AdafruitSI4713 Library Documentation

Release 1.0

Tony DiCola

Feb 20, 2018

Contents

CircuitPython module for SI4713 FM RDS transmitter.

CHAPTER 1

Dependencies

This driver depends on:

- Adafruit CircuitPython
- Bus Device

Please ensure all dependencies are available on the CircuitPython filesystem. This is easily achieved by downloading the Adafruit library and driver bundle.

CHAPTER 2

Usage Example

See examples/simpletest.py for a demo of the usage.

CHAPTER 3

Contributing

Contributions are welcome! Please read our Code of Conduct before contributing to help this project stay welcoming.

CHAPTER 4

Building locally

4.1 Zip release files

To build this library locally you'll need to install the circuitpython-build-tools package.

```
python3 -m venv .env
source .env/bin/activate
pip install circuitpython-build-tools
```

Once installed, make sure you are in the virtual environment:

```
source .env/bin/activate
```

Then run the build:

```
circuitpython-build-bundles --filename_prefix adafruit-circuitpython-si4713 --library_
↪location .
```

4.2 Sphinx documentation

Sphinx is used to build the documentation based on rST files and comments in the code. First, install dependencies (feel free to reuse the virtual environment from above):

```
python3 -m venv .env
source .env/bin/activate
pip install Sphinx sphinx-rtd-theme
```

Now, once you have the virtual environment activated:

```
cd docs
sphinx-build -E -W -b html . _build/html
```

This will output the documentation to `docs/_build/html`. Open the index.html in your browser to view them. It will also (due to -W) error out on any warning like Travis will. This is a good way to locally verify it will pass.

CHAPTER 5

Table of Contents

5.1 Simple test

Ensure your device works with this simple test.

Listing 5.1: examples/simpletest.py

```python
# Simple demo of using the SI4743 RDS FM transmitter.
# Author: Tony DiCola
import time

import board
import busio

import adafruit_si4713


# Specify the FM frequency to transmit on in kilohertz.  As the datasheet
# mentions you can only specify 50khz steps!
FREQUENCY_KHZ = 102300  # 102.300mhz


# Initialize I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)

# Initialize SI4713.
si4713 = adafruit_si4713.SI4713(i2c)
# Alternatively you can specify the I2C address of the device if it changed:
#si4713 = adafruit_si4713.SI4713(i2c, address=0x11)
# Also if you hooked up the reset line you can specify that too.  Make sure
# to pass in a DigitalInOut instance:
#import digitalio
#reset = digitalio.DigitalInOut(board.D5)
#si4713 = adafruit_si4713.SI4713(i2c, reset=reset)

```

```
# Measure the noise level for the transmit frequency (this assumes automatic
# antenna capacitance setting, but see below to adjust to a specific value).
noise = si4713.received_noise_level(FREQUENCY_KHZ)
# Alternatively measure with a specific frequency and antenna capacitance.
# This is not common but you can specify antenna capacitance as a value in pF
# from 0.25 to 47.75 (will use 0.25 steps internally).  If you aren't sure
# about this value, stick with the default automatic capacitance above!
#noise = si4713.received_noise_level(FREQUENCY_KHZ, 0.25)
print('Noise at {0:0.3f} mhz: {1} dBuV'.format(FREQUENCY_KHZ/1000.0, noise))

# Tune to transmit with 115 dBuV power (max) and automatic antenna tuning
# capacitance (default, what you probably want).
si4713.tx_frequency_khz = FREQUENCY_KHZ
si4713.tx_power = 115

# Configure RDS broadcast with program ID 0xADAF (a 16-bit value you specify).
# You can also set the broadcast station name (up to 96 bytes long) and
# broadcast buffer/song information (up to 106 bytes long).  Setting these is
# optional and you can later update them by setting the rds_station and
# rds_buffer property respectively.  Be sure to explicitly specify station
# and buffer as byte strings so the character encoding is clear.
si4713.configure_rds(0xADAF, station=b"AdaRadio", rds_buffer=b"Adafruit g0th Radio!")

# Print out some transmitter state:
print('Transmitting at {0:0.3f} mhz'.format(si4713.tx_frequency_khz/1000.0))
print('Transmitter power: {0} dBuV'.format(si4713.tx_power))
print('Transmitter antenna capacitance: {0:0.2} pF'.format(si4713.tx_antenna_
↪capacitance))

# Set GPIO1 and GPIO2 to actively driven outputs.
si4713.gpio_control(gpio1=True, gpio2=True)

# Main loop will print input audio level and state and blink the GPIOs.
print('Broadcasting...')
while True:
    # Print input audio level and state.
    print('Input level: {0} dBfs'.format(si4713.input_level))
    print('ASQ status: 0x{0:02x}'.format(si4713.audio_signal_status))
    # 'Blink' GPIO1 and GPIO2 alternatively on and off.
    si4713.gpio_set(gpio1=True, gpio2=False)  # GPIO1 high, GPIO2 low
    time.sleep(0.5)
    si4713.gpio_set(gpio1=False, gpio2=True)  # GPIO1 low, GPIO2 high
    time.sleep(0.5)
```

5.2 adafruit_si4713

CircuitPython module for the SI4713 RDS FM transmitter. See examples/simpletest.py for a demo of the usage. Based on the Arduino library at: https://github.com/adafruit/Adafruit-Si4713-Library/

- Author(s): Tony DiCola

class adafruit_si4713.**SI4713**(*i2c*, ***, *address=<sphinx.ext.autodoc._MockObject object>*, *reset=None*, *timeout_s=0.1*)

SI4713 RDS FM transmitter. Initialize by specifying:

- i2c: The I2C bus connected to the board.

