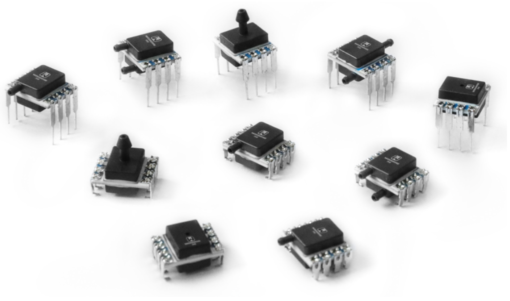
WPS08/04系列压力传感器

产品介绍

WPS08/04 系列是由深圳市沃感科技有限公司自主研发的压阻式高精度高性能硅陶瓷压力传感器，内置 ASIC 对传感器的偏移、温度效应、灵敏度及非线性度进行校准和温度补偿，可在多种压力测量范围和温度范围提供数字和模拟信号输出。此系列传感器可用于差压和表压的测量,可直接安装在标准印刷电路板上使用。此系列压力传感器适用于无腐蚀性、非离子气体（例如空气和其它干燥气体）。所有产品遵循 ISO9001 标准设计制造并经过严格的生产校准，出厂检验确保产品的一致性和可靠性。



特点

- 数字输出/模拟输出
- 工作温度-20~+85℃
- 倒钩状压力接口或无端口
- 精度：±0.25%FSS
- 差压，表压类型
- 睡眠模式
- 总误差带（TEB）：最高±1.0%
- 供电方式：3.3V 或 5V 供电
- 分辨率 24 或 14 位

应用

- 工业自动化
- 泄露测试
- 医疗器械
- 风道静压
- 楼宇空调
- 肺活量计

产品规格

额定参数

SPI 输出

参数	最小值	最大值	单位
电源电压 V_{DD}	-0.3	6	Vdc
数字接口时钟频率:	50	800	kHz
ESD 敏感度（人体模式）		3	kV
存储温度	-40	125	°C
过载压力	2 倍满量程		
爆破压力	3 倍满量程		
焊接时间及温度	波峰焊	250 °C 条件下最长 15 秒	
	回流焊	250 °C 条件下最长 4 秒	

I2C 输出

参数	最小值	最大值	单位
电源电压 V_{DD}	-0.3	6	Vdc
数字接口时钟频率:	100	400	kHz
ESD 敏感度（人体模式）		3	kV
存储温度	-40	125	°C
过载压力	2 倍满量程		
爆破压力	3 倍满量程		
焊接时间及温度	波峰焊	250 °C 条件下最长 15 秒	
	回流焊	250 °C 条件下最长 4 秒	

模拟输出

参数	最小值	最大值	单位
模拟电源电压 V_{DD}	-0.3	6	Vdc
模拟地电压	0	0	V
模拟和数字 IO 管脚电压	-0.3	V_{DD}+0.3	V
ESD 敏感度（人体模式）	-	3	kV
存储温度	-40	125	℃
过载压力	2 倍满量程		
爆破压力	3 倍满量程		
焊接时间及温度	波峰焊	250℃条件下最长 15秒	
	回流焊	250℃条件下最长 4秒	

工作参数

SPI 输出

参数	最小值	典型值	最大值	单位
电源电压： 3.3Vdc 5.0 Vdc 3.3 Vdc 或 5.0 Vdc 具体取决于所选型号	3.0 4.75	3.3 5.0	3.6 5.25	Vdc
电源电流： 3.3 Vdc 电源 5.0 Vdc 电源	- -	1.6 2.0	2.1 3	mA
工作温度范围	-20	-	85	°C
启动时间（从加电到数据准备就绪）	-	2.8	7.3	ms
响应时间	-	0.46	-	ms
低电平 电压	-	-	0.2	V_{DD}
高电平电压	0.8	-	-	V_{DD}
负载电阻	1	4.7	-	kOhm
精度	-0.25	-	0.25	%FSS BFSL
分辨率	-	14	-	位

I2C 输出

参数	最小值	典型值	最大值	单位
电源电压： 3.3 Vdc 5.0 Vdc 3.3 Vdc 或 5.0 Vdc 具体取决于所选型号	3.0 4.75	3.3 5.0	3.6 5.25	Vdc
电源电流： 3.3 Vdc 电源 5.0 Vdc 电源	- -	1.6 2.0	2.1 3	mA
工作温度范围	-40	-	125	°C
启动时间（从加电到数据准备就绪）	-	2.8	7.3	ms
响应时间	-	0.46	-	ms
低电平电压	-	-	0.2	Vsupply
高电平电压	0.8	-	-	Vsupply
SDA 和 SCL 上的负载电阻	1	-	-	kOhm
精度	-0.25	-	0.25	%FSS BFSL
分辨率	-	14	-	位
I2C 默认通讯地址	0X28			

参数	最小值	典型值	最大值	单位
电源电压： 3.3 Vdc	3.0	3.3	3.6	Vdc
电源电流： 3.3 Vdc 电源	-	-	2.0	mA
待机电流（25°C）		-	0.1	mA
工作温度范围	-20	-	85	°C
启动时间（从加电到数据准备就绪）	-	2.5	-	ms
低电平电压	-	-	0.2	V_{DD}
高电平电压	0.8	-	-	V_{DD}
负载电阻	1	4.7	-	kOhm
精度	-0.25	-	0.25	%FSS BFSL
分辨率	-	24	-	位
默认通讯地址		0X78		

模拟输出

参数	最小值	典型值	最大值	单位
电源电压： 5 Vdc 3.3 Vdc	 4.0 3.0	 5.0 3.3	 5.5 3.6	Vdc
平均工作电流： 模拟输出，最小更新率 模拟输出，最大更新率	 - -	 0.6 1.2	 0.8 1.8	mA
上电复位的电平	1.7	-	2.7	V
工作温度范围	-20	-	85	°C
启动时间（从加电到数据输出）	-	-	12	ms
采样率	-	1	-	KHZ
低电平电压	-	-	0.2	V_{DD}
高电平电压	0.8	1	-	V_{DD}
负载电容	0	1	15	nF
精度	-0.25	-	0.25	%FSS BFSL

注意：

- 该传感器没有反向极性保护。将错误的引脚与电源连接或者接地可能会导致电气故障。
- 补偿温度范围是指传感器可以在特定的性能限制下产生与压力成比例的输出的温度范围。
- 工作温度范围是指传感器可以产生与压力成比例的输出的温度范围，但不一定在特定性能限制范围之内。
- 精度：相对适用于在 25°C 时的压力范围内所测输出的最佳直线 (BFSL) 的最大输出偏差。包括所有因压力非线性、压力迟滞性和不可重复性造成的误差。
- 总误差带（TEB）：相对整个补偿温度和压力范围内理想传递函数的最大偏差。包括所有因零点、满量程、压力非线性、压力迟滞性、不可重复性、热零点偏移、热量程偏移和热迟滞性造成的误差。
- 满量程 (FSS) 是指在压力范围最大限制值 (Pmax.) 和最小限制值 (Pmin.) 处测得的输出信号之间的代数差。
- 过载压力：可安全施加到产品的最大压力，使产品在压力返回到工作压力范围时规格保持不变。施加过高的压力可能会对产品造成永久损坏。除非另有规定，否则这适用于工作温度范围内任何温度下的所有可用压力口。
- 爆破压力：可施加到产品的任意压力口而不造成压力媒介脱离的最大压力。在经受任何超过爆破压力的压力之后，产品将不能正常工作。

