Project description:

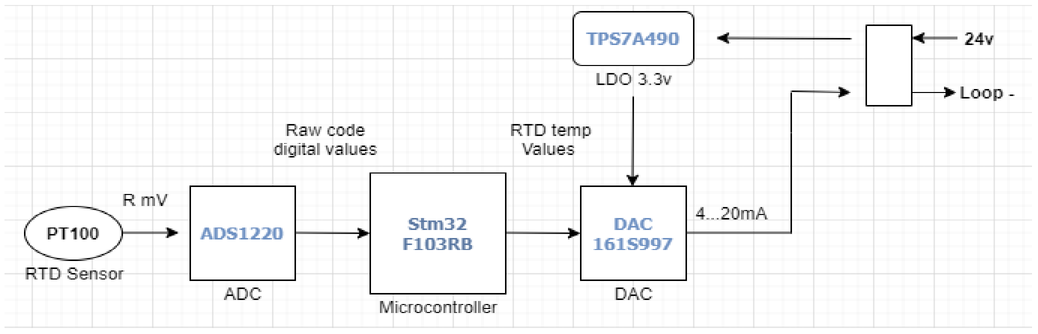
It's an industrial temperature sensor, based on RTD PT100, which offers the possibility to read the measurements in situ or from a remote terminal,

the purpose of this system is to keep the traceability of the storage's temperature for products in industrial environement.

Requirements

- Operating in -10°C to +100°C temperature range,
- Ensure accuracy measurements +/- 1.5°C
- Must ensure minimum of 2 Days autonomy in case of mains supply failure.
- Include a charge circuit.
- Include a serial communication for debug purpose.

Functionnal block diagram



Theory of operation

The RTD sensor is based on the resistance variation according to temperature. The sensor needs to be excited with an external current source (about 1 mA) to provide a differential analog signal.

Before feeding the ADC input with the RTD voltage, this one should be conditionned using a differential / instrumentation amplifier.

The ADC converts the conditionned voltage signal into a digital one, (a sort of raw code), obtained through comparison with reference voltage of the ADC , the digital output values are proportional to RTD variation.

The microcontroller converts the input raw code into RTD values, then apply CVD formula to obtain temperature value.

Components selection

Signal conditionning and digital conversion

These operations are ensured by the “ADS1220” ADC from TexasInstruments (TI), this ADC is 24-bit which provides a high precision measurements.
The “ADS1220” provides a current through its pins “AIN”, once excited, the RTD sensor provides an analog voltage Vrtd_(mV), proportional to the variation of the resistance.

The “ADS1220” contains a Programmable Gain Amplifier (PGA) which amplifies the differential voltage, the ADC converts the signal into a digital one, the resulted signal is a raw code.

The chip can operate under a wide supply range : from 2.3v to 5.5v and uses SPI protocol.

Microcontroller

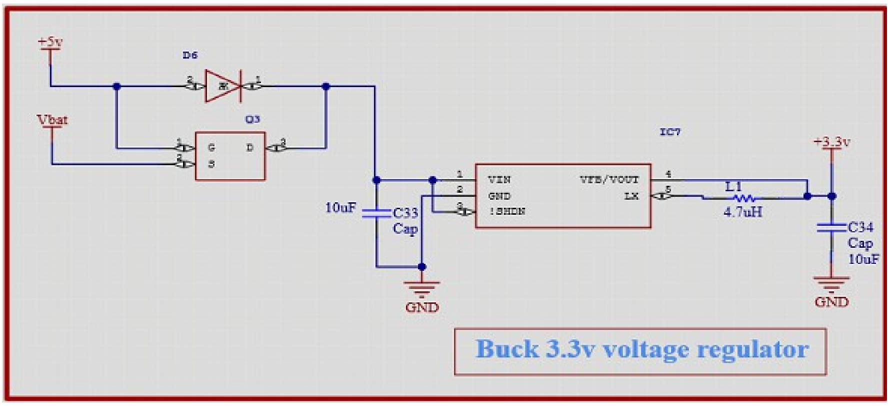
As i'm familiar with stm32 ARM-M3 microcontroller, I decided to use STM32F103RB series, this MCU offers all needed peripherals : SPI, I2C , 2xUARTS, GPIOs ...

The microcontroller can be supplied from 2.0v to 3.6v and draws 46mA in run mode. Moreover, it provides different modes for low power consumption through : sleep mode, stop and standby mode.

For the project purpose, the “F103RB” fits the requirements.

Hardware Design

Power supply block



The system can be powered from a +5v DC provided by an external charger, or through a battery.

There are two batteries connected in parallel, which provide 3600mAh capacity, and 4.2v (Vbat).

The p-Channel Mosfet (Q3) acts as a switch between external +5v and Vbat.

1) When the mains supply is present, means that Vg > Vs+Vth (as Vg = +5v and Vbat = 4.2v).

   Vth = -0.55v

   Vgs >0 -> the Mosfet is Off mode, the current is flowing through D6, thus, the system is powered by +5v.

2) When the mains supply is absent(case of power failure), it means that Vg < Vs (as Vg = 0v, and Vbat = 4.2v)

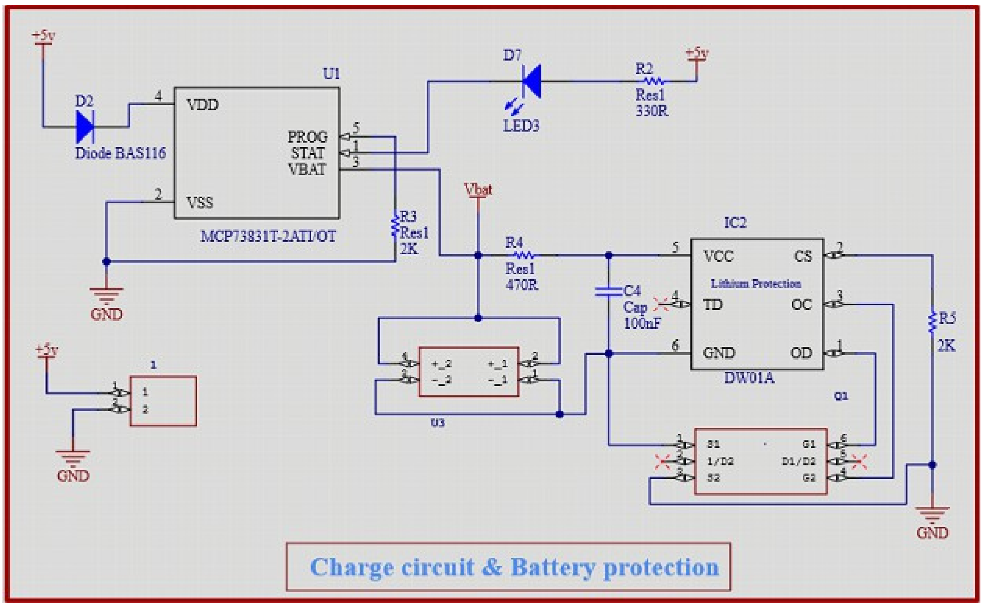
   Vgs <0 -> the Mosfet is ON mode, the current is flowing through drain and source , thus, the system is powered by Vbat.

