequency.

void SI4735::getStatus (uint8_t, uint8_t)
Gets the current status of the Si47XX device.

void SI4735::getStatus ()
Tune the receiver.

void SI4735::getAutomaticGainControl ()
Queries Automatic Gain Control STATUS.

void SI4735::setAutomaticGainControl (uint8_t AGCDIS, uint8_t AGCIDX)
Automatic Gain Control setup.

void SI4735::setAvcAmMaxGain (uint8_t gain)
Sets the maximum gain for automatic volume control.

void SI4735::getCurrentReceivedSignalQuality (uint8_t INTACK)
Overrides the AGC setting.

void SI4735::getCurrentReceivedSignalQuality (void)
Queries the status of the Received Signal Quality (RSQ) of the current channel (FM_RSQ_STATUS)

Detailed Description

Function Documentation

void SI4735::getAutomaticGainControl ()

Queries Automatic Gain Control STATUS.

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

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.

```
01096 {
01097     uint8_t cmd;
01098
01099     if (currentTune == FM_TUNE_FREQ)
01100     { // FM TUNE
01101         cmd = FM_AGC_STATUS;
01102     }
01103     else
01104     { // AM TUNE - SAME COMMAND used on SSB mode
01105         cmd = AM_AGC_STATUS;
01106     }
01107
01108     waitToSend();
01109
01110     Wire.beginTransmission(deviceAddress);
01111     Wire.write(cmd);
01112     Wire.endTransmission();
01113
01114     do
01115     {
01116         waitToSend();
01117         Wire.requestFrom(deviceAddress, 3);
01118         currentAgcStatus.raw[0] = Wire.read(); // STATUS response
01119         currentAgcStatus.raw[1] = Wire.read(); // RESP 1
01120         currentAgcStatus.raw[2] = Wire.read(); // RESP 2
01121     } while (currentAgcStatus.refined.ERR);    // If error, try get AGC
status again.
01122 }
```

References SI4735::waitToSend().

uint16_t SI4735::getCurrentFrequency ()

Gets the current frequency.

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()

```
01032 {
01033     return currentWorkFrequency;
01034 }
```

void SI4735::getCurrentReceivedSignalQuality (uint8_t *INTACK*)

Overrides the AGC setting.

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

This method sould be called berore call getCurrentRSSI(), getCurrentSNR() etc. Command FM_RSQ_STATUS

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 75 and 141

Parameters

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

```
01196 {
01197         uint8_t arg;
01198         uint8_t cmd;
01199         int sizeResponse;
01200 
01201         if (currentTune == FM_TUNE_FREQ)
01202         { // FM TUNE
01203             cmd = FM_RSQ_STATUS;
01204             sizeResponse = 8; // Check it
01205         }
01206         else
01207         { // AM TUNE
01208             cmd = AM_RSQ_STATUS;
01209             sizeResponse = 6; // Check it
01210         }
01211 
01212         waitToSend();
01213 
01214         arg = INTACK;
01215         Wire.beginTransmission(deviceAddress);
01216         Wire.write(cmd);
01217         Wire.write(arg); // send B00000001
01218         Wire.endTransmission();
01219 
01220         // Check it
01221         // do
01222         //{
01223             waitToSend();
01224             Wire.requestFrom(deviceAddress, sizeResponse);
01225             // Gets response information
01226             for (uint8_t i = 0; i < sizeResponse; i++)
01227                 currentRqsStatus.raw[i] = Wire.read();
01228         //} while (currentRqsStatus.resp.ERR); // Try again if error found
01229 }
```

References SI4735::waitToSend().

void SI4735::getCurrentReceivedSignalQuality (void)

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

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 75 and 141

Parameters

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

```
01243 {
01244     getCurrentReceivedSignalQuality(0);
01245 }
```

uint16_t SI4735::getFrequency (void)

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

Device Status Information

The method status do it an more. See getStatus below.

See also

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

```
01009 {
01010     si47x_frequency freq;
01011     getStatus(0, 1);
01012 
01013     freq.raw.FREQL = currentStatus.resp.READFREQL;
01014     freq.raw.FREQH = currentStatus.resp.READFREQH;
01015 
01016     currentWorkFrequency = freq.value;
01017     return freq.value;
01018 }
```

void SI4735::getStatus ()

Tune the receiver.

