Technical Note

008371
Issue 1

ABP3 Board Mount Pressure Sensor
SPI Communication Protocol

SPI BUS configuration

The Serial Peripheral Interface (SPI) is a simple bus system for synchronous serial communication between one Master and one or more Sensors. It operates either in full-duplex or half-duplex mode, allowing communication to occur in either both

directions simultaneously, or in one direction only. The Master device initiates an information transfer on the bus and generates clock and control signals. Sensor devices are controlled by the Master through individual Sensors Select (SS) lines and are active only when selected. Both the ABP2 Series SPI and the ABP3 Series SPI sensors operate in full-duplex mode only, with data transfer from the Sensors to the Master. This data transmission uses four, unidirectional bus lines. The Master controls SCLK, MOSI and SS; the sensor controls MISO. The ABP3 Series ASIC uses an SPI slave interface to provide direct access to its entire memory map. An external SPI master (e.g., microcontroller) initiates all transactions. Data is transferred in 16-bit words, MSB first, and framed by the NCS (chip select) signal:

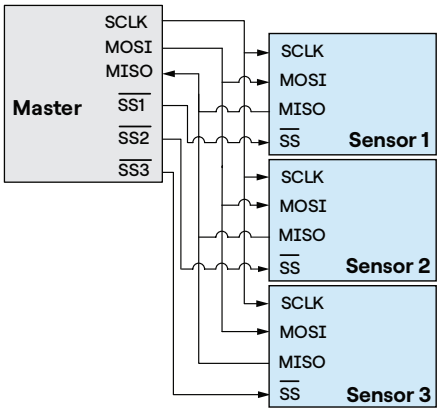
- Start: NCS goes LOW
- End: NCS goes HIGH

The SPI interface operates in Mode 0 (Clock Polarity (CPOL)= 0, Clock Phase (CPHA)= 0), where the clock idles LOW and data is sampled on the rising edge of SCLK. This ensures proper synchronization between the master and sensor during communication. The interface supports CRC-protected and unprotected frames for robust communication.

The existing ABP2 Sensor SPI interface is command driven protocol in which the microcontroller explicitly starts measurements and later retrieves compensated pressure and temperature data through status polling or the EOC signal. It provides basic status monitoring but does not implement frame level CRC protection.

In contrast the ABP3 sensors SPI interface is memory-mapped register-based architecture that allows direct access to read the pressure values, temperature values and diagnostic information. It supports SPI clock frequencies up to 10MHz and incorporates CRC4/CRC8 communication protection, advanced configurable SPI timing, and extensive diagnostic status reporting, which makes the ABP3 SPI communication more robust and automotive-oriented communication interface which intern makes it significantly more advanced than the existing ABP2 SPI communication.

Data transmission lines
SCLK: Signal clock
MOSI: Master out/sensor in
MISO: Master in/sensor out
SS: Sensor select



SPI data transfer

To communicate with the ABP3 Series SPI sensors de-assert the Sensor Select (SS/NCS) line. At this point, the sensor is no longer idle and will begin sending data once a clock is received. ABP3 Series SPI sensors are configured for SPI operation in Mode 0 (Clock Polarity is 0 and Clock Phase is 0). The ABP3 Series SPI sensors are designed to respond to requests from a Master device.

SPI command format
Mode: single shot mode – (standard digital listing)

The ABP3 Series can be configured at the factory to function in three different operating modes: single-shot, periodic, and continuous.

- **Single-shot (default mode):** Sensor will perform a single measurement and update its output register whenever an external trigger is provided. The device remains active only during data acquisition and then stays in idle mode to minimize power consumption. The maximum update rate is 4 kHz
- **Periodic:** Sensor performs periodic measurements based on the sample rate preloaded at the factory; maximum update rate is 4 kHz
- **Continuous:** Sensor performs continuous measurements, based on the sample rate preloaded at the factory, never leaving the power state; maximum update rate is 50 kHz

Note: For custom update rates ranging from 0.5k to 50k samples per second, as well as technical support for periodic mode and continuous mode, please contact Honeywell.