Voltage regulator:

The “MCP1603” Buck regulator, which provides a fixed +3.3v output voltage with high efficiency, the use of L1 inductor and the decoupling capacitor (C34) connected to the load are reducing the output voltage ripple and filter the noise/voltage spikes.

The regulator can operate from Vbat or +5v and provide 500 mA load current.

Charge and circuit protection



Power management system controls the charge of the battery

The charge management controller monitors the charging process for the battery.

The "MCP738312" enhance the safety and efficiency of the battery, it uses a constant current and voltage charge algorithm. as we are using a +4.2v battery with 3600mAh capacity in our circuit, the "MCP" can provide a selectable voltage regulation of 4.2v, and constant charge current of 500mA.

The "MCP" is supplied with +5v, the charge current is determined by the resistor R3, 500mA is the maximum current charge that can be provided by the circuit for Vsupp = +5v in the datasheet, so the 2k resistors is applied into PROG pin of the IC.

Battery protection circuit (DW01):

Protects lithium ion/polymer battery from damage or degrading the lifetime due to overcharge, overdischarge, and/or overcurrent.

"DW01-P" is an IC used for a +4.2 battery voltage , it protects the battery according to voltages threshold : Vocp(over charge protection) , Vodp (over discharge protection).

The circuit is powered by Vbat as we can see in the schematic.
R4 resistor is used for ESD (electro-static-discharge) protection, the recommended value is 470R.

C4 capacitor is used to reduce power fluctuation, 100nF is a suitable value.
R5 resistor prevents large current when a charge is connected in reverse, 2K is recommended.

The dual N-channel MOSFET (Q1) consists of two N-channel mosfets, the first is connected to "OC" pin is a charge control mosfet, it is responsible for inhibition of charging when the voltage of the battery cell exceeds the

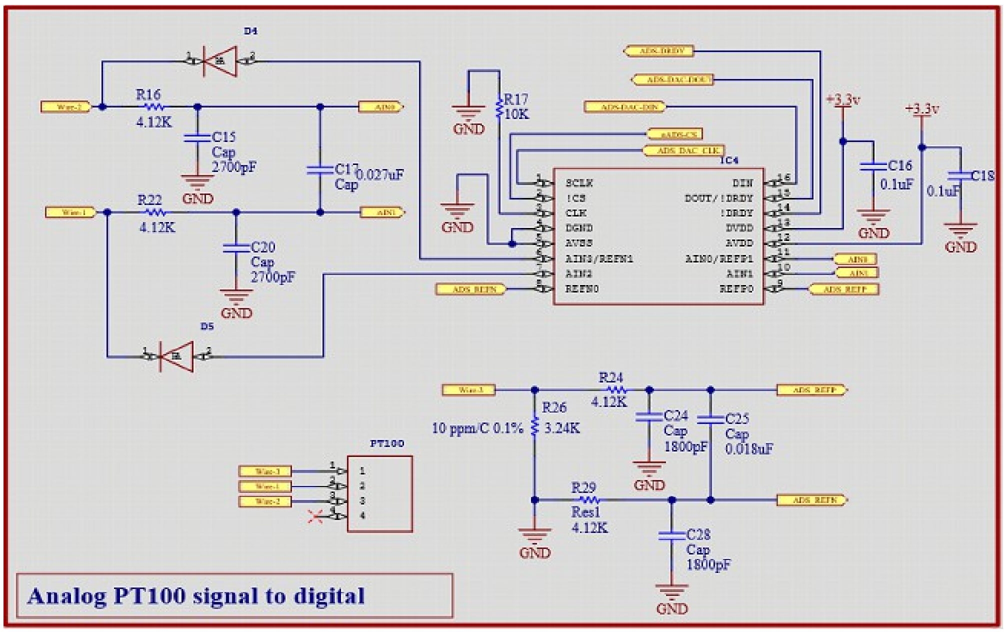
overcharge protection voltage (Vocp) which is fixed at +4.25v .
The second N-channel MOSFET is connected to "OD" pin is a discharge control MOSFET, it is responsible for the discharge inhibition when the voltage of the battery cell goes below the overdischarge protection voltage (Vodp) which is fixed at +2.4 v.

The "DW01" detects the discharge current in the case when we have a connected load which draws a current greater than a threshold current (ith), by measuring the voltage of "CS" pin, if this voltage exceeds the over current protection voltage (Voip :150mV typ value), then the discharge control mosfet is turned off and the discharge is inhibited.

The over threshold current (ith) is determined by the turn on resistance (RDSON) of the dual N-channel mosfet, and can be calculated using the formula :
RDSon = Voip/(2*Ith).

So, when we design an over threshold current to be 2A for example, we must choose a RDSOn of the mosfet to be 37.5mOhm .
MOSFET design : we must make sure that all these parameters are taken into account :
- Vgsth of the dual N-channel mosfet is < (Vod, Voc),
- Vgsmax > (Vod, Voc),
- Vds < Vload or Vbattery

ADC circuit for PT100 analog signal



The ADS1220 is a 24-bit ADC, .
The circuit measures the differential voltage between "AIN0" and "AIN1", which corresponds to the RTD voltage, performs the conversion of this input voltage, and stores the digital values which are proportionnal to the input signal, into a data buffer, the SPI interface is used to read the conversion data, R/W the device configuration registers and control device operations.

The ADC provides a current IDAC1+IDAC2 (500mA) through its pins "AIN2", "AIN3" , once excited, the RTD sensor provides an analog voltage Vrtd_(mV), proportional to the variation of the resistance.
The ADS1220 contains a PGA which amplifies the diffrential voltage ,the ADC converts the signal into digital ones, the resulted signal is a raw code.
The microcontroller records the raw ADC code from the ADS1220, applies an offset

and gain calibration, converts the ADC code into resistance values, performs linear interpolation to overcome the non-linearity of the RTD, converts the resistance value into temperature using CVD (callendar-van-Dusen) equation.

The pT100 is 3 wire,

“AIN” stands for analog inputs

the differential RTD voltage is measured between Wire1->AIN0 and Wire2->AIN1.

the circuit is using a ratiometric measurement approach, which means that the reference voltage [Vref = R26*(IDAC1+IDAC2)]
and RTD voltage [Vrtd = (Rrtd*IDAC1)] are derived from the same excitation source,
IDAC1 and IDAC2 are routed from AIN2 and AIN3 and configured by software.

