Library for Silicon Laboratories Si4735 radio receiver chip and an Arduino microcontroller

Copyright 2012, 2013 Michael J. Kennedy

May 13, 2013

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. To view a copy of the GNU Lesser General Public License, visit these two web pages:

http://www.gnu.org/licenses/gpl-3.0.html

http://www.gnu.org/licenses/lgpl-3.0.html

This software can be found at the following web site:

https://sourceforge.net/projects/arduino-si4735/files/

The Mercurial repository can be downloaded with the following shell command:

```
hg clone http://hg.code.sf.net/p/arduino-si4735/si4735 Si4735
```

To report bugs or make suggestions contact me at:

michael.joseph.kennedy.1970 [_a_t_] gmail [dot] com

This library is a fork of previous libraries. Older versions can be found here:

- Ryan Owens, https://www.sparkfun.com/products/10342
- Wagner Sartori Junior, https://github.com/trunet/Si4735
- Jon Carrier, https://github.com/jjcarrier/Si4735

Warning: This library is not completely compatible with these other versions of the library! Most importantly, usage of begin() has been changed. ***You MUST change your code! If you don't, the program will still compile, but fail to work correctly!*** See section "Initializing the radio and library and setting the receive band" below for further details.

This library can be used with the following SparkFun boards:

- DEV-10342, "SI4735 AM & FM Receiver Shield," http://www.sparkfun.com/products/10342
- WRL-10906, "Si4735 FM/AM Radio Receiver Breakout," http://www.sparkfun.com/products/10906

Both contain a Si4735-C40 radio receiver chip. The library was tested with a shield marked on the bottom with a date of "4/14/11."

The Si4735 supports all modes available in the Si47xx family except for Weather Band (WB) receive and FM transmit. As a result, most chips in this family can be used with this library, if you are careful to not access missing modes. In some cases, small modifications to the library may be needed for full compatibility. Note: We do not yet support the auxiliary input mode found on the Si4735-D60 or later. Also note: The Si4700/01/02/03/08/09 chips are older and use a completely different command syntax. Therefore, they cannot be used with this library.

This library supports all current Arduinos including the 2009 (Duemilanove), Uno, Mega, Mega 2560, Leonardo, and Due.

To understand how to use this library, you **MUST** read this document and the Si4735.h file. You should also read through the example applications found in the examples directory. It will also be helpful to read the comments in the source code. Everything has been heavily commented to make understanding and customizing the library easy.

Warning: If you are going to either view or edit the library's source code or the Changes.text file, you **MUST** use a **programmer's text editor**. I recommend one of the following, multi-platform editors:

- Geany: http://www.geany.org/
- SciTE: http://www.scintilla.org/SciTE.html

Note: I have not yet tested the Arduino Due code because I do not currently have this board.

Note: I have not yet tested the I2C code because the Si4735 shield I used for testing does not permit using I2C. Also, I do not currently have the breakout board which can use I2C.

Background information

For more information on the Silicon Labs Si4735, see the following documents. (Check SparkFun and Silicon Labs web sites.):

- Data sheet: Si4734/35-C40 — Gives hardware description. (Get version on SparkFun web site.)
- Application Note: AN332: "Si47xx Programming Guide" — Explains the Si4735 (and similar chips) from a software perspective.

  Warning: The "Si47xx Programming Guide" implies that each command returns a status byte with the current interrupts. However, in most cases, the status byte contains random garbage. The exceptions are:

  - The CTS interrupt is always up-to-date. The radio chip always updates the CTS bit in the status byte whenever the CTS interrupt changes. Therefore, you can always get the current status of CTS by reading the status byte, regardless of the previous command sent.
  - A few commands return a status byte with their interrupt updated:

Command	Interrupt
TUNE_STATUS	STC
RSQ_STATUS	RSQ
RDS_STATUS	RDS

  - When an interrupt pulse is received by the Arduino, the only way to accurately know the status of the non CTS interrupts is to send the GET_INT_STATUS command and then read its status byte.
- Application Note: AN383: "Si47xx Antenna, Schematic, Layout, and Design Guidelines" — Gives important information on designing antennas for the radio. Please note that the Si4735 shield is based on the design given in section 10 "Whip Antenna for SW Receive on AMI" (page 46 in revision 0.6.)

