ZSSC3230 I2C Driver User Guide

Introduction

This document provides setup information and programming guidelines for interacting with the ZSSC3230 capacitive sensor signal conditioner IC by Renesas using the ZSSC3230_I2C_Driver Arduino library.

The ZSSC3230 is a low power capacitance to digital converter, with an I2C output. This makes it perfect for reading low capacitance, variable sensors in the range of 0pF to 30pF.

The I2C driver is written in the C language and can be adapted to several different microcontrollers. In this guide, we will use an Arduino to communicate with the ZSSC3230 IC.

This document will cover:

- How to setup the IC.
- How to take basic readings.
- How to configure the analogue to digital converter (ADC).
- The calibration process of the IC.
- How to take signal conditioned readings.
- How to use in high frequency applications.
- Some example applications.
- Public methods in the ZSSC3230_I2C_Driver Arduino Library

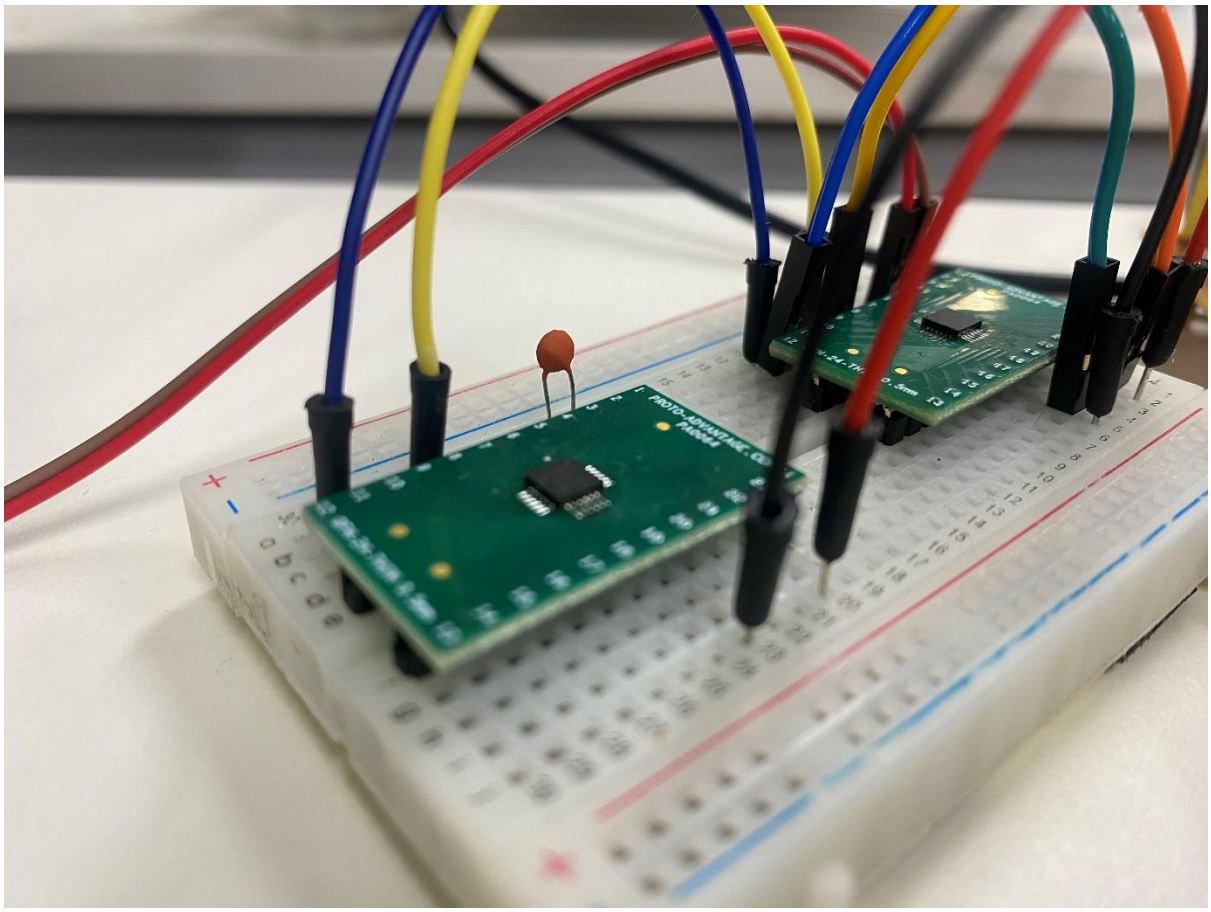
Written by Alex Dunn, 12 Feb, 2024

Table of Contents

1 General Information

This guide uses an Arduino Uno microcontroller and the Arduino IDE software to flash the microcontroller. It uses the ZSSC3230_I2C_Driver Arduino library available here: https://github.com/ad039/ZSSC3230_I2C_Driver

For easy development applications, the ZSSC3230 IC can placed on a surface mount adaptor, allowing it to be placed into a breadboard.