R16, R22, C15, C20 and C17 are forming a first-order differential and common-mode RC filter, and placed on the ADC input.

As well as R24, R29, C24, C25 and C28 which are placed on the reference input

R26 is a reference resistor which serves to generate the reference voltage for the ADC and also sets the common-mode voltage for RTD measurement.

To avoid self-heating for RTD element, the excitation current for pt100 is chosen as IDAC1 = IDAC2 = 250uA, thus, 500uA current is flowing through reference resistor (R26)

The ADS1220 datasheets recommends setting Vref at near half analog supply (AVDD) :
Vref = 1.62v

Rref = [Vref / (IDAC1 + IDAC2)] = 3.24KR

After setting Rref, the PGA (Gain)is configured in software, its values is selected to match the maximum input signal to FSR : [Vin_max = (Rrtd_max * IDAC1)]
Considering Rrtd_max = 391R at 850°c, Vin_max = 97mV
so, when using Vref = 1.62v, the maximum gain that can be applied to input signal is determined through this formula :

[PGA = (Vref / Vin_max)] = 16.49,

We check the nearest value on FSR table in datasheet : 16 is the nearset value,

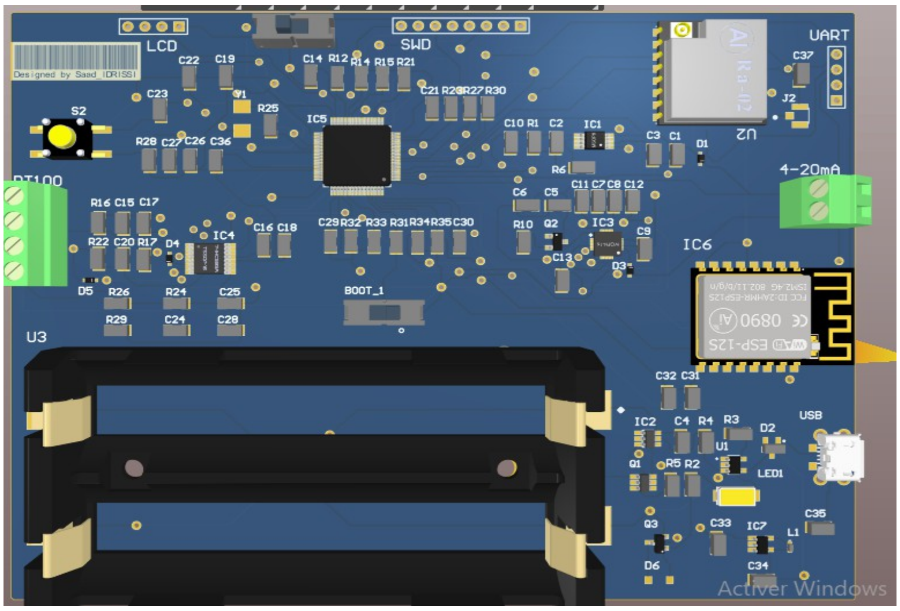
at this gain value , the ADS1220 offers a FSR value as described in the following formula :

the full scall range is then calculated : [FSR = (Vref / PGA)] = +/- 101mV represents the max FSR which can be applied to PGA

that means that the maximum input signal provided by RTD sensor cannot swing over +/-101mV

which allows a good marge for maximum input signal : 97 mV is lower than 101mV
This range allows for margin with respect to initial accuracy and drift of the IDACs and reference resistor

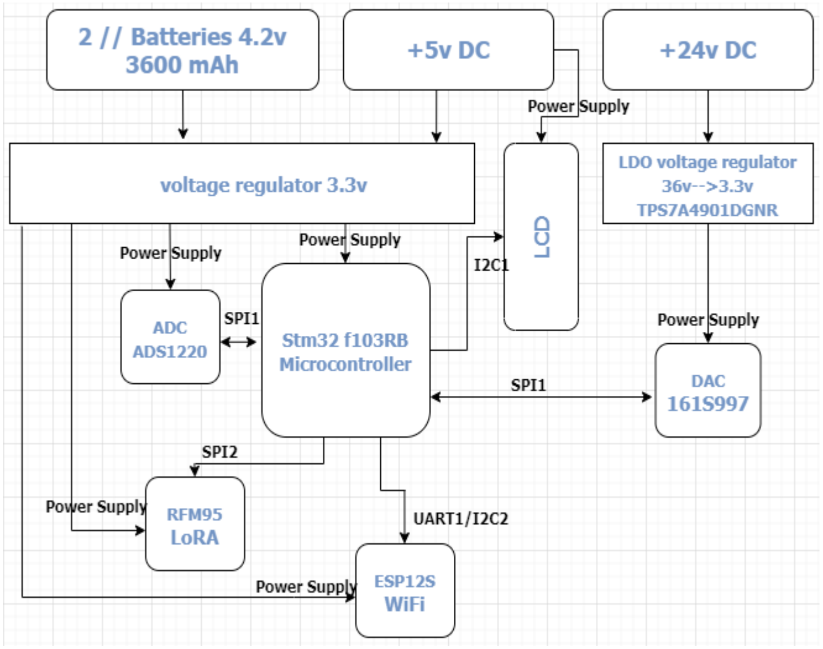
PCB designed with Altium Designer:



Firmware design

We start creating our system architecture by drawing a software block diagrams that includes the relationships between the various modules of the software system, which gives a global view of the system, we will go then more deeper and highlight the interaction (communication protocol) between the microcontroller and each module and how to configure them through registers, and writing a pseudo-code to summarize the steps for initializing and configuring each module.

software block diagram



a) Microcontroller to serial terminal intercation (UART) :
The first part that should be developped is the communication between the MCU and serial terminal (laptotp),
it allows to get an idea whether our MCU is working or not, and used for debugging purpose.

b) Microcontroller to ADS1220 interaction (SPI):

The ADS1220 "ADC" communicates with the microcontroller using a serial communication interface (SPI) which stands for serial peripheral interface.
The communication is initiated by the Master (MCU in our case), which select the slave (ADS1220) trough CS pin (software select), and generates a clock signal, at every clock cycle, the master sends a bit to the slave using MOSI pin, at the same time, the slave sends a bit to the master through MISO pin.
this opration is based on full duplex mode, where master/slave can communicates with each ones simultaneously.

The MCU will perform Read and write operations on the ADC, which will be explained later in the pseudo-code section

c) Microcontroller to LCD interaction (I2C):

The LCD is using I2C protocol, (Inter-Integrated Circuit), it acts as the slave, the device master is the microcontroller,
I2C is a half-duplex protocol which uses 2 lines for communication, SDA for data and SCL for generating clock.
Both LCD and MCU can take control over the SDA line by pull it low, but not at the same time.SCL is always driven by the master device.
In our case, we are using LCD only to display temperature values, that means only I2C-write operations are performed on LCD which includes LCD configuration and sending the data.