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

See also

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

```
01080 {
01081     getStatus(0, 1);
01082 }
```

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

Gets the current status of the Si47XX device.

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

See also

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

Parameters

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

```
01047 {
01048     si47x_tune_status status;
01049     uint8_t cmd = (currentTune == FM_TUNE_FREQ) ? FM_TUNE_STATUS :
AM_TUNE_STATUS;
01050 
01051     waitToSend();
01052 
01053     status.arg.INTACK = INTACK;
01054     status.arg.CANCEL = CANCEL;
01055 
01056     Wire.beginTransmission(deviceAddress);
01057     Wire.write(cmd);
01058     Wire.write(status.raw);
01059     Wire.endTransmission();
01060     // Reads the current status (including current frequency).
01061     do
01062     {
01063         waitToSend();
01064         Wire.requestFrom(deviceAddress, 8); // Check it
01065         // Gets response information
01066         for (uint8_t i = 0; i < 8; i++)
01067             currentStatus.raw[i] = Wire.read();
01068     } while (currentStatus.resp.ERR); // If error, try it again
01069     waitToSend();
01070 }
```

References SI4735::waitToSend().

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

Automatic Gain Control setup.

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)).

```
01141 {
01142     si47x_agc_overrride agc;
01143
01144     uint8_t cmd;
01145
01146     cmd = (currentTune == FM_TUNE_FREQ) ? FM_AGC_OVERRIDE : AM_AGC_OVERRIDE;
01147
01148     agc.arg.AGCDIS = AGCDIS;
01149     agc.arg.AGCIDX = AGCIDX;
01150
01151     waitToSend();
01152
01153     Wire.beginTransmission(deviceAddress);
01154     Wire.write(cmd);
01155     Wire.write(agc.raw[0]);
01156     Wire.write(agc.raw[1]);
01157     Wire.endTransmission();
01158
01159     waitToSend();
01160 }
```

References SI4735::waitToSend().

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

setAvcAmMaxGain()

Parameters

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

```
01174                                           {
01175     uint16_t aux;
01176     aux = ( gain > 12 && gain < 193 )? (gain * 340) : (48 * 340);
01177     currentAvcAmMaxGain =  gain;
01178     sendProperty(AM_AUTOMATIC_VOLUME_CONTROL_MAX_GAIN, aux);
01179 }
```

Host and slave MCU setup

Functions

void SI4735::reset (void)
Reset the SI473X

void SI4735::waitToSend (void)
Reset the Si47XX device.

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)
Configure the Si47XX to power it up.

void SI4735::analogPowerUp (void)
Power the receiver up. Call setPowerUp before call this method.

void SI4735::powerDown (void)
Deprecated. Use radioPowerUp.

Detailed Description

Function Documentation

void SI4735::analogPowerUp (void)

Power the receiver up. Call setPowerUp before call this method.

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)

Deprecated. Use radioPowerUp.

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.beginTransmission(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)

Configure the Si47XX to power it up.

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.beginTransmission(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 chenge the defaul 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 apllied)]
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 -> GPO2 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 frequancy 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)

Reset the Si47XX device.

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 si47x_rqs_status	resp	

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 (0â€“127 dBÎ¼V).
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 si47x_rds_status	resp	

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	RDSSYNCFOUND: 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	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	RDSSYNCLOST: 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 Structures

union si47x_agc_status
struct si47x_agc_status.refined
union si47x_agc_overrride
struct si47x_agc_overrride.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_overrride

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_overrride	arg	
uint8_t	raw[2]	

struct si47x_agc_overrride.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 passand 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 manupulation. *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 aproach 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 si473x_powerup	arg	
uint8_t	raw[2]	

struct si473x_powerup.arg

Data Fields:

uint8_t	CTSIEN: 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 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 manupulation.

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) Interrup; 0 = interrupt has not been triggered.
uint8_t	RSSI	Read Frequency Low byte.
uint8_t	SNR	Received Signal Strength Indicator (dBÎ¼V)
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 si47x_firmware_query_library	resp	

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	

Si4735-D60 Single Side Band (SSB) support

Functions

void SI4735::setSSBBfo (int offset)

Sets the SSB Beat Frequency Offset (BFO).

