Programming guide for XENSIV™ PAS CO2

Target application: demand control ventilation

About this document

Scope and purpose

This application note serves as a programming starting guide and will focus on the setup and communication of the XENSIV™ PAS CO2 sensor driven by a microcontroller. Main focus will be the I²C functionality along with a quick example of how to set up communication and start basic measurement using the Cypress PSoC® 6 WiFi-BT Pioneer Kit and the Arduino Due.

Intended audience

Application engineers, system engineers and system architects of HVAC systems.

Table of contents

About this document **1**
Table of contents **1**
1 Introduction **2**
1.1 Introduction to the I²C bus 3
1.2 I²C communication protocol 3
2 I²C application circuit **5**
3 Initialization sequence **6**
4 Quick start with the PSoC® 6 WiFi-BT Pioneer Kit **7**
4.1 Bridge Control Panel 8
4.2 Basic code for starting measurement 10
4.3 Additional functionality 12
5 Quick start with the Arduino Due **14**
5.1 Arduino IDE 14
5.2 Basic Arduino code for starting measurement 16
5.3 Arduino library 20
6 UART interface **22**
6.1 Write transactions 23
6.2 Read transactions 24
6.3 Arduino examples 25
7 PWM interface **29**
Revision history **33**

Downloaded from Arrow.com.

1 Introduction

The XENSIV™ PAS CO2 sensor is a real carbon dioxide (CO_2) sensor in an unprecedentedly small form factor. Designed on the basis of a unique photoacoustic spectroscopy (PAS) concept, the sensor saves more than 75 percent space compared to existing commercial real CO_2 sensors. Its direct ppm readings, SMD capability and simple design allow for quicker and easier integration into customers' systems in low- and high-volume applications alike.

The photoacoustic principle can be traced back to over 100 years ago, first discovered by Alexander Graham Bell in 1880. The photoacoustic effect involves the formation of sound waves (pressure changes) following light absorption in a material sample. The sound signal is quantified by detectors such as microphones. In order to obtain this effect, the light intensity must vary. A PAS gas sensor is based on the principle that gases absorb light in a specific wavelength of the infrared spectrum. CO_2 molecules, for example, have strong absorption in the λ = 4.2 µm wavelength.

The XENSIV™ PAS CO2 sensor module integrates, on the same PCB, the PAS transducer, a microcontroller for signal processing, algorithms and a MOSFET. As depicted in the block diagram, the PAS transducer includes: i) a proprietary infrared emitter with blackbody radiation, which is periodically chopped by the MOSFET; ii) a narrow-band optical filter passing the CO_2 specific wavelength λ = 4.2 µm, significantly improving the sensor selectivity compared to other gases, including humidity; and iii) Infineon's high-SNR (signal-to-noise ratio) MEMS microphone XENSIV™ IM69D130, detecting the pressure changes generated by the CO_2 molecules. All the components are developed and designed in-house in accordance with Infineon's high-quality guidelines. The sensor therefore benefits from Infineon's illustrious record of accomplishments in MEMS design and acoustic capabilities, resulting in it being best-in-class for price/performance.

The XENSIV™ PAS CO2 sensor is ideal for smart-home and building automation as well as various indoor air quality IoT devices such as air purifiers, thermostats, weather stations and personal assistants. The sensor enables end users to track, understand and improve the air quality surrounding them in a timely and highly energy-efficient manner.

In the following the I^2C interface will be explained in detail and at the end of the application note the UART and PWM interface will be covered.

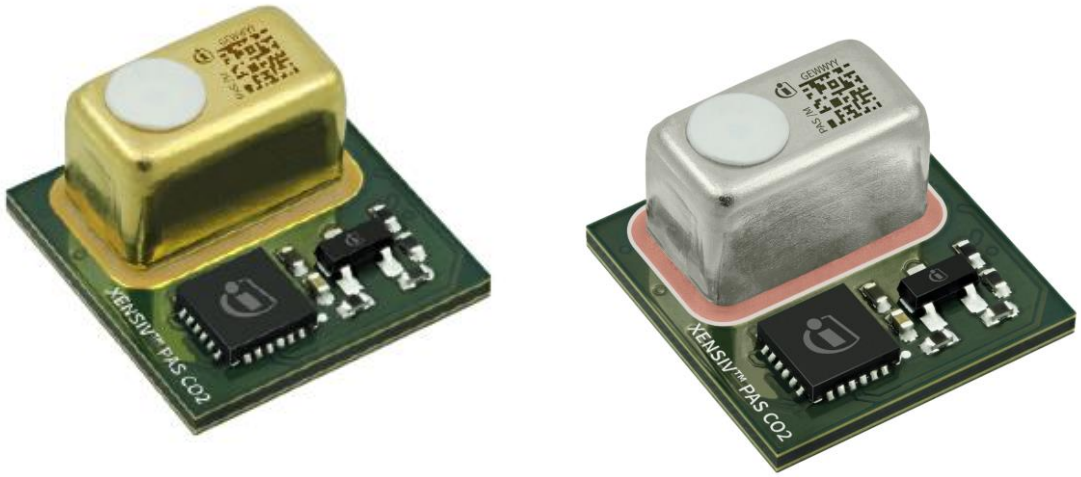


Figure 1 XENSIV™ PAS CO2 sensor (left) and XENSIV™ PAS CO2 5V sensor (right)

Downloaded from Arrow.com.

1.1 Introduction to the I²C bus

The I²C bus is a bidirectional two-line bus, enabling communication between any kind of integrated circuit that supports this protocol, either by hardware or software. Examples of such ICs are LCD controllers, EEPROMs, RAMs, data converters or general-purpose microcontrollers. The main advantage of this protocol is its two-line interface, as shown in Figure 2.

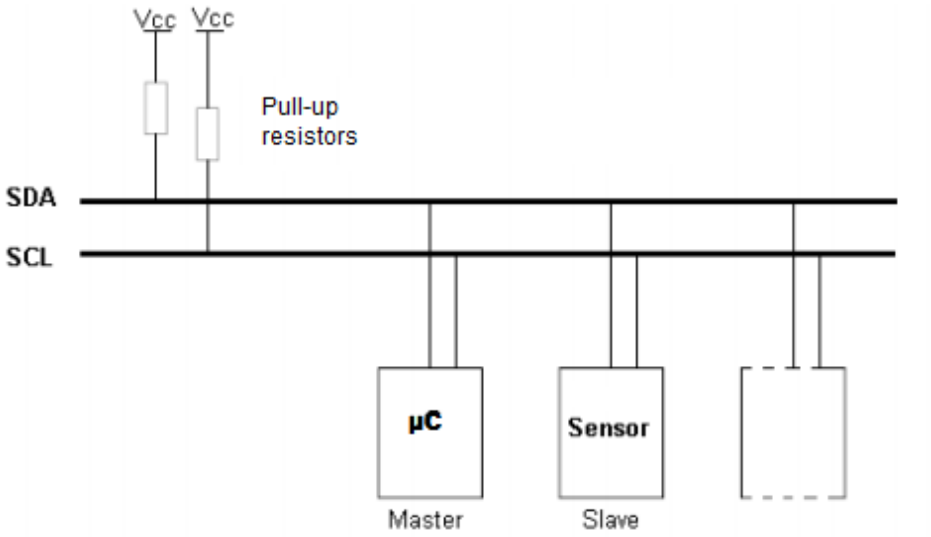


Figure 2 Example of an I²C bus configuration

1.2 I²C communication protocol

The word “master” refers to a device that initiates and terminates a transfer and also provides the clock signals on line SCL. Master devices operate as transmitters or receivers.

At the start of each transfer, a slave is addressed by its own unique address. A transfer consists of a start condition, the data bits, an acknowledge bit and possibly a stop condition. This concept is shown in Figure 3.

A start condition is defined by a falling edge at SDA while the SCL is high. A stop condition is defined by a rising edge at SDA while the SCL line is high. When transmitting data, no changes at the SDA line while the clock is high are permitted, otherwise this will result in a stop or a start condition! In order to avoid this, the master should change the data at SDA only when the SCL line is low.

After the transmission of the 8 data bits, the master sets the SDA line to high and the slave acknowledges the transfer by pulling the SDA line to ground. This indicates a successful transfer.

Master-read mode is not exactly the same. The master still provides the clock but the slave now submits the data (requested by the master) at the SDA line. At the end of the transmission, the master does not acknowledge (SDA is set and remains high).

For further information the interested reader is referred to the original I²C-bus specification (UM10204 Rev 6, NXP Semiconductors).

An example for reading the status of the XENSIV™ PAS CO2 sensor is shown in Figure 4. An introduction to the basic registers will be given in the latter half of this application note. A more detailed description of all available registers and functions of the XENSIV™ PAS CO2 sensor can be found in a separate application note.

Downloaded from Arrow.com.

