E+E *IHR PARTNER IN DER SENSOR TECHNOLOGIE*
ELEKTRONIK ®
Ges.m.b.H.

APPLICATION NOTE   **AN1808-1**

Application Note

Utilising the E2 Interface on EE894

Rev. 1.2 07/2020

Introduction:

The E2 interface is used for the digital, bidirectional data transmission between a master and a slave device.
The data transmission takes place via synchronous and serial modes, the master being responsible for generating the clock signal. The slave cannot send any data independently.

This document illustrates the use of an E2 Interface (ref. [1]) based on a simple example: the temperature, the relative humidity, the CO_2 concentration, the ambient pressure and the status of an E+E measuring device are to be read periodically via E2 Interface. It provides a brief hardware description, explains the data transmission principle and gives a software example (in language C).

CONTENT

1 Hardware Setup

The E2 Interface is designed for a master/slave setup. In this example the master is an 8051-based processor. The pins P0.2 (SCL) and P0.3 (DAT) are used as clock and data lines. Both pins are configured as open drain I/O pins and connected to the bus-high-voltage via two external pull-up resistors.
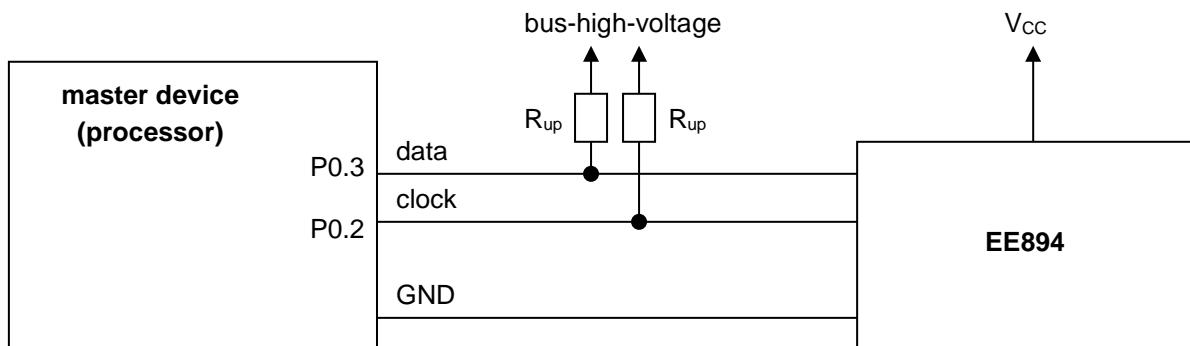
Figure 1: Hardware master / slave setup

Note: _Observe the compatibility of the voltage levels between the E2 Interface levels and the master processor._

2 Readable Parameters

Following parameters/values [hex] can be read via E2 interface:

Command:	Format	Kind / Description	Return value	Unit
Group (two bytes)	unsigned integer		**EE894:** 0x037E (894$_d$)	
Sub-group	byte		0x09	
Available measured variables	byte		0x0F	
Statusbyte:[1]	byte		0x0_	
Measuring value 1:	unsigned integer	rel. humidity	0 … 10000 (0 % … 100 %)	0.01 %
Measuring value 2:	unsigned integer	temperature	23315 … 35815 (-40 °C … +85 °C)	0.01 K
Measuring value 3:	unsigned integer	ambient pressure	3000 … 11000 (300 mbar … 1100 mbar)	0.1 mbar
Measuring value 4:	unsigned integer	CO_2 concentration	0 … range	ppm

[1] Gives information on whether last measurement was successful

2.1 Available Parameters in the Custom Area

- Firmware-Mainversion
- Firmware-Subversion
- Offset RH, T, p, CO_2
- Gain RH, T, p, CO_2
- Upper calibration point RH, T, p, CO_2
- Lower calibration point RH, T, p, CO_2
- Last customer adjustment of RH, T, p, CO_2
- Serial number
- Part name
- Error code
- Global measurement time interval

2.2 Electrical Requirements

Symbol	Parameter	Minimum	Maximum	Unit	Remark
V_{DD}	bus-high-voltage	3.3	5.2	V	
f_{CLK}	clock frequency	500	5000	Hz	The highest achievable data rate depends on the combination of line capacity and the pull-up resistors.
Rup	pull-up resistor	4.7	100	$k\Omega$	

2.3 Error Code List

Error Code	Description
1	supply voltage low detected

3 Data Transmission Method

The only command necessary for data requests is "read byte from slave" (ref. [1]), which is bidirectional and allows one single data byte to be sent from the slave to the master. The master notifies the slave with a control byte which data byte is required. The slave answers within the same frame with the requested data byte and a checksum. For transmitting a value consisting of several bytes, the master shall send the command "read byte from slave" for each one of the bytes (multi-stage transmission).

The detailed structure of the command "read byte from slave" is described in ref. [1] (Section 2.3.1.).

