APPLICATION NOTE

ABSTRACT
This application note demonstrates how to write I^{2}C-bus driver software for the LPC9xx microcontroller family from Philips Semiconductors.

In addition to the driver software, a small demo application program is given. All together this note offers users a quick start in writing a complete LPC9xx I^{2}C system application.

AN10155
Philips LPC9xx microcontroller in I^{2}C applications

Author: Paul Seerden

2002 Jun 21

Philips Semiconductors

INTRODUCTION

The I²C bus consists of two wires carrying information between the devices connected to the bus. Each device has its own address and can act as a master or as a slave during a data transfer. A master is the device that initiates the data transfer and generates the clock signals needed for the transfer. At that time any addressed device is considered a slave. The I²C bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. However, the example software given in this application note only supports (single) master transfers.
The I²C interface on the LPC9xx is identical to the standard byte - style I²C interface found on devices such as the 8xC552, except for the bit rate selection. The I²C interface of the LPC9xx conforms to the 400 kHz I^{2}C specification.

The I^{2}C-bus format

An I²C transfer is initiated with the generation of a start condition. This condition will set the bus busy. After that, a message is transferred that consists of an address and a number of data bytes. This I²C message may be followed either by a stop condition or a repeated start condition. A stop condition will release the bus master-ship. A repeated start offers the possibility to send / receive more than one message to / from the same or different devices, while retaining bus master-ship. Stop and (repeated) start conditions can only be generated in master mode.
Data and addresses are transferred in eight bit bytes, starting with the most significant bit. During the 9th clock pulse, following the data byte, the receiver must send an Acknowledge bit to the transmitter. The slave may stretch clock pulses (for timing causes). A 7-bits slave address and a R/W direction bit always follow a start condition.

SOFTWARE

The application's (master) view on the I²C bus is quite simple: the application can send a ***message*** to an I²C device (slave). Also, the application must be able to exchange a ***group of messages***, optionally

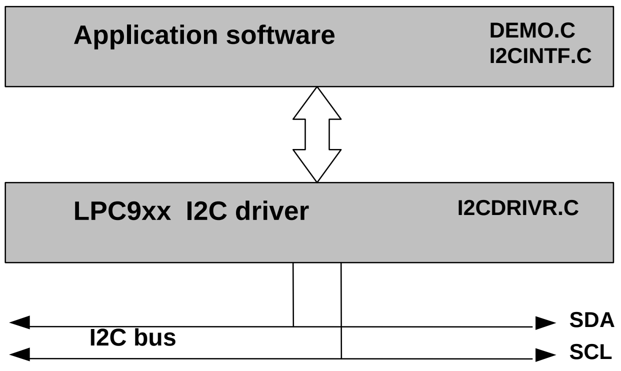

Figure 1 *Software structure and modules*

addressed to different devices, without losing bus master-ship. This is called a ***Transfer***. So, one Transfer consists of one or more messages. A transfer always begins with a Start condition and is ended by a Stop condition.
If a transfer contains more than one message, then each message is separated by a “repeated Start” condition and only the last message is followed by a Stop condition.

General format and explanation of an I²C transfer:

S	SLV_W	**A**	SUB	**A**	**S**	SLV_R	**A**	*D1*	**A**	*D2*	**A**	……….	**A**	*Dn*	**N**	**P**

S	: (repeated) Start condition.	SLV_W	: Slave address and Write bit.
A	: Acknowledge on last byte.	SLV_R	: Slave address and Read bit.
N	: No Acknowledge on last byte.	SUB	: Sub-address.
P	: Stop condition.	*D1 ... Dn*	: Block of data bytes.

Inputs (application's view) to the I^{2}C driver are:

⇒ The number of messages to exchange (transfer).

⇒ The slave address of the I²C device for each message.

⇒ The data direction (read/write) for each message.

⇒ The number of bytes in each message.

⇒ In case of a write message: the data bytes to be written to the slave.

Outputs (application's view) from the I^{2}Cdriver are:

⇒ Status information (success or error code).

⇒ Number of messages actually transferred (not the requested number of messages in case of an error).

⇒ For each read message: The data bytes read from the slave.

LPC9XX DRIVER DESCRIPTION

The I²C driver module (I2CDRIVR.C) contains two 'callable' interface functions:

I2C_Init: This function directly programs the I²C interface hardware and must be called only once after 'reset', but before any transfer function is executed.