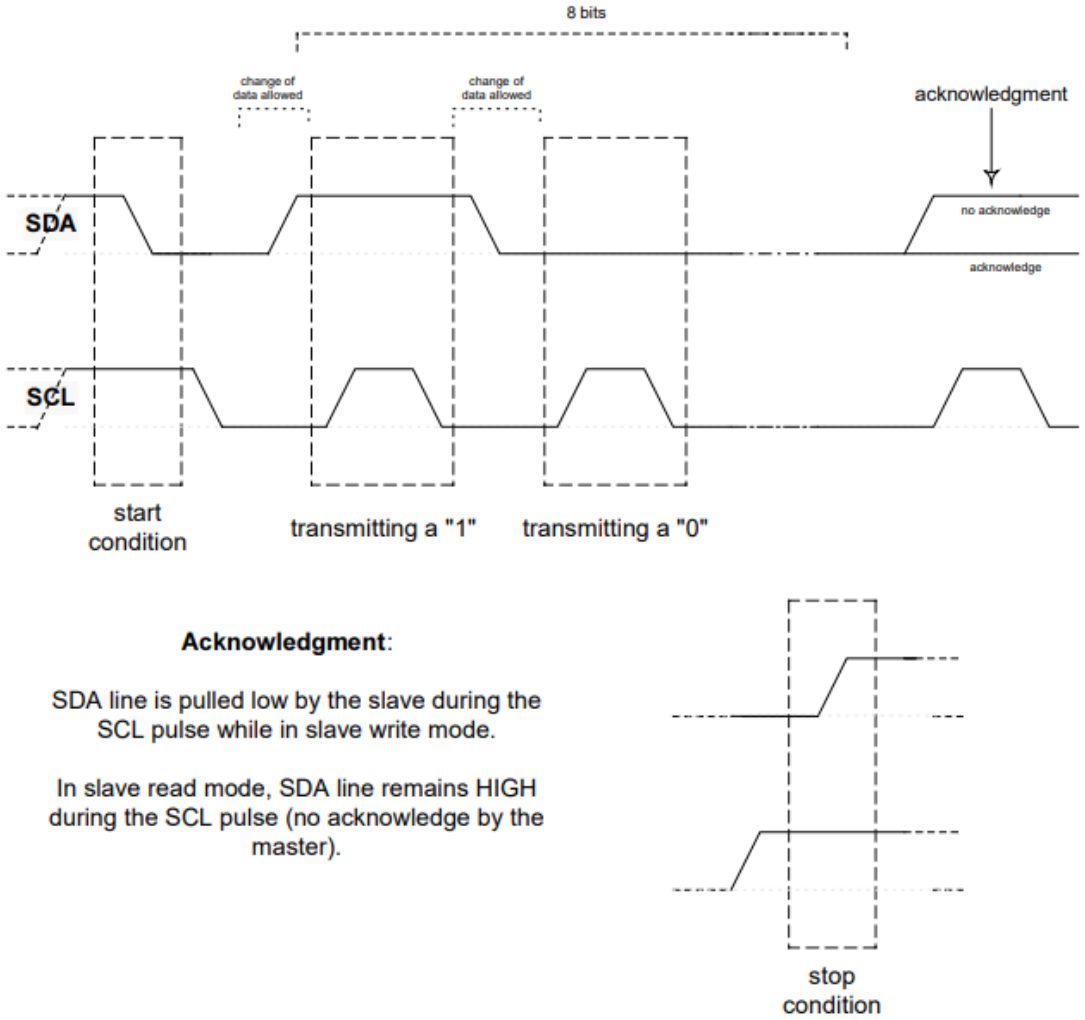


Figure 3 Example of an I²C transaction

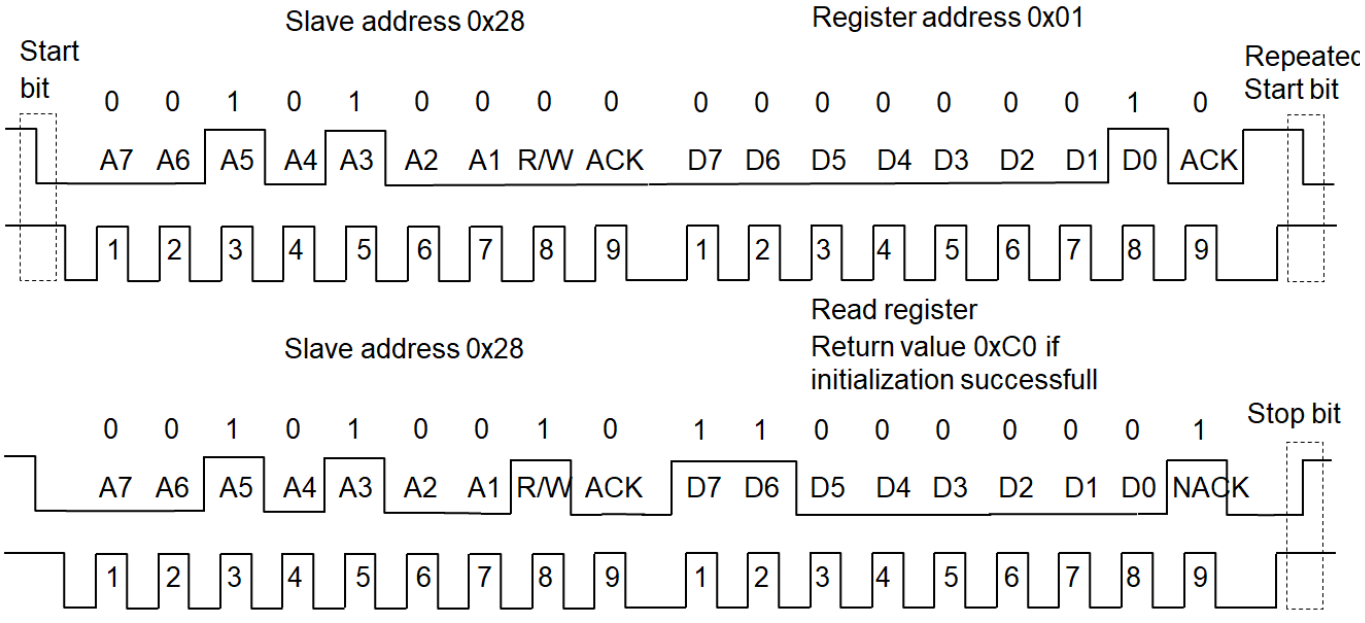


Figure 4 I²C transaction for reading the status of the sensor

Downloaded from Arrow.com.

2 I²C application circuit

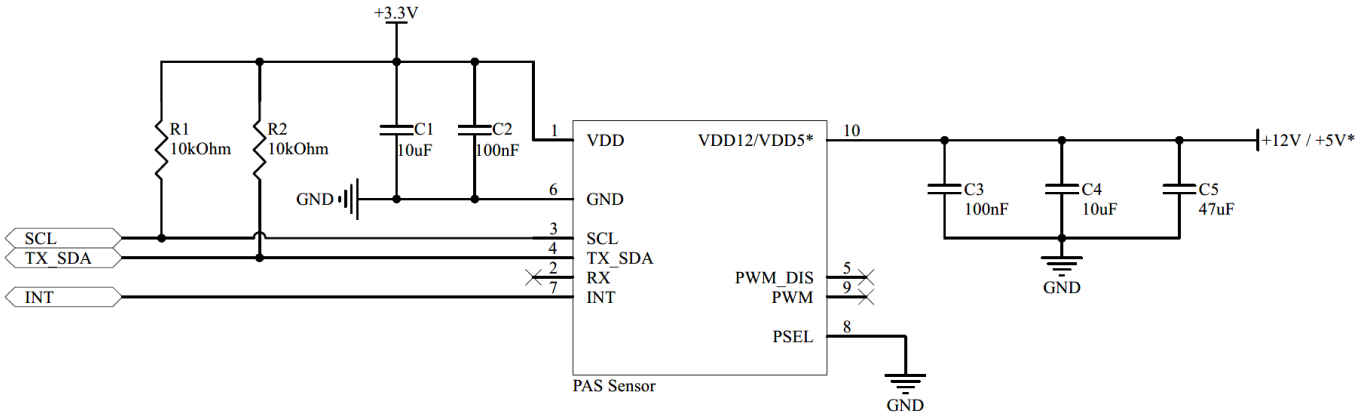


Figure 5 Application circuit example

Table 1 Pin descriptions

Pin	Symbol	Type	Function
1	VDD	Power supply (3.3 V)	3.3 V digital power supply
2	RX	Input	UART receiver pin (3.3 V domain)
3	SCL	Input/Output	I²C clock pin (3.3 V domain)[3]
4	TX_SDA	Input/Output	UART transmitter pin/I²C data pin (3.3 V domain)[3]
5	PWM_DIS	Input	PWM disable input pin (3.3 V domain)[2]
6	GND	Ground	Ground
7	INT	Output	Interrupt output pin (3.3 V domain)
8	PSEL	Input	Communication interface select input pin (3.3 V domain)[1]
9	PWM	Output	PWM output pin (3.3 V domain)
10	VDD12 / VDD5*	Power supply (12 V / 5 V*)	12 V / 5 V* power supply for the IR emitter

[1] High level selects UART and low level selects I²C. It is recommended to the user to hard wire the pin to VDD or GND, depending on the wanted interface.

[2] If PWM_DIS is hard wired to GND to enable the PWM output, the device will start in continuous mode and not idle mode which needs to be considered when changing the measurement period.

[3] Values of pull-up resistances should be adjusted according to the overall application and used clock frequency.

*Is referring to XENSIV™ PAS CO2 5V which is using 5 V for the emitter instead of the 12 V compared to XENSIV™ PAS CO2

Downloaded from Arrow.com.

3 Initialization sequence

In order not to damage the sensor or other components, a certain initialization sequence must be followed:

- Connect all the necessary pins as seen in Figure 5 of the XENSIV™ PAS CO2 Sensor2Go Kit to the microcontroller (power off).
- Power on the microcontroller (3.3 V supply for the communication).
- Supply 12 V / 5 V* externally for the heater.
- Run script/code (configure settings, start measurement, etc.).



Figure 6 Initialization sequence

When powering off the sensor, the sequence must be reversed.

Note: To prevent race conditions, it is recommended that the logic supply with 3.3 V is fully booted first, followed by the heater supply with 12 V / 5 V.*

Downloaded from Arrow.com.

4 Quick start with the PSoC® 6 WiFi-BT Pioneer Kit

PSoC® 6 bridges the gap between expensive, power-hungry application processors and low-performance microcontrollers. The ultra-low-power PSoC 6 microcontroller architecture offers the processing performance needed by IoT devices, eliminating the tradeoffs between power and performance. The PSoC 6 microcontroller contains a dual-CPU architecture, with both CPUs on a single chip. It has an ARM® Cortex-M4 for high-performance tasks, and an ARM® Cortex-M0+ for low-power tasks. With security built in, your IoT system is protected.

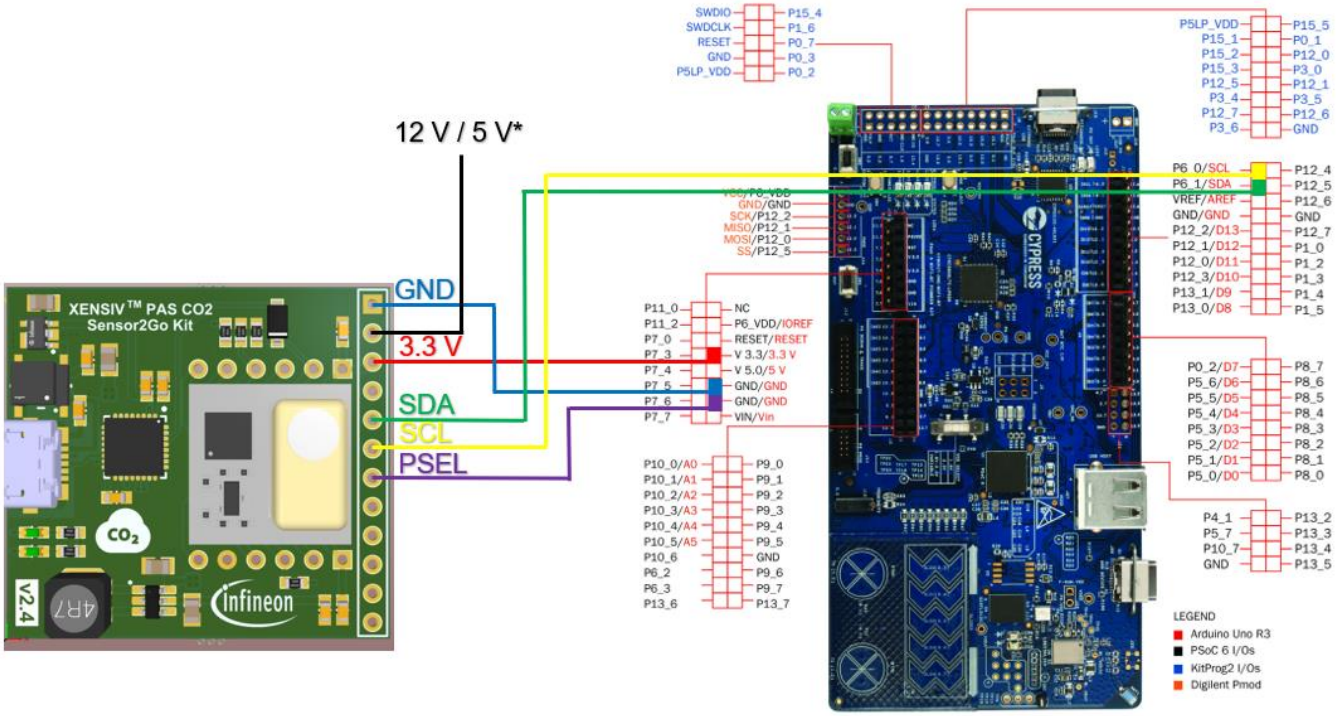


Figure 7 XENSIV™ PAS CO2 Sensor2Go Kit I²C interface connection to the PSoC® 6 WiFi-BT Pioneer Kit

Table 2 Pin connections

Position	Symbol	Connection to the PSoC® 6 WiFi-BT Pioneer Kit
1	GND	Ground
2	VDD12 / VDD5*	12 V / 5 V* power supply (externally)[3)]
3	VDD3.3	3.3 V digital power supply[3)]
4	RX	Not connected
5	TX/SDA	I²C data pin (3.3 V domain)
6	SCL	I²C clock pin (3.3 V domain)
7	PSEL	Ground
8	INT	Not connected (in this case)
9	PWM_DIS	Ground
10	PWM	Not connected (in this case)
11	SWD	Not connected (in this case)
12	SWCLK	Not connected (in this case)

[3)] Power supply tolerance ±10 percent

Downloaded from Arrow.com.