4 Measurement Timing

	Minimum	Typical	Maximum
t_{pwrup*} (power up)	4.7s		16.2s
t_{meas} (measurement)		0.8s	
t_{mti} (measurement time interval) ±6.25%	15s		3600s

* see chapter 6.1

Examples:
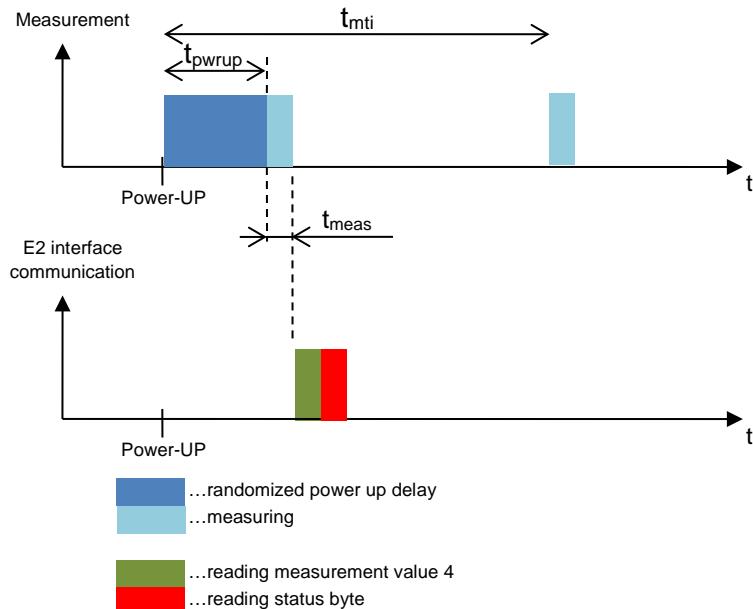
5 Timing for Write Commands

Writing a byte to the device (with control byte 0x10) takes ≤150ms and can be done by writing the flash memory. During the writing time E2 communication interrupts are deactivated. The attempt to communicate with the device while the flash is being written forces the clock low extension which holds the clock line low until the write routine has finished.

Note: When writing the measurement interval (address 0xC6 and 0xC7) both values will be written together into the flash. Writing will start after sending both bytes and will cause a communication delay of ≤300ms.

6 Optimizing the Power Consumption

6.1 Operation Modes

EE894 module is designed to change its operation mode based on the actual status of measurement or communication. The supply current is different for each operation mode and it is shown given below as well as in Figure 3.

Mode	Supply current	Description
Sleep mode	typ. 410 μA	The module is waiting for measurement or communication request
Warm-up mode	typ. 10 mA	The module is in warm-up mode. Duration 450 ms before a measurement is taken.
Communication mode	typ. 10 mA	Initiated by an interrupt on the E2 Bus and lasts as long as the communication is ongoing
Measuring mode	max. 350 mA	current peak, caused by flashing the infrared lamp. For details see Figure 2.
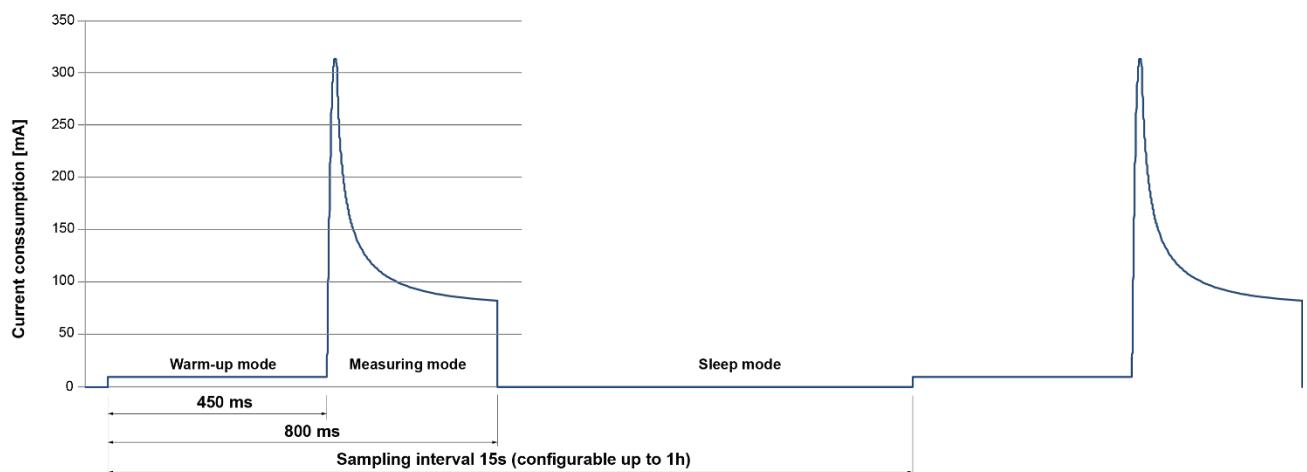
Figure 2: Power Consumption

Setting a longer measuring interval extends the sleep mode time.

