APPLICATION NOTE

AN0105

Application Note

Utilising the E2 Interface

Rev. 1.0 05/2015

Relevant for:

This application note applies to EE03, EE08, EE07, EE871 and EE893

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 and the status of an E+E measuring device are to be read periodically via E2 Interface. It provides a brief description of the hardware, explains the principle of the data transmission 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-range 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.

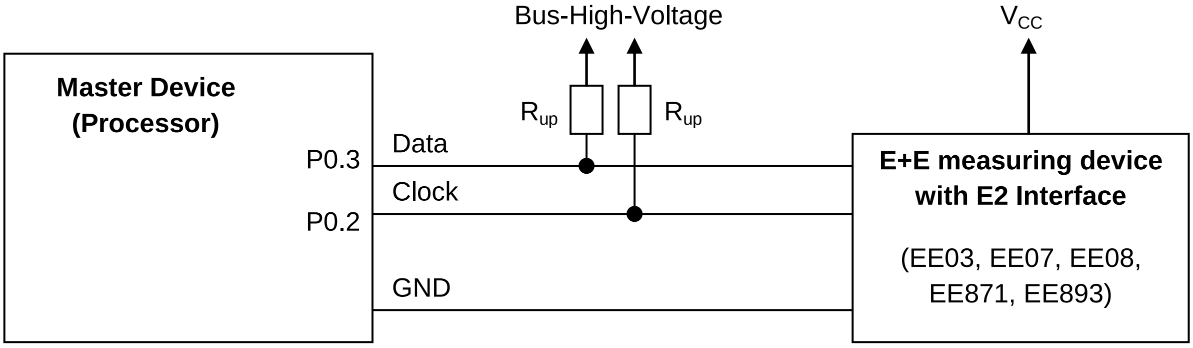

Figure 1: Hardware Master and Slave setup

Note: Note the compatibility of the voltage level between the E2 Interface levels and the Master processor.

2 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].

2.1 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].

2.1.1 Temperature

The data transmission method is explained using a multi-stage temperature request. For transmitting a 16 bit temperature (T) value the command ”Read Byte from Slave“ must be executed twice.
The T measured value corresponds to the “measuring value 2” of the module (ref. [2]).
For consistent data it is necessary to request first the low-byte of a measured value (ref. [1]).

Steps for T value request:

Step 1
Perform ”Read Byte from Slave“ command for reading the temperature Low-Byte:

- Apply Start condition and control byte (0xA1) to the bus
- Read in and check ACK/NACK of the Slave
- Read in data byte temperature Low-Byte (Temp_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 temperature High-Byte:

Step 2

- Apply Start condition and control byte (0xB1) to the bus
- Read in and check ACK/NACK of the Slave
- Read in data byte temperature High-Byte (Temp_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 T value.
It combines the High- and Low bytes to form a 16-bit value and divides this by 100. For T in °C one shall substract an offset of 273.15.

$$\text{Temperature}[°C] = (\text{Temp\_high} * 256 + \text{Temp\_low}) / 100 - 273.15$$

Figure 1 provides the flow chart of this sequence.