环境规格

参数	特性
湿度：仅干燥气体	0% 到 95% RH
寿命	至少为 100 万次循环

- 寿命可能因传感器使用的特定应用而有所变化。

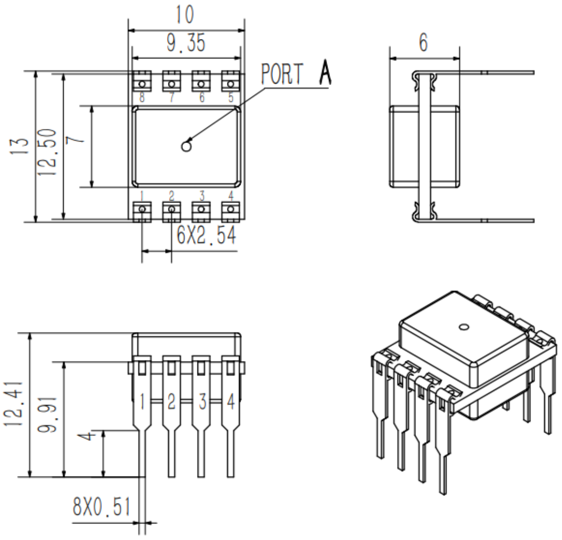
脚位定义

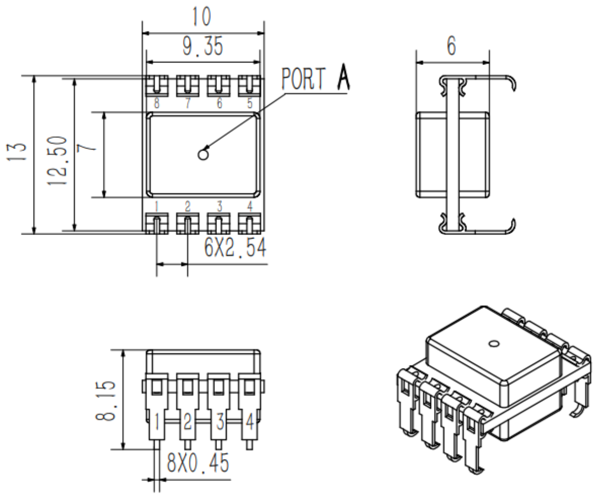
输出类型	引脚 1	引脚 2	引脚 3	引脚 4	引脚 5	引脚 6	引脚 7	引脚 8
I2C	GND	V_{DD}	SDA	SCL	NC	NC	NC	NC
SPI	GND	V_{DD}	MISO	SCLK	SS	NC	NC	NC
模拟输出	NC	电源	信号	地	NC	NC	NC	NC