Peripheral test:

Before starting to configure and program the devices, we need to implement each protocol (UART, SPI , I2C) and test it independantly, in this manner, we ensure that the communication between the MCU and each device is working.

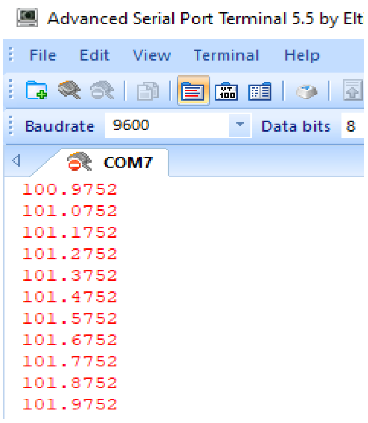
1) UART test:

We write a small code which sends a float value to the serial monitor, we are using UART1 peripheral (PA9->Tx, PA10->Rx).
This small program will serve to display the temperature values on serial monitor.

The code :

```
#include "stm32f10x.h"                  // Device header
#include "printf.h"
#include <math.h>
#include <stdio.h>


void USART1_Init(void);
void USART_write(int ch);
void DelayTimerUs(int n);


void USART1_Init(void){
  RCC->APB1ENR |= RCC_APB1ENR_USART1EN;  // Enable clock USART2 APB1 bus
                                            (Activate the peripheral USART1)
  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;     // enable clock for USART2TX which is
                                            connected to PA2 P180 Ref manual
  RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;
  GPIOA->CRH  |= GPIO_CRH_MODE9_0 | GPIO_CRH_MODE9_1;
  GPIOA->CRH  |= GPIO_CRH_CNF9_1  | GPIO_CRH_CNF9_0 ;

  USART1->BRR  |= 0xEA6;
  USART1->CR1  |= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE;

}

void USART_write(int ch){

      while(!(USART1->SR & USART_SR_TXE)){}  // we check if the transmit buffer
                                               is empty before sending the data

 USART1->DR = (ch );     // contains the received or transmitted data
                         // we put the data which we will send in
                            DR register
      }

int main (void){

      USART1_Init();
      float myfloat = 100.9752;

      while(1){
            printf_("%.4f \r\n", myfloat);
            myfloat+=0.1;}

}
```

printed values :


Advanced Serial Port Terminal 5.5 by Elt
File Edit View Terminal Help
Baudrate 9600
Data bits 8
COM7

```
100.9752
101.0752
101.1752
101.2752
101.3752
101.4752
101.5752
101.6752
101.7752
101.8752
101.9752
```

2) SPI test :

This code includes the GPIo's and SPI peripheral configuration .
We set the SPI frequency to 72Mhz / 128 = 562.5Khz

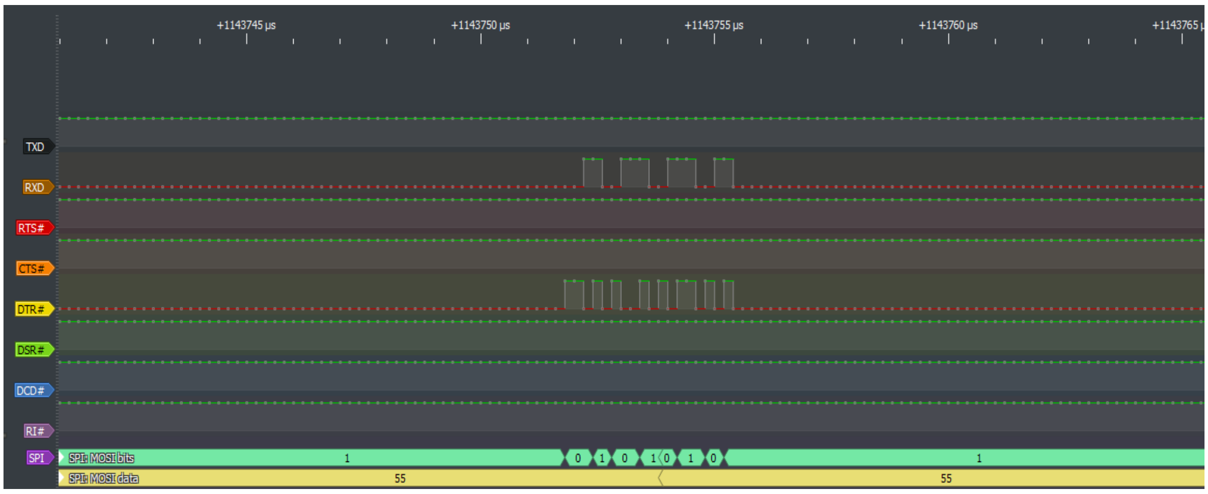
we send a data = 0x55 through MOSI line, this value correspond to 01010101b which is convenient for debug.
the SPI pins are connected as follow : (PA5->CLK, PA7->MOSI line, PA6->MISO, PA4->CS),