Using the SparkFun Si4735 Arduino Shield

The SparkFun Si4735 Arduino Shield requires modification to work correctly:

- There is no voltage level shifting on Si4735's GPO1 (MISO) and GPO2 (INT) outputs. The Si4735 must run at 3.3 V while most Arduinos run at 5 V. The shield converts all Arduino outputs to 3.3 V logic levels but does nothing to the Si4735 outputs. This is a problem because, when running at 5 V, an input must reach 3 V for the AVR to treat it as high (V_{IH}):

  $$V_{CC} \times .6 = 5V \times .6 = 3V$$

  However, the minimum guaranteed output voltage for a high signal from the Si4735 radio chip is (V_{OH}):

  $$V_{IO} \times .8 = 3.3V \times .8 = 2.64V$$

  Because the radio chip's output for a high signal can be less than the 3 V required by the AVR, either some form of level shifting from 3.3 V to 5 V must be added or an Arduino running at 3.3 V must be used. 3.3 V Arduinos include the Due, the 3.3 V versions of the Pro and Pro Mini from SparkFun, and the Seeeduino set to run at 3.3 V.

  See section "Level shifting between 5 and 3.3 volts" below for more information.

- The AREF pin is connected to ground on the shield's printed circuit board. AREF should never be grounded as this makes using the AVR's analog to digital converter impossible. Fixing this is optional if you do not need analog to digital conversion. However, you never know what changes you will make to your radio project in the future.

  To fix this, you need to break the connection between the Arduino's AREF input and ground. Probably the best solution is to cut the eight tiny traces between the pad and ground plane with a sharp knife, X-Acto blade, or razor blade. There are four traces on each side of the circuit board. Do this before soldering a connector to the shield.

  Another option is to use a connector that does not connect the shield's AREF pad to the Arduino's AREF input.

- The RF/Antenna section is based on application note AN383, section 10 "Whip Antenna for SW Receive on AMI," page 46 in revision 0.6. However, there is one small difference: The application note shows a connection between the other pin of the double throw switch and the AM loop antenna. This shorts out the 33 pF capacitor when the switch is in AM mode. SparkFun omitted this connection. Fixing this mistake is only needed for AM Medium Wave (common AM service) and Long Wave reception. However, if you do fix it, it should improve AM Medium Wave reception. On my board, RSSI (signal strength) went up by 5.

  Note: The switch is mislabeled by SparkFun as "AM/LW" and "FM/SW." It should be labeled as "FM/AM/LW" and "SW."

Using the SparkFun Si4735 shield with various Arduinos

The SparkFun Si4735 shield only supports the "original" Arduinos usually based on the ATMega328P including the 2009 (Duemilanove), Uno, and their clones. The problem is that the Mega, Leonardo, and Due have their SPI port located on different pins than the shield. As a result, you cannot plug the shield into these Arduinos. Instead, you must use wires and connectors to build a custom cable between these Arduinos and the shield.

The pin assignments for the SPI port can be found in Table 1.

Name	Shield Pin	Mega Pin	Leonardo Pin	Due Pin
MOSI	11	51	16	75
MISO	12	50	14	74
SCK	13	52	15	76

Table 1: Pin assignments for the SPI port

On the Leonardo and Due, the MISO, SCK, and MOSI pins are only available on the ICSP connector. See: http://arduino.cc/en/Reference/SPI.

Adapting the library to use the SparkFun Si4735 breakout board

This library can be adapted to make use of all of the Si4735 breakout board's features by passing arguments to begin() and setMode() and occasionally by manually editing the top part of the Si4735.h file.

Selecting SPI or I2C

Unlike the Si4735 shield, the Si4735 breakout board can use either the SPI or I2C bus. By default, the library uses SPI. You can switch to I2C by commenting out the "`#define Si47xx_SPI`" line by adding "`//`" to the beginning like this:

```
//#define Si47xx_SPI
```

Note: For your convenience, there are two versions of this library available for download. One for SPI users and one for I2C users. The only difference is the defining of the Si47xx_SPI macro and which bus library the example programs include.

The Si4735 has very specific power up requirements which the library will handle for you by controlling the radio's power and reset pins. By default, power is controlled by Arduino pin 8 and reset by Arduino pin 9.