4.1 Bridge Control Panel

Bridge Control Panel is a simple debugging tool that comes with PSoC Programmer. It is used to communicate with target devices over I²C/UART/SPI serial communication interfaces.

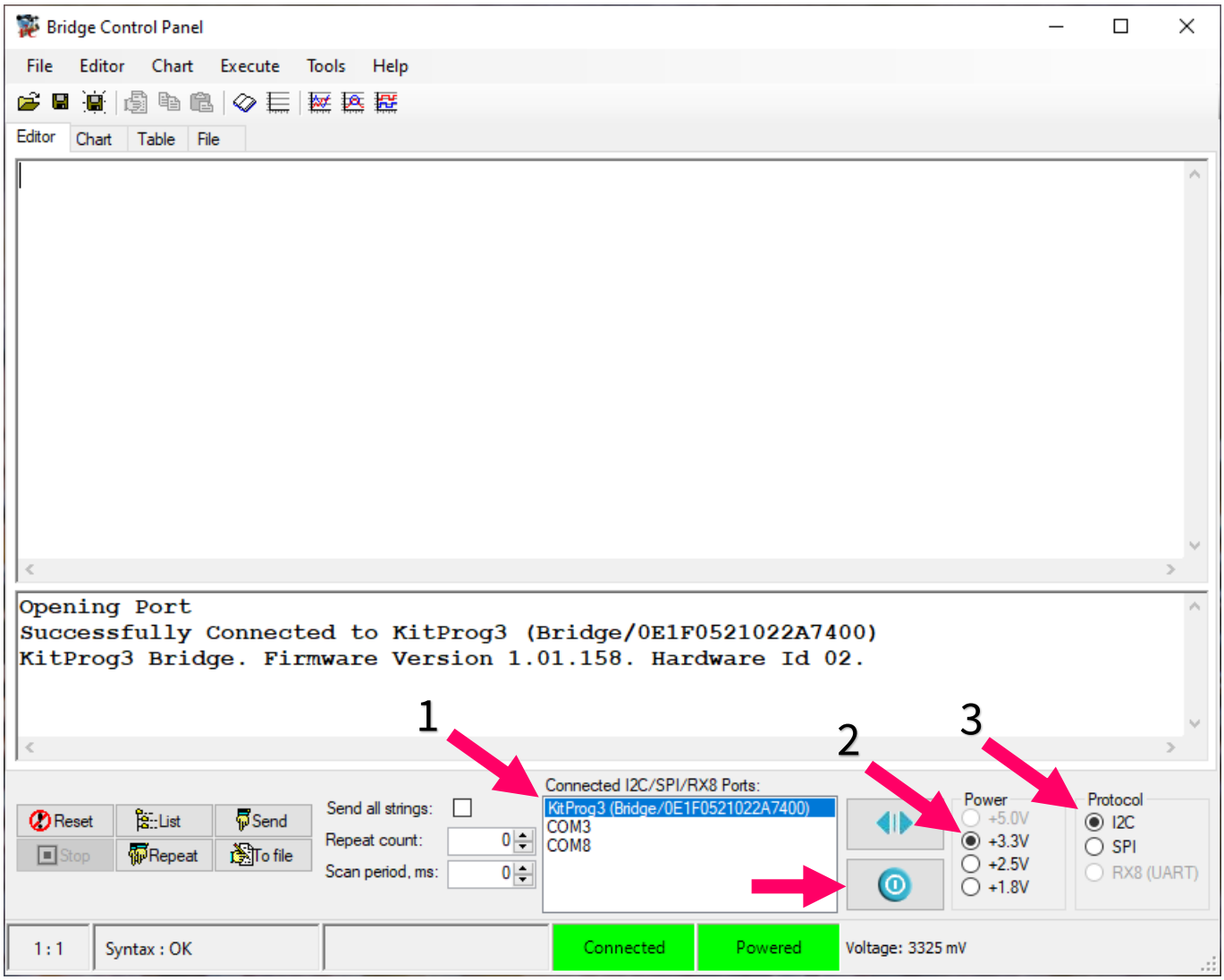


Figure 8 Initialization of the device

After wiring up the XENSIV™ PAS CO2 Sensor2Go Kit and the PSoC® 6 WiFi-BT Pioneer Kit as shown in Figure 7, the following steps cover initialization and communication with the sensor:

1. Select COM port accordingly (KitProg3).

2. Select 3.3 V in the Power menu.

3. Select I²C protocol.

With the “Toggle power” button you can switch the 3.3 V power supply on and off. Only after the 3.3 V is supplied can the 12 V / 5 V* be supplied externally safely. Make sure when powering off the 3.3 V supply with the “Toggle power” button to first power off the external 12 V / 5 V* supply and then the 3.3 V supply to avoid damaging the device.

Downloaded from Arrow.com.

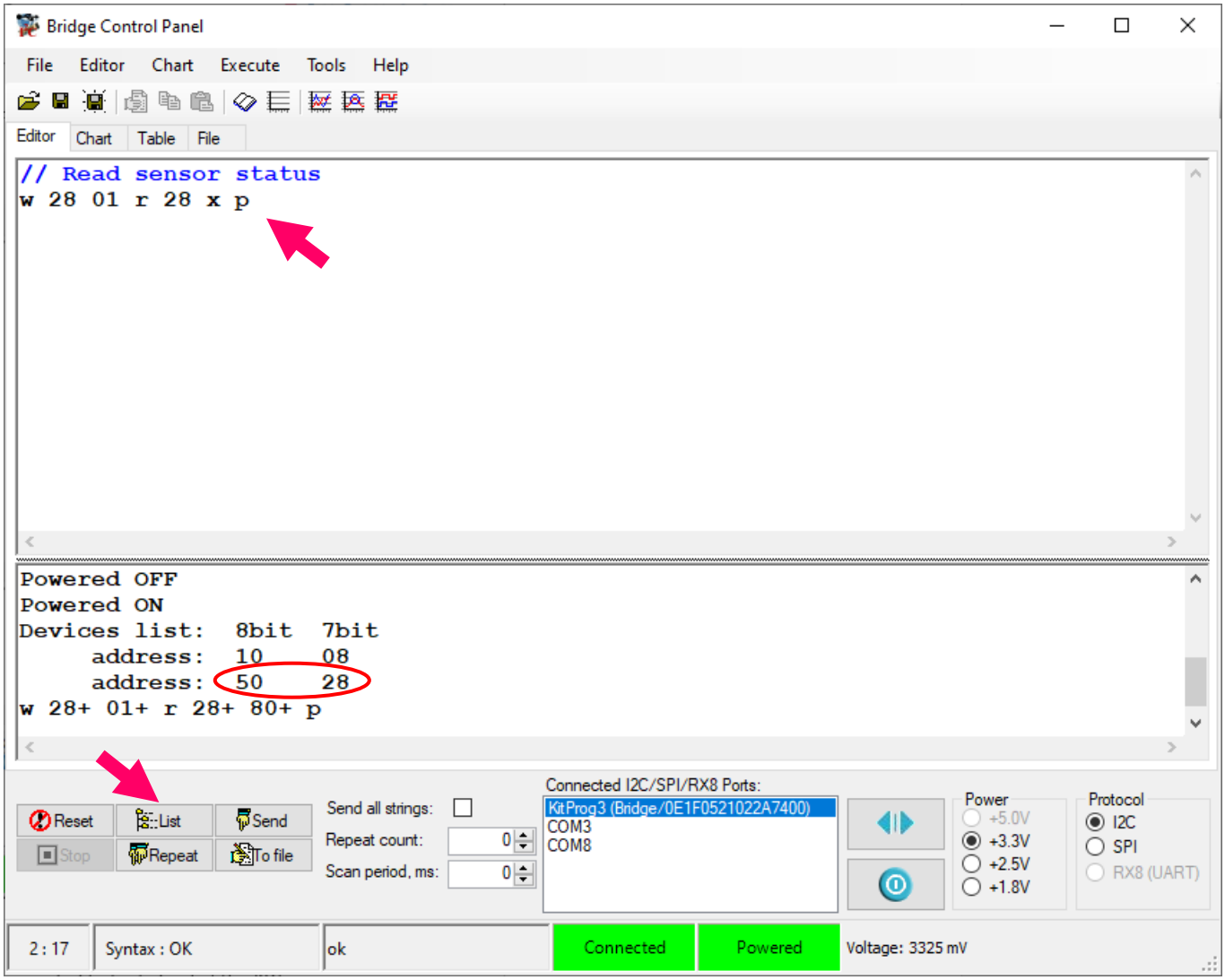


Figure 9 Testing I²C communication

After powering up both supplies in the respective order it is recommended to first check the basic I²C communication by pressing the “List” button. The bridge control panel will now list all the available I²C slave devices available on the bus. Check the terminal for slave address 0x28 (0x50 8 bit), which is the address of XENSIV™ PAS CO2.

Known issues and mistakes are:

- No slave devices listed when checking for slave response with the “List” button (error message in the terminal: “No device found”)
  - Make sure all wires are properly connected and not loose.
- Bridge control panel is crashing
  - Make sure there is no shortage as a result of wrongly connected supply pins.
- Device doesn’t receive commands (e.g. w 28- 01- r 28- FF- p)
  - Make sure I²C wires (SDA/SCL) are properly connected and not loose.

The commands are written in the “Editor” window and are sent and executed by pressing “Enter”.

Downloaded from Arrow.com.

4.2 Basic code for starting measurement

In this section the basic operation for starting a measurement with the XENSIV™ PAS CO2 will be described.

After initializing the device according to the sequence in chapter 3 and section 4.1, the first thing recommended to do is check if the initialization went correctly without any errors. This can be done by checking the **sensor status register**. After that, and before starting a measurement, it is recommended to check and set the pressure compensation. This can be done with the two **pressure compensation registers**. The default pressure set in these registers is 1015 hPa. Now everything is set to start a measurement. There are two different measurement modes available: single-shot measurement and continuous measurement. The mode can be configured in the **measurement mode configuration register**. When using the continuous measurement mode, the measurement period can be defined in the two **measurement period configuration registers** beforehand. Either way, after configuring the measurement mode, a measurement sequence is triggered. By reading the **measurement status register** it is possible to check if the measurement sequence is completed and thus if a new CO_2 concentration value is available in the **CO_2 concentration result register**.

Sensor status register (SENS_STS, address: 0x01)

```
1  w 28 01 r 28 x p
```

If the sensor is initialized correctly the return value is 0xC0.