Honeywell Technologies

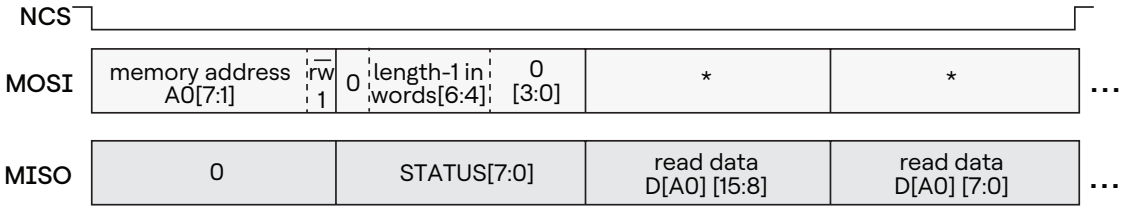
Steps to read temperature, pressure, and status output

The SPI slave interface provides access to read Pressure data, temperature and status register from the memory map of the ABP3 sensor. An external SPI master (e.g. a µC) can read three 16-bit words starting at address 0×2E. This will read the temperature, corrected sensor signal values and the corresponding (synchronized) STATUS bits.

Random read with SPI: Reads data from the consecutive memory address of the device starting at the specified memory address. All read/writes start at word aligned addresses (i.e. LSB of the memory address equals 0) and read an even number of bytes.

Note: In SPI communication, when CRC is enabled, the system accepts only CRC-protected frames, ensuring data integrity. When CRC is disabled, the system accepts any incoming data frame regardless of CRC validation.

Random read



Random read without CRC

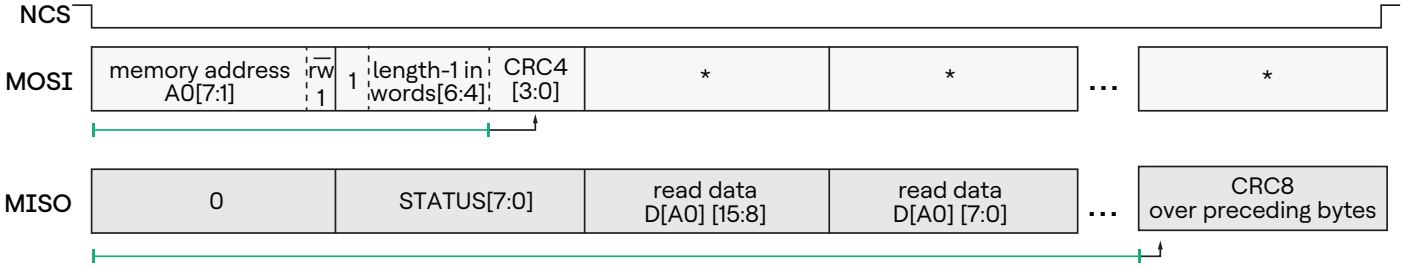
Byte #	0	1	2	3	4	5	6	7
MOSI	0×2F	0×20	*	*	*	*	*	*
MOSI Comment	Memory address 0×2E +1 with LSB signal read bit set	0 010 0000 0: Frame without CRC 010: length -1 = 3 Words 0000: Should be Zero	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
MISO	0×00	0×1E	0×7D	0xF2	0×82	0×EA	0×00	0×1E
MISO Comment	—	—	0×2E: DSP_T [15:8]	0×2E: DSP_T [7:0]	0×30: DSP_S [15:8]	0×30: DSP_S [7:0]	0×32: STATUS_SYNC [15:8]	0×32: STATUS_SYNC [7:0]

To read data is transferred MSB first, high byte before low byte will be the data order.