When powering up, the radio must be told to use SPI or I2C. No action is required to select I2C. To select SPI, the radio's interrupt (GPO2) pin must be driven high. This should be done with a 10 kΩ pull-up resistor connected between the radio's interrupt pin and 3.3 V. (Note that this is how the Si4735 shield selects SPI.)

Warning: I2C support requires version 1.0.0 or above of the Arduino development software.

Configuring SPI

The library defaults to Arduino pin 10 as the Slave Select (SS) for the radio's SPI port on all Arduinos. This assignment is arbitrary and can be changed in Si4735.h by breakout board users.

Configuring interrupts

By default, the library *usually* uses Arduino pin 2 as the interrupt input for compatibility with the Si4735 shield. However, you may change this by editing RADIO_INT_PIN and RADIO_EXT_INT in Si4735.h.

RADIO_INT_PIN gives the Arduino pin number to use and RADIO_EXT_INT gives the corresponding AVR interrupt number. When using an AVR based Arduino, if you change one of these, you ***must*** change the other so that it matches Table 2 below!

Note: Pins 2 and 3 are used by the I2C port on the Leonardo. Because of this, when I2C is selected on the Leonardo, the interrupt input is moved to pin 7.

Please note that ARM based Arduinos such as the Due do not use RADIO_EXT_INT. Also, they can use any pin as the interrupt input.

Each type of AVR based Arduino has its interrupt inputs on different pins. Table 2 gives the Arduino pin number (RADIO_INT_PIN) and its corresponding AVR interrupt number (RADIO_EXT_INT) for each type of Arduino board.

AVR based Arduino	Arduino pin RADIO_INT_PIN	AVR interrupt RADIO_EXT_INT
"Original,"	2	0
Uno, etc.	3	1
Mega,	21	0
Mega 2560	20	1
	19	2
	18	3
	2	4
	3	5
Leonardo	3	0
	2	1
	0	2
	1	3
	7	6

Table 2: Mapping from AVR Interrupt Pin to Interrupt Number

Warning: The AVR's hardware interrupt numbers for the Mega and Mega 2560 do **not** match the interrupt numbers used when calling attachInterrupt()! Use the hardware interrupt numbers from the above table, not the interrupt numbers used with attachInterrupt()!

Please note that pin 7 on the Leonardo is a valid interrupt input, despite the fact that attachInterrupt() does not currently support it.

Configuring the radio's clock input

If MODE_OPT_NO_XTAL is passed to setMode(), the library will set XOSCEN to false in ARG1 of the POWER_UP command. Otherwise, XOSCEN will be true in ARG1 of the POWER_UP command.

MODE_OPT_NO_XTAL should *not* be passed to setMode() when a 32768 Hz crystal is attached to the radio's RCLK and GPO3 pins. (This is how the Si4735 shield and the Si4707 breakout board are setup. Therefore, users of these boards should call setMode() without MODE_OPT_NO_XTAL.) Other methods of supplying the radio with a clock signal should call setMode() with MODE_OPT_NO_XTAL. (For example, setMode(FM, MODE_OPT_NO_XTAL).) See the POWER_UP command in the "Si47xx Programming Guide" for more information on the XOSCEN argument.

If you want digital audio output on the Si4735, you must generate a suitable clock signal for the radio and send it to the radio's RCLK or DCLK (GPO3) pin. You must also enable digital audio output by specifying the correct audio mode when calling setMode(). See section 5.15 "Reference Clock" in the Si4735 data sheet and the REFCLK_FREQ and REFCLK_PRESCALE properties in the "Si47xx Programming Guide."

Level shifting between 5 and 3.3 volts

The Si4735 must run at 3.3 V or less. Most Arduinos run at 5 V. When using these Arduinos, level shifting is required. Level shifters work in either one direction (unidirectional) or in both directions (bidirectional). I2C requires a bidirectional shifter with open drain (open collector) outputs while SPI and the other I/O signals only require a unidirectional shifter.

Warning: If the Arduino and radio chip will be sharing the SPI bus with other chips, then the radio's MISO pin (labeled GPO1 in the data sheet) requires a level shifter with a 3-state output.