Pressure compensation registers (PRES_REF_H and PRES_REF_L, address: 0x0B and 0x0C)

```
1  w 28 0B r 28 x p
2  w 28 0C r 28 x p
3  w 28 0B 03 p
4  w 28 0C F5 p
```

Registers PRES_REF_L and PRES_REF_H are used to store the ambient atmospheric pressure. The concatenation of PRES_REF_H (MSB) and PRES_REF_L (LSB) defines the pressure value that shall be considered by the device. The concatenated pressure value is coded as an unsigned short integer (1 bit = 1 hPa). In this example the pressure is set to 1013 hPa. For correct operation, the user shall ensure that the pressure value programmed is within the specified pressure operating range of the device. This valid range of operation is 750 hPa to 1150 hPa.

User also needs to ensure to calculate the correct ambient atmospheric pressure based on the located altitude. The relationship between altitude and atmospheric pressure is inverse, which means that as altitude increases, atmospheric pressure decreases. This relationship is due to the density of the air above a given point on Earth's surface. At sea level, the density of the atmosphere is at its maximum, which results in a higher atmospheric pressure. As you move higher in altitude, the amount of air above you decreases, which in turn reduces the density of the atmosphere and the air pressure decreases. This relationship is also affected by changes in atmospheric conditions, such as temperature and humidity. To calculate the atmospheric pressure at a given altitude, the following equation can be used:

$$P = P_0 * \left[1 - \left(\frac{L * h}{T_0}\right)\right]^{\frac{g * M}{R * L}}$$

Where,

- P is the pressure at the given altitude (in hectopascals)
- P_0 is the pressure at sea level (in hectopascals) with 1013.207 hPa
- L is the temperature lapse rate, which is a constant value of -0.0065 $\frac{K}{m}$ up to an altitude of 11 km

Downloaded from Arrow.com.

- *h* is the altitude above sea level (in meters)
- T_0 is the standard temperature at sea level (in Kelvin), which is a constant value of 288.15 K
- *g* is the acceleration due to gravity, which is a constant value of 9.80665 $\frac{m}{s^2}$
- *M* is the molar mass of air, which is a constant value of 0.0289644 $\frac{kg}{mol}$
- *R* is the universal gas constant, which is a constant value of 8.31447 $\frac{J}{K * mol}$

For your convenience you can copy paste this equation for further use and insert “h” for the height.

P = 1013.207*(1-((6.5.*h)/(288150))).^5.255; // h .. height in meter

Examples: The device requires a value in hPa as input. As a result, 1000 m altitude would result in 898 hPa and respectively 2000 m results in 794 hPa.

Measurement period configuration registers (MEAS_RATE_H and MEAS_RATE_L, address: 0x02 and 0x03)

```
1  w 28 02 00 p
2  w 28 03 0A p
```

Registers MEAS_RATE_H and MEAS_RATE_L define the measurement period used in continuous mode. The concatenation of MEAS_RATE_H (MSB) and MEAS_RATE_L (LSB) defines the period. The concatenated value is coded as a two’s complement signed short integer (1 bit = 1 s). In this example the measuring rate is set to 10 s. The configurable range is from 0005_H (5 s) to $0FFF_H$ (4095 s). When writing to MEAS_RATE_H and MEAS_RATE_L, the new value is not immediately considered by the device. It is internally latched at the next transition from idle mode to continuous mode.

Measurement mode configuration register (MEAS_CFG, address: 0x04)

```
1  w 28 04 02 p
2  w 28 04 01 p
```

This register defines the operation settings of the device. With code in line 1 a single-shot measurement is triggered and with code in line 2 the continuous measurement mode is configured. Note that after one measurement sequence the emitter needs at least 10 s to cool down. Measurement rate values in continuous mode below 5 s are treated as being equal to 5 s. For single-shot measurement make sure to delay the following measurement sequence by at least 60 s for accurate readings.

Measurement status register (MEAS_STS, address: 0x07)

```
1  w 28 07 r 28 x p
```

This register displays status information of the sensor. Once a measurement sequence is completed and the new CO_2 concentration value is available the return value of this register is 0x10.

CO_2 concentration result register (CO2PPM_H and CO2PPM_L, address: 0x05 and 0x06)

```
1  w 28 05 r 28 x p
2  w 28 06 r 28 x p
```

Registers CO2PPM_H and CO2PPM_L are used to store the result of the last CO_2 concentration measurement. The concatenation of CO2PPM_H (MSB) and CO2PPM_L (LSB) defines the CO_2 concentration value. The concatenated

Downloaded from Arrow.com.

CO_2 concentration value is coded as a two's complement signed short integer (1 bit = 1 ppm). This field is updated at the end of each measurement sequence. When reading the CO_2 concentration value, the user shall first read registers CO2PPM_H and then CO2PPM_L.

Summary

```
1  w 28 01 r 28 x p // Read sensor status
2  w 28 0B r 28 x p // Read pressure (MSB)
3  w 28 0C r 28 x p // Read pressure (LSB)
4  w 28 0B 03 p // Set pressure (MSB)
5  w 28 0C F5 p // Set pressure (LSB)
6  w 28 02 00 p // Set measurement period (MSB)
7  w 28 03 0A p // Set measurement period (LSB)
8  w 28 04 02 p // Trigger continuous measurement
9  (w 28 04 01 p) // Trigger single shot measurement
10 w 28 07 r 28 x p // Read measurement status
11 w 28 05 r 28 x p // Read CO2 concentration (MSB)
12 w 28 06 r 28 x p // Read CO2 concentration (LSB)
```

Attention: ***Full detailed register map has been covered in a separate application note (see product page)***

Note: *The pressure compensation register should be updated regularly to compensate for barometric pressure variation.*

4.3 Additional functionality

In this section additional possible functions of the XENSIV™ PAS CO2 will be introduced. The full description of the available functionality will be covered in a separate application note.

Interrupt pin configuration register (INT_CFG, address: 0x08)

```
1  w 28 08 18 p // INT configuration as early measurement start notification
```

This register defines the configuration of pin INT. Pin INT is a multi-purpose output pin that can be configured to perform several functions. The electrical configuration can be either set as push pull and low active or push pull and high active.

The following functions can be configured. Alarm threshold violation notification pin, Data Ready notification pin, sensor busy notification pin and early measurement start notification pin. In this example the interrupt is configured as push pull and high active early measurement start notification pin. The indication if an interrupt did occur can be read in the measurement status register as well as the clearing of the sticky bits.

Alarm threshold register (ALARM_TH_H and ALARM_TH_L, address: 0x09 and 0x0A)

```
1  w 28 09 03 p // Set alarm threshold (MSB)
2  w 28 0A E8 p // Set alarm threshold (LSB)
```

Registers ALARM_TH_H and ALARM_TH_L define the value used as a threshold for the alarm violation. The concatenation of ALARM_TH_H (MSB) and ALARM_TH_L (LSB) define the threshold value that shall be considered by the device. The concatenated value is coded as a two's complement signed short integer (1 bit = 1 ppm). In this example the alarm threshold is set to 1000 ppm. The indication if a threshold violation did occur can be read in the measurement status register.

Downloaded from Arrow.com.

Automatic baseline offset compensation reference (CALIB_REF_H and CALIB_REF_L, address: 0x0D and 0x0E)

```
1  w 28 0D 01 p // Set baseline reference (MSB)
2  w 28 0E 90 p // Set baseline reference (LSB)
```

Registers CALIB_REF_H and CALIB_REF_L define the reference value used for the automatic baseline offset compensation or forced compensation. The concatenation of CALIB_REF_H (MSB) and CALIB_REF_L (LSB) define the reference value. The concatenated offset value is coded as a 2's complement signed short integer (1 bit = 1 ppm). In this example the automatic baseline offset compensation reference is set to 400 ppm. For correct operation, the user shall ensure that the compensation value programmed is within the specified operating range of the device. This valid range of operation is 350 ppm to 1500 ppm. The automatic baseline offset compensation or forced compensation can be enabled/disabled in the measurement mode configuration register and be reset in the soft reset register. More details regarding the automatic baseline offset compensation and forced compensation are covered in a separate application note (see product page).

Scratch pad register (SCRATCH_PAD, address: 0x0F)

```
1  w 28 0F 01 p
2  w 28 0F r 28 x p
```

This register provides a readable and writable address space for data integrity test during runtime. This register is not associated with a specific hardware functionality.

Soft reset register (SENS_RST, address: 0x10)

```
1  w 28 10 A3 p // Triggers soft reset
2  w 28 10 DF p // Disables filter
3  w 28 10 FE p // Enables filter
4  w 28 10 BC p // Resets ABOC context
5  w 28 10 FC p // Resets forced compensation
6  w 28 10 CF p // Saves forced compensation immediately
```

This register is used to trigger a soft reset. It also covers the settings of the filter and resetting the automatic baseline offset compensation or forced compensation. By default, the filter is enabled and can be disabled by writing 0xDF to this register. With writing 0xFE the filter is enabled again. The context of the automatic baseline offset compensation can be reset with writing 0xBC and for the reset of the forced compensation it is 0xFC.

Downloaded from Arrow.com.

5 Quick start with the Arduino Due

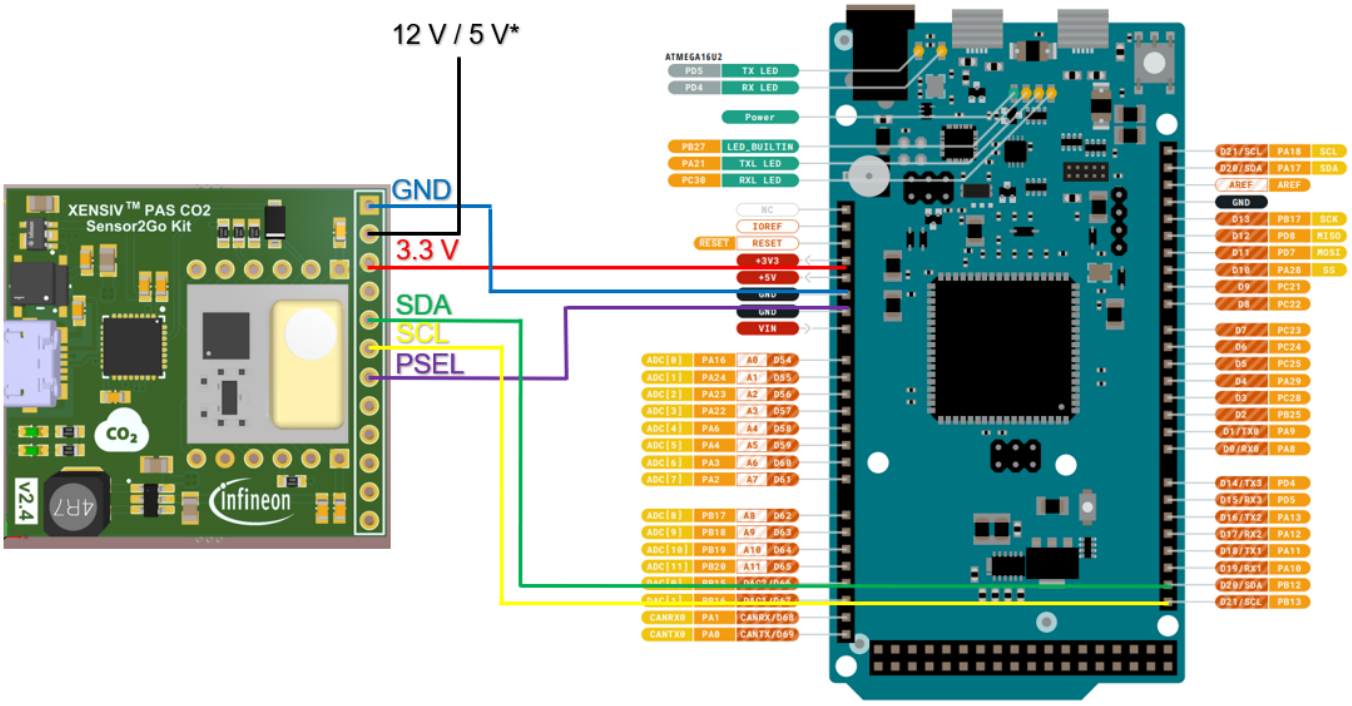


Figure 10 XENSIV™ PAS CO2 Sensor2Go Kit I²C interface connection to the Arduino Due[4)]

The pin connection of the device to the Arduino Due is equivalent to the connection to the PSoC® 6 WiFi-BT Pioneer Kit. When using another Arduino or other digital pins, make sure that the respective pull-up resistors are available.