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

Sets the SSB receiver mode.

void SI4735::setSSBDspAfc (uint8_t DSP_AFCDIS)

Sets DSP AFC disable or enable.

void SI4735::setSSBSoftMute (uint8_t SMUTESEL)

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

void SI4735::setSSBAutomaticVolumeControl (uint8_t AVCEN)

Sets SSB Automatic Volume Control (AVC) for SSB mode.

void SI4735::setSSBAvcDivider (uint8_t AVC_DIVIDER)

Sets AVC Divider.

void SI4735::setSBBSidebandCutoffFilter (uint8_t SBCUTFLT)
Sets SBB Sideband Cutoff Filter for band pass and low pass filters.

void SI4735::setSSBAudioBandwidth (uint8_t AUDIOBW)
SSB Audio Bandwidth for SSB mode.

void SI4735::setSSB (uint8_t usblsb)
Set the radio to AM function.

void SI4735::setSSB (uint16_t fromFreq, uint16_t toFreq, uint16_t intialFreq, uint16_t step, uint8_t usblsb)
void SI4735::sendSSBModeProperty ()
Sends the property command to the device.

si47x_firmware_query_library SI4735::queryLibraryId ()
Query the library information of the Si47XX device.

void SI4735::patchPowerUp ()
This method can be used to prepare the device to apply SSBRX patch.

void SI4735::ssbSetup ()
Starts the Si473X device on SSB (same AM Mode).

void SI4735::ssbPowerUp ()
This function can be useful for debug and test.

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.

bool SI4735::downloadPatch (int eeprom_i2c_address)
Transfers the content of a patch stored in a eeprom to the SI4735 device.

Detailed Description

Function Documentation

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.

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;

See also

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

Parameters

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

Returns

false if an error is found.

```
02427 {
02428     uint8_t content;
02429     register int i, offset;
02430     // Send patch to the SI4735 device
02431     for (offset = 0; offset < (int) ssb_patch_content_size; offset += 8)
02432     {
02433         Wire.beginTransmission(deviceAddress);
02434         for (i = 0; i < 8; i++)
02435         {
02436             content = pgm_read_byte_near(ssb_patch_content + (i + offset));
02437             Wire.write(content);
02438         }
02439         Wire.endTransmission();
02440
02441         // Testing download performance
02442         // approach 1 - Faster - less secure (it might crash in some
architectures)
02443         delayMicroseconds(MIN_DELAY_WAIT_SEND_LOOP); // Need check the
minimum value
02444
02445         // approach 2 - More control. A little more secure than approach 1
02446         /*
02447         do
02448         {
02449             delayMicroseconds(150); // Minimum delay founded (Need check the
minimum value)
02450             Wire.requestFrom(deviceAddress, 1);
02451         } while (!(Wire.read() & B10000000));
02452         */
02453
02454         // approach 3 - same approach 2
02455         // waitToSend();
02456
02457         // approach 4 - safer
02458         /*
02459         waitToSend();
02460         uint8_t cmd_status;
02461         Uncomment the lines below if you want to check erro.
02462         Wire.requestFrom(deviceAddress, 1);
02463         cmd_status = Wire.read();
02464         The SI4735 issues a status after each 8 byte transfered.
02465         Just the bit 7 (CTS) should be seted. if bit 6 (ERR) is seted, the
system halts.
```

```
02466         if (cmd_status != 0x80)
02467            return false;
02468         */
02469     }
02470     delayMicroseconds(250);
02471     return true;
02472 }
```

bool SI4735::downloadPatch (int *eeprom_i2c_address*)

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

ATTENTION: Under construction...