Caution: All unused inputs on any chip should be connected to something to keep the chip stable and reduce power consumption. The three easiest ways to do this are to connect an unused input to either ground, the chip's power pin (labeled V_{DD} or V_{CC}), or a pull-up resistor also connected to the chip's power pin.

Unidirectional level shifting from 5 to 3.3 volts

This converts a high voltage (5 V) output from the Arduino to a low voltage (3.3 V) for the radio's input by using chips with high voltage tolerant inputs.

- A simple and reliable high to low shifter is the 74HC4050 buffer gate. This is the shifter found on the SparkFun Si4735 shield. It can shift at least a 15 V signal to as little as 3 V.

  To use this chip with the radio, connect the 74HC4050's power pin (labeled V_{DD} or V_{CC}) to 3.3 V and the GND pin to ground. Then connect an Arduino output to a buffer gate's input. Finally, connect the gate's output to the radio's input.

- A newer chip that can do high to low level shifting is the SN74LVC244 3-state buffer gate, which can shift from a maximum of 5.5 V down to as little as 1.65 V. As of this writing, this chip is available from Texas Instruments in DIP form.

  To use this chip with the radio, connect the 74LVC244's power pin (labeled V_{DD} or V_{CC}) to 3.3 V and the GND pin to ground. Then connect an Arduino output to a buffer gate's input. Finally, connect the gate's output to the radio's input. To make the 74LVC244 work, you must also enable its outputs by grounding its two Output Enable (1OE and 2OE) pins.

Unidirectional level shifting from 3.3 to 5 volts

This converts a low voltage (3.3 V) output from the radio to a high voltage (5 V) for the Arduino's input.

DANGER: Other people's versions of this library (including the version from SparkFun) always seem to drive the INT (GPO2) and MISO (GPO1) pins high to select SPI mode. However, the data sheet clearly says you do not need to drive MISO high to select SPI. Also, driving INT high is unnecessary with the Si4735 shield because it includes a pull-up resistor on the radio's INT pin. This version of the library does not drive the INT or MISO pins high. If you use a unidirectional level shifter, then you **MUST** either use this version of the library or modify the other version's source code to prevent driving these pins high. Otherwise, you will be temporarily connecting two outputs, which **WILL** eventually damage your hardware!

- Many years ago, it was common to mix CMOS and TTL families of chips. (Today, we only use CMOS.) Because these two families are incompatible, special versions of CMOS chips were made with TTL compatible inputs. The two most important families with this feature are named HCT and AHCT. Notice the "T" at the end of the names! The chips in these families accept an input of 2 V through 5 V as a valid high signal. As a result, these chips are ideal for level shifting from 3.3 V to 5 V.

  Caution: Because the HCT and AHCT chips must have 5 V applied to their power pin (labeled V_{DD} or V_{CC}), these chips can only be used when level shifting to 5 V.

  As mention above, the radio's MISO pin should have a level shifter with a 3-state output. A chip with that ability is the 74AHCT125 3-state buffer gate chip. It contains four buffer gates, each with an independent active low Output Enable (OE) pin to control the 3-state capability. (Note: You may substitute a 74HCT125 chip if you want.)

  To use the 74AHCT125, connect the chip's power pin (labeled V_{DD} or V_{CC}) to 5 V and GND pin to ground. Then connect as follows:

To level shift the radio's MISO (GPO1) pin, connect it to the input of a buffer gate on the 74AHCT125. Then connect the gate's output to the Arduino's MISO input. Finally, connect the buffer gate's Output Enable pin directly to the Arduino's Slave Select (SS) pin. (Arduino pin 10 by default. See section "Configuring SPI" to change this.)

To level shift the radio's interrupt (GPO2) pin, connect it to the input of a buffer gate on the 74AHCT125. Then connect the gate's output to the Arduino's interrupt input. (Usually Arduino pin 2. See section "Configuring interrupts" for more information.) Finally, connect the buffer gate's Output Enable pin to ground to keep the output "on" at all times.

Because chips in the AHCT family are a recent design, they are robust and can operate at very high speeds. (About 100 MHz for the 74AHCT125.) As a result, I recommend using a 74AHCT125 chip when unidirectional level shifting from 3.3 to 5 volts.