[4)] Pinout of Arduino Due from https://content.arduino.cc/assets/Pinout-Due_latest.pdf

5.1 Arduino IDE

After installing the Arduino IDE, make sure to install the right package “Arduino SAM Boards” (32-bit ARM® Cortex-M3) including the Arduino Due with the board manager (see Figure 11). Make sure to select the respective board and COM port in the Tools dropdown menu (see Figure 12). Use the programming port for uploading sketches and communicating with the Arduino Due. It is recommended to first check with the “Blink” example if communication with the Arduino Due is present and responsive. After the communication with the Arduino Due is set and confirmed, implementation of the code can begin.

Downloaded from Arrow.com.

Quick start with the Arduino Due

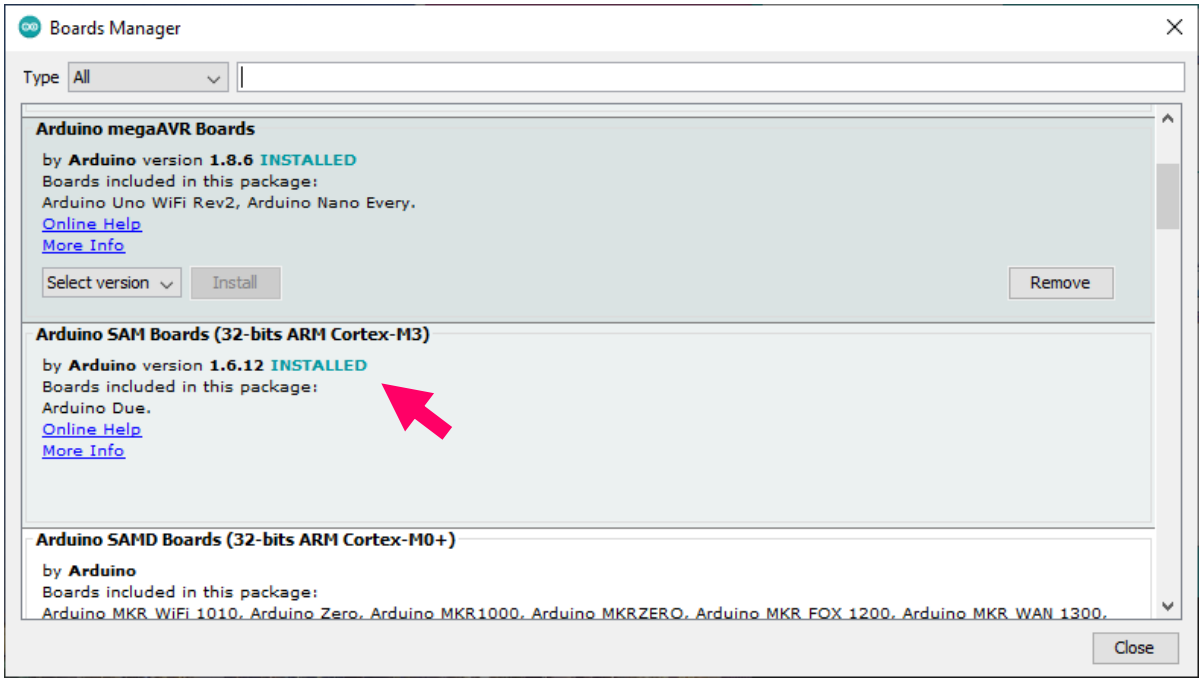


Figure 11 Arduino IDE settings: board manager

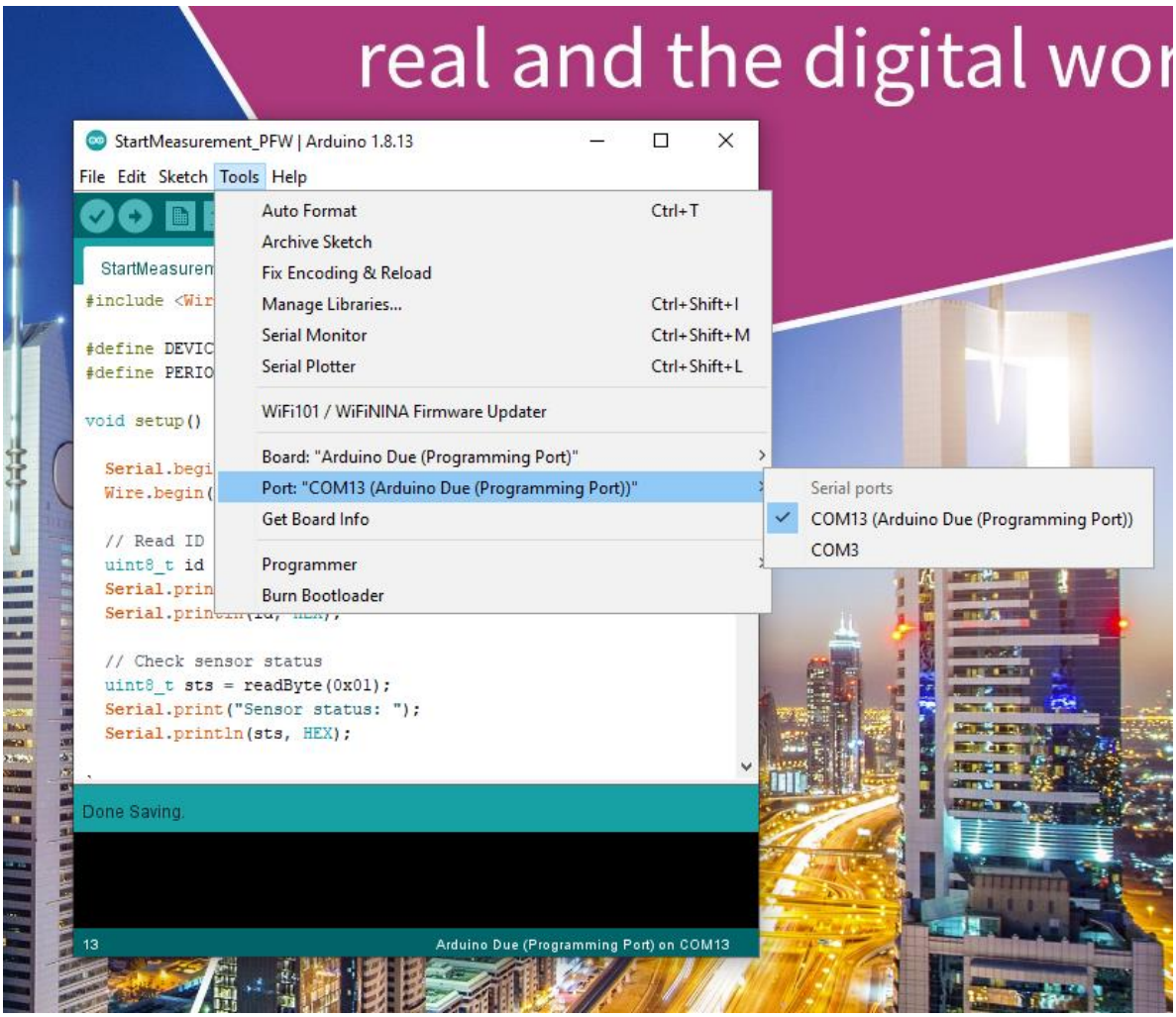


Figure 12 Arduino IDE settings: selecting the port

Downloaded from Arrow.com.

5.2 Basic Arduino code for starting measurement

It is recommended to implement functions for read and write commands. After that the device can be initialized, checked and operated. A measurement can be started just like with the PSoC® 6 WiFi-BT Pioneer Kit by writing and reading the responding registers. One thing to note is that it is important to set sufficient delays so that no command packages get lost or skipped. Following are four Arduino code examples: one for reading the register, one for writing into the register, and one script each for utilizing these functions to start a single-shot measurement or continuous mode measurement.

readByte

```
uint8_t readByte(uint8_t regAddress)
{
  Wire.beginTransmission(deviceAddress);
  Wire.write(regAddress);
  Wire.endTransmission(false);
  //request 1 byte from slave
  if (Wire.requestFrom(deviceAddress, 1U, 1U) > 0)
  {
    return Wire.read();
  }
  else
  {
    return 0x0;
  }
}
```

writeByte

```
bool writeByte(uint8_t regAddress, uint8_t data)
{

  Wire.beginTransmission(deviceAddress);
  Wire.write(regAddress);
  Wire.write(data);
  if (Wire.endTransmission() != 0)
  {
    return false;
  }
  else
  {
    return true;
  }
}
```


Downloaded from Arrow.com.

Quick start with the Arduino Due

Code for starting a single-shot measurement

```
#include <Wire.h>

#define deviceAddress 0x28
#define PERIOD 10000

void setup() {

  Serial.begin(115200);
  Wire.begin();

  // Check sensor status
  uint8_t sts = readByte(0x01);
  Serial.print("Sensor status: ");
  Serial.println(sts, HEX);

  // Idle mode
  writeByte(0x04, 0x00);
  delay(400);

  // Set pressure
  writeByte(0x0B, 0x03);
  writeByte(0x0C, 0xF5);
}

void loop()
{
  // Trigger single measurement
  writeByte(0x04, 0x01);
  delay(1150);

  // Get PPM value
  uint8_t value1 = readByte(0x05);
  delay(5);
  uint8_t value2 = readByte(0x06);
  delay(5);

  // Calculate ppm value
  int16_t result = value1 << 8 | value2;
  Serial.print("CO2: ");
  Serial.print(result);
  Serial.println(" ppm");

  delay(PERIOD);

}
```


Downloaded from Arrow.com.

Code for starting a continuous mode measurement

```
#include <Wire.h>
#define deviceAddress 0x28

void setup() {

  Serial.begin(115200);
  Wire.begin();

  // Read ID
  uint8_t id = readByte(0x00);
  Serial.print("ID: ");
  Serial.println(id, HEX);

  // Check sensor status
  uint8_t sts = readByte(0x01);
  Serial.print("Sensor status: ");
  Serial.println(sts, HEX);

  // Idle mode
  writeByte(0x04, 0x00);
  delay(400);

  // Set measurement rate to 10 s
  writeByte(0x02, 0x00);
  writeByte(0x03, 0x0A);

  // Configure continuous mode
  writeByte(0x04, 0x02);
}

void loop() {

  // Poll measurement status
  uint8_t meas_sts = readByte(0x07);
  delay(100);

  if (meas_sts == 0x10) {

    // Get PPM value
    uint8_t value1 = readByte(0x05);
    delay(5);
    uint8_t value2 = readByte(0x06);
    delay(5);

    // Calculate ppm value
    int16_t result = value1 << 8 | value2;
    Serial.print("CO2: ");
    Serial.print(result);
    Serial.println(" ppm");
  }
  delay(1000);
}
```


Downloaded from Arrow.com.