2 ZSSC3230 IC

The ZSSC3230 is a low power, CMOS integrated circuit for accurate capacitance to digital conversions. It incorporates sensor-specific correction of capacitive sensor signals, allowing each sensor to be individually calibrated by the user. The ZSSC3230 has an I2C interface to communicate with microcontrollers at a maximum I2C clock speed of 3.4MHz. It is configurable up to 30pF, with an output resolution scalable up to 18-bit.

For more in depth information on the ZSSC3230 chip itself please read through the datasheet: https://www.renesas.com/us/en/document/dst/zssc3230-datasheet?r=465426

The ZSSC3230 has two modes of reading a capacitive sensor:

- Differential Mode
- Single-Ended Mode

Differential mode connects both ends of the capacitor to the ZSSC3230 sensor, whilst single-ended mode connects one end of the capacitor to the ZSSC32320 and the other end to ground. For all the demonstrations within this guide, the ZSSC3230 will be configured in differential mode.

3 Wiring

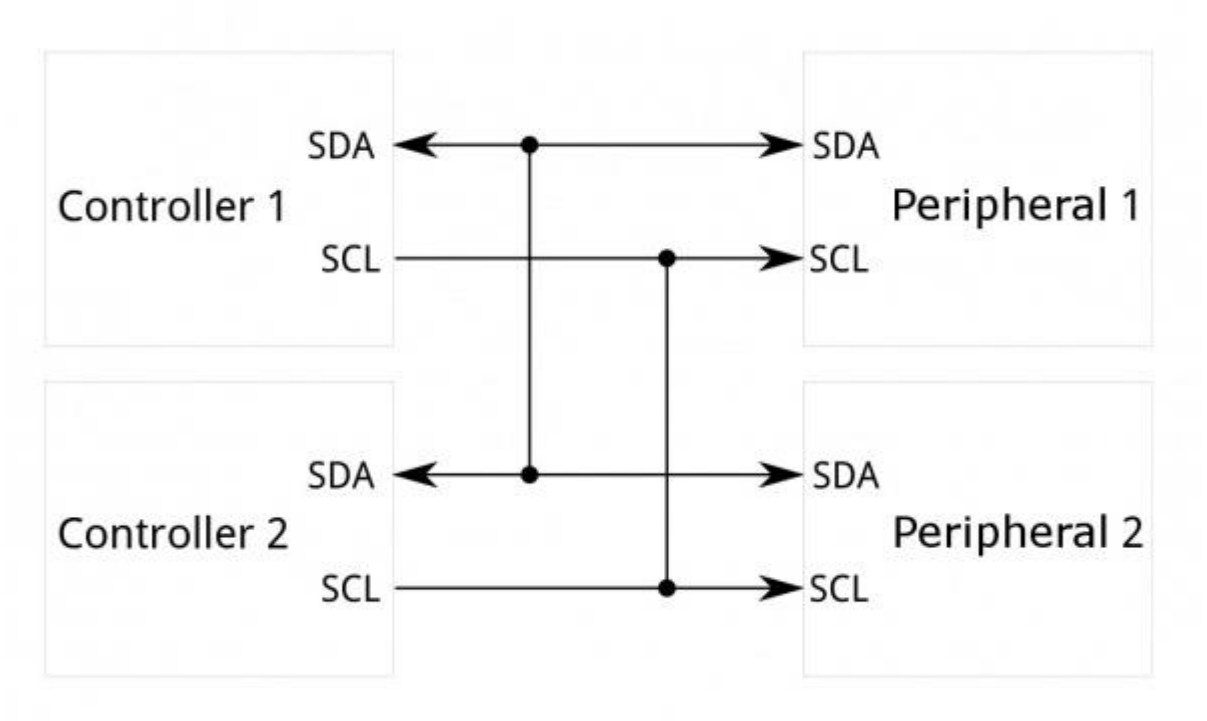
To communicate with the ZSSC32320 you will need to connect the following pins to your microcontroller of choice.