结构参数

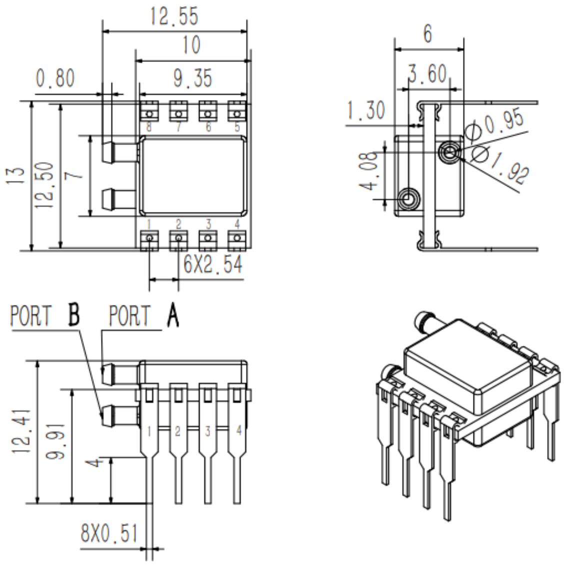
单位：mm

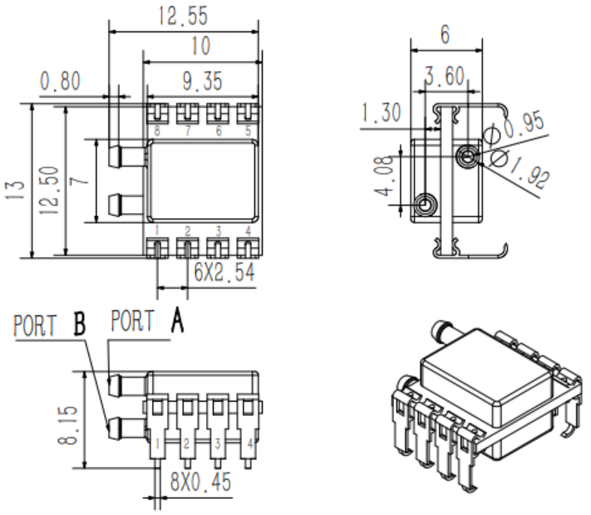
NN：无端口



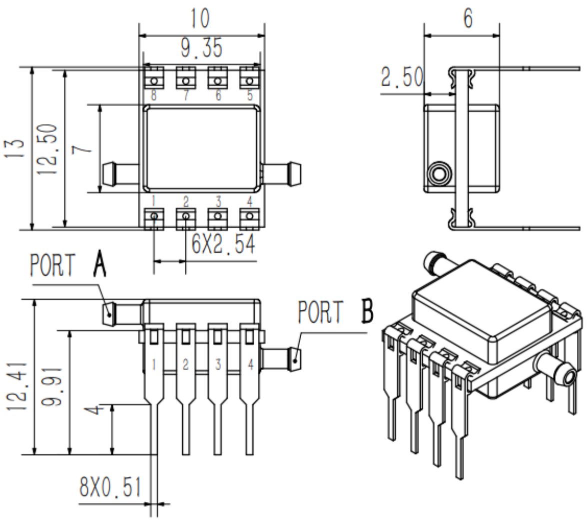


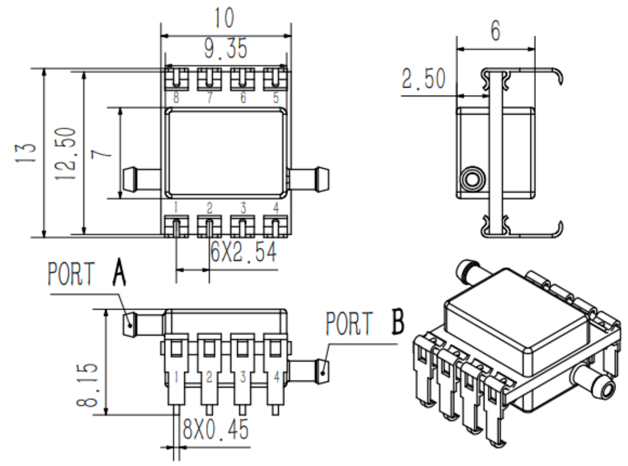
RR：双径向带刺端口，同侧



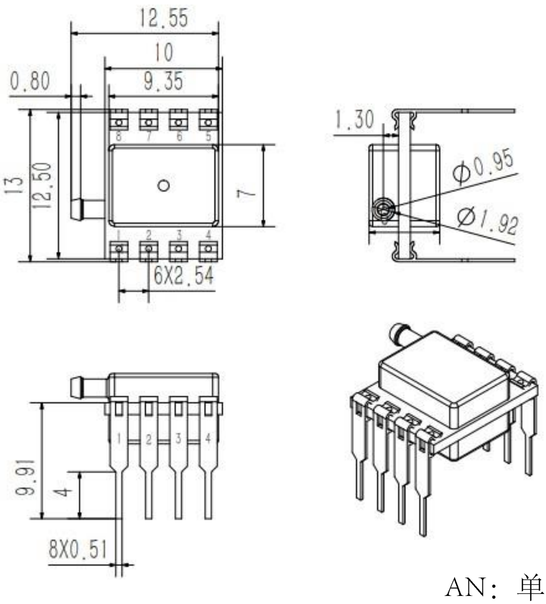


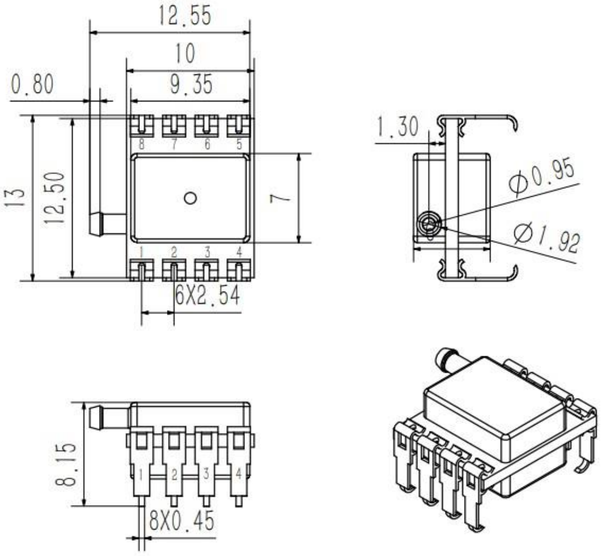
DR：双径向带刺端口，对侧



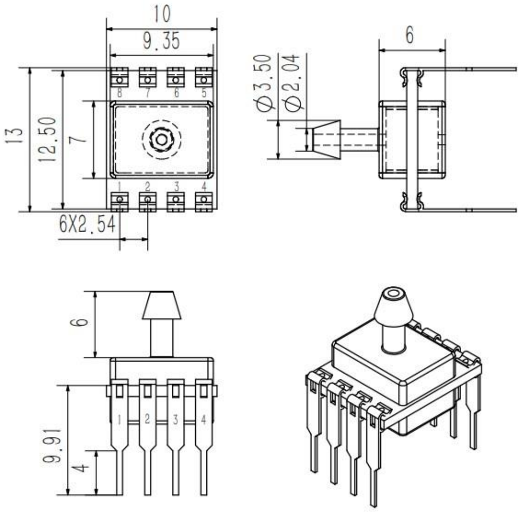


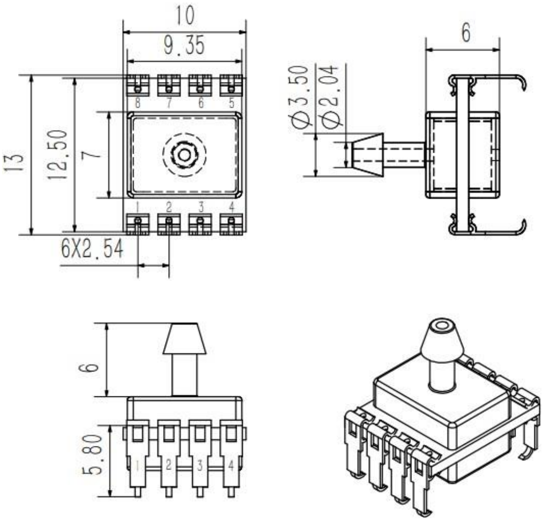
RN：单径向带刺端口





AN：单轴向带刺端口





订购信息

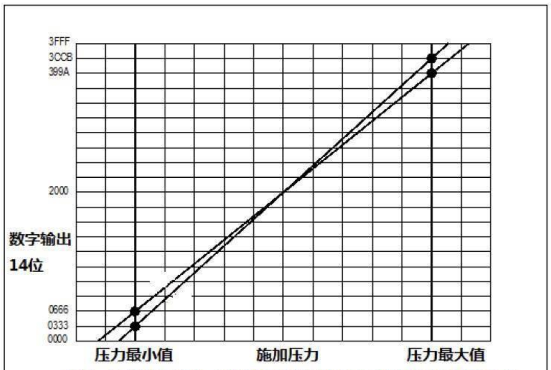
例如:	WPS	08D	006K	G	05	A	NN	3	H
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

序号	具体意义	详细描述
1	产品识别码	WPS
2	封装	04 S: 4 PIN　SIP（单列直插） 08 D: 8 PIN　DIP（双列直插） 08 M: 8 PIN　SMT（表面贴装）
3	压力范围	004K(4KPA) 006K(6KPA) 010K(10KPA) 016K(16KPA) 025K(25KPA) 040K(40KPA) 050K(50KPA) 060K(60KPA) 100K(100KPA) 160K(160KPA) 250K(250KPA) 400K(400KPA) 600K(600KPA) 700K(700KPA) 001G(1MPA) 注：当产品的量程在此表内未体现出来时按以下示例进行扩充 KPA 的产品均为---K 示例：005K 代表 5KPA PA 的产品均 P 示例：010P 代表 10PA
4	压力类型	G：表压 D：差压
5	TEB范围	10：±1.0%FS 15：±1.5%FS 20：±2.0%FS
6	输出类型	A：模拟电压输出 C：I2C 输出14位 D：SPI 输出 I：I2C输出24位
7	压力端口	NN：无端口 AN：单轴向带刺端口 RN：单径向带刺端口 RR：双径向带刺端口，同侧 DR：双径向带刺端口，对侧
8	供电电压	3：3.3VDC 5：5.0VDC
9	补偿温度	H：-20 C°~85C° M：0 C°~85C° L：0 C°~50C°

压力和温度输出对应公式

I2C/SPI 输出（分辨率14位）

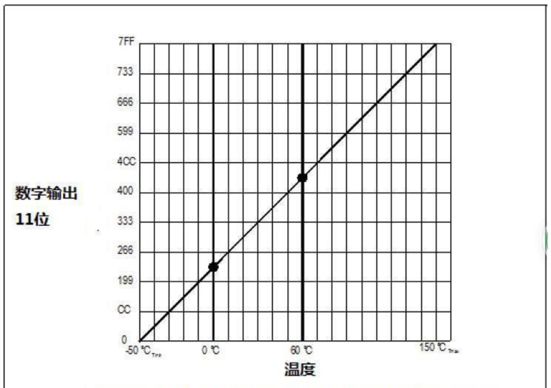
压力转换方程



A型：压力数字输出=80%*16383/（压力最大值-压力最小值）*（施加压力-压力最小值）+10%*16383

B型：将上述公式80%改为90%，10%改为5%

温度转换方程



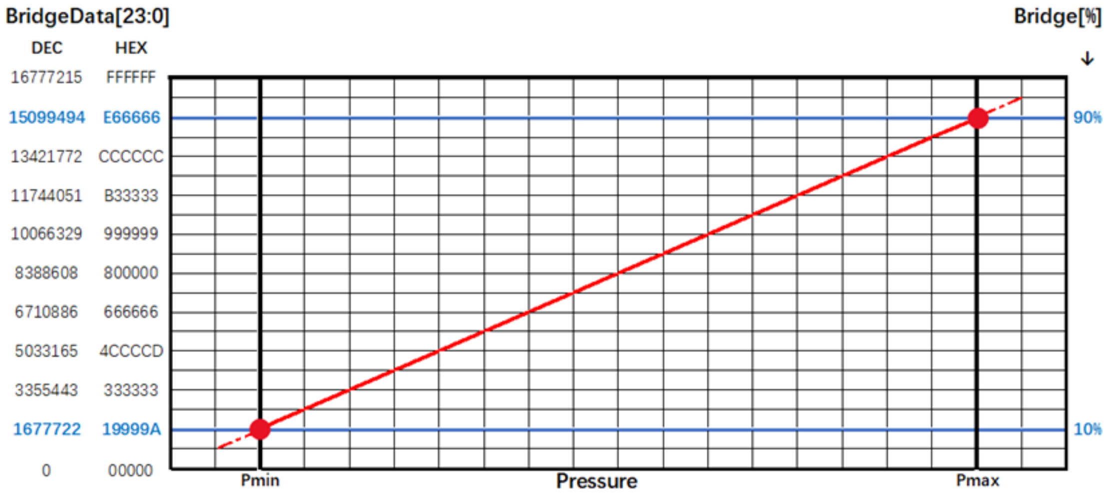
温度数字输出=（待测温度-（-50））*2047/(150-(-50))

高百分比的传感器输出

输出百分比	数字计数（十进制）
0	0
10	1638
50	8192
90	14745
100	16383

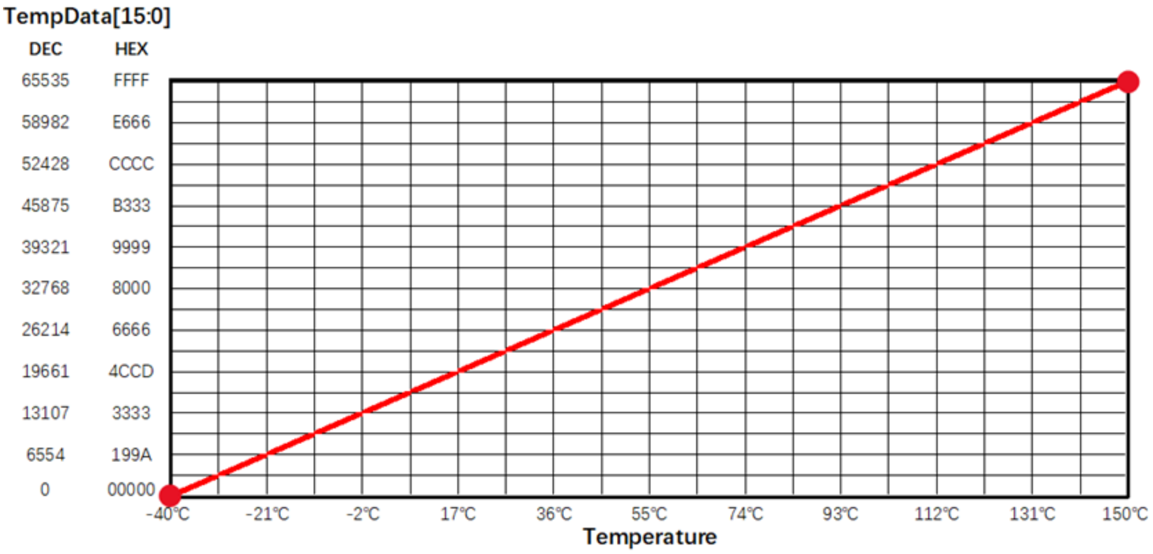
I2C 输出（分辨率24位）

压力转换方程



$$Pressure=(BridgeData[23:0]-10\%*16777215)*\frac{Pmax-Pmin}{80\%*16777215}+Pmin$$