See also

the sketch write_ssb_patch_eeprom.ino (TO DO)

Parameters

eeprom_i2c_address	

Returns

false if an error is found.

```
02489 {
02490     int ssb_patch_content_size;
02491     uint8_t cmd_status;
02492     int i, offset;
02493     uint8_t eepromPage[8];
02494
02495     union {
02496         struct
02497         {
02498             uint8_t lowByte;
02499             uint8_t highByte;
02500         } raw;
02501         uint16_t value;
02502     } eeprom;
02503
02504     // The first two bytes are the size of the patches
02505     // Set the position in the eeprom to read the size of the patch content
02506     Wire.beginTransmission(eeprom_i2c_address);
02507     Wire.write(0); // writes the most significant byte
02508     Wire.write(0); // writes the less significant byte
02509     Wire.endTransmission();
02510     Wire.requestFrom(eeprom_i2c_address, 2);
02511     eeprom.raw.highByte = Wire.read();
02512     eeprom.raw.lowByte = Wire.read();
02513
02514     ssb_patch_content_size = eeprom.value;
02515
02516     // the patch content starts on position 2 (the first two bytes are the
size of the patch)
02517     for (offset = 2; offset < ssb_patch_content_size; offset += 8)
02518     {
02519         // Set the position in the eeprom to read next 8 bytes
02520         eeprom.value = offset;
02521         Wire.beginTransmission(eeprom_i2c_address);
02522         Wire.write(eeprom.raw.highByte); // writes the most significant byte
02523         Wire.write(eeprom.raw.lowByte);  // writes the less significant byte
02524         Wire.endTransmission();
02525
02526         // Reads the next 8 bytes from eeprom
02527         Wire.requestFrom(eeprom_i2c_address, 8);
02528         for (i = 0; i < 8; i++)
02529             eepromPage[i] = Wire.read();
02530
02531         // sends the page (8 bytes) to the SI4735
02532         Wire.beginTransmission(deviceAddress);
02533         for (i = 0; i < 8; i++)
```

```
02534             Wire.write(eepromPage[i]);
02535         Wire.endTransmission();
02536
02537         waitToSend();
02538
02539         Wire.requestFrom(deviceAddress, 1);
02540         cmd_status = Wire.read();
02541         // The SI4735 issues a status after each 8 byte transfered.
02542         // Just the bit 7 (CTS) should be seted. if bit 6 (ERR) is seted,
the system halts.
02543         if (cmd_status != 0x80)
02544             return false;
02545     }
02546     delayMicroseconds(250);
02547     return true;
02548 }
```

References SI4735::waitToSend().

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.

```
02341 {
02342     waitToSend();
02343     Wire.beginTransmission(deviceAddress);
02344     Wire.write(POWER_UP);
02345     Wire.write(0b00110001);          // Set to AM, Enable External Crystal
Oscillator; Set patch enable; GPO2 output disabled; CTS interrupt disabled.
02346     Wire.write(SI473X_ANALOG_AUDIO); // Set to Analog Output
02347     Wire.endTransmission();
02348     delayMicroseconds(2500);
02349 }
```

References SI4735::waitToSend().

si47x_firmware_query_library SI4735::queryLibraryId ()

Query the library information of the Si47XX device.

SI47XX PATCH RESOURCES

Used to confirm if the patch is compatible with the internal device library revision.

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 64 and 215-220.

struct si47x_firmware_query_library

Returns

a struct si47x_firmware_query_library (see it in SI4735.h)

```
02302 {
02303     si47x_firmware_query_library libraryID;
02304
02305     powerDown(); // Is it necessary
02306
02307     // delay(500);
02308
02309     waitToSend();
02310     Wire.beginTransmission(deviceAddress);
02311     Wire.write(POWER_UP);
02312     Wire.write(0b00011111);          // Set to Read Library ID, disable
interrupt; disable GPO2OEN; boot normaly; enable External Crystal Oscillator  .
02313     Wire.write(SI473X_ANALOG_AUDIO); // Set to Analog Line Input.
02314     Wire.endTransmission();
02315
02316     do
02317     {
02318         waitToSend();
02319         Wire.requestFrom(deviceAddress, 8);
02320         for (int i = 0; i < 8; i++)
02321             libraryID.raw[i] = Wire.read();
02322     } while (libraryID.resp.ERR); // If error found, try it again.
02323
02324     delayMicroseconds(2500);
02325
02326     return libraryID;
02327 }
```

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

void SI4735::sendSSBModeProperty () [protected]

Sends the property command to the device.