1. Connect pin 20 to ground
2. Connect pin 2 to one end of capacitor
3. Connect pin 3 to the other end of the capacitor (order does not matter)
4. Connect pin 11 to SDA
5. Connect pin 8 to SCL
6. Connect pin 23 to 3.3V

Ensure that there are pull-up resistors (about 4.7kOhm) between SDA and 3.3V, and SCL and 3.3V.

4 I2C Interface

To communicate with the ZSSC3230, the I2C protocol is used. I2C is designed to allow one or more "controller" devices to communicate with multiple "peripheral" devices using only four wires: ground, power, serial data (SDA) and serial clock (SCL). The communication takes place over the SDA and SCL wires, where SDA transfers the data and SCL controls the speed of the data transfer. Each peripheral device has a 10-bit or 7-bit address. The controller can ask to communicate with a peripheral by sending out this address first and then waits for the peripheral to acknowledge. For the ZSSC3230, only a 7-bit address is supported, which is standard for most sensors. This allows for 112 different sensors to be placed on the I2C bus. The default address is set to 0x48, so if you want to communicate with multiple sensors you will need to change the address of the sensor first. You can use the ZSSC3230_set_i2c_address example from the ZSSC3230_I2C_Driver Arduino library to change the address of your sensor.



The data rate of I2C devices is determined by the controller, with standard clock speeds defined as:

I2C Mode	Clock Speed
Standard	100kHz
Fast	400kHz
Fast-Plus	1MHz
High-Speed	3.4MHz
Ultra-Fast	5MHz

The ZSSC3230 sensor supports up to High-Speed Mode. For High-Speed Mode the I2C address should be set between $0x04_{HEX}$ to $0x07_{HEX}$. For all other modes (Standard, Fast and Fast-Plus) the I2C address can be set anywhere from 0x00 to 0x03 and 0x08 to 0x7F.

5 Begin Communication with the ZSSC3230 and Read the Raw ADC Capacitance.

To begin communicating with your ZSSC3230 sensor, please ensure that it is wired to your microcontroller as shown in Section 0. For this document we will be communicating with the sensor using an Arduino Uno microcontroller development board and the Arduino IDE. In this section, you will run through the initialisation of the ZSSC3230_I2C_Driver Arduino library and how to use it to communicate with your sensor.

5.1 ZSSC3230 Library Installation

To install and include the ZSSC3230_I2C_Driver library in your sketch, follow these steps:

1. Download the ZSSC3230_I2C_Driver Library .zip file from the link: https://github.com/ad039/ZSSC3230_I2C_Driver
2. Open the Arduino IDE
3. Open a new sketch by going to File->New
4. Go to the Sketch tab and select Include Library->Add .ZIP Library
5. In the explore window that pops up, locate the ZSSC3230_I2C_Driver.zip file, select it and select Open.
6. Include the library in your sketch by going to Sketch->Include Library-> ZSSC3230_I2C_Driver

5.2 Simple ZSSC3230 Setup and ADC Read Program

The following will run you through how to begin a basic communication with the ZSSC3230 and read the raw ADC capacitance. This program and other examples can be accessed from File->Examples-> ZSSC3230_I2C_Driver. The following program is the same as ZSSC3230_read_raw_capacitance.

Create a ZSSC3230 object at the top of your code. This will allow you to communicate with the ZSSC3230 sensor using the ZSSC3230 library's functions. You will also need to include the "`Wire.h`" library as seen in the figure below. This controls the I2C communication and gives us functions to easily read and write.

```
10  #include <ZSSC3230.h>
11  #include <Wire.h>
12
13  // create a ZSSC3230 object
14  ZSSC3230 zssc3230;
```

Inside `setup()`, you can initialise Serial communication using `Serial.begin(115200)`. This will allow you to write sensor values to the serial monitor.
You can also initialise the Wire library, which will allow you to communicate using I2C. This is a dependency for the ZSSC3230 library and must be included. To set the speed of the I2C clock you can use the function `Wire.setClock(clockFrequency)`, where `clockFrequency` is the frequency of the clock in Hertz.

```
16  void setup() {
17    // begin serial communication
18    Serial.begin(115200);
19    while (!Serial)
20      delay(10);
21
22    // begin wire communication
23    Wire.begin();
24
```

Next initialise the ZSSC3230 (still inside `setup()`). To do this you need to call the `zssc3230.begin()` function. This has a Boolean output so if the sensor can be communicated with, `.begin()` returns 1. If the sensor could not be found and setup, `.begin()` returns 0. This can be used in an if statement to print to the serial monitor if the sensor could be found or not. This is all included in a while loop so if the ZSSC3230 cannot be found, the Arduino will keep trying to connect.

```
// initialise the zssc3230 sensor
while(1) {
  if (zssc3230.begin()) {
  Serial.println("ZSSC3230 Detected");
  break;
  }
  else {
  Serial.println("ZSSC3230 Not Detected. Trying Again...'
  delay(500);
  }
}
```