After a reset the module starts the first measurement after 5s to 15s; the timing (t_{pwrup}) is defined by a randomizer and is specific for each EE894. The randomizer is constant for each power up but varies from module to module.
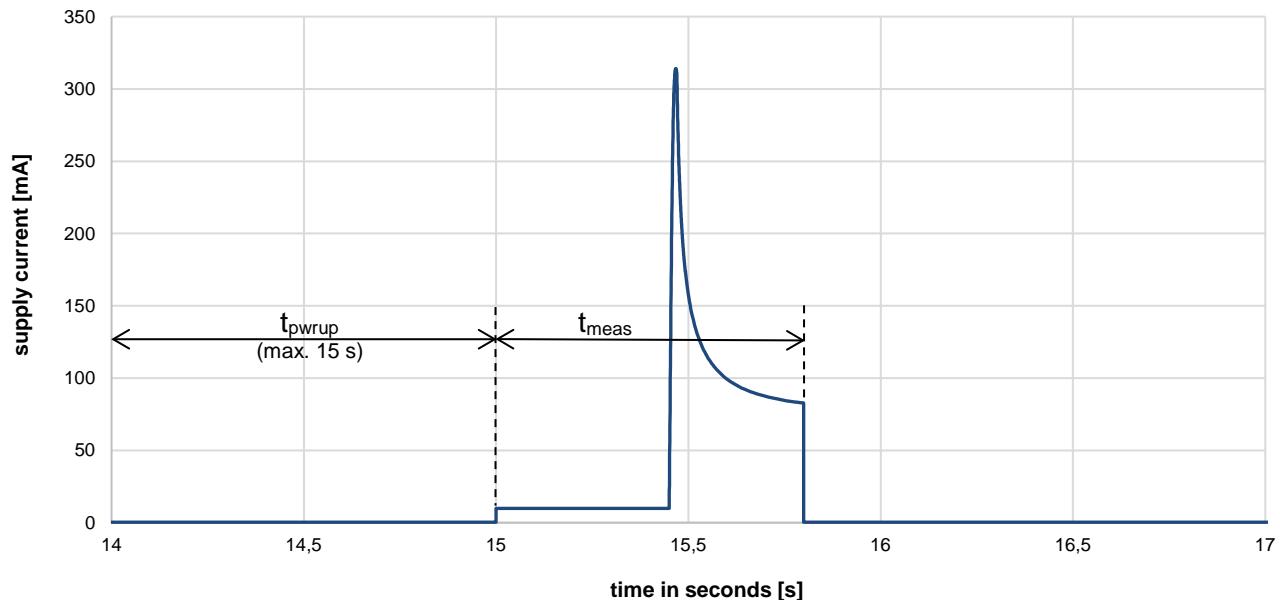
Figure 3: Supply current in measuring mode @23°C

6.2 Measuring Time Interval

The peak current during the measurement mode depends on the infrared lamp and cannot be reduced. The average power consumption can be reduced by increasing the measuring time interval.

The measuring time interval can be set by writing the interval time to custom addresses 0xC6 and 0xC7 as an unsigned integer value in units of 0.1 seconds. Figure 4 shows the impact of the measuring time interval on the average supply current.
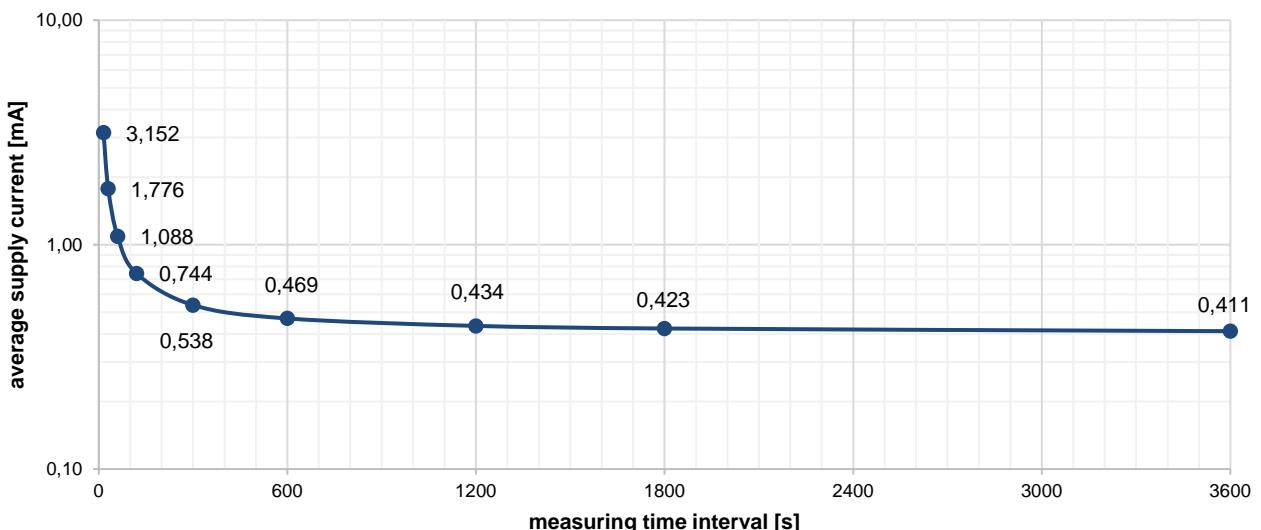
Figure 4: Average supply current as function of the measuring interval

6.3 Communication

Communication on the bus forces the module to switch to communication mode. The power consumption of the CO_2 module can be reduced by avoiding unnecessary communication.
For low power consumption it is important to avoid disturbances on the bus, which would prevent the CO_2 module from getting into sleep mode.

7 Setting the Measuring Time Interval

Following steps are necessary for setting the measurement time interval.

7.1 Check Current Measuring Time

- Set the read pointer with control byte 0x50 to address 0xC6
- Read the global measurement interval low-byte with control byte 0x51
- Read the global measurement interval high-byte with control byte 0x51 (pointer increments automatically after reading a byte)
- Calculate the measuring time interval using the formula:

$$measuring\ time\ interval = \frac{measuring\ time\ interval\ low\ byte + measuring\ time\ interval\ high\ byte * 256}{10}$$

Example:

$$measuring\ time\ interval = \frac{150 + 0 * 256}{10} = \frac{150}{10} = 15s$$

7.2 Set the Measuring Time Interval

Setting the measuring time interval requires always writing both the low and the high byte of the global measuring interval (bytes 0xC6 and 0xC7 in the custom area). Values smaller than 150 (15 s) and higher than 36000 (3600 s) will be ignored by the firmware.