Optionally specify:

- address: The I2C address if it has been changed.
- **reset: A DigitalInOut instance connected to the board's reset line,** this will be used to perform a soft reset when necessary.
- **timeout_s: The amount of time (in seconds) to wait for a command to** succeed. If this timeout is exceed a runtime error is thrown.

`audio_signal_status`
Retrieve the ASQ or audio signal quality status value from the chip. This is a byte that indicates if the transmitted input audio signal is overmodulating (too high) or above/below input audio level thresholds. See page 25 of AN332 for more discussion of this value: https://www.silabs.com/documents/public/application-notes/AN332.pdf

`configure_rds` (*program_id*, *station=None*, *rds_buffer=None*)
Configure and enable the RDS broadcast of the specified program ID. Program ID must be a 16-bit value that will be broacast on the RDS bands of the transmitter. Specify optional station and RDS buffer strings that will be used to broadcast the station and currently playing song information, or later set the rds_station and rds_buffer to change these values too. The station value is up to 96 bytes long, and the buffer is up to 106 bytes long. Note this will configure RDS properties of the chip for a typical North American RDS broadcast (deviation, mix, repeat, etc. parameters).

`gpio_control` (*gpio1=False*, *gpio2=False*, *gpio3=False*)
Control the GPIO outputs of the chip. Each gpio1, gpio2, gpio3 parameter is a boolean that indicates if that GPIO channel (corresponding to GPIO1, GPIO2, GPIO3 of the chip respectively) is driven actively (True) or is high-impedence/off (False). By default any unspecified GPIO is set to high-impedence/off unless otherwise provided.

`gpio_set` (*gpio1=False*, *gpio2=False*, *gpio3=False*)
Drive the GPIO outputs of the chip that are enabled with active output. Each gpio1, gpio2, gpio3 parameter is a boolean that indicates if the associated GPIO (corresponding to GPIO1, GPIO2, GPIO3 of the chip respectively) is driven high (True) or low (False). By default all GPIO are assumed to be set low (False) unless otherwise specified. Note that you must first set GPIOs to active output with the gpio_control function to see their output physically change.

`input_level`
Read the input level of audio to the chip and return it in dBfs units.

`interrupt_status`
Read the interrupt bit status of the chip. This will return a byte value with interrupt status bits as defined by the radio, see page 11 of the AN332 programming guide: https://www.silabs.com/documents/public/application-notes/AN332.pdf

`rds_buffer`
Set the RDS broadcast buffer to the specified byte string. Can be at most 106 bytes long and will be padded with blank spaces if less.

`rds_station`
Set the RDS broadcast station to the specified byte string. Can be at most 96 bytes long and will be padded with blank spaces if less.

`received_noise_level` (*frequency_khz*, *antenna_capacitance=0*)
Measure the received noise level for the specified frequency (in kilohertz, 76mhz - 108mhz and must be a multiple of 50) and return its value in units of dBuV (decibel microvolts). Will use automatic antenna capacitance tuning by default, otherwise specify an antenna capacitance in pF from 0.25 to 47.75 (only steps of 0.25pF are supported).

reset()
: Perform a reset of the chip using the reset line. Will also perform necessary chip power up procedures.

set_tx_power_capacitance(*tx_power*, *capacitance*)
: Set both the transmit power (in dBuV, from 88-115) and antenna capacitance of the transmitter. Capacitance is a value specified in pF from 0.25 to 47.75 (in 0.25 steps), or 0 to indicate automatic tuning. You typically don't need to use this function unless you want explicit control of tuning antenna capacitance, instead for simple transmit power changes use the tx_power property (which assumes automatic antenna capacitance).

tx_antenna_capacitance
: Read the transmit antenna capacitance in pico-Farads (pF). Use the set_tx_power_capacitance function to change this value (must also change transmit power at the same time). It's uncommon to adjust this beyond the automatic tuning option!

tx_frequency_khz
: Get and set the transmit frequency of the chip (in kilohertz). See AN332 page 19 for a discussion of the constraints on this value, in particular only a multiple of 50khz can be specified, and the value must be between 76 and 108mhz.

tx_power
: Get and set the transmit power in dBuV (decibel microvolts). Can be a value within the range of 88-115, or 0 to indicate transmission power is disabled. Setting this value assumes auto-tuning of antenna capacitance, see the set_tx_power_capacitance function for explicit control of setting both transmit power and capacitance if needed.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index