Configure the sensor using the `.configure_sensor()` function.

```
// set sensor configuration to be differential mode,
zssc3230.configure_sensor(TYPE_DIFFERENTIAL, LEAKAGE...
```

Inside loop(), you can now read the raw ADC value by calling the function `.read_raw_cap()` and printing its value onto the serial monitor.

```
int raw_cap = zssc3230.read_raw_cap();
Serial.println(raw_cap);
```

5.3 Advanced ZSSC3230 Setup

The ZSSC3230 has a configurable ADC resolution which ranges from 12-bit to 18-bit. It also has a configurable capacitance range and offset so you can increase the resolution of the capacitance measured. This can all be set up in the `.configure_sensor()` function.

5.4 Capacitance Range and Offset

Before calibrating the ZSSC3230, it is important to determine the optimum range and zero shift offset you require. You should choose a cap_offset that places you as close to the centre of the range of capacitances you would like to measure. You should then choose a cap_range that covers the desired capacitance range. Note if you choose 10pF as your range, this will extend in the positive and negative direction from your offset, creating an effective signal range of 20pF.

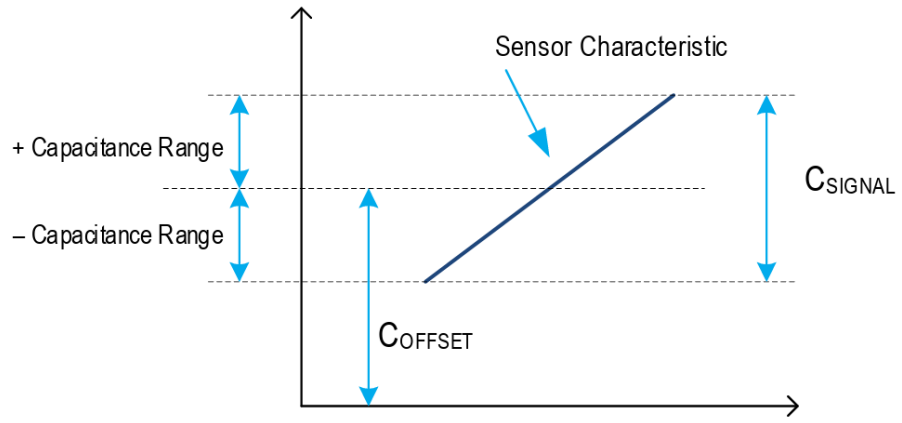


Figure 1 - Capacitive Input Signal Conditions (Renesas, 2022)

When you take an ADC reading using the `read_raw_cap()` function, the internal math of the ZSSC3230 converts the range of the N-bit ADC from $[0,\ 2^N - 1]$ to $[-2^{N-1}, 2^{N-1} - 1]$. This retains the same range, however, the return is a twos compliment number, where 0 is equivalent to the capacitance offset, -2^{N-1} is the lower bound of the capacitance range and $2^{N-1} - 1$ is the upper bound of the range.

6 Calibration

To calibrate your ZSSC3230 sensor, you can use the ZSSC3230_calibrate example included with the ZSSC2320_I2C_Driver Arduino library. Before running this example, please ensure you have three accurate capacitors within your desired capacitance range. Ideally:

- Capacitor 1 should be near the lower end of the range.
- Capacitor 2 should be near the middle of the range.
- Capacitor 3 should be near the upper end of the range.

You will also need to install the ZSSC323x Evaluation Software before beginning:

https://www.renesas.com/us/en/products/sensor-products/sensor-signal-conditioners-ssc-afe/zssc3230-low-power-high-resolution-capacitive-sensor-signal-conditioner#design_development