温度转换方程

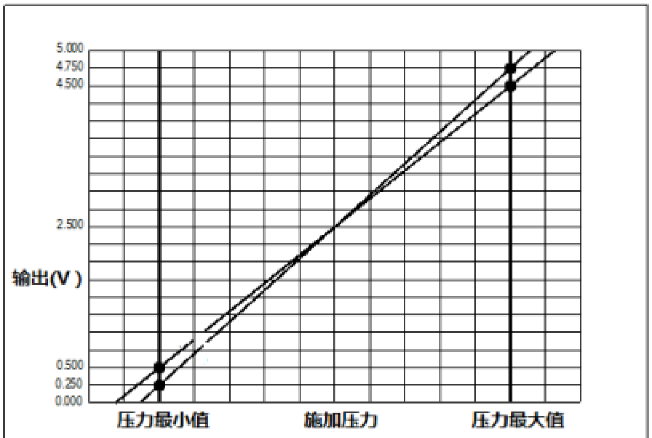


$$\text{Temperature}=\frac{\text{TempData[15:0]}}{65535}*190-40$$

高百分比的传感器输出

输出百分比	数字计数（十进制）
0	0
10	1677722
50	8388608
90	15099494
100	16777216

模拟输出



A型：输出电压=80%*输入电压/（压力最大值-压力最小值）*（施加压力-压力最小值）+0.500

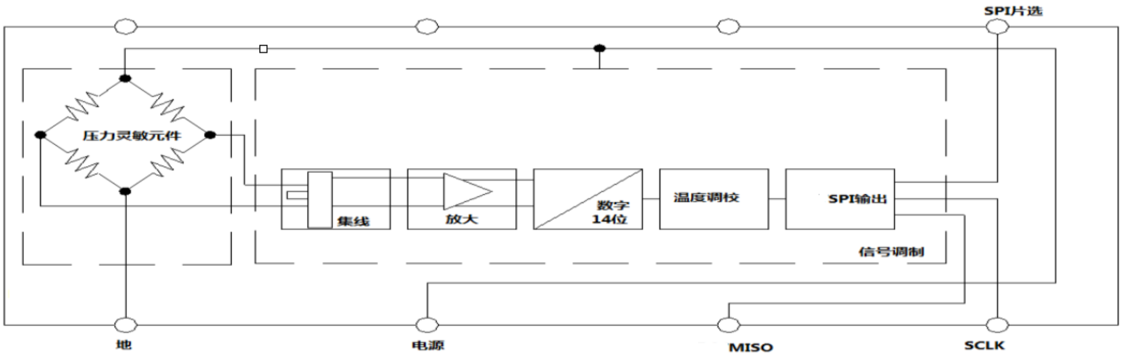
B型：将上述公式80%改为90%，0.500改为0.250

高百分比的传感器输出

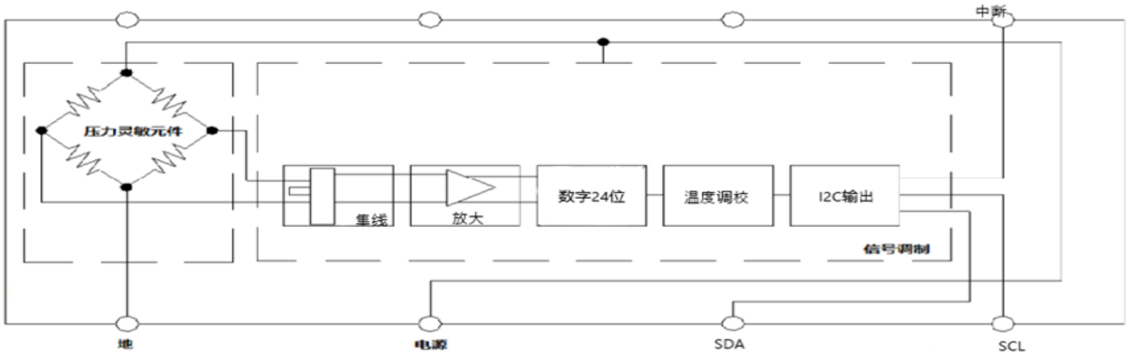
输出百分比	模拟放大（5V）
0	0
5	0.25
10	0.5
50	2.5
90	4.5
95	4.75
100	5

等效电路

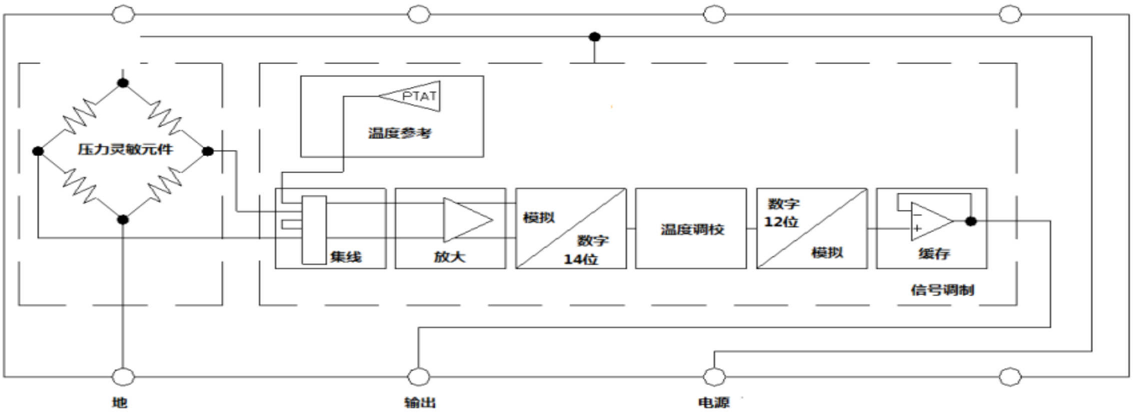
SPI 输出



I2C 输出



模拟输出



注意

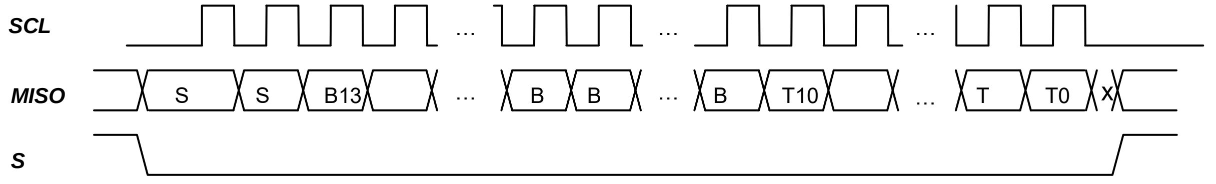
- 建议将传感器的压力口 A 朝下放置，这样系统中的颗粒就不容易进入并停留在压力传感器内。
- 规格参数如有更改，不再另行通知。
- 除特殊说明外，产品均为压力转换方程A 型传感器。
- 有关更多信息请联系沃感销售人员。

SPI 输出

SPI Read_DF(数据读取)

为了简化解释和说明，以下部分只说明下降沿 SPI 极性。SPI 接口在 SCLK 下降沿后会有数据变化。这里以 MISO 为例介绍 SCLK 的上升。整个输出数据包是 4 个字节(32 位)。高桥数据字节排在前面，低桥数据字节排在后面。然后发送 11 位修正温度 (T[10:0]):首先是 T[10:3]字节，然后是[T[2:0]， xxxxx)字节。最后一个字节的最后 5 位是未知的，应该在应用程序中屏蔽掉。如果用户只需要正确的压力值，则可以在第二个字节之后终止读取。如果还需要修正的温度，但只需要 8 位分辨率，则可以在读取 3dbyte 后终止读取。

SPI Output Packet with Falling Edge SPI_Polarity



Packet = [{S(1:0),B(13:8)}, {B(7:0)}, {T(10:3)},{T(2:0),xxxxx}] Where
S(1:0) = Status bits of packet (normal, command, busy, diagnostic) B(13:8) = Upper 6 bits of 14-bit bridge data.
B(7:0) = Lower 8 bits of 14-bit bridge data.
T(10:3) = Corrected temperature data (if application does not require corrected temperature, terminate read early)