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

```
02259 {
02260     si47x_property property;
02261     property.value = SSB_MODE;
02262     waitToSend();
02263     Wire.beginTransmission(deviceAddress);
02264     Wire.write(SET_PROPERTY);
02265     Wire.write(0x00);                    // Always 0x00
02266     Wire.write(property.raw.byteHigh); // High byte first
02267     Wire.write(property.raw.byteLow);  // Low byte after
02268     Wire.write(currentSSBMode.raw[1]); // SSB MODE params; freq. high byte
first
02269     Wire.write(currentSSBMode.raw[0]); // SSB MODE params; freq. low byte
after
02270
02271     Wire.endTransmission();
02272     delayMicroseconds(550);
02273 }
```

References SI4735::waitToSend().

Referenced by SI4735::setSBBSidebandCutoffFilter(), SI4735::setSSBAudioBandwidth(), SI4735::setSSBAutomaticVolumeControl(), SI4735::setSSBAvcDivider(), SI4735::setSSBConfig(), SI4735::setSSBDspAfc(), and SI4735::setSSBSoftMute().

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

SBCUTFLT	0 or 1; see above

```
02160 {
02161     currentSSBMode.param.SBCUTFLT = SBCUTFLT;
02162     sendSSBModeProperty();
02163 }
```

References SI4735::sendSSBModeProperty().

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

Set the radio to SSB (LW/MW/SW) function.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; pages 13 and 14

Parameters

fromFreq	minimum frequency for the band
toFreq	maximum frequency for the band
initialFreq	initial frequency
step	step used to go to the next channel
usblsb	SSB Upper Side Band (USB) and Lower Side Band (LSB) Selection; value 2 (banary 10) = USB; value 1 (banary 01) = LSB.

```
02238 {
02239     currentMinimumFrequency = fromFreq;
02240     currentMaximumFrequency = toFreq;
02241     currentStep = step;
02242
02243     if (initialFreq < fromFreq || initialFreq > toFreq)
02244         initialFreq = fromFreq;
02245
02246     setSSB(usblsb);
02247
02248     currentWorkFrequency = initialFreq;
02249     setFrequency(currentWorkFrequency);
02250     delayMicroseconds(550);
02251 }
```

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

usblsb	upper or lower side band; 1 = LSB; 2 = USB

```
02210 {
02211     // Is it needed to load patch when switch to SSB?
02212     // powerDown();
02213     // It starts with the same AM parameters.
02214     setPowerUp(1, 1, 0, 1, 1, SI473X_ANALOG_AUDIO);
02215     radioPowerUp();
02216     // ssbPowerUp(); // Not used for regular operation
02217     setVolume(volume); // Set to previus configured volume
02218     currentSsbStatus = usblsb;
```

```
02219     lastMode = SSB_CURRENT_MODE;
02220 }
```

References SI4735::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.

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

AUDIOBW	the valid values are 0, 1, 2, 3, 4 or 5; see description above

```
02190 {
02191     // Sets the audio filter property parameter
02192     currentSSBMode.param.AUDIOBW = AUDIOBW;
02193     sendSSBModeProperty();
02194 }
```

References SI4735::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

AVCEN	0 = Disable AVC; 1 = Enable AVC (default).

```
02126 {
02127     currentSSBMode.param.AVCEN = AVCEN;
02128     sendSSBModeProperty();
02129 }
```

References SI4735::sendSSBModeProperty().

void SI4735::setSSBAvcDivider (uint8_t *AVC_DIVIDER*)

Sets AVC Divider.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

AVC_DIVIDER	SSB mode, set divider = 0; SYNC mode, set divider = 3; Other values = not allowed.

```
02141 {
02142     currentSSBMode.param.AVC_DIVIDER = AVC_DIVIDER;
02143     sendSSBModeProperty();
02144 }
```

References SI4735::sendSSBModeProperty().

void SI4735::setSSBBfo (int *offset*)

Sets the SSB Beat Frequency Offset (BFO).