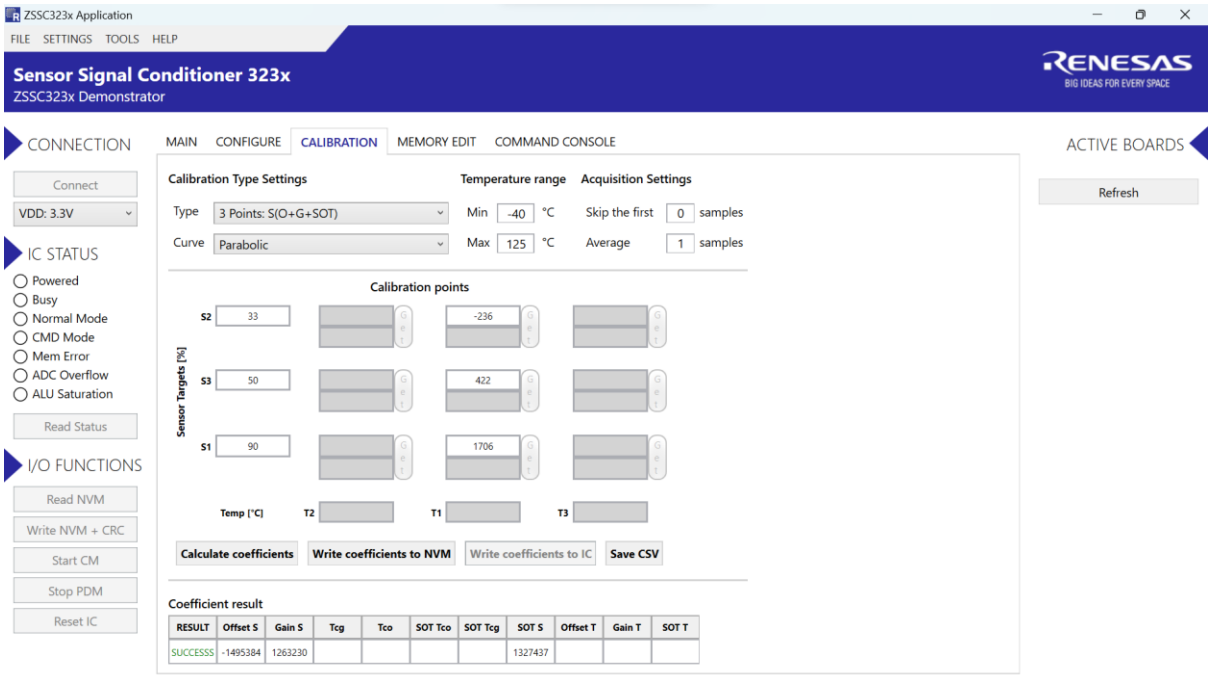
if you have any issues understanding the Evaluation Software, please see the guide from Renesas:

https://www.renesas.com/us/en/document/mah/zssc3230-evaluation-kit-manual?r=465426

We are primarily interested in the calibration page, you should not need to change any settings in the evaluation software.

You will also need to set your desired configuration for the ZSSC3230 using the `configure_sensor()` function in `setup()`, before you run the example code.

Run the ZSSC3230_calibration example and open your serial terminal. You should see prompts in the serial terminal to place your first capacitance sensor, your second capacitance sensor and then your third capacitance sensor. For each of these, the ZSSC3230 and Arduino will calculate an average ADC value over 100 samples (this should only take a second) and will print the value to you.



Following the steps outlined in the program, you will need to select the calibration tab in the ZSSC323x Evaluation Software. This window will look like the above figure. Select the “3 Points: S(O+G+SOT)” option in Type and the “Parabolic” option in Curve. Enter your sensor targets as a percentage of the

full range in the left column and the corresponding ADC values you recorded earlier in the middle column. Clicking the “Calculate coefficients” button at the base will result in a value underneath Offset_S, Gain_S and SOT_S. You can then enter these values into the Serial monitor when prompted and select yes if the values are correct. The ZSSC3230 should try and calibrate itself now. The example will let you know if the calibration was successful or not in the Serial monitor.

If the calibration was successful, the sensor will start sending calibrated capacitance values in picofarads (pF).. You can check this value against your test capacitors and see how good the calibration was!

6.1 SSC Capacitance Readings

Once you have calibrated your ZSSC3230 Sensor, you can now run the ZSSC3230_read_ssc_capacitance example. Alternatively, you can create your own code using the `read_ssc_cap()` function to read and return a floating-point capacitance in pF.

7 High Frequency Readings

For applications requiring fast readings of one or more sensors, the ZSSC3230 has a cyclic mode that is optimal for this. To initiate cyclic mode, call the `cyclic_mode()` function. Once called, you should only read the ssc capacitance using the `read_ssc_cap_cyc()` function. You cannot write to the ZSSC3230 during this time, unless you are telling it to go to sleep using `sleep_mode()`.