应用示例

C code example for SPI with Read_DF4 command

ReadWithSPI.c

```
/*
ReadWithSPI.c reads the digital output simply at any time and be assured the data is no older than the selected response time specification by checking the status of the 2 MSBs of the bridge high byte data */


 /*PB0 = SCLK*/
 /*PB1 = MISO*/
 /*PB2 = SS*/
 #include —iom164p.h‖
 #define DF2        2
 #define DF3        3
 #define DF4        4
 unsigned char bufptr[4];
 void Init(void)
{
/* P0 = SCLK – output */
/* P1 = MISO – input */
/* P2 = SS – output */
/* P7, P6, P5, P4, P3, P2, P1, P0 */
/* O O O O O O I O */
/* 1 1 1 1 1 1 1 1 */
 DDRB = 0xfd;
 PORTB = 0xfc;
 }
voidBitDelay(void)
{
char delay; delay = 0x03;
 do

{
```

```
while(--delay) ;
_NOP();
return;
}
unsigned char GetOneByte (void)
{
unsigned char data=0;
  unsigned char i;
 for (i=0; i<8; i++)
{
 BitDelay();
  SCLK=1;
  BitDelay();
data=data<<1;
if (PINB & 0x02)
  data=data | 1;
  SCLK=0;
BitDelay()
}
return (data);
}
unsigned char ReadSA191D(unsigned char DF_Command)
{
  unsingned char i;
  SCLk=0;
  SS=0;
 BitDelay();
for (i=0; i<(DF_Command); i++)
{
   bufptr[i] = GetOneByte ();            /* 1 byte of read sequence */

}
SS=1;
BitDelay();
}
void main (void)
```

```
{
float Pressure, Temperature; unsigned
int Dpressure,Dtemperature;

 float P1= 819.15;          /* P1= 5% * 16383 – B type*/

 float P2= 15563.85;        /* P2= 95% *16383 – B type*/
 float Pmax= 2.0;
 float Pmin= -2.0;
 Init();
 do
 {
   ReadSA191D (DF4);        /*Read_DF4 command – data fetch 4 bytes */
   If((bufptr [0] & 0xc0)==0)            /*test status of the 2 MSBs of the bridge high byte of data*/
 {
    Dpressure= ((unsigned int) (bufptr [0] & 0x3f) <<8) + (bufptr [1]);
    Dtemperature= (((unsigned int) bufptr [2]) <<3) + bufptr [3];
    Pressure= (((float) Dpressure)-P1) * (Pmax-Pmin) / P2+Pmin;
    Temperature= ((float) Dtemperature) * 200 / 2047-50;
```

I2C 输出（24位分辨率）

使用 0xAC 命令利用传感器内部校准算法获取校准值的说明

发送 0xAC 命令获取校准值的步骤如下：

一、发写命令

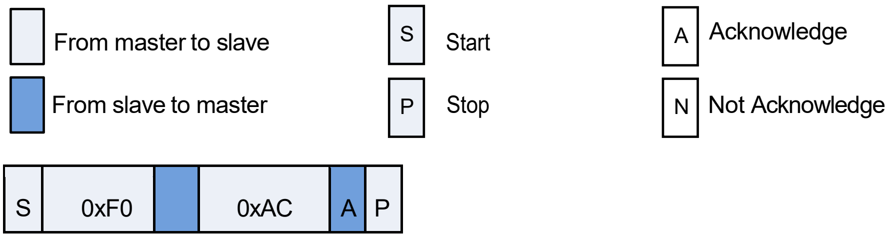


图 1 写命令

写命令中的 0xF0 表示默认的 7bits I2C 设备地址为 0x78，最后 1bit 为 0 表示主设备写操作。

二、等待

发送完写命令后需要等待一段时间再发送读命令，因为内部完成整个测量需要一段时间。等待的时长取决于OTP(Address: 0x14)的[13:11]电桥过采样率和 OTP(Address: 0x14)的[15:14]温度过采样率的设置。对照附录的表 1 和表 2，等待的时长=tP+tT。 等待的时长不需要计算，可以通过不断的读 IIC 状态字的方式判断出是否采集已经完成。

三、读数

要保证写指令和读指令的时间间隔大于测量的时长才能够读出校准数据，读数格式如图 2 所示，读命令中的 0xF1 表示默认的 7bits I2C 设备地址为 0x78， 最后 1bit 为 1 表示主设备读操作。读到的校准数据共 6 个字节，依次为 1 字节状态字，3 字节电桥校准值，2 字节温度校准值。

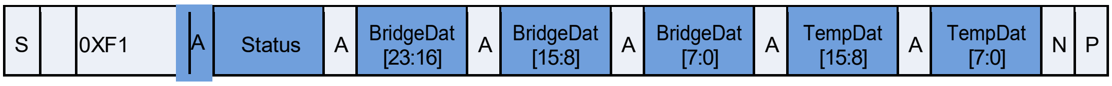


图 2 I2C 读出 5 字节校准后的压力和温度值

四、换算

读到校准数据后，需要将以 AD 值形式表示的无符号数迚行简单的换算。 为方便理解我们假设读到的校准数据为：

0x04 0x9B 0xB0 0xC5 0x56 0xAA

0x04 为状态字 Bit5 为 1 表明最近一次 I2C 忙，需要等待一段时间。如果 Bit5 为 0 表明设备非忙，可以读取数据。关于状态字各比特的详细描述请参见附录。

0x9B 0xB0 0xC5 三个字节为电桥校准值

0x56 0xAA 两个字节为温度校准值

电桥校准值换算：将 0x9B 0xB0 0xC5 转换为十迚制数为 10203333,

本次计算假设校准时使用的量程为 20Kpa-120Kpa，对应的 AD 输出为 1677722~15099494（10%AD~90%AD）

根据输入输出关系校准公式得到：

实际压力值=（120-20）/（15099494-1677722）*（10203333-1677722）+20=83.5208 Kpa

温度校准值换算：将 0x56 0xAA 转换为十进制数为 22186，由于读取到的校准数据是以百分比形式表示的，这个百分比在数值上等于我们换算得到的十进制数与16bits 无符号数的最大值（65535）之比，所以在换算百分比时可进行如下计算

22186/65536*100%=33.85%

温度的校准范围规定为-40℃—150℃ 所以校准值=（150—（-40））*33.85%—40=24.32℃

附录

表 1 压力过采样率和测量时间对照表

OSR_Pressure[13:11]（二进制）	对应的过采样率	测量时间 tP(ms)
000	32768	203
001	16384	105
010	8192	56
011	4096	31
100	2048	19
101	1024	13
110	512	10

表 2　温度过采样率和测量时间对照表

OSR_Temperature[15:14]（二进制）	对应的过采样率	测量时间 tT(ms)
00	2048	19
01	4096	31
10	8192	56
11	16384	105

表 3 Status 字节比特位描述

比特位	意　义	描　述
Bit7	保留	固定为 0
Bit6	上电指示 (Power indication)	1 设备上电（VDDB on）； 0 设备掉电
Bit5	忙闲指示(Busy indication)	1 设备忙，表明最近一次 I2C 命令所要求读取的数据还未有效。如果设备忙，新的命令将不被处理。0 表明最近一次 I2C 命令所要求读 取的数据已经准备好被读取
Bit4	保留	固定为 0
Bit[3]	工作状态 (Mode Status)	0 NOR mode 1 CMD mode
Bit2	存储器数据完整性指示 (Memory integrity/error flag)	0 表示 OTP 存储器数据完整性测试 (CRC) 通过， 1 表示完整性测试失败。对数据完整性的测试只在上电过程中(POR)计算一次，所以被写入的新 CRC 值只能在接下来的 POR 之后使用。
Bit1	保留	固定为 0
Bit0	保留	固定为 0