- A robust low to high voltage level shifter is a gate with an open drain output such as the 74HC266 exclusive-nor gate. To use this chip for level shifting, do the following: The gate chip's power pin (labeled V_{DD} or V_{CC}) is connected to 3.3 V and the GND pin is connected to ground. One of the gate's inputs is connected to 3.3 V and the other input is connected to an Si4735 output that requires level shifting. Then the gate's output is connected to the Arduino's input. Finally, each gate output used requires a separate 4.7 kΩ pull-up resistor connected to 5 V.

  The schematic in Figure 1 summarizes how to convert each signal.

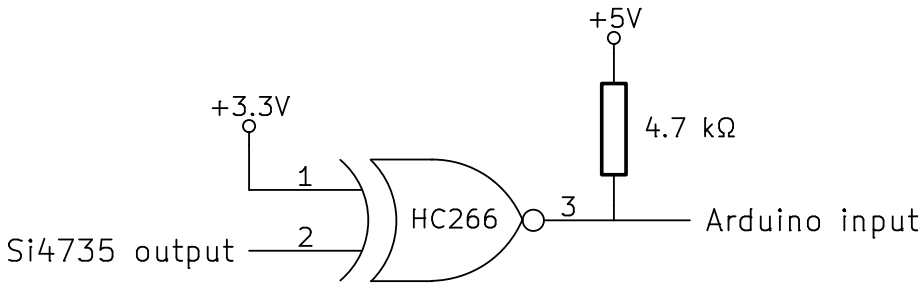

Figure 1: Open drain level shifter

  Because pull-up resistors are slow to bring the signal high, this level shifter only operates at slower speeds. I conservatively estimate it can handle 1 MHz or less.

- The MC14504B or CD4504B chip is a robust, low to high level shifter. No pull-up resistor is required. Because this chip is a very old design, this level shifter only operates at slower speeds. I conservatively estimate it can handle 1 MHz or less.

- The CD40109B chip is a robust, low to high level shifter that is very similar to the MC14504B, however, the CD40109B is slower.

Bidirectional level shifting between 3.3 and 5 volts

This converts between a low voltage input or output on the radio and a high voltage input or output on the Arduino. The level shifting circuit automatically detects if the two sides change direction. All of these options can be used with either SPI, I2C, or any other I/O signal, unless otherwise specified.

Please note that some of these options have 3-state outputs controlled by an active *high* Output Enable pin. However, SPI's MISO signal requires a 3-state output controlled by an active *low* Output Enable pin connected to Slave Select (SS). Because none of these options support this, they cannot easily be used to level shift SPI's MISO signal, if the Arduino and radio are sharing the SPI bus with other devices.

- The TXB0108 chip is designed as a robust general purpose level shifter. Adafruit sells a breakout board[1] with this chip. Because this chip is a recent design, it can operate at very high speeds (20

[1] http://www.adafruit.com/products/395

MHz). Note that this chip will not work if either side has a pull-up resistor less than 50 kΩ. Because I2C requires a stronger pull-up resistor, typically 10 kΩ, and because I2C requires a level shifter with open drain outputs (which this chip does not do), this level shifter cannot be used with I2C.

- The TXS0102 chip is designed as a robust I2C level shifter. Unfortunately, no one is currently selling a breakout board with this chip.
- It is possible to build a bidirectional level shifter with a transistor and two pull-up resistors. A version of this circuit is available from SparkFun as part BOB-8745[2] and from Adafruit[3]. I personally used this method while developing my software. This shifter is specifically intended for use with I2C and has open drain (open collector) outputs. (It does not have 3-state outputs.) It should be fine for SPI provided you keep the clock speed at around 400 kHz, the same speed I2C normally uses.
- The PCA9306 chip is designed as a robust I2C level shifter. SparkFun sells breakout board BOB-10403[4] with this chip. Caution: Check comments for BOB-10403 on the SparkFun web site. Some people had problems with this board.

A terrible level shifter

In Internet posts, some people have suggested using a diode in series to level shift from 3.3 V to 5 V. I checked this idea with my brother who is a professional electrical engineer who designs digital ICs for a living. He said this *might* work **sometimes** for the SPI data pin (MISO) but will **never** work for the interrupt pin. He said the technique will only work if the signal changes state between high and low very rapidly. Depending on the data sent over the SPI bus, this may or may not happen. However, because the interrupt line remains static most of the time, the diode cannot raise the voltage of the radio's interrupt signal.