The SPI code :

```
#include "stm32f10x.h"                  // Device header
#define ADS1220_CS    0x10
/* Definition of GPIO stm32 Port Bits Used for Communication */

/*
//Pins configuration

SPI_CLK -> PA5 (AF)
SPI_MISO -> PA6 (AF)
SPI_MOSI -> PA7 (AF)
SPI_CS -> PA4 (GPIO)

void GPIOa_Init(void);
void SPI1_init(void);
void DelayTimerUs(int n);
uint8_t SPI_Send(uint8_t data);


void GPIOa_Init(void)  // set gpios for ADS1220 and stm32 I/O
{

// We are using PortA and PortC for SPI1 communication, first we should enbale
clock for those ports
      RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
      RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

}

void SPI1_init(void){

// Setup (PA4-->NSS /PA5-->SCK / PA6-->MISO / PA7-->MOSI)  as Alternate function
for SPI1

// PA4 --> CS as Output General-Purpose push-pull (50Mhz) ref p167-161
      GPIOA->CRL |=  GPIO_CRL_MODE4_0 ;
      GPIOA->CRL |= GPIO_CRL_MODE4_1;
      GPIOA->CRL &=~ GPIO_CRL_CNF4_0;
      GPIOA->CRL &=~ GPIO_CRL_CNF4_1;

      GPIOA->BSRR |= GPIO_BSRR_BS4; // Set idle state as high
      GPIOA->CRL |=  GPIO_CRL_MODE5_0 ;
      GPIOA->CRL |= GPIO_CRL_MODE5_1;
      GPIOA->CRL &=~ GPIO_CRL_CNF5_0;
      GPIOA->CRL |= GPIO_CRL_CNF5_1;

// PA6-->MISO as Full-Duplex MASTER -->Input pull up
      GPIOA->CRL &=~  GPIO_CRL_MODE6_0 ;
      GPIOA->CRL &=~ GPIO_CRL_MODE6_1;
      GPIOA->CRL &=~ GPIO_CRL_CNF6_0;
      GPIOA->CRL |= GPIO_CRL_CNF6_1;

// PA7-->MOSI as Full-Duplex MASTER -->AF push-pull
      GPIOA->CRL |=  GPIO_CRL_MODE7_0 ;
      GPIOA->CRL |= GPIO_CRL_MODE7_1;
      GPIOA->CRL &=~ GPIO_CRL_CNF7_0;
      GPIOA->CRL |= GPIO_CRL_CNF7_1;

// Enable SPI1 clock
     RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
// set RXONLY to 0 to select Full-Duplex Mode
      SPI1->CR1 &=~SPI_CR1_RXONLY ;

//2 : Configure SPI1  P707-743
      SPI1->CR1 &=~SPI_CR1_BIDIMODE; // Use 2 lines Full duplex
      SPI1->CR1 &=~ SPI_CR1_DFF;      // 8 bits Data Frame Format p742
      SPI1->CR1 &=~ SPI_CR1_LSBFIRST;  // MSB first

// Clock Phase
      SPI1->CR1 &=~ SPI_CR1_CPOL; // Set clock Plority
      SPI1->CR1 &=~ SPI_CR1_CPHA;  // set Clock Phase

// Baud rate control
      SPI1->CR1 |= SPI_CR1_BR_2| SPI_CR1_BR_1 ; // Fpclk=72Mhz/128 = 562.5 Khz
      SPI1->CR1 &=~ SPI_CR1_BR_0  ;


//Software slave select  --> In sofwtare NSS , PA4 could be used as GPIO
//Set as master (SSI must be high), with software managed NSS
      SPI1->CR1 |= SPI_CR1_MSTR | SPI_CR1_SSI | SPI_CR1_SSM;
// Enable SPI1 peripheral
      SPI1->CR1 |= SPI_CR1_SPE ;
      }

uint8_t SPI_Send(uint8_t data){

   while(!(SPI1->SR & SPI_SR_TXE)) {}
   *(__IO uint8_t *)&SPI1->DR = data;
            DelayTimerUs(100);

                  return data;
      }



int main(void){

      GPIOa_Init();
      USART1_Init();
       SPI1_init();

      while(1)
            {
            int TX = SPI_Send(0x55);
            DelayTimerUs(10000);

            }
}
```

This capture is taken from the logic analyzer :

DTR trace correponds to CLOCK signal
RXD trace correponds to MOSI line
as we can see, for clock_polarity(CPOL) = 0 and clock_phase(CPHA) = 0 (data is captured on rising edge and shifted out on the falling edge)



3) I2C test :

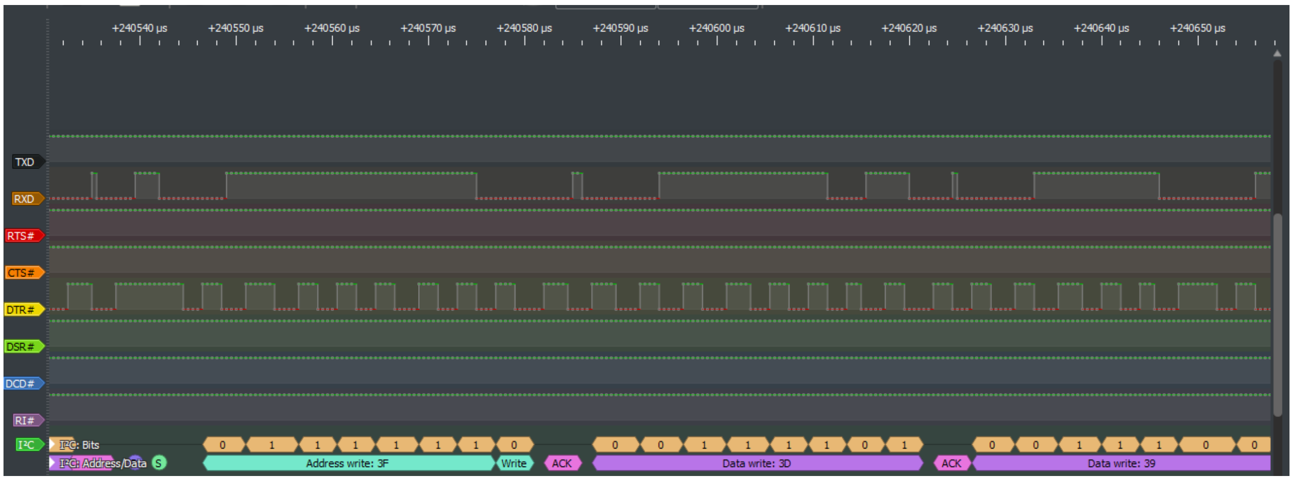
We configure the GPIO's for I2C protocol PB6->SCL1 & PB7->SDA1.

We set I2C frequence as standard mode : 100Khz
The LCD address is 0x7E

The code is described as follow :

```
#include "stm32f10x.h"                    // Device header
#include "stdint.h"
#include <stdio.h>


void GPIO_Init(void);
void I2C1_Init(void);
void DelayTimerUs1(int n);
void I2C_send (uint8_t slave_addr, uint8_t data);


void GPIO_Init(void)
{

      // Port B is used for I2C1 communication :


      RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;

      RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;

      RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;

        GPIOB->CRL &=~ (GPIO_CRL_MODE6_1 | GPIO_CRL_MODE7_1);
        GPIOB->CRL |=  GPIO_CRL_MODE6_0 | GPIO_CRL_MODE7_0;
        GPIOB->CRL |=  GPIO_CRL_CNF6_0  | GPIO_CRL_CNF7_0;
        GPIOB->CRL |=  GPIO_CRL_CNF6_1  | GPIO_CRL_CNF7_1;


}

void I2C1_Init(void)

{
      I2C1->CR1 |= I2C_CR1_SWRST;
      I2C1->CR1 &=~ I2C_CR1_SWRST;
      I2C1->CR2 |= 36;
      I2C1->CCR |= 180; // 100Khz
      I2C1->TRISE |= 37;
      I2C1->CR1 |= I2C_CR1_PE;
}

void I2C_send (uint8_t slave_addr, uint8_t data)

{
   uint8_t  temp2, temp0;

  while (I2C1->SR2& I2C_SR2_BUSY){};

  if ((I2C1->SR2 & I2C_SR2_BUSY) == 1) {}

      I2C1->CR1 |= I2C_CR1_ACK;
      I2C1->CR1 |= I2C_CR1_START;

// 2) Check START condtion send

      while (!(I2C1->SR1 & I2C_SR1_SB));

temp2 = I2C1->SR1;  // EV5  if the SB flag is not 1, do nothing,this means that
                       the START condtion was not sent yet

// 3) Transmit the slave addresse

      I2C1->DR = slave_addr; // 0x7F

while (!(I2C1->SR1 & I2C_SR1_ADDR)) ; // wait for ADDR flag in status register,
                                      //if there was a match for this slave
                                      //address, clear flag

  temp0 = I2C1->SR1 | I2C1->SR2;  // EV6 clear ADDR flag by reading SR2

// 4) Send the Data to data_register

while(!(I2C1->SR1 & I2C_SR1_TXE));

            I2C1->DR = data;

while (! (I2C1->SR1 & I2C_SR1_BTF)); // Wait for bte transfer to finish

I2C1->CR1 = I2C_CR1_STOP;  // send another start condition if we have another
                              byte
      }

int main (void){

        GPIO_Init();
        I2C1_Init();
while(1)
            {
            I2C_send(0x3F, "test");
            }

}
```