1. MOSI (Master Out Slave in):
   The command sent by the master device consists of bytes transmitted on the MOSI line. The first byte is used to indicate the address to read from, with the LSB marked to signal a read operation.
   The Command Format for without CRC:
   - Byte 0: Address (0×2E) plus an increment for reading operation
   - Byte 1: Other control bits (data length). ([7:4] = Number of words to read back – 1, [3:0] = 0)

   The Command Format for with CRC:
   - Byte 0: Address (0×2E) plus an increment for reading operation hence 0×2F
   - Byte 1: Other control bits (CRC info, data length). ([7] = 1(frame with CRC), [6:4] = Number of words to read back – 1, [3:0] = 1100(CRC4))
2. MISO (Master in Slave Out):
   After sending the command via MOSI, the slave device responds on the MISO line, refer the below table for the expected output format.

Random read



Legend

8 bit of data sent by SPI master

8 bit of data sent by device

Random read with CRC

Byte #	0	1	2	3	4	5	6	7	8
MOSI	0×2F	0xAC	*	*	*	*	*	*	*
MOSI Comment	Memory address 0×2E +1 with LSB signal read bit set	1 010 1100 0: Frame without CRC 010: length -1 = 3 Words 1100: CRC4	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
MISO	0×00	0X1E	0×7D	0XF2	0×82	0XEA	0×00	0X1E	0X28
MISO Comment	—	—	0×2E: DSP_T [15:8]	0×2E: DSP_T [7:0]	0×30: DSP_S [15:8]	0×30: DSP_S [7:0]	0×32: STATUS_ SYNC [15:8]	0×32: STATUS_ SYNC [7:0]	CRC8 check-sum

The **memory address field** sets the byte address of the first memory location to be read. Only word (16 bit) aligned read are supported, thus the LSB of the memory address field is not part of the frame.

CRC calculation for SPI

The ABP3 sensor uses CRC4 and CRC8 for data verification with the following parameters

Polynomial CRC4: 0×03 ($x4 + x1 + x0$)
Initialization Value: 0xF
CRC4 is calculated over the **header bytes only** – (Byte0 and Byte1 before CRC bits are added)
Polynomial CRC8: 0×07 ($x8 + x2 + x1 + x0$)
Initial Value: 0xF3
CRC8 is calculated over the **entire transmitted frame before CRC8** – (Byte 0, Byte 1(with CRC4) and data bytes)
Input: All the data bytes received from the slave output: 8 -bit checksum Byte

Algorithm process:

1. Start with CRC = 0xF3
2. For each data byte: XOR the byte with current CRC
3. Shift 8 bits to the left
4. If MSB (bit 7) was 1 after XOR: XOR with Polynomial 0×07
5. Repeat until all bits processed

The SPI random read procedure is common for reading any register in the device, and it holds good for registers like 0×2E as well as 0×30. Register 0×2E is used when temperature, corrected pressure, and STATUS_SYNC data are required together, whereas register 0×30 can be used when only the corrected pressure output is needed. To read only pressure, a random read is initiated starting at address 0×30 by sending 0×31 as the address byte to indicate a read operation. Since pressure is a single 16-bit value, the number of words to read back is configured as one, so the control byte is set to 0×00. After the command is sent, the slave returns the pressure data on the MISO line. The first data byte corresponds to pressure bits [15:8], followed by the second byte containing pressure bits [7:0]. This method allows efficient reading of only the required pressure data without accessing temperature or status registers.

Data acquisition in single shot mode and periodic/ continuous mode

In periodic and continuous modes, the sensor automatically performs measurements at the configured interval without any external trigger. For every update cycle, the sensor acquires raw data, converts it to digital form, and updates the corrected temperature and pressure registers, such as 0×2E and 0×30. In these modes, the host can directly read the required registers using the standard random read method, since the data is always refreshed internally. In contrast, in single-shot mode the sensor remains in sleep or idle state to minimize power consumption. A measurement is performed only when an external trigger is issued by the host. This trigger is generated by writing the START command (0×8B93) into the command register 0×22. Once this command is written, the sensor performs a single measurement, after which the host can wait for the required conversion time and then read the corrected pressure and temperature data from registers 0×30 and 0×2E using the same read mechanism.