Quick start with the Arduino Due

For the continuous mode, synchronization between application microcontroller and device needs to be considered. It shall be noted that when a measurement sequence is initiated, the device does not respond to any incoming frame for the duration of the measurement (~ 1 s) and will instead response with NACK. There are two options to handle that based on the example.

- Configure the interrupt as data ready and monitor that if the measurement is done
- Repoll the measurement status again after receiving the NACK response (example in the Arduino library available)

To ensure accurate and reliable measurement, the sensor must be operated within its recommended measurement rate of 60 seconds. This measurement rate also helps to reduce power consumption and prolong the longevity of the device. However, for certain use cases and demo purposes, faster measurement intervals might be necessary. Therefore, an adaptive polling algorithm has been implemented that can be used to adjust the measurement rate based on the CO2 concentration changes. This algorithm has the benefit of maintaining accuracy and reducing power consumption while still ensuring fast measurements when required.

Code for adaptive polling

```
1    PAS_delta_change = 1.0 * (100.0 / (co2ppm_old + 1) * co2ppm) - 100;
2    If (abs(PAS_delta_change) >= 7.0 || abs(co2ppm_old - co2ppm) >= 75)
   {
3      if (PERIODIC_MEAS_INTERVAL_IN_SECONDS != 5) {
4        PERIODIC_MEAS_INTERVAL_IN_SECONDS = 5;
5        PAS_ModeSwitch = 1;
6      } else {
7        PAS_ModeSwitch = 0;
8        Serial.println("PAS: already in FAST mode");
9      }
10   } else {
11     if (PERIODIC_MEAS_INTERVAL_IN_SECONDS != 60) {
12       if (PAS_min_FAST_records == 0) {
13         Serial.println("PAS: switch to SLOW mode");
14         PERIODIC_MEAS_INTERVAL_IN_SECONDS = 60;
15         PAS_ModeSwitch = 1;
16         PAS_min_FAST_records = 10;
17       } else {
18         Serial.print("PAS: IGNORE switch to SLOW mode: ");
19         Serial.print(PAS_min_FAST_records);
20         Serial.print(" Polling speed: ");
21         Serial.println(PERIODIC_MEAS_INTERVAL_IN_SECONDS);
22         PAS_min_FAST_records--;
23       }
24     } else {
25       PAS_ModeSwitch = 0;
26       Serial.println("PAS: already in SLOW mode");
27     }
28   }
29   co2ppm_old = co2ppm;
30   if (PAS_ModeSwitch = 1) {
31     err = cotwo.startMeasure(PERIODIC_MEAS_INTERVAL_IN_SECONDS);
32   }
```


Downloaded from Arrow.com.

The algorithm above is designed to adjust the measurement rate of the sensor based on changes in CO2 concentration. The algorithm starts by calculating the percentage change in CO2 concentration from the previous measurement using the PAS_delta_change variable. If the absolute value of the change is greater than or equal to a set threshold value (in this case 7.0) or the difference between the current and previous CO2 concentration is greater than or equal to 75 ppm, the measurement rate is changed to 5 seconds to increase frequency. If the change is less than the specified threshold, the measurement rate is changed to 60 seconds to reduce frequency. The algorithm includes a minimum number of fast measurements before transitioning back to the slow measurement rate. This way, the algorithm ensures that the measurements are taken with an optimal interval, reducing the number of unnecessary measurements and conserving power while maintaining accuracy and stability. The code snippet can be integrated in the user's routine or also in the provided Arduino library.

5.3 Arduino library

The core library is C based and provides a platform-independent driver for the XENSIV™ PAS CO2 sensor. It provides full access to all features of the sensor. The driver consists of 4 files.

Table 3 Core C driver

Source code	Description
xensiv_pasco2_ver.h	Contains the exact version of the XENSIV™ PAS CO2 sensor
xensiv_pasco2_regs.h	Contains the register definitions for interacting with the XENSIV™ PAS CO2 sensor
xensiv_pasco2.h	Contains full functions for interacting with the XENSIV™ PAS CO2 sensor
xensiv_pasco2.c	Contains full functions for interacting with the XENSIV™ PAS CO2 sensor

Full documentation and overview to the driver can be found on the GitHub.

https://github.com/Infineon/sensor-xensiv-pasco2

And the documentation with detail explanation for all macros, enumerations and functions can be found here:

https://infineon.github.io/sensor-xensiv-pasco2/html/group__group__board__libs.html

The Arduino library is using this core C driver with a C++ wrapper to follow the ecosystem design pattern so that Arduino users find in this library what they are used to.

Table 4 Arduino library

Source code	Description
pas-co2-pal-ino.cpp	Target platform-specific implementation
pas-co2-platf-ino.hpp	Default board definition and selection by conditional compiling
pas-co2-ino.hpp	Contains functions for interacting with the XENSIV™ PAS CO2 sensor adapted from the core driver
pas-co2-ino.cpp	Contains functions for interacting with the XENSIV™ PAS CO2 sensor adapted from the core driver

Downloaded from Arrow.com.

Quick start with the Arduino Due

The Arduino library includes 6 examples which the user can modify and use as a reference.

Table 5 Arduino examples

Example	Description
alarm-notification	Readout of the sensor CO2 concentration based on threshold crossing and synched via hardware interrupt
device-id	Readout of the sensor devices product and revision identifiers
single-shot-mode	Readout of the sensor CO2 concentration value using single shot measurement mode
continuous-mode	Readout of the sensor CO2 concentration value using continuous measurement mode
forced-compensation	Set CO2 reference offset using forced compensation
early-notification	Readout of the sensor CO2 concentration based on early notification synched via hardware interrupt

Full documentation of the Arduino library can be found on the GitHub.

https://github.com/Infineon/arduino-pas-co2-sensor

Downloaded from Arrow.com.

6 UART interface

When UART is selected as serial communication interface with setting pin PSEL to high, the device acts as an UART slave. As a result, it is recommended that the master uses a time out mechanism. The device operates via UART for point to point communication and therefore bus operation is not supported.

The basic format of a valid UART frame is: 1 start bit, 8 data bits, no parity bit and 1 stop bit. The baud rate is 9.6kbps. The master combines several UART frame into a message (read or write). The combination of master request and salve answer defines a transaction.

If the device detects a valid incoming message, it shall respond it with an acknowledge frame. Otherwise, it will issue a NAK notification.

It shall be noted that when a measurement sequence is initiated, the device does not respond to any incoming frame for the duration of the measurement sequence (either ACK or NAK). A message sent during a measurement sequence shall be therefore resent by the master once the measurement sequence is completed.

The device does not support bulk read and write operations. Only singly data bytes can be read or written within one transaction.

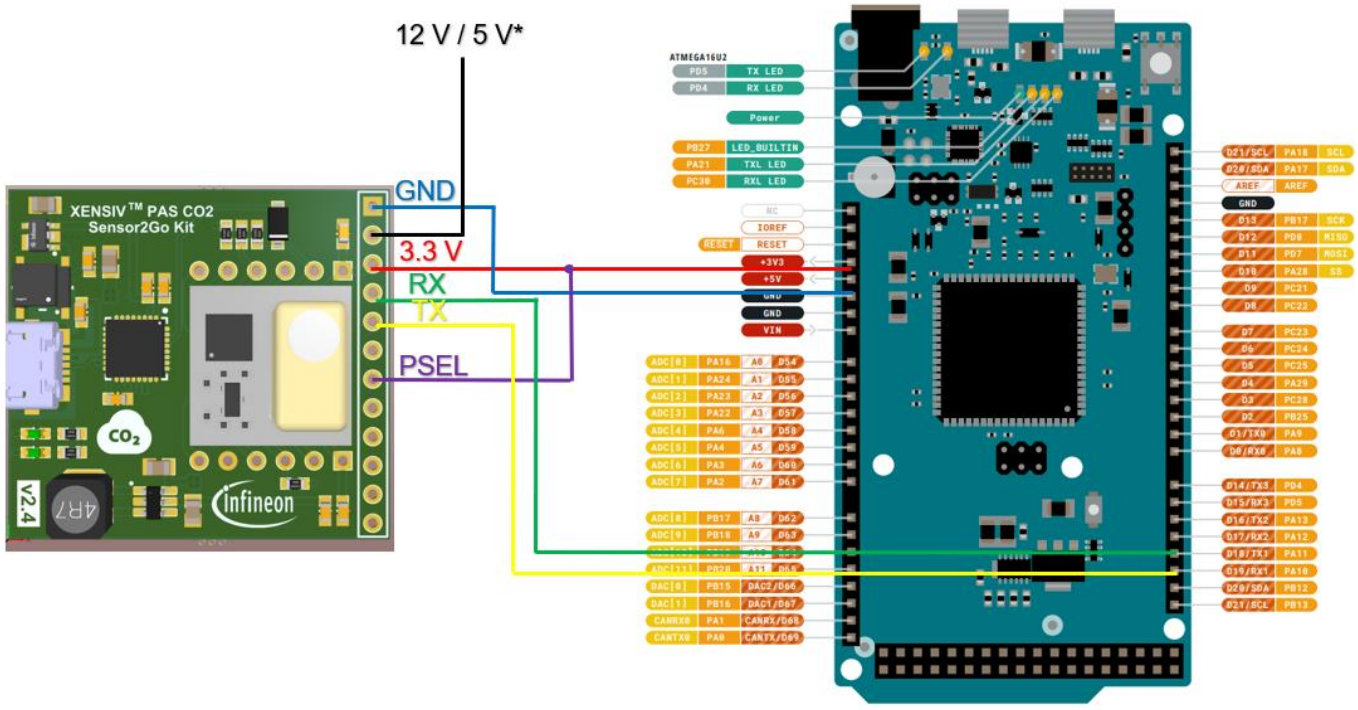


Figure 13 XENSIV™ PAS CO2 Sensor2Go Kit UART interface connection to the Arduino Due[4)]

Downloaded from Arrow.com.

6.1 Write transactions

A Write transaction is initiated by the message made by the frame sequence below:

Frame	Description	Frame Payload	Comments
1	Initiate request	0x57 (ASCII code for “W”) or 0x77 (ASCII code for “w”)	
2	Delimiter	0x2C (ASCII code for “,”)	
3	Address 1 (4 most significant bits)	ASCII code of the hex value of the 4 most significant bits of the register address.	E.g.: in order to write register at address $0F_H$, the payload should be 0x30 (ASCII code for ”0”).
4	Address 2 (4 least significant bits)	ASCII code of the hex value of the 4 least significant bits of the register address.	E.g.: in order to write register at address $0F_H$, the payload should be 0x46 (ASCII code for ”F”)
5	Delimiter	0x2C (ASCII code for “,”)	
6	Data 1 (4 most significant bits)	ASCII code of the hex value of the 4 most significant bits of the written data.	E.g.: in order to write data 42_H, the payload should be 0x34 (ASCII code for ”4”).
7	Data 2 (4 least significant bits)	ASCII code of the hex value of the 4 least significant bits of the written data.	E.g.: in order to write data 42_H, the payload should be 0x32 (ASCII code for ”2”).
8	End of message	0x0A (ASCII code for line feed “\n”)	
1-8	Full message	0x57 0x2C 0x30 0x46 0x2C 0x34 0x32 0x0A	Combined and translated format: W,0F,42\n

At the end of the reception the incoming message, the device answers with an answer message.