Reliability of level shifted signals

When using the Si4735 shield with a 5 V Arduino, it is important that reliable level shifting is used on the MISO SPI pin. When an interrupt is received, the library sends a GET_INT_STATUS command and then reads the status byte it returns. If the MISO signal is unreliable, the interrupt code received by the Arduino could be corrupted. This could cause the library to falsely believe that an interrupt it was waiting for has not yet arrived.

This problem will be most noticeable, if the STC (Seek/Tune Complete) interrupt is missed after changing the radio's frequency. In this case, the library will wait forever for the STC interrupt, despite the fact the radio has sent it. At this point, the only solution is manually pressing the Arduino's reset button.

Another common symptom of an unreliable MISO connection is `getFrequency()` or `checkFrequency()` returning a bad frequency.

To help prevent this problem, I make the following recommendations:

- If possible, use a 3.3 V Arduino so no level shifting is required.
- If you must use a 5 V Arduino, use a robust level shifter.
- Keep the SPI bus frequency low. The library's default frequency is very slow.
- Do not use a bread board with the level shifter. All those metal strips inside the bread board pickup stray RF noise and also slow down signals with stray capacitance.
- Keep any connecting wires as short as possible.

[2] http://www.sparkfun.com/products/8745
[3] http://www.adafruit.com/products/757
[4] http://www.sparkfun.com/products/10403

- Keep the radio's hardware away from your computer's fan and ventilation holes and any other sources of RF noise. While testing the library, I once placed the radio on top of a ventilation hole in my computer's case. The RF noise pouring out of this hole caused my radio to completely malfunction. Once I moved the radio a few inches away, the radio began to work.

- Solder the connections, if possible. If you are having problems with a signal and it's not soldered, try disconnecting and then reconnecting its wires or try replacing its wires.

- Wrap the final hardware design in sheet metal to protect it from stray RF noise. However, make sure that any antennas are outside this shielding or you will not receive any stations.

Initializing the radio and library and setting the receive band

You must call begin() to initialize this library before calling any other methods. When called, it initializes the SPI or I2C bus. Because you may wish to do this yourself or with another library, you may tell begin() to skip initializing these buses by calling begin(BEGIN_DO_NOT_INIT_BUS).

The second argument to begin() is the bus argument. For SPI, it gives the clock divider to pass to SPIClass :: setClockDivider(). If no bus argument is given to begin(), RADIO_SPI_CLOCK_DIV in Si4735.h is used.

Example code to set the SPI bus clock to divide by 8. On a 16 MHz Arduino, this will set the SPI clock to $16 \div 8 = 2$ MHz:

```
begin(BEGIN_DEFAULT, SPI_CLOCK_DIV8);
```

For I2C, the bus argument gives the address of the radio. The radio will respond to one of two I2C addresses. The address used by the radio is selected by the signal given to the radio's SEN/SS pin. The following table gives the possible addresses:

I2C bus argument for begin()	SEN pin
RADIO_I2C_ADDRESS_HIGH	HIGH
RADIO_I2C_ADDRESS_LOW	LOW

If no bus argument is given, RADIO_I2C_ADDRESS in Si4735.h is used. By default, RADIO_I2C_ADDRESS equals RADIO_I2C_ADDRESS_HIGH. Note that the Si4707 breakout board has a 10 kΩ pull-up resistor connected to SEN to make it go high.

Example code to set the I2C address to its low value:

```
begin(BEGIN_DEFAULT, RADIO_I2C_ADDRESS_LOW);
```

The radio has a low power "off" mode and modes for each receive band. After calling begin(), the radio is in the low power RADIO_OFF mode. You change the radio's mode by calling setMode(). For example, calling setMode(FM) powers up the radio and sets the receive mode to the FM band. See Si4735.h for a list of all supported modes.

The radio's current mode is returned by getMode().

By calling setMode(RADIO_OFF), the radio goes back into low power mode. To kill all power to the radio, call end(). To restore power later, call begin().