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 paches 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++. Actally, 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.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; pages 5 and 23

Parameters

offset	16-bit signed value (unit in Hz). The valid range is -16383 to +16383 Hz.

```
02022 {
02023
02024     si47x_property property;
02025     si47x_frequency bfo_offset;
02026
02027     if (currentTune == FM_TUNE_FREQ) // Only for AM/SSB mode
02028         return;
02029
02030     waitToSend();
02031
02032     property.value = SSB_BFO;
02033     bfo_offset.value = offset;
02034
02035     Wire.beginTransmission(deviceAddress);
02036     Wire.write(SET_PROPERTY);
02037     Wire.write(0x00);                   // Always 0x00
02038     Wire.write(property.raw.byteHigh); // High byte first
02039     Wire.write(property.raw.byteLow);  // Low byte after
02040     Wire.write(bfo_offset.raw.FREQH);  // Offset freq. high byte first
02041     Wire.write(bfo_offset.raw.FREQL);  // Offset freq. low byte first
02042
02043     Wire.endTransmission();
02044     delayMicroseconds(550);
02045 }
```

References SI4735::waitToSend().

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

Sets the SSB receiver mode.

You can use this method for:

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 disbable automatic volume control (AVC) function.

See also

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE; page 24

Parameters

AUDIOBW	SSB Audio bandwidth; 0 = 1.2KHz (default); 1=2.2KHz; 2=3KHz; 3=4KHz; 4=500Hz; 5=1KHz.
SBCUTFLT	SSB side band cutoff filter for band passand 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).
AVC_DIVIDER	set 0 for SSB mode; set 3 for SYNC mode.
AVCEN	SSB Automatic Volume Control (AVC) enable; 0=disable; 1=enable (default).
SMUTESEL	SSB Soft-mute Based on RSSI or SNR.
DSP_AFCDIS	DSP AFC Disable or enable; 0=SYNC MODE, AFC enable; 1=SSB MODE, AFC disable.

```
02071 {
```

```
02072     if (currentTune == FM_TUNE_FREQ) // Only AM/SSB mode
02073         return;
02074
02075     currentSSBMode.param.AUDIOBW = AUDIOBW;
02076     currentSSBMode.param.SBCUTFLT = SBCUTFLT;
02077     currentSSBMode.param.AVC_DIVIDER = AVC_DIVIDER;
02078     currentSSBMode.param.AVCEN = AVCEN;
02079     currentSSBMode.param.SMUTESEL = SMUTESEL;
02080     currentSSBMode.param.DUMMY1 = 0;
02081     currentSSBMode.param.DSP_AFCDIS = DSP_AFCDIS;
02082
02083     sendSSBModeProperty();
02084 }
```

References SI4735::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

DSP_AFCDIS	0 = SYNC mode, AFC enable; 1 = SSB mode, AFC disable

```
02096 {
02097     currentSSBMode.param.DSP_AFCDIS = DSP_AFCDIS;
02098     sendSSBModeProperty();
02099 }
```

References SI4735::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

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

```
02111 {
02112     currentSSBMode.param.SMUTESEL = SMUTESEL;
02113     sendSSBModeProperty();
02114 }
```

References SI4735::sendSSBModeProperty().

void SI4735::ssbPowerUp ()

This function can be useful for debug and test.

```
02371 {
02372     waitToSend();
02373     Wire.beginTransmission(deviceAddress);
02374     Wire.write(POWER_UP);
02375     Wire.write(0b00010001); // Set to AM/SSB, disable interrupt; disable
GPO2OEN; boot normaly; enable External Crystal Oscillator  .
02376     Wire.write(0b00000101); // Set to Analog Line Input.
02377     Wire.endTransmission();
02378     delayMicroseconds(2500);
02379
02380     powerUp.arg.CTSIEN = 0;          // 1 -> Interrupt anabled;
02381     powerUp.arg.GPO2OEN = 0;         // 1 -> GPO2 Output Enable;
02382     powerUp.arg.PATCH = 0;           // 0 -> Boot normally;
```

```
02383     powerUp.arg.XOSCEN = 1;           // 1 -> Use external crystal
oscillator;
02384     powerUp.arg.FUNC = 1;             // 0 = FM Receive; 1 = AM/SSB
(LW/MW/SW) Receiver.
02385     powerUp.arg.OPMODE = 0b00000101; // 0x5 = 00000101 = Analog audio
outputs (LOUT/ROUT).
02386 }
```

References SI4735::waitToSend().

void SI4735::ssbSetup ()

Starts the Si473X device on SSB (same AM Mode).