I2C traces :



As we can see : the I2C transaction doesn't correspond to the configuration , we should see the slave adress (0x7E) transmitted by the MCU on SDA line every rising edge of clock signal.
As we are performing only a Write transaction,at the 8th clock rising edge , the SDA line should be pulled low (0) because the master is still taking the control of the data line.
After that, the MCU release the SDA line, waiting for the slave (LCD) to send back the Aknowledgment bit.

Unfortunately, none of those operations have been identified on I2C lines, which seems illogical according to the configurations and the datasheet directives,
The slave is working with 0x7F adress, but in the logic analyzer is found on 0x3F adress
 the code seems to be correct.

After some investigations, I figured out that F1 series from STM32 MCU's (the series which i'm using)are suffering from some bugs, which leads to similar problems when working with bare metal.

Another alternative is the use of HAL I2C library as they are implementing a specific work-around for it.

Pseudo-Code (ADS1220)

Before performing the registers configurations, we need to implement functions that allow to communicate with the device , mainly Write and Read functions which depends on the used communication protocol , SPI in this case.
These functions are written following the datasheet requirements.

This pseudo-code highlights the following steps :

- Write data (MCU to ADC)
- Read data (ADC to MCU)
- Write ADC registers configurations
- Read ADC registers configurations
- ADC initialization
- Read ADC values

Write operation : The MCU sends data to ADS1220 through MOSI line

SPI is using shift register to shift out the data to be transmitted bit by bit each clock cycle, 8 clock cycles for 1 byte transfer

- Pull down cheap select pin (CS)
- puts the data into transmission buffer
- the flag check if the transmisison buffer is empty through TXE flag
- the data is transferred to MOSI shift register.
- the Data is shifted out according to MCU clock cycles.

Read operation : The MCU reads data from ADS1220 through MISO line

The ADS1220 needs 8 clock cycle to shift out 8-bit data from DOUT pin through MISO line, the Data in MISO shift register is transferred to Rx buffer,after the 8th clock edge.
The RXNE flag is set , indicates that the data is ready to be read from DR.
When the SPI_DR register is read, the SPI peripheral returns the buffered value.

- Pull down cheap select pin (CS)
- Send dummy byte (0xFF) which correponds to 8 clock cycles (shift out data from ADS1220)
- The RXNE flag checks wether Rx buffer is empty or not
  wait until RXNE is set which indicates that there is a data into the buffer.
- Read data register
- Return the buffered value

Write ADC registers configurations :

There is a specified command to write in ADC registers : “WREG” command.
As described in Datasheet, this 8-bit command = (0100rrnn)
 rr = Register Adress
 nn = number of bytes to write in config register (we use 1 byte -> 00)

- Write the following command : (WREG|addr<<2)
 Since we are including the register address as argument in Write-register function,
for example :
we use “bitwise or” operator between (0100 00 00 | 01<<2) -> configuring the register at 0x01 adress,and write 1 byte data
In this case, we shift the address 0x01 by 2 to get : 0100 01 00

- Write the data (configuration value) in the register .

Read ADC registers configurations :

As sanity check, The read-register function reads back and returns all configuration registers , using RREG command

RREG = (0010 rr nn) same logic as WREG command, the only difference is the use of read() function (desribed previously) to get the data from ADS1220.

- Send (RREG|addr<<2) : addr stands for register address, from which we want to read back to configuration value.
- Call “read()” function which will return the content of DR (register config value in this case).
- Store te Read() function return value, in a 1 byte data .
- Return this data (called register value)

ADC intialization :

The ADS1220 datasheet page 48 gives the pseudo-code sequence with the required steps to set up the device.

Read ADC values operation

This function allows to read the converted values (digital code) generated by the ADC, we use “RDATA” command, the digital code is 24-bit large.
We cannot retrieve the 24 bit data at one time.

- Create a 24-bit data, where we can store the ADC output value

Send RDATA command;
Call read() function and stores its return value into the 24-bit data
(this operation will retrieve the first 8-bit data from ADC)
Shift left the data by 8 .
Call read() function to retrieve the second 8-bit data from ADC and store it into the 24-bit data.
Shift left by 8
Call read() function to retrieve the third 8-bit data.
Stores the result into the 24-bit data.

At the end of these operations and after sending 24 clock cycle, the 24-bit data should contains the 24-bit output ADC value.

We store the result into 32-bit data, and returns it.

Registers configuration :

The device is fully configured through 4 registers, and controlled by 6 commands using SPI mode 1 (CPOL = 0, CPHA=1).
In order to configure the device, we should see the Register map section in the datasheet, this part shows each register, the correponding address and the corresponding bit, each bit has a specific task, so we should be careful and configure each bit according to our technical need.
The ADS1220 has 4 registers of 8-bits (1 byte), accessible through SPI using RREG and WREG commands.

1) Register 0 (offset= 00h)

The register 0 is used to configure 3 parameters :
- MUX[3:0] : input multiplexer configuration, to select the analog input of the PGA, as we are using 3 wire RTD, the resistance is measured between 2 wires (white-red), the third wire is used for compensation, therefore, we connect the (white--> AIN0) & (red-->AIN1) which correspond to analog input, respectively to AINp and AINn of PGA, and that correspond to set MUX[3:0] =0000
- GAIN[2:0] : The ADS1220 amplifies the RTD voltage (mV) using the PGA, then compares the
resulting voltage against the reference voltage, to generate the digital ouptut values,
The maximum measured RTD voltage could be (0.5mA*400 ohm) = 200mV which is very weak and cannot generates useful signal at the input of the ADC,
this is why we can choose a GAIN = 16, so that we obtain +3.2v signal, we set GAIN[2:0]= 100
- PGA_BYPASS : Disables and bypasses the internal low-noise PGA, The PGA is always enabled for gain settings 8 to 128, regardless of the PGA_BYPASS setting.
We set the bit Enabled by default, PGA_BYPASS = 0

These configurations correspond to the value : 0x08 for register 0.