Writing Start command to CMD register without CRC:

Byte 0: Address (0×22).

Byte 1: Other control bits (data length). ([7:4] = Number of words to read back – 1, [3:0] = 0)

Byte 2 and Byte 2: Start Command to write.

Byte #	0	1	2	3
MOSI	0×22	0×00	0×8B	0×93

(without CRC Protection)

Writing the START command to the CMD register with CRC:

Byte 0: Address (0×22).

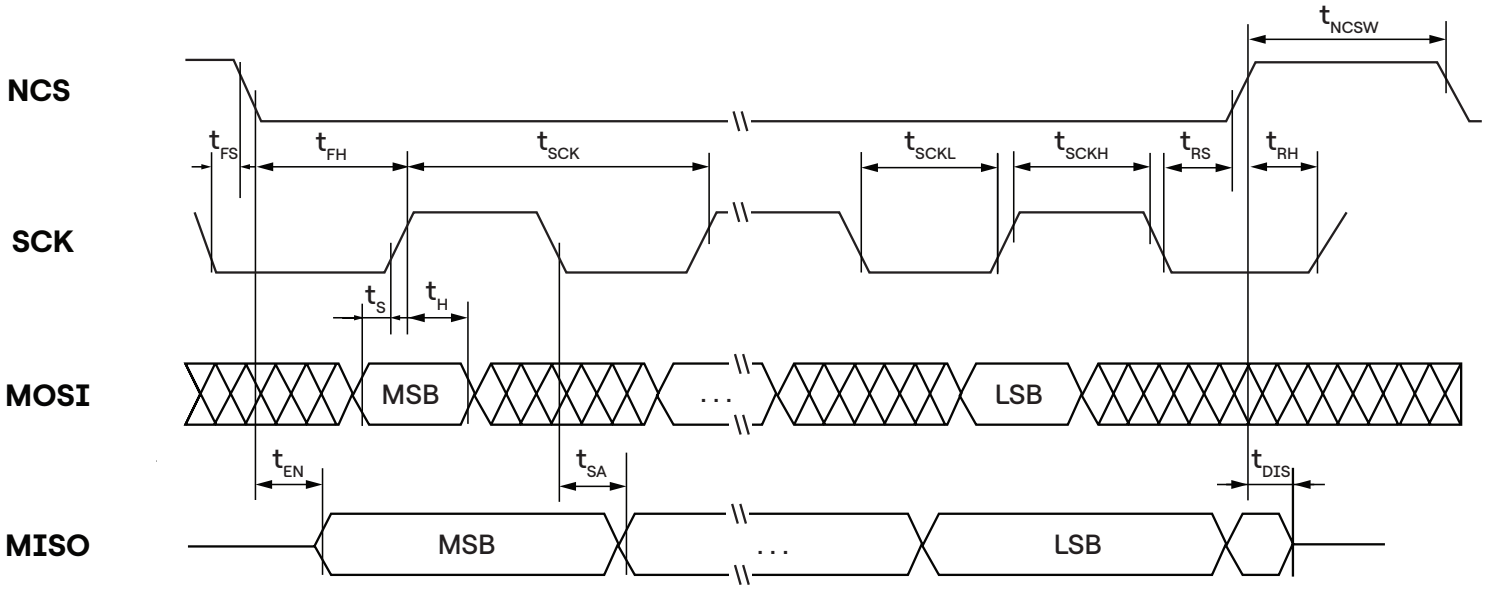
Byte 1: Other control bits (data length). (CRC info, data length). ([7] = 1(frame with CRC), [6:4] = Number of words to read back – 1, [3:0] = 1110(CRC4)).

Byte 2 and Byte 2: Start Command to write. Then the Byte 4 to be 0×3E.