Same SI4735::setup optimized to improve loading patch performance

```
02359 {
02360     // setPowerUp(powerUp.arg.CTSIEN, 0, 0, 1, 1, SI473X_ANALOG_AUDIO);
02361     reset();
02362     // radioPowerUp();
02363 }
```

References SI4735::reset().

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 corrent 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 intialFreq, 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.

bool SI4735::isCurrentTuneFM ()
Returns true if the current function is FM (FM_TUNE_FREQ).

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 }
```

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).

```
00623 {
00624     return (currentTune == FM_TUNE_FREQ);
00625 }
```

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 previus 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

fromFreq	minimum frequency for the band
toFreq	maximum frequency for the band
initialFreq	initial frequency
step	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 previus 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

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

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

References SI4735::setFM().

void SI4735::setFrequency (uint16_t *freq*)

Set the frequency to the corrent 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

uint16_t	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
or USB
00445         currentFrequencyParams.arg.FAST = 1;                  // Used just
on AM and FM
00446         currentFrequencyParams.arg.FREEZE = 0;                // Used just
on FM
00447     }
00448
00449     Wire.beginTransmission(deviceAddress);
00450     Wire.write(currentTune);
00451     Wire.write(currentFrequencyParams.raw[0]); // Send a byte with FAST and
FREEZE information; if not FM must be 0;
00452     Wire.write(currentFrequencyParams.arg.FREQH);
00453     Wire.write(currentFrequencyParams.arg.FREQL);
00454     Wire.write(currentFrequencyParams.arg.ANTCAPH);
00455     // If current tune is not FM sent one more byte
00456     if (currentTune != FM_TUNE_FREQ)
00457         Wire.write(currentFrequencyParams.arg.ANTCAPL);
00458
00459     Wire.endTransmission();
00460     waitToSend();                 // Wait for the si473x is ready.
00461     currentWorkFrequency = freq; // check it
00462     delay(MAX_DELAY_AFTER_SET_FREQUENCY); // For some reason I need to delay
here.
00463 }
```

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

step	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

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

```
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)
Wait for the Si47XX device is ready to receive a command.

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.beginTransmission(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

uint8_t	resetPin Digital Arduino Pin used to RESET command
uint8_t	interruptPin interrupt Arduino Pin (see your Arduino pinout). If less than 0, iterrupt disabled
uint8_t	defaultFunction
uint8_t	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 anabled or disable;
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*)

Wait for the Si47XX device is ready to receive a command.

Starts the Si473X device.

Use this setup if you are not using interrupt resource

Parameters

uint8_t	resetPin Digital Arduino Pin used to RESET command
uint8_t	defaultFunction

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

Si47XX filter setup

Functions

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

Selects the bandwidth of the channel filter for AM reception.

Detailed Description

Function Documentation

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. It works only in AM / SSB (LW/MW/SW)

See also

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

Parameters

AMCHFLT	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).
AMPLFLT	Enables the AM Power Line Noise Rejection Filter.

```
00650 {
00651     si47x_bandwidth_config filter;
00652     si47x_property property;
```

```
00653
00654     if (currentTune == FM_TUNE_FREQ) // Only for AM/SSB mode
00655         return;
00656
00657     if (AMCHFLT > 6)
00658         return;
00659
00660     property.value = AM_CHANNEL_FILTER;
00661
00662     filter.param.AMCHFLT = AMCHFLT;
00663     filter.param.AMPLFLT = AMPLFLT;
00664
00665     waitToSend();
00666     this->volume = volume;
00667     Wire.beginTransmission(deviceAddress);
00668     Wire.write(SET_PROPERTY);
00669     Wire.write(0x00);                   // Always 0x00
00670     Wire.write(property.raw.byteHigh); // High byte first
00671     Wire.write(property.raw.byteLow);  // Low byte after
00672     Wire.write(filter.raw[1]);          // Raw data for AMCHFLT and
00673     Wire.write(filter.raw[0]);          // AMPLFLT
00674     Wire.endTransmission();
00675     waitToSend();
00676 }
```

References SI4735::waitToSend().

Tools method

Functions

void SI4735::sendProperty (uint16_t propertyValue, uint16_t param)

wait for interrupt (useful if you are using interrupt resource)

Detailed Description

Function Documentation

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

wait for interrupt (useful if you are using interrupt resource)

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.