应用示例

```
/* ****************************************************************************** *

 * WPS_IIC.c
 * Date: 20XX/XX/XX
 * Revision: 1.0.0
 *
 * Usage: IIC read and write interface
 * ****************************************************************************/
#include "WPS_IIC.h"
//IIC clock line sbit
SCL = P1 ^ 1;
//IIC data line sbit
SDA = P1 ^ 0;
//Set the input and output mode of IIC data pin
#define  Set_SDA_INPUT() \
    P1MDOUT   &=   0xFE;   \
    P1 |= 0X01
#define Set_SDA_OUTPUT() P1MDOUT |=
0x01;
////Delay function needs to be defined void
DelayUs(unsigned char i) {
}
  //Start signal
  void
  Start(void) {
   SDA = 1;
   DelayUs(2);
   SCL = 1;
   DelayUs(2);
   SDA = 0;
   DelayUs(2);
   SCL = 0;
}
   //Stop signal
   void Stop(void)
   {
```

```
    Set_SDA_OUTPUT(
    ); SDA = 0;
    DelayUs(2);
    SCL = 1;
    DelayUs(2);
    SDA = 1;
    DelayUs(2);
}
//Read ACK signal
unsigned char Check_ACK(void)
{
      unsigned char ack;
     Set_SDA_INPUT();
    SCL = 1;
    DelayUs(2);
    ack  =  SDA;
    SCL = 0;
    Set_SDA_OUTPUT(
    ); return ack;
}
//Send ACK signal
void Send_ACK(void)
{
Set_SDA_OUTPUT(
); SDA = 0;
DelayUs(2);
SCL = 1; DelayUs(2);
SCL = 0; DelayUs(2);
}
//Send one byte
void SendByte(unsigned char
byte) {
unsigned char i = 0;
Set_SDA_OUTPUT(
); do
{
 if (byte & 0x80)
 {
              SDA = 1;
 }
 else
```

```
                {
                          SDA = 0;
                }
                DelayUs(2);
                SCL = 1;
                DelayUs(2);
                byte <<= 1;
                i++;
                SCL = 0;
}
    } while (i < 8);
    SCL = 0;
  //Receive one byte
  unsigned char
  ReceiveByte(void) {
  unsigned char i = 0, tmp = 0;
  Set_SDA_INPUT();
  do
  {
                  tmp <<= 1;
                  SCL = 1;
                  DelayUs(2);
                  if (SDA)
                  {
                            tmp |= 1;
                  }
                  SCL = 0;
                  DelayUs(2)
                  ; i++;
      } while (i < 8);
      return tmp;
  }
  //Write a byte of data through IIC
  uint8 BSP_IIC_Write(uint8 address, uint8 *buf, uint8
  count) {
     unsigned char timeout,
     ack; address &= 0xFE;
     Start();
     DelayUs(2);
     SendByte(address);
     Set_SDA_INPUT();
     DelayUs(2);
     timeout = 0;
```

```
        do
        {
                 ack =
                 Check_ACK();
                 timeout++;
                 if (timeout == 10) {

                              Stop();
                              return 1;

                 }
    } while (ack);
    while (count)

      {

                SendByte(*buf);
                Set_SDA_INPUT(
                ); DelayUs(2);

                  timeout = 0;
                 do
                 {
         ack = Check_ACK();
         timeout++;
         if (timeout == 10)
         {
                    return 2;

                                 }
                      } while (0);
                      buf++;
                      count--;
          }
          Stop();
          return 0;

      }

       //Read a byte of data through IIC
       uint8 BSP_IIC_Read(uint8 address, uint8 *buf, uint8
       count) {

       unsigned char timeout,
       ack; address |= 0x01;
       Start();
       SendByte(address);
       Set_SDA_INPUT();
       DelayUs(2);
       timeout = 0;
       do
       {
```

```
              ack =
              Check_ACK();
              timeout++;
              if (timeout == 4)
              {
                        Stop();
                        return 1;
              }
     } while (ack);
     DelayUs(2);
    while (count)
     {
              *buf = ReceiveByte();
              if (count != 1)
                        Send_ACK();
                  buf++;
                  count--;
        }
         Stop();
        return 0;
     }
      /* ***************************************************************************** *
          * WPS_IIC.h
           * Date: 20XX/XX/XX
         * Revision: 1.0.0
     *
    * Usage: IIC read and write interface
     * *****************************************************************************/#ifndef
     WPS_IIC_H_
     #define    WPS_IIC_H_
          uint8 BSP_IIC_Write(uint8 IIC_Address, uint8 *buffer, uint8
        count); uint8 BSP_IIC_Read(uint8 IIC_Address, uint8 *buffer,
                                  uint8 count);
    #endif
    /* ***************************************************************************** *
    * WPS.c
     * Date: 20XX/XX/XX
    * Revision: 1.0.0 *
    * Usage: Sensor Driver file for WPS
    * *****************************************************************************/
    #include "WPS.h"
    #include "WPS_IIC.h"
```

```
  // Define the upper and lower limits of the calibration pressure
  #define PMIN 20000.0  //Full range pressure for example 20Kpa
  #define PMAX 120000.0  //Zero Point Pressure Value, for example 120Kpa
  #define DMIN 3355443.0  //AD value corresponding to pressure zero, for example 20%AD=2^24*0.2
  #define DMAX 13421772.0  //AD Value Corresponding to Full Pressure Range, for example 80%AD=2^24*0.8

    //The 7-bit IIC address of the JHM1200 is 0x78
    uint8 Device_Address = 0x78 << 1;

    //Delay function needs to be defined
    void DelayMs(uint8 count)
    {
    }

    //Read the status of IIC and judge whether IIC is busy
    uint8 WPS_IsBusy(void)
    {
      uint8 status;
      BSP_IIC_Read(Device_Address, &status,
      1); status = (status >> 5) & 0x01;
      return status;
    }


 /**
  * @brief Using the 0xAC command to calculate the actual pressure and temperature using the WPS internal
algorithm
  * @note  Send 0xAC, read IIC status until IIC is not busy
  * @note  The returned data is a total of six bytes, in order: status word, three-byte pressure value, two-byte
temperature value
  * @note  The returned three-byte pressure value is proportional to the 24-bit maximum value 16777216.
According to this ratio,

            the actual pressure value is again converted according to the calibration range.
  * @note  The returned two-byte temperature value is proportional to the 16-bit maximum value 65536.
According to this ratio,
            the actual pressure value is again converted according to the calibration range.
  * @note  Zero pressure point and full pressure point of calibration pressure correspond to 20kpa and
120Kpa, respectively
  * @note  The zero point of the calibration temperature is -40°C and the full point is 150°C
  * @note  The pressure actual value is calculated according to the span pressure unit is Pa, temperature
actual value temp unit is 0.01°C
  */
 void
 WPS_get_cal(void) {
```

```
    uint8 buffer[6] = {0};
    uint32 Dtest = 0;
    uint16 temp_raw = 0;
    double pressure = 0.0, temp = 0.0;

    //Send 0xAC command and read the returned six-byte data
    buffer[0] = 0xAC;
    BSP_IIC_Write(Device_Address, buffer, 1);

                 if
    (WPS_IsBusy())
    DelayMs(5);
                 {
    while (1)
    {

                              DelayMs(1);
                 }
                 else
                              break;
      }
      BSP_IIC_Read(Device_Address, buffer,
      6);
    //The returned pressure and temperature values are converted into actual values according to the
  calibration range
   Dtest = ((uint32)buffer[1] << 16) | ((uint16)buffer[2] << 8) | buffer[3];
   temp_raw = ((uint16)buffer[4] << 8) | (buffer[5] << 0);
   pressure = (PMAX-PMIN)/(DMAX-DMIN)*(Dtest-DMIN)+PMIN; temp =
   (double)temp_raw / 65536;
   temp = temp * 19000 - 4000;
  }
/* ******************************************************************************* *

* File : WPS.h
 *

 * Date : 20XX/XX/XX
 *
* Revision : 1.0.0 *

* Usage: Sensor Driver for WPS sensor
 * ******************************************************************************/#ifndef
 WPS_H
 #define    WPS_H__
                       __

 void WPS_get_cal(void);

 #endif
```