The hardware I²C registers of the LPC9xx are programmed. Port pins P1.2 and P1.3, which correspond to the I²C functions SCL and SDA respectively, are set to the open drain mode. In our example the bit rate is programmed to 100 Kbit/s at an internal oscillator frequency of 7.373 MHz. To adapt this, change the I2SCLH/L values.

I2C_Transfer: This function is used to actually perform a transfer. If a transfer is started, this function returns immediately (completely interrupt driven). All parameters affected by an I²C master transfer are grouped within two structures. The user fills these structures and then calls the function to perform a transfer. The two data structures are listed below.

```
typedef struct
{
    BYTE            nrMessages;
    I2C_MESSAGE     **p_message;
} I2C_TRANSFER;
```

The first structure contains the number of messages and a pointer to an array of pointers to message blocks (second structure), as the two common parameters for one I²C transfer. The driver keeps a local copy of these parameters and leaves the contents unchanged. The message blocks are defined in a second data structure:

```
typedef struct
{
    BYTE  address;    /* The I2C slave device address */
    BYTE  nrBytes;    /* nr of bytes to read or write */
    BYTE  *buf;       /* pointer to data array */
} I2C_MESSAGE;
```

The LSB bit of the (slave) address parameter determines the direction of the message (write = 0 and read = 1). The array ***buf*** must contain data supplied by the application in case of a write message. The user should notice that checking, to ensure that the buffer pointed to by ***buf*** is at least nrBytes in length, cannot be done by the driver. In case of a read message, the driver fills the array. It's the user's responsibility to ensure that the buffer, pointed to by ***buf*,** is large enough to receive ***nrBytes*** bytes.

After completing the transmission or reception **of each byte** (address or data), the SI flag in the I2CON register is set. An interrupt is generated and the interrupt service handler of the I2C driver will be called. At that time register I2STAT holds the status code.

At the end of **a complete** transfer, together with the generation of a STOP condition, the I2C driver calls a function ***(I2C_Ready)*** inside the application program, passing the transfer status and the number of messages successfully transferred. This "**call back**" function has a reserved name and must be provided by the user (application). The transfer status (okay, error, time-out, etc.) can be checked by the application. An example of how this can be done is shown in the module I2CINTFC.C.

DEMO (APPLICATION) PROGRAM

Both files DEMO.C and I2CINTFC.C use the I2C driver module to implement a simple application. They are intended as examples to show how to use the driver routines. As an example the demo application drives a PCF8574A I/O expander with connections to 8 LED's (see figure 2). The demo program runs the LED's turn by turn every second.

The module I2CINTFC.C gives an example of how to implement a few basic transfer functions. These functions allow the user to communicate with most of the available I²C devices and serve as a *layer* between the application and the driver software. This *layered approach* allows support for new devices (micro-controllers) without re-writing the high-level (device-independent) code.

Furthermore, the module I2CINTFC.C contains the functions *StartTransfer*, in which the actual call to the driver program is done, and the function *I2C_Ready*, which is called by the driver after the completion of a transfer. The flag **drvStatus** is used to test/check the state of a transfer. In the *StartTransfer* function a software time-out loop is programmed. If a transfer has failed (error or time-out) the *StartTransfer* function prints an error message (using the UART of the LPC9xx) and it does a retry of the transfer.