Calculation of byte values:

$$measuring\ time\ interval\ low\ byte = (measuring\ time\ interval * 10)\ MOD\ 256$$

$$measuring\ time\ interval\ high\ byte = (measuring\ time\ interval * 10)/256$$

MOD…modulo operation
For division only integer values are relevant.

Example for 60s interval:

$$measuring\ time\ interval\ low\ byte = (60 * 10)\ MOD\ 256 = 600\ MOD\ 256 =\ 88$$

$$measuring\ time\ interval\ high\ byte = \frac{(60*10)}{256} = \frac{600}{256} = 2$$

Writing values to the custom area:
- Write the low byte with control byte 0x10 (*direct write) to custom area* and address 0xC6.
- Write the high byte with control byte 0x10 (*direct write) to custom area* and address 0xC7.

8   Measured Value Request

The following sequences explain the request of data for the standard interface address "0". For other addresses please view the definition of the control byte in ref. [1] (chapter 2.3.1.).

8.1   Relative Humidity

The data transmission method is explained using a multi-stage relative humidity (RH) request. For transmitting a 16 bit value the command "read byte from slave" must be executed twice.
The RH value corresponds to "measurement value 1"of the module.
For consistent data it is necessary to request first the low-byte of a measured value (ref. [1]).

Steps for RH value request:

Step 1
Perform "read byte from slave" command for reading the relative humidity low byte (0x81):

- Apply start condition and control byte (0x81) to the bus
- Read in and check ACK/NACK of the slave
- Read in data byte relative humidity low byte (rh_low)
- Send acknowledgement to the slave
- Read in checksum from the slave
- Send NACK and stop condition to the slave (the first „read byte from slave" command is thereby completed)
- Verify the checksum

If the checksum is correct, the second „read byte from slave" command can be performed for reading in the relative humidity high-byte:

Step 2

- Apply Start condition and control byte (0x91) to the bus
- Read in and check ACK/NACK of the slave
- Read in data byte relative humidity high-byte (rh_high)
- Send acknowledgement to the slave
- Read in checksum from the slave
- Send NACK and stop condition to the slave (The second „read byte from slave" command is thereby completed)
- Verify the checksum

Step 3
Upon a positive verification of the checksum, the master determines the RH value.
It combines the high and low bytes to form a 16-bit value and divides this by 100.

rel. humidity in % = (rh_high * 256 + rh_low) / 100

Figure 5 provides the flow chart of this sequence.
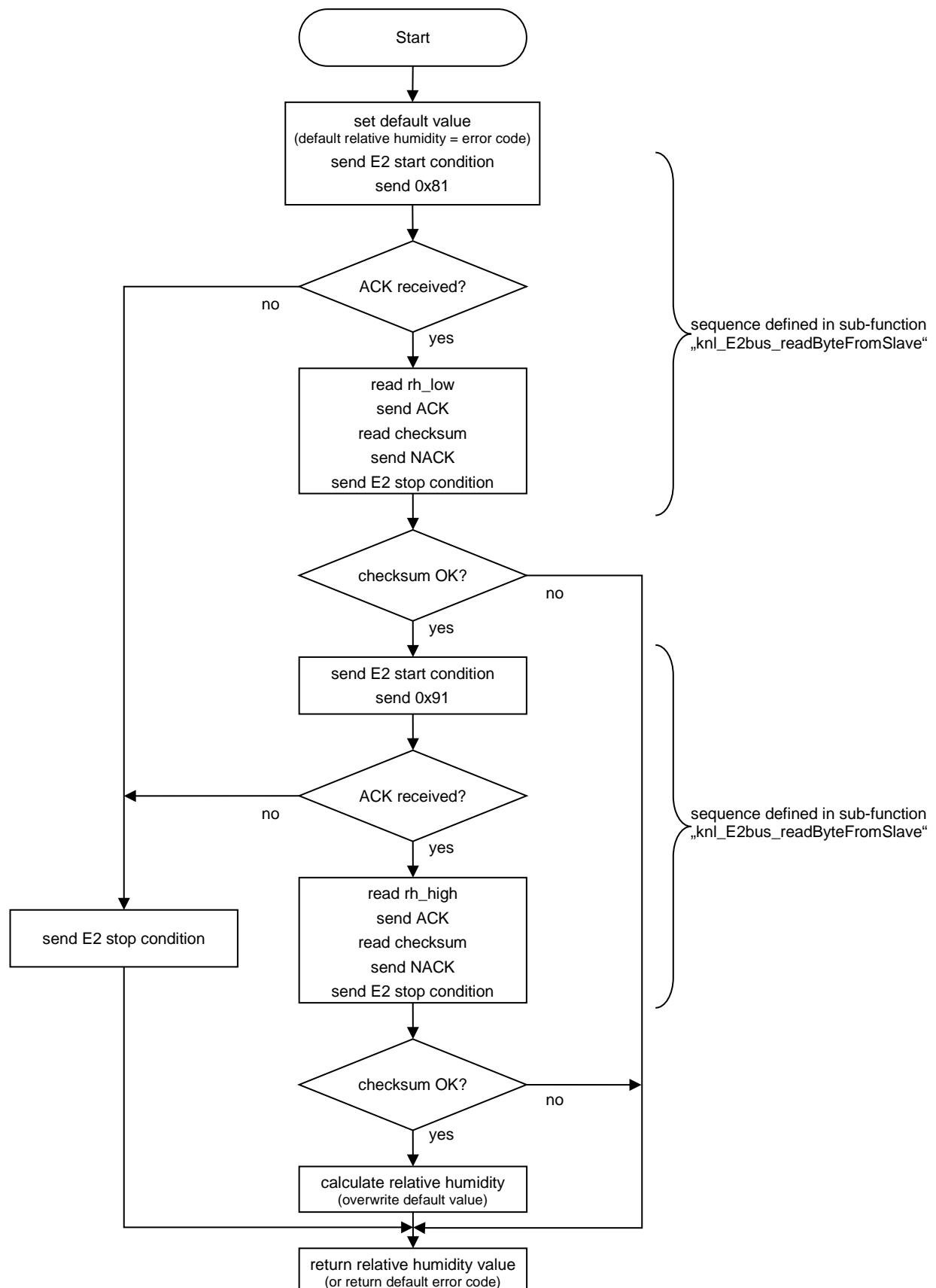
Figure 5: Flow chart of a RH value request (= simplified flow of function "fl_E2bus_Read_RH")