Byte #	0	1	2	3
MOSI	0×22	0×8E	0×93	0×3E

(with CRC Protection)

SPI timing parameters



Note: SPI interface timing diagram with 1st SCK edge sample and 2nd SCK edge shift (for CFG_SPI_I2C.spi_cpha = 0, CFG_SPI_I2C.spi_spol = 0, CFG_SPI_I2C.spi_ncspol = 0, CFG_SPI_I2C.spi_early_miso = 0)

No.	Description	Condition	Symbol	Min.	Typ.	Max.	Unit
1	SPI clock frequency ($1/t_{sck}$)*)	—	SPI_F_SCK	–	–	10	MHz
2	SPI clock low time (t_{sckl})*)	—	SPI_T_SCKL	35	–	–	ns
3	SPI clock high time (t_{sckh})*)	—	SPI_T_SCKH	35	–	–	ns
4	Time between two SPI frames (t_{NCSW})*)	—	SPI_T_NCSW	1000	–	–	ns
5	MOSI setup time (t_S)*)	—	SPI_T_S	10	–	–	ns
6	MOSI hold time (t_H)*)	—	SPI_T_N	10	–	–	ns
7	NCS to MISO drive (t_{EN})*)	—	SPI_T_EN	–	–	40	ns
8	NCS to MISO high-Z (t_{DIS})*)	—	SPI_T_DIS	–	–	40	ns
9	SCK to MISO time (t_{SA})*)	CFG_SPI_I2C.spi_miso_strong = 1	SPI_T_SA	–	–	40	ns
10	SCK setup time vs. falling edge of NCS (t_{FS})*)	—	SPI_T_FS	40	–	–	ns
11	SCK hold time vs. falling edge of NCS (t_{FH})*)	—	SPI_T_FH	40	–	–	ns
12	SCK setup time vs. rising edge of NCS (t_{RS})*)	—	SPI_T_RS	40	–	–	ns
13	SCK hold time vs. rising edge of NCS (t_{RH})*)	—	SPI_T_RN	40	–	–	ns

*) Not tested in production

Status registers

When 6 bytes of data are read after writing the 0×2E register, the last 2 bytes will be the STATUS_SYNC register. These two bytes are copied from the STATUS register.

0×32(STATUS SYNC)
STATUS_SYNC BYTES EXPLANATION

Bit (Meaning)	Status
15 (dsp_t_missed)	1 = when dsp_t_up was 1 and DSP_T updated again
14 (dsp_s_missed)	1 = when dsp_s_up was 1 and DSP_S updated again
13	always 0
12	always 0
11 (com_crc_error)	1 = communication CRC error occurs
10 (dsp_sat)	1 = will be set 1 when DSP_T/DSP_S saturation occurs to prevent overflow
9	always 0
8 (bc_fail)	1 = will be set to 1 when bridge check failure occurs
7 (bs_fail)	1 = will be set to 1 when bridge supply failure occurs
6	always 0
5	always 0
4 (dsp_t_up)	1 = will be set to 1 when DSP_T register is updated with new value
3 (dsp_s_up)	1 = will be set to 1 when DSP_S register is updated with new value
2	always 0
1	always 0
0 (idle)	1 = chip is in busy state 0 = chip is in idle state

HW_DIAG (0×34)

Bit	Status
15	Confirms **validated bridge supply fault**
14	Indicates **non-volatile memory access failure**
[13:0]	Always 0

The bits STATUS_SYNC [15:5,0] are identical to the bits STATUS [15:5,0].

The bits STATUS_SYNC [4:3] are copied from the STATUS register when the corresponding DSP registers are read. First reading the DSP registers and then STATUS_SYNC ensures that both values are consistent to each other.

Data conversion

The above-mentioned pressure and temperature counts are in unsigned 16-bit format. The user needs to convert the counts to a 16-bit signed format for proper interpretation of the output as per datasheet specifications.

Equation 1: Unsigned counts to signed counts

Please follow the following steps to suitably convert the data into signed format.