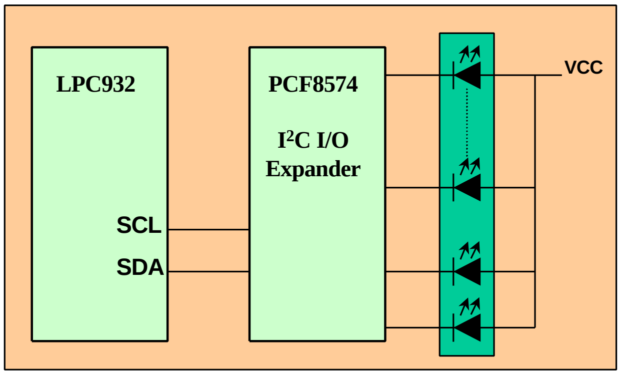

Figure 2 *Demo application*

DEMO.C LISTING

```
/*********************************************************
* LPC932 I2C demoprogram
* 1. Use T0 to generate a 1 second timer tick.
* 2. Flash main Led every second
* 3. Run leds connected to PCF8574 every second
*    using LPC932 interrupt driven I2C driver.
*********************************************************/

#include "i2cexprt.h"

#define PCF8574A_WR 0x70       /* i2c address         */

rom char hello[] = "LPC2 - I2C test March 2002\n";

static _bit second = 0;    /* one second passed flag  */
static WORD count;
static BYTE iopBuf[1];
static I2C_MESSAGE iopMsg;

static void UART_Init(void)
{
    P1M1 &= 0xFE;
    P1M2 |= 0x01;
    BRGR1  = 0x01;
    BRGR0  = 0x70;
    BRGCON = 3;
    SCON = 0x52;       /* mode 1, receiver enable     */
}

static void ua_outchar(char c)
{
    while (!TI)  ;
    SBUF = c;
    TI = 0;
}

void PrintString(rom char *s)
{
    while (*s)
    {
        if (*s == '\n')
            ua_outchar('\r');
        ua_outchar(*s);
       s++;
    }
}
```

```
// LPC2 - OSC = 7,373 MHz, PRE = 2
```

```
// reload_value = -OSC/(PRE*500Hz) = -7373 = 2 msec
```

```
interrupt(1) void T0_Interrupt(void)
{
    TR0 = 0;             /* Stop timer 0              */
    TL0 = 0x33;          /* set timer 0 reload value  */
    TH0 = 0xe3;
    TR0 = 1;             /* Restart timer 0           */
    if (++count > 500)   /* 500 * 2 msec = 1 sec      */
    {
        count  = 0;
        second = 1;      /* one second passed         */
    }
}

static void T0_Init(void)
{
    count = 0;
    TMOD = 1;          /* Timer 0 = Mode 1, 16 Bit    */
    TL0 = 0x33;        /* set timer 0 reload value    */
    TH0 = 0xe3;
    ET0 = 1;           /* enable timer 0 interrupt    */
    TR0 = 1;           /* start timer 0               */
}

static void RunLeds(void)
{
  static BYTE ioport;

    switch (ioport)       /* run I2C IOport leds      */
    {
      case 0x7f: ioport = 0xfe;   break;
      case 0xbf: ioport = 0x7f;   break;
      case 0xdf: ioport = 0xbf;   break;
      case 0xef: ioport = 0xdf;   break;
      case 0xf7: ioport = 0xef;   break;
      case 0xfb: ioport = 0xf7;   break;
      case 0xfd: ioport = 0xfb;   break;
      case 0xfe: ioport = 0xfd;   break;
      default:   ioport = 0xfe;   break;
    }
    iopBuf[0] = ioport;
    I2C_Write(&iopMsg);
}

void main(void)
{
    TRIM = 0x3C;       /* clock out at P3.0           */

    T0_Init();         /* initialize Timer 0          */
    UART_Init();       /* initialize UART             */
    I2C_Init();        /* initialize I2C bus          */
    EA = 1;            /* General interrupt enable    */

    iopMsg.address = PCF8574A_WR;
    iopMsg.buf     = iopBuf;
    iopMsg.nrBytes = 1;
    iopBuf[0] = 0xff;
    I2C_Write(&iopMsg);

    PrintString(hello);

    while (1)
    {
        if (second)
        {
            second = 0;
            Led = !Led;        /* toggle the LED       */
            RunLeds();
        }
    }
}
```

I2CINTF.C LISTING

```
#include "i2cexprt.h"

extern void PrintString(rom char *s);

rom char retryexp[] = "retry counter expired\n";
rom char bufempty[] = "buffer empty\n";
rom char nackdata[] = "no ack on data\n";
rom char nackaddr[] = "no ack on address\n";
rom char timedout[] = "time-out\n";
rom char unknowst[] = "unknown status\n";

static I2C_MESSAGE   *p_iicMsg[2];
static I2C_TRANSFER  iicTfr;

static BYTE drvStatus;
```

```
void using(1) I2C_Ready(BYTE status, BYTE nr)
{
    drvStatus = status;
}

static void StartTransfer(void)
{
  WORD timeOut;
  BYTE retries = 0;

    do
    {
        drvStatus = I2C_BUSY;
        I2C_Transfer(&iicTfr);

        timeOut = 0;
        while (drvStatus == I2C_BUSY)
        {
            if (++timeOut > 40000)
                drvStatus = I2C_TIME_OUT;
        }

        if (retries == 6)
        {
            PrintString(retryexp);
            return;
        }
        else
            retries++;

        switch (drvStatus)
        {
          case I2C_OK: break;
          case I2C_NO_DATA:
              PrintString(bufempty);  break;
          case I2C_NACK_ON_DATA:
              PrintString(nackdata);  break;
          case I2C_NACK_ON_ADDRESS:
              PrintString(nackaddr);  break;
          case I2C_TIME_OUT:
              PrintString(timedout);  break;
          default: PrintString(unknowst);  break;
        }
    } while (drvStatus != I2C_OK);
}

void I2C_Write(I2C_MESSAGE *msg)
{
    iicTfr.nrMessages = 1;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg;

    StartTransfer();
}



void I2C_WriteRepWrite(I2C_MESSAGE *m1,I2C_MESSAGE *m2)
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = m1;
    p_iicMsg[1] = m2;

    StartTransfer();
}

void I2C_WriteRepRead(I2C_MESSAGE *m1, I2C_MESSAGE *m2)
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = m1;
    p_iicMsg[1] = m2;
    StartTransfer();
}

void I2C_Read(I2C_MESSAGE *msg)
{
    iicTfr.nrMessages = 1;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = msg;

    StartTransfer();
}

void I2C_ReadRepRead(I2C_MESSAGE *m1, I2C_MESSAGE *m2)
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = m1;
    p_iicMsg[1] = m2;

    StartTransfer();
}

void I2C_ReadRepWrite(I2C_MESSAGE *m1, I2C_MESSAGE *m2)
{
    iicTfr.nrMessages = 2;
    iicTfr.p_message  = p_iicMsg;
    p_iicMsg[0] = m1;
    p_iicMsg[1] = m2;

    StartTransfer();
}
```