2) Register 1 (offset = 01h)
This register is used to configure 5 parameters:

-DR[2:0] : controls the data rate at which the device make the conversion (SPS) samples-per-second, we choose the normal mode which correspond to 20 SPS, in addition, At 20 SPS, the digital filter (Hardware section) offers simultaneous 50-Hz and 60-Hz rejection for noisy industrial.
applications. We set DR[2:0] = 000
- MODE[1:0] : These bits control the operating mode the device operates in, we select normal mode, which correspond to 256-kHz modulator clock by default, we set MODE[1:0] =00
- CM : This bit sets the conversion mode for the device, we set CM =1 to select continous conversion mode
- TS : temperature sensor mode, this bit enables the internal temperature sensor and puts the device in
temperature sensor mode, in our case, we will not use it, so TS = 0
- BCS : Burn-out current sources
This bit controls the 10-?A, burn-out current sources, it can be used to detect sensor faults such as wire
breaks and shorted sensors.. we set BCS = 0

These configurations correspond to the value : 0x04 for register 1.

3) Register 2 (offset = 02h)
This register is used to configure 4 parameters :
-VREF[1:0] :Voltage reference selection
These bits select the voltage reference source that is used for the conversion. As we use an external reference voltage across 3.24k ref resistance, connected to REFP0 and REFN0 dedicated inputs, we set VREF[1:0] = 01.
- 50/60[1:0] : FIR filter configuration
These bits configure the filter coefficients for the internal FIR filter.
These bits are used together with the 20-SPS setting in normal mode, we set 50/60[1:0] = 01, which correspond to simultaneous 50-Hz and 60-Hz rejection.
-PSW : Low-side power switch configuration, this bit configures the behavior of the low-side switch connected between AIN3/REFN1 and AVSS (see the hardware section), we set PSW = 0 by default open.
- IDAC[2:0] :IDAC current setting
These bits set the current for both IDAC1 and IDAC2 excitation current sources. The maximum excitation current for an RTD should not exceed 1mA, otherwise the resulted temperature will be biased due to self-heating, we set the both current to be 250uA, which gives a total current of 0.5mA, so we set IDAC[2:0] = 100

These configurations correspond to the value : 0x54 for register 2.

4) Register 3 (offset = 03h)

This register controls 3 parameters :
- I1MUX[2:0] : IDAC1 routing configuration. These bits select the channel where IDAC1 is routed to, following the schematic (Hardware section), IDAC1 is connected to AIN2, so we set I1MUX[2:0] = 011
- I2MUX[2:0] : IDAC2 routing configuration.These bits select the channel where IDAC2 is routed to, following the schematic (Hardware section), IDAC2 is connected to AIN3, so we set I2MUX[2:0] = 100
- DRDYM : DRDY mode. This bit controls the behavior of the DOUT/DRDY pin when new data are ready.
We set DRDYM = 0 , by default, to indicate when data are ready, by using only the dedicated DRDY pin.

These configurations correspond to the value : 0x70 for register 3.

Pseudo-Code (LCD)

In this section, I will explain how the LCD interacts with the microcontroller through I2C protocol,
LCD has 2 internal registers instruction and Data register which can be selected using RS pin.

We use mainly 2 functions : Send Command and Send Data

This pseudo-code highlights the functions used for programming the LCD:

- LCD Pins initialization
- Send a Command to the LCD
- Send a Data to the LCD
- LCD intitialization

1) LCD Pins initialization (after setting I2C)

```
- RS pin : used for register selection , (RS = 0 for Instruction
                                           RS = 1 for Data)

- R/W : Reading from or Writing to LCD  ( R/W = 1 ? Reading
                                          R/W = 0 ? Writing)

- E : Enable pin is set to 1 (for some millisec) when we send Data or commands,
      after transfer we again make it ground EN = 0.
```

- D0 to D7 : 8 bits data pins, used to transfer / read from LCD

Since we are using the ¨PCF8574AT module , which provides I2C port interface for LCD, we use only SDA and SCL pin.

Theory of operation :

PS : We are using “I2C_send (uint8_t slave_addr, uint8_t data)” function as described in I2C_test section, to interact with the LCD,

Each character lcd has 8-bit data , these character can be transferred in 4-bit as well as 8-bit mode.

Since we are using 4-bit mode, each 8-bit character is divded into two 4-bits nibbles, High nibble is sent first following by the lower nibble.
Thus, a 8-bit character needs 2 strocks to be transferred to the LCD.In 4-bit mode character is displayed on lcd in two pulse signals.

The first higher four nibbles of a character are sent to the lcd with an enable stroke (E=1 then E=0) . Than the lower four nibbles are send with enable stroke (E=1 then E=0).

8-bit data LCD are represented as follow : bit7->bit4 for data, bit3->bit0 for configuration

D7	D6	D5	D4	Ret	E	R/W	RS
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

2) Sending a command to the LCD

Supposing we need to send a 8-bit command (CMD) to the LCD in 4-bits mode
CMD = 1011 0001.

- Isolate the highest 4-bits : (CMD & 0XF0) ? 1011 0000 and store the result into temp1 variable.

```
 Temp1 = (CMD & 0xF0).
```

- Include the following configurations bits using OR logic
  RE = 1, E=1, R/w = 0, RS =0 (Write on instruction register)

```
temp2 = ( temp1 | 0x0C) -> 1011 1100  : highest 4-bit command with E enabled.
```

- Include the following configurations bits :
  RE = 1, E=0, R/w = 0, RS =0

```
temp3 = ( temp1 | 0x08) -> 1011 1000  : highest 4-bit command with E disabled.
```

- Send temp2 using I2C-Send() function.
- Send temp3 using I2C_Send() function.

- Shifting the lower 4-bits of the 8-bit command (CMD) 4-bits left, then using a mask to isolate them using AND logic operator

```
Lower-bits = ( (CMD<<4) & 0xF0) -> 0001 0000
```

- Send the Lower-bits with (E = 1) using I2C_Send() :

```
    (Lower-bits | 0x0C)
```

- Send the Lower-bits with (E = 0) using I2C_Send() :

```
    (Lower-bits | 0x08)
```

3) Sending Data to the LCD

We are using the same logic as used in sending a command section, we set the pin RS = 1, since we are writing on Data register.