1) If the received 16-bit unsigned data is greater than 32767 (i.e., 2^{15} - 1), this must be converted to a 16-bit signed value using Equation 1(a) below.
2) If the received 16-bit unsigned data is less than or equal to 32767 (i.e., 2^{15} - 1), then the corresponding 16-bit signed counts are equal to the received unsigned count as in Equation 1(b).

Equation 1(a):
If Unsigned counts > 32767:
Signed Counts = ((Unsigned Counts + 2^{15}) % 2^{16}) - 2^{15}

Equation 1(b):
If Unsigned counts ≤ 32767:
Signed Counts = Unsigned Counts

Example: Calculate the pressure for an -1 psi to 1 psi differential sensor with a 10 %FSS to 90 %FSS calibration when the recorded pressure count from the sensor is 65500 (in decimal):

Signed Counts = ((65500 + 2^{15}) % 2^{16}) - 2^{15}
Signed Counts = -36
Output(%FSS) = (-36+32000)/640
Output(%FSS) = 49.94

Equation 2: Pressure Sensor Transfer Function
Output(%FSS) = (Signed Counts + 32000)/640

Equation 3: Temperature Output Transfer Function
Output(%FSS) = (Signed Counts + 16485)/387.88

Pressure output

The ABP3 Series sensor pressure output may be expressed by the transfer function of the device as shown in Equation 2:

Equation 2: Pressure output function

$$Pressure = \frac{(Output - Output_{min}) * (P_{max} - P_{min})}{Output_{max} - Output_{min}} + P_{min}$$

Where,
$Output_{min}$ = output at minimum pressure [counts]
$Output_{max}$ = output at maximum pressure [counts]
Output = digital pressure reading [counts]
P_{max} = Maximum pressure range
P_{min} = Minimum pressure range
Pressure = pressure reading [mBar, Bar, psi, kPa, etc.]

Example
Calculate the pressure for a -2.5 mBar to 2.5 mBar differential sensor with a 10 % to 9 0% calibration, and a pressure output of 12750 counts.

The maximum counts +32000(2^{15}) and minimum counts is = -32000(-2^{15}) and the span to be considered (90-10) % = 80 %

$Output_{max}$ = +25600 counts (90 % of 2^{15} counts or 0×6400)

$Output_{min}$ =-25600 counts (10 % of 2^{15} counts or 0×9C00)

Output = 12750 counts

Pressure = pressure in mBar

$$Pressure = \frac{(12750 - (-25600)) \times (2.5-(-2.5))}{25600-(-25600)} + (-2.5)$$

$$Pressure = \frac{191750}{51200} - 2.5$$

$$Pressure = 1.24 \text{ mBar}$$

Temperature output

The ABP3 Series temperature output may be expressed by the transfer function as shown in Equation 3:

Equation 3: Temperature output transfer function

$$Temperature = \frac{T_{out} + 16485}{387.88}$$

Where,

Temperature = calculated temperature output in °C

T_{out} = digital temperature output in counts (decimal)

Example

Calculate the temperature in °C for a temperature output of -6783 (decimal) counts:

$$Temperature = \frac{(-6783+16485)}{387.88}$$

$$Temperature = \frac{9702}{387.88}$$

$$Temperature = 25.01°C$$

Demo Arduino code

Below is the reference demo code to communicate and read the data from the sensor using a microcontroller

```
#include <SPI.h>

const int CS_PIN = 10;

/* Registers */
#define REG_OUT_2E   0×2E
#define REG_OUT_30   0×30
#define REG_CMD      0×22
#define CMD_START_SINGLESHOT 0×8B93

/* CRC configs */
const unsigned char crc4_spi_poly = 0×03;
const unsigned char crc4_spi_init = 0×0F;

const unsigned char crc8_spi_poly = 0×07;
const unsigned char crc8_spi_init = 0xF3;

/* CS helpers */
static inline void csLow()  { digitalWrite(CS_PIN, LOW); }
static inline void csHigh() { digitalWrite(CS_PIN, HIGH); }

/* ------------------------------------------------------------ */
/* Command helpers */
byte cmdByte0(byte address, bool isRead) {
 byte a = (address & 0xFE);
 return isRead ? (a | 0×01) : a;
}

byte cmdByte1(byte wordCount, bool crcEnable) {
 byte len = ((wordCount - 1) & 0×07) ← 4;
 return (crcEnable ? 0×80 : 0×00) | len;
}

/* ------------------------------------------------------------ */
/* CRC FUNCTIONS (copied from working code) */

char calc_crc4(unsigned char polynom, unsigned char init, un-
signed char* data, unsigned int len)
{
 unsigned char shifter = init;
 for(int i=0;i<len;i++)
 {
  for(int j=7;j>=0;j--)
  {
   if((i >= len - 1) && (j < 4)) break;

   if( ((shifter >> 3) & 1) != ((data[i] >> j)&1) )
    shifter = (shifter ← 1) ^ polynom;
   else
    shifter = (shifter ← 1);

   shifter &= 0×0F;
  }
 }
 return shifter & 0×0F;
}
```

```
char calc_crc8(unsigned char polynom, unsigned char init, unsigned char* data, unsigned int len)
{
  unsigned char shifter = init;
  for(int i=0;i<len;i++)
  {
    for(int j=7;j>=0;j--)
    {
      if( ((shifter >> 7) & 1) != ((data[i] >> j)&1) )
        shifter = (shifter ← 1) ^ polynom;
      else
        shifter = (shifter ← 1);
    }
  }
  return shifter & 0xFF;
}

/* ------------------------------------------------------------ */
/* WRITE (unchanged) */
void spiWriteWord(byte address, uint16_t dataWord)
{
  csLow();
  SPI.beginTransaction(SPISettings(125000, MSBFIRST, SPI_MODE0));

  SPI.transfer(cmdByte0(address, false));
  SPI.transfer(cmdByte1(1, false));
  SPI.transfer((dataWord >> 8) & 0xFF);
  SPI.transfer(dataWord & 0xFF);

  SPI.endTransaction();
  csHigh();
}

/* ------------------------------------------------------------ */
/* ☑ READ WITHOUT CRC (unchanged) */
void spiReadWords(byte address, byte wordCount, uint16_t *out)
{
  csLow();
  SPI.beginTransaction(SPISettings(125000, MSBFIRST, SPI_MODE0));

  SPI.transfer(cmdByte0(address, true));
  SPI.transfer(cmdByte1(wordCount, false));

  for (byte i = 0; i < wordCount; i++)
  {
    byte hi = SPI.transfer(0×00);
    byte lo = SPI.transfer(0×00);
    out[i] = ((uint16_t)hi ← 8) | lo;
  }

  SPI.endTransaction();
  csHigh();
}

/* ------------------------------------------------------------ */
/* ☑ ☑ READ WITH CRC (integrated properly) */
bool spiReadWordsCRC(byte address, byte wordCount, uint16_t *out)
{
  unsigned char header[2];
  unsigned char crc4;
  unsigned char crc8;

  unsigned char high, low;

  csLow();

  header[0] = cmdByte0(address, true);
  header[1] = (1 ← 7) | ((wordCount - 1) ← 4);

  crc4 = calc_crc4(crc4_spi_poly, crc4_spi_init, header, 2);
  header[1] |= crc4;

  SPI.beginTransaction(SPISettings(125000, MSBFIRST, SPI_MODE0));

  unsigned char b  = SPI.transfer(header[0]);
  unsigned char ac = SPI.transfer(header[1]);

  crc8 = calc_crc8(crc8_spi_poly, crc8_spi_init, &b, 1);
  crc8 = calc_crc8(crc8_spi_poly, crc8, &ac, 1);

  for(int i=0;i<wordCount;i++)
  {
    high = SPI.transfer(0×00);
    low  = SPI.transfer(0×00);

    out[i] = ((uint16_t)high ← 8) | low;

    crc8 = calc_crc8(crc8_spi_poly, crc8, &high, 1);
    crc8 = calc_crc8(crc8_spi_poly, crc8, &low, 1);
  }

  /* pipeline byte */
  high = SPI.transfer(0×00);
  crc8 = calc_crc8(crc8_spi_poly, crc8, &high, 1);

  SPI.endTransaction();
  csHigh();

  /* ☑ PRINT CRC STATUS */
  Serial.print("CRC RESULT = 0x");
  Serial.println(crc8, HEX);

  if(crc8 == 0)
  {
    Serial.println("→ CRC MATCH ☑");
    return true;
  }
  else
  {
    Serial.println("→ CRC MISMATCH ✗");
    return false;
  }
}
```

```
/* ------------------------------------------------------------ */
void setup()
{
  Serial.begin(115200);

  SPI.begin();
  pinMode(CS_PIN, OUTPUT);
  csHigh();
}

/* ------------------------------------------------------------ */
void loop()
{
  /* 1. Trigger single-shot */
  spiWriteWord(REG_CMD, CMD_START_SINGLESHOT);
  delay(2);

  uint16_t block[4];

  /* ☑ READ WITH CRC */
  Serial.println("====== READ 0x2E WITH CRC ======");
  spiReadWordsCRC(REG_OUT_2E, 3, block);

  Serial.print("0x2E block: ");
  Serial.print("W0=0x"); Serial.print(block[0], HEX);
  Serial.print(" W1=0x"); Serial.print(block[1], HEX);
  Serial.print(" W2=0x"); Serial.println(block[2], HEX);

  /* ☑ Pressure */
  uint16_t p;
  spiReadWordsCRC(REG_OUT_30, 1, &p);

  Serial.print("0x30 Pressure word = 0x");
  Serial.println(p, HEX);

  delay(1000);
```