Speed is the major advantage when using cyclic mode over sleep mode. In sleep mode, the `read_ssc_cap()` function is a blocking function, delaying the microprocessor between asking for a ssc capacitance reading and receiving the reading. This is because it takes time for the ZSSC3230 to perform the required ADC reading and calibration mathematics. The exact delay is shown in the `read_ssc_cap()` function description. If you want to use cyclic mode, it is important to use the `read_ssc_cap_cyc()` function as there is no delay in it.

In cyclic mode, the ZSSC3230 performs its reading and calibration cyclically, without the need for a request from the I2C controller. The ZSSC3230 will automatically perform a reading, update its memory, and instantly begin a new reading. The speed of the reading is dependant on two settings: the ADC resolution and the noise mode. Referring to the data in the table below, with a 12-bit ADC and noise mode off, you can read new data from the ZSSC3230 every 1.77ms. If you were to read the data every 1ms, you would read the same value twice before reading a new value.

ADC Resolution	Conversion Time for Full Temperature Compensated Measurement, Typical (µs)	
	Noise mode off	Noise mode on
12	1770	2550
14	2310	3850
16	3390	6440
18	5520	11630

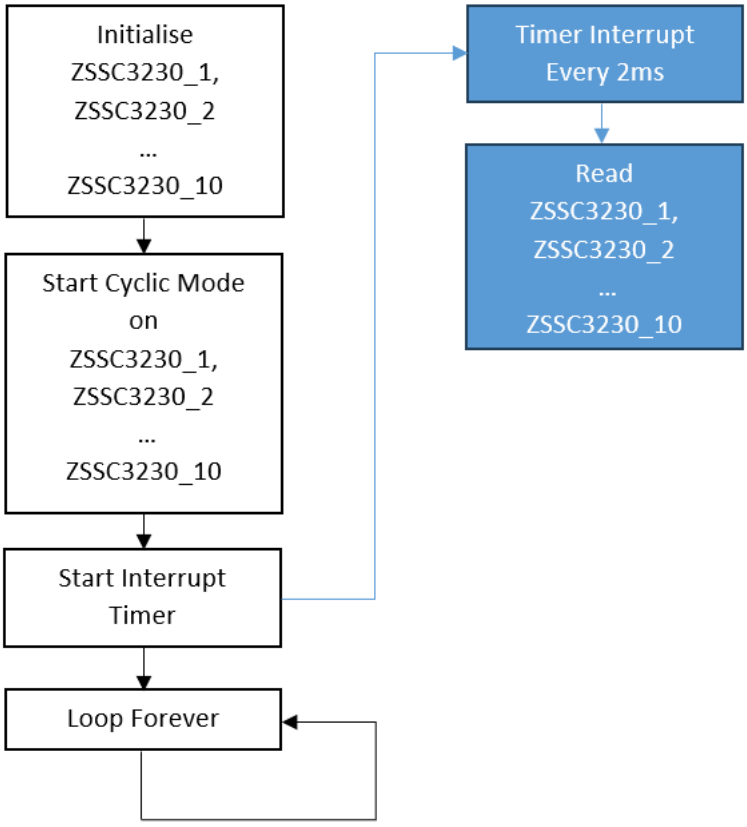
7.1 High Frequency Reading Multiple Sensors Example

Cyclic mode is extremely useful for reading multiple sensors at high frequencies. Using just I2C fast mode (400kHz), a 12-bit ADC resolution and noise mode off, it is theoretically possible to read up to 16 sensors in 2ms. Increasing this I2C speed to fast mode plus, it is theoretically possible to read 42 individual ZSSC3230 sensors in 2ms, all with new data. Below is a table with the theoretical time to perform an I2C transaction with one ZSSC3230 sensor and read a buffer from it.

I2C Speed (kHz)	Read (us)
100 (Normal)	470
400 (Fast)	117.5
1000 (Fast Plus)	47

The number of sensors in a real-world application that can be read in this 2ms margin will be slightly reduced due to some waiting time in the I2C process as well as processor time to write this new data through UART (Serial) or over Bluetooth. Experimentation will need to be conducted to find the maximum number of sensors your system can handle.

The workflow for a system with I2C clock speed at 400kHz and 10 ZSSC3230 sensors configured to 12-bit ADC and noise mode off could be:



This system will interrupt the infinite loop every 2ms to read 10 ZSSC3230 sensors consecutively. Theoretically, at an I2C clock speed of 400kHz, this should take 1.175ms, leaving time for other calculations or communications. Inside the infinite loop, you could take the data from all the ZSSC3230 sensors, write it to a buffer and transfer it over Bluetooth to a phone.