8.2 Temperature

The temperature value can be read similar to relative humidity as described above. The control bytes are 0xA1 for the low byte and 0xB1 for the high byte of the measured ambient pressure value. The actual temperature value is calculated as follows:

Temperature in °C = (Temp_high * 256 + Temp_low) / 100 – 273.15

8.3 Ambient Pressure

The ambient pressure value can be read similar to relative humidity as described above. The control bytes are 0xC1 for the low byte and 0xD1 for the high byte of the measured ambient pressure value. The actual ambient pressure value is calculated as follows:

Ambient pressure in mbar = (pres_high * 256 + pres_low) / 10

8.4 CO_2 Concentration

The CO_2 concentration can be read similar to relative humidity as described above. The corresponding control bytes for median filtered CO_2 values are 0xE1 for low byte and 0xF1 for high byte of the measured CO_2 concentration. The actual CO_2 concentration is calculated as follows:

CO_2 in ppm = (CO2_high * 256 + CO2_low)

8.5 Status Request

To guarantee the validity of values following the measured value requests, one shall read the status of the slave. This is done by performing a further "read byte from slave" command and the control byte 0x71 applied to the bus.
After receiving the status byte, the checksum shall be read in and verified (Figure 5).
As shown in ref. [1], the first bit (Bit 0/LSB) is assigned to the measured value for the relative humidity, the second bit (Bit 1) for temperature, the third bit (Bit 2) for ambient pressure (deviating from [1]) and the fourth bit (Bit 3) for CO_2 concentration.
These bits provide information on the validity the measured values: a low signal ("0") indicates that a correct measured value has been received, while a high signal ("1") indicates a faulty measured value, caused for example by an insufficient power supply.

9   Software Examples

9.1   General

In this application note, the functions executed by the E2 interface are grouped together in separate software modules. This allows for simple integration and reusability of the code.
By including the header file in the main module of the master code as specified below, the example functions can be used directly for reading the relative humidity, the temperature, the ambient pressure, the CO_2 concentration and the status byte.
The header files specify the required definitions of the variables and the function prototypes for creating a simple software module for the E2 Interface.

9.2   Main Software Module

After the initialisation, typically customer-specific actions are executed in a continuous loop in the main module. One option calling the interface routines in a favourable sequence is provided by using symbols.

9.2.1   Code Example

```
:
#include "fl_E2bus.h"
:
:
void main (void)
{
   unsigned int SensorType;             // Variable for Sensortype
   unsigned char AvPhMes;               // Variable for Available Physical Measurements
   unsigned char SensorSubType;         // Variable for Sensor-Subtype
   float humidity, temperature, ambient pressure, CO2_Mean;
                                        // Variable for measuring values
   unsigned char Status;                // Variable for Statusbyte
   :
   :
   // initialise µP
   :
   :
   SensorType = fl_E2bus_Read_SensorType(); // read Sensortype from E2-Interface
   SensorSubType = fl_E2bus_Read_SensorSubType();
                                        // read Sensor Subtype from E2-Interface
   AvPhMes = fl_E2bus_Read_AvailablePhysicalMeasurements();
                                        // read available physical Measurements from

   E2while(1)                           // main loop
   {
      :
      :
      humidity = fl_E2bus_Read_RH();        // Read Measurement Value 1 (rel. Humidity [%RH])
      temperature = fl_E2bus_Read_Temp();   // Read Measurement Value 2 (Temperature [°C])
      air_pressure = fl_E2bus_Read_pres();  // Read Measurement Value 3 (Ambient pressure
                                            [mbar])
      CO2_Mean = fl_E2bus_Read_CO2_MEAN();  // Read Measurement Value 4 (CO2 MEAN [ppm])
      Status = fl_E2bus_Read_Status();      // Read Statusbyte from E2-Interface
      // analyse status and measured values
      :
      :
   }
}
```