```
00690 {
00691     si47x_property property;
00692     si47x_property param;
00693
00694     property.value = propertyValue;
00695     param.value = parameter;
00696     waitToSend();
00697     Wire.beginTransmission(deviceAddress);
00698     Wire.write(SET_PROPERTY);
00699     Wire.write(0x00);
00700     Wire.write(property.raw.byteHigh); // Send property - High byte - most
significant first
```

```
00701     Wire.write(property.raw.byteLow);  // Send property - Low byte - less
significant after
00702     Wire.write(param.raw.byteHigh);    // Send the argments. High Byte -
Most significant first
00703     Wire.write(param.raw.byteLow);     // Send the argments. Low Byte - Less
significant after
00704     Wire.endTransmission();
00705     delayMicroseconds(550);
00706 }
```

References SI4735::waitToSend().

Tune

Functions

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

Look for a station (Automatic tune)

void SI4735::seekStationUp ()

Search for the next station.

void SI4735::seekStationDown ()

Search the previous station.

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.

void SI4735::setSeekAmSpacing (uint16_t spacing)

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

void SI4735::setSeekSrnThreshold (uint16_t value)

Sets the SNR threshold for a valid AM Seek/Tune.

void SI4735::setSeekRssiThreshold (uint16_t value)

Sets the RSSI threshold for a valid AM Seek/Tune.

Detailed Description

Function Documentation

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

Look for a station (Automatic tune)

See also

Si47XX PROGRAMMING GUIDE; AN332; pages 55, 72, 125 and 137

Parameters

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

```
01260 {
01261     si47x_seek seek;
01262
01263     // Check which FUNCTION (AM or FM) is working now
01264     uint8_t seek_start = (currentTune == FM_TUNE_FREQ) ? FM_SEEK_START :
AM_SEEK_START;
01265
01266     waitToSend();
01267
01268     seek.arg.SEEKUP = SEEKUP;
01269     seek.arg.WRAP = WRAP;
01270
01271     Wire.beginTransmission(deviceAddress);
01272     Wire.write(seek_start);
01273     Wire.write(seek.raw);
01274
01275     if (seek_start == AM_SEEK_START)
01276     {
01277         Wire.write(0x00); // Always 0
01278         Wire.write(0x00); // Always 0
01279         Wire.write(0x00); // Tuning Capacitor: The tuning capacitor value
01280         Wire.write(0x00); //                   will be selected
automatically.
01281     }
01282
01283     Wire.endTransmission();
01284     delay(100);
01285 }
```

References SI4735::waitToSend().

void SI4735::seekStationDown ()

Search the previous station.

See also

seekStation(uint8_t SEEKUP, uint8_t WRAP)

```
01309 {
01310     seekStation(0, 1);
01311     delay(50);
01312     getFrequency();
01313 }
```

void SI4735::seekStationUp ()

Search for the next station.

See also

seekStation(uint8_t SEEKUP, uint8_t WRAP)

```
01295 {
01296     seekStation(1, 1);
01297     delay(50);
01298     getFrequency();
01299 }
```

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

uint16_t	bottom - the bottom of the AM band for seek
uint16_t	top - the top of the AM band for seek

```
01326 {
01327     sendProperty(AM_SEEK_BAND_BOTTOM, bottom);
01328     sendProperty(AM_SEEK_BAND_TOP, top);
01329 }
```

void SI4735::setSeekAmSpacing (uint16_t *spacing*)

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

See also

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

Parameters

uint16_t	spacing - step in KHz

```
01341 {
01342     sendProperty(AM_SEEK_FREQ_SPACING, spacing);
01343 }
```

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 dBÎ¼V.

See also

Si47XX PROGRAMMING GUIDE; AN332; page 127

```
01369 {
01370     sendProperty(AM_SEEK_RSSI_THRESHOLD, value);
01371 }
```

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

```
01355 {
01356     sendProperty(AM_SEEK_SNR_THRESHOLD, value);
01357 }
```

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 manupulation. 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_overrride
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_overrride.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

See also

Si47XX PROGRAMMING GUIDE AN332
https://www.silabs.com/documents/public/application-notes/AN332.pdf

AN332 REV 0.8 UNIVERSAL PROGRAMMING GUIDE

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