8 Methods

8.1 begin()

Description

Initialise the ZSSC3230. This should only be called once per sensor.

Syntax

```
ZSSC3230.begin(uint8_t deviceAddress, TwoWire &wirePort)
```

Parameters

- *deviceAddress*: i2c address of the device, non shifted, default 0x48
- *wirePort*: pointer to the TwoWire object used to run I2C, default "Wire"

Returns

- *true*: if the ZSSC3230 could be connected to and setup was successful.
- *false*: if the ZSSC3230 could not be connected to or the setup was not successful.

8.2 soft_reset()

Description

Soft reset the zssc3230.

Syntax

```
ZSSC3230.soft_reset(void)
```

8.3 sleep_mode()

Description

Set the ZSSC32320 into sleep mode, this is the chip's low power mode. It waits dormant until asked for a reading by a controller.

Syntax

```
ZSSC3230.sleep_mode(void)
```

Returns

- *0*: success.
- *1*: data too long to fit in transmit buffer.
- *2*: received NACK on transmit of address.
- *3*: received NACK on transmit of data.
- *4*: other error.
- *5*: timeout

8.4 cyclic_mode()

Description

Place the ZSSC3230 into cyclic mode. This will continuously calculate ssc capacitance data. This data can be read using `read_ssc_cap_cyc()`.

Syntax

```
ZSSC3230.cyclic_mode(void)
```

Returns

- *0*: success.
- *1*: data too long to fit in transmit buffer.
- *2*: received NACK on transmit of address.
- *3*: received NACK on transmit of data.
- *4*: other error.
- *5*: timeout

8.5 command_mode()

Description

Enter command mode. This does not really add much functionality so it is recommended to stay in sleep or cyclic mode. See the ZSSC3230 datasheet pg. 27 for more information on command mode.

Syntax

```
ZSSC3230.command_mode(void)
```

Returns

- *0*: success.
- *1*: data too long to fit in transmit buffer.
- *2*: received NACK on transmit of address.
- *3*: received NACK on transmit of data.
- *4*: other error.
- *5*: timeout

8.6 read_ssc_cap()

Description

Read the sensor signal conditioned (ssc) capacitance while in sleep mode. This requires the ZZSC3230 to be calibrated. This function asks for a ssc reading and waits the required conversion time before reading the ssc data from the ZSSC3230. The conversion times are listed below. Because of the conversion time, this function is a blocking function, delaying the microcontroller until the conversion is complete and the I2C buffer read.

ADC Resolution	Conversion Time for Full Temperature Compensated Measurement, Typical (µs)	
	Noise mode off	Noise mode on
12	1770	2550
14	2310	3850
16	3390	6440

18	5520	11630

Syntax

```
ZSSC3230.read_ssc_cap(void)
```

Returns

The floating-point capacitance in pF.

8.7 read_ssc_cap_cyc()

Description

Reads the ssc capacitance in cyclic mode. This requires the ZSSC3230 to be calibrated. This is optimal for high frequency readings as it is not a blocking function.

Syntax

```
ZSSC3230.read_ssc_cap_cyc(void)
```

Returns

The floating-point capacitance in pF

8.8 read_raw_temp()

Description

Reads the raw temperature through the temperature ADC. This is a 14-bit ADC with a factory calibrated range of -40°C to 125°C.

Syntax

```
ZSSC3230.read_raw_temp(void)
```

Returns

The raw temperature as a floating point number in °C.

8.9 read_raw_cap()

Description

Reads the raw ADC capacitance. The ZSSC3230 converts the ADC range from $[0, 2^N - 1]$ to $[-2^{N-1}, 2^{N-1} - 1]$, where N is the bit resolution of the ADC. This still has the same effective range, however, a raw reading of 0 is equivalent to the middle of the ADC range.

Syntax

```
ZSSC3230.read_raw_cap(void)
```

Returns

The ADC value as a signed integer within the range $[-2^{N-1}, 2^{N-1} - 1]$.

8.10 read_nvm_reg()

Description

Read an NVM register from the ZSSC3230 over I2C.

Syntax

```
ZSSC3230.read_nvm_reg(uint8_t *buffer, uint8_t regAdd)
```

Parameters

- **buffer*: a pointer to a buffer of size uint8_t to write the ZSSC3230 data onto.
- *regAdd*: the NVM register address you want to read. Valid is 0x00 to 0x18.

Returns

- *true*: if the I2C read has been successful.
- *false*: if the I2C read has failed.

8.11 read_zssc3230()

Description

Read data from the ZSSC3230 over I2C.