9.3   Header Files

9.3.1   Header File for Functions

For implementing the E2 Interface module routines following header file (fl_E2bus.h) shall be imported into the master code main module:

```
/***********************************************************************/
/*      headerfile for "fl_E2bus.c" module                             */
/***********************************************************************/


#ifndef ___FL_E2BUS_INCLUDED
#define ___FL_E2BUS_INCLUDED

//      constant definition
//-----------------------------------------------------------------------
#define CB_TYPELO      0x11    // ControlByte for reading Sensortype Low-Byte
#define CB_TYPESUB     0x21    // ControlByte for reading Sensor-Subtype
#define CB_AVPHMES     0x31    // ControlByte for reading Available physical measurements
#define CB_TYPEHI      0x41    // ControlByte for reading Sensortype High-Byte
#define CB_STATUS      0x71    // ControlByte for reading Statusbyte
#define CB_MV1LO       0x81    // ControlByte for reading Measurement value 1 Low-Byte
#define CB_MV1HI       0x91    // ControlByte for reading Measurement value 1 High-Byte
#define CB_MV2LO       0xA1    // ControlByte for reading Measurement value 2 Low-Byte
#define CB_MV2HI       0xB1    // ControlByte for reading Measurement value 2 High-Byte
#define CB_MV3LO       0xC1    // ControlByte for reading Measurement value 3 Low-Byte
#define CB_MV3HI       0xD1    // ControlByte for reading Measurement value 3 High-Byte
#define CB_MV4LO       0xE1    // ControlByte for reading Measurement value 4 Low-Byte
#define CB_MV4HI       0xF1    // ControlByte for reading Measurement value 4 High-Byte
#define E2_DEVICE_ADR 0        // Address of E2-slave-Device

//      declaration of functions
//-----------------------------------------------------------------------
unsigned int fl_E2bus_Read_SensorType(void);        // read Sensortype from E2-Interface
unsigned char fl_E2bus_Read_SensorSubType(void);     // read Sensor Subtype from E2-Interface
unsigned char fl_E2bus_Read_AvailablePhysicalMeasurements(void);
                                // read available physical Measurements from E2-Interface
float fl_E2bus_Read_RH(void);          // Read Measurement Value 1 (relative Humidity [%RH])
float fl_E2bus_Read_Temp(void);        // Read Measurement Value 2 (Temperature [°C])
float fl_E2bus_Read_pres(void);        // Read Measurement Value 3 (Ambient pressure [mbar])
float fl_E2bus_Read_CO2_MEAN(void);  // Read Measurement Value 4 (CO2 MEAN [ppm])
unsigned char fl_E2bus_Read_Status(void);   // read Statusbyte from E2-Interface

#endif
```

9.3.2 Header File for Sub-Functions

```
/***********************************************************************/
/*    headerfile for "knl_E2bus.c" module                              */
/***********************************************************************/

#ifndef ___KNL_E2BUS_INCLUDED
#define ___KNL_E2BUS_INCLUDED

// constant definition
//--------------------------------------------------------------------------------
#define  RETRYS              3        // number of read attempts
#define DELAY_FACTOR         2        // delay factor for configuration of interface speed

// pin assignment
//--------------------------------------------------------------------------------
sbit E2_SCL = P0^2;                                  // Clock-Line
sbit E2_SDA = P0^3;                                  // Data-Line


// definition of structs
//--------------------------------------------------------------------------------
typedef struct st_E2_Return
{
   unsigned char DataByte;
   unsigned char Status;
}st_E2_Return;


// declaration of functions
//--------------------------------------------------------------------------------
st_E2_Return knl_E2bus_readByteFromslave( unsigned char ControlByte );
void knl_E2bus_start(void);
void knl_E2bus_stop(void);
void knl_E2bus_sendByte(unsigned char);
unsigned char knl_E2bus_readByte(void);
void knl_E2bus_delay(unsigned int value);
char knl_E2bus_check_ack(void);
void knl_E2bus_send_ack(void);
void knl_E2bus_send_nak(void);

void knl_E2bus_set_SDA(void);
void knl_E2bus_clear_SDA(void);
bit knl_E2bus_read_SDA(void);
void knl_E2bus_set_SCL(void);
void knl_E2bus_clear_SCL(void);

#endif
```

9.4 Software Functions of the E2 Interface Software Module

The following functions allow the compilation of a complete E2-Interface software module using the definitions in the header files. This code can be adapted easily for the available processor. For this it is only necessary to check and possibly to change the DELAY_FACTOR, the HW-Pins (E2_SCL, E2_SDA) and the designated functions.

9.4.1 Required Includes

```
// Includes
//----------------------------------------------------------------------
#include "f410.h"                    // SFR declarations µC-specific
#include "knl_E2bus.h"
#include "fl_E2bus.h"
```

9.4.2 Relative Humidity Request

```
float fl_E2bus_Read_RH(void)          // Read Measurement Value 1 (relative Humidity [%RH])
{  st_E2_Return xdata E2_Return;
   float RH;
   unsigned char RH_LB, RH_HB;

   RH = -1;

   E2_Return = knl_E2bus_readByteFromslave(CB_MV1LO|(E2_DEVICE_ADR<<1));
   RH_LB = E2_Return.DataByte;

   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromslave(CB_MV1HI|(E2_DEVICE_ADR<<1));
      RH_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  RH = (RH_LB + (float)(RH_HB)*256) / 100;
      }
   }

   return RH;
}
```

9.4.3 Temperature Request

```
float fl_E2bus_Read_Temp(void)        // Read Measurement Value 2 (Temperature [°C])
{  st_E2_Return xdata E2_Return;
   float Temp;
   unsigned char Temp_LB, Temp_HB;

   Temp = -300;

   E2_Return = knl_E2bus_readByteFromslave(CB_MV2LO|(E2_DEVICE_ADR<<1));
   Temp_LB = E2_Return.DataByte;

   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromslave(CB_MV2HI|(E2_DEVICE_ADR<<1));
      Temp_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  Temp = (Temp_LB + (float)(Temp_HB)*256) / 100 - 273.15;
      }
   }

   return Temp;
}
```