I2CDRIVR.C LISTING

```
#include "i2cexprt.h"

/*    Immediate data to write into I2CON              */
/*    CRSEL = 0 -> CLK determined by I2SCLH / L       */

#define GENERATE_STOP     0x54  /* STO=1, STA=0, SI=0 */
#define RELEASE_BUS_ACK   0x44  /* STO=STA=SI=0, AA=1 */
#define RELEASE_BUS_NOACK 0x40  /* STO=STA=SI=AA=0    */
#define RELEASE_BUS_STA   0x64  /* (rep)START, STA=1  */


static I2C_TRANSFER *tfr;  /* Ptr to active trf block */
static I2C_MESSAGE  *msg;  /* ptr to active msg block */

static BYTE msgCount;      /* Nr of messages to sent  */
static BYTE dataCount;     /* bytes send/received     */


interrupt(6) using(1) void I2C_Interrupt(void)
{
    switch(I2STAT)
    {
      case 0x00:         /* Bus Error has occured     */
        I2CON = GENERATE_STOP;
        break;
      case 0x08:
      case 0x10:
        I2DAT = msg->address;
        I2CON = RELEASE_BUS_ACK;
        break;
      case 0x18:
      case 0x28:
        if (dataCount < msg->nrBytes)
        {
            I2DAT = msg->buf[dataCount++];
            I2CON = RELEASE_BUS_ACK;
        }
        else
        {
            if (msgCount < tfr->nrMessages)
            {
                dataCount = 0;
                msg = tfr->p_message[msgCount++];
                I2CON = RELEASE_BUS_STA;
            }
            else
            {
                I2CON = GENERATE_STOP;
                I2C_Ready(I2C_OK, msgCount);
            }
        }
        break;
      case 0x20:
      case 0x48:
        I2C_Ready(I2C_NACK_ON_ADDRESS, msgCount);
        I2CON = GENERATE_STOP;
        break;
      case 0x30:
        I2C_Ready(I2C_NACK_ON_DATA, msgCount);
        I2CON = GENERATE_STOP;
        break;
      case 0x38:
        I2CON = RELEASE_BUS_STA;
        break;
      case 0x40:
        if (msg->nrBytes == 1)
            I2CON = RELEASE_BUS_NOACK;
        else
            I2CON = RELEASE_BUS_ACK;
        break;
      case 0x50:
        msg->buf[dataCount++] = I2DAT;
        if (dataCount + 1 == msg->nrBytes)
            I2CON = RELEASE_BUS_NOACK;
        else
            I2CON = RELEASE_BUS_ACK;
        break;
      case 0x58:
        msg->buf[dataCount] = I2DAT;
        if (msgCount < tfr->nrMessages)
        {
            dataCount = 0;
            msg = tfr->p_message[msgCount++];
            I2CON = RELEASE_BUS_STA;
        }
        else
        {
            I2CON = GENERATE_STOP;
            I2C_Ready(I2C_OK, msgCount);
        }
        break;
      default: break;
    }
}

void I2C_Init(void)
/*****************/
{
// Fpclk = 7.373 Mhz internal oscillator
// I2c speed = Fpclk / (2*(I2SCLH+I2SCLL)

    P1M1 |= 0x0C;   /* P1.2 and P1.3 to open drain    */
    P1M2 |= 0x0C;
    I2ADR = 0x26;      /* default slave address       */
    I2SCLH = 19;       /* speed ~100KHz, 50% duty     */
    I2SCLL = 19;
    I2CON = RELEASE_BUS_ACK;  /* enable I2C hardware  */
    EI2C  = 1;                /* enable I2C interrupt */
}

void I2C_Transfer(I2C_TRANSFER *p)
```

```
{
    tfr = p;
    msgCount  = 0;
    dataCount = 0;
    msg = tfr->p_message[msgCount++];
    I2CON = RELEASE_BUS_STA;
}
```