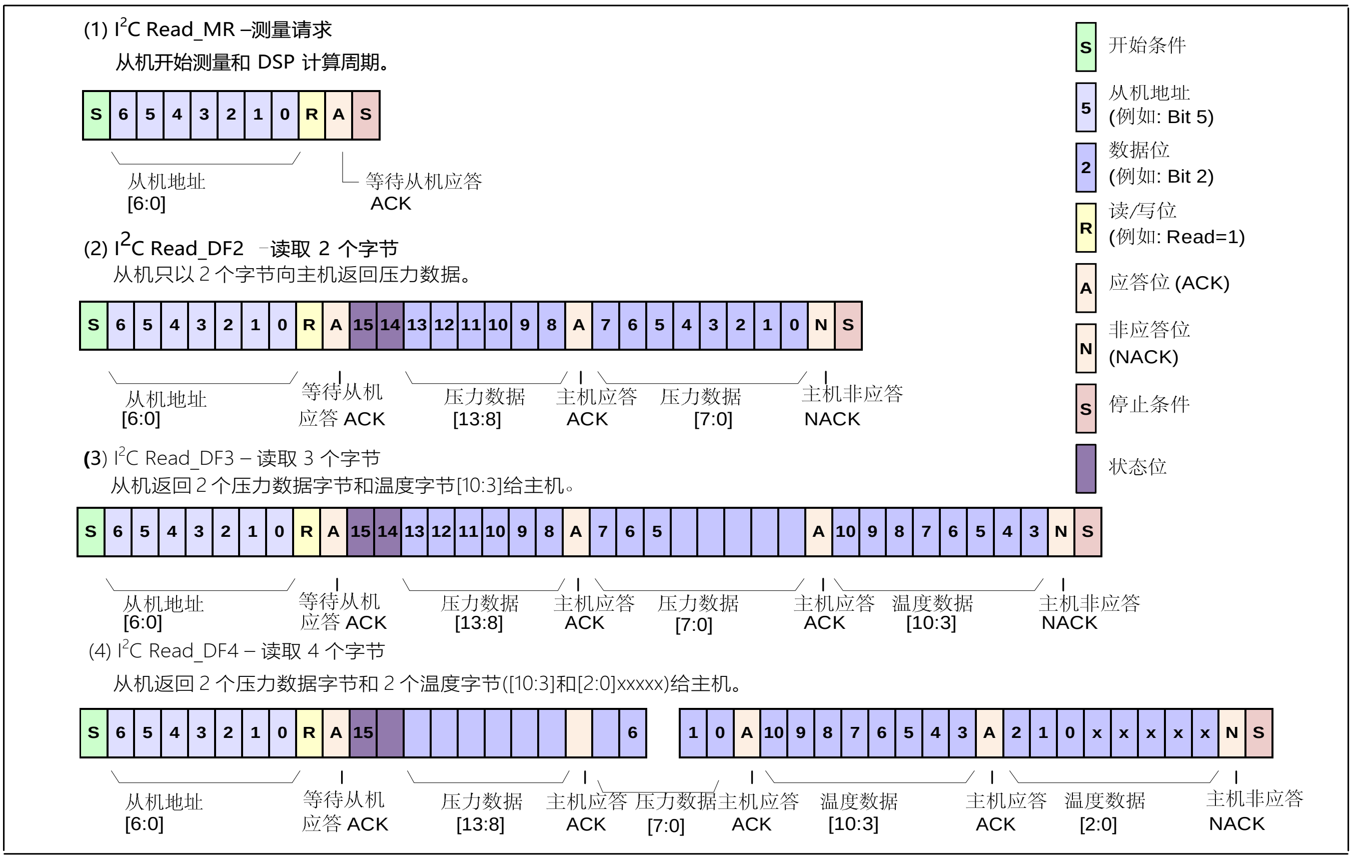
I2C 输出（14位分辨率）

读取操作

- 主机发送 7 位的 I2C 地址，第 8 位为 1(读)。作为从动装置的传感器会发送一个应答(ACK)来表示成功
- 传感器有 4 条 I2C 读取命令:READ_MR、READ_DF2、READ_DF3、READ_DF4。下图显示了四个 I2C 读命令中的 3 个测量数据包结构，下面将进一步解释。
- 对于 READ_DF3 数据获取命令（数据获取 3 个字节)，传感器返回 3 个字节:两个字节的数据，两个状态位作为最高有效位(MSBs)，然后一个字节的温度数据(8 位精度)。在接收到所需的数据字节数后，主机发送非应答（NACK）和停止条件来终止读操作。
- 对于 READ_DF4 命令，主机延迟发送非应答(NACK)，并继续读取额外的最后一个字节，以获得完全校正的 11 位温度测量值。在这种情况下，数据包最后一个字节的最后 5 位是不确定的，应该在应用程序中屏掉。
- 如果不需要校正温度，则使用 READ_DF2 命令。主机在两个字节的数据之后终止读取操作。

I2CREAD(数据读取):

对于数据获取命令，传感器返回的数据字节数由主机发送非应答（NACK） 和停止条件决定。



(1) I^2C Read_MR –测量请求

从机开始测量和 DSP 计算周期。

(2) I^2C Read_DF2 –读取 2 个字节

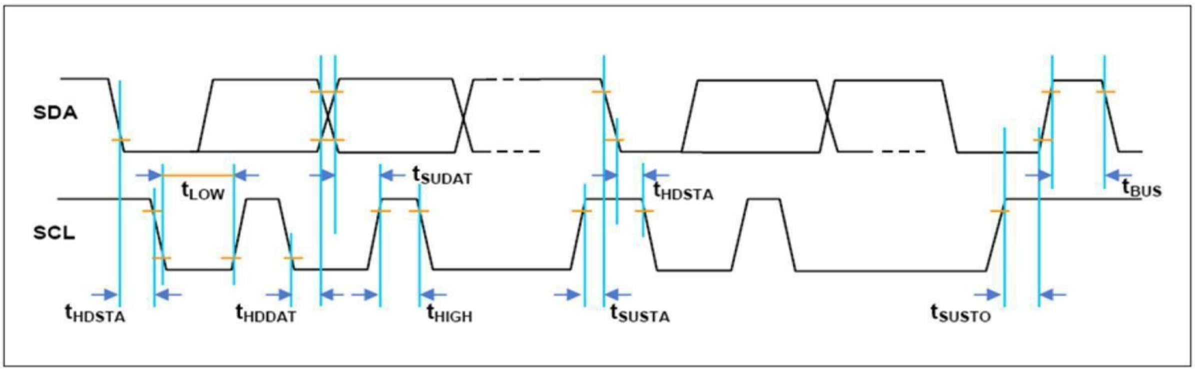
从机只以 2 个字节向主机返回压力数据。

(3) I^2C Read_DF3 – 读取 3 个字节

从机返回 2 个压力数据字节和温度字节[10:3]给主机。

(4) I^2C Read_DF4 – 读取 4 个字节

从机返回 2 个压力数据字节和 2 个温度字节([10:3]和[2:0]xxxxx)给主机。



参数	符号	MIN	TYP	MAX	UNITS
SCL 时钟频率	FSCL	100		400	kHz
启动条件在 SCL 边缘的保持时间	tHDSTA	0.1			µs
时钟低电平宽度	tLOW	0.6			µs
时钟高电平宽度	tHIGH	0.6			µs
启动条件在 SCL 边缘的保持时间	tSUSTA	0.1			µs
SDA 数据在 SCL 边缘的保持时间	tHIDDAT	0.0			µs
SDA 数据在 SCL 边缘的准备时间	tSUDAT	0.1			µs
SCL 上停止条件的准备时间	tSUSTO	0.1			µs
停止条件和启动条件之间的总线空闲时间	tBUS	2.0			µs

传感器 I2C 协议与通用 I2C 协议的差异

- 注意:传感器协议与通用 I2C 协议有三个不同之处
- 启停条件—在 CLK 线路上没有变化(中间没有时钟脉冲)会为下一次通信造成通信错误，即使下一次启动条件是正确的且有时钟脉冲。必须发送一个额外的启动条件，以恢复正常的通信。
- 重启条件—数据传输过程中当 CLK 时钟线处于高电平时， SDA 边缘下降也会造成通信失败，必须发送一个额外的启动条件才能进行正确的通信。
- 开始条件不允许第一个 SCL 边缘上升，同时 SDA 边缘下降。如果使用第一个位为 0 的 I2C 地址，SDA 必须从起始条件到第一个位保持在低电平。

诊断功能-状态位

- 该传感器具有诊断功能，确保系统运行稳定。诊断状态是通过 2 个最高有效位(MSBs)的高字节数据的状态传输来表示的。

状态位(2 个最高有效位（MSBs） 的输出包)	说明
00	运行正常，数据包良好
01	命令模式下的设备(不适用于正常操作)
10*	过期数据:自上一个测量周期以来已经获取的数据
11	诊断条件的存在

注*:如果在上电复位后，在第一次测量之前或第一次测量期间进行数据取回，则返回“stale”，但该数据实际上是无效的，因为第一次测量还没有完成。

- 当两个最高有效位（MSBs)为 11 时，会显示以下故障之一;
  - 无效的 EEPROM 签名
  - 桥的正或负的丢失
  - 桥输入短
  - 桥源损耗
- 所有的诊断在下一个测量周期检测到，并在随后的数据采集中报告。一旦诊断被报告，诊断状态位将不会改变， 除非诊断的原因被固定和电源上电复位被执行。

睡眠模式

- 在睡眠模式下，在命令窗口之后，传感器将断电，直到主机发送一个 Read_MR 命令。 Read_MR将唤醒传感器并开始一个测量周期。 如果命令为 Read_MR，该部件执行温度、自动归零(AZ)和桥接测量，然后进DSP校正计算。 有效值被写入数字输出寄存器，传感器再次关机