9.4.4 Ambient Pressure Request

```
float fl_E2bus_Read_pres(void)          // Read Measurement Value 3 (Ambient pressure [mbar])
{  st_E2_Return xdata E2_Return;
   float pres;
   unsigned char pres_LB, pres_HB;

   pres = -1;

   E2_Return = knl_E2bus_readByteFromslave(CB_MV3LO|(E2_DEVICE_ADR<<1));
   pres_LB = E2_Return.DataByte;

   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromslave(CB_MV3HI|(E2_DEVICE_ADR<<1));
      pres_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  pres = (pres_LB + (float)(pres_HB)*256) / 10;
      }
   }

   return pres;
}
```

9.4.5 CO$_2$ Request

```
float fl_E2bus_Read_CO2_MEAN(void)   // Read Measurement Value 4 (CO2 MEAN [ppm])
{  st_E2_Return xdata E2_Return;
   float CO2_MEAN;
   unsigned char CO2_LB, CO2_HB;

   CO2_MEAN = -1;

   E2_Return = knl_E2bus_readByteFromslave(CB_MV4LO|(E2_DEVICE_ADR<<1));
   CO2_LB = E2_Return.DataByte;

   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromslave(CB_MV4HI|(E2_DEVICE_ADR<<1));
      CO2_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  CO2_MEAN = CO2_LB + (float)(CO2_HB)*256;
      }
   }

   return CO2_MEAN;
}
```

9.4.6 Status Request

```
unsigned char fl_E2bus_Read_Status(void)     // read Statusbyte from E2-Interface
{  st_E2_Return xdata E2_Return;

   E2_Return = knl_E2bus_readByteFromslave(CB_STATUS|(E2_DEVICE_ADR<<1));
   if(E2_Return.Status == 1)
   {  E2_Return.DataByte = 0xFF;
   }

   return E2_Return.DataByte;
}
```

9.4.7 Sensor Type Request

```
unsigned int fl_E2bus_Read_SensorType(void) // read Sensortype from E2-Interface
{  st_E2_Return xdata E2_Return;
   unsigned int Type;
   unsigned char Type_LB, Type_HB;

   Type = 0xFFFF;

   E2_Return = knl_E2bus_readByteFromslave(CB_TYPELO|(E2_DEVICE_ADR<<1));
   Type_LB = E2_Return.DataByte;
   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromslave(CB_TYPEHI|(E2_DEVICE_ADR<<1));
      Type_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  Type = Type_LB + (unsigned int)(Type_HB)*256;
      }
   }

   return Type;
}
```

9.4.8 Sensor Subtype Request

```
unsigned char fl_E2bus_Read_SensorSubType(void)     // read Sensor Subtype from E2-Interface
{  st_E2_Return xdata E2_Return;

   E2_Return = knl_E2bus_readByteFromslave(CB_TYPESUB|(E2_DEVICE_ADR<<1));
   if(E2_Return.Status == 1)
   {  E2_Return.DataByte = 0xFF;
   }

   return E2_Return.DataByte;
}
```

9.4.9 Available Physical Quantities Request

```
unsigned char fl_E2bus_Read_AvailablePhysicalMeasurements(void)
// read available physical Measurements from E2-Interface
{  st_E2_Return xdata E2_Return;

   E2_Return = knl_E2bus_readByteFromslave(CB_AVPHMES|(E2_DEVICE_ADR<<1));
   if(E2_Return.Status == 1)
   {  E2_Return.DataByte = 0xFF;
   }

   return E2_Return.DataByte;
}
```