I2CEXPRT.H LISTING

```
_sfrbyte TRIM    _at(0x96);

_sfrbyte P1M1    _at(0x91);
_sfrbyte P1M2    _at(0x92);
_sfrbyte P3M1    _at(0xb1);
_sfrbyte P3M2    _at(0xb2);

_sfrbyte BRGCON  _at(0xbd);
_sfrbyte BRGR0   _at(0xbe);
_sfrbyte BRGR1   _at(0xbf);

_sfrbyte I2ADR   _at(0xdb);
_sfrbyte I2CON   _at(0xd8);
_sfrbyte I2DAT   _at(0xda);
_sfrbyte I2SCLH  _at(0xdd);
_sfrbyte I2SCLL  _at(0xdc);
_sfrbyte I2STAT  _at(0xd9);
_sfrbyte IEN1    _at(0xe8);

_sfrbit  EI2C    _atbit(IEN1,0);

#define Led     P1_7       /* microcore board led     */


/*******************************************************/
/*  E X P O R T E D   D A T A   S T R U C T U R E S  */
/*******************************************************/

typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef unsigned long    LONG;


typedef struct
{
    BYTE   address;
    BYTE   nrBytes;
    BYTE   *buf;
} I2C_MESSAGE;


typedef struct
{
    BYTE        nrMessages;
    I2C_MESSAGE  **p_message;
} I2C_TRANSFER;


/*******************************************************/
/* E X P O R T E D   D A T A  D E C L A R A T I O N S */
/*******************************************************/

/**** Status Errors ****/

#define I2C_OK      0    /* transfer ended No Errors   */
#define I2C_BUSY    1    /* transfer busy              */
#define I2C_ERR     2    /* err: general error         */
#define I2C_NO_DATA      3   /* err: No data in block */
#define I2C_NACK_ON_DATA     4  /* err: Nack on data  */
#define I2C_NACK_ON_ADDRESS  5  /* err: Nack on addr  */
#define I2C_NOT_PRESENT      6  /* Device not present */
#define I2C_ARBITRATION_LOST 7  /* Arbitration lost   */
#define I2C_TIME_OUT         8  /* Time out occurred  */
#define I2C_SLAVE_ERROR      9  /* slave mode error   */
#define I2C_INIT_ERROR      10  /* Init (not done)    */


/*******************************************************/
/*      F U N C T I O N   P R O T O T Y P E S        */
/*******************************************************/

extern void I2C_Transfer(I2C_TRANSFER *p);
extern void I2C_Init(void);
extern void using(1) I2C_Ready(BYTE status, BYTE nr);

extern void I2C_Write(I2C_MESSAGE *msg);
extern void I2C_WriteRepWrite(I2C_MESSAGE *msg1,
I2C_MESSAGE *msg2);
extern void I2C_WriteRepRead(I2C_MESSAGE *msg1,
I2C_MESSAGE *msg2);
extern void I2C_Read(I2C_MESSAGE *msg);
extern void I2C_ReadRepRead(I2C_MESSAGE *msg1,
I2C_MESSAGE *msg2);
extern void I2C_ReadRepWrite(I2C_MESSAGE *msg1,
I2C_MESSAGE *msg2);
```

REFERENCES

For further details please refer to the following publications:

- Datasheets:
  www.semiconductors.philips.com
- Example Programs:
  http://www.keil.com/download/c51.asp
- Brochures / leaflets:
  “The I2C-bus and how to use it”

Philips LPC9xx microcontroller in I^{2}C applications

AN10155

Contact information

For additional information please visit http://www.semiconductors.philips.com.

For sales offices addresses send e-mail to: sales.addresses@www.semiconductors.philips.com.

© Koninklijke Philips Electronics N.V. 2002
All rights reserved. Printed in U.S.A

Date of release: 06-02
Document order number: 9397 750 10014

Let's make things better.

Philips
Semiconductors

PHILIPS