- 在一个测量序列之后，在下一个测量可以被执行之前，主机必须发送一个 Read_DF 命令，它将获取 2、3 或 4 字节的数据，而不唤醒传感器。当执行 Read_DF 时，返回的数据包将是最后一次测量，状态位设置为“valid。在 Read_DF 完成后，状态位将被设置“stale”。 下一个 Read_MR 将再次唤醒该部件，并开始一个新的测量周期。 如果在测量周期仍在进行时发送了 Read_DF，则数据包的状态位将被读取为“stale”。

注意:I2C™Read_MR 函数也可以使用 I2C™Read_DF2 或 Read_DF3 命令来完成，并且忽略的“stale”数据将被恢复。

I2C使用Read_DF4命令的C代码示例

上电时，内部上拉关闭，PORTB初始化为所有输入，外部上拉将SDA和SCL线拉高，PORTB输出锁存位SCL和SDA初始化为零。例程WriteSDA和WriteSCL根据参数“state”的值切换各自的数据方向位。当状态为“1”时，端口引脚被配置为输入(外部引上拉高)。当状态为“O”时，端口引脚被配置为输出，锁存器驱动引脚低。WriteSDA和WriteSCL是非常简单的例程，可以将它们合并到各自的调用例程中，从而进一步减少代码的大小。

General Calling Sequence for the Routines

```
SendStartBit();          /*start*/
SendByte(byt e);         /*send address or command MSB
GetOneByte();            first*/ /*read one byte from serial
SendStop();              stream */ /*stop*/
```

PORTB on the ATmega164P is used to communicate with SA18D transducer. Bit assignments are as follows:

```
I2C.c
/*PB0 =SDA*/
/*PB1 = SCL*/

#include “i2c.h”
void WriteSCL(unsigned char state)
{
 if (state)
     DDRB &= 0xfd;          /* input ... pullup will pull high or Slave will drive low */
 else
     DDRB |= 0x02;                 /* output ... port latch will drive low */

}

void WriteSDA(unsigned char state)
```

```
  {
     if (state)
       DDRB &= 0xfe;              /* input ... pullup will pull high or Slave will drive low */

    else
       DDRB |= 0x01;                      /* output ... port latch will drive low */

  }

  unsigned char SetSCLHigh(void)

  {

     WriteSCL(1);                         /* release SCL*/

     /* set up timer counter 0 for timeout */

     t0_timed_out = FALSE;                /* will be set after approximately 34 us */

     TCNT0 = 0;                   /* clear counter */

     TCCR0 = 1;                   /* ck/1 .. enable counting */

     /* wait till SCL goes to a 1 */

     while (! (PINB & 0x02) && !t0_timed_out) ;

     TCCR0 = 0;                   /* stop the counter clock */

     return(t0_timed_out);

}

void BitDelay(void)

{

  char delay;

  delay = 0x03;

  do

  {

   _NOP();

  } while (--delay);

}

/* Routine SendStopBit generates an TWI stop bit assumes SCL is low stop bit is a 0 to 1 transition on SDA while
SCL is high
           ______________

       /

SCL  ___/
                 _______

                /
```

```
  SDA _________/
  */
void SendStopBit(void)
{
  WriteSDA(0);
  BitDelay();
  SetSCLHigh(
  ); BitDelay();
  WriteSDA(1);
  BitDelay();
}
/* Routine SendStartBit generates an start bit start bit is a 1 to 0 transition on SDA while SCL is high
          _____________
         /
SCL ___/
     ___________
                  \
SDA   \_____
*/
void SendStartBit(void)
 {
  WriteSDA(1);
  BitDelay();
  SetSCLHigh()
  ; BitDelay();
  WriteSDA(0);
  BitDelay();
  WriteSCL(0);
  BitDelay();
}
unsigned char SendByte(unsigned char byte)
{
  unsigned char i;
  unsigned char error;
  for (i = 0; i < 8; i++)
```

```
{
    WriteSDA(byte & 0x80);                 /* if > 0 SDA will be a
    byte = byte << 1;                 1 */  /* send each bit */
   BitDelay();
   SetSCLHigh();
   BitDelay();
   WriteSCL(0);
   BitDelay();
  }
  /* now for an ack */
  /* Master generates clock pulse for ACK */
  WriteSDA(1);                          /* release SDA ... listen for ACK */
  BitDelay();
  SetSCLHigh                          /* ACK should be stable ... data not allowed to change when SCL is
(); high */
   /* SDA at 0 ?*/
    error = (PINB & 0x01);              /* ack didn't happen if bit 0 = 1 */
   WriteSCL(0);
    BitDelay();
    return(error);
}
unsigned char GetOneByte(unsigned char lastbyte)
{
  /* lastbyte ==1 for last byte */
 unsingned char i;
 unsigned char data;
  DDRB &=0xfe;    /* release SDA ... listen for slave output */
  data=0;
  for (i=0; i<8;i++)
  {
    SetSCLHigh() ;                      /* Slave output should be stable ... data not allowed to change when
SCL is high  */
```

```
     BitDelay();
    data=data<<1;
     if (PINB &0x01)
      data=data |1;
     WriteSCL(0);
    BitDelay();
}
      /*send ACK*/
      WriteSDA (lastbyte); /* no ack on last byte ... lastbyte = 1 for the lastbyte */
      BitDelay(); SetSCLHigh();
       BitDelay();
       WriteSCL(0);
       BitDelay();
       WriteSDA(1) ;
       BitDelay();
      return (data);
 }
```

ReadWithPollingI2C.c

```
 /*
ReadWithPollingI2C.c reads the digital output simply at any time and be assured the data is no older than the selected response time specification by checking the status of the 2 MSBs of the bridge high byte data
*/
#include “i2c.h”
extern unsigned char GetOneByte(unsigned char lastbyte);
extern unsigned char SendByte(unsigned char byte);
extern void SendStartBit(void);
extern void SendStopBit(void);
extern void BitDelay(void);
extern unsigned char SetSCLHigh(void);
extern void WriteSDA(unsigned char state);
extern void WriteSCL(unsigned char state);
```

```
unsigned char SA181DO_Address;

unsigned char bufptr[4];

void Init (void)


{

   __disable_interrupt();

   /* P0 = SDA - bidirectional */ /

   * P1 = SCL - output */

   /* P7, P6, P5, P4, P3, P2, P1,

   P0 */ /* O O O O O O O O */ /*

   1 1 1 1 1 1 1 1 */

   DDRB = 0xff;

   PORTB = 0xfc;

  /*setup SA181DO device address*/

   SA181DO_Address=0x28;

   /*

   The factory setting for I2C slave address is 0x28, 0x36 or 0x46 depending on the interface type
selected from the ordering information.

   For this sample code, 0x28 is used for Slave address of SA181DO.

    */

 }

 unsigned char ReadSA181DO(unsigned char DF_Command)

 {
     unsigned char i;
  unsigned char error;

  SendStartBit();

  if (SendByte((SA181DO_Address<<1) +re {ad))                    /*send salve address byte*/

   {
    return (1);    /*check error*/
    }

    for (i=0; i< (DF_Command-1); i++)

    {

   bufptr[i] =GetOneByte (0);                      /* 1 byte of read sequence */
```

```
  }
  bufptr[DF_Command-1] = GetOneByte(1);              /* 1 signals last byte of read sequence */

 SendStopBit();

 return (0);

}

 void main (void)
{
   float Pressure,Temperature;

     unsigned int Dpressure, Dtemperature;

     float P1=819.15;                   /* P1= 5% * 16383 – A type*/

     float P2=15563.85;                 /* P2= 95% * 16383 – A type*/

     float Pmax=2.0;

      float Pmin=-2.0

Init();

do

 {

 ReadSA181DO (DF4);    /*Read_DF4 command – data fetch 4 bytes */

 If ((bufptr [0] & 0xc0) ==0x00)/*test status of the 2 MSBs of the bridge high byte of data*/

 {

 Dpressure= ((unsigned int) (bufptr [0] & 0x3f) <<8) + (bufptr [1]);

 Dtemperature= (((unsigned int) bufptr [2]) <<3) + bufptr [3];



 Pressure= (((float) Dpressure)-P1) * (Pmax-Pmin) / P2+Pmin;

 Temperature= ((float) Dtemperature) * 200 / 2047 -50;

   }

  }

  while(1);
```

```
}  /* main */
```

I2C.h

```
#include “iom164p.h”

#define    DF2    2
#define    DF3    3
#define     DF4   4

#define write 0
 #define read 1
```