Syntax

```
ZSSC3230.read_zssc3230(uint8_t *buffer, int len)
```

Parameters

- **buffer*: a pointer to a buffer of size uint8_t to write the ZSSC3230 data onto.
- *len*: the number of bytes requested from the ZSSC3230.

Returns

- *true*: if the I2C read has been successful.
- *false*: if the I2C read has failed.

8.12 write_zssc3230()

Description

Write data to the ZSSC3230 over I2C.

Syntax

```
ZSSC3230.write_zssc3230(uint8_t regAdd, uint16_t data)
```

Parameters

- *regAdd*: the register address to send over I2C. This is a command to access NVM memory or ask for a capacitance sample.

- *data*: if writing to NVW memory, this is where the data is placed. Otherwise write this as 0x0000.

Returns

- *0*: success.
- *1*: data too long to fit in transmit buffer.
- *2*: received NACK on transmit of address.
- *3*: received NACK on transmit of data.
- *4*: other error.
- *5*: timeout

8.13 calibrate_zssc3230()

Description

This function uploads the value of Offset_S, Gain_S and SOT_S to the NVM memory of the ZSSC3230 in sign magnitude format. This function only works for 2 Points: S(O+G) and 3 Points: S(O+G+SOT) calibration. If using the 2 Points: S(O+G), you can set SOT_S to 0.

Syntax

```
ZSSC3230.calibrate_zssc3230(int Offset_S, int Gain_S, int SOT_S)
```

Parameters

- *Offset_S*: The capacitance offset coefficient “Offset S”.
- *Gain_S*: The capacitance gain coefficient “Gain S”.
- *SOT_S*: The second order coefficient “SOT S”.

Returns

true if all I2C writes have been successful.

8.14 configure_sensor()

Description

Configure register 0x12 in the ZSSC3230 NVM. This controls the sensor capture type, sensor leakage mode, capacitance range, noise mode, adc resolution and capacitance shift. Once this function has been run, the ZSSC3230 will hold this configuration until you decide to change it again.

Syntax

```
ZSSC3230.configure_sensor(SENSCAP_TYPE sct, SENSOR_LEAKAGE slc, CAP_RANGE cap_range,
NOISE_MODE noise_mode, ADC_RES adc_res, CAP_OFFSET cap_shift)
```

Parameters

Parameters	Description	Argument Options	Example Configuration
`sct`	Sensor capture type	`TYPE_DIFFERENTIAL` `TYPE_SINGLE_END_SENSOR`	`TYPE_DIFFERENTIAL`

slc	Sensor leakage cancellation	LEAKAGE_CANCELLATION_OFF LEAKAGE_CANCELLATION_ON	LEAKAGE_CANCELLATION_OFF
cap_range	Range of the capacitance (0.5 to 16pF in 0.5pF increments). The range extends either side of the zero-shift offset.	RANGE_0_pF_5 RANGE_1_pF_0 RANGE_1_pF_5 RANGE_2_pF_0 . . . RANGE_15_pF_0 RANGE_15_pF_5 RANGE_16_pF_0	RANGE_15_pF_0
noise_mode	Noise mode	NOISE_MODE_OFF NOISE_MODE_ON	NOISE_MODE_OFF
adc_res	Analogue to digital converter resolution	ADC_12_BIT ADC_14_BIT ADC_16_BIT ADC_18_BIT	ADC_12_BIT
cap_shift	Capacitance zero-shift offset (0 to 15.75pF in 0.25pF increments)	OFFSET_0_pF_0 OFFSET_0_pF_25 OFFSET_0_pF_5 OFFSET_0_pF_75 OFFSET_1_pF_0 OFFSET_1_pF_25 . . . OFFSET_15_pF_0 OFFSET_15_pF_25 OFFSET_15_pF_50 OFFSET_15_pF_75	OFFSET_15_pF_0

Returns

true if I2C write was completed successfully.

8.15 set_i2c_address()

Description

Set the I2C address of the zssc3230 in its NVM memory. Please note that:

Syntax

```
ZSSC3230.set_i2c_address(uint8_t i2cAddress)
```

Parameters

- *i2cAddress*: The desired 7-bit address (unshifted)

Returns

True if the I2C transaction was successful.

8.16 enable_debugging()

Description

Set the I2C address of the zssc3230 in its NVM memory. Please note that:

Syntax

```
ZSSC3230.enableDebugging(uint8_t i2cAddress)
```

Parameters

- *i2cAddress*: The desired 7-bit address (unshifted)

Returns

True if the I2C transaction was successful.