Warning: Unlike previous versions of this library from other people, begin() does not set the receive band. Call setMode() to do this. As a result, the following code fragment:

```
radio.begin(FM);
```

should be changed to:

```
radio.begin();
radio.setMode(FM);
```

Open the Changes.text file in a programmer's text editor to see why this change was made.

setMode() sets up the given band's top and bottom frequency and its spacing based on the previously set locale. (See section "Localization.") You may override these band settings by calling setBandTop(), setBandBottom(), and setSpacing() after calling setMode(). To get the current band settings, call getBandTop(), getBandBottom(), and getSpacing().

There are a small number of differences between the Si4735-C40 and Si4735-D60. As a result, setMode() sends the GET_REV command to the radio and saves the results in the radio.revision structure. This permits your software and this library to determine which chip is being used and adapt themselves to it. For a list of differences and compatibly concerns, see Appendix A and B in the "Si47xx Programming Guide." (Please note that the audio noise bug in FM receive mode documented in Appendix B is corrected by this library.)

Using the library with or without interrupts

Previous versions of this library from other people did not support receiving interrupt signals from the radio chip. Instead, the library and application would poll the radio to see if some event had occurred yet. This approach causes a large amount of bus traffic between the Arduino and radio chips. This is undesirable because bus traffic can introduce noise into the radio chip which can be heard in the audio output. Also, the manuals say that bus traffic can harm tuning operations when using an external crystal connected to the RCLK and GPO3 pins with the radio chip's internal oscillator circuit. (Please note that the Si4735 shield uses this setup, and therefore, shield users should avoid any bus traffic while tuning a new station.)

This library supports waiting for an interrupt signal from the radio chip before asking it what event has taken place. Or, you can choose to poll the radio at any time.

The library automatically tells the radio to send STC (Seek/Tune Complete), RDS (Radio Data System), and RSQ (Received Signal Quality) interrupts.

To keep the design simple, the library does not use an interrupt handler. Instead we just poll the AVR's interrupt registers to see when an interrupt pulse has been received from the radio chip. On ARM based Arduinos, a short interrupt handler sets a flag variable and then returns. Later, the library polls this flag variable to check for interrupts.

Tuning

To change to a particular frequency, call tuneFrequency(). To increment or decrement the current frequency, call frequencyUp() or frequencyDown().

To seek to the next or previous frequency, call seekUp() or seekDown(). To cancel a seek operation call cancelSeek().

To manually ask the radio if a seek or tune operation has completed, call getFrequency(). It immediately asks the radio if tuning is complete and, if so, its current frequency. checkFrequency() works the same but only asks the radio if the STC interrupt has been received.

currentFrequency() returns the radio's current frequency or 0 if unknown. (The frequency is usually unknown because a seek operation is underway.) Note: currentFrequency() does not indicate if the previous

tune or seek operation has completed. It only returns the frequency from the last tune operation or call to getFrequency() or checkFrequency().

Calling waitSTC() will not return until the STC interrupt has been received. This is useful after calling tuneFrequency(), frequencyUp(), frequencyDown(), seekUp(), or seekDown().

Methods tuneFrequencyAndWait(), frequencyUpAndWait(), and frequencyDownAndWait() automatically call waitSTC() after changing the frequency.

Note: Seeking on some bands (like SW) can take a long time to find a station. Therefore, it is recommended that the application using this library handle seeks in the following way. First, call seekUp() or seekDown(). Second, enter a loop which looks for two events: either the user wants to cancel the seek or the STC interrupt has been received because the seek has finished. The following incomplete code fragment shows how to do this:

```
radio.seekUp();
word station;
bool user_cancel_seek=false;
do{
   //add code here to see if user wants to cancel seek

   if(user_cancel_seek == true){
      station = radio.cancelSeek();
      break;
   }
   //check if seek finished, that is, STC interrupt received
   station = radio.checkFrequency();
}while(station == 0);
```

RDS

There are two methods for reading RDS data, getRDS() and checkRDS(). getRDS() immediately asks the radio to send any RDS data by sending an FM_RDS_STATUS command. checkRDS() first checks if an RDS interrupt has been received. If so, checkRDS() calls getRDS(). Otherwise checkRDS() returns immediately. RDS data is stored inside the rds structure, located in the public part of the Si4735 class structure.

getCallSign() translates an RBDS (USA) PI code into the station's call letters. Note, this works for US stations because their PI codes are based on their call letters. In other countries, PI codes are arbitrarily assigned. We would need to create a large database matching PI codes to station IDs.

getProgramTypeStr() translates the ProgramType code into an English message. Works for both RDS and RBDS.

getLocalTime() and getLocalDateTime() return the local time returned by the station.