Downloaded from Arrow.com.

UART interface

If the write operation is valid, the device (slave) answers it with the following message:

Frame	Description	Value	Comments
1	ACK	0x06 (ASCII code for “ACK”)	
2	End of message	0x0A (ASCII code for line feed “\n”)	

If the write operation is not valid, the device answers with the following message:

Frame	Description	Value	Comments
1	NAK	0x15 (ASCII code for “NAK”)	
2	End of message	0x0A (ASCII code for line feed “\n”)	

6.2 Read transactions

A Read transaction is initiated by the message made by the frame sequence below:

Frame	Description	Frame Payload	Comments
1	Initiate request	0x52 (ASCII code for “R”) or 0x72 (ASCII code for “r”)	
2	Delimiter	0x2C (ASCII code for “,”)	
3	Address 1 (4 most significant bits)	ASCII code of the hex value of the 4 most significant bits of the register address.	E.g.: in order to write register at address $0F_H$, the payload should be 0x30 (ASCII code for ”0”).
4	Address 2 (4 least significant bits)	ASCII code of the hex value of the 4 least significant bits of the register address.	E.g.: in order to write register at address $0F_H$, the payload should be 0x46 (ASCII code for ”F”)
5	End of message	0x0A (ASCII code for line feed “\n”)	
1-5	Full message	0x52 0x2C 0x30 0x46 0x0A	Combined and translated format: R,0F\n

Downloaded from Arrow.com.

UART interface

At the end of the reception the incoming message, the device answers with an answer message.

If the read operation is valid, the device (slave) answers it with the following message:

Frame	Description	Value	Comments
1	Data 1 (4 most significant bits)	ASCII code of the hex value of the 4 most significant bits of the written data.	
2	Data 2 (4 least significant bits)	ASCII code of the hex value of the 4 least significant bits of the written data.	
3	End of message	0x0A (ASCII code for line feed “\n”)	

If the read operation is not valid, the device answers with the following message:

Frame	Description	Value	Comments
1	NAK	0x15 (ASCII code for “NAK”)	
2	End of message	0x0A (ASCII code for line feed “\n”)	

6.3 Arduino examples

Following the example commands of the write and read transactions above, a small Arduino example is shown. The scratch pad register is used which provides a readable and writable field for testing.

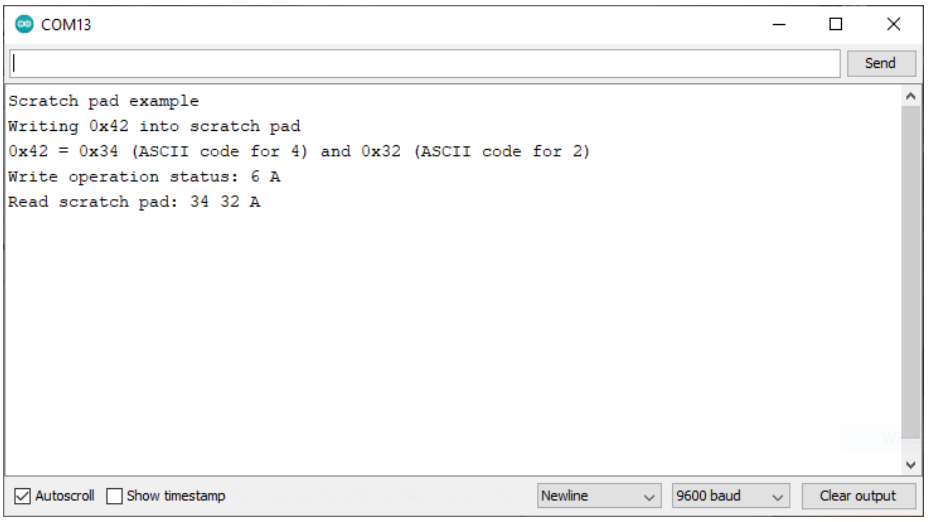


Figure 14 Scratch pad example via UART

Downloaded from Arrow.com.

UART interface

Scratch pad example

```
byte i;
int incomingByte[4];

void setup() {

  Serial.begin(9600);
  Serial1.begin(9600);

  Serial.println("Scratch pad example");
  Serial.println("Writing 0x42 into scratch pad");
  Serial.println("0x42 = 0x34 (ASCII code for 4) and 0x32 (ASCII code for 2)");

  // Write into scratch pad
  Serial1.write("W,0F,42\n");
  delay(50);

  // Read scratch pad
  Serial1.write("R,0F\n");
  delay(50);

  // Write received data to buffer
  while (Serial1.available() > 0) {
    incomingByte[i] = Serial1.read();
    i++;
  }

  // Read received data from buffer
  Serial.print("Write operation status: ");

  for (i = 0; i < 2; i = i + 1) {
    Serial.print(incomingByte[i], HEX);
    Serial.print(" ");
  }

  Serial.println();

  Serial.print("Read scratch pad: ");
  for (i = 2; i < 5; i = i + 1) {
    Serial.print(incomingByte[i], HEX);
    Serial.print(" ");
  }
}

void loop() {

}
```


Downloaded from Arrow.com.

UART interface

The next example is showing how to use the write and read transactions to start a measurement in continuous mode and reading out the CO2 concentration. This example includes a function which translates the ASCII values to HEX values for correct readings in ppm.

Continuous mode with ASCII conversion example

```
byte i;
char incomingByte[255], CO2_MSB[255], CO2_LSB[255];
int CO2MSB, CO2LSB;
long res = 0L ;

void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);

  Serial1.write("R,00\n");
  delay(50);

  Serial.println("Read FW Version");

  // Write received data to buffer
  while (Serial1.available() > 0) {
    incomingByte[i] = Serial1.read();
    i++;
  }

  // Read received data from buffer
  Serial.print("Response: ");
  for (i = 0; i < 3; i = i + 1) {
    Serial.print(incomingByte[i]);
    Serial.print(" ");
  }

  Serial.println("Start measurement with 10s sampling rate");

  // Idle mode
  Serial1.write("W,04,00\n");
  delay(100);

  // Set measurement rate to 10 s
  Serial1.write("W,02,00\n");
  delay(100);

  Serial1.write("W,03,0A\n");
  delay(100);

  // Configure continuous mode
  Serial1.write("W,04,02\n");
  delay(12000);

  // Write received data to buffer
  while (Serial1.available() > 0) {
    incomingByte[i] = Serial1.read();
    i++;
  }
}
```


Downloaded from Arrow.com.

Continuous mode with ASCII conversion example

```
51
52 void loop() {
53
54   i = 0;
55   Serial1.write("R,05\n");
56   delay(50);
57
58   // Write received data to buffer
59   while (Serial1.available() > 0) {
60     CO2_MSB[i] = Serial1.read();
61     i++;
62   }
63
64   for (i = 0 ; i < 2 ; i++) {
65     res <<= 4 ;
66     res += hex2bin (CO2_MSB[i]) ;
67   }
68   CO2MSB = res;
69
70   i = 0;
71   Serial1.write("R,06\n");
72   delay(50);
73
74   // Write received data to buffer
75   while (Serial1.available() > 0) {
76     CO2_LSB[i] = Serial1.read();
77     i++;
78   }
79
80   for (i = 0 ; i < 2 ; i++) {
81     res <<= 4 ;
82     res += hex2bin (CO2_LSB[i]) ;
83   }
84   CO2LSB = res;
85
86   int16_t result = CO2MSB << 8 | CO2LSB;
87   Serial.print("CO2: ");
88   Serial.print(result);
89   Serial.println(" ppm");
90
91   delay(10000);
92 }
93
94 int hex2bin (char c)
95 {
96   if (c >= '0' && c <= '9')
97     return c - '0' ;
98   if (c >= 'A' && c <= 'F')
99     return c - 'A' + 10 ;
100         if (c >= 'a' && c <= 'f')
101           return c - 'a' + 10 ;
102       }
```


Downloaded from Arrow.com.

7 PWM interface

In case no communication interface is available PWM_DIS pin can be used to control the device. PWM_DIS is first asserted after the power on boot sequence (not after soft reset) and the level of the pin is checked. If a low level is detected, an internal interrupt routine configures the device into continuous mode and a measurement sequence is started. At the end of each measurement sequence, the level of pin PWM_DIS is polled. If it is high, then the device is configured back to idle mode and output pin PWM is disabled.

Pin PWM offers the possibility to read out the CO2 concentration by delivering a PWM signal whose timing information contain the CO2 concentration value. At the end of each measurement sequence, the device updates the PWM timing with the measured CO2 concentration. To enable the PWM output two conditions must be met:

- PWM output needs to be enabled by software in the measurement mode configuration register
- PWM_DIS pin needs to be set to GND

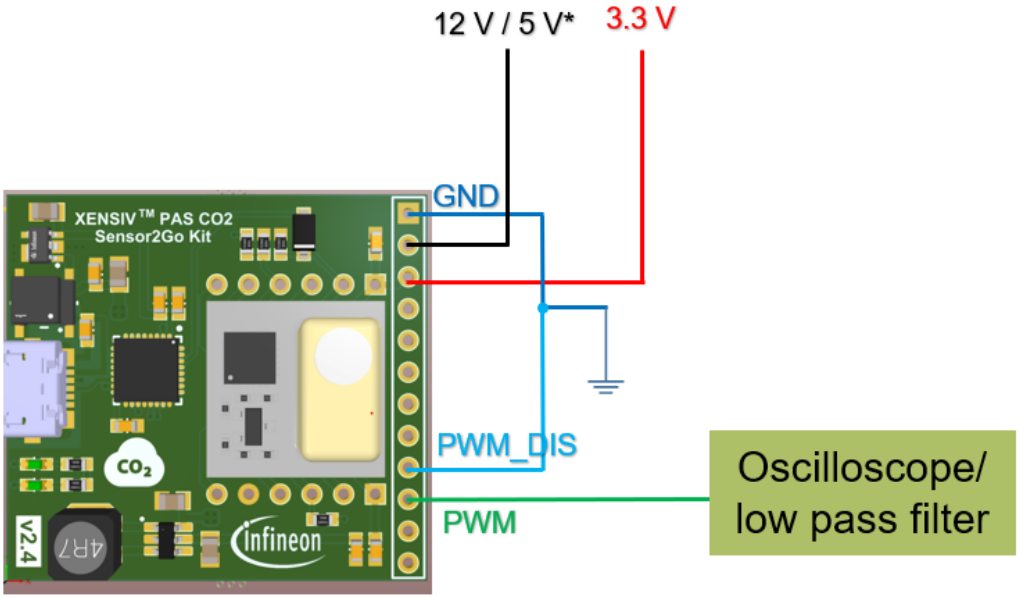


Figure 15 XENSIV™ PAS CO2 Sensor2Go Kit PWM connection example

The main specifications of the PWM signal are summarized below. The output signal can be converted by either directly measuring the pulse-duration or alternatively by employing a low-pass filter and measuring the output voltage.

Parameter	Value
Base frequency	80Hz
Duty cycle	Linear from 0% (0ppm) to 100% (10,000ppm)
Resolution	1 ppm (1.25µs)

Downloaded from Arrow.com.

PWM interface

Typically, the PWM signal is converted to a voltage signal via a low pass filter. Since there's an inherent trade-off between settling time, ripple and current consumption, the ideal parameterization of the low pass filter differs depending on the application. It needs to be considered that the ripple on the PWM introduces potentially an error on the CO2 concentration reading. An estimation on the introduced error is shown below.