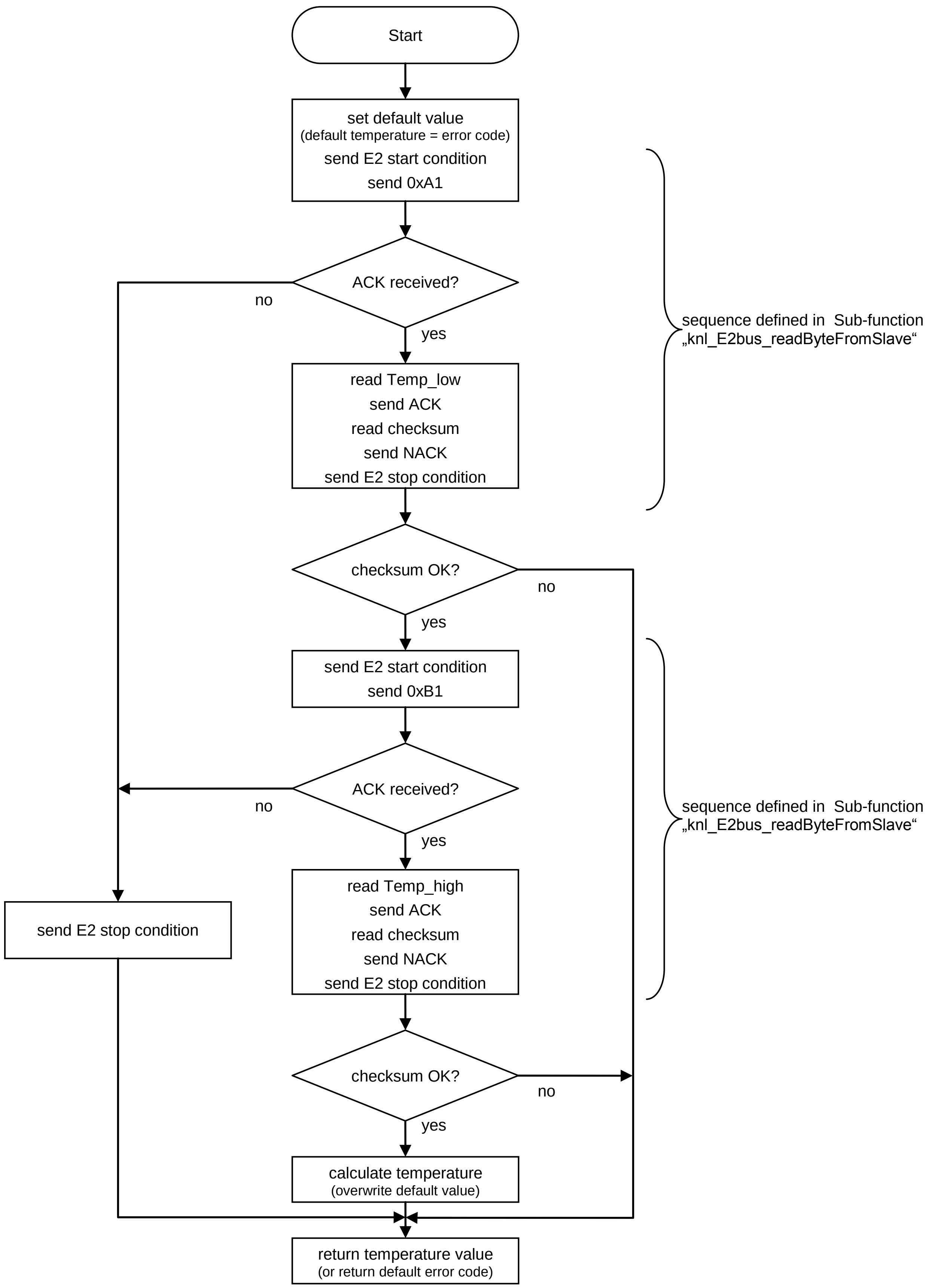

Figure 1: Flow chart of a T value request (= simplified flow of function “fl_E2bus_Read_Temp”)

2.1.2 Relative Humidity

The measured humidity value can be read similar to temperature as described above. The control bytes are 0x81 for the Low Byte and 0x91 for the High Byte of the measured humidity value. The actual relative humidity value is calculated as follows:

rel. humidity[%RH] = (rh_high * 256 + rh_low) / 100

2.1.3 CO_2

The measured CO_2 value can be read similar to temperature as described above. The corresponding control bytes for raw CO2 values are 0xC1 for Low Byte and 0xD1 for High Byte and for mean (corrected) CO2 values are 0xE1 for Low Byte and 0xF1 for High Byte of the measured CO_2 value. The actual CO_2 concentration is calculated as follows:

CO_2 [ppm] = (CO2_high * 256 + CO2_low)

2.2 Status Request

To guarantee the validity of measured values following the measured value requests, one must 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 1)
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 air velocity 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 a sensor failure.

Besides the information on the validity of the last measurement, the status request has also another purpose. After a status request from the Master, a new measurement is started at the Slave. During the measuring time, the Slave cannot process any enquiries via the E2 Interface. For details please view the product specific documents “Additional specification for E2 interface” (ref. [2] [6]).
Therefore, one shall read in the required measured values first, and then execute a status request. By doing this the validity of the last measured values can be evaluated, and a new measurement is started at the same time. After waiting for the module-specific measuring time, the new values can be requested again.