Supposing we need to send a 8-bit Data to the LCD in 4-bits mode

- Isolate the highest 4-bits : (Data & 0XF0)

```
 Data_High_bits = (Data & 0xF0)
```

- Include the following configurations bits using OR logic
  RE = 1, E=1, R/w = 0, RS =1 (Write on Data register)

```
Data_1 = ( Data_High_bits | 0x0D)  : highest 4-bit Data with E enabled.
```

- Include the following configurations bits :
  RE = 1, E=0, R/w = 0, RS =1

```
Data_2 = ( Data_High_bits | 0x09)  : highest 4-bit command with E disabled.
```

- Send Data_1 using I2C-Send() function.
- Send Data_2 using I2C_Send() function.

- Shifting the lower 4-bits of the 8-bit Data 4-bits left, then using a mask to isolate them using AND logic operator

  Data_Low_bits = ((Data<<4) & 0xF0)

- Send the Lower-bits with (E = 1) using I2C_Send() :

  (Data_Low_bits | 0x0D).

- Send the Lower-bits with (E = 0) using I2C_Send() :

  (Data_Low_bits | 0x09).

4) LCD intitialization:

- Send initialization sequence : send 0x30 command , 3 times with delays

- Send 0x20 command : to select 4-bit mode.

- Send 0x28 command : Set Data length 4 bit-mode, N=1 for 2 line display , F =0 (5x8 characters).

- Send 0x08 command : Display off, Cursor Off, Blink off

- Send 0x01 command : Clear Display.

- Send 0x06 command : Entry mode set -> I/D = 1 (increment cursor)&S = 0(no shift).

- Send 0x0C command : Display on/off control --> D = 1, C and B = 0. (Cursor and blink, last two bits).

Sensor calibration & Uncertainty evaluation

it is a decisive step, because, with the use of a reference standard instrument, it allows to judge the quality of the measurement result of the sensor, through their statistical dispersion.

Following the 17025 ISO STANDARD, The sensor calibration process consists of comparison in several temperature points between the pt100 sensor, and a reference standard with a hight measurement capabilities, determining the precision and reproductibility of the measurement results, which gives a knowledge about the reliability of the pt100 sensor and gives a quality insurance.

After the calibration process, we get the final measurement result which is reprensented as follow :

- Measured_value +/- uncertainty : the uncertainty corresponds to the range of possible values, within which the real value of the measurement lies.

- We get the measurement error which correponds to the difference between the measured value and the real value at each temperature point.

The calibration results are documented on a calibration certificate.

This report highlights the calibration results with the measurement uncertainty, the reference standard, the traceability and the documentation of applied procedure.

PT100 Sensor calibration:

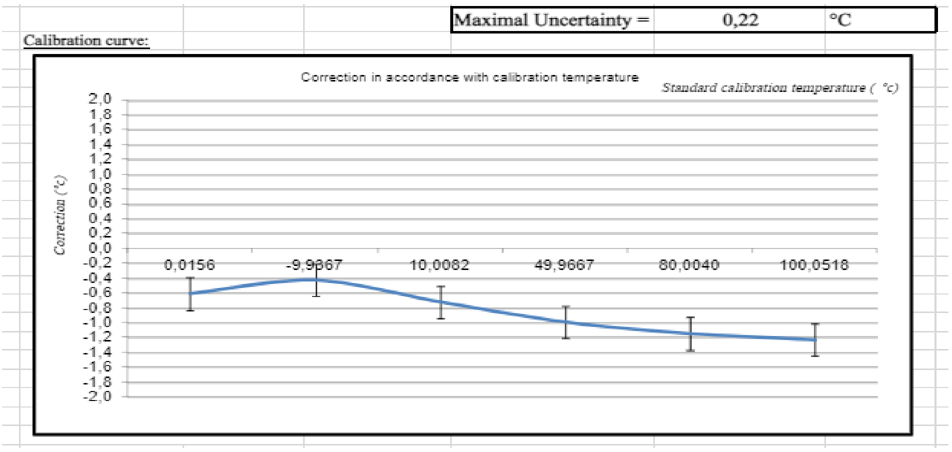
- Measurement Specification :

    Measurement range : -10°C to 100°C
    Calibration points : 0°C/-10°C/+10°C/+50°C/+80°C/+100°C
    Resolution : 0,0001°C
    Acceptable tolerance (Process requirement) : +/- 1.5°C

- Calibration report (Before adjustement)

    Accuracy : 0.5°C for negative temperature points
    The Maximal Error is 1.23°C, reported at +100°C
    Uncertainty : +/- 0.22°C

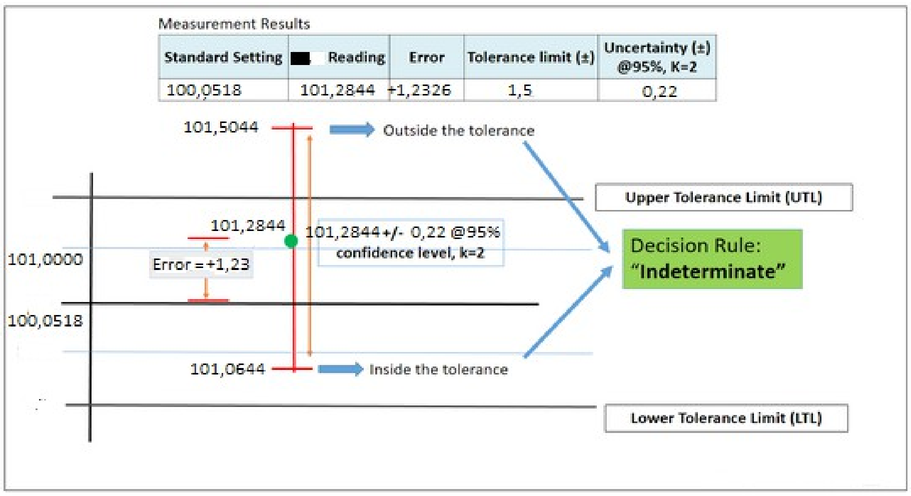
calibration curve :



When using the pt100 sensor at these points,
we should apply the corrections reported in this certificate

T_real = (T_read + Correction) (For each temperature point).

Acceptability within limits tolerance (before adjustement) :


Measurement Results

Standard Setting	[redacted] Reading	Error	Tolerance limit (±)	Uncertainty (±) @95%, K=2
100,0518	101,2844	+1,2326	1,5	0,22

101,5044
Outside the tolerance
Upper Tolerance Limit (UTL)
101,2844
101,2844+/- 0,22 @95% confidence level, k=2
101,0000
Error = +1,23
Decision Rule: "Indeterminate"
100,0518
101,0644
Inside the tolerance
Lower Tolerance Limit (LTL)

As we can see in the schematic,
the pt100 sensor measurement results are acceptable in the whole range within +/-1.5°C limits tolerance.

To achieve an acceptability within a limit tolerance lower than +/- 1.5°C, we need to perform an adjustement, which will give a more accurate result by reducing measurement error.