Concentration	Duty cycle	Expected error	ON time	Target voltage	Ripple	Error introduced
400 ppm	4%	+/- 42 ppm	0.5 ms	0.132V	+/- 8mV	+/- 24 ppm
5000 ppm	50%	+/- 180 ppm	6.25ms	1.65V	+/- 50mV	+/-150 ppm

The number of PWM pulse issued at pin PWM depend on the device's configuration which is covered by the measurement mode configuration register. In PWM single pulse mode only a single PWM pulse is generated before the device goes inactive. In PWM pulse train mode, a pulse train of 160 pulses (approx. 2sec) is issued before the device goes inactive. For calculating in single pulse mode, the high duty time of the pulse needs to be compared against the 80 Hz frequency considering the resolution of 1 ppm equals to 1.25 µs. For calculating in pulse train mode, the duty cycle needs to calculated considering the values 0% (0ppm) to 100% (10,000ppm).

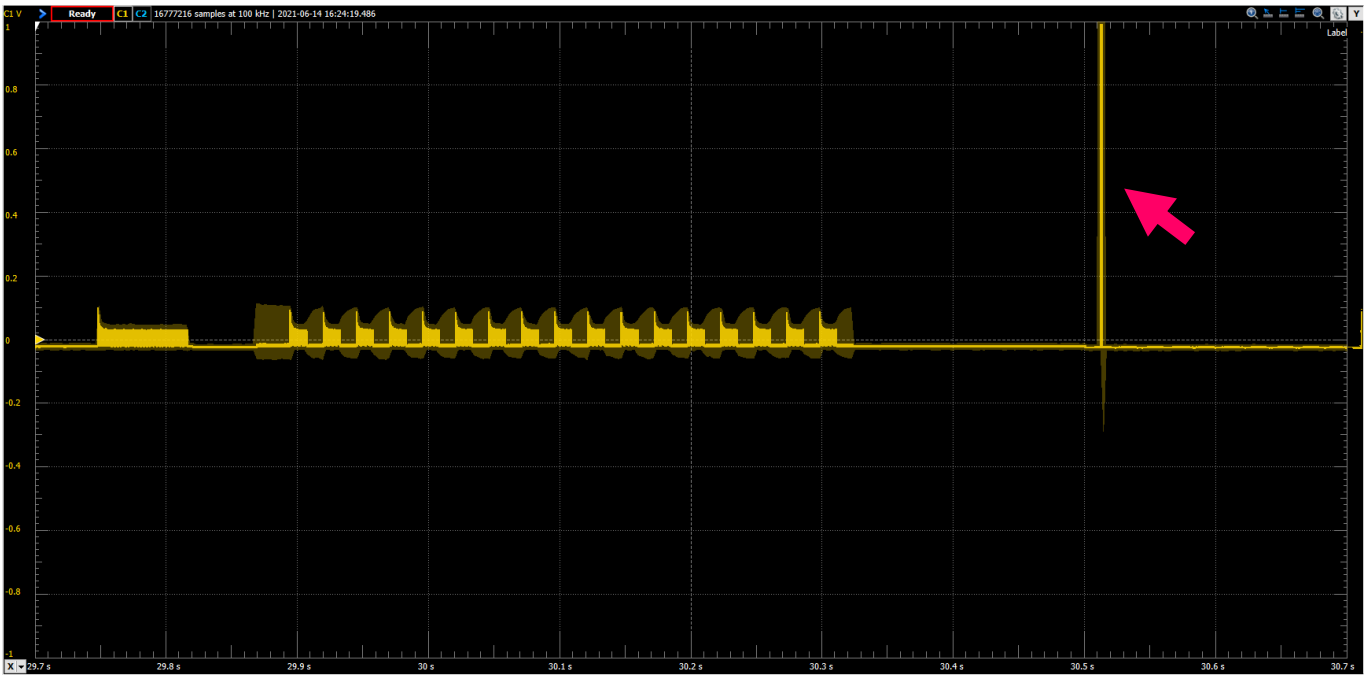


Figure 16 PWM output in single pulse mode

Downloaded from Arrow.com.

PWM interface

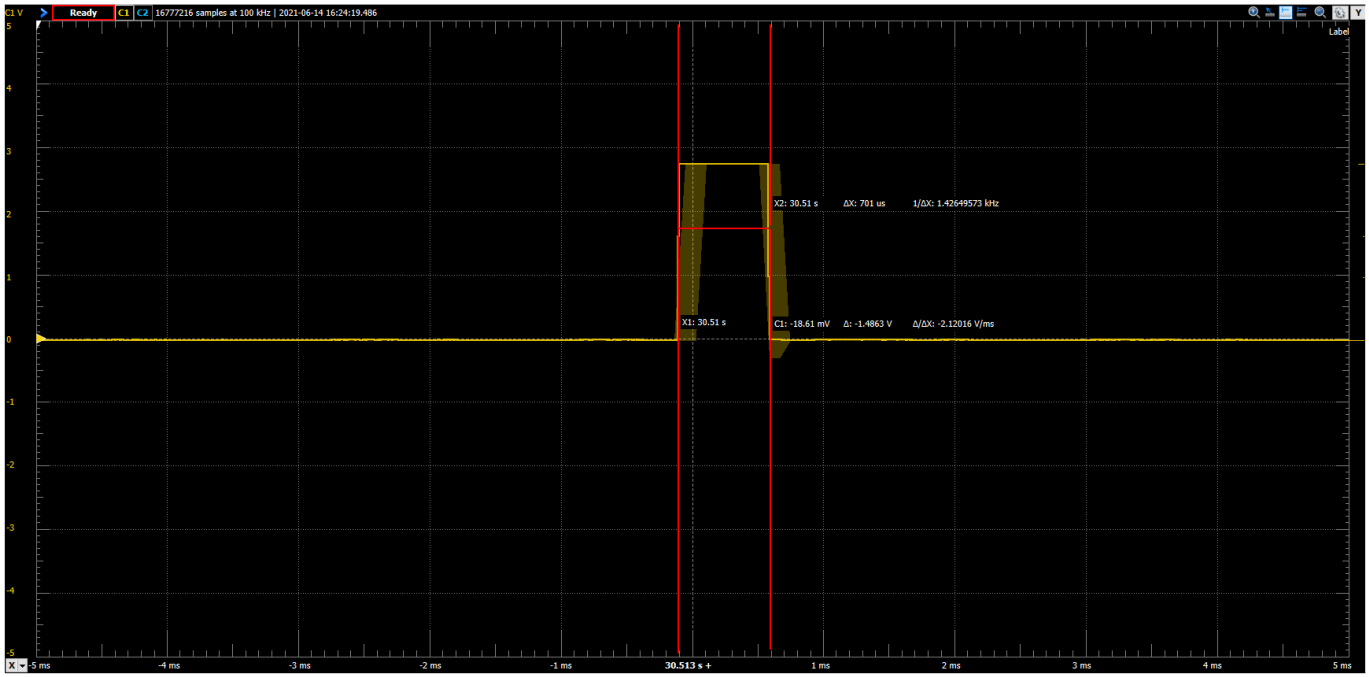


Figure 17 PWM output in single pulse mode (zoomed to pulse)

The measured high duty time of the PWM pulse in the example in Figure 14 is 701 µs. Calculating from the resolution where 1.25 µs equals to 1 ppm, the calculated CO2 concentration reading is 560 ppm.

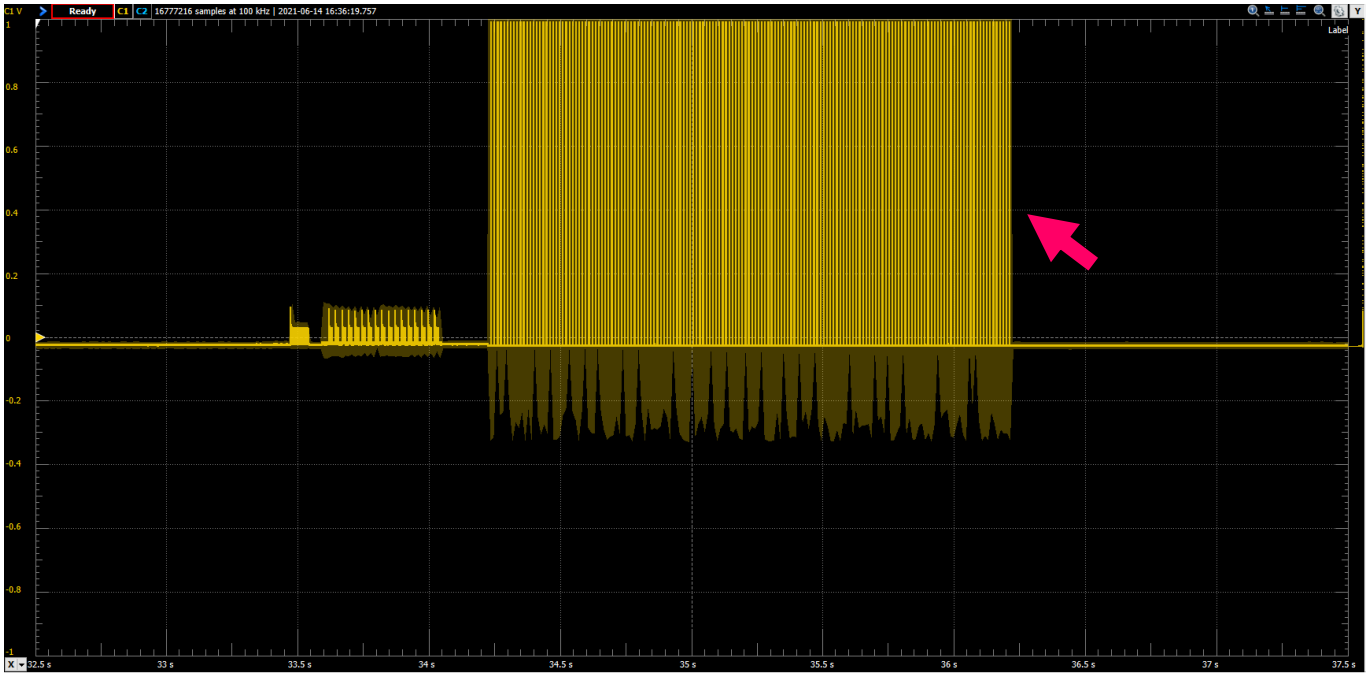


Figure 18 PWM output in train pulse mode

Downloaded from Arrow.com.

PWM interface



Figure 19 PWM output in train pulse mode (zoomed to pulse)

The measured high duty time of one PWM pulse of the PWM pulse train in the example in Figure 16 is 631.18 µs. The measured low duty time is 11.76 ms which results in a duty cycle of 5.37 % (= 537 ppm).

In case of any technical questions please visit our community forum and have a look if similar questions are already posted or create a new one.

https://community.infineon.com/t5/CO-sensor/bd-p/CO2Sensors

Downloaded from Arrow.com.

Revision history

Document version	Date of release	Description of changes
V 1.0	04.11.2020	Creation
V 2.0	01.07.2021	Added description to UART, PWM and additional functionality
V 2.1	01.07.2022	Added section for available libraries
V 2.2	01.04.2024	Added UART examples, pressure calculation based on altitude, adaptive polling and XENSIV™ PAS CO2 5V relevant information

Downloaded from Arrow.com.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-04-01

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2024 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AN_2011_PL38_2011_134235

IMPORTANT NOTICE

The information contained in this user manual is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this user manual must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this user manual.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.

Downloaded from Arrow.com.