3 Software Examples for 8051 Processors

3.1 General

In this Application Note, the functions the E2 Interface executes are grouped together in separate software modules. This achieves 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 temperature value, the relative humidity, the CO_2-concentration and the status byte.
The header files specify the required definitions of the variables and the function prototypes to be able to create a simple software module for the E2 Interface.

3.2 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.

3.2.1 Example of Codes

```
:
#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, CO2_Raw, 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.v Humidity [%RH])
      temperature = fl_E2bus_Read_Temp();    // Read Measurement Value 2 (Temperature [°C])
      CO2_Raw = fl_E2bus_Read_CO2_RAW();     // Read Measurement Value 3 (CO2 RAW [ppm])
      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
      :
      :
   }
}
```

3.3 Header Files

3.3.1 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 (relativ Humidity [%RH])
float fl_E2bus_Read_Temp(void);      // Read Measurement Value 2 (Temperature [°C])
float fl_E2bus_Read_CO2_RAW(void);   // Read Measurement Value 3 (CO2 RAW [ppm])
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
```

3.3.2 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( 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
```

3.4 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 just the DELAY_FACTOR, the HW-Pins (E2_SCL, E2_SDA) and the designated functions are to checked and, if necessary, changed.

3.4.1 Required includes

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

3.4.2 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;
}
```

3.4.3 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;
}
```

3.4.4 CO2 Request (raw values)

```
float fl_E2bus_Read_CO2_RAW(void)     // Read Measurement Value 3 (CO2 RAW [ppm])
{  st_E2_Return xdata E2_Return;
   float CO2_RAW;
   unsigned char CO2_LB, CO2_HB;

   CO2_RAW = -1;

   E2_Return = knl_E2bus_readByteFromSlave(CB_MV3LO|(E2_DEVICE_ADR<<1));
   CO2_LB = E2_Return.DataByte;

   if(E2_Return.Status == 0)
   {  E2_Return = knl_E2bus_readByteFromSlave(CB_MV3HI|(E2_DEVICE_ADR<<1));
      CO2_HB = E2_Return.DataByte;

      if(E2_Return.Status == 0)
      {  CO2_RAW = CO2_LB + (float)(CO2_HB)*256;
      }
   }

   return CO2_RAW;
}
```

3.4.5 CO2 Request (mean values)

```
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;
}
```

3.4.6 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;
}
```

3.4.7 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;
}
```

3.4.8 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;
}
```

3.4.9 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;
}
```

3.5 Sub-Functions

```
// 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 frequence for E2-Interface communiation
   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)
}

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

3.6 Return Values

The routines above use following format for the return values:

Humidity (float):

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

Temperature (float):

Return Value	Meaning
> -273.15	temperature in °C (according to measurement range)
-300	Error code

CO_2 (float):

Return Value	Meaning
>=0	CO_2-concentration (according on the scaling of the slave device)
-1	Error code

Status (unsigned char):

The meaning of the individual bits in the status bytes is defined for all modules with E2 Interface in [1].

3.7 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 E+E device (Slave) please see [2] [6].

Important:
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“!

4 Literature References:

[1]	E2 Interface – Specification
[2]	EE03 Additional Specification E2 Interface
[3]	EE07 Additional Specification E2 Interface
[4]	EE08 Additional Specification E2 Interface
[5]	EE871 Additional Specification E2 Interface
[6]	EE893 Additional Specification E2 Interface

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.

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. E+E Elektronik assumes no responsibility for errors and omissions, and shall not accept responsibility for any consequences resulting from the use of information included herein. Additionally, E+E Elektronik assumes no responsibility for the functioning of features or parameters not described. E+E Elektronik reserves the right to make changes without further notice. E+E Elektronik makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does E+E Elektronik assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including consequential or incidental damages without limitation. E+E Elektronik products are not designed, intended, or authorised for use in applications intended to support or sustain life, or for any other application in which the failure of the E+E Elektronik product could create a situation where personal injury or death may occur. Should the buyer purchase or use E+E Elektronik products for any such unintended or unauthorised application, the buyer shall indemnify and hold E+E Elektronik harmless against all claims and damages.