9.5 Sub-Functions

```c
// Includes
//-------------------------------------------------------------------------
#include "f410.h"                      // SFR declarations µC-specific
#include "knl_E2bus.h"


// Definitions
//-------------------------------------------------------------------------
#define  ACK        1
#define  NAK        0
#define  DELAY_FACTOR 2


st_E2_Return knl_E2bus_readByteFromslave( unsigned char ControlByte )
                                        // read byte from slave with controlbyte
{  unsigned char Checksum;
   unsigned char counter=0;
   st_E2_Return xdata E2_Return;

   E2_Return.Status = 1;

   while (E2_Return.Status && counter<RETRYS)
                                        // RETRYS...Number of read attempts
   {
      knl_E2bus_start();               // send E2 start condition
      knl_E2bus_sendByte( ControlByte ); // send 0xA1 (example for reading Temp_Low byte)

      if ( knl_E2bus_check_ack() == ACK ) // ACK received?
      {  E2_Return.DataByte = knl_E2bus_readByte();
                                        // read Temp_low (example for reading Temp_Low byte)
         knl_E2bus_send_ack();         // send ACK
         Checksum = knl_E2bus_readByte(); // read checksum
         knl_E2bus_send_nak();         // send NACK

         if ( ( ( ControlByte + E2_Return.DataByte ) % 0x100 ) == Checksum )
                                        // checksum OK?
            E2_Return.Status = 0;
      }
      knl_E2bus_stop();                // send E2 stop condition
      counter++;
   }
   return E2_Return;
}

void knl_E2bus_start(void)                          // send start condition to E2-Interface
{  knl_E2bus_set_SDA();
   knl_E2bus_set_SCL();
   knl_E2bus_delay(30);
   knl_E2bus_clear_SDA();
   knl_E2bus_delay(30);
}


void knl_E2bus_stop(void)                           // send stop condition to E2-Interface
{  knl_E2bus_clear_SCL();
   knl_E2bus_delay(20);
   knl_E2bus_clear_SDA();
   knl_E2bus_delay(20);
   knl_E2bus_set_SCL();
   knl_E2bus_delay(20);
   knl_E2bus_set_SDA();
}
```

```
void knl_E2bus_sendByte(unsigned char value)        // send byte to E2-Interface
{   unsigned char mask;

    for ( mask = 0x80; mask > 0; mask >>= 1)
    {   knl_E2bus_clear_SCL();
        knl_E2bus_delay(10);

        if ((value & mask) != 0)
        {   knl_E2bus_set_SDA();
        }
        else
        {   knl_E2bus_clear_SDA();
        }

        knl_E2bus_delay(20);
        knl_E2bus_set_SCL();
        knl_E2bus_delay(30);
        knl_E2bus_clear_SCL();
    }
    knl_E2bus_set_SDA();
}


unsigned char knl_E2bus_readByte(void)              // read Byte from E2-Interface
{   unsigned char data_in = 0x00;
    unsigned char mask = 0x80;

    for (mask=0x80;mask>0;mask >>=1)
    {   knl_E2bus_clear_SCL();
        knl_E2bus_delay(30);

        knl_E2bus_set_SCL();
        knl_E2bus_delay(15);
        if (knl_E2bus_read_SDA())
        {   data_in |= mask;
        }
        knl_E2bus_delay(15);
        knl_E2bus_clear_SCL();
    }
    return data_in;
}


char knl_E2bus_check_ack(void)                      // check for acknowledge
{   bit input;

    knl_E2bus_clear_SCL();
    knl_E2bus_delay(30);

    knl_E2bus_set_SCL();
    knl_E2bus_delay(15);
    input = knl_E2bus_read_SDA();
    knl_E2bus_delay(15);
    if(input == 1)                                  // SDA = LOW ==> ACK, SDA = HIGH ==> NAK
        return NAK;
    else
        return ACK;
}


void knl_E2bus_send_ack(void)                       // send acknowledge
{   knl_E2bus_clear_SCL();
    knl_E2bus_delay(15);

    knl_E2bus_clear_SDA();
    knl_E2bus_delay(15);
    knl_E2bus_set_SCL();
    knl_E2bus_delay(28);
    knl_E2bus_clear_SCL();
    knl_E2bus_delay(2);
    knl_E2bus_set_SDA();
}
```

```
void knl_E2bus_send_nak(void)                          // send NOT-acknowledge
{  knl_E2bus_clear_SCL();
   knl_E2bus_delay(15);

   knl_E2bus_set_SDA();
   knl_E2bus_delay(15);
   knl_E2bus_set_SCL();
   knl_E2bus_delay(30);
   knl_E2bus_set_SCL();
}


void knl_E2bus_delay(unsigned int count)               // knl_E2bus_delay function
{  volatile unsigned int count2;
   count2 = count;
   count2 = count2 * DELAY_FACTOR;
   // adapt "DELAY_FACTOR" to match the target frequency for E2-Interface communication
   while (--count2 != 0);
}


// adapt this code for your target processor !!! Value = 1 ==> Physical Signal is High, Value
= 0 ==> Physical Signal is Low
void knl_E2bus_set_SDA(void)
{  E2_SDA = 1;                          // set port-pin (SDA)
}

void knl_E2bus_clear_SDA(void)
{  E2_SDA = 0;                          // clear port-pin (SDA)
}

bit knl_E2bus_read_SDA(void)
{  return E2_SDA;                       // read SDA-pin status
}

void knl_E2bus_set_SCL(void)
{  E2_SCL = 1;                          // set port-pin (SCL)
   while(E2_SCL == 0);                  // wait for released SCL line in case of clock-stretching
                                           by slave
}

void knl_E2bus_clear_SCL(void)
{  E2_SCL = 0;                          // clear port-pin (SCL)
}
```

9.6 Return Values

The routines above use following format for the return values:

Relative Humidity (float):

Return Value	Meaning
0.0...100.0	0.0…100.00% relative humidity
-1	Error code

Temperature (float):

Return Value	Meaning
-40.0…+85.0	Temperature in °C according to measurement range
-300	Error code

Ambient Pressure (float):

Return Value	Meaning
300.0…1100.0	300…1100 mbar
-1	Error code

CO_2 Concentration(float):

Return Value	Meaning
0…range	CO_2 concentration according to the scaling of the EE894
-1	Error code

Status (unsigned char):
> The definition of the individual bits in the status bytes can be found in [1].

9.7 Bus Transmission Speed

The bus speed is defined by the clock frequency of the master processor and the DELAY_FACTOR constant (see function: delay). For the maximum bus speed which can be achieved by a specific EE894 please see chapter 2.2.

Please note:
The time delay is realised with a simple waiting loop. The optimisation level of the compiler should be selected so as to ensure that this loop is maintained and not „optimised out"!

10 Literature References

[1]	E2 Interface – Specification

Contact information

E+E ELEKTRONIK GES.M.B.H.
Langwiesen 7
4209 Engerwitzdorf
Austria

Tel.: +43 (7235) 605-0
Fax: +43 (7235) 605-8
E-Mail: info@epluse.com
Homepage: www.epluse.com

For your local contact please visit the homepage.