⚠ WARNING
PERSONAL INJURY

DO NOT USE these products as safety or emergency stop devices or in any other application where failure of the product could result in personal injury.

Failure to comply with these instructions could result in death or serious injury.

⚠ WARNING
MISUSE OF DOCUMENTATION

- The information presented in this datasheet is for reference only. Do not use this document as a product installation guide.
- Complete installation, operation, and maintenance information is provided in the instructions supplied with each product.

Failure to comply with these instructions could result in death or serious injury.

Warranty/Remedy

Honeywell Technologies warrants goods of its manufacture as being free of defective materials and faulty workmanship. Honeywell Technologies standard product warranty applies unless agreed to otherwise by Honeywell Technologies in writing; please refer to your order acknowledgement or consult your local sales office for specific warranty details. If warranted goods are returned to Honeywell Technologies during the period of coverage, Honeywell Technologies will repair or replace, at its option, without charge those items it finds defective. **The foregoing is buyer's sole remedy and is in lieu of all other warranties, expressed or implied, including those of merchantability and fitness for a particular purpose. In no event shall Honeywell Technologies be liable for consequential, special, or indirect damages.**

While Honeywell Technologies may provide information or engineering support for its products through Honeywell personnel, literature and website, it is the buyer's sole responsibility to determine the suitability of the Honeywell product(s) for the buyer's requirements.

Specifications may change without notice. The information we supply is believed to be accurate as of this writing. However, Honeywell assumes no responsibility for its use.

For more information

Honeywell Technologies services its customers through a worldwide network of sales offices and distributors. For application assistance, current specifications, pricing, or the nearest Authorized Distributor, visit our website or call:

USA/Canada	+1 302 613 4491
Latin America	+1 305 805 8188
Europe	+44 1344 238258
Japan	+81 (0) 3-6730-7152
Singapore	+65 6355 2828
Greater China	+86 4006396841

Visit our webite
https://automation.honeywell.com/

Honeywell Technologies
830 East Arapaho Road
Richardson, TX 75081
www.honeywell.com

008371-1-EN | 1 | 09/26
© 2026 Honeywell International Inc.