RSQ

There are two methods for reading RSQ information, getRSQ() and checkRSQ(). getRSQ() immediately asks the radio for the current RSQ information by sending an RSQ_STATUS command. checkRSQ() first checks if an RSQ interrupt has been received. If so, checkRSQ() calls getRSQ(). Otherwise checkRSQ() returns immediately.

The RSQ interrupt is sent by the radio when certain thresholds have occurred. These thresholds are configured by setting properties with the setProperty() method. For FM, these properties include:

```
PROP_FM_RSQ_INT_SOURCE
PROP_FM_RSQ_SNR_HI_THRESHOLD
PROP_FM_RSQ_SNR_LO_THRESHOLD
```

```
PROP_FM_RSQ_RSSI_HI_THRESHOLD
PROP_FM_RSQ_RSSI_LO_THRESHOLD
PROP_FM_RSQ_MULTIPATH_HI_THRESHOLD (Not on Si4735-C40)
PROP_FM_RSQ_MULTIPATH_LO_THRESHOLD (Not on Si4735-C40)
PROP_FM_RSQ_BLEND_THRESHOLD
```

AM has similar properties. Not all properties are supported by all chips. See the "Si47xx Programming Guide" for more information on these properties.

Manually reading interrupts

You may manually ask the radio what interrupts have been sent by calling getInterrupts(). This sends a GET_INT_STATUS command to the radio, saves the result internally, and then returns the result.

Also, you may call currentInterrupts() to ask the library what interrupts have been received. If a new interrupt signal has been received by the Arduino, currentInterrupts() calls getInterrupts(). If no new interrupt signal has been received, currentInterrupts() returns the previously saved interrupt status.

Localization

Localization is done in two parts: an ITU region and a locale within that region. Both are set by calling setRegionAndLocale().

The ITU (International Telecommunication Union) has divided the world into 3 regions. For the most part, radio broadcasting is the same throughout each region. In some cases, the library subdivides regions into subregions. To handle exceptions within each region or subregion, you may also set a locale.

For example, the USA is in region 2, subregion North America. However, unlike other countries, it uses RBDS instead of RDS. Therefore, to setup the library for the USA:

```
radio.setRegionAndLocale(REGION_2_NA, LOCALE_US);
```

Please note that the library defaults to the USA.

For locations that do not need localization beyond specifying their region and subregion, a special locale code is used: LOCALE_OTHER. For example, most of Europe should work by calling:

```
radio.setRegionAndLocale(REGION_1, LOCALE_OTHER);
```

setRegionAndLocale() should only be called while the radio is in the low power RADIO_OFF mode.

getRegion() returns the current region and subregion code. getLocale() returns the current locale code for the current region and subregion.

See the Si4735.h file for the list of region and locale constants and more information on their use.

Volume

The volume can be changed by calling volumeUp() or volumeDown(). Both methods take an argument giving the amount to increase or decrease the volume. Also, you can set the volume to a given value by calling setVolume(). All three methods return the new volume setting. To read the current volume setting at any time, call getVolume().

mute() and unmute() activate or deactivate muting. You can also toggle mute by calling toggleMute(). It then returns the new mute status. getMute() returns true if mute is currently active.

The volume and mute methods can be called while the radio is powered up in a receive mode or the radio is in the low power RADIO_OFF mode.

Sending commands to the radio

The most useful commands already have library methods. However, you may choose to manually send commands to the radio by calling sendCommand() or sendCommand_P(). To get the results, call getStatus() for the single byte status code or getResponse() for the longer 16 byte response. For your convenience, predefined command codes can be found in Si4735.h.

Many features are controlled with properties. Call setProperty() to set a property and getProperty() to read its current value. For your convenience, predefined property codes can